



Hochschule
Zittau/Görlitz
UNIVERSITY OF APPLIED SCIENCES

Web Engineering 3

Script

Inhaltsverzeichnis

1.	Ablauf & Planung	5
2.	Anforderungen	5
2.1.	Projekt	5
2.2.	Beleg	5
2.3.	Präsentation bzw. Verteidigung	5
3.	Übersicht	6
3.1.	Struktur einer Full Stack Web Application	6
3.2.	Spring Übersicht	7
3.2.1.	Geschichte von Spring	7
3.2.2.	Springboot	7
3.3.	Svelte Übersicht	9
3.4.	Tailwind CSS Übersicht	10
4.	Grundbegriffe	11
4.1.	Client Side	11
4.2.	Server Side	11
4.3.	API	11
4.4.	REST	11
4.4.1.	Unabhängigkeit	11
4.4.2.	Stateless	11
4.4.3.	Kommunikation zwischen Client und Server	11
4.5.	HTTP	12
4.5.1.	Request	12
4.5.1.1.	Header	12
4.5.1.2.	Pfad	12
4.5.2.	Response	12
4.5.3.	Methoden	12
4.5.3.1.	GET	12
4.5.3.2.	POST	12
4.5.3.3.	PUT	13
4.5.3.4.	DELETE	13
4.5.3.5.	Idempotente Methoden	13
4.5.4.	Status Codes	14
4.6.	DTO	15
4.7.	MVC	15
4.8.	CRUD	15
4.9.	ORM	15
5.	Technologien	16
5.1.	Frontend	16
5.2.	Backend	16
5.3.	Database	16
5.4.	Testing	16
5.5.	DevOps	16
6.	User Stories	17
6.1.	User Stories für das finale Projekt	17
7.	Backend	18
7.1.	Spring	18
7.1.1.	Komponenten einer Spring Anwendung	18

7.1.1.1.	Controller	18
7.1.1.1.1.	Mappings	18
7.1.1.1.2.	URI Patterns	18
7.1.1.1.3.	Error Handling	19
7.1.1.2.	Service	21
7.1.1.2.1.	Transactional Methoden	21
7.1.1.3.	Entity	23
7.1.1.3.1.	@Entity Annotation	23
7.1.1.3.2.	@Table Annotation	23
7.1.1.3.3.	@Id Annotation	23
7.1.1.3.4.	@Column Annotation	24
7.1.1.3.5.	Referenzen auf andere Tables	24
7.1.1.3.5.1.	@OneToMany und @ManyToOne	24
7.1.1.3.5.2.	@OneToOne	25
7.1.1.3.5.3.	@ManyToMany	26
7.1.1.4.	Repository	28
7.1.1.5.	DTO	29
7.1.1.6.	Mapper	30
7.1.1.7.	Application Konfiguration	31
7.1.2.	Spring Data	32
7.1.2.1.	Spring Data JDBC	32
7.1.2.2.	Spring Data JPA	33
7.1.2.3.	Spring Data R2DBC	34
7.1.2.4.	Spring Data REST	35
7.1.3.	IoC Container	36
7.1.4.	Dependency Injection	37
7.1.4.1.	Constructor Injection	37
7.1.4.2.	Setter Injection	37
7.1.5.	Method Injection	38
7.1.6.	Beans	39
7.1.6.1.	Spring Inversion of Control (IoC) Container	39
7.1.6.2.	Annotationen für Beans [1]	40
7.1.6.3.	Scoping	40
7.1.6.3.1.	Singleton	40
7.1.6.3.2.	Prototype	41
7.1.6.3.3.	Request	42
7.1.6.3.4.	Session	42
7.1.6.3.5.	Application	43
7.1.6.3.6.	WebSocket	43
7.1.7.	Aspect Oriented Programming (AOP)	44
7.1.8.	Struktur für ein Projekt	45
7.1.9.	OpenAPI UI	46
7.1.10.	Authentication	47
7.2.	Jakarta EE	48
7.3.	Lombok	49
7.4.	Buildtools	50
7.4.1.	Gradle	50
7.4.2.	Maven	51
7.5.	Documentation	53

8. Frontend	54
8.1. Frameworks	54
8.1.1. React	54
8.1.2. Svelte	55
8.1.3. VueJS	56
8.1.4. Angular	57
8.2. Web Components	58
8.3. CSS	59
9. Dev Ops	60
9.1. Docker	60
9.1.1. Dockerfile	60
9.1.2. Image	60
9.1.3. Container	60
9.1.4. Volumes	60
9.1.5. Networking	60
9.1.6. Compose	60
9.2. Podman	61
9.3. Nginx	62
10. Debugging	63
10.1. Backend Debugging	63
10.2. Frontend Debugging	64
10.2.1. Browser DevTools	64
10.2.1.1. HTML/CSS Inspection	64
10.2.1.2. JavaScript Console	64
10.2.1.3. JavaScript Debugger	64
10.2.1.3.1. Source Map	64
10.2.1.4. Network Operations	65
10.2.2. IDE Debugging	66
10.2.3. Extensions	67
11. Seminare	68
11.1. Installieren der wichtigen Software	68
11.1.1. IntelliJ	68
11.1.2. Docker	68
11.1.3. Podman	68
11.1.4. nodejs	68
Quellenverzeichnis	70

1. Ablauf & Planung

- Ziel des Moduls: Projektarbeit mit einem Beleg und einer Präsentation als finales Ziel
- Projekt sollte einen Großteil der Aspekte der Web Entwicklung abdecken
- Basis für das Projekt: User-Stories, die das Projekt leiten sollen
- Gruppenarbeit möglich
- Beleg mit Code als Abgabe am Ende des Semesters

2. Anforderungen

2.1. Projekt

- Datenbank, Backend, Frontend mit Container Deployment (Docker, Podman, ...)
- Dokumentation der REST API Endpunkte mit OpenAPI o.ä.
- Einige Tests in Front- und Backend. Komplettes Test coverage wird nicht vorausgesetzt
- Einreichung des Repositories (ZIP, Link zu GitHub oder andere VCS)

2.2. Beleg

- Seitenanzahl nicht festgelegt. Bewegt sich wahrscheinlich um 20 Seiten, wird aber nicht vorausgesetzt
- Beschreibung, wie Anforderungen aus den User Stories umgesetzt wurden
- Umsetzung beschreiben
- Gründe für Entscheidungen bei der Entwicklung darstellen
- Dokumentation der einzelnen Software Bestandteile

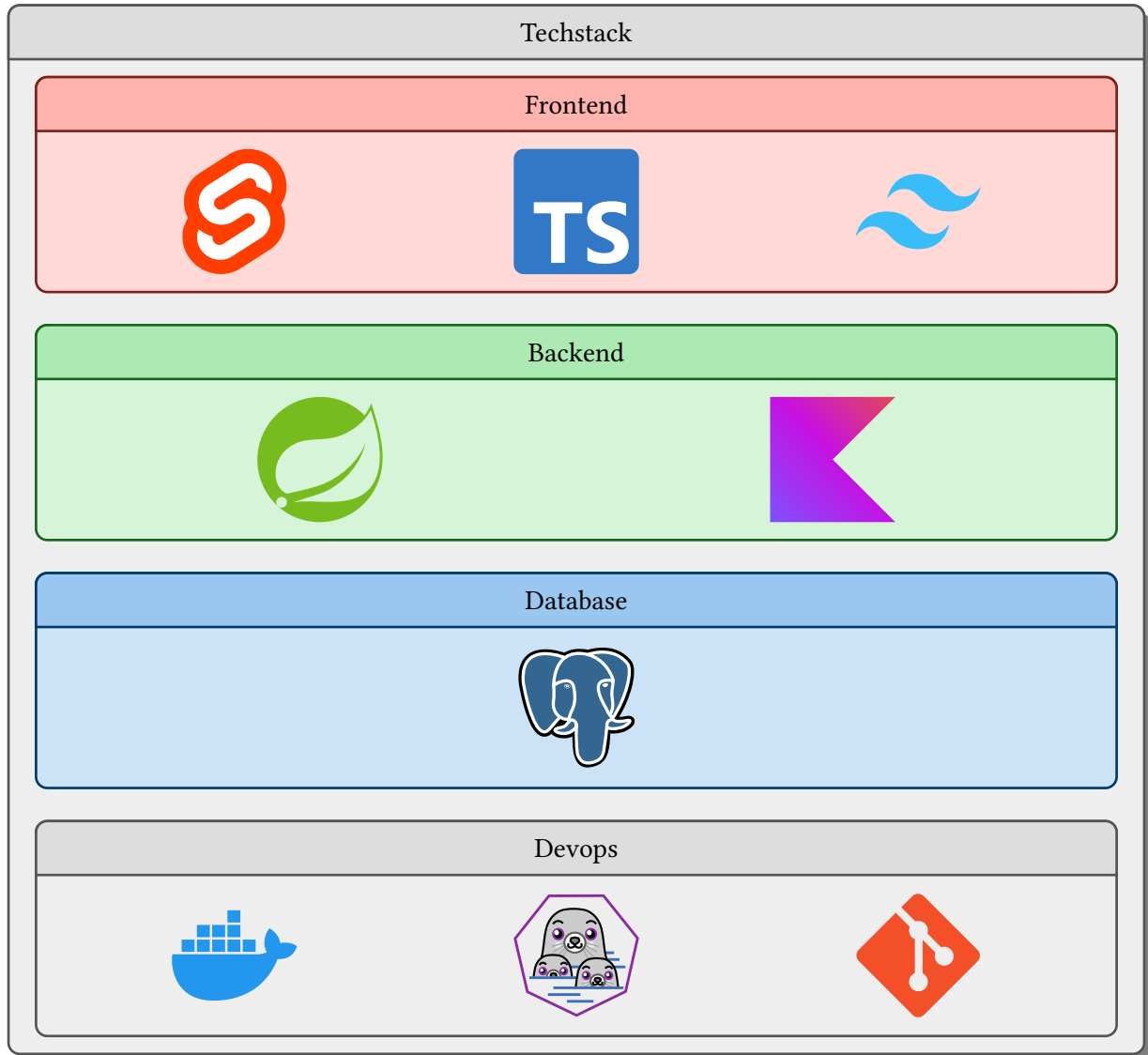
2.3. Präsentation bzw. Verteidigung

- Demonstration des finalen Produkts
- Vorstellung der Umsetzung
- Kurzes Zeigen von ausgewählten Programmbestandteilen, die als wichtig angesehen werden

3. Übersicht

3.1. Struktur einer Full Stack Web Application

Wie der Name vermuten lässt, wird in einer Full Stack Anwendung jeder Aspekt von einem Tech Stack implementiert. Im Bereich der Web Entwicklung ist dies meist eine Kombination aus einem Backend und einem Frontend. Für das Backend kann auch noch eine Datenbank bereitgestellt werden. Welche Technologien zu einem Projekt gehören wird im geplant. Folgender Stack kommt hier in der Vorlesung zum Einsatz



3.2. Spring Übersicht

Spring ist, vereinfacht gesagt, ein Framework, welches Infrastruktur bereitstellt, die das Entwickeln von Java basierten Anwendungen vereinfachen soll. Um das zu erreichen kommt es mit einigen Features daher wie zum Beispiel Dependency Injection und einer Liste an Modulen wie zum Beispiel:

- Spring JDBC
- Spring MVC
- Spring Security
- Spring Test

...

Diese Module sollen die Entwicklungszeit von oft gewollten Funktionalitäten stark verringern. [2] Durch den Modularen Aufbau des Spring Frameworks ist es Entwicklern auch offen gestellt, welche Module sie wirklich in ihr Projekt mit übernehmen wollen. Die Kern Module sind dabei alle Module um den IoC Container. Dazu gehören Dependency Injection Module und ein Konfigurations Modell.

Über die Kernfunktionen hinaus werden noch weitere Architekturen wie Messaging, Daten Austausch, Persistenz und Web unterstützt. Für Web bietet Spring auch noch das, auf Servlet basierende, Spring MVC Framework an. Als Alternative für Web gibt es auch noch Spring WebFlux. [3]

3.2.1. Geschichte von Spring

Spring wurde im Jahre 2003 als Antwort auf die hohe Komplexität von J2EE Spezifikationen erschaffen. Heutzutage existiert es komplementär neben Jakarta EE und seinem Vorgänger Java EE. Spring hat sich dabei einige Spezifikationen von Java EE angeeignet. Dazu gehören:

- Servlet API
- WebSocket API
- Concurrency Utilities
- JSON Binding API
- Bean Validation
- JPA
- JMS

Neben diesen Spezifikationen unterstützt Spring auch Dependency Injection und Common Annotations. Diese basierten früher javax Packages.

Seit Spring 6.0 wurden die Spezifikationen auf das Level von Jakarta EE 9 gehoben. Damit wurde auch die javax Packages als Basis ausrangiert und durch den jakarta Namespace ersetzt. Kompatibilität mit EE 10 wurde auch bereits hergestellt.

Auch die Anwendungsbereiche von Spring Applikationen haben sich über die Zeit verändert. Früher wurden Anwendungen entwickelt um auf einem Application Server eingesetzt zu werden. Heute wird mit Springboot eher in einer Devops- und Cloud-Freundlichen Weise entwickelt. Dafür wurde der Servlet Container in das Programm eingebettet und sein Austauschen trivialisiert. Seit Spring 5 können so auch WebFlux Applikationen ohne die Servlet API laufen und somit auch auf Servern eingesetzt werden, die keine Servlet Container sind (zum Beispiel Netty). [4]

3.2.2. Springboot

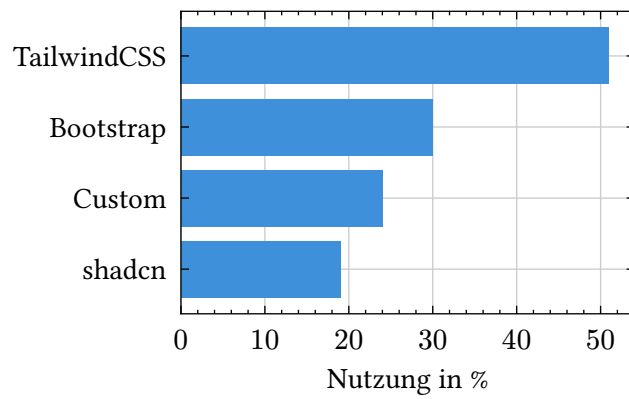
Springboot ermöglicht die Erstellung von Spring Anwendungen, die mit minimaler Konfiguration lauffähig sind. Es ist dabei eine Erweiterung des Spring Frameworks. [2] Dazu werden Webserver, wie zum Beispiel Tomcat, direkt mitgeliefert. Um den Start der Entwicklung zu vereinfachen werden außerdem *Starter Dependencies* bereitgestellt. Durch Springboot wird außerdem die Konfiguration

durch XML, wie sie öfter bei Spring benötigt wird, komplett umgangen. Somit stellt Springboot eine Abstraktion des Spring Frameworks dar und vereinfacht damit dessen Nutzung. [5]

3.3. Svelte Übersicht

3.4. Tailwind CSS Übersicht

Die aktuelle Verteilung der CSS Frameworks nach dem State of CSS 2025 [6]:



4. Grundbegriffe

4.1. Client Side

Alles, womit der Nutzer interagiert, ist Client Side. Damit es für den Nutzer so einfach wie möglich ist, die Anwendung zu bedienen wird eine gute UI mit guter UX benötigt. All das gehört zum Frontend.

4.2. Server Side

Alles, was auf dem Server passiert ist Backend Programmierung. Hier werden Funktionalitäten für das Frontend bereitgestellt. Dazu gehören zum Beispiel Daten verarbeiten und bereitstellen oder Authorisation.

4.3. API

Das Frontend und Backend kommunizieren über APIs. Sie können Anfragen vom Client an den Server and umgekehrt verarbeiten. Des weiteren dienen sie dazu, Anwendungen in Schichten zu unterteilen aber dennoch Kommunikation zwischen diesen Schichten zu ermöglichen.

APIs und andere Funktionen können dem Oberbegriff der Middleware zugeordnet werden.

4.4. REST

REpresentational **S**tate **T**ransfer ist ein Architektur Stil um Standards zwischen Computer Systemen im Web zu etablieren, die Kommunikation zwischen diesen Systemen vereinfachen. RESTful Systeme zeichnen sich vor allem dadurch aus, dass sie keinen State besitzen und die Angelegenheiten von Server und Client separieren. [7]

In einer RESTful Umgebung werden außerdem CRUD (Section 4.8) Operationen direkt auf HTTP Methoden (Section 4.5.3) gemapped. [8]

4.4.1. Unabhängigkeit

Die Implementation von Server und Client sollte unabhängig voneinander sein. Änderungen im Code bei einer der beiden Komponenten sollte nicht die Funktionalität des jeweils anderen beeinflussen. Nur das Nachrichtenformat muss beiden Seiten bekannt sein. Wenn dieses Format durchgehend eingehalten wird, können Änderungen ohne Probleme gemacht werden.

4.4.2. Stateless

Weder der Server noch der Client wissen etwas über den Zustand des jeweils anderen. Damit können beide alle Nachrichten, die sie empfangen, verstehen ohne Kontext dafür zu besitzen. Erreicht wird das durch das Nutzen von Ressourcen.

4.4.3. Kommunikation zwischen Client und Server

Die Kommunikation zwischen Client und Server findet über Requests und Responses statt. Der Client sendet Requests an den Server um Daten zu erhalten, erstellen oder modifizieren. Der Server antwortet mit einer Response.

4.5. HTTP

4.5.1. Request

Die Request vom Client hat einige Bestandteile:

- Ein **HTTP Verb**, dass die Art der Operation definiert
- Einen **Header**, der Informationen über die Request enthält
- Einen **Pfad** zu einer Ressource
- Einen optionalen **Body**, der weitere Daten enthält

4.5.1.1. Header

Der Header wird angegeben, welche Art von Ressource der Client akzeptiert. Definiert wird das im accept Feld. Diese Ressourcen werden über MIME Types (Multipurpose Internet Mail Extensions) definiert. Der Grundaufbau eines MIME Types ist wie folgt: type/subtype;parameter=value. Das parameter Feld ist dabei optional.

Eingie Beispiele für MIME Types: image/png, audio/wav, application/json

4.5.1.2. Pfad

Der Pfad definiert, auf welcher Ressource die Operation ausgeführt werden soll. Es ist dabei anzustreben, dass in den APIs diese Pfade so gesetzt werden, dass sie gut vom Client lesbar sind. So sollte der erste Teil des Pfades die Pluralform der Ressource sein.

Beispiel: store.com/customers/223/orders/12

Dieses Format erlaubt einfache Lesbarkeit des Pfades, auch wenn man selbst nicht mit der API vertraut ist.

4.5.2. Response

Wenn der Server mit einer Menge an Daten antworten will muss er einen Content Type in den Header seiner Antwort packen. Auch hier werden wieder MIME Types genutzt.

Dazu wird auch noch ein Status Code angehängen.

4.5.3. Methoden

HTTP definiert eine Menge an Verben, damit das Ziel einer Request einfacher zu erkennen ist und auch direkt klar ist, was das zu erwartende Ergebnis der Anfrage ist. Die folgenden vier HTTP Verben kommen dabei am häufigsten zum Einsatz kommen: **GET, POST, PUT, DELETE**

4.5.3.1. GET

Die GET Methode stellt eine Anfrage an den Server, eine Ressource zu transferieren. GET Anfragen auf die gleiche Ressource sollten immer die gleichen Ergebnisse liefern. Damit stellt GET den Hauptmechanismus zum Ressourcen Erhalten dar.

Der Client sollte nie Content mit einer GET Request generieren.

GET Requests haben die Möglichkeit gecached zu werden. Dieser Cache kann dann genutzt werden, um zukünftige Requests zu erfüllen.

Es ist zu beachten, dass wenn Ressourcen nur über URIs angefragt werden, potentiell sicherheitskritische Informationen in dieser URI landen können. Wenn es nicht möglich ist, diese Informationen in weniger kritische zu transformieren wird das Nutzen einer POST Request mit den Daten im Request Content empfohlen. [9]

4.5.3.2. POST

Die POST Methode wird genutzt, um die transferierten Daten in der Request nach den Spezifikationen des Servers zu verarbeiten. Einige Beispiele hier sind:

- Daten, die in Input Felder eingetragen wurden, zu übergeben
- Nachrichten Posten, zum Beispiel in Foren, Social Media usw.
- Erstellen einer neuen Ressource
- Daten an eine bereits existierende Ressource anhängen

Der Server gibt dann mit Status Codes an, was das Ergebnis der POST Request ist. Die erwarteten Status Codes sind hier: 206 (Partial Content), 304 (Not Modified), 416 (Range Not Satisfiable)

Wenn durch die POST Request eine neue Ressource erstellt wurde, sollte der Server mit 201 (Created) antworten und den Ort der neuen Ressource in die Response packen. [10]

4.5.3.3. PUT

Die PUT Methode erstellt eine Anfrage an den Server, eine bereits vorhandene Ressource zu ersetzen oder neu zu erstellen, basierend auf den Daten in der Anfrage. Wenn die angesprochene Ressource noch nicht existiert, wird sie neu erstellt. Nach dem Erstellen einer neuen Ressource muss der Server den Client darüber mit dem Status Code 201 (Created) informieren.

Wurde kein neuer Eintrag angelegt, muss der Server den Client über den Erfolg der Request mit dem Status Code 20 (OK) oder 204 (No Content) informieren. Der Server sollte die Daten in der PUT Request validieren. Hier ist vor allem das Ziel zu überprüfen, ob die bereitgestellten Daten mit der ausgewählten Ressource übereinstimmen. Sollte dies nicht der Fall sein, kann der Server versuchen diese Daten in das richtige Format zu bringen, oder er informiert den Client über das fehlerhafte Datenformat. Die Status Codes für Fehler in diesen Daten sind 409 (Conflict) und 415 (Unsupported Media Type). [11]

4.5.3.4. DELETE

Die DELETE Methode erstellt eine Request an den Server, eine Ressource zu entfernen. Je nachdem, wie das Löschen im Server definiert wurde, werden nur Referenzen auf die Ressource gelöscht oder auch die Ressource selbst entfernt. DELETE Requests sollten nur auf Ressourcen zugelassen werden, die ein definierten Ablauf für das Löschen besitzen.

Wenn die DELETE Methode erfolgreich war, sollte der Server mit einem der folgenden Status Codes antworten:

- 202 (Accepted) wenn das Löschen wahrscheinlich erfolgreich sein wird, aber noch nicht durchgeführt wurde
- 204 (No Content) Löschen wurde ausgeführt und keine weiteren Informationen sind nötig
- 200 (OK) Löschen war erfolgreich und die Response enthält noch Informationen über den aktuellen Status

[12]

4.5.3.5. Idempotente Methoden

Eine Methode wird dann als idempotent angesehen, wenn sie bei multipler Ausführung den gleichen Effekt auf dem Server haben. Diese multiple Ausführung ist vor allem dann wichtig, wenn man automatisch Anfragen erneut ausführen möchte. Zum Beispiel, wenn eine Anfrage fehlschlägt und automatisch eine Neue versucht wird.

PUT und DELETE sind dabei automatisch idempotent. Außerdem sind *safe request methods* idempotent.

Die Definition von idempotent ist dabei nur wichtig für den Inhalt, den der Nutzer angefragt hat. Der Server kann immer noch Logs über Anfragen führen oder andere Nebeneffekte implementieren, die den idempotenten Status nicht gefährden.

Der Client sollte Requests mit nicht idempotenten Methoden nicht automatisch erneut ausführen, außer es ist bekannt, dass die Implementation dieser Methode doch idempotent ist. [13]

4.5.4. Status Codes

Wenn der Server eine Response an den Client schickt wird auch immer ein Response Code mitgeliefert. Diese geben Informationen über den Erfolg der Operation. Der Status Code ist immer ein drei stelliger Integer und reicht von 100 bis 599.

Die erste Ziffer gibt dabei die Klasse der Response an. Die letzten beiden haben keine Kategorisierung. Folgende Bedeutungen sind den ersten Ziffern zuzuweisen:

- 1xx (Informational): Die Request wurde erhalten und wird verarbeitet
- 2xx (Successful): Die Request wurde erfolgreich erhalten, verstanden und akzeptiert
- 3xx (Redirection): Es müssen weitere Schritte durchgeführt werden, damit die Request verarbeitet werden kann
- 4xx (Client Error): Die Request enthält falschen Syntax oder kann nicht erfüllt werden
- 5xx (Server Error): Der Server konnte eine eigentlich valide Request nicht erfüllen

[14]

Hier sind einige der meist genutzten Status Codes:

Status Code	Bedeutung
200 (OK)	Die standard Antwort für eine erfolgreiche Request
201 (CREATED)	Die standard Antwort für eine erfolgreiche Request, die eine neue Ressource anlegen sollte.
204 (NO CONTENT)	Die standard Antwort für eine erfolgreiche Request, die keine Daten in ihrem Body zurückschickt
400 (BAD REQUEST)	Die Request konnte nicht verarbeitet werden. Gründe könnten sein: falscher Syntax, zu große Datenmengen usw.
403 (FORBIDDEN)	Der Client hat keine Rechte auf diese Ressource zuzugreifen
404 (NOT FOUND)	Die Gewünschte Ressource konnte nicht gefunden werden
500 (INTERNAL SERVER ERROR)	Die generische Antwort für einen unerwarteten Fehler, wenn es keine weiteren Informationen über die Art des Fehlers gibt.

4.6. DTO

Data-Transfer-Object bündelt mehrere Datenfelder in einem Objekt, damit sie in einem Aufruf übertragen werden können. Da alle Daten in einem Objekt vorhanden sind, gibt es bei der Serialisierung auch nur einen Punkt, an dem die Daten umgewandelt werden was spätere Änderungen stark vereinfacht. Zu guter Letzt tragen sie auch zur Teilung von den Domain Models und der Darstellungs Schicht bei, damit beide unabhängig sich ändern können.

4.7. MVC

Model-View-Controller ist eine Design Pattern, welches genutzt wird um User Interfaces, Daten und Kontroll Logik zu implementieren. Ein großer Fokus ist dabei die Separierung von der Business Logik und der visuellen Darstellung. Durch diese Separierung soll das Arbeiten mit mehreren Team Mitgliedern erleichtert werden und die Wartbarkeit der Software erhöht werden.

MVC besteht aus drei Komponenten deren Aufgaben wie folgt definiert sind:

- **Model:** Verwaltet Daten und Business Logik
- **View:** Übernimmt Layout und Darstellung
- **Controller:** Leitet Befehle zum Model und View weiter

[15]

4.8. CRUD

CRUD beschreibt die vier Grundoperationen, die benötigt werden, um mit persistenten Daten zu interagieren.

- **Create:** Neue Daten Einträge werden gespeichert.
- **Read:** Existierende Daten werden ausgelesen
- **Update:** Existierende Daten werden aktualisiert
- **Delete:** Daten werden gelöscht

[8]

4.9. ORM

Object-Relational-Mapping ist eine Technik, die es erlaubt Daten von einer relationalen Datenbank zu Objekten in einer Programmiersprache zu konvertieren. Damit wird eine virtuelle Objekt Datenbank erstellt, die innerhalb eines Programms genutzt werden kann. [16]

Mithilfe von ORM Frameworks können so auch CRUD Operationen ausgeführt werden. Dabei ist es nicht notwendig, SQL-Befehle selbst zu schreiben, da sie vom Framework selbst generiert werden.

[17]

5. Technologien

5.1. Frontend

- Svelte/SvelteKit
- Vue
- React
- Angular
- Astro
- Vanilla
- Rails
- Laravell
- Symphony
- Vite/Webpack
- NodeJS/Bun/Deno/Yarn
- TailwindCSS
- PRIME

5.2. Backend

- Tomcat Servlet zu Spring Beans (Aufbau)
- Spring
- Micronaut
- Quarkus
- Kotlin
- Gradle

5.3. Database

- PostgreSQL

5.4. Testing

- Cypress, Playwright

5.5. DevOps

- Docker
- Podman

6. User Stories

Kernpunkte einer User Story [18]:

- Wer ist der User
- Was will der User machen
- Warum will der User das machen
- Weitere Informationen sind optional

Template [18]:

AS A {user|persona|system}

INSTEAD OF {current condition}

I WANT TO {action} IN {mode} TIME | IN {differentiating performance units} TO {utility performance units}

SO THAT {value of justification}

NO LATER THAN {best by date}

6.1. User Stories für das finale Projekt

7. Backend

7.1. Spring

7.1.1. Komponenten einer Spring Anwendung

Beispielkomponenten Anhand eines Users. Dieser User hat folgende Felder:

- Name
- Alter

7.1.1.1. Controller

```
1  @RestController
2  @RequestMapping("/path/to/controller")
3  class Controller @Autowired constructor(
4      private val service: Service
5  ) {
6      @GetMapping
7      @ResponseStatus(HttpStatus.OK)
8      fun getEntities(): List<GetEntityDto> {
9
10     }
11 }
```

Kotlin

7.1.1.1.1. Mappings

Mit der `@RequestMapping` Annotation können Anfragen auf Methoden in einem Controller gemapped werden. Es kommt mit unterschiedlichen Parametern, die Zuordnung über verschiedene Wege erlauben. Folgende Parameter werden unterstützt: URL, Http Methode, Request Parameter, Header und Media Typen.

Für HTTP Methoden gibt es weitere Annotationen als Abkürzungen von `RequestMapping`:

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`

Diese Annotation sind selbst mit `RequestMapping` versehen, decken aber einen kleineren Bereich ab. Diese Annotationen wurden eingeführt, da ein Controller die meisten seiner Methoden auf eine bestimmte HTTP Methode mappen sollte. `@RequestMapping` würde da nicht in Frage kommen, da es auf jede beliebige HTTP Methode in betracht zieht.

Nutzen von mehreren `@RequestMapping` Annotationen

Hinweis 1

Es kann immer nur eine `@RequestMapping` an einem Element geben. Ein Element ist in diesem Fall eine Klasse, Methode oder ein Interface. Sollten dennoch mehrere Annotationen an einem Element vorhanden sein, wird nur die erste genutzt. Diese Regel gilt auch für Annotationen, die auf `@RequestMapping` basieren.

[19]

7.1.1.1.2. URI Patterns

Beispiele:

- `"/resources/ima?e.png"` - Ein Zeichen wird abgeglichen

- `"/resources/*.png"` - Eine beliebige Anzahl an Zeichen wird abgeglichen
- `"/resources/**"` - Mehrere Pfad Segmente werden abgeglichen
- `"/projects/{project}/versions"` - Pfad Element wird abgeglichen und als Variable ausgelesen
- `"/projects/{project:[a-z]+}/versions"` - Pfad Element wird abgeglichen mit Regex und als Variable ausgelesen

Pfad Element, die mit `{}` als Variable gespeichert wurden, können mit `@PathVariable` ausgelesen werden.

```
1 @GetMapping("/owners/{ownerId}/pets/{petId}")
2 fun findPet(@PathVariable ownerId: Long, @PathVariable petId: Long): Pet {
3
4 }
```

Pfad Variablen können auch schon auf der Klassen Ebene deklariert werden.

```
1 @Controller
2 @RequestMapping("/owners/{ownerId}")
3 class OwnerController {
4     @GetMapping("/pets/{petId}")
5     fun findPet(@PathVariable ownerId: Long, @PathVariable petId: Long): Pet {
6
7     }
8 }
```

Variablen aus der URI werden automatisch in den korrekten Typ konvertiert. Einfache Typen wie zum Beispiel `int`, `long` oder `Date` werden direkt unterstützt. Konvertierungen für komplexere Typen können implementiert werden.

Wenn die namen im Pfad und der Variable nicht übereinstimmen kann auch ein Name angegeben werden, nachdem gesucht werden soll.

```
1 @PathVariable("customName")
```

[20]

7.1.1.1.3. Error Handling

```
1 class CustomException(message: String): RuntimeException(message) {
2
3 }
```

```
1 @ControllerAdvice
2 class ExceptionControllerAdvice {
3     fun handleCustomException(
4         ex: CustomException
5     ): ResponseEntity<ErrorMessageModel> {
6         val errorMessage = ErrorMessageModel(
7             HttpStatus.NOT_FOUND.value(),
8             ex.message
9         )
10        return ResponseEntity(
```

```

11     errorMessage,
12     HttpStatus.NOT_FOUND
13 )
14 }
15 }

```


[21]

@ControllerAdvice wird hier alle Controller im Programm ansprechen. Alternativ kann das Einflussgebiet eingeschränkt werden. Hier sind einige Beispiele zur Einschränkung:

```

1 @ControllerAdvice(annotations = [RestController::class])
2 class Advice

```

 Kotlin

Die Klasse wird nur noch Controller, die mit @RestController annotiert sind, ansprechen.

```

1 @ControllerAdvice("org.example.controllers")
2 class Advice

```

 Kotlin

Die Klasse wird nur noch Controller, die sich in dem angegebenen Package Pfad befinden, ansprechen.

```

1 @ControllerAdvice(
2     assignableTypes = [
3         ControllerInterface::class,
4         AbstractController::class
5     ]
6 )
7 class Advice

```

 Kotlin

Die Klasse wird nur noch Controller ansprechen, die auf das Interface oder den abstrakten Controller zugewiesen werden können. [22]

7.1.1.2. Service

```
1 @Service
2 class EntityService (
3     private val entityRepository: EntityRepository
4 ) {
5     // service functions
6 }
```

Kotlin

Der Service enthält die Business Logik der Anwendung. Um einen Service zu deklarieren wird die `@Service` Annotation genutzt. Typischerweise sollte ein Service mindestens eine Funktion pro Controller Mapping enthalten. Diese Funktion sollte das einzige sein, was der Controller aufruft.

7.1.1.2.1. Transactional Methoden

Für manche Funktionen im Service kann es sinnvoll sein, die `@Transactional` Annotation zu nutzen. Mit dieser Annotation wird das Spring Transaction Management genutzt, um Transaktionen durchzuführen. Ohne diese Annotation würde, zum Beispiel, Spring Data JPA oder Hibernate die Transaktion durchführen.

[23]

Die `@Transactional` Annotation sollte in folgenden Situationen genutzt werden:

- **Datenbank Operation mit mehreren Schritten:** Wenn mehrere Operationen erfolgreich sein oder fehlschlagen müssen. Ein Beispiel hier wäre eine Transaktion in einem Banksystem. Beim Sender muss das Geld entfernt werden und beim Empfänger muss das Geld hinzugefügt werden. Durch `@Transactional` würde bei einem Fehlschlag einer dieser Operation die Datenbank konsistent bleiben und keine Änderungen committed werden.
- **Wenn die Operation kaskadierende Updates durchführt:** Wenn eine Operation Daten in einer oder mehrerer anderer Tabellen beeinflusst, sollte `@Transactional` verwendet. Ein Beispiel hier wäre das Entfernen eines Users, welches Löschungen in anderen Tabellen auslösen würde. `@Transactional` sorgt hier dafür, dass nur alle Änderungen zusammen angenommen werden. Bereits ein Fehler sorgt für den Abbruch aller Operationen.
- **Wenn volatile Daten benutzt werden:** Wenn Fehler erwartet werden oder sehr wahrscheinlich sind. Durch `@Transactional` wird Korruption von Daten in solchen Situationen vermieden.
- **Wenn bei Daten parallelität bzw. concurrency erwartet wird:** Wenn mehrere Nutzer parallel an einer Datenbank arbeiten können, kann `@Transactional` genutzt werden, um Operationen voneinander zu isolieren. Änderungen werden erst sichtbar, wenn alle dazugehörigen Operationen erfolgreich beendet wurden.


Beispiel für eine Entity mit Funktion bei der `@Transactional` benutzt werden sollte.

```
1 @Entity(name = "user")
2 @Table(name = "user")
3 class User(
4     @Id
5     @GeneratedValue(strategy = GenerationType.AUTO)
6     val id: Long? = null,
7     @OneToMany(
8         mappedBy = "user",
9         cascade = [CascadeType.REMOVE]
10 )
```

Kotlin

```
11  val roles: List<Role> = emptyList()
12 )
```

```
1  @Transactional
2  fun deleteUser(id: Long) {
3      val toDeleteUser = getUserById(id)
4      userRepository.delete(toDeleteUser)
5  }
```

 Kotlin

Wann wird `@Transactional` nicht benötigt:

- **Updates die nur einen Schritt enthalten und nicht kaskadieren:** Wenn nur eine Tabelle oder Record nacheinander geändert wird und keine Einflüsse auf andere Tabellen vorhanden sind und Daten nicht zwingend konsistent sein müssen wird `@Transactional` nicht benötigt.
- **Wenn strikte Konsistenz nicht benötigt wird**
- **Methoden, die keine Daten modifizieren**

[24]

7.1.1.3. Entity

```
1 @Entity(name = "entityName")
2 @Table(name = "entityName")
3 class Entity {
4     @Id
5     @GeneratedValue(strategy = GenerationType.IDENTITY)
6     val id: Long? = null,
7     // ...
8 }
```

Kotlin

7.1.1.3.1. @Entity Annotation

Um eine Entity zu erstellen, muss eine Klasse mit @Entity annotated werden. Bei der @Entity Annotation kann ein Name angegeben werden. Dieser Name wird verwendet, wenn in Queries Zugriff auf die Entity vorkommt. Wenn kein Name vorhanden ist, wird der Klassenname verwendet. [25]

```
1 @Entity(name = "student")
```

Kotlin

7.1.1.3.2. @Table Annotation

Dazu kann noch eine @Table Annotation angebracht werden. Wenn keine vorhanden ist, wird eine Tabelle mit dem Namen der Entity in der Datenbank erstellt. Mit der @Table Annotation kann ein eigener Name dafür festgelegt werden. Außerdem kann auch ein Schema Name angegeben werden. [25]

```
1 @Table(name="STUDENT", schema="SCHOOL")
```

Kotlin

7.1.1.3.3. @Id Annotation

Jede Entity muss einen Primary Key besitzen. Wenn ein Datenfeld mit @Id annotated wurde, ist es der Primary Key. Die Id wird automatisch generiert. Wie diese Generation geschieht kann über @GeneratedValue festgelegt werden. Folgende Optionen stehen zur Verfügung:

- AUTO
- TABLE
- SEQUENCE
- IDENTITY

Wenn AUTO genutzt wird, wird automatisch entschieden, welche der Strategien gerade genutzt werden soll.

[25]

Id einer Entity muss als **null** definiert sein

Hinweis 2

Eine Id muss nullable sein und als default Wert null besitzen. Wenn das nicht der Fall ist, wird Spring beim Erstellen einer neuen Entity einen Error werfen. Grund hierfür ist, dass eine definierte Id Spring davon ausgehen lässt, dass es diesen Eintrag bereits in der Datenbank gibt. Wenn die Id null ist, weiß Spring, dass es sich hier um einen neuen Eintrag handelt.

```
1 @Id
2 @GeneratedValue(strategy = GenerationType.AUTO)
3 val id: Long? = null
```

Kotlin

7.1.1.3.4. @Column Annotation

Datenfelder in einer Entity können mit der @Column genauer beschrieben werden. Folgende Informationen können über diese Annotation angegeben werden:

- name: Der Name des Columns in der Datenbank.
- length: Die maximale Länge des Datenfeldes. Zum Beispiel wäre das bei einem String die Anzahl der Zeichen.
- nullable: Definiert, ob das Datenfeld null sein darf oder immer einen Wert besitzen muss.
- unique: Definiert, ob dieses Datenfeld ein unique Key sein soll.

[25]

7.1.1.3.5. Referenzen auf andere Tables

Oft muss eine Entity Referenzen auf andere Entities benutzen. So zum Beispiel, wenn es User und Rollen Entities gibt. Je nachdem, wie die Implementation geschehen muss, könnten hier @OneToMany oder @ManyToMany Relationen genutzt werden.

@OneToMany würde zum Einsatz kommen, wenn ein User nur eine Rolle haben kann, aber eine Rolle auf mehrere User zugeordnet sein kann.

@ManyToMany würde zum Einsatz kommen, wenn ein User mehrere Rollen haben kann und eine Rolle auch mehrere User besitzen kann.

7.1.1.3.5.1. @OneToMany und @ManyToOne

Es wird dem oben genannten Beispiel gefolgt. Ein User besitzt eine Role und eine Role hat mehrere User zugewiesen. Der User ist in diesem Fall das Child und die Role ist der Parent. Das Child ist dabei der Besitzer der Relation. Dafür besitzt Role eine Id als Primary Key, die als role_id in Form eines Foreign Key im User Table gespeichert wird.

Da Role mehrere User besitzen kann, wird hier eine Liste erstellt, die mit Objekten vom Typ User befüllt werden kann. Außerdem wird die @OneToMany Annotation hier angebracht, mit der Information, dass die Variable "role" in der User Klasse genutzt wird, um auf die Role, also die eigene Instanz, zu verweisen. Das mappedBy Attribut weist außerdem auf die Tatsache hin, dass hier keine Verwaltung von Foreign Keys stattfinden wird. Neben mappedBy kommt auch cascade und orphanRemoval oft vor. Mit dem cascade können Operationen angegeben werden, die auch alle Childobjekte betreffen sollen. Werden hier keine Informationen angegeben, werden zum Beispiel Childobjekte nicht gelöscht, wenn das Parent Objekt gelöscht wird. Foreign Keys werden auch erhalten, was zu Problemen führen kann.


orphanRemoval kann auch genutzt werden, damit Child Objekte automatisch gelöscht werden, wenn sie von ihrem Parent Objekt getrennt werden.

In der User Klasse muss nun die role Variable vorhanden sein. Durch das Hinzufügen von @ManyToOne wird eine bidirektionale Relation erschaffen. So können wir auch in der User Klasse alle Daten über Role erhalten.

Außerdem ist die @JoinColumn Annotation vorhanden. Sie sollte typischerweise auf der Seite der Relation vorhanden sein, die die Relation besitzt.

In den meisten Fällen ist es die Seite mit der @ManyToOne Annotation. Die Angabe eines referencedColumnName ist nicht zwingend notwendig aber empfohlen. Wenn sie nicht vorhanden ist, wird automatisch nach dem primary Key in der Entity gesucht.

```
1 @Entity(name = "user")
2 @Table(name = "user")
```

 Kotlin


```

3  class User (
4
5      @ManyToOne
6      @JsonBackReference
7      @JoinColumn(name = "role_id", referencedColumnName = "id")
8      var role: Role? = null
9
10 )

```

```

1  @Entity(name = "role")
2  @Table(name = "role")
3  class Role(
4
5      @OneToMany(
6          mappedBy = "role"
7      )
8      var users: List<User> = emptyList()
9
10 )

```

Kotlin

[26], [27]

7.1.1.3.5.2. @OneToOne

Die @OneToOne Annotation wird genutzt, wenn eine Entity nur eine Instanz einer Anderen enthält. Es wird dabei wieder @JoinColumn verwendet, allerdings nur bei der besitzenden Seite der Relation. Die andere Seite benötigt nur @OneToOne mit dem mappedBy Attribut. [28]

```

1  class User (
2      @OneToOne
3      @JsonBackReference
4      @JoinColumn(name = "address_id", referencedColumnName = "id")
5      var address: Address? = null
6  )

```

Kotlin

```

1  class Address (
2      @OneToOne(mappedBy = "address")
3      var user: User? = null
4  )

```

Kotlin

7.1.1.3.5.3. @ManyToMany

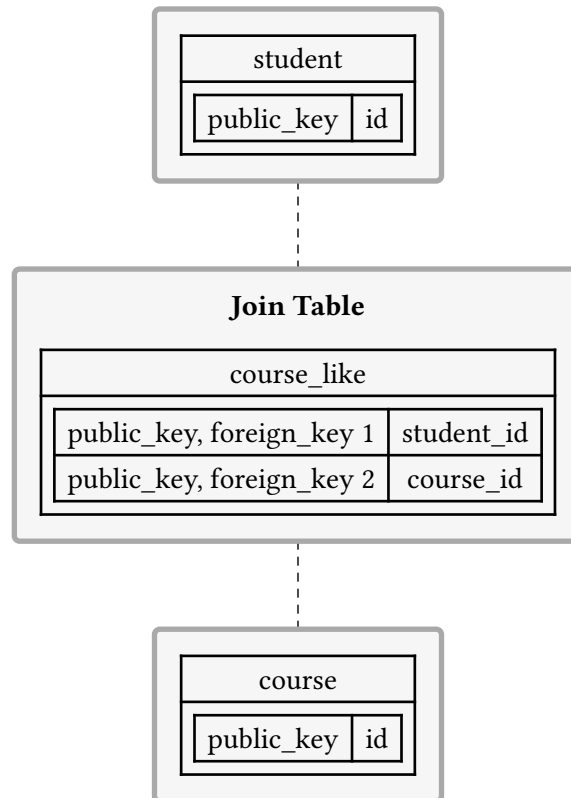


Figure 1: Aufbau einer Many to Many Relation mit einem Join Table

Für Many to many Beziehungen muss ein neuer Table angelegt werden, die Beziehungen speichert. In diesem Table werden Einträge gespeichert, die Foreign Keys zu den jeweiligen Entitäten enthalten. Somit wird für jeden Eintrag in der many to many Beziehung ein Eintrag in dieser Tabelle angelegt. Dieser daraus entstehende Table wird Join Table genannt.

In Spring wird diese Relation mit der @ManyToMany Annotation erstellt. Beide Seiten in der Beziehung erlarten diese Annotation. Auf der besitzenden Seite der Beziehung muss auch noch konfiguriert werden, wie das verbinden der Tabellen ablaufen soll. Diese Konfiguration geschieht mit der @JoinTable Annotation durchgeführt.

@JoinTable benötigt einen Namen, der für den Join Table genutzt werden soll. In dem joinColumns Feld wird die Verbindung zu der besitzenden Seite eingetragen mit @JoinColumn mit dem Namen der id. Zuletzt wird mit einem inverseJoinColumns die andere Seite der Relation mit einem @JoinColumn verbunden. Auch hier wird der Name der id benötigt.

Die Benutzung von @JoinColumn und @JoinTable ist dabei nicht zwingend notwendig. JPA ist in der Lage die Verbindung selbst herzustellen. Allerdings kann die dabei genutzte Strategie nicht immer die sein, die wir uns erwünschen, vor allem wenn unsere Namenskonventionen von denen in JPA definierten abweichen.


Auf der Zielseite muss nur der Name des Feldes angegeben werden, der die Relation mapped. [29]

```
1 class Student (
2     @ManyToMany
3     @JoinTable(
4         name = "course_like",
5         joinColumns = @JoinColumn(name = "student_id"),
```

```
6     inverseJoinColumns = @JoinColumn(name = "course_id")
7 )
8     val likedCourses: List<Course>
9 )
10
11 class Course (
12     @ManyToMany(mappedBy = "likedCourses")
13     val likes: List<Students>
14 )
```

7.1.1.4. Repository

```
1 @Repository
2 interface EntityReposirory: JpaRepository<Entity, Long> {
3
4 }
```



7.1.1.5. DTO


Das DTO ist eine Bündelung von mehreren Datenfeldern in ein Objekt. Der Inhalt dieses Objekts richtet sich nach seinem jeweiligen Einsatzzweck. So könnte ein DTO, welches als Response zu einer GET Request (Section 4.5.3.1) gehören soll, alle Felder der angesprochenen Entity enthalten. Sollten Verweise auf andere Tabellen vorhanden sein, könnten diese in Form einer ID übergeben werden oder direkt alle gewünschten Daten aus der Entity enthalten.

Ein DTO, welches zum Erstellen einer Entity genutzt werden würde, also in einer POST Request (Section 4.5.3.2), würde hingegen nur Daten enthalten, die zum Initialisieren benötigt werden. Bei einem User in einem Shop könnten das zum Beispiel Name, Adresse, Email, Passwort usw. sein.


Grundsätzlich lässt sich sagen, dass der Inhalt eines DTO immer so gewählt werden sollte, dass er für den Anwendungsfall gut zugeschnitten ist. So wird es wahrscheinlich dazu führen, dass es zu einer Entity mehrere DTOs gibt, die zu unterschiedlichen Situationen im Anwendungsprozess passen.

Es folgen einige Beispiele für DTOs die zu einer Entity gehören:

```
1 class Entity(  
2     val id: Long?,  
3     val name: String,  
4     val age: Integer  
5 )
```



```
1 data class GetEntityDto (  
2     val id: Long,  
3     val name: String,  
4     val age: Integer  
5 )
```



```
1 data class PostEntityDto (  
2     val name: String,  
3     val age: Integer  
4 )
```



7.1.1.6. Mapper

Die Aufgabe des Mappers ist es, ein DTO in eine Entity oder eine Entity in ein DTO zu überführen. In den meisten Fällen ist das eine triviale Aufgabe, da nur Werte aus Feldern kopiert werden müssen.

```
1 class Entity(Kotlin
2     val id: Long?,
3     val name: String,
4     val age: Integer
5 )
```

```
1 fun convertEntityToGetEntityDto(entity: Entity): GetEntityDto {Kotlin
2     return GetEntityDto(
3         id = entity.id!!,
4         name = entity.name,
5         age = entity.age
6     )
7 }
```

7.1.1.7. Application Konfiguration

```
1  # application.yml
2
3  server:
4    port: 8080
5
6  spring:
7    datasource:
8      url: jdbc:postgresql://localhost:5432/database
9      username: database_username
10     password: database_password
11   jpa:
12     hibernate:
13       ddl-auto: create-drop
14     show-sql: true
15     properties:
16       hibernate:
17         format_sql: true
```



7.1.2. Spring Data

7.1.2.1. Spring Data JDBC

7.1.2.2. Spring Data JPA

7.1.2.3. Spring Data R2DBC

7.1.2.4. Spring Data REST

7.1.3. IoC Container

In der Anwendung wird der IoC Container durch `org.springframework.context.ApplicationContext` representiert. Er instantiiert, konfiguriert und assembled Beans. Die Instruktionen für diese Operationen werden dem Container durch das Lesen von Konfigurations-Metadaten übergeben. Diese Metadaten können über folgende Wege definiert werden:

- Annotationen
- Konfigurations-Klassen mit Factory Methoden
- XML Dateien
- Groovy Scripts

Die manuelle Erstellung des IoC Containers ist in den meisten Fällen nicht von Nöten.

Spring kombiniert die vom Entwickler erstellen Klassen mit den Konfigurations-Metadaten, damit nach der Initialisierung des `ApplicationContext` ein konfiguriertes und ausführbares System bereitsteht [30].

7.1.4. Dependency Injection

[Ressource zu Dependency Injection \[31\]](#)

Das Ziel der Dependency Injection ist es, Abhängigkeiten zu entkoppeln. Diese Entkopplung macht den Code lesbarer und das Testen einfacher. Eine Klasse definiert nur noch, was sie für Abhängigkeiten benötigt. Sie sucht aber nicht selbst nach diesen Abhängigkeiten. Sie werden durch einen Container bereitgestellt. Das definieren der benötigten Abhängigkeiten kann durch Constructor Argumente, Factory Methoden oder Properties geschehen. Der Container übergibt beim Erstellen einer Bean die benötigten Abhängigkeiten. Die Bean hat in diesem Fall keine Kontrolle über die Erstellung oder den Ort ihrer Abhängigkeiten [31].

Bei Spring gibt es zwei Methoden zur Dependency Injection: **Constructor** basierte Injection oder **Setter** basierte Injection.

Constructor oder Setter DI

Richtlinie 3

Der Constructor sollte verpflichtende Abhängigkeiten enthalten.

Setter Methoden eignen sich gut für optionale Abhängigkeiten. `@Autowired` kann bei Settern genutzt werden, damit die Property eine verpflichtende Abhängigkeit wird. Der Constructor sollte da aber bevorzugt werden.

7.1.4.1. Constructor Injection

Der Container ruft einen Constructor mit so vielen Argumenten auf, wie Abhängigkeiten benötigt werden. Jedes Argument repräsentiert dabei eine Abhängigkeit.

```
1 class ExampleClass(private val dependency: Dependency) {  
2  
3 }
```

Kotlin

7.1.4.2. Setter Injection

Der Container ruft die Setter Methoden in den erstellten Beans auf, nachdem ein Constructor ohne Argumente aufgerufen wurde.

```
1 class ExampleClass {  
2     lateinit var dependency: Dependency  
3 }
```

Kotlin

7.1.5. Method Injection

[Ressource für Method Injection](#) [32]

7.1.6. Beans

Beans

Definition 4

Jedes Objekt, welches Teil der Anwendung ist und von dem Spring IoC Container verwaltet wird, ist eine Bean. Eine Bean kann instantiated, assembled oder anderweitig von dem Spring IoC container gemanaged werden. [33]

7.1.6.1. Spring Inversion of Control (IoC) Container

Ein Objekt definiert seine Abhängigkeiten, ohne diese zu erstellen. Der gesamte Lebenszyklus der Abhängigkeiten wird an den IoC Container ausgelagert. [1]

Dieser Ansatz wird dann wichtig, wenn in einem großen Projekt nur bestimmte Instanzen von Klassen benötigt werden oder eine Instanz im gesamten Projekt genutzt werden soll. Das Verwalten solcher Abhängigkeiten wird schnell kompliziert und Fehleranfällig.

Der Spring IoC Container löst dieses Problem. Wir als Entwickler müssen nur korrekte Metadaten zur Konfiguration bereitstellen. Der Container erledigt den Rest. [33]

Beispiel

```
1 public class Company {
2     private Address address;
3
4     public Company(Address address) {
5         this.address = address;
6     }
7 }
```

Java

```
1 public class Address {
2     private String street;
3     private int number;
4
5     public Address(String street, int number) {
6         this.street = street;
7         this.number = number;
8     }
9 }
```

Java

Traditionelle Erstellung der Abhängigkeiten:

```
1 Address address = new Address("High Street", 1000);
2 Company company = new Company(address);
```

Java


Herangehensweise mit Beans

```
1 @Component
2 public class Company {
3     // this body is the same as before
4 }
```

Java

Konfiguration des IoC Containers mit Metadaten zu den Address Beans:

```
1 @Configuration
2 @ComponentScan(basePackageClasses = Company.class)
3 public class Config {
4     @Bean
5     public Address getAddress() {
6         return new Address("High Street", 1000);
7     }
8 }
```

 Java

Die Config Klasse erstellt eine Bean vom Typ Address. Mit der @ComponentScan Annotation wird auch schon nach Beans im Container geschaut, die vom gleichen Typ sind, hier Company.


Um den IoC Container zu erschaffen, wird eine Instanz von AnnotationConfigApplicationContext benötigt.

```
1 ApplicationContext context = new
  AnnotationConfigApplicationContext(Config.class);
```

 Java

Die Funktionalität der Beans kann wie folgt verifiziert werden:

```
1 Company company = context.getBean("company", Company.class);
2 assertEquals("High Street", company.getAddress().getStreet());
3 assertEquals(1000, company.getAddress().getNumber());
```

 Java

7.1.6.2. Annotationen für Beans [1]

- @Component: Eine generelle Angabe, die eine Klasse als Spring Bean markiert
- @Service: Eine Klasse, die einen Service darstellt
- @Repository: Eine Klasse, die ein Repository darstellt, welches mit der Persistence-Layer interagiert
- @Controller: Eine Klasse, die einen Controller, im Spring Model-View-Controller darstellt

7.1.6.3. Scoping

7.1.6.3.1. Singleton

Eine einzelne Instanz einer Bean, die in der gesamten Anwendung geteilt wird [1]. Diese Instanz wird in einem Cache aus Singleton Beans gespeichert. Jede zukünftige Anfrage und Referenz auf diese Bean gibt dieses Objekt aus dem Cache zurück. Der Singleton Scope ist der standard Scope für eine Bean. Keine spezielle Annotation ist notwendig [34].

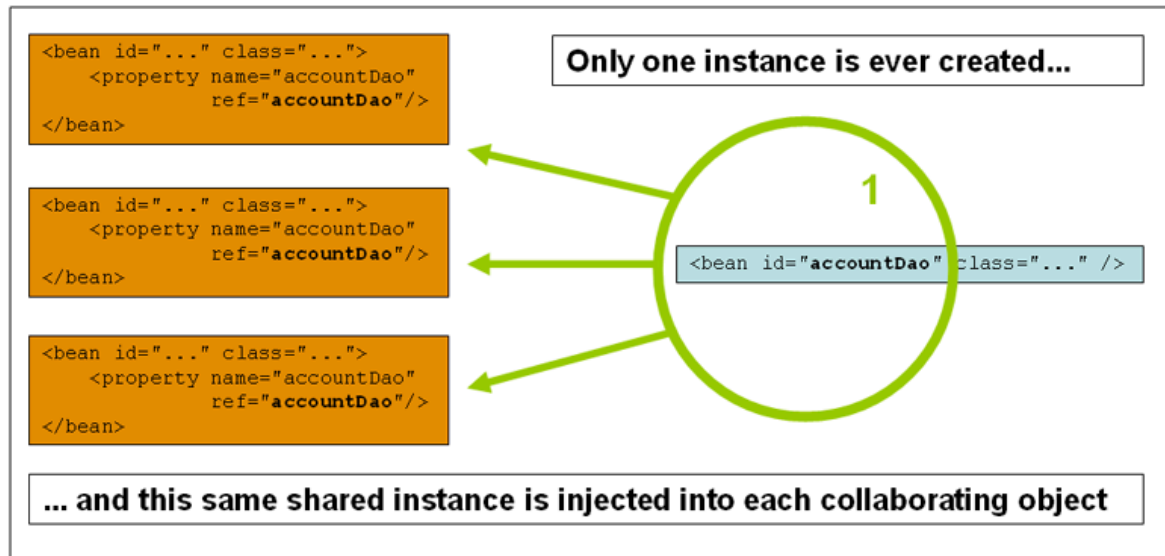


Figure 2: Funktionalität des Singleton Scopes

```
1 <bean
2   id="accountService"
3   class="com.something.DefaultAccountService"
4 />
```

XML

Einsatz von Singleton Beans

Richtlinie 5

Singleton Beans sollten für stateless Beans eingesetzt werden.

7.1.6.3.2. Prototype

Eine neue Instanz der Bean wird bei jeder Anfrage erstellt [1]. Diese Anfrage kann durch Injection in eine andere Bean oder durch eine Anfrage durch `getBean()` geschehen [34].

Spring verwaltet, anders als bei anderen Beans, nicht den kompletten Lebenszyklus einer Prototype Bean. Das Löschen einer Prototype Bean muss manuell durch den Client geschehen. Ein eigens definierter Bean Post-Processor kann genutzt werden, damit der Container Ressourcen, die von Prototype Beans gehalten werden, freigibt [34].

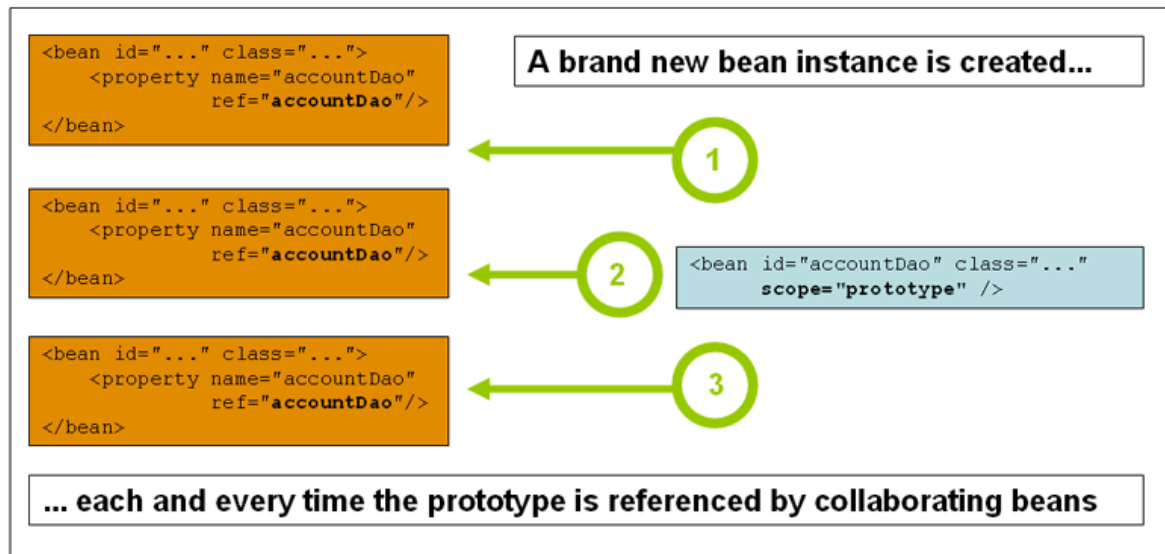


Figure 3: Funktionalität des Prototype Scopes

```

1 <bean
2   id="accountService"
3   class="com.something.DefaultAccountService"
4   scope="prototype"
5 />

```

XML

Einsatz von Prototype Beans

Richtlinie 6

Prototype Beans sollten für stateful Beans eingesetzt werden.

7.1.6.3.3. Request

Eine einzelne Instanz wird für jede HTTP Anfrage erstellt [1]. Die Erstellte Bean existiert nur so lange, wie die HTTP Anfrage bearbeitet wird. Andere Beans, vom gleichen Typ, die aber zu anderen HTTP Anfragen gehören, werden die Änderungen nicht sehen. Sobald die Anfrage abgearbeitet wurde, wird die Bean, die zu der Anfrage gehört, entfernt [34].

```

1 <bean
2   id="loginAction"
3   class="com.something.LoginAction"
4   scope="request"
5 />

```

XML

```

1 @RequestScope
2 @Component
3 class LoginAction {
4   // ...
5 }

```


Kotlin

7.1.6.3.4. Session


Eine einzelne Instanz wird für jede HTTP Session erstellt [1]. Die erstellte Bean wird praktisch auf die HTTP Session scoped. Der State der Bean kann so lange beliebig geändert werden, die die

Session aktiv ist. Andere Beans, vom gleichen Typ, die aber zu anderen HTTP Sessions gehören, werden die Änderungen nicht sehen. Wenn die HTTP Session beendet wird, wird auch die dazugehörige Bean entfernt [34].

```
1 <bean
2   id="userPreferences"
3   class="com.something.UserPreferences"
4   scope="session"
5 />
```



```
1 @SessionScope
2 @Component
3 class UserPreferences {
4   // ...
5 }
```




7.1.6.3.5. Application

Ähnlich wie beim Singleton Scope, wird hier eine Bean für die gesamte Web Anwendung erstellt. Diese Bean wird auf die ServletContext Ebene gescoped und als Attribut von ServletContext gespeichert. Folgende Unterschiede sind im Vergleich zu Singletons zu finden:


- Es existiert eine Bean pro ServletContext
- Es wird exposed als Attribut von ServletContext

[34]

```
1 <bean
2   id="appPreferences"
3   class="com.something.AppPreferences"
4   scope="application"
5 />
```



```
1 @ApplicationScope
2 @Component
3 class AppPreferences {
4   // ...
5 }
```



7.1.6.3.6. WebSocket

Der WebSocket Scope ist an den Lebenszyklus einer WebSocket gekoppelt [34].

Weitere Informationen: <https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html> [35]

TODO: WebSocket Scope Kapitel ausbauen.

7.1.7. Aspect Oriented Programming (AOP)

7.1.8. Struktur für ein Projekt

module		
	dtos	
		CreateEntityDto
		EditEntityDto
		GetEntityDto
	mapper	
		CreateEntityDtoMapper
		EditEntityDtoMapper
		GetEntityDtoMapper
	Controller	
	Entity	
	Repository	
	Service	
Application		

- module
- ▶ dtos
- ▶ mapper
- ▶ Controller, Entity, Repository, Service
- Application

Disclaimer

Hinweis 7

Hier handelt sich um Richtlinien. Die wirkliche Situation kann von diesen abweichen, sollte sich die Anpassung besser für das Erreichen der Ziele eignen.

Jeder Controller sollte idealerweise einen Mapping für jede HTTP Operation enthalten: GET, POST, PUT, DELETE

Es sollten keine weiteren Pfade in den Mappings vorhanden sein, außer IDs in PUT und DELETE

Parameter sollten als DTO im Request Body übergeben werden. Nur bei GET müssen es einzelne Request Parameter sein.

Die Funktionen im Controller sollten keine eigene Logik enthalten. Sie sollen nur eine Funktion im zugehörigen Service aufrufen.

Mappings im Controller sollten durch die `@ReponseStatus` Annotation einen Http Status zurückgeben.

Die Rückgabewerte von Mappings sollte fast immer ein Dto sein.

Sowohl PUT, POST als auch UPDATE sollten die betroffenen oder erlangten Entities als DTO zurückgeben.

7.1.9. OpenAPI UI

```
1 // build.gradle.kts
2 dependencies {
3     implementation("org.springdoc:springdoc-openapi-starter-webmvc-ui:2.8.4")
4 }
```

Kotlin

URL: <http://localhost:8080/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config>

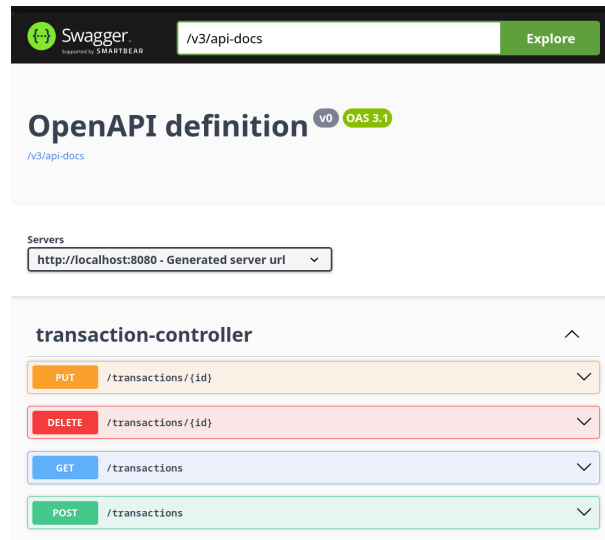


Figure 4: Swagger UI zum Darstellen und Testen der REST Endpunkte

7.1.10. Authentication

7.2. Jakarta EE

7.3. Lombok

7.4. Buildtools

7.4.1. Gradle

7.4.2. Maven

7.5. Documentation

8. Frontend

8.1. Frameworks

8.1.1. React

8.1.2. Svelte

8.1.3. VueJS

8.1.4. Angular

8.2. Web Components

8.3. CSS

9. Dev Ops

9.1. Docker

9.1.1. Dockerfile

9.1.2. Image

9.1.3. Container

9.1.4. Volumes

9.1.5. Networking

9.1.6. Compose

9.2. Podman

9.3. Nginx

10. Debugging

- Lesen von Dokumentation und Stack Traces
- Intelligentes Nutzen von Google
- Logging: Print von unterschiedlichen Daten um Funktionalität sicherzustellen
- Debuggers in IDEs: Pausieren der Ausführung, Durchschreiten von Codestücken
- Reproduzieren des Problems in einer minimalen Umgebung, die im schlimmsten Fall auch mit anderen geteilt werden kann
- Test Driven Development als Strategie Probleme früher zu finden
- Nutzen von Sprachen mit expliziten Types (z.B.: TypeScript statt JavaScript)

10.1. Backend Debugging

10.2. Frontend Debugging

10.2.1. Browser DevTools

[Firefox DevTools User Docs](#)

10.2.1.1. HTML/CSS Inspection

- Live HTML und CSS anpassen um Änderungen zu sehen
- Visuelle Darstellungen von Parametern wie Padding, Margins usw.

10.2.1.2. JavaScript Console

- Log output lesen
- Interagieren mit der Website durch JavaScript

10.2.1.3. JavaScript Debugger

- Durchschreiten von JavaScript auf der Website

10.2.1.3.1. Source Map

- Wichtig für minified JavaScript (zum Beispiel erstellt durch Frameworks)
- Originaler Code bleibt erhalten und kann für das Debugging genutzt werden
- Source Map muss generiert werden
- In der transformierten Datei muss mit einem Kommentar auf die Source Map verwiesen werden

```
1 //# sourceMappingURL=http://example.com/path/sourcemap.map
```

JS JavaScript

Generierung:

- **Svelte** generiert SourceMaps automatisch. Früher wurde eine Compiler Option benötigt.

```
1 // svelte.config.js
```

JS JavaScript

```
2 const config = {
3   compilerOptions: {
4     enableSourceMap: true
5   },
6 }
```

React und **Vue** generieren SourceMaps automatisch. Sie können durch Compiler Option explizit aktiviert oder deaktiviert werden

```
1 /* tsconfig.node.json */
```

☒ JSON

```
2 {
3   "compilerOptions": {
4     "sourceMap": true
5   }
6 }
```

Angular generiert SourceMaps automatisch. Es wird durch eine Compiler Option aktiviert.

```
1  /* angular.json */
```

☒ JSON

```
2  "projects": {
3    "vite-project": {
4      "architect": {
5        "build": {
```



```
6      "configuration": {
7      "development": {
8      "sourceMap": true
9      }
10     }
11   }
12 }
13 }
14 }
```

10.2.1.4. Network Operations

- Auflistung aller Netzwerk Anfragen der Website
- Genaue Untersuchung aller Daten, die zu den einzelnen Anfragen gehören

10.2.2. IDE Debugging

10.2.3. Extensions

11. Seminare

11.1. Installieren der wichtigen Software

11.1.1. IntelliJ

IntelliJ ist eine IDE von JetBrains, welche mit einer langen Liste an Features daherkommt und oft in der Industrie als Entwicklungsumgebung genutzt wird. Es bietet sehr viele Funktionen, die die Entwicklung mit Spring und Kotlin vereinfachen. Als Student ist es möglich die Ultimate Version kostenlos zu erhalten.

Die Installation empfiehlt sich über die Toolbox App. Ein Account wird dafür bei JetBrains benötigt. Wenn dieser als Studentenaccount verifiziert ist, kann so auch die Ultimate Version heruntergeladen werden.

<https://www.jetbrains.com/help/idea/installation-guide.html#toolbox>

11.1.2. Docker

[Installation von Docker Desktop auf Windows](#)

[Installation von Docker auf Linux](#)

[Docker Desktop Installation auf Mac](#)

11.1.3. Podman

[Installierung von Podman auf allen Plattformen](#)

11.1.4. nodejs

[NVM](#)

[NVM für Windows](#)

Quellenverzeichnis

- [1] geekforgs9hp, "Spring Boot - Dependency Injection and Spring Beans." [Online]. Available: <https://www.geeksforgeeks.org/advance-java/spring-boot-dependency-injection-and-spring-beans/>
- [2] baeldung, "A Comparison Between Spring and Spring Boot." [Online]. Available: <https://www.baeldung.com/spring-vs-spring-boot>
- [3] "Spring Framework Overview." [Online]. Available: <https://docs.spring.io/spring-framework/reference/overview.html>
- [4] "History of Spring and the Spring Framework." [Online]. Available: <https://docs.spring.io/spring-framework/reference/overview.html#overview-history>
- [5] "Spring Boot." [Online]. Available: <https://spring.io/projects/spring-boot>
- [6] "State of CSS 2025." [Online]. Available: <https://2025.stateofcss.com/en-US/other-tools/>
- [7] "What is REST?." [Online]. Available: <https://www.codecademy.com/article/what-is-rest>
- [8] "What is CRUD? Explained." [Online]. Available: <https://www.codecademy.com/article/what-is-crud-explained>
- [9] "Method Definitions - GET." [Online]. Available: <https://httpwg.org/specs/rfc9110.html#GET>
- [10] "Method Definitions - POST." [Online]. Available: <https://httpwg.org/specs/rfc9110.html#POST>
- [11] "Method Definitions - PUT." [Online]. Available: <https://httpwg.org/specs/rfc9110.html#PUT>
- [12] "Method Definitions - DELETE." [Online]. Available: <https://httpwg.org/specs/rfc9110.html#DELETE>
- [13] "Idempotent Methods." [Online]. Available: <https://httpwg.org/specs/rfc9110.html#idempotent.methods>
- [14] "Status Codes." [Online]. Available: <https://httpwg.org/specs/rfc9110.html#status.codes>
- [15] "MVC." [Online]. Available: <https://developer.mozilla.org/en-US/docs/Glossary/MVC>
- [16] "Object-relational mapping." [Online]. Available: https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping
- [17] "Was ist Object-Relational Mapping (ORM)." [Online]. Available: <https://www.it-schulungen.com/wir-ueber-uns/wissensblog/was-ist-object-relational-mapping-orm.html>
- [18] Jim Schiel, "The Anatomy of a User Story." [Online]. Available: <https://resources.scrumalliance.org/Article/anatomy-user-story>
- [19] "Mapping Requests." [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html>
- [20] "URI Patterns." [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html#mvc-ann-requestmapping-uri-templates>
- [21] "Error Handling for REST with Spring in Kotlin." [Online]. Available: <https://www.baeldung.com/kotlin/spring-rest-error-handling>
- [22] "Controller Advice." [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-advice.html>

- [23] “When should we use @Transactional annotation?.” [Online]. Available: <https://stackoverflow.com/questions/78132448/when-should-we-use-transactional-annotation>
- [24] “Transactions 101: when and how to use them in the Spring Framework.” [Online]. Available: <https://www.twoday.lt/blog/transactions-101-when-and-how-to-use-them-in-the-spring-framework>
- [25] “Defining JPA Entities.” [Online]. Available: <https://www.baeldung.com/jpa-entities>
- [26] “Hibernate One to Many.” [Online]. Available: <https://www.baeldung.com/hibernate-one-to-many>
- [27] “Spring Boot One To Many Relationship.” [Online]. Available: <https://medium.com/@hk09/spring-boot-one-to-many-relationship-e617183b7861>
- [28] “One-to-One Relationship in JPA.” [Online]. Available: <https://www.baeldung.com/jpa-one-to-one>
- [29] “Many-To-Many Relationship in JPA.” [Online]. Available: <https://www.baeldung.com/jpa-many-to-many>
- [30] “Container Overview.” [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/basics.html>
- [31] “Dependency Injection.” [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html>
- [32] “Method Injection.” [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-method-injection.html>
- [33] Nguyen Nam Thai, “What Is a Spring Bean.” [Online]. Available: <https://www.baeldung.com/spring-bean>
- [34] “Bean Scopes.” [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html#beans-factory-scopes-singleton>
- [35] “WebSocket Scope.” [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html>