

Web Engineering 3 - Script

Inhaltsverzeichnis

Web Engineering 3 - Script	1
Ablauf & Planung	2
Anforderungen	2
Projekt	2
Beleg	2
Präsentation bzw. Verteidigung	2
Technologien	2
Frontend	2
Backend	2
Database	3
Testing	3
DevOps	3
User Stories	3
User Stories für das finale Projekt	3
Backend	3
Springboot	3
Struktur für ein Projekt	3
OpenAPI UI	4
Controller	5
Entity	5
Dto	5
Mapper	5
Repository	6
Repository Queries	6
Service	6
Application YAML	6
Debugging	6
Frontend	6
Browser DevTools	6
HTML/CSS Inspection	6
JavaScript Console	6
JavaScript Debugger	6
Source Map	6
Network Operations	7
IDE Debugging	8
Debugging Extensions	8
Bibliography	8

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

Anforderungen

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)

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

Präsentation bzw. Verteidigung

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

Technologien

Frontend

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

Backend

- Tomcat Servlet zu Spring Beans (Aufbau)
- Spring
- Micronaut

- Quarkus
- Kotlin
- Gradle

Database

- PostgreSQL

Testing

- Cypress, Playwright

DevOps

- Docker
- Podman

User Stories

Kernpunkte einer User Story [1]:

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

Template [1]:

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}

User Stories für das finale Projekt

Backend

Springboot

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 1

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.

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>

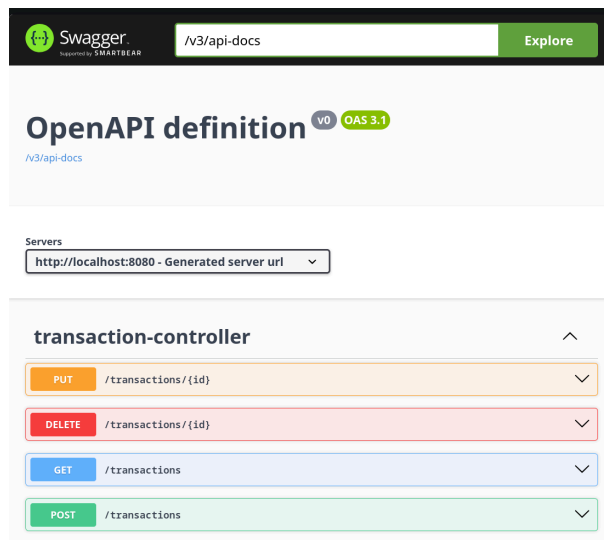


Figure 1: Swagger UI zum Darstellen und Testen der REST Endpunkte

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 }

```

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  }

```

Dto

```

1  data class EntityDto (
2      // data
3  )

```

Mapper

```

1  fun convertEntityDtoToEntity(entityDto: EntityDto): Entity {

```

```
2 // convert
3 }
```

Repository

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

◀ Kotlin

Repository Queries

Service

```
1 @Service
2 class EntityService @Autowired constructor(
3     private val entityReposirory: EntityReposirory
4 ) {
5     // service functions
6 }
```

◀ Kotlin

Application YML

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)

Frontend

Browser DevTools

Firefox DevTools User Docs

HTML/CSS Inspection

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

JavaScript Console

- Log output lesen
- Interagieren mit der Website durch JavaScript

JavaScript Debugger

- Durchschreiten von JavaScript auf der Website

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
5   }
6 }
```

 JSON

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

Network Operations

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

IDE Debugging

Debugging Extensions

Bibliography

[1] J. Schiel, “The Anatomy of a User Story.” ScrumAlliance.