Web Engineering

Entwicklung von Full-Stack-Web-Anwendungen mit Spring Boot

Script für Web Engineering 3

Inhaltsverzeichnis

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

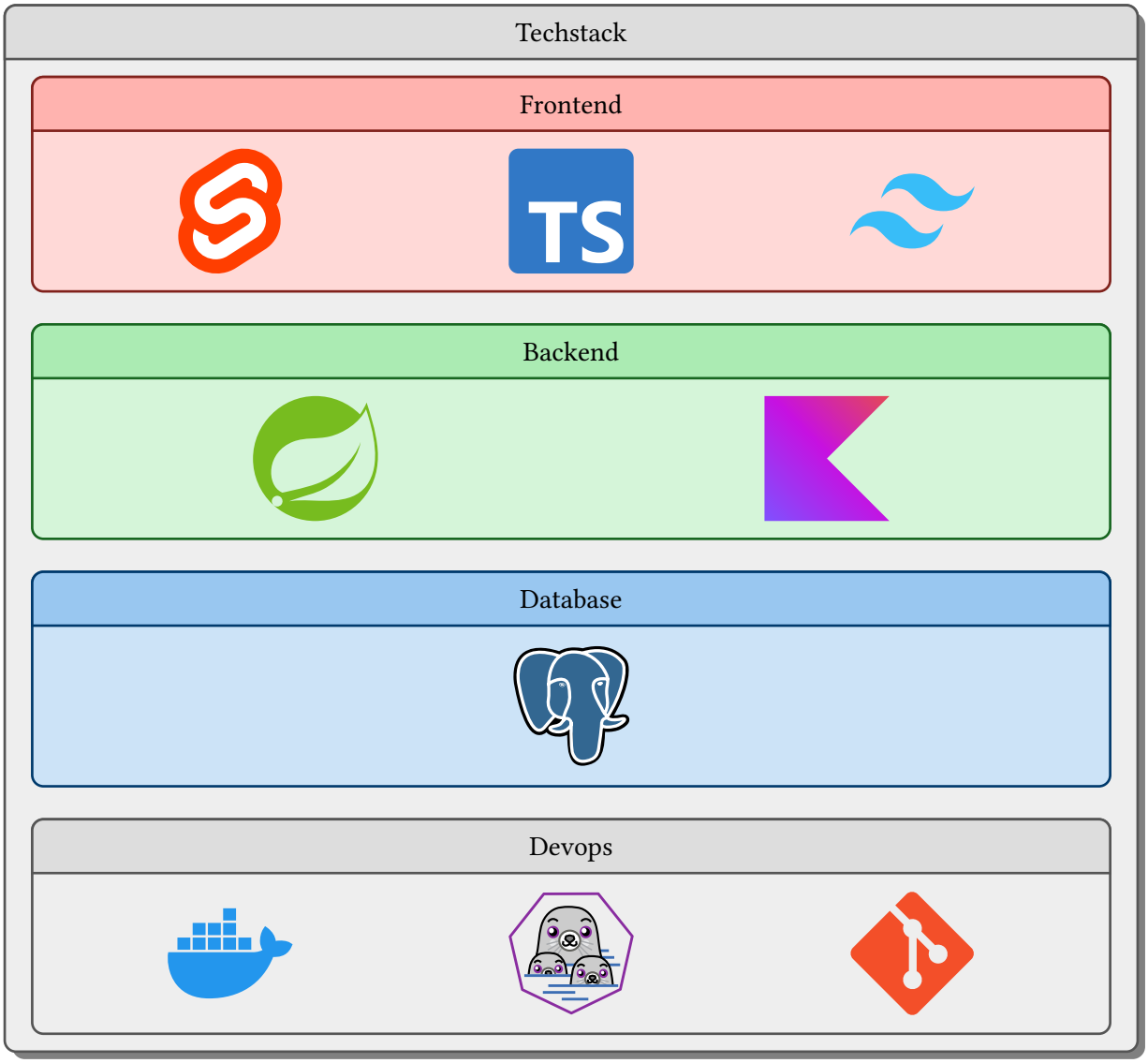
2.3. Präsentation bzw. Verteidigung

- 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
- Sping MVC
- Sping 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 untersützt. 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 Vorgängner Java EE. Spring hat sich dabei eingige Spezifikationen von Java EE angeeignet. Dazu gehören:

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

Neben diesen Spezifikationen untersützt 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. Kompatiblität 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 eingetzt 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 Serven eingesetzt werden, die keine Servlet Container sind (zum Beispiel Netty). [4]

3.2.2. Springboot

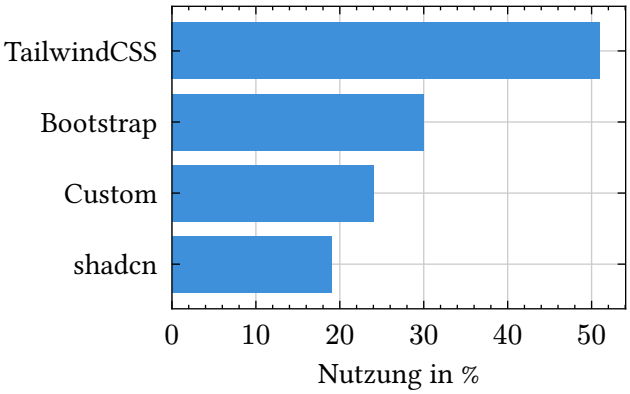
Springboot ermöglicht die Erstellung von Spring Anwendungen, die mit minimaler Konfiguration lauffähig sind. Es ist dabei eine Erweiterung des Spring Frameworks. [2] Dazu werden Webserver, wie zum Beispiel Tomcat, direkt mitgeliefert. Um den Start der Entwicklung zu vereinfachen werden außerdem *Starter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

Beispiel für eine Response:

```
1  HTTP/1.1 200 OK
2  ETag: "f60e0978bc9c458989815b18ddad6d75"
3  Last-Modified: Thu, 10 Jan 2013 01:45:22 GMT
4  Content-Type: application/vnd.collection+json
5
6  { "collection":
7    {
8      "version" : "1.0",
9      "href" : "http://www.youtypeitwepostit.com/api/",
10     "items" : [
11       { "href" : "http://www.youtypeitwepostit.com/api/messages/
         21818525390699506",
12         "data": [
13           { "name": "text", "value": "Test." },
14           { "name": "date_posted", "value": "2013-04-22T05:33:58.930Z" }
15         ],
16         "links": []
17       },
```

```
18        { "href" : "http://www.youtypeitwepostit.com/api/messages/
          3689331521745771",
19          "data": [
20            { "name": "text", "value": "Hello." },
21            { "name": "date_posted", "value": "2013-04-20T12:55:59.685Z" }
22          ],
23          "links": []
24        },
25        { "href" : "http://www.youtypeitwepostit.com/api/messages/
          7534227794967592",
26          "data": [
27            { "name": "text", "value": "Pizza?" },
28            { "name": "date_posted", "value": "2013-04-18T03:22:27.485Z" }
29          ],
30          "links": []
31        }
32      ]
33    },
34    "template": {
35      "data": [
36        {"prompt": "Text of message", "name": "text", "value":""}
37      ]
38    }
39  }
```

Jede HTTP Response kann in drei Teile aufgeteilt werden:

Der Status Code: Der Response Code beschreibt das Ergebnis der Request.

Der Body: Inhalt der Response in einem definierten Dokumenten Format. Das Format wird im Header festgelegt.

Die Header der Response: Der Header enthält eine Abfolge von Key-Value Paaren, die die Response und ihren Body beschreiben.

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 Spzifikationen 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 (Creted) 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 bereitgestellen 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 Requets 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 (Rediraction): 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 genutzen 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-**T**ransfer-**O**bject 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-**V**iew-**C**ontroller 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**: Neute Daten Einträge werden gespeichert.
- **Read**: Existierende Daten werden ausgelesen
- **Update**: Existierende Daten werden aktualisiert
- **Delete**: Daten werden gelöscht

[8]

4.9. ORM

Object-**R**elational-**M**apping 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 Instanziierens 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

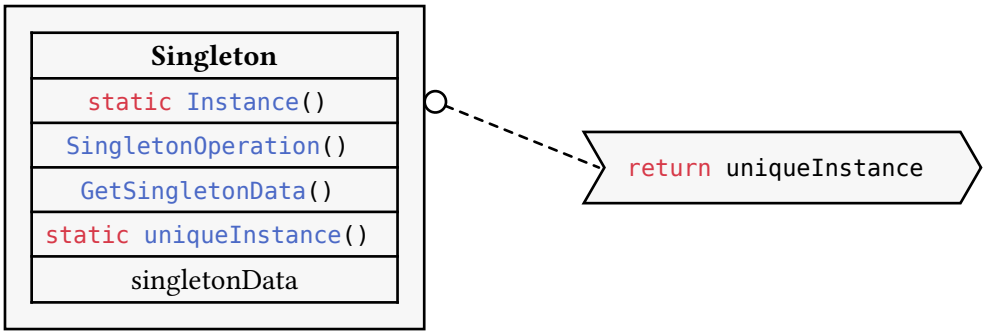


Figure 1: Struktur des Singleton Patterns

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

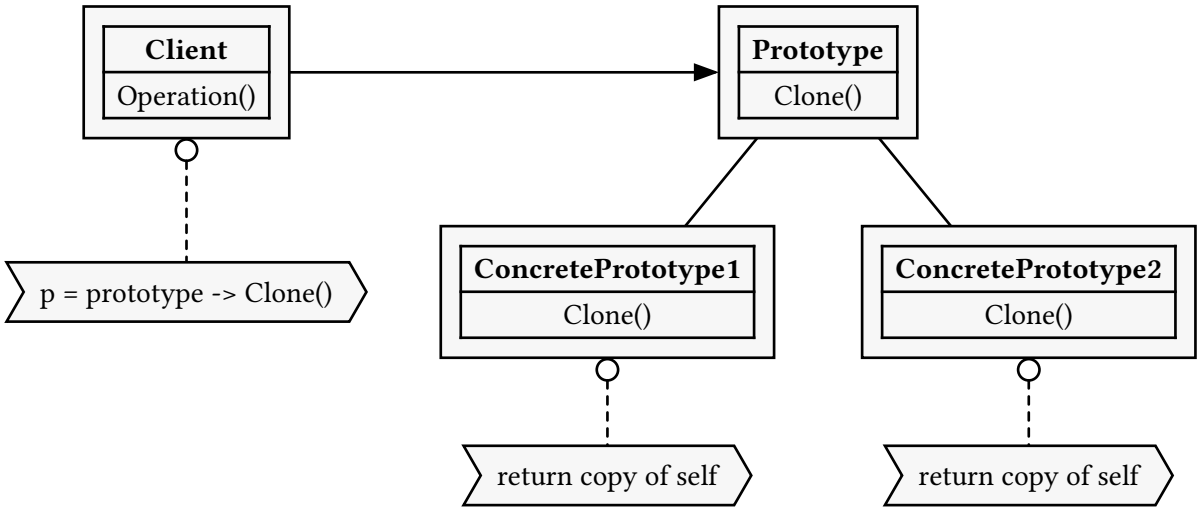


Figure 2: Struktur des Prototype-Patterns

- **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 Instanziieren 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 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 injeziert 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 InversionofControl Container zum Einsatz. Neben IoC Containern kommen auch EJBs (Jakarta Enterprise Beans) oft zum Einsatz. [21]

Beispiel zu IoC:

```ruby
puts 'What is your name?'
name = gets
process_name(name)
puts 'What is your quest?'
quest = gets
process_quest(quest)
```

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

```ruby
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text “What is Your Name?”}
name_label.pack
name = TkEntry.new(root).pack
name.bind(“FocusOut”) {process_name(name)}
quest_label = TkLabel.new() {text “What is Your Quest?”}
quest_label.pack
quest = TkEntry.new(root).pack
```

```
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:

- Contructor Injection, Type 3 IoC (Section 8.1.4.1)
- Setter Injection, Type 2 IoC (Section 8.1.4.2)
- Interface Injection, Type 1 IoC

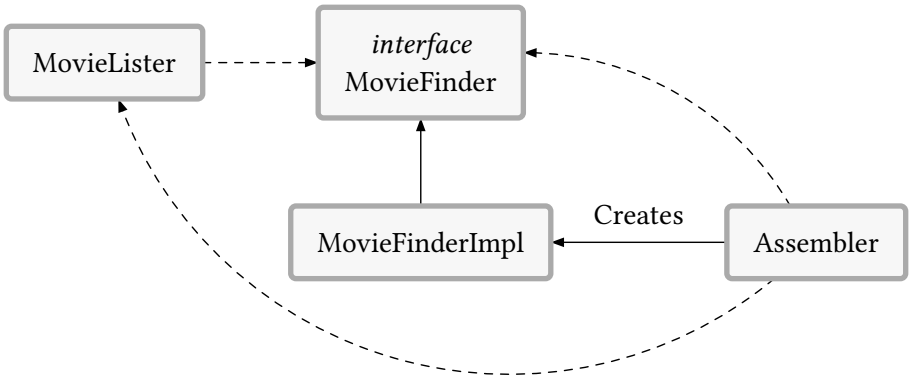


Figure 3: Abhängigkeiten für Dependency Injection

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

7. Architektur von Web Services

7.1. REST API Design

[23]

8. Backend

8.1. Spring

8.1.1. Komponenten einer Spring Anwendung

Beispielkomponenten Anhand eines Users. Dieser User hat folgende Felder:

- Name
- Alter

8.1.1.1. Controller

```kotlin
@RestController
@RequestMapping("/path/to/controller")
class Controller @Autowired constructor(
  private val service: Service
) {
  @GetMapping
  @ReponseStatus(HttpStatus.OK)
  fun getEntities(): List<GetEntityDto> {

  }
}
```

8.1.1.1.1. 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.

Nuzten 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.

[24]

8.1.1.1.2. 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 abgelgichen
- `"/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.

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

}
```

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

```kotlin
@Controller
@RequestMapping("/owners/{ownerId}")
class OwnerController {
  @GetMapping("/pets/{petId}")
  fun findPet(@PathVariable ownerId: Long, @PathVariable petId: Long): Pet {

  }
}
```

Variablen aus der URI werden automatisch in den korrekten Typ konvertiert. Einfache Typen wie zum Beispiel `int`, `long` oder `Date` werden direkt unterstüzt. 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.

```kotlin
@PathVariable("customName")
```

[25]

8.1.1.1.3. Error Handling

```kotlin
class CustomException(message: String): RuntimeException(message) {

}
```

```kotlin
 @ControllerAdvice
 class ExceptionControllerAdvice {
   fun handleCustomException(
     ex: CustomException
   ): ResponseEntity<ErrorMessageModel> {
     val errorMessage = ErrorMessageModel(
       HttpStatus.NOT_FOUND.value(),
       ex.message
     )
     return ResponseEntity(
```

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

[26]

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

```kotlin
@ControllerAdvice(annotations = [RestController::class])
class Advice
```

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

```kotlin
@ControllerAdvice("org.example.controllers")
class Advice
```

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

```kotlin
@ControllerAdvice(
  assignableTypes = [
    ControllerInterface::class,
    AbstractController::class
    ]
  )
class Advice
```

Die Klasse wird nur noch Controller ansprechen, die auf das Interface oder den abstrakten Controller zugewiesen werden können. [27]

8.1.1.2. Service

```kotlin
@Service
class EntityService (
  private val entityReposirory: EntityReposirory
) {
  // service functions
}
```

Der Service enthält die Business Logik der Anwendung. Um einen Service zu deklaieren 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.

8.1.1.2.1. 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.
[28]

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

- **Datenbank Operation mit mehreren Schritten**: Wenn mehrere Operationen erfolreiche 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 werder. Durch `@Transactional` würde bei einem Fehlschlag einer dieser Operation die Datenbank konsistent bleiben und keine Änderungen commited 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.

```kotlin
@Entity(name = "user")
@Table(name = "user")
class User(
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  val id: Long? = null,
  @OneToMany(
      mappedBy = "user",
      cascade = [CascadeType.REMOVE]
  )
```

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

```kotlin
@Transactional
fun deleteUser(id: Long) {
    val toDeleteUser = getUserById(id)
    userRepository.delete(toDeleteUser)
}
```

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**

[29]

8.1.1.3. Entity

```kotlin
@Entity(name = "entityName")
@Table(name = "entityName")
class Entity {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  val id: Long? = null,
  // ...
}
```

8.1.1.3.1. `@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. [30]

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

8.1.1.3.2. `@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. [30]

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

8.1.1.3.3. `@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.
[30]

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 Eitrag bereits in der Datenbank gibt. Wenn die Id null ist, weiß Spring, dass es sich hier um einen neuen Eintrag handelt.

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

8.1.1.3.4. `@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.

[30]

8.1.1.3.5. 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.

8.1.1.3.5.1. `@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 notwending aber empfohlen. Wenn sie nicht vorhanden ist, wird automatisch nach dem primary Key in der Entity gesucht.

```kotlin
@Entity(name = "user")
@Table(name = "user")
```

```
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

[31], [32]

8.1.1.3.5.2. `@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. [33]

```
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

8.1.1.3.5.3. `@ManyToMany`

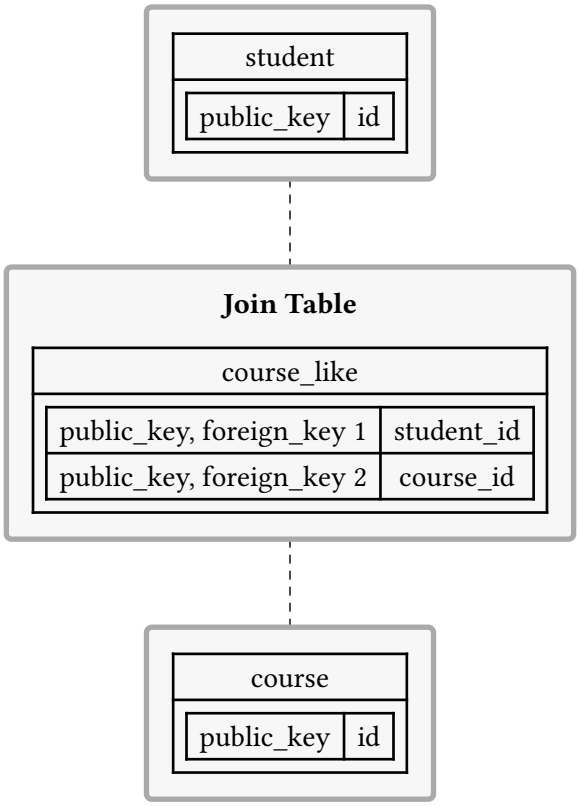


Figure 4: Aufbau einer Many to Many Relation mit einem Join Table

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 erlaten 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 notwending. 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 angeben werden, der die Relation mapped. [34]

```kotlin
class Student (
  @ManyToMany
  @JoinTable(
    name = "course_like",
    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  )
```

8.1.1.4. Repository

```kotlin
@Repository
interface EntityReposirory: JpaRepository<Entity, Long> {

}
```

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:

```kotlin
@Repository
interface UserReposirory: JpaRepository<User, Long> {

}
```

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 vorraussetzen.

8.1.1.4.1. 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.

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

8.1.1.4.2. Manuelle Queries

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

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

[35]

8.1.1.5. 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 gewollten Daten aus der Entity enthalten.

Ein DTO, welches zum Erstellen einer Entity genützt 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, Addresse, 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:

```kotlin
class Entity(
  val id: Long?,
  val name: String,
  val age: Integer
)
```

```kotlin
data class GetEntityDto (
  val id: Long,
  val name: String,
  val age: Integer
)
```

```kotlin
data class PostEntityDto (
  val name: String,
  val age: Integer
)
```

8.1.1.6. 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.

```kotlin
class Entity(
  val id: Long?,
  val name: String,
  val age: Integer
)
```

```kotlin
fun convertEntityToGetEntityDto(entity: Entity): GetEntityDto {
  return GetEntityDto(
    id = entity.id!!,
    name = entity.name,
    age = entity.age
  )
}
```

8.1.1.7. Application Konfiguration

```yaml
# application.yml

server:
  port: 8080

spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/database
    username: database_username
    password: database_password
  jpa:
    hibernate:
      ddl-auto: create-drop
    show-sql: true
    properties:
      hibernate:
        format_sql: true
```

8.1.2. Spring Data & Spring Data JPA

Das Ziel von Spring Data ist es, ein auf Spring basierendes Programmiermodell bereitzustellen, mit dem man auf Daten zugreifen kann. Neben diesem Zugriff sollen auch die Eigenschaften des Data Stores erhalten bleiben.

Der Fokus liegt dabei auf die einfache Nutzen von Datenbanken, Cloud basierten Datendiensten, Map-Reduce Frameworks und Zugriffstechnologien. Spring Data selbst ist dabei ein übergreifendes Projekt, welches viele Subprojekte besitzt, die zum Beispiel auf eine bestimmte Datenbank zugeschnitten sind.

Folgende Features stellt Spring Data bereit:

- Repositories und Object-Mapping-Abstraktionen
- Dynamische Query Generierung basierend auf Methoden Namen
- Support für `Auditing` mit Informationen über `created` und `last changed`
- Integrierung von eigenem Repository-Code ist möglich

[36]

Spring Data JPA erlaubt das einfach Implementieren von Repositories, die auf der Java Persistence API basieren. Das Ziel ist es dabei die Menge an Boilerplate Code zu verringern und eine Schnelle Erstellung von Repository Interfaces zu ermöglichen. Spring soll dabei den Erstellung von allen Aufgaben im Hintergrund übernehmen. [37]

Ein beispiel Repository kann dabei wie folgt aussehen:

```java
@Entity
class Person {

  @Id @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String name;

  // getters and setters omitted for brevity
}

interface PersonRepository extends Repository<Person, Long> {

  Person save(Person person);

  Optional<Person> findById(long id);
}
```

Hier kommt ein einfaches Repository zum Einsatz. Je nachdem, wie viele Funktionen benötigt werden, können auch auch komplexere Repositories, wie zum Beispiel `JpaReposiroty` genutzt werden. [38]

8.1.2.1. Konzepte

Das zentrale Interface hier ist das `Repository`. Es benötigt dafür als Argumente eine Domain-Klasse an, die es verwalten soll, zusammen mit dem Typ ihres Identifiers. Dieses Interface stellt dabei eine Basis dar. Es ist ein “Maker Interface”, welches die angegebenen Typen erhalten soll und wird dann oft mit komplexeren Repositories erweitert.

Ein Beispiel wäre hier das `CRUD Repository`, welches durch `Repository` erweitert wird und es mit mehr Funktionalitäten erweitert.

```java
public interface CrudRepository<T, ID> extends Repository<T, ID> {

  <S extends T> S save(S entity);

  Optional<T> findById(ID primaryKey);

  Iterable<T> findAll();

  long count();

  void delete(T entity);

  boolean existsById(ID primaryKey);

  // … more functionality omitted.
}
```

Die hier definierten Methoden decken alle CRUD Operationen ab.

- `save` zum Erstellen und aktualisieren
- `findById`, `findAll`, `count` und `existsById` zum Auslesen von Datenbank
- `delete` zum Löschen

[39]

8.1.2.2. Query Methoden Definieren

[40]

8.1.2.3. Probleme in Anwendungsfällen und ihre Lösung

[41]

8.1.3. 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 [42].

8.1.4. 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 bentötigten Abhängigkeiten. Die Bean hat in diesem Fall keine Kontrolle über die Erstellung oder den Ort ihrer Abhängigkeiten [43].

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

Contructor oder Setter DI — Richtlinie 3

Der Contructor 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.

8.1.4.1. Contructor Injection - Type 3 IoC

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

```kotlin
class ExampleClass(private val dependency: Dependency) {

}
```

8.1.4.2. Setter Injection - Type 2 IoC

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

```kotlin
class ExampleClass {
  lateinit var dependency: Dependency
}
```

8.1.5. Method Injection

Ressource für Method Injection [44]

8.1.6. 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. [45]

8.1.6.1. 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. [45]

Beispiel

```java
public class Company {
  private Address address;

  public Company(Address address) {
    this.address = address;
  }
}
```

```java
public class Address {
  private String street;
  private int number;

  public Address(String street, int number) {
    this.street = street;
    this.number = number;
  }
}
```

Traditionelle Erstellung der Abhängigkeiten:

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

Herangehensweise mit Beans

```java
@Component
public class Company {
    // this body is the same as before
}
```

Konfiguration des IoC Containers mit Metadaten zu den Address Beans:

```java
@Configuration
@ComponentScan(basePackageClasses = Company.class)
public class Config {
    @Bean
    public Address getAddress() {
        return new Address("High Street", 1000);
    }
}
```

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.

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

Die Funktionalität der Beans kann wie folgt verifiziert werden:

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

8.1.6.2. 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-Veiew-Controller darstellt

8.1.6.3. Scoping

8.1.6.3.1. Singleton

Singleton-Beans folgen dem Singleton-Design-Pattern. Theorie zu diesem Pattern ist zu finden in Section 5.1.

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 [46].

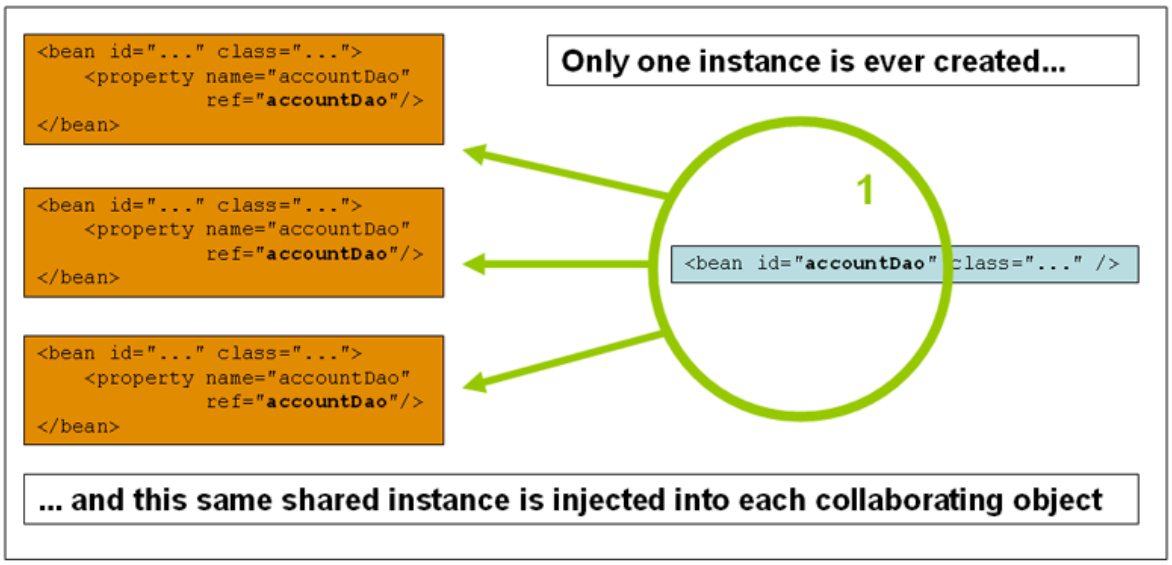


Figure 5: Funktionalität des Singleton Scopes

```xml
<bean
  id="accountService"
  class="com.something.DefaultAccountService"
/>
```

Einsatz von Singleton Beans Richtlinie 5

Singleton Beans sollten für stateless Beans eingesetzt werden.

8.1.6.3.2. Prototype

Prototype-Beans folgen dem Prototype-Design-Pattern. Theorie zu diesem Pattern ist zu finden in Section 5.2.

Eine neue Instanz der Bean wird bei jeder Anfrage erstellet [1]. Diese Anfrage kann durch Injection in eine andere Bean oder durch eine Anfrage durch `getBean()` geschehen [46].

Spring verwaltet, anders als bei anderen Beans, nicht den kompletten Lebenszyklus einer Prototype Bean. Das Löschen einer Prototype Bean muss manuel durch den Client geschehen. Ein eigens definierter Bean Post-Processor kann genutzt werden, damit der Container Ressourcen, die von Prototype Beans gehalten werden, freigibt [46].

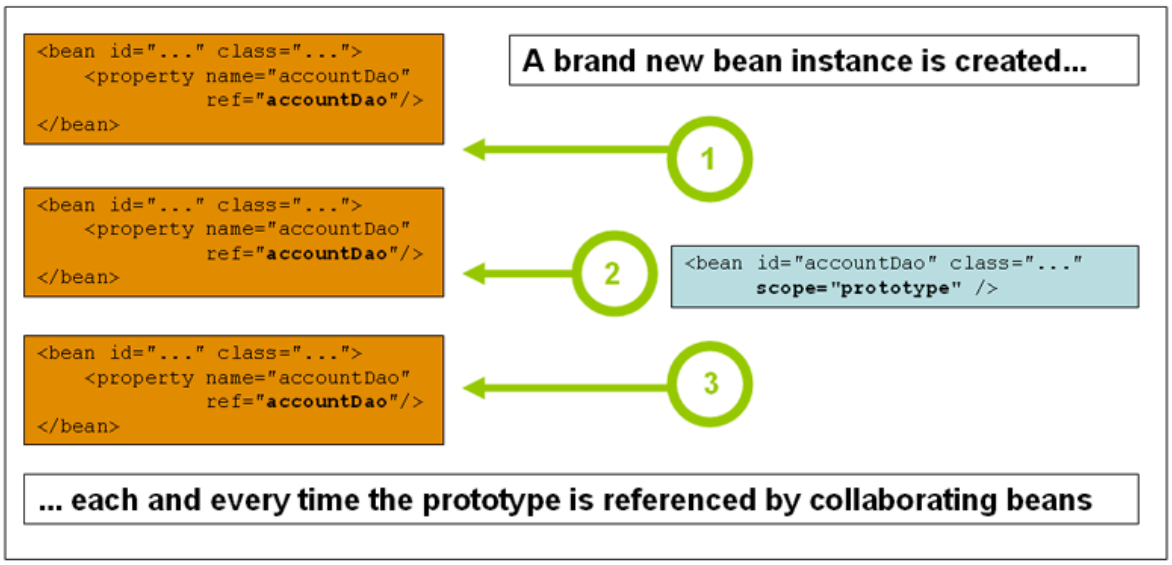


Figure 6: Funktionalität des Prototype Scopes

```xml
<bean
  id="accountService"
  class="com.something.DefaultAccountService"
  scope="prototype"
/>
```

Einsatz von Prototype Beans Richtlinie 6

Prototype Beans sollten für stateful Beans eingesetzt werden.

8.1.6.3.3. 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 [46].

```xml
<bean
  id="loginAction"
  class="com.something.LoginAction"
  scope="request"
/>
```

```kotlin
@RequestScope
@Component
class LoginAction {
  // ...
}
```

8.1.6.3.4. Session

Eine einzelne Instanz wird für jede HTTP Session erstellt [1]. Die erstelle Bean wird praktisch auf die HTTP Session gescoped. 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 [46].

```xml
<bean
  id="userPreferences"
  class="com.something.UserPreferences"
  scope="session"
/>
```

```kotlin
@SessionScope
@Component
class UserPreferences {
  // ...
}
```

8.1.6.3.5. 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`

[46]

```xml
<bean
  id="appPreferences"
  class="com.something.AppPreferences"
  scope="application"
/>
```

```kotlin
@ApplicationScope
@Component
class AppPreferences {
  // ...
}
```

8.1.6.3.6. WebSocket

Der WebSocket Scope ist an den Lebenszyklus einer WebSocket gekoppelt [46].
Weitere Informationen: https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html [47]

TODO: WebSocket Scope Kapitel ausbauen.

8.1.7. Aspect Oriented Programming (AOP)

8.1.8. 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.

8.1.9. OpenAPI UI

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

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

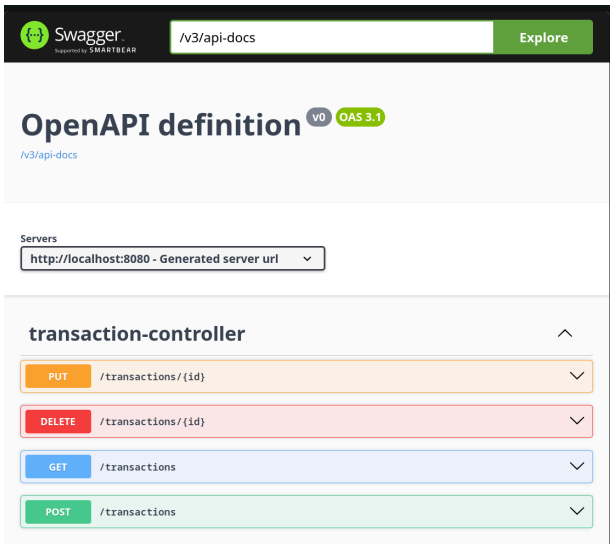
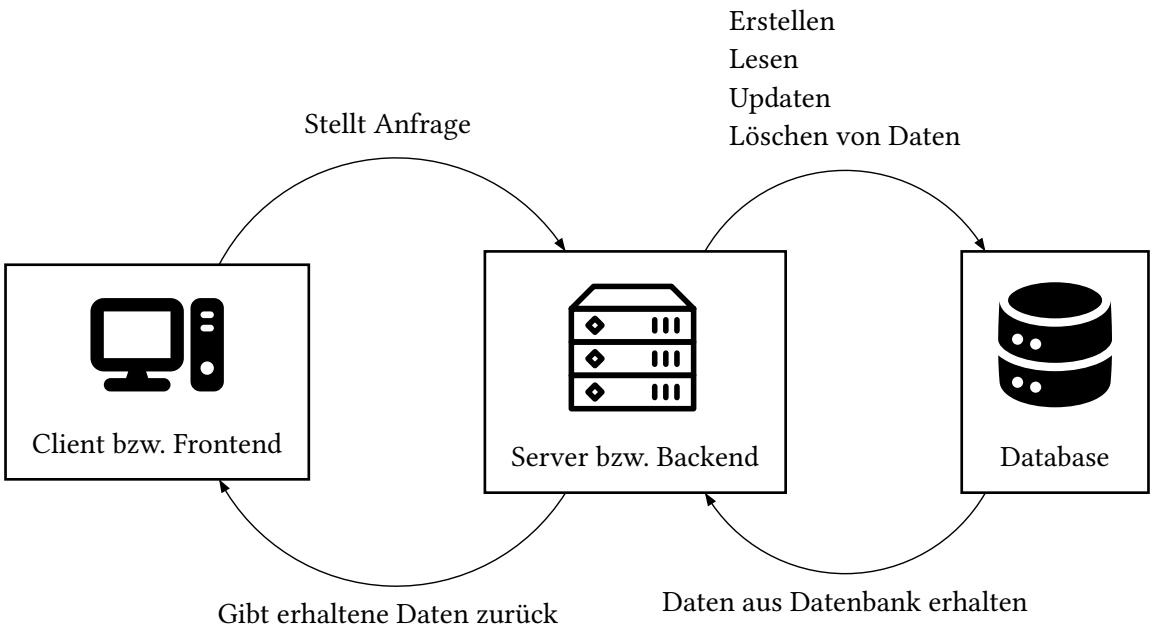


Figure 7: Swagger UI zum Darstellen und Testen der REST Endpunkte

8.1.10. Authentication

8.1.11. Ablauf einer Anfrage an ein Spring Backend

Kontext der Spring Anwendung:
Innerhalb einer Fullstack Anwendung existiert die Spring Anwendung im Backend. Das Frontend stellt Anfragen an das Spring Backend. Das Backend verarbeitet die Anfragen und gibt bei Bedarf Daten an das Frontend zurück.



Beispiel: Ein Nutzer benutzt eine Finanzapp. Er möchte einen Teil seines Geldes zu einem Budget hinzufügen. Dafür füllt er in der App zwei Felder aus. Das Zielbudget und die Menge an Geld, die dem Budget zugewiesen werden soll. Nach Bestätigung der Eingaben wird eine Anfrage von der App an das Backend geschickt. Diese Anfrage enthält alle Informationen, die das Backend benötigt, um die Budgetanpassung zu verarbeiten. Dazu gehören zum Beispiel: User ID, Konto ID, Budget ID, Wert. All diese Werte werden im Backend genutzt, um in der Datenbank den passenden Nutzer, Konto und Budget zu finden. Danach kann überprüft werden, ob diese Transaktion stattfinden darf und der Wert wird auf das Budget gutgeschrieben. Das Backend kann nun als Antwort auf die Anfrage den neuen Wert des Budgets an das Frontend zurücksckschicken, damit der Nutzer in der App sieht, dass sich der Wert des Budgets angepasst hat.

Entity:
Die Entity ist die Beschreibung eines Tabelleneintrages in der Datenbank. Felder in der Tabelle werden dabei als Variable deklariert. Genauere Definitionen können durch `@Column` festgelegt werden. Jede Entity benötigt außerdem einen Primary Key, der mit der `@Id` Annotation festgelegt werden. Wenn eine Menge an Daten aus dem Table ausgelesen werden, ist jeder Eintrag in der Tabelle eine Entity Instanz.

```kotlin
@Entity(name = "todoitem")
@Table(name = "todoitem")
class TodoItem (
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  val id: Long? = null,
  @Column(name = "name", nullable = false)
  val name: String = "",
```

```
9 )
```

```java
@Entity(name = "todoitem")
@Table(name = "todoitem")
public class TodoItem {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  @Column(name = "name", nullable = false)
  private String name;
}
```

Beispiel: Ich frage in der Datenbank alle Todo Items ab, die bis nächste Woche Freitag fertig sein sollen. 5 Todos wurden dabei gefunden. Ich erhalte vom Repository eine Liste mit 5 Elementen. Jedes Element ist dabei eine Instanz der TotoItems Entity, die die Daten des jeweiligen Tabelleneintrages enthält.

Repository:
Das Repository ist die Schnittstelle zur Datenbank. Es erlaubt das Erstellen und Suchen von Entities. Ein Repository kann sich immer nur um eine Art von Entity kümmern. Diese Entity wird beim Erstellen des Repositories angegeben, zusammen mit dem Typen der ID.

```kotlin
@Repository
interface TodoItemRepository : JpaReposiroty<TodoItem, Long> {

}
```

```java
@Repository
public interface TodoItemRepository extends JpaReposiroty<TodoItem, Long> {

}
```

```
1    POST http://localhost:8080/todos HTTP/1.1                    } (1) Request Line
2    accept: */*                                                  } (2) Headers
3    Content-Type: application/json                               }
4
5    {                                                            } (3) JSON Body
6      "name": "todoItemName",                                    }
7      "description": "newDescription",                           }
8      "done": true,                                              }
9      "created": "2025-10-23T15:06:08.738Z",                     }
10     "shouldBeDoneBy": "2025-11-04T08:06:06.297Z",              }
11   }
```

Es wird eine Anfrage an das Spring Backend gestellt. Vom Frontend kommen folgenden Daten in der Request:

- Eine **POST** Request an die url `http://localhost:8080/todos`
- Es wird Content in Form von **JSON** mitgeschickt
- Der Content im Body der Request. Hier werden alle Daten für eine neues Todo Item mitgeschickt

```JSON
{
  "name": "todoItemName",
  "description": "newDescription",
  "done": true,
  "created": "2025-10-23T15:06:08.738Z",
  "shouldBeDoneBy": "2025-11-04T08:06:06.297Z",
}
```

Das Spring Backend nimmt diese Request an der passenden Methode im Controller entgegen. Die passende Methode wird dabei wie folgt gefunden.

Es werden alle Klassen in Betracht gezogen, die eine `@RestController` Annotation besitzen. Danach wird der Pfad in der URL wichtig.

```
http://localhost:8080/todos
```

Der passende Controller muss dabei eine `@RequestMapping` Annotation besitzen mit diesem Pfad. Ein Controller, der also wie folgt definiert ist, wird für diese Request genutzt:

```Kotlin
@RestController
@RequestMapping("/todos")
class TodoItemController {
  // Methoden
}
```

In diesem Controller muss nun noch die passende Funktion gefunden werden. Da nach `/todos` kein weiterer Pfad kommt, wird hier nur die HTTP Methode betrachtet. Es handelt sich hier um eine POST Request. Die passende Methode muss also mit `@PostMapping` auf die POST Methode gemapped sein. In dem Controller wird also die Methode genutzt, die wie folgt definiert ist:

```Kotlin
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
fun createTotoItem(
    @RequestBody createTodoItemDto: CreateTodoItemDto
): GetTodoItemDto {
    return todoItemService.createTodoItem(createTodoItemDto)
}
```

Diese Methode wird nun aufgerufen. Dabei nimmt sie ein DTO entgegen. Dieses DTO ist, praktisch gesehen, der JSON Body der Request in Form eines Objekts in Java/Kotlin.

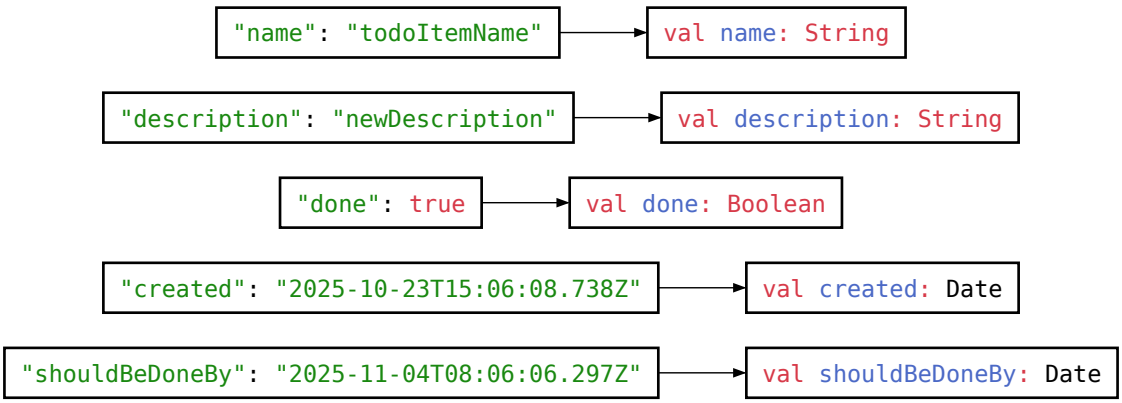
Für jedes JSON Feld wird dabei eine Variable in der Klasse bevölkert.

```JSON
{
  "name": "todoItemName",
  "description": "newDescription",
```

```Kotlin
data class
CreateTodoItemDto(
  val name: String,
```

```
    "done": true,
    "created":
    "2025-10-23T15:06:08.738Z",
    "shouldBeDoneBy":
    "2025-11-04T08:06:06.297Z",
}
```

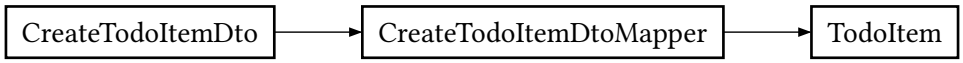
```
    val description: String,
    val done: Boolean,
    val created: Date,
    val shouldBeDoneBy: Date
)
```



Dieses DTO wird als Parameter an die Controller Funktion übergeben. Die Controller-Funktion ruft dann eine passende Methode im Service auf und übergibt das DTO.

Der Service enthält dabei die Logik der Anwendung. Theoretisch könnte dieser Code auch direkt im Controller stehen, aber im Sinne der Ordnung, sollte der Controller nur eine Methode im Service aufrufen.

Die Methode im Service, die hier benötigt wird, soll ein neues Todo Item erstellen. Damit ein neues Todo Item in der Datenbank gespeichert werden kann, muss das DTO in eine Instanz der Entity umgewandelt werden. Für dieses Umwandeln kommt ein Mapper zum Einsatz.



Dieser Mapper überführt alle Felder aus dem DTO in die passenden Felder der Entity.

```kotlin
fun fromCreateTodoItemDto(
    createTodoItemDto: CreateTodoItemDto
): TodoItem {
    return TodoItem(
        name = createTodoItemDto.name,
        description = createTodoItemDto.description,
        done = createTodoItemDto.done,
        created = createTodoItemDto.created,
        shouldBeDoneBy = createTodoItemDto.shouldBeDoneBy
    )
}
```

Diese Mapper Funktion wird im Service aufgerufen, um eine Instanz des neuen Todo Items zu erhalten.

```kotlin
val newTodoItem = fromCreateTodoItemDto(createTodoItemDto)
```

Damit diese neue Entity in der Datenbank gespeichert werden kann, wird die `save` Funktion im Repository aufgerufen. Diese Funktion kann entweder einen neuen Eintrag in der Datenbank anlegen oder einen vorhandenen updaten. Welche dieser Funktionen aufgerufen wird entscheided die übergebene Entity.

- Wenn die ID in der Entity `null` ist, wird ein neuer Eintrag in der Datenbank angelegt.
- Hat die ID einen definierten Wert, wird in der Datenbank der Eintrag mit der passenden ID gesucht und mit den neuen Werten überschrieben.

```kotlin
val savedTodoItem = todoItemRepository.save(newTodoItem)
```

Nach einer POST Request soll die neu erstellte Entity wieder in einer Response an das Frontend geschickt werden. Dazu muss die Entity in ein `GetTodoItemDto` überführt werden.

Dieses DTO wird an den Controller übergeben. Der Controller gibt das DTO als JSON Objekt in der Response an das Frontend zurück.

```
connection: keep-alive
content-type: application/json
date: Tue,04 Nov 2025 08:06:30 GMT
keep-alive: timeout=60
transfer-encoding: chunked

{
  "id": 1,
  "name": "string",
  "description": "string",
  "done": true,
  "created": "2025-11-04T08:06:06.296+00:00",
  "shouldBeDoneBy": "2025-11-04T08:06:06.297+00:00",
  "userId": 1
}
```

(1) Headers

(2) JSON Body

8.2. Jakarta EE

8.3. Lombok

8.4. Buildtools

8.4.1. Gradle

8.4.2. Maven

8.5. Documentation

9. Frontend

9.1. Frameworks

9.1.1. React

9.1.2. Svelte

9.1.3. VueJS

9.1.4. Angular

9.2. Web Components

9.3. CSS

10. Dev Ops

10.1. Docker

10.1.1. Dockerfile

10.1.2. Image

10.1.3. Container

10.1.4. Volumes

10.1.5. Networking

10.1.6. Compose

10.2. Podman

10.3. Nginx

11. Werkzeuge

11.1. IntelliJ

11.1.1. 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`

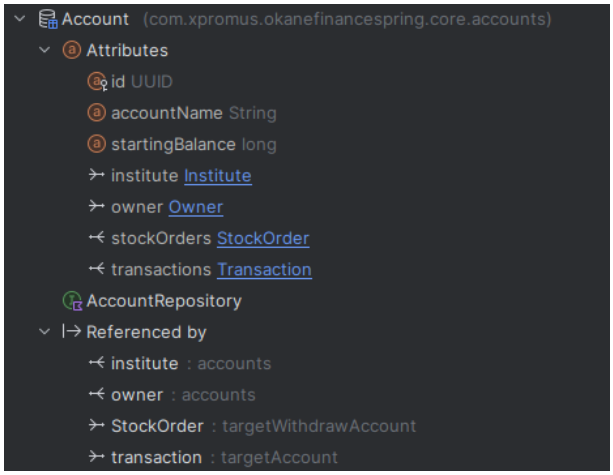


Figure 16: Ansicht des Persistence View mit Relationen zu anderen Entities

11.1.2. 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`

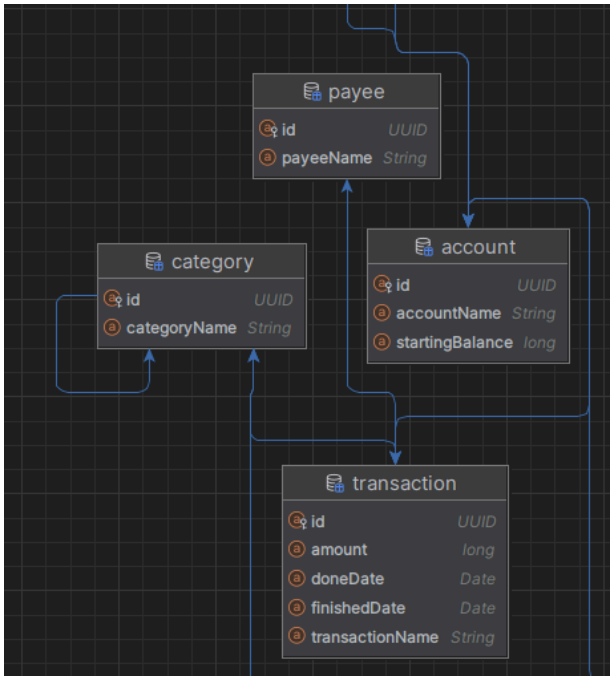


Figure 17: Das Entity Relationship Diagram zu einer Finanzanwendung

12. 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)

12.1. Backend Debugging

12.2. Frontend Debugging

12.2.1. Browser DevTools

Firefox DevTools User Docs

12.2.1.1. HTML/CSS Inspection

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

12.2.1.2. JavaScript Console

- Log output lesen
- Interagieren mit der Website durch JavaScript

12.2.1.3. JavaScript Debugger

- Durchschreiten von JavaScript auf der Website

12.2.1.3.1. 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

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

Generierung:

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

```JavaScript
// svelte.config.js
const config = {
  compilerOptions: {
    enableSourcemap: true
  },
}
```

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

```JSON
/* tsconfig.node.json */
{
  "compilerOptions": {
    "sourceMap": true
  }
}
```

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

```JSON
/* angular.json */
"projects": {
  "vite-project": {
    "architect": {
      "build": {
```

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

12.2.1.4. Network Operations

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

12.2.2. IDE Debugging

12.2.3. Extensions

13. Seminare

13.1. Installieren der wichtigen Software

13.1.1. 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

13.1.2. Docker

Installation von Docker Desktop auf Windows

Installation von Docker auf Linux

Docker Dekstop Installation auf Mac

13.1.3. Podman

Installierung von Podman auf allen Platformen

13.1.4. 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] Mark Massé, *REST API Design Rulebook*. 2012.

[24] “Mapping Requests.” [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html

[25] “URI Patterns.” [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html#mvc-ann-requestmapping-uri-templates

[26] “Error Handling for REST with Spring in Kotlin.” [Online]. Available: https://www.baeldung.com/kotlin/spring-rest-error-handling

[27] “Controller Advice.” [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-advice.html

[28] “When should we use @Transactional annotation?.” [Online]. Available: https://stackoverflow.com/questions/78132448/when-should-we-use-transactional-annotation

[29] “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

[30] “Defining JPA Entities.” [Online]. Available: https://www.baeldung.com/jpa-entities

[31] “Hibernate One to Many.” [Online]. Available: https://www.baeldung.com/hibernate-one-to-many

[32] “Spring Boot One To Many Relationship.” [Online]. Available: https://medium.com/@hk09/spring-boot-one-to-many-relationship-e617183b7861

[33] “One-to-One Relationship in JPA.” [Online]. Available: https://www.baeldung.com/jpa-one-to-one

[34] “Many-To-Many Relationship in JPA.” [Online]. Available: https://www.baeldung.com/jpa-many-to-Many

[35] Eugen Paraschiv, “Introduction to Spring Data JPA.” [Online]. Available: https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa

[36] “Spring Data.” [Online]. Available: https://spring.io/projects/spring-data#overview

[37] “Spring Data JPA.” [Online]. Available: https://spring.io/projects/spring-data-jpa

[38] “Getting Started.” [Online]. Available: https://docs.spring.io/spring-data/jpa/reference/jpa/getting-started.html

[39] “Core Concepts.” [Online]. Available: https://docs.spring.io/spring-data/jpa/reference/repositories/core-concepts.html

[40] “Defining Query Methods.” [Online]. Available: https://docs.spring.io/spring-data/jpa/reference/repositories/query-methods-details.html

[41] “JPA with Spring Boot: A Comprehensive Guide with Examples.” [Online]. Available: https://medium.com/@bshiramagond/jpa-with-spring-boot-a-comprehensive-guide-with-examples-e07da6f3d385

[42] “Container Overview.” [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/basics.html

[43] “Depency Injection.” [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html

[44] “Method Injection.” [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-method-injection.html

[45] Nguyen Nam Thai, “What Is a Spring Bean.” [Online]. Available: https://www.baeldung.com/spring-bean

[46] “Bean Scopes.” [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html#beans-factory-scopes-singleton

[47] “WebSocket Scope.” [Online]. Available: https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html