

Web Engineering 3 - Script

Inhaltsverzeichnis

Web Engineering 3 - Script	1
Ablauf & Planung	3
Anforderungen	3
Projekt	3
Beleg	3
Präsentation bzw. Verteidigung	3
Technologien	3
Frontend	3
Backend	3
Database	4
Testing	4
DevOps	4
User Stories	4
User Stories für das finale Projekt	4
Backend	4
springboot	4
Beans	4
Spring Inversion of Control (IoC) Container	4
Annotationen für Beans [1]	6
Scoping	6
Singleton	6
Prototype	7
Request	7
Session	8
Application	8
WebSocket	9
Struktur für ein Projekt	10
OpenAPI UI	11
Komponenten einer Spring Anwendung	12
Controller	12
Entity	12
Dto	12
Mapper	12
Repository	12
Repository Queries	12
Service	12
Application YML	13
Debugging	13
Frontend	13
Browser DevTools	13
HTML/CSS Inspection	13
JavaScript Console	13
JavaScript Debugger	13
Source Map	13
Network Operations	14

IDE Debugging	14
Debugging Extensions	14
Bibliography	14

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 [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}

User Stories für das finale Projekt

Backend

springboot

Beans

Beans

Definition 1

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. [3]

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. [3]

Beispiel

```
1 public class Company {
2     private Address address;
3
4     public Company(Address address) {
5         this.address = address;
6     }
7 }
```

```
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 }
```

Traditionelle Erstellung der Abhängigkeiten:

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

Herangehensweise mit Beans

```
1 @Component
2 public class Company {
3     // this body is the same as before
4 }
```

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 }
```

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

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

Scoping

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 [4].

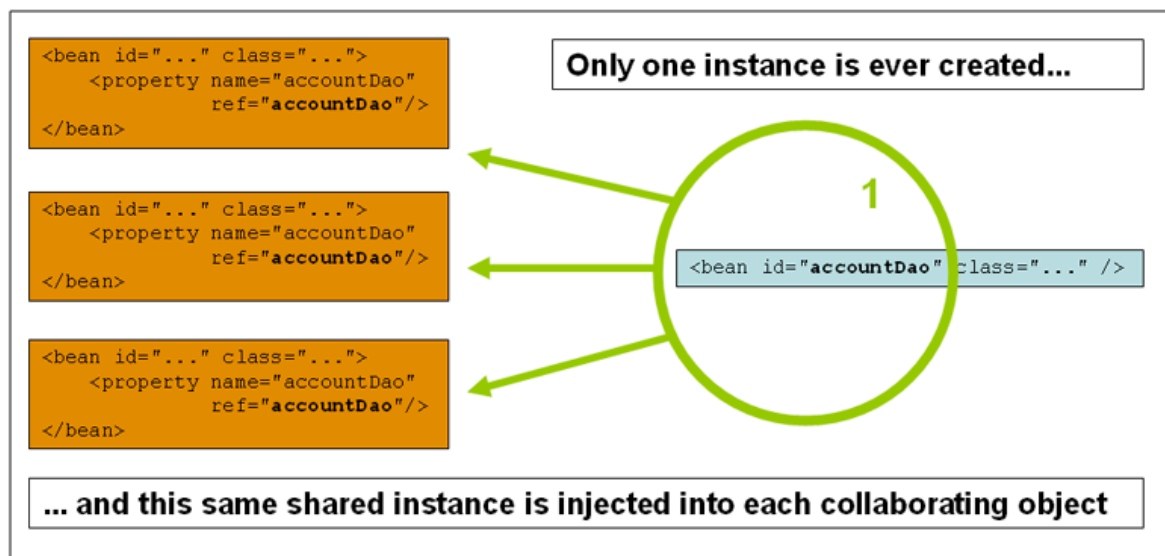


Figure 1: Funktionalität des Singleton Scopes

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

XML

Singleton Beans sollten für stateless Beans eingesetzt werden.

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 [4].

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 [4].

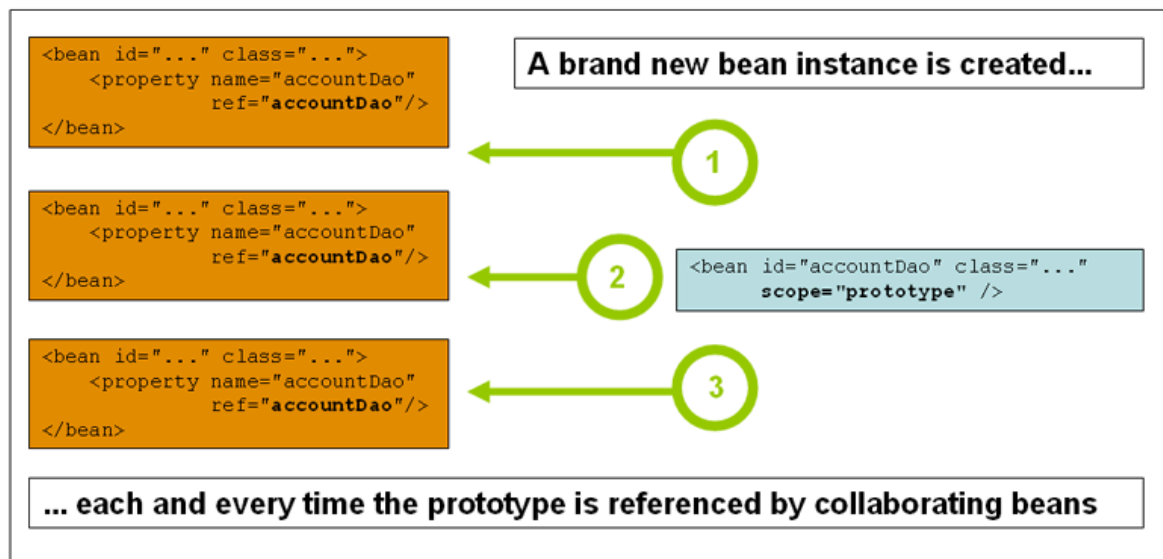


Figure 2: Funktionalität des Prototype Scopes

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

XML

Prototype Beans sollten für stateful Beans eingesetzt werden.

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 [4].

```
1 <bean
2   id="loginAction"
3   class="com.something.LoginAction"
```

XML

```
4     scope="request"
5 />
```

```
1 @RequestScope
2 @Component
3 class LoginAction {
4     // ...
5 }
```

 Kotlin

Session

Eine einzelne Instanz wird für jede HTTP Session erstellt [1]. Die erstellte 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 [4].

```
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

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

[4]

```
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

WebSocket

Der WebSocket Scope ist an den Lebenszyklus einer WebSocket gekoppelt [4].

Weitere Informationen: <https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html> [5]

TODO: WebSocket Scope Kapitel ausbauen.

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 4

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>

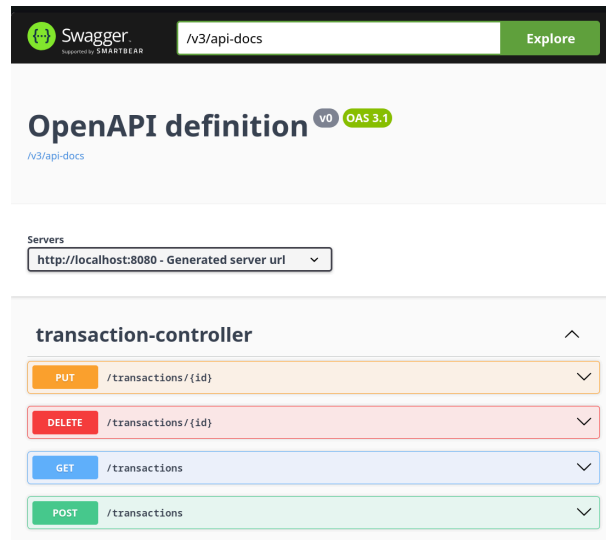


Figure 3: Swagger UI zum Darstellen und Testen der REST Endpunkte

Komponenten einer Spring Anwendung

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

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

Dto

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

◀ Kotlin

Mapper

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

◀ Kotlin

Repository

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

◀ Kotlin

Repository Queries

Service

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

◀ Kotlin

```

3  private val entityRepository: EntityRepository
4  ) {
5      // service functions
6  }

```

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] 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] Nguyen Nam Thai, "What Is a Spring Bean." [Online]. Available: <https://www.baeldung.com/spring-bean>
- [4] "Bean Scopes." [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html#beans-factory-scopes-singleton>
- [5] "WebSocket Scope." [Online]. Available: <https://docs.spring.io/spring-framework/reference/web/websocket/stomp/scope.html>