



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. Design Patterns	16
5.1. Singleton	16
5.1.1. Wann sollte das Singleton-Pattern verwendet werden?	16
5.1.2. Struktur	16
5.1.3. Vorteile	16
5.2. Prototype	17
5.2.1. Struktur	17
5.2.2. Vorteile	17
5.2.3. Nachteile	18
6. Inversion of Control & Dependency Injection	19
6.1. Inversion of Control	19

6.2.	Dependency Injection	20
7.	Backend	21
7.1.	Spring	21
7.1.1.	Komponenten einer Spring Anwendung	21
7.1.1.1.	Controller	21
7.1.1.1.1.	Mappings	21
7.1.1.1.2.	URI Patterns	21
7.1.1.1.3.	Error Handling	22
7.1.1.2.	Service	24
7.1.1.2.1.	Transactional Methoden	24
7.1.1.3.	Entity	26
7.1.1.3.1.	@Entity Annotation	26
7.1.1.3.2.	@Table Annotation	26
7.1.1.3.3.	@Id Annotation	26
7.1.1.3.4.	@Column Annotation	27
7.1.1.3.5.	Referenzen auf andere Tables	27
7.1.1.3.5.1.	@OneToMany und @ManyToOne	27
7.1.1.3.5.2.	@OneToOne	28
7.1.1.3.5.3.	@ManyToMany	29
7.1.1.4.	Repository	31
7.1.1.4.1.	Automatische Queries durch Methoden	31
7.1.1.4.2.	Manuelle Queries	31
7.1.1.5.	DTO	32
7.1.1.6.	Mapper	33
7.1.1.7.	Application Konfiguration	34
7.1.2.	Spring Data	35
7.1.2.1.	Spring Data JDBC	35
7.1.2.2.	Spring Data JPA	36
7.1.2.3.	Spring Data R2DBC	37
7.1.2.4.	Spring Data REST	38
7.1.3.	IoC Container	39
7.1.4.	Dependency Injection	40
7.1.4.1.	Constructor Injection - Type 3 IoC	40
7.1.4.2.	Setter Injection - Type 2 IoC	40
7.1.5.	Method Injection	41
7.1.6.	Beans	42
7.1.6.1.	Spring Inversion of Control (IoC) Container	42
7.1.6.2.	Annotationen für Beans [1]	43
7.1.6.3.	Scoping	43
7.1.6.3.1.	Singleton	43
7.1.6.3.2.	Prototype	44
7.1.6.3.3.	Request	45
7.1.6.3.4.	Session	45
7.1.6.3.5.	Application	46
7.1.6.3.6.	WebSocket	46
7.1.7.	Aspect Oriented Programming (AOP)	47
7.1.8.	Struktur für ein Projekt	48
7.1.9.	OpenAPI UI	49
7.1.10.	Authentication	50

7.2.	Jakarta EE	51
7.3.	Lombok	52
7.4.	Buildtools	53
7.4.1.	Gradle	53
7.4.2.	Maven	54
7.5.	Documentation	56
8.	Frontend	57
8.1.	Frameworks	57
8.1.1.	React	57
8.1.2.	Svelte	58
8.1.3.	VueJS	59
8.1.4.	Angular	60
8.2.	Web Components	61
8.3.	CSS	62
9.	Dev Ops	63
9.1.	Docker	63
9.1.1.	Dockerfile	63
9.1.2.	Image	63
9.1.3.	Container	63
9.1.4.	Volumes	63
9.1.5.	Networking	63
9.1.6.	Compose	63
9.2.	Podman	64
9.3.	Nginx	65
10.	Werkzeuge	66
10.1.	IntelliJ	66
10.1.1.	Persistence View	66
10.1.2.	Entity Relationship Diagram	66
11.	Debugging	67
11.1.	Backend Debugging	67
11.2.	Frontend Debugging	68
11.2.1.	Browser DevTools	68
11.2.1.1.	HTML/CSS Inspection	68
11.2.1.2.	JavaScript Console	68
11.2.1.3.	JavaScript Debugger	68
11.2.1.3.1.	Source Map	68
11.2.1.4.	Network Operations	69
11.2.2.	IDE Debugging	70
11.2.3.	Extensions	71
12.	Seminare	72
12.1.	Installieren der wichtigen Software	72
12.1.1.	IntelliJ	72
12.1.2.	Docker	72
12.1.3.	Podman	72
12.1.4.	nodejs	72
	Quellenverzeichnis	74

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

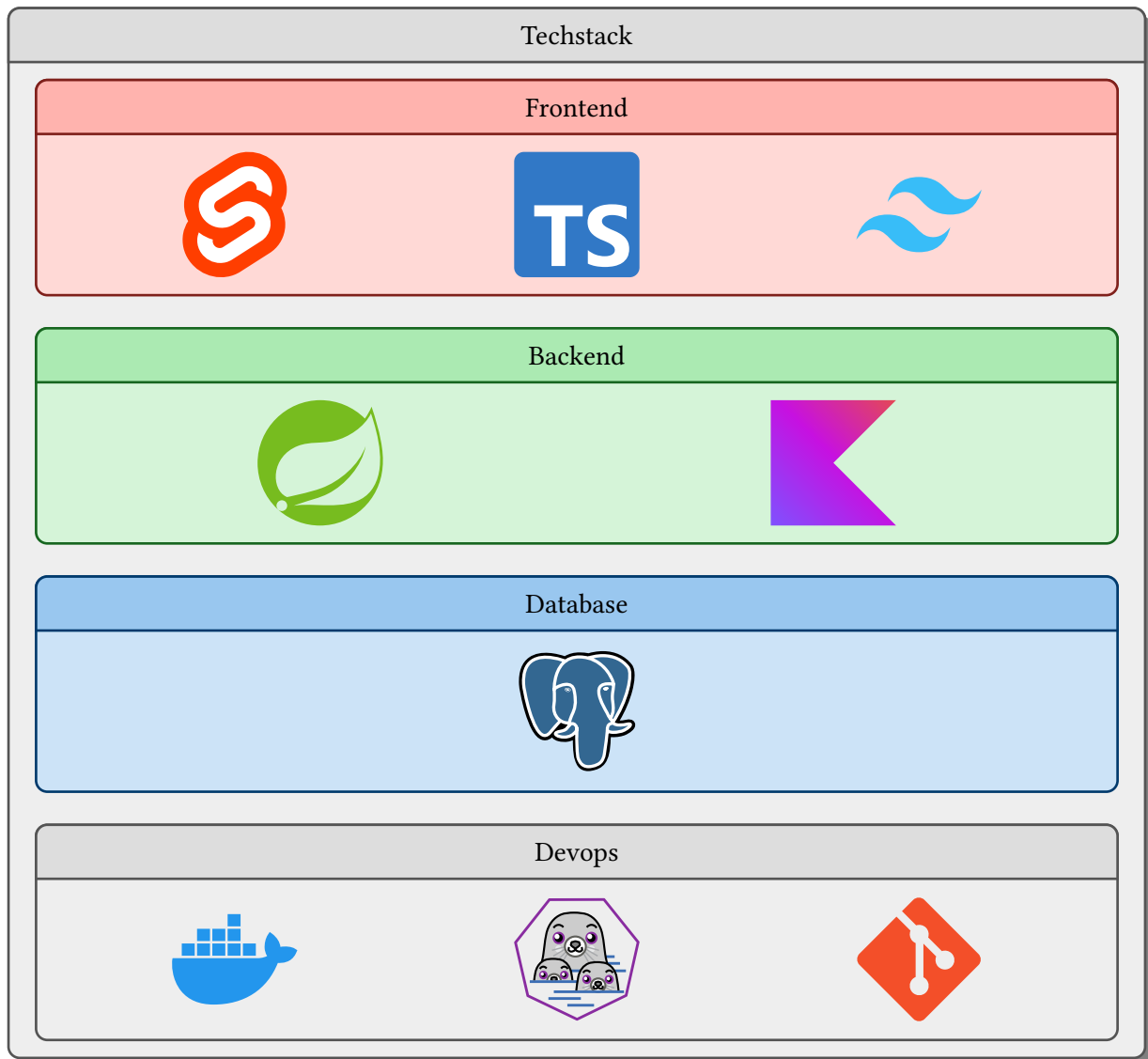
2.3. Präsentation bzw. Verteidigung

- 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

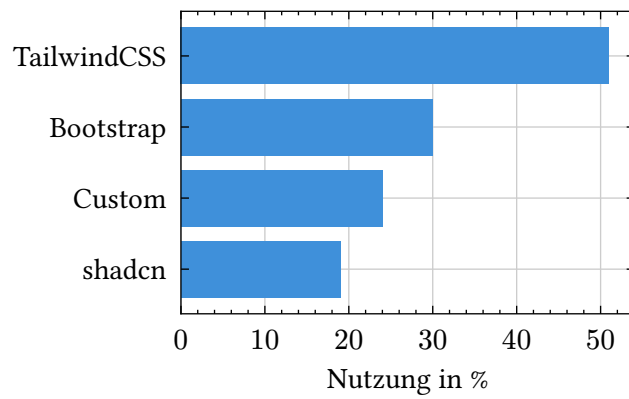
Springboot ermöglicht die Erstellung von Spring Anwendungen, die mit minimaler Konfiguration lauffähig sind. Es ist dabei eine Erweiterung des Spring Frameworks. [2] Dazu werden Webserver, wie zum Beispiel Tomcat, direkt mitgeliefert. Um den Start der Entwicklung zu vereinfachen werden außerdem *Starter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

5.1. Singleton

Das Ziel eines Singletons ist es, dass eine Klasse nur eine Instanz besitzen kann. Diese Instanz soll immer dann zur Verfügung gestellt werden, wenn eine andere Klasse eine Instanz der Singleton-Klasse benötigt.

Der Zugriff auf eine einzelne Instanz kann theoretisch über eine globale Variable gelöst werden. Die würde uns aber nicht davon abhalten, weitere Instanzen zu erstellen. Deshalb sollte die Singleton-Klasse selbst dafür sorgen, dass sie ihre einzige Instanz managt und dabei neue Versuche des Instanzierens blockiert. Neben diesem Blockieren sollte die Klasse auch Zugang zu ihrer einzigen Instanz zur Verfügung stellen. Dabei sollte der Zugang zu der Singleton-Instanz nur durch die Instance-Methode in der Singleton-Klasse möglich sein.

5.1.1. Wann sollte das Singleton-Pattern verwendet werden?

Es sollte nur eine Instanz einer Klasse existieren und es sollte von einem gut definierten Punkt erreichbar sein.

Wenn eine einzelne Instanz einer Klasse durch Subklassen erweiterbar sein sollte. Dabei sollten Clients in der Lage sein, die erweiterte Instanz nutzen zu können, ohne den Code zu modifizieren.

5.1.2. Struktur

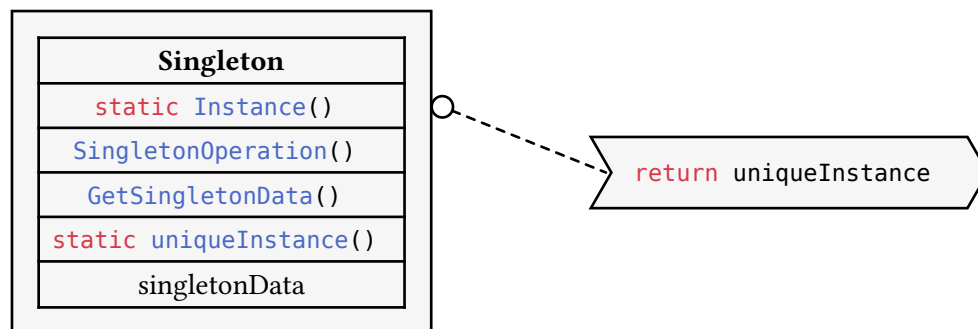


Figure 1: Struktur des Singleton Patterns

5.1.3. Vorteile

1. **Kontrollierter Zugriff auf eine einzige Instanz:** Da die Singleton-Klasse ihre eigene Instanz verwaltet, kann sie genau festlegen, wie auf sie zugegriffen wird und wer auf sie zugreifen kann.
2. **Reduzierter Namespace:** Das Singleton-Pattern stellt eine Verbesserung zu globalen Variablen dar. Der Namespace wird dabei nicht mit globalen Variablen unnötig belegt.
3. **Erlaubt das Anpassen von Operationen und Repräsentation:** Es erlaubt einfaches Wechseln von erlaubten Klasseninstanzen. Die gleiche Logik, die nur eine Instanz erlaubt, kann einfach auf eine beliebige festgesetzte Anzahl erweitert werden. Nur die Zugriffsfunktion muss sich dabei ändern.
4. **Höhere Flexibilität als Klassenoperationen:** Statische Klassenoperationen können die gleiche Funktionalität wie ein Singleton enthalten. Allerdings erlaubt diese Methodik nur schwer das Erstellen von mehreren Instanzen. Außerdem sind diese Operationen nicht `virtual`, also können sie nicht überschrieben werden.

[18]

5.2. Prototype

Das Ziel von Prototypen ist es, basierend auf einer existierenden Instanz (dem Prototypen) neue Kopien zu erstellen.

Es sollte genutzt werden, wenn ein System unabhängig davon sein sollte, wie seine Produkte erstellt, zusammengebaut und repräsentiert werden, und:

- Wenn Klassen instanziiert werden sollen, die zur Runtime spezifiziert wurden, zum Beispiel durch dynamic loading.
- Um zu vermeiden, dass man eine Hierarchie an Factories baut, die die Klassenhierarchie kopiert.
- Wenn Klassen nur eine begrenzte Anzahl an State besitzen können. Es könnte da von Vorteil sein, diese Varianten als Prototypen zu definieren.

5.2.1. Struktur

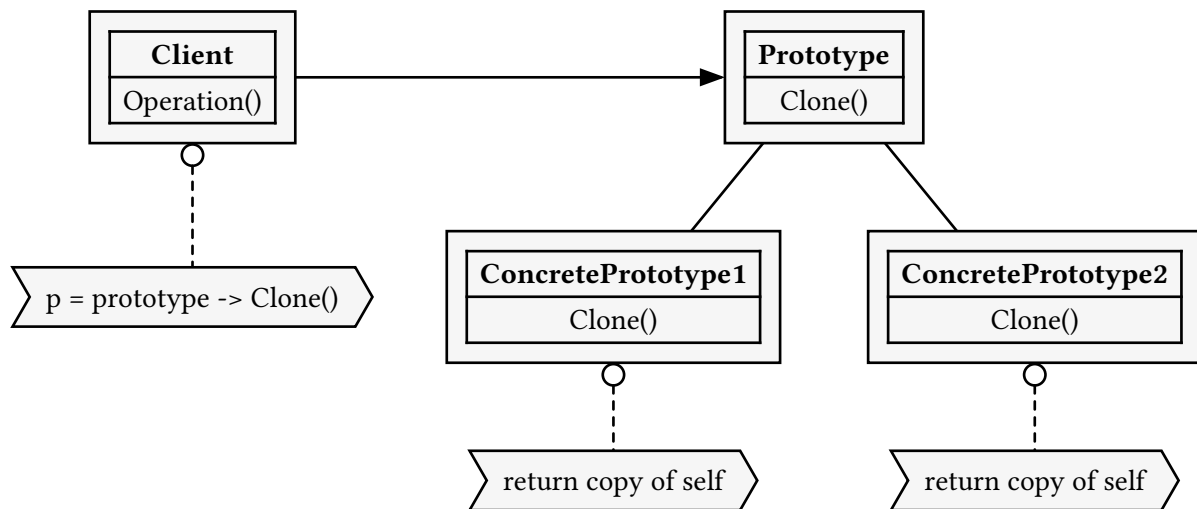


Figure 2: Struktur des Prototype-Patterns

- **Prototype:** Deklariert ein Interface, damit es kopiert werden kann.
- **ConcretePrototype:** Implementiert eine Operation, damit es sich selbst kopieren kann.
- **Client:** Erstellt ein neues Objekt, indem der Prototype gefragt wird, ob er eine neue Kopie von sich selbst erstellen kann.

5.2.2. Vorteile

1. **Hinzufügen und Entfernen von Produkten zur Laufzeit:** Durch das Registrieren einer Prototyp-Instanz beim Client kann eine Produktklasse in das System integriert werden. Das kann, anders als bei anderen Patterns, zur Laufzeit geschehen.
2. **Spezifizierung von neuen Objekten durch das Ändern von Werten:** Dynamische Systeme erlauben das Definieren von neuem Verhalten durch Objektkomposition, zum Beispiel durch das Anpassen von Variablen. Neue Arten von Objekten werden somit durch das Instanzieren von existierenden Klassen als Prototypen von Client-Objekten erzeugt. Der Client kann somit neue Möglichkeiten erlangen, indem er Aufgaben an Prototypen auslagert.
3. **Spezifizierung von neuen Objekten durch das Ändern von Struktur:** Bauen von Objekten durch Teile und Unterteile. Komplexe Strukturen können so vom Nutzer erzeugt werden. Das Prototype-Pattern unterstützt solche Ansätze durch Deep Copy, die im Parent-Teil implementiert sein muss.
4. **Reduziertes Subclassing:** Anders als Factories erlaubt das Prototype-Pattern das Erstellen von neuen Objekten ohne spezifische Factory-Methoden. Dadurch wird keine Creator Klassenhierarchie benötigt.

5. **Dynamische Konfiguration einer Anwendung durch Klassen:** Wenn die Umgebung es erlaubt, können dynamisch geladene Klassen durch das Prototype-Pattern instanziiert werden. Der Constructor der Klasse kann nicht in dieser Umgebung genutzt werden. Deshalb wird eine Instanz automatisch beim Laden erstellt und mit einem Prototype-Manager registriert. Über den Prototype-Manager können diese Klassen dann abgefragt werden.

5.2.3. Nachteile

Jede Subklasse eines Prototypen muss die Clone-Operation besitzen. Das kann in manchen Fällen schwierig werden. Zum Beispiel, wenn die Klasse bereits existiert oder wenn die Zielklasse andere Objekte enthält, die sich nicht zum Kopieren eignen oder Zirkelverweise besitzen.

[19]

6. Inversion of Control & Dependency Injection

Mit der steigenden Komplexität von J2EE und dem Aufkommen von unterschiedlichen Problemen wurde die Entwicklung von IoC Containern vorangetrieben. Diese Container sollen eine Anwendung aus unterschiedlichen Komponenten aus unterschiedlichen Projekten in eine funktionierende Anwendung bauen. Viele der Produkte, die dabei entstanden sind, setzen im Kern auf Dependency Injection. [20]

6.1. Inversion of Control

Inversion of Control ist ein Phänomen, welches oft der definierende Faktor von Frameworks ist. Vor allem, wenn man Funktionalitäten erweitern oder neu hinzufügen möchte. Das Ziel ist es dabei, dem Framework die Kontrolle über alles zu geben, dass ich als Nutzer geschrieben habe. Dabei wird folgendem Prinzip aus Hollywood gefolgt: “Don’t call us, we’ll call you” [21]

Funktionen, die vom User definiert werden, um ein Framework anzupassen, werden vom Framework aufgerufen und nicht vom User. Das Framework spielt dabei die Rolle eines Hauptprogramms, welches die Aktivitäten der Anwendung koordiniert und sequenziert. Das Framework kann damit als ein erweiterbares Skelett für eine Anwendung dienen. Generische Algorithmen in Frameworks können so durch den User auf spezifische Anwendungsfälle zugeschnitten werden. [22]

Inversion of Control ist ein essentieller Bestandteil eines Frameworks, welches es von einer Library differenziert. Eine Library stellt aufrufbare Funktionen bereit, die vom User aufgerufen werden können. Ein Framework hingegen enthält ein abstrakteres Design, welches mit mehr eingebauten Funktionalitäten und Verhalten daherkommt. Damit man es benutzen kann, muss selbst definiertes Verhalten an unterschiedlichen Stellen in das Framework injiziert werden. Das kann durch Subklassen oder selbst definierte Klassen geschehen. Das Framework ruft dann den vom User erstellen Code an an diesen Punkten auf.

Dependency Injection ist dabei eine von vielen Formen von Inversion of Control. In Spring kommt sie mit dem **Inversion of Control** Container zum Einsatz. Neben IoC Containern kommen auch EJBs (Jakarta Enterprise Beans) oft zum Einsatz. [21]

Beispiel zu IoC:

```
1 puts 'What is your name?'
2 name = gets
3 process_name(name)
4 puts 'What is your quest?'
5 quest = gets
6 process_quest(quest)
```

 Ruby

Der Code hier ist in Kontrolle, wann Aufrufe stattfinden und wann sie verarbeitet werden.

```
1 require 'tk'
2 root = TkRoot.new()
3 name_label = TkLabel.new() {text "What is Your Name?"}
4 name_label.pack
5 name = TkEntry.new(root).pack
6 name.bind("FocusOut") {process_name(name)}
7 quest_label = TkLabel.new() {text "What is Your Quest?"}
8 quest_label.pack
9 quest = TkEntry.new(root).pack
```

 Ruby

```
10 quest.bind("FocusOut") {process_quest(quest)}
11 Tk.mainloop()
```

Das Windowing System hat nun die Kontrolle darüber, wann Aufrufe stattfinden.

6.2. Dependency Injection

Die Grundidee hier ist, dass es ein Assembler Objekt gibt, welches Felder in Klassen, nach definierten Anforderungen bevölkert. Das kann zum Beispiel die Implementation von einem Interface sein.

Es gibt dabei drei Möglichkeiten der Dependency Injection:

- Constructor Injection, Type 3 IoC (Section 7.1.4.1)
- Setter Injection, Type 2 IoC (Section 7.1.4.2)
- Interface Injection, Type 1 IoC

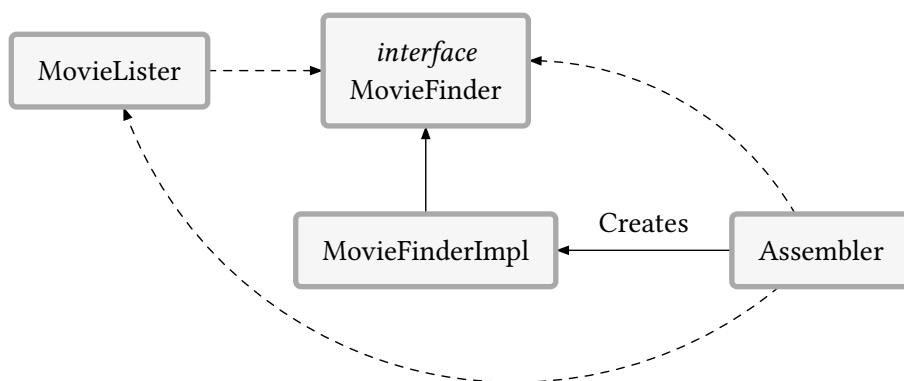


Figure 3: Abhängigkeiten für Dependency Injection

In Figure 3 benötigt die MovieLISTER Klasse eine Implementation von dem MovieFinder Interface. Diese Implementation wird durch den Assembler erstellt, um die Abhängigkeit in MovieLISTER zu erfüllen.

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.

[23]

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")
```

[24]

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 }

```


[25]

@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. [26]

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.

[27]

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**

[28]

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. [29]

```
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. [29]

```
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.

[29]

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.

[29]

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

[30], [31]

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. [32]

```

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

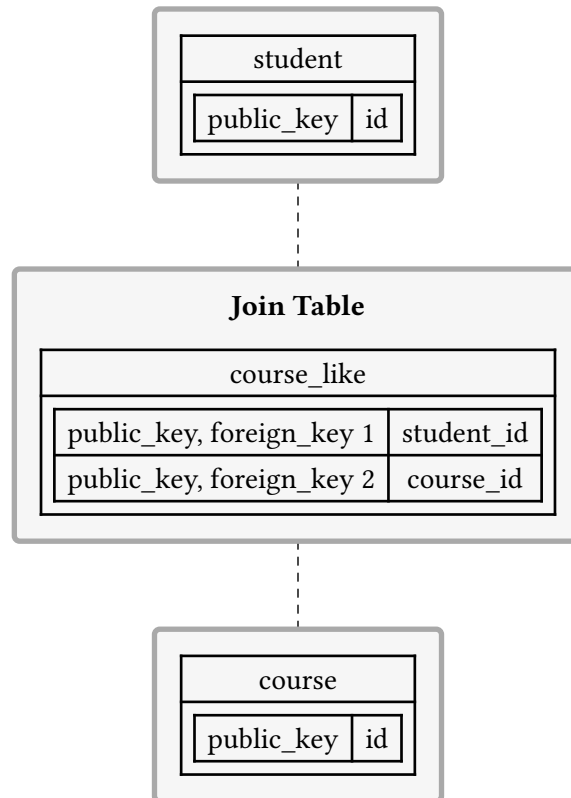


Figure 4: Aufbau einer Many to Many Relation mit einem Join Table

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 erliten 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. [33]

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

Kotlin

```
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 EntityRepository: JpaRepository<Entity, Long> {
3
4 }
```

Kotlin

Das Repository ist der Hauptinteraktionspunkt mit der Datenbank. Es wird als Interface erstellt, welches mit der Annotation `@Repository` versehen ist. Die Art des Repositories wird durch Vererbung des gewählten Repository Typen festgelegt. Unterschiedliche Repositories bringen dabei unterschiedliche Features. Die meisten sind dabei Erweiterungen von Repositories, die weniger Funktionen enthalten. Neben dem Repository Typen muss auch angegeben werden, welche Entity hier verwaltet wird und welche Klasse die Id der Entity besitzt. Für einen User mit einer id vom Typ Long würde ein `JpaRepository` wie folgt aussehen:

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

Kotlin

Es stellt Funktionen bereit um auf der Datenbank einfache CRUD Operationen auszuführen. Dazu werden Möglichkeiten bereitgestellt, eigene SQL Queries zu verfassen oder sie automatisch generieren zu lassen:

- Definierung einer neuen Methode im Interface
- SQL Query bereitstellen über die `@Query` Annotation
- Nutzung von Querydsl
- Eigene Queries erstellen mit JPA Named Queries

Die Nutzung von Querydsl wird vor allem bei vielen kleinen Anfragen empfohlen, da diese reduziert werden können, auf eine kleinere Anzahl an Blöcken. JPA Names Queries werden nicht empfohlen, da sie das Schreiben von XML voraussetzen.

7.1.1.4.1. Automatische Queries durch Methoden

Spring Data analysiert in jedem Repository alle vorhandenen Methoden und versucht aus den Methodennamen eine SQL Query zu generieren. Es folgt ein Beispiel, in einem Repository, welches Instanzen von Usern enthält, die ein Feld mit einem Namen besitzen.

```
1 @Repository
2 interface UserInterface : JpaRepository<User, Long> {
3     fun findUserByName(name: String): MutableList<User>
4 }
```

Kotlin

7.1.1.4.2. Manuelle Queries

Mit der `@Query` Annotation kann eine eigene Query manuell erstellt werden.

```
1 @Query("SELECT u FROM User u WHERE LOWER(u.name) = LOWER(:name)")
2 fun retrieveUserByName(
3     @Param("name") name: String
4 ): User
```

Kotlin

[34]

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 [35].

7.1.4. Dependency Injection

Für genauere Informationen zu Dependency Injection wird Kapitel (Section 6) empfohlen.

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 [36].

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 - Type 3 IoC

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 - Type 2 IoC

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](#) [37]

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. [38]

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. [38]

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 [39].

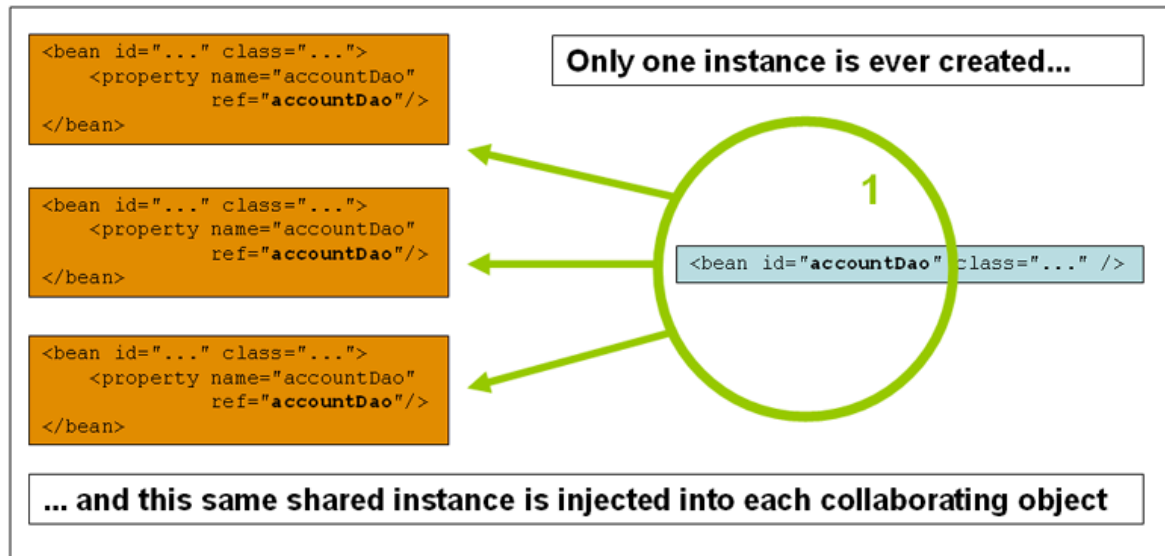


Figure 5: Funktionalität des Singleton Scopes

```

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 [39].

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 [39].

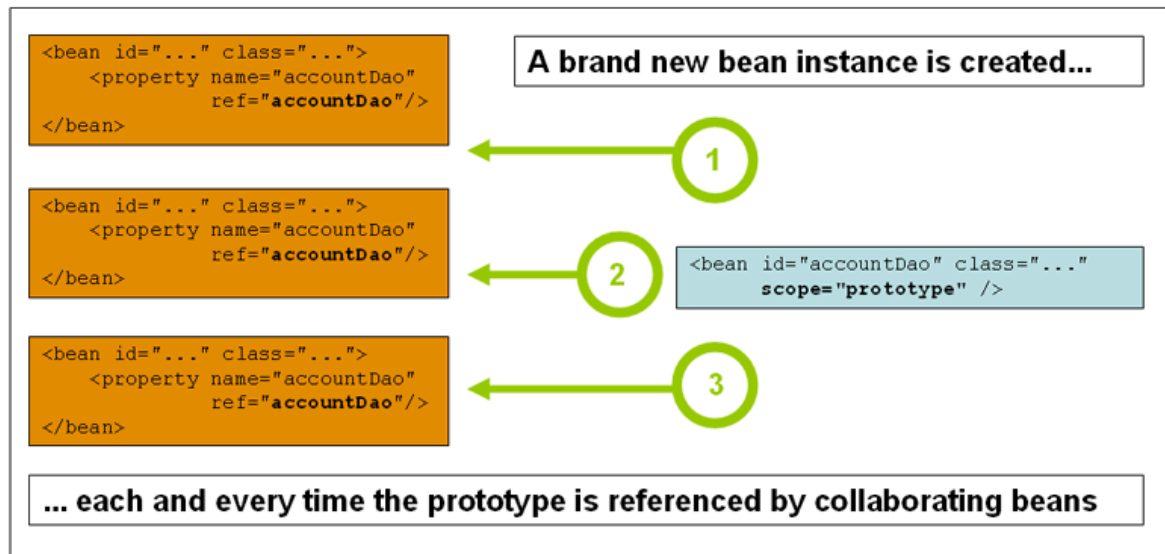


Figure 6: Funktionalität des Prototype Scopes

```

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 [39].

```

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 [39].

```
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

[39]

```
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 [39].

Weitere Informationen: <https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html> [40]

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>

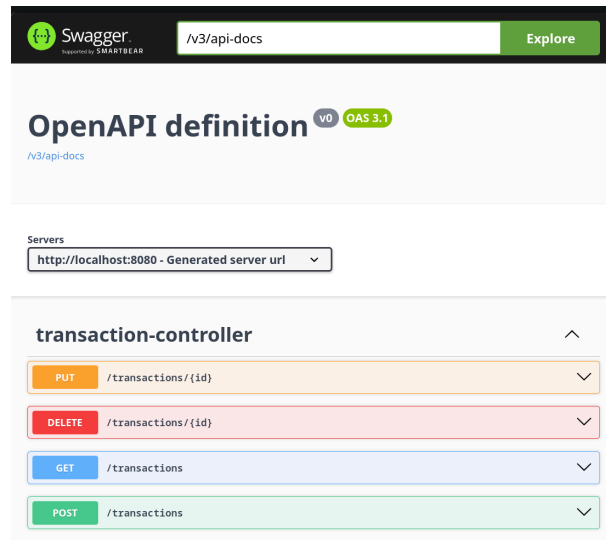


Figure 7: Swagger UI zum Darstellen und Testen der REST Endpunkte

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. Werkzeuge

10.1. IntelliJ

10.1.1. Persistence View

Jede Entity hat ein Datenbank Icon neben ihrer Klassendefinition. Über einen Klick auf dieses Icon wird ein Dropdown geöffnet, der unterschiedliche Optionen enthält. Um den Persistence View zu öffnen, klickt man auf **Select in Persistence View**

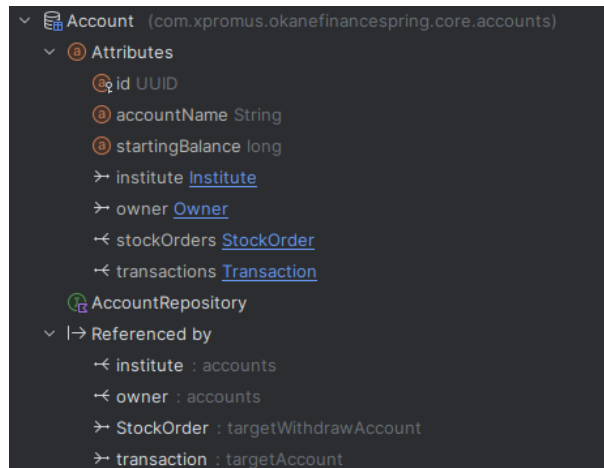


Figure 8: Ansicht des Persistence View mit Relationen zu anderen Entities

10.1.2. Entity Relationship Diagram

Jede Entity hat ein Datenbank Icon neben ihrer Klassendefinition. Über einen Klick auf dieses Icon wird ein Dropdown geöffnet, der unterschiedliche Optionen enthält. Um das Show Entity Relationship Diagram zu öffnen, klickt man auf **Show Entity Relationship Diagram**

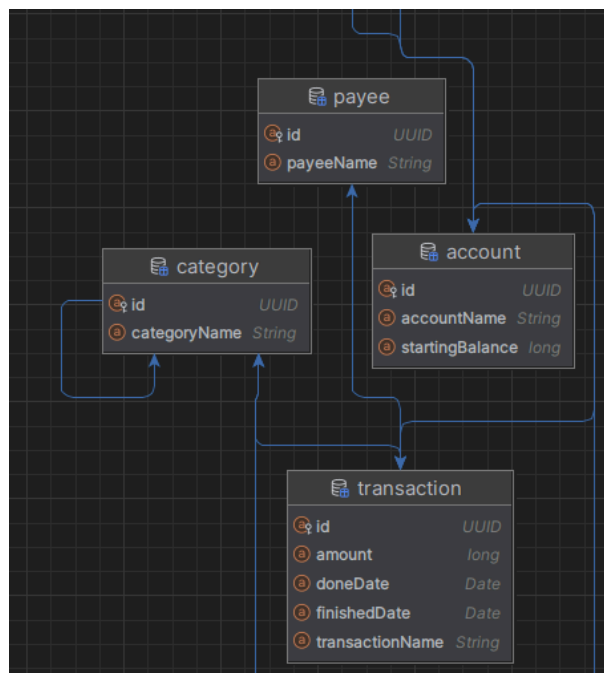


Figure 9: Das Entity Relationship Diagram zu einer Finanzanwendung

11. 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)

11.1. Backend Debugging

11.2. Frontend Debugging

11.2.1. Browser DevTools

[Firefox DevTools User Docs](#)

11.2.1.1. HTML/CSS Inspection

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

11.2.1.2. JavaScript Console

- Log output lesen
- Interagieren mit der Website durch JavaScript

11.2.1.3. JavaScript Debugger

- Durchschreiten von JavaScript auf der Website

11.2.1.3.1. 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

```
1 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299 1300 1301 1302 1303 1304 1305 1306 1307 1308 1309 1310 1311 1312 1313 1314 1315 1316 1317 1318 1319 1320 1321 1322 1323 1324 1325 1326 1327 1328 1329 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339 1340 1341 1342 1343 1344 1345 1346 1347 1348 1349 1350 1351 1352 1353 1354 1355 1356 1357 1358 1359 1360 1361 1362 1363 1364 1365 1366 1367 1368 1369 1370 1371 1372 1373 1374 1375 1376 1377 1378 1379 1380 1381 1382 1383 1384 1385 1386 1387 1388 1389 1390 1391 1392 1393 1394 1395 1396 1397 1398 1399 1400 1401 1402 1403 1404 1405 1406 1407 1408 1409 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423 1424 1425 1426 1427 1428 1429 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439 1440 1441 1442 1443 1444 1445 1446 1447 1448 1449 1450 1451 1452 1453 1454 1455 1456 1457 1458 1459 1460 1461 1462 1463 1464 1465 1466 1467 1468 1469 1470 1471 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481 1482 1483 1484 1485 1486 1487 1488 1489 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501 1502 1503 1504 1505 1506 1507 1508 1509 1510 1511 1512 1513 1514 1515 1516 1517 1518 1519 1520 1521 1522 1523 1524 1525 1526 1527 1528 1529 1530 1531 1532 1533 1534 1535 1536 1537 1538 1539 1540 1541 1542 1543 1544 1545 1546 1547 1548 1549 1550 1551 1552 1553 1554 1555 1556 1557 1558 1559 1560 1561 1562 1563 1564 1565 1566 1567 1568 1569 1570 1571 1572 1573 1574 1575 1576 1577 1578 1579 1580 1581 1582 1583 1584 1585 1586 1587 1588 1589 1590 1591 1592 1593 1594 1595 1596 1597 1598 1599 1600 1601 1602 1603 1604 1605 1606 1607 1608 1609 1610 1611 1612 1613 1614 1615 1616 1617 1618 1619 1620 1621 1622 1623 1624 1625 1626 1627 1628 1629 1630 1631 1632 1633 1634 1635 1636 1637 1638 1639 1640 1641 1642 1643 1644 1645 1646 1647 1648 1649 1650 1651 1652 1653 1654 1655 1656 1657 1658 1659 1660 1661 1662 1663 1664 1665 1666 1667 1668 1669 1670 1671 1672 1673 1674 1675 1676 1677 1678 1679 1680 1681 1682 1683 1684 1685 1686 1687 1688 1689 1690 1691 1692 1693 1694 1695 1696 1697 1698 1699 1700 1701 1702 1703 1704 1705 1706 1707 1708 1709 1710 1711 1712 1713 1714 1715 1716 1717 1718 1719 1720 1721 1722 1723 1724 1725 1726 1727 1728 1729 1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740 1741 1742 1743 1744 1745 1746 1747 1748 1749 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759 1760 1761 1762 1763 1764 1765 1766 1767 1768 1769 1770 1771 1772 1773 1774 1775 1776 1777 1778 1779 1780 1781 1782 1783 1784 1785 1786 1787 1788 1789 1790 1791 1792 1793 1794 1795 1796 1797 1798 1799 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 1810 1811 1812 1813 1814 1815 1816 1817 1818 1819 1820 1821 1822 1823 1824 1825 1826 1827 1828 1829 1830 1831 1832 1833 1834 1835 1836 1837 1838 1839 1840 1841 1842 1843 1844 1845 1846 1847 1848 1849 1850 1851 1852 1853 1854 1855 1856 1857 1858 1859 1860 1861 1862 1863 1864 1865 1866 1867 1868 1869 1870 1871 1872 1873 1874 1875 1876 1877 1878 1879 1880 1881 1882 1883 1884 1885 1886 1887 1888 1889 1890 1891 1892 1893 1894 1895 1896 1897 1898 1899 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031 2032 2033 2034 2035 2036 2037 2038 2039 2040 2041 2042 2043 2044 2045 2046 2047 2048 2049 2050 2051 2052 2053 2054 2055 2056 2057 2058 2059 2060 2061 2062 2063 2064 2065 2066 2067 2068 2069 2070 2071 2072 2073 2074 2075 2076 2077 2078 2079 2080 2081 2082 2083 2084 2085 2086 2087 2088 2089 2090 2091 2092 2093 2094 2095 2096 2097 2098 2099 2100 2101 2102 2103 2104 2105 2106 2107 2108 2109 2110 2111 2112 2113 2114 2115 2116 2117 2118 2119 2120 2121 2122 2123 2124 2125 2126 2127 2128 2129 2130 2131 2132 2133 2134 2135 2136 2137 2138 2139 2140 2141 2142 2143 2144 2145 2146 2147 2148 2149 2150 2151 2152 2153 2154 2155 2156 2157 2158 2159 2160 2161 2162 2163 2164 2165 2166 2167 2168 2169 2170 2171 2172 2173 2174 2175 2176 2177 2178 2179 2180 2181 2182 2183 2184 2185 2186 2187 2188 2189 2190 2191 2192 2193 2194 2195 2196 2197 2198 2199 2200 2201 2202 2203 2204 2205 2206 2207 2208 2209 2210 2211 2212 2213 2214 2215 2216 2217 2218 2219 2220 2221 2222 2223 2224 2225 2226 2227 2228 2229 2230 2231 2232 2233 2234 2235 2236 2237 2238 2239 2240 2241 2242 2243 2244 2245 2246 2247 2248 2249 2250 2251 2252 2253 2254 2255 2256 2257 2258 2259 2260 2261 2262 2263 2264 2265 2266 2267 2268 2269 2270 2271 2272 2273 2274 2275 2276 2277 2278 2279 2280 2281 2282 2283 2284 2285 2286 2287 2288 2289 2290 2291 2292 2293 2294 2295 2296 2297 2298 2299 2300 2301 2302 2303 2304 2305 2306 2307 2308 2309 2310 2311 2312 2313 2314 2315 2316 2317 2318 2319 2320 2321 2322 2323 2324 2325 2326 2327 2328 2329 2330 2331 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342 2343 2344 2345 2346 2347 2348 2349 2350 2351 2352 2353 2354 2355 2356 2357 2358 2359 2360 2361 2362 2363 2364 2365 2366 2367 2368 2369 2370 2371 2372 2373 2374 2375 2376 2377 2378 2379 2380 2381 2382 2383 2384 2385 2386 2387 2388 2389 2390 2391 2392 2393 2394 2395 2396 2397 2398 2399 2400 2401 2402 2403 2404 2405 2406 2407 2408 2409 2410 2411 2412 2413 2414 2415 2416 2417 2418 2419 2420 2421 2422 2423 2424 2425 2426 2427 2428 2429 2430 2431 2432 2433 2434 2435 2436 2437 2438 2439 2440 2441 2442 2443 2444 2445 2446 2447 2448 2449 2450 2451 2452 2453 2454 2455 2456 2457 2458 2459 2460 2461 2462 2463 2464 2465 2466 2467 2468 2469 2470 2471 2472 2473 2474 2475 2476 2477 2478 2479 2480 2481 2482 2483 2484 2485 2486 2487 2488 2489 2490 2491 2492 2493 2494 2495 2496 2497 2498 2499 2500 2501 2502 2503 2504 2505 2506 2507 2508 2509 2510 2511 2512 2513 2514 2515 2516 2517 2518 2519 2520 2521 2522 2523 2524 2525 2526 2527 2528 2529 2530 2531 2532 2533 2534 2535 2536 2537 2538 2539 2540 2541 2542 2543 2544 2545 2546 2547 2548 2549 2550 2551 2552 2553 2554 2555 2556 2557 2558 2559 2560 2561 2562 2563 2564 2565 2566 2567 2568 2569 2570 2571 2572 2573 2574 2575 2576 2577 2578 2579 2580 2581 2582 2583 2584 2585 2586 2587 2588 2589 2590 2591 2592 2593 2594 2595 2596 2597 2598 2599 2600 2601 2602 2603 2604 2605 2606 2607 2608 2609 2610 2611 2612 2613 2614 2615 2616 2617 2618 2619 2620 2621 2622 2623 2624 2625 2626 2627 2628 2629 2630 2631 2632 2633 2634 2635 2636 2637 2638 2639 2
```

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

11.2.1.4. Network Operations

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

11.2.2. IDE Debugging

11.2.3. Extensions

12. Seminare

12.1. Installieren der wichtigen Software

12.1.1. 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>

12.1.2. Docker

[Installation von Docker Desktop auf Windows](#)

[Installation von Docker auf Linux](#)

[Docker Desktop Installation auf Mac](#)

12.1.3. Podman

[Installierung von Podman auf allen Plattformen](#)

12.1.4. 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] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software." pp. 127–134, 1994.
- [19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software." pp. 117–126, 1994.
- [20] Martin Fowler, "Inversion of Control Containers and the Dependency Injection pattern." [Online]. Available: <https://martinfowler.com/articles/injection.html>
- [21] Martin Fowler, "Inversion of Control." [Online]. Available: <https://martinfowler.com/bliki/InversionOfControl.html>
- [22] Ralph E. Johnson and Brian Foote, "Designing Reusable Classes." [Online]. Available: <http://www.laputan.org/drc/drc.html>

- [23] "Mapping Requests." [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html>
- [24] "URI Patterns." [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html#mvc-ann-requestmapping-uri-templates>
- [25] "Error Handling for REST with Spring in Kotlin." [Online]. Available: <https://www.baeldung.com/kotlin/spring-rest-error-handling>
- [26] "Controller Advice." [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-advice.html>
- [27] "When should we use @Transactional annotation?." [Online]. Available: <https://stackoverflow.com/questions/78132448/when-should-we-use-transactional-annotation>
- [28] "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>
- [29] "Defining JPA Entities." [Online]. Available: <https://www.baeldung.com/jpa-entities>
- [30] "Hibernate One to Many." [Online]. Available: <https://www.baeldung.com/hibernate-one-to-many>
- [31] "Spring Boot One To Many Relationship." [Online]. Available: <https://medium.com/@hk09/spring-boot-one-to-many-relationship-e617183b7861>
- [32] "One-to-One Relationship in JPA." [Online]. Available: <https://www.baeldung.com/jpa-one-to-one>
- [33] "Many-To-Many Relationship in JPA." [Online]. Available: <https://www.baeldung.com/jpa-many-to-many>
- [34] "Introduction to Spring Data JPA." [Online]. Available: <https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa>
- [35] "Container Overview." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/basics.html>
- [36] "Dependency Injection." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html>
- [37] "Method Injection." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-method-injection.html>
- [38] Nguyen Nam Thai, "What Is a Spring Bean." [Online]. Available: <https://www.baeldung.com/spring-bean>
- [39] "Bean Scopes." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html#beans-factory-scopes-singleton>
- [40] "WebSocket Scope." [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html>