



Hochschule  
Zittau/Görlitz  
UNIVERSITY OF APPLIED SCIENCES

# **Web Engineering 3**

*Script*

# Inhaltsverzeichnis

|  |    |
|--|----|
| 1. Ablauf & Planung .....                                  | 4  |
| 2. Anforderungen .....                                     | 4  |
| 2.1. Projekt .....   | 4  |
| 2.2. Beleg .....   | 4  |
| 2.3. Präsentation bzw. Verteidigung .....                  | 4  |
| 3. Technologien .....                                      | 5  |
| 3.1. Frontend .....  | 5  |
| 3.2. Backend .....   | 5  |
| 3.3. Database .....  | 5  |
| 3.4. Testing .....   | 5  |
| 3.5. DevOps .....  | 5  |
| 4. User Stories .....                                      | 6  |
| 4.1. User Stories für das finale Projekt .....             | 6  |
| 5. Backend .....   | 7  |
| 5.1. Spring .....  | 7  |
| 5.1.1. IoC Container .....                                 | 7  |
| 5.1.2. Dependency Injection .....                          | 8  |
| 5.1.2.1. Constructor Injection .....                       | 8  |
| 5.1.2.2. Setter Injection .....                            | 8  |
| 5.1.3. Method Injection .....                              | 9  |
| 5.1.4. Beans .....   | 10 |
| 5.1.4.1. Spring Inversion of Control (IoC) Container ..... | 10 |
| 5.1.4.2. Annotationen für Beans [1] .....                  | 11 |
| 5.1.4.3. Scoping .....                                     | 11 |
| 5.1.4.3.1. Singleton .....                                 | 11 |
| 5.1.4.3.2. Prototype .....                                 | 12 |
| 5.1.4.3.3. Request .....                                   | 13 |
| 5.1.4.3.4. Session .....                                   | 13 |
| 5.1.4.3.5. Application .....                               | 14 |
| 5.1.4.3.6. WebSocket .....                                 | 14 |
| 5.1.5. Aspect Oriented Programming (AOP) .....             | 15 |
| 5.1.6. Struktur für ein Projekt .....                      | 16 |
| 5.1.7. OpenAPI UI .....                                    | 17 |
| 5.1.8. Komponenten einer Spring Anwendung .....            | 18 |
| 5.1.8.1. Controller .....                                  | 18 |
| 5.1.8.2. Entity .....                                      | 18 |
| 5.1.8.3. Dto .....   | 18 |
| 5.1.8.4. Mapper .....                                      | 18 |
| 5.1.8.5. Repository .....                                  | 18 |
| 5.1.8.5.1. Repository Queries .....                        | 18 |
| 5.1.8.6. Service .....                                     | 18 |
| 5.1.9. Application YML .....                               | 19 |
| 5.1.10. Authentication .....                               | 20 |
| 5.1.11. Gradle .....                                       | 21 |
| 6. Frontend .....  | 22 |
| 6.1. Frameworks .....                                      | 22 |
| 6.1.1. React .....   | 22 |

|            |                            |    |
|------------|----------------------------|----|
| 6.1.2.     | Svelte .....               | 23 |
| 6.1.3.     | VueJS .....                | 24 |
| 6.1.4.     | Angular .....              | 25 |
| 6.2.       | Web Components .....       | 26 |
| 6.3.       | CSS .....                  | 27 |
| 7.         | Debugging .....            | 28 |
| 7.1.       | Frontend .....             | 28 |
| 7.1.1.     | Browser DevTools .....     | 28 |
| 7.1.1.1.   | HTML/CSS Inspection .....  | 28 |
| 7.1.1.2.   | JavaScript Console .....   | 28 |
| 7.1.1.3.   | JavaScript Debugger .....  | 28 |
| 7.1.1.3.1. | Source Map .....           | 28 |
| 7.1.1.4.   | Network Operations .....   | 29 |
| 7.1.2.     | IDE Debugging .....        | 29 |
| 7.1.3.     | Debugging Extensions ..... | 29 |
|            | Quellenverzeichnis .....   | 30 |

# **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. Technologien**

### **3.1. Frontend**

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

### **3.2. Backend**

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

### **3.3. Database**

- PostgreSQL

### **3.4. Testing**

- Cypress, Playwright

### **3.5. DevOps**

- Docker
- Podman

## 4. User Stories

Kernpunkte einer User Story [2]:

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

**Template** [2]:

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}

### 4.1. User Stories für das finale Projekt

## 5. Backend

### 5.1. Spring

#### 5.1.1. 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 [3].

### 5.1.2. Dependency Injection

Ressource zu Dependency Injection [4]

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 [4].

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.

#### 5.1.2.1. 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

#### 5.1.2.2. 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



### **5.1.3. Method Injection**

Ressource für Method Injection [5]

#### 5.1.4. 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. [6]

##### 5.1.4.1. 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. [6]

##### 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

#### 5.1.4.2. 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

#### 5.1.4.3. Scoping

##### 5.1.4.3.1. 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 [7].

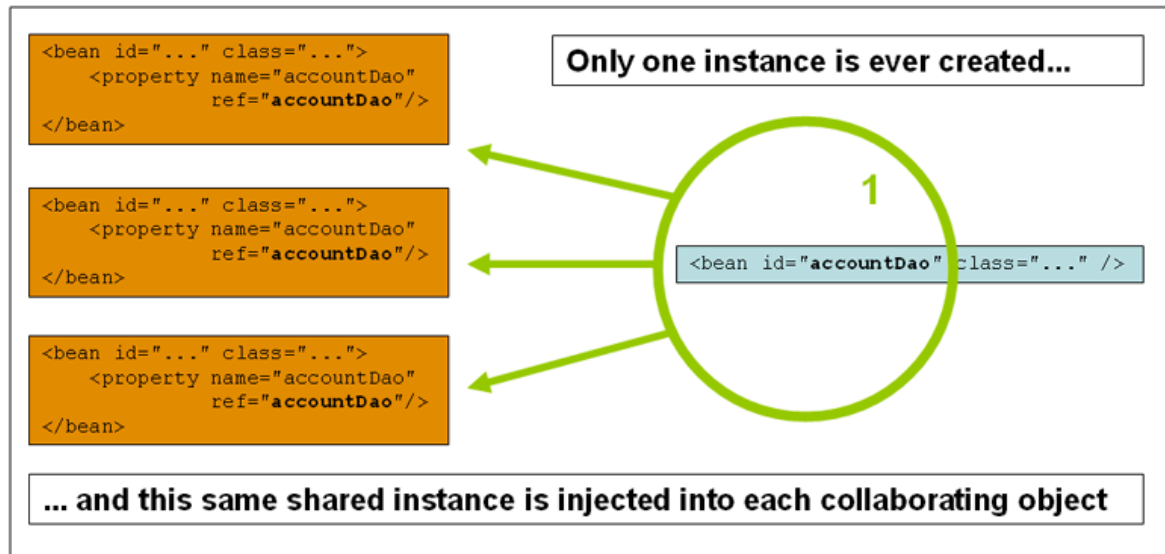


Figure 1: Funktionalität des Singleton Scopes

```

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.

#### 5.1.4.3.2. 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 [7].

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 [7].

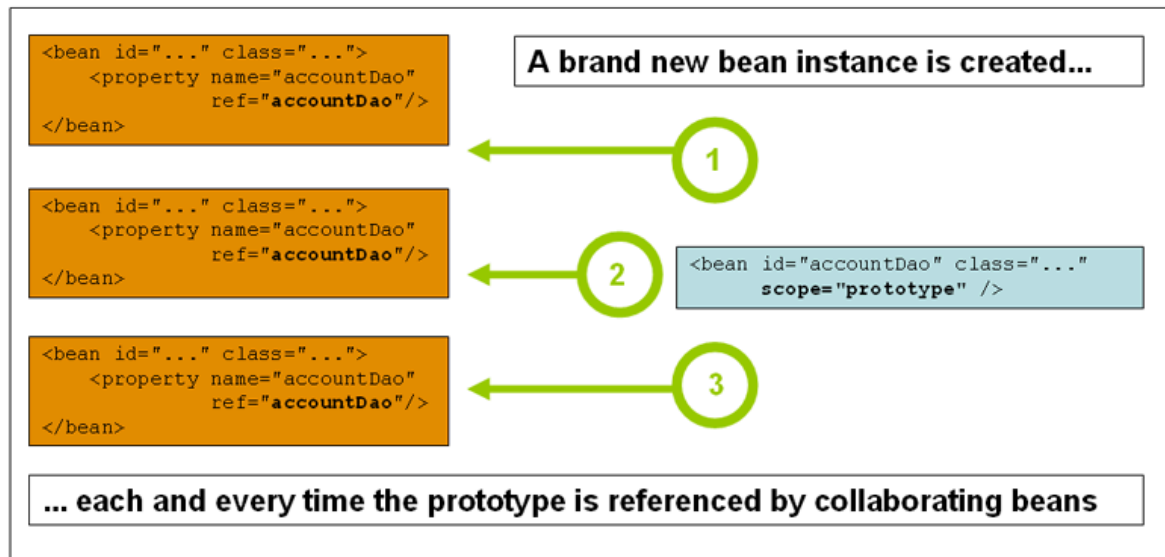


Figure 2: Funktionalität des Prototype Scopes

```

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.

##### 5.1.4.3.3. 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 [7].

```

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

##### 5.1.4.3.4. Session

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

Session aktiv ist. Andere Beans, vom gleichen Typ, die aber zu anderen HTTP Sessions gehören, werden die Änderungen nicht sehen. Wenn die HTTP Session beendet wird, wird auch die dazugehörige Bean entfernt [7].

```
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

#### 5.1.4.3.5. 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

[7]

```
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

#### 5.1.4.3.6. WebSocket

Der WebSocket Scope ist an den Lebenszyklus einer WebSocket gekoppelt [7].

Weitere Informationen: <https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html> [8]

**TODO:** WebSocket Scope Kapitel ausbauen.

### 5.1.5. Aspect Oriented Programming (AOP)

### 5.1.6. 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.



### 5.1.7. 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>

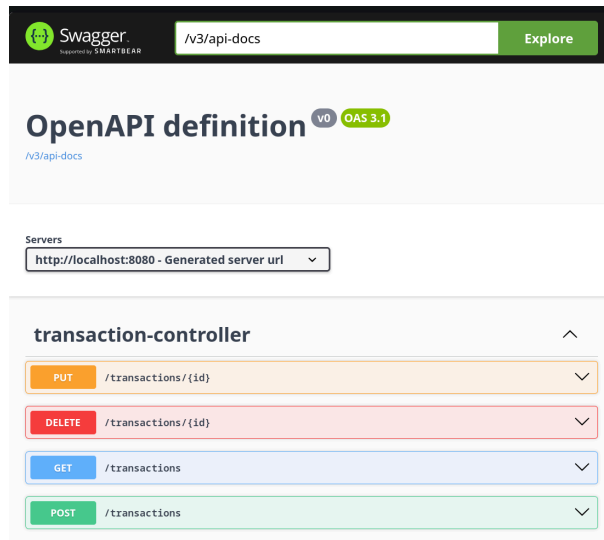


Figure 3: Swagger UI zum Darstellen und Testen der REST Endpunkte

## 5.1.8. Komponenten einer Spring Anwendung

### 5.1.8.1. 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

### 5.1.8.2. 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

### 5.1.8.3. Dto

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

◀ Kotlin

### 5.1.8.4. Mapper

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

◀ Kotlin

### 5.1.8.5. Repository

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

◀ Kotlin

#### 5.1.8.5.1. Repository Queries

### 5.1.8.6. Service

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

◀ Kotlin

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

#### 5.1.9. Application YML

#### **5.1.10. Authentication**

### **5.1.11. Gradle**

## **6. Frontend**

### **6.1. Frameworks**

#### **6.1.1. React**

### **6.1.2. Svelte**

### 6.1.3. VueJS



#### 6.1.4. Angular

## **6.2. Web Components**

### **6.3. CSS**

## 7. 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)

### 7.1. Frontend

#### 7.1.1. Browser DevTools

Firefox DevTools User Docs

##### 7.1.1.1. HTML/CSS Inspection

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

##### 7.1.1.2. JavaScript Console

- Log output lesen
- Interagieren mit der Website durch JavaScript

##### 7.1.1.3. JavaScript Debugger

- Durchschreiten von JavaScript auf der Website

##### 7.1.1.3.1. Source Map

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

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

JS JavaScript

#### Generierung:

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

```
1 // svelte.config.js
2 const config = {
3   compilerOptions: {
4     enableSourceMap: true
5   },
6 }
```

JS JavaScript

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

```
1 /* tsconfig.node.json */
2 {
3   "compilerOptions": {
4     "sourceMap": true
```

JSON

```
5  }  
6 }
```

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

```
1  /* angular.json */  
2  "projects": {  
3    "vite-project": {  
4      "architect": {  
5        "build": {  
6          "configuration": {  
7            "development": {  
8              "sourceMap": true  
9            }  
10         }  
11       }  
12     }  
13   }  
14 }
```

JSON

#### 7.1.1.4. Network Operations

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

#### 7.1.2. IDE Debugging

#### 7.1.3. Debugging 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] Jim Schiel, "The Anatomy of a User Story." [Online]. Available: <https://resources.scrumalliance.org/Article/anatomy-user-story>
- [3] "Container Overview." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/basics.html>
- [4] "Dependency Injection." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html>
- [5] "Method Injection." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-method-injection.html>
- [6] Nguyen Nam Thai, "What Is a Spring Bean." [Online]. Available: <https://www.baeldung.com/spring-bean>
- [7] "Bean Scopes." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html#beans-factory-scopes-singleton>
- [8] "WebSocket Scope." [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html>