Matura POSxHIF 2023

Boigner Thomas, Cserich Philipp, Aslan Cemil (5BHIF)

Table of Contents

Ihr Auftrag!

Sehr geehrte Damen und Herren,

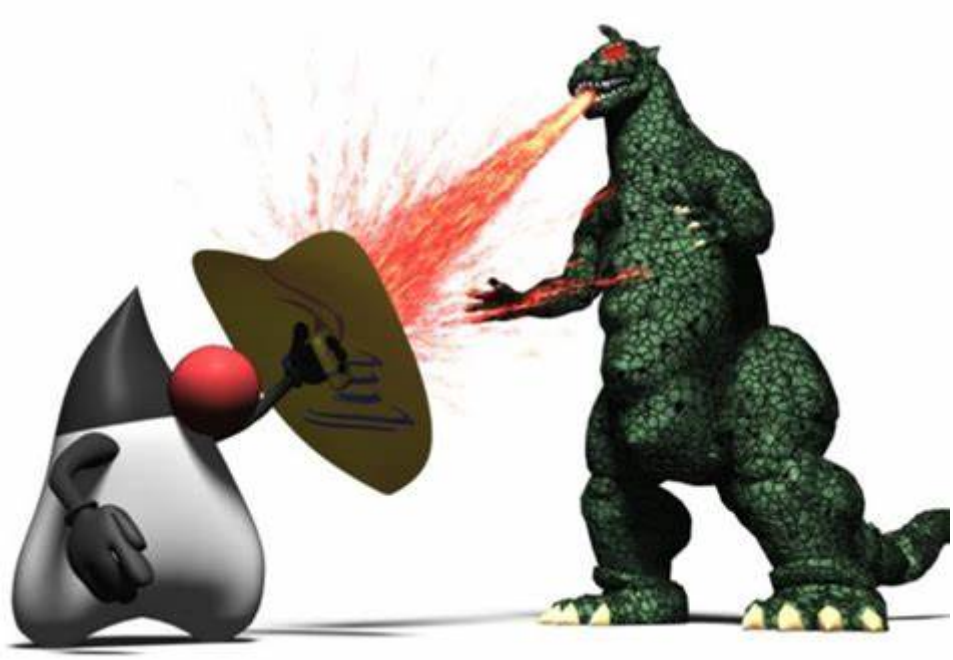
Dieses Dokument dient als Beilage zur Schriftlichen Matura der HTL Spengergasse für das Jahr 2023. Es wurde im Sinne der Vorbereitung für die schriftliche Reifeprüfung von Mitgliedern der 5BHIF erstellt, und dient Ihnen als Hilfestellung für den Antritt.

Ihre Mission startet JETZT!

Viel Erfolg bei der Matura!

(Das folgende Gerät wird in kürze selbstzerstört)

MfG, **Aslan Cemil, Boigner Thomas, Cserich Philipp**



Design Patterns

Creational Patterns

Singleton

Verwendung einer einzigen zentralen Instanz einer Klasse, um mehrere Replikate desselben Objekts zu vermeiden.

Factory

A factory is a class that creates objects. It can be used to create objects of a specific type, or to create objects of a specific type depending on the input parameters.

Abstract Factory

Eine Abstract Factory ist eine Factory, die andere Factories erzeugt. Sie kann verwendet werden, um Factories eines bestimmten Typs zu erstellen oder um Factories eines bestimmten Typs in Abhängigkeit von den Eingabeparametern zu erstellen.

Builder

Das Builder-Muster wird verwendet, um komplexe Objekte mit vielen Parametern zu erstellen. Es wird verwendet, um einen Konstruktor mit vielen Parametern zu vermeiden, der schwer zu lesen und nutzen ist, weshalb Parameter nach belieben weggelassen werden können, wenn man den Builder anstatt eines konstruktors nutzt. (Man erspart sich 50 unterschiedliche konstruktoren für jedes Instanzierungsszenario zu modellieren)

In Java wird das Builder-Muster durch eine einfache Annotation implementiert, die oben auf die Klasse geschrieben werden kann. Die Annotation heißt @Builder und ist Teil der Lombok-library.

Dependency Injection

Dependency Injection ist ein Entwurfsmuster, das zur Entkopplung von Komponenten in einer Softwareanwendung verwendet wird. Es ist ein Konzept, bei dem eine Klasse ihre Abhängigkeiten nicht selbst instanziert, sondern sie von einer externen Quelle erhält, z.B. durch Übergabe von Objekten als Parameter oder durch Aufruf von Methoden auf einem Injektions-Framework.

Der java equivalent nutzt die @Autowired annotation von SpringBoot.

Prototype

Das Entwurfsmuster Prototype ist ein Erstellungsmuster, mit dem Sie neue Objekte durch Kopieren oder Klonen bestehender Objekte erstellen können, anstatt sie von Grund auf neu zu erstellen.

Die Hauptidee hinter dem Prototype-Muster besteht darin, den Prozess der Objekterstellung zu abstrahieren und eine Möglichkeit zu bieten, neue Objekte durch Kopieren oder Klonen

bestehender Objekte zu erstellen, anstatt sie von Grund auf neu zu erstellen. Dies kann nützlich sein, wenn die Erstellung neuer Objekte teuer oder kompliziert ist und wenn Sie viele ähnliche Objekte erstellen müssen.

Bei diesem Muster wird zunächst ein Prototyp-Objekt erstellt, das als Vorlage für alle neuen Objekte dient. Anschließend werden neue Objekte erstellt, indem der Prototyp kopiert und seine Eigenschaften nach Bedarf geändert werden. Dieser Prozess der Erstellung neuer Objekte aus einem Prototyp wird als Klonen bezeichnet.

Es gibt zwei Möglichkeiten, das Prototypenmuster zu implementieren: Shallow copy und Deep copy.

- Bei der shallow copy wird ein neues Objekt erstellt und alle Felder des ursprünglichen Objekts in das neue Objekt kopiert. Wenn jedoch einige der Felder Verweise auf andere Objekte sind, werden diese Verweise ebenfalls kopiert, was bedeutet, dass sowohl das ursprüngliche Objekt als auch das neue Objekt die gleichen Verweise auf die anderen Objekte haben werden. Dies kann zu unerwartetem Verhalten führen, wenn eines der gemeinsam genutzten Objekte geändert wird.
- Bei der deep copy hingegen wird ein neues Objekt erstellt und alle Felder des ursprünglichen Objekts in das neue Objekt kopiert. Handelt es sich bei einem der Felder um einen Verweis auf ein anderes Objekt, werden auch diese Verweise kopiert, aber es werden auch neue Kopien der referenzierten Objekte erstellt. Auf diese Weise wird sichergestellt, dass das ursprüngliche Objekt und das neue Objekt über separate Kopien aller referenzierten Objekte verfügen.

Adapter

Ein Designpattern namens "Adapter" ermöglicht die Zusammenarbeit von Objekten mit unterschiedlichen Schnittstellen. Es wird verwendet, um die Schnittstelle einer bestehenden Klasse in eine andere Schnittstelle zu ändern, die vom Kunden erwartet wird, ohne den ursprünglichen Code zu ändern.

Wenn eine vorhandene Klasse über eine gewisse Funktionalität verfügt, aber eine andere Schnittstelle hat, als der Kunde erwartet, kann das Adaptermuster hilfreich sein. Die Erstellung einer neuen Klasse, die als Adapter zwischen dem Client und der alten Klasse dient, ist einer Änderung der bestehenden Klasse vorzuziehen, die sich gelegentlich als schwierig oder sogar unmöglich erweisen kann.

Die Adapterklasse implementiert die Schnittstelle, die der Client erwartet, und enthält eine Instanz der bestehenden Klasse. Wenn der Client eine Methode des Adapters aufruft, übersetzt der Adapter die Anfrage in ein Format, das die bestehende Klasse verstehen kann, und delegiert die Anfrage an die bestehende Klasse. Die Antwort der existierenden Klasse wird dann wieder in ein Format übersetzt, das der Client verstehen kann und vom Adapter zurückgegeben.

Es gibt zwei Arten von Adapter pattern:

Class Adapter pattern: Dieses Muster nutzt die Vererbung, um die Schnittstelle einer Klasse an eine andere anzupassen. Die Adapterklasse erweitert die bestehende Klasse und implementiert die vom Client erwartete Schnittstelle. Dadurch kann der Adapter die Implementierung der bestehenden Klasse wiederverwenden, während er gleichzeitig die vom Client erwartete Schnittstelle bereitstellt.

Object Adapter pattern: Dieses Muster verwendet Komposition, um die Schnittstelle einer Klasse an eine andere anzupassen. Die Adapterklasse enthält eine Instanz der bestehenden Klasse und implementiert die vom Client erwartete Schnittstelle. Wenn der Client eine Methode des Adapters aufruft, delegiert der Adapter die Anfrage an die bestehende Klasse.

Decorator

Das Decorator pattern, erlaubt einem Objekt dynamisch neue funktionalitäten zuzuordnen ohne das andere Objekte der gleichen Klasse davon betroffen sind. Es ist sinnvoll einzusetzen um beispielsweise das Single Resopnsibility Principle einzuhalten, da dieses Pattern es erlaubt, die funktionalität einer Klasse in mehrere Klassen aufzuteilen.

Facade

Eine Facade oder im deutschen auch Fassade ist ein Entwurfsmuster, welches eine 'front-facing' Schnittstelle im Programm darstellt, welche die eigentliche dahinterligende komplexität des Programs verstecken soll.

Strategy - Policy Pattern

Das strategy oder Policy pattern, definiert eine "Strategy" welche während der Runtime verändert werden kann. Dadurch ist der Prozess nicht bereits zu 100% im sourcecode vordefiniert sonder verändert sich durch die Anforderungen im laufenden Programm.

Iterator

Wird genutzt um einen container an elementen (List, Array usw.) durchzuiterieren und alle darin liegenden Elemente zu verarbeiten, verändern oder sonstiges.

Observer

Das Observer pattern, ist ein Entwurfsmuster, welches es erlaubt, dass ein Objekt (Subject) eine Liste von anderen Objekten (Observers) verwaltet, welche bei einer Änderung des Subjects benachrichtigt werden.

Visitor

Das Visitor pattern, ist ein Entwurfsmuster, welches es erlaubt, dass ein Objekt (Visitor) eine Liste von anderen Objekten (Elements) verwaltet, welche bei einer Änderung des Visitors benachrichtigt werden. (Umgekehrter Observer)

Software Principles

Open Closed Principle (OCP)

> "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"

Single Responisbiltiy Principle

> "A module should be responsible to one, and only one, actor."

Enums

```
1 public enum Test{
2     A(1),
3     B(2),
4     C(3)
5 
6     Test(Integer i){
7         this.i = i;
8     }
9 
10    private Integer i;
11 }
```

Während normale Enums meist zur Unterscheidung von vorgegebenen/statischen Werten verwendet werden, können sie durch Hinzufügen von Parametern über einen Konstruktor erweitert werden.

Stream types :

IntStream

- der IntStream ist ein Stream der zur Verarbeitung von Integer Werten dient dies kann zum Beispiel als ersatz für eine for Schleife dienen, wobei die Range der jeweiligen Werte durch folgende Syntax angegeben werden kann :

```
1 IntStream.range(0, 10).forEach(System.out::println);
```

Dieser Stream gibt die Werte von 0 bis 9 aus. Zur Verwendung des InstStreams für mappings anderer Objektklassen können unterschiedliche Methoden verwendet werden, wie zum Beispiel **mapToObj**. Dadurch kann der gestreamte wert innerhalb eines anderen Objektes verwendet bzw gespeichert werden oder eine methode wiederholt aufgerufen werden.

Stream

Streams können bei jeder Liste verwendet werden, wobei die Liste durch die Methode **stream()** in einen Stream umgewandelt wird. Dieser wird daraufhin sequentiell durchlaufen und kann durch verschiedene Methoden manipuliert werden.

Filter

```
1 int minGrade = 80;
2
3 List<Student> filteredStudents = students.stream()
4         .filter(s -> s.getGrade() >= minGrade)
5         .toList();
```

Collect

```
1 List<String> names = students.stream()
2         .filter(s -> s.getAge() > 18 && s.getState().equals("Bayern"))
3         .map(Student::getName)
4         .collect(Collectors.toList())
```

Collect is used to convert a stream into a collection. In this case, we are converting the stream into a new list.

Map

Mapping a new value onto an existing instance

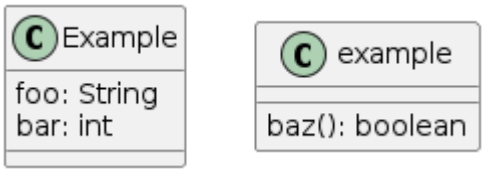
Domain Model

In diesem Kapitel schauen wir uns an wie ein Domain Model mittels UML dargestellt werden kann und wie es mittels JPA implementiert wird.

UML

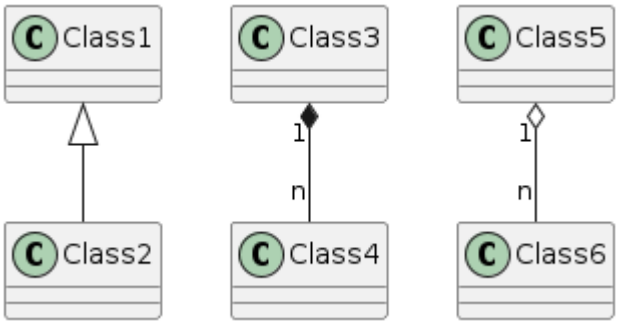
Mit Hilfe von UML können Klassendiagramm der Domain Klassen Dargestellt werden

Beispiel einer UML Klasse



In diesem Diagramm sieht man eine Klasse mit einer Variable foo vom Datentypen String, einer Variable bar vom Datentypen bar und eine Methode baz() die einen boolean zurück gibt.

Beziehungen zwischen Klassen



Die Beziehungen in diesem Diagramm können in drei Arten unterteilt werden:

Extension

Eine Extension relationship gibt an, dass eine Klasse von einer anderen erbt. In diesem Beispiel erbt Class1 von Class2 und erhält damit alle Attribute von Class2.

Aggregation

Eine Aggregation relationship gibt an, dass ein Child **alleine** ohne seinen Parent existieren kann. In diesem Fall kann Class4 auch ohne Class3 existieren, das heißt, dass wenn ein Objekt von Class4 gelöscht wird, das Objekt von Class3 weiter existieren. In dem Diagramm ist dann noch zusätzlich angegeben, dass Class3 mehrere Instanzen von Class4, aber Class4 nur eine Instanz von Class3 haben kann.

Composition

Eine Composition relationship gibt an, dass ein Child **nicht alleine** ohne seinen Parent existieren kann. In diesem Fall kann Class6 nicht ohne Class5 existieren, das heißt, dass wenn ein Objekt von Class5 gelöscht wird auch automatisch die mit ihm assoziierten Objekten von Class6 gelöscht werden. In dem Diagramm ist dann noch zusätzlich angegeben, dass Class5 mehrere Instanzen von

Class6, aber Class6 nur eine Instanz von Class5 haben kann.

Model

Entities

Entities können in einer Datenbank gespeichert werde und sind durch eine ID eindeutig gekennzeichnet.

@Entity: Diese Annotation makiert eine Klasse als Entity, damit JPA sie als solche erkennt.
@Table: Diese Annotation macht es möglich Attribute, wie den Namen oder das Schema der zu der Entity erstellten Datenbank Tabelle zu ändern.
@Id: Diese Annotation makiert die Variable als einzigartige Id der Entity und als Primary Key der erstellten Tabelle.
@Column: Diese Annotation macht es möglich Attribute, wie den Namen, die Länge, nullable- und unique constraints, der zu der Variable erstellten Datenbank column zu ändern.
@Transient: Variablen mit dieser Annotation werden nicht in der Datenbank gespeichert.

AbstractPersistable

Abstract Persistable ist eine Basis Klasse für Entities, die direkt eine Id implementiert.

Relationships

1 zu 1
Können über die @OneToOne Annotation angegeben werden. Auf einer Seite können wir nun mit @JoinColumn einen Foreign Key Column erstellen und auf der anderen mit mappedBy auf diese referenzieren. Außerdem kann mit dem cascade Parameter von @OneToOne das cascading angegeben werden.

1 zu n
Können mit @OneToMany auf der Seite die mehrere Referenzen und @ManyToOne auf der Seite die eine Referenz hat angegeben werden. Die Seite mit mehreren Referenzen muss diese zwingend in einer Collection speicher. Auf einer Seite können wir nun mit @JoinColumn einen Foreign Key Column erstellen und auf der anderen mit mappedBy auf diese referenzieren. Außerdem kann mit dem cascade Parameter von @OneToOne das cascading angegeben werden.

Value Objects

Value Objects können mit der @Embaddable annotation annotiert werden. Sie können dann mit @Embedded zu einer Entity hinzugefügt werden. Im Vergleich zu Entities haben Value Objects keine Id und es wird für sie keine extra Tabelle in der Datenbank erstellt. Die Werte der Value Objects werden einfach zu der Tabelle der Entitäten hinzugefügt die auf sie referenzieren. Sollten Value Objects in einer Collection gespeichert werden muss die @ElementCollection Annotation verwendet werden.

Enums

Um einen Enum Wert in einer Entity zu speichern wird die @Enumerated Annotation verwedent. Zusätzlich kann mit dieser Annotation noch angegeben werden welchen EnumType das Enum hat. Sollten Enums in einer Collection gespeichert werden muss die @ElementCollection Annotation verwendet werden.

Cemil Aslan

Persistence

Die Persistence-Schicht beinhaltet die Repositories für die jeweiligen Domains. Jede Domain hat seine eigene Repository. Spring Data kreiert ein neue Repository , überprüft er die Methoden die innerhalb des Interfaces deklariert wurden und erstellt nach der Methodenname eine Query.

Beispiel:

Wir haben eine Domain User mit den angebenen Attributen.

```
1 @Table(name="user-table")
2 @Entity
3 public class User extends AbstractPersistable<Long>
4 {
5 private String name;
6 private Integer age;
7 private String email;
8 private LocalDate birthDate;
9 private Boolean active;
10 }
```

Das Repository für User würde folgender Massen aussehen.

```
1 @Repository
2 public interface UserRepository extends JpaRepository<User, Long> {
3 }
```

Die wichtigsten Bausteine für das Repository sind das `@Repository` und das `extends JpaRepositroy<User,Long>`. Als Parameter werden `JpaRepository<Domain-class, Datatype of ID>` übergeben. Dannach kann man beliebige Methoden für das Repository deklarieren.

Wenn man IntelliJ Ultimate hat, hilft sie dir bei dem Aussuchen der Methodenname. Jedoch gibt es einige Punkte auf denen man achten muss.

Methoden in dem Repository

Die JpaRepository-Class bietet uns folgende Methoden an

Methodenname	Funktionalität
findAll()	gettet eine List von allen existierenden Entities in der Datenbank
findAll(..)	gettet eine List von existierenden Entities und sortiert sie nach den angegebenen Parametern
save(...)	speichert Entities in der Datenbank
flush()	Column 2
saveAndFlush()	Column 2, row 5
deleteInBatch()	Column 2, row 6

Bedingungskeywords

Equality

Equality kann auf mehreren Wegen geprüft werden:

```
1 List<User> findByName(String name); //wiedergibt User mit dem angebenen Namen
2
3 List<User> findByNameIs(String name); // macht genau dasselbe was oben steht, ist
  lesbarer
4
5 List<User> findByNameEquals(String name); // macht genau dasselbe was oben steht,
  ist lesbarer
6
7 List<User> findbyNameIsNot(String name);
```

Equality für booleans brauchen keine Parameter, wenn man sie wie gefolgt angibt:

```
1 List<User> findByActiveTrue();
2 List<User> findByActiveFalse();
```

Ähnlichkeit

Ähnlichkeit-Funktionen sind

- StartingWith(String prefix)
- EndingWith(String prefix)
- Containing(String prefix)

- Like(String pattern)

```
1 List<User> findByNameStartingWith(String prefix); // User deren Name mit Prefix
  beginnet
2
3 List<User> findByNameEndingWith(String suffix); // User deren Name mit Suffix endet
4
5 List<User> findByNameContaining(String infix); // User deren Name Infix beinhaltet
```

Es ist möglich mit einem bestimmten Namen-Pattern nach einem User zu queryn.

```
1 List<User> findByNameLike(String likePattern); // User deren Name in dem Pattern ist
```

```
1 String likePattern = "a%b%c";
2 userRepository.findByNameLike(likePattern);
```

Vergleich

```
1 List<User> findByAgeLessThan(int age); // wiedergibt User, die jünger als age sind
2
3 List<User> findByAgeLessThanEquals(int age); // wiedergibt User, die jünger oder
  gleich alt wie age sind
4
5 List<User> findByAgeGreaterThan(Integer age); // wiedergibt User, die älter als age
  sind
6
7 List<User> findByAgeGreaterThanEqual(Integer age); // wiedergibt User, die älter
  oder gleich alt wie age sind
```

Es ist auch möglich ein Altersbereich zu definieren

```
1 List<User> findByAgeBetween(Integer startAge, Integer endAge);
```

Ebenso ist es möglich eine Collection von Alter zu definieren, womit man dann die User bekommt, deren Alter in der Collection enthalten ist

```
1 List<User> findByAgeIn(Collection<Integer> ages);
```

Auch nach Datum kann gequeried werden:

```
1 List<User> findByBirthDateAfter(ZonedDateTime birthDate); // User, die nach
  birthDate auf die Welt gekommen sind
2
```

```
3 List<User> findByBirthDateBefore(ZonedDateTime birthDate); // User, die vor
  birthDate auf die Welt gekommen sind
```

Multiple Condition

Bei Multiple Condition sind die Keyword `And` und `Or` nötig

Entweder wird User nach seinem Namen gesucht, oder nach seinem Geburtstag oder nach beiden:

```
1 List<User> findByNameOrBirthDate(String name, ZonedDateTime birthDate);
```

Entweder wird der User nach seinem Namen oder nach seinem Geburtstag und Active, oder nach beiden gesucht:

```
1 List<User> findByNameOrBirthDateAndActive(String name, ZonedDateTime birthDate,
  Boolean active);
```

Sortieren der Ergebnisse

Um die Ergebnisse zu sortieren verwendet man OrderBy:

```
1 List<User> findByNameOrderByName(String name); // sortiert nach Namen
```

`ASC` ist default eingestellt und steht für aufsteigend sortiert:

```
1 List<User> findByNameOrderByNameAsc(String name)
```

`DESC` ist das Gegenteil von ASC und steht für absteigend sortieret:

```
1 List<User> findByNameOrderByNameDesc(String name);
```

Presentation

Der Presentation-Layer ist für die Rest-Request der Backend-Applikation verantwortlich.

Rest-Request:

Request	**Usage**
GET	Gibt JSON-Objects zurück
POST	Nimmt JSON-OBjects auf, wird hauptsächlich für erstellen von Daten verwendet
PUT	Datensatz updaten
Patch	teilweise updaten eines Datensatzes
DELETE	Datensatz löschen

Für die folgenden Erklärungen wird folgendes Entity verwendet:

```
1 @Table(name="user-table")
2 @Entity
3 public class User extends AbstractPersistable<Long>
4 {
5 private String name;
6 private Integer age;
7 private String email;
8 private LocalDate birthDate;
9 private Boolean active;
10 }
```

REST Controller in Spring Boot

Um Routes für unsere Backend-Applikation zu erstellen, bietet Spring Boot unterschiedliche Annotations für die Rest-Requests an

Annotation	Request
@GetMapping	GET
@PutMapping	Put
@PostMapping	POST
@PatchMapping	PATCH
@DeleteMapping	DELETE

Für die folgenden Beispiele erstellen wir eine RestController-Class:

```
1 @RestController
2 @RequestMapping("/api/user")
3 public class UserRestController {
4
5     private final UserService userService;
6
7 }
```

Die Annotation `@RestController` definiert, dass UserRestController eine RestController-Class ist und `@RequestMapping` definiert die Route für den RestController

GET-Controller

Die folgende Methode ist eine Simple-Get-Methode, welche alle User, die in der Datenbank sind als ein JSON-List zurück gibt.

```
1     @GetMapping("")
2     public HttpEntity<List<UserDto>> getAllUser(){
3         List<UserDto> userList = userService.getAllUser();
4         return (userList.isEmpty())
5                 ? ResponseEntity.noContent().build()
6                 : ResponseEntity.ok(userList);
7     }
```

Ebenso ist es möglich über den RestController einen Datensatz zu bekommen. Dies kann man verwirklich in dem man die Annotation `@PathVariable` verwendet

```
1     @GetMapping("{id}")
2     public HttpEntity<UserDto> getUserbyId(@PathVariable Long id){
```

```
3       UserDTO user = userService.getUserbyId(id);
4        return (user.isEmpty())
5                ? ResponseEntity.noContent().build()
6                : ResponseEntity.ok(user);
7    }
```

Durch `@RequestParam` können wir unsere GET-Methode verschärfern. Die folgende Methode hat den RequestParam includeResults, welches einen defaultValue no hat. Wenn value von includeResults yes ist, wiedergibt die Methode mehr infos über den User.

```
1 @GetMapping("/user/")
2     public HttpEntity<List<UserDto>> getUserStatistics(@RequestParam
  (value="includeResults", defaultValue = "no") String includeResults){
3        List<UserDto> userStatistics = null;
4        if(includeTestResults.equals("yes")){
5            userStatistics = userService.getDetailedUserStatistic();
6            return (userStatistics.isEmpty())
7                    ? ResponseEntity.noContent().build()
8                    : ResponseEntity.ok(userStatistics);
9        }
10       userStatistics = userService.getPersonStatistic();
11       return (userStatistics.isEmpty())
12               ? ResponseEntity.noContent().build()
13               : ResponseEntity.ok(userStatistics);
14    }
```

POST-Controller

```
1     @PostMapping({"/"})
2     public HttpEntity<UserDto> createUser(@RequestBody UserMutateCommand
  userCreateMutateCommand){
3        UserDto user = userService.createUser(userCreateMutateCommand);
4        return ResponseEntity.ok().body(userDto);
5     }
```

DELETE-Controlller

```
1     @DeleteMapping({"/{id}"})
2     public HttpEntity<UserDto> deleteUser(@PathVariable long id){
3        userService.deleteUser(id)
4        return ResponseEntity.ok().body(userDto);
5     }
```

PUT-Controller

```
1  @PutMapping("/{id}")
2     public HttpEntity<UserDto> replaceUser(@Valid @PathVariable Long id,@RequestBody
   MutateUserCommand mutateUserCommand){
3         return ResponseEntity.ok(new UserDto(userService.
   replaceUser(id,mutateUserCommand)));
4     }
```

PATCH-Controller

```
1  @PatchMapping({"id"})
2     public HttpEntity<UserDto> partiallyUpdateBilling(@PathVariable long id, @Valid
   @RequestBody MutateUserCommand command) {
3         return ResponseEntity.ok(new UserDto(userService.partiallyUpadte(id,
   command));
4     }
```

Service

Der Service Layer dient zum Managen der Business Logik, die in den Controllern nicht abgebildet werden kann. Kompliziertere Prozesse und Abläufe werden hier abgebildet bzw. berücksichtigt.

Dto

Data transfer object used to transfer data between layers in your program

Sollte als record umgesetzt sein und einen großteil der originaldaten des models enthalten. **KEINE** 1 zu 1 abbildung des models!

> Anderer Aufbau = WICHTIG und RICHTIG :D

```
1 public record TeamDto(String name, String description, LocalDate creationDate,
  List<Member> memberList) {
2     public TeamDto(Team team){
3         this(team.getName(),team.getDescription(),team.getCreationDate
  (),team.getMembers());
4         log.debug("TeamDto from Team : {} has been built !", team);
5     }
6 }
```

MutateCommand

Wird verwendet um die Daten vom Client zu erhalten und im Service in ein dto umzuwandeln. Form bzw Platzhalter für die Daten die zukünftig in das Model kommen sollen.

Path : Client → Controller → Service

Abstract

word explenations and examples

Sources

- [1] https://www.example.com
- [2] https://www.example.com
- [3] https://www.example.com