



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. Svelte Übersicht .....	6
3.3. Tailwind CSS Übersicht .....	7
3.4. Spring Übersicht .....	8
3.4.1. Geschichte von Spring .....	8
3.4.2. Springboot .....	8
4. Grundbegriffe .....	10
4.1. Client Side .....	10
4.2. Server Side .....	10
4.3. API .....	10
4.4. REST .....	10
4.4.1. Unabhängigkeit .....	10
4.4.2. Stateless .....	10
4.4.3. Kommunikation zwischen Client und Server .....	10
4.5. HTTP .....	11
4.5.1. Request .....	11
4.5.1.1. Header .....	11
4.5.1.2. Pfad .....	11
4.5.2. Response .....	11
4.5.3. Verben .....	11
4.5.3.1. GET .....	11
4.5.3.2. POST .....	11
4.5.3.3. PUT .....	12
4.5.3.4. DELETE .....	12
4.5.4. Status Codes .....	12
4.6. MVC .....	14
4.7. ORM .....	14
5. Technologien .....	15
5.1. Frontend .....	15
5.2. Backend .....	15
5.3. Database .....	15
5.4. Testing .....	15
5.5. DevOps .....	15
6. User Stories .....	16
6.1. User Stories für das finale Projekt .....	16
7. Backend .....	17
7.1. Spring .....	17
7.1.1. Spring Data .....	17
7.1.1.1. Spring Data JDBC .....	17
7.1.1.2. Spring Data JPA .....	18
7.1.1.3. Spring Data R2DBC .....	19

7.1.1.4. Spring Data REST .....	20
7.1.2. IoC Container .....	21
7.1.3. Dependency Injection .....	22
7.1.3.1. Constructor Injection .....	22
7.1.3.2. Setter Injection .....	22
7.1.4. Method Injection .....	23
7.1.5. Beans .....	24
7.1.5.1. Spring Inversion of Control (IoC) Container .....	24
7.1.5.2. Annotationen für Beans [1] .....	25
7.1.5.3. Scoping .....	25
7.1.5.3.1. Singleton .....	25
7.1.5.3.2. Prototype .....	26
7.1.5.3.3. Request .....	27
7.1.5.3.4. Session .....	27
7.1.5.3.5. Application .....	28
7.1.5.3.6. WebSocket .....	28
7.1.6. Aspect Oriented Programming (AOP) .....	29
7.1.7. Struktur für ein Projekt .....	30
7.1.8. OpenAPI UI .....	31
7.1.9. Komponenten einer Spring Anwendung .....	32
7.1.9.1. Controller .....	32
7.1.9.2. Entity .....	32
7.1.9.3. Dto .....	32
7.1.9.4. Mapper .....	32
7.1.9.5. Repository .....	32
7.1.9.5.1. Repository Queries .....	32
7.1.9.6. Service .....	32
7.1.10. Application YML .....	33
7.1.11. Authentication .....	34
7.2. Jakarta EE .....	35
7.3. Lombok .....	36
7.4. Buildtools .....	37
7.4.1. Gradle .....	37
7.4.2. Maven .....	38
8. Frontend .....	40
8.1. Frameworks .....	40
8.1.1. React .....	40
8.1.2. Svelte .....	41
8.1.3. VueJS .....	42
8.1.4. Angular .....	43
8.2. Web Components .....	44
8.3. CSS .....	45
9. Debugging .....	46
9.1. Backend Debugging .....	46
9.2. Frontend Debugging .....	47
9.2.1. Browser DevTools .....	47
9.2.1.1. HTML/CSS Inspection .....	47
9.2.1.2. JavaScript Console .....	47
9.2.1.3. JavaScript Debugger .....	47

9.2.1.3.1. Source Map .....	47
9.2.1.4. Network Operations .....	48
9.2.2. IDE Debugging .....	49
9.2.3. Extensions .....	50
Quellenverzeichnis .....	52

## **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

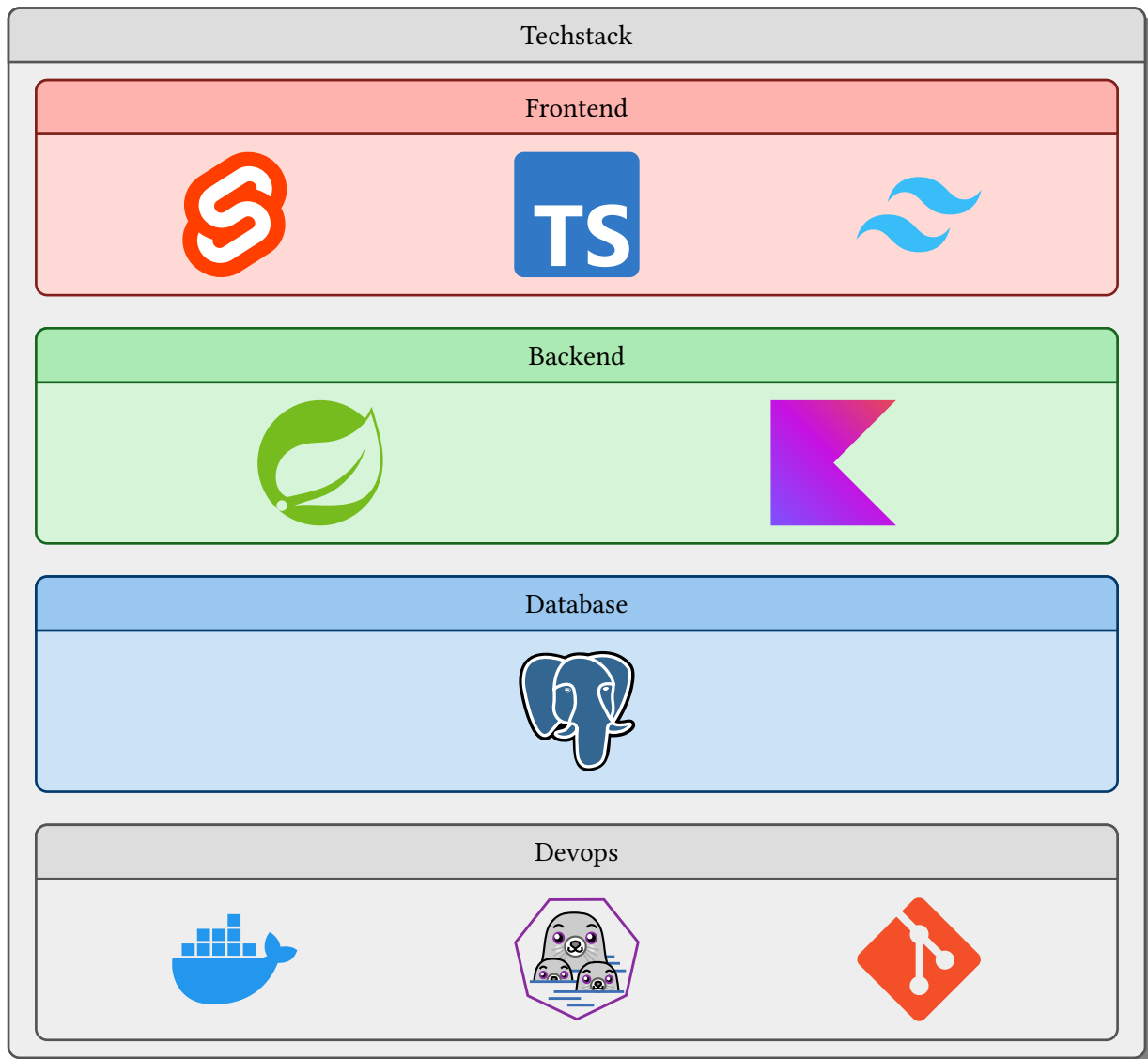
### **2.3. Präsentation bzw. Verteidigung**

- 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

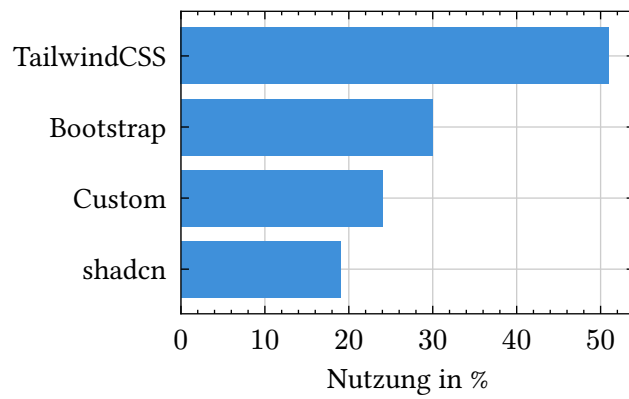
Wie der Name vermuten lässt, wird in einer Full Stack Anwendung jeder Aspekt von einem Tech Stack implementiert. Im Bereich der Web Entwicklung ist dies meist eine Kombination aus einem Backend und einem Frontend. Für das Backend kann auch noch eine Datenbank bereitgestellt werden. Welche Technologien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



#### 3.2. Svelte Übersicht

### 3.3. Tailwind CSS Übersicht

Die aktuelle Verteilung der CSS Frameworks nach dem State of CSS 2025 [2]:



### 3.4. 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. [3] 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. [4]

#### 3.4.1. 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 Anwendungsbereich von Spring Applikationen hat 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). [5]

#### 3.4.2. Springboot

Springboot ermöglicht die Erstellung von Spring Anwendungen, die mit minimaler Konfiguration lauffähig sind. Es ist dabei eine Erweiterung des Spring Frameworks. [3] 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. [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

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

#### 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. Verben

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. [8]

#### 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. [9]

#### 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). [10]

#### 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

[11]

#### 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. 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.

Status Code	Bedeutung
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.

Für jedes HTTP Verb ist ein Status Code vorgesehen, der bei Erfolg zurückgegeben wird:

- GET - 200 (OK)
- POST - 201 (CREATED)
- PUT - 200 (OK)
- DELETE - 204 (NO CONTENT)

#### **4.6. MVC**

#### **4.7. ORM**

## **5. Technologien**

### **5.1. Frontend**

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

### **5.2. Backend**

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

### **5.3. Database**

- PostgreSQL

### **5.4. Testing**

- Cypress, Playwright

### **5.5. DevOps**

- Docker
- Podman

## 6. User Stories

Kernpunkte einer User Story [12]:

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

**Template** [12]:

AS A {user|persona|system}

INSTEAD OF {current condition}

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

SO THAT {value of justification}

NO LATER THAN {best by date}

### 6.1. User Stories für das finale Projekt



## **7. Backend**

### **7.1. Spring**

#### **7.1.1. Spring Data**

##### **7.1.1.1. Spring Data JDBC**

#### **7.1.1.2. Spring Data JPA**

#### **7.1.1.3. Spring Data R2DBC**

#### **7.1.1.4. Spring Data REST**

### 7.1.2. 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 [13].

### 7.1.3. Dependency Injection

Ressource zu Dependency Injection [14]

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 [14].

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

#### Constructor oder Setter DI

Richtlinie 1

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.

#### 7.1.3.1. Constructor Injection

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

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

Kotlin

#### 7.1.3.2. Setter Injection

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

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

Kotlin

#### **7.1.4. Method Injection**

Ressource für Method Injection [15]

### 7.1.5. Beans

#### Beans

#### Definition 2

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. [16]

#### 7.1.5.1. 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. [16]

#### 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

#### 7.1.5.2. 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

#### 7.1.5.3. Scoping

##### 7.1.5.3.1. 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 [17].

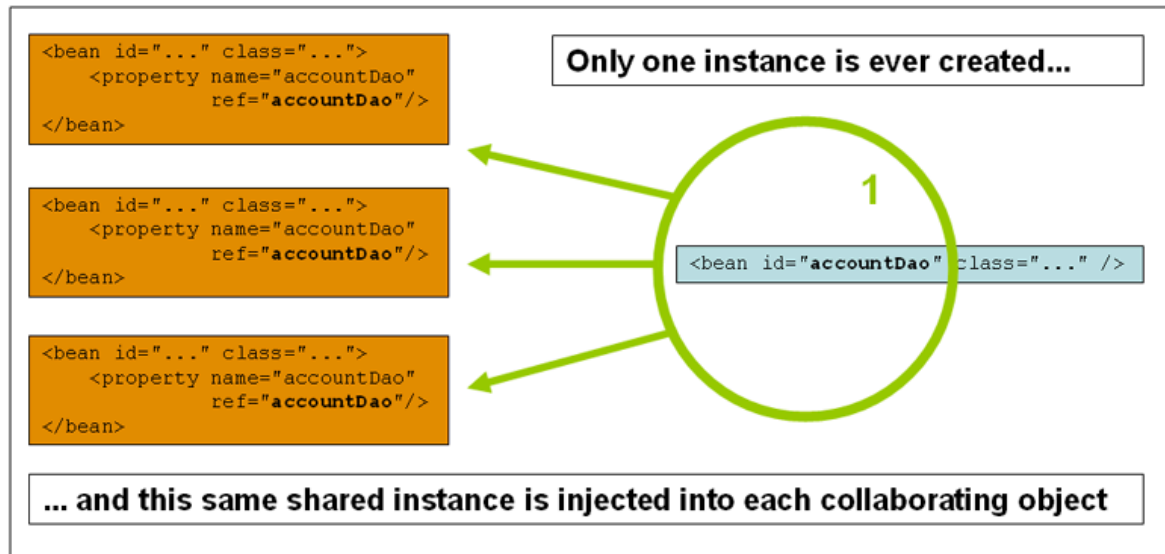


Figure 1: Funktionalität des Singleton Scopes

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

XML

#### Einsatz von Singleton Beans

Richtlinie 3

Singleton Beans sollten für stateless Beans eingesetzt werden.

#### 7.1.5.3.2. 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 [17].

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 [17].

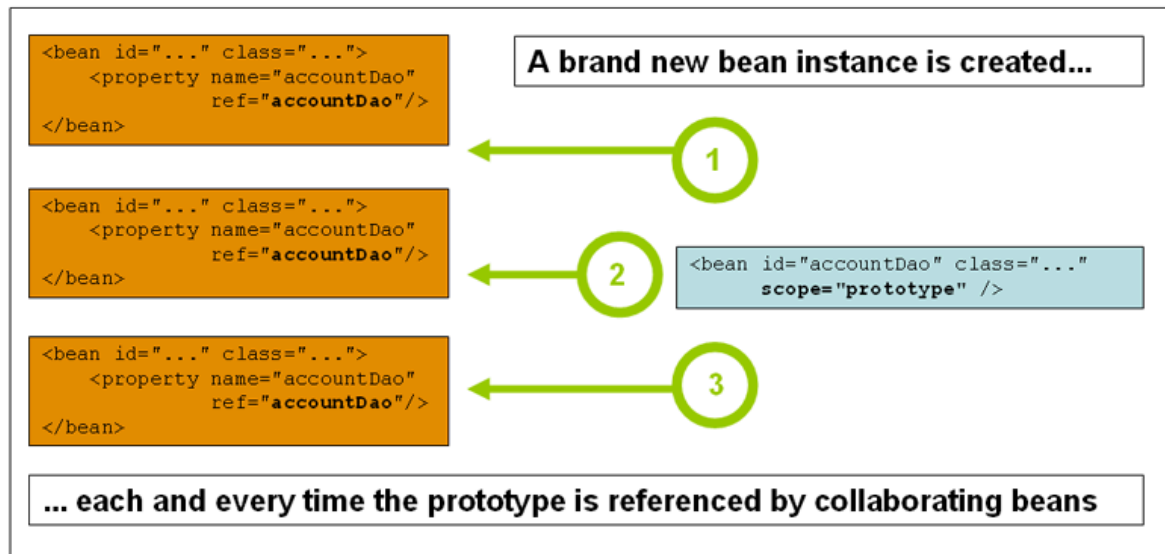


Figure 2: Funktionalität des Prototype Scopes

```

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

```

XML

#### Einsatz von Prototype Beans

Richtlinie 4

Prototype Beans sollten für stateful Beans eingesetzt werden.

#### 7.1.5.3.3. 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 [17].

```

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

#### 7.1.5.3.4. Session

Eine einzelne Instanz wird für jede HTTP Session erstellt [1]. Die erstelle 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 [17].

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

XML

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

Kotlin

#### 7.1.5.3.5. 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

[17]

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

XML

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

Kotlin

#### 7.1.5.3.6. WebSocket

Der WebSocket Scope ist an den Lebenszyklus einer WebSocket gekoppelt [17].

Weitere Informationen: <https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html> [18]

**TODO:** WebSocket Scope Kapitel ausbauen.

#### **7.1.6. Aspect Oriented Programming (AOP)**

### 7.1.7. 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

Wichtig 5

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.

### 7.1.8. 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>

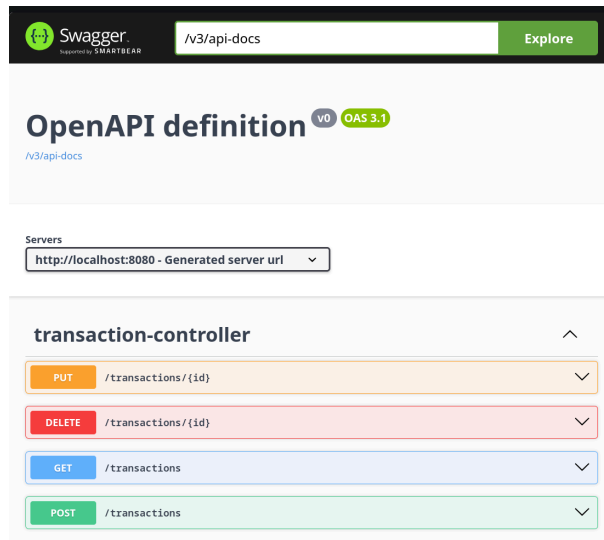


Figure 3: Swagger UI zum Darstellen und Testen der REST Endpunkte

## 7.1.9. Komponenten einer Spring Anwendung

### 7.1.9.1. 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

### 7.1.9.2. 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

### 7.1.9.3. Dto

```
1  data class EntityDto (
2      // data
3  )
```

◀ Kotlin

### 7.1.9.4. Mapper

```
1  fun convertEntityDtoToEntity(entityDto: EntityDto): Entity {
2      // convert
3  }
```

◀ Kotlin

### 7.1.9.5. Repository

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

◀ Kotlin

#### 7.1.9.5.1. Repository Queries

### 7.1.9.6. Service

```
1  @Service
2  class EntityService @Autowired constructor(
```

◀ Kotlin



```
3  private val entityReposirory: EntityReposirory
4  ) {
5    // service functions
6  }
```

#### 7.1.10. Application YML

#### **7.1.11. Authentication**

## **7.2. Jakarta EE**

### **7.3. Lombok**

## **7.4. Buildtools**

### **7.4.1. Gradle**

#### **7.4.2. Maven**



## **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. 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)

### 9.1. Backend Debugging

## 9.2. Frontend Debugging

### 9.2.1. Browser DevTools

# Firefox DevTools User Docs

### 9.2.1.1. HTML/CSS Inspection

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

### 9.2.1.2. JavaScript Console

- Log output lesen
- Interagieren mit der Website durch JavaScript

### 9.2.1.3. JavaScript Debugger

- Durchschreiten von JavaScript auf der Website

#### 9.2.1.3.1. Source Map

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

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

## JavaScript

**Generierung:**

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

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

JS JavaScript

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

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

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

☒ JSON

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

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

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

☒ JSON

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

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

#### 9.2.1.4. Network Operations

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



### 9.2.2. IDE Debugging

### 9.2.3. Extensions



## 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] "State of CSS 2025." [Online]. Available: <https://2025.stateofcss.com/en-US/other-tools/>
- [3] baeldung, "A Comparison Between Spring and Spring Boot." [Online]. Available: <https://www.baeldung.com/spring-vs-spring-boot>
- [4] "Spring Framework Overview." [Online]. Available: <https://docs.spring.io/spring-framework/reference/overview.html>
- [5] "History of Spring and the Spring Framework." [Online]. Available: <https://docs.spring.io/spring-framework/reference/overview.html#overview-history>
- [6] "Spring Boot." [Online]. Available: <https://spring.io/projects/spring-boot>
- [7] "What is REST?." [Online]. Available: <https://www.codecademy.com/article/what-is-rest>
- [8] "Method Definitions - GET." [Online]. Available: <https://httpwg.org/specs/rfc9110.html#GET>
- [9] "Method Definitions - POST." [Online]. Available: <https://httpwg.org/specs/rfc9110.html#POST>
- [10] "Method Definitions - PUT." [Online]. Available: <https://httpwg.org/specs/rfc9110.html#PUT>
- [11] "Method Definitions - DELETE." [Online]. Available: <https://httpwg.org/specs/rfc9110.html#DELETE>
- [12] Jim Schiel, "The Anatomy of a User Story." [Online]. Available: <https://resources.scrumalliance.org/Article/anatomy-user-story>
- [13] "Container Overview." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/basics.html>
- [14] "Dependency Injection." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html>
- [15] "Method Injection." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-method-injection.html>
- [16] Nguyen Nam Thai, "What Is a Spring Bean." [Online]. Available: <https://www.baeldung.com/spring-bean>
- [17] "Bean Scopes." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html#beans-factory-scopes-singleton>
- [18] "WebSocket Scope." [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html>