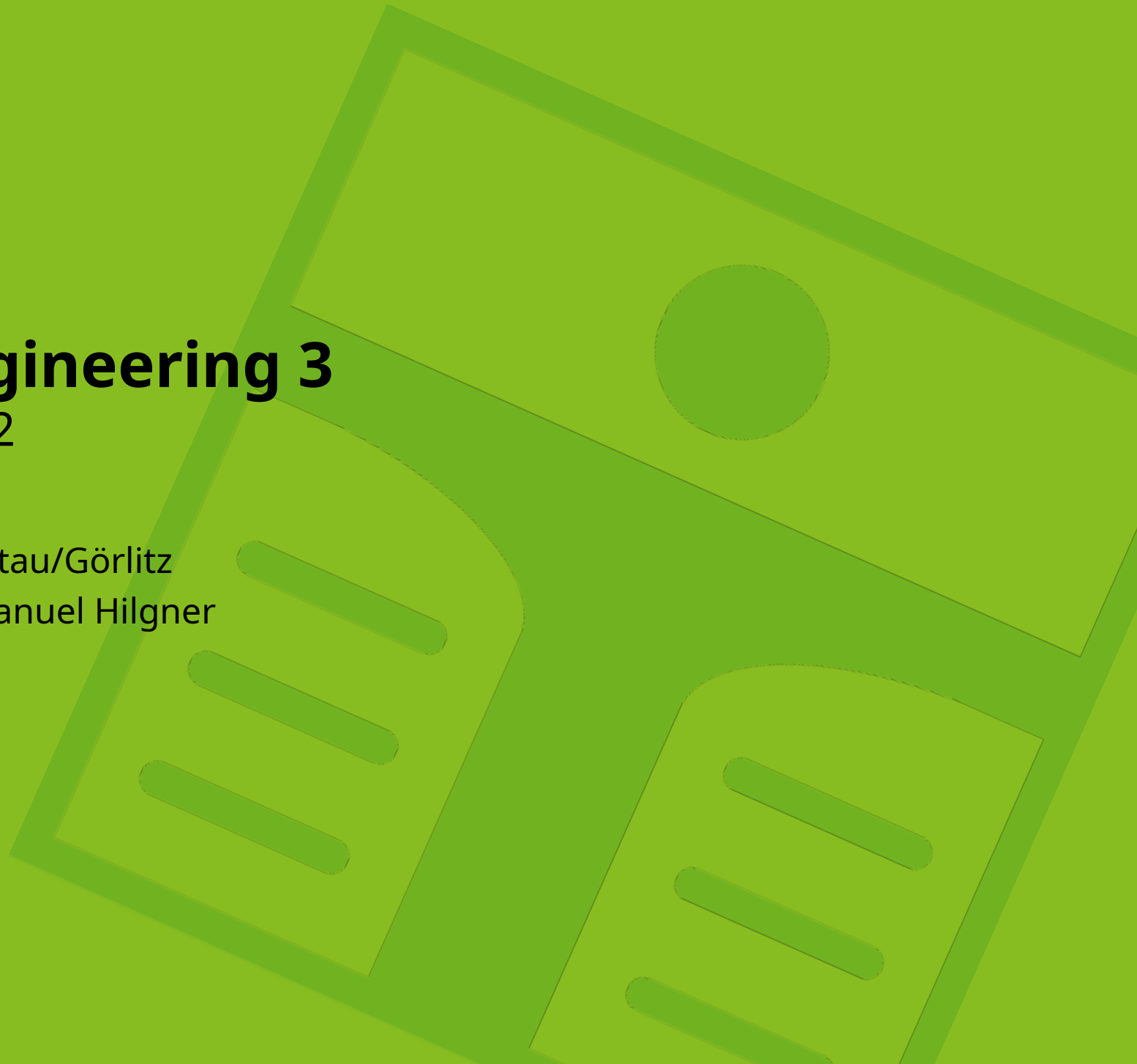


Web Engineering 3

Vorlesung 2

Hochschule Zittau/Görlitz

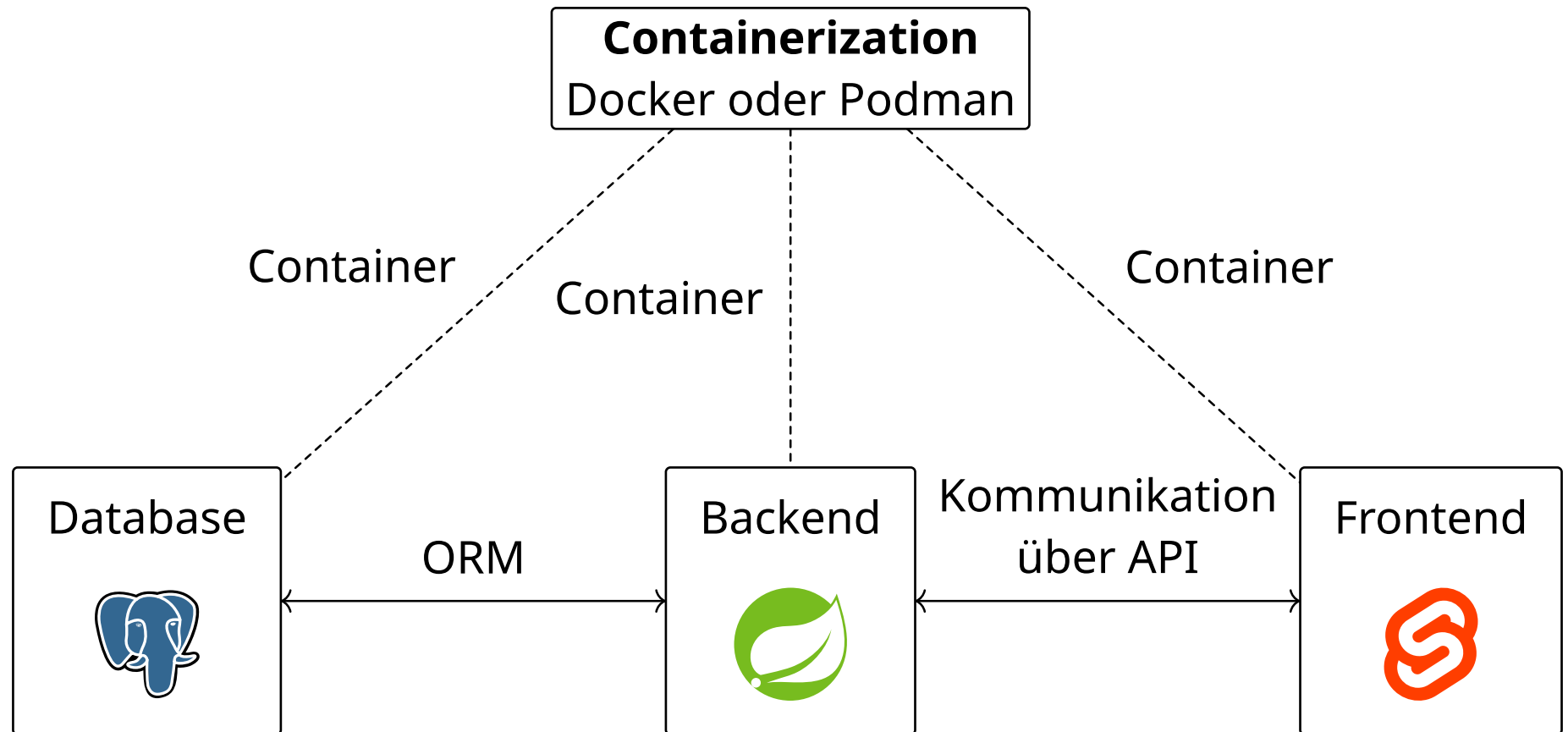
Christopher-Manuel Hilgner

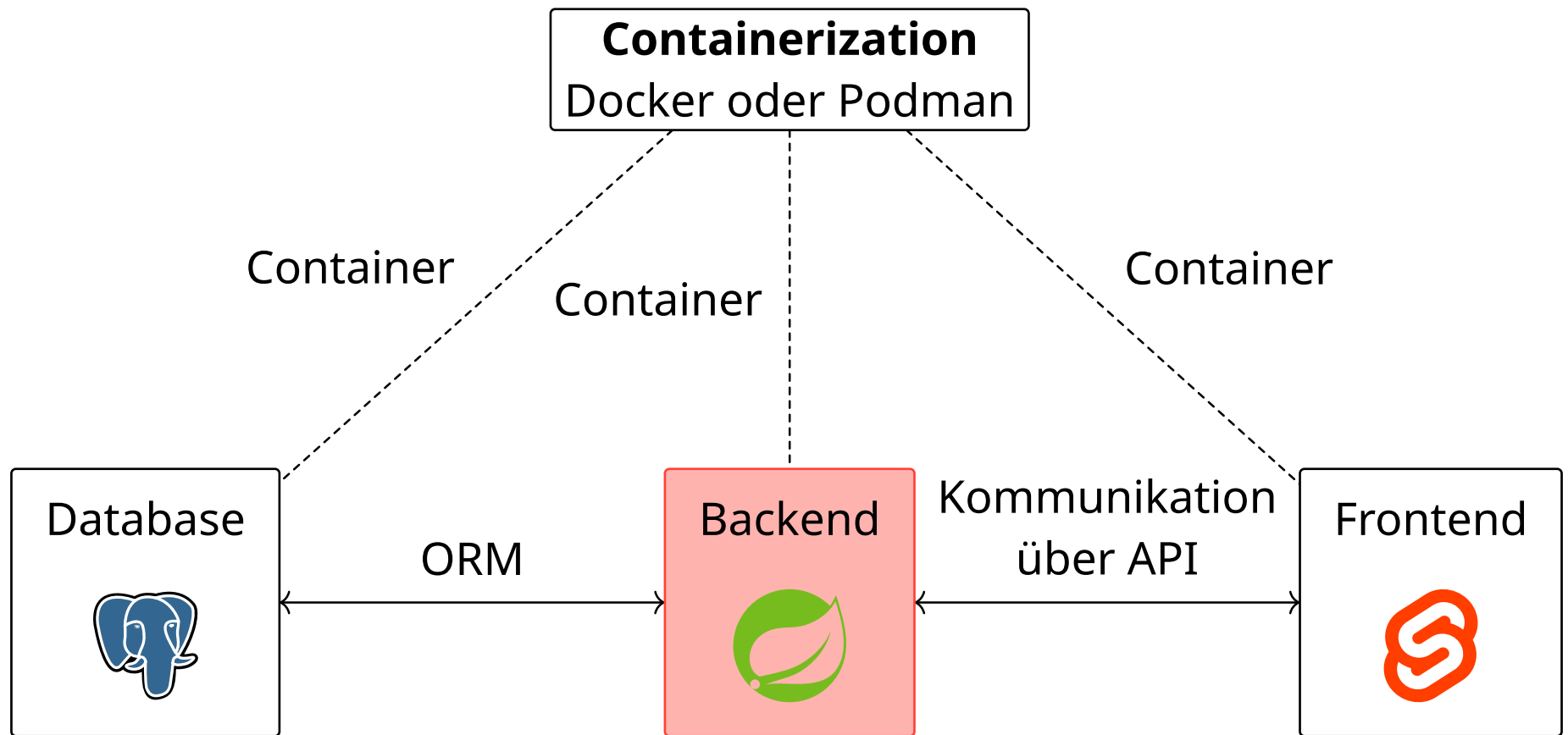


Agenda

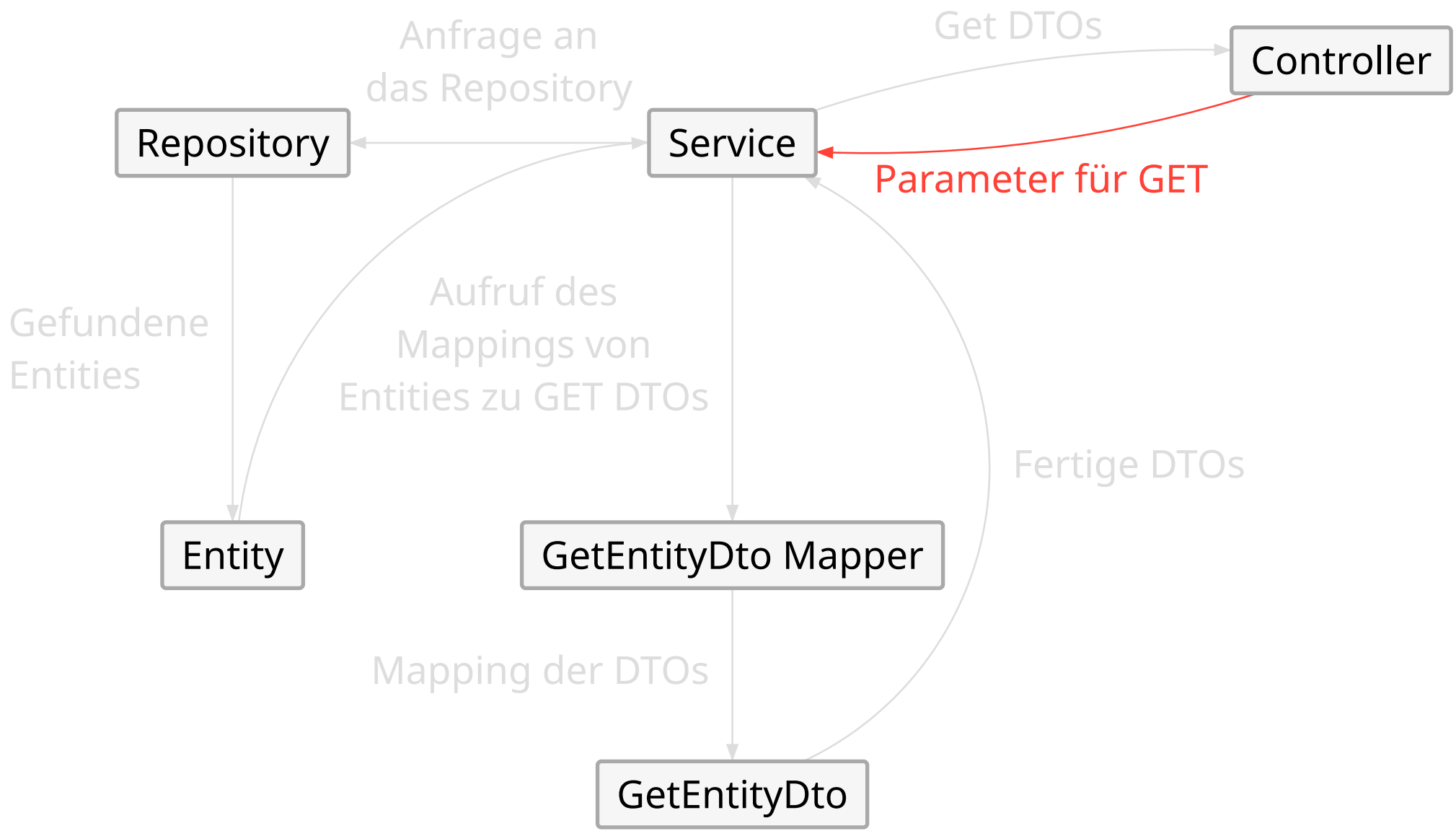
- Spring Komponenten
- Entity
- Repository
- Controller
- Service
- DTO
- Mapper

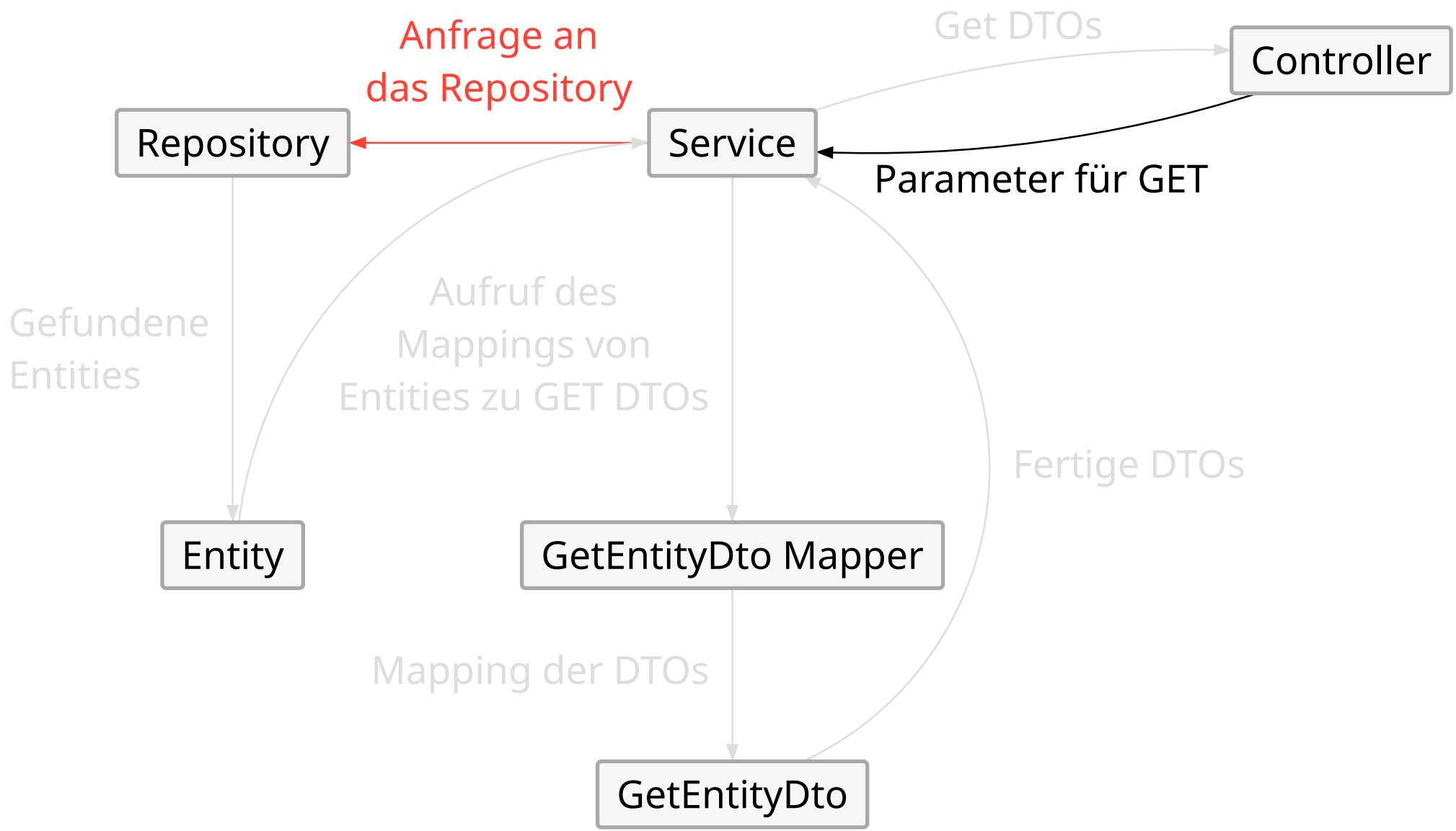
Letzte Woche

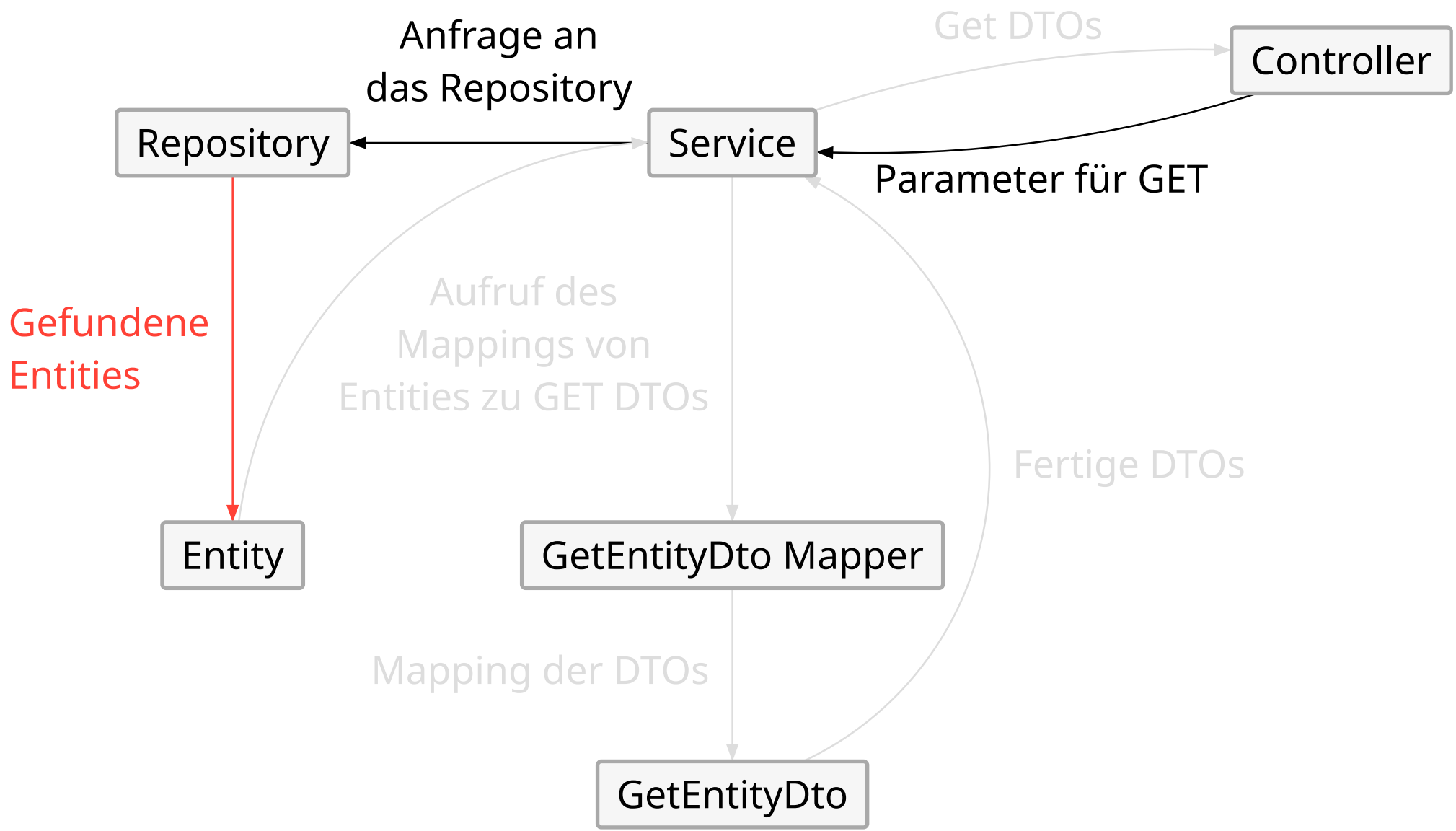


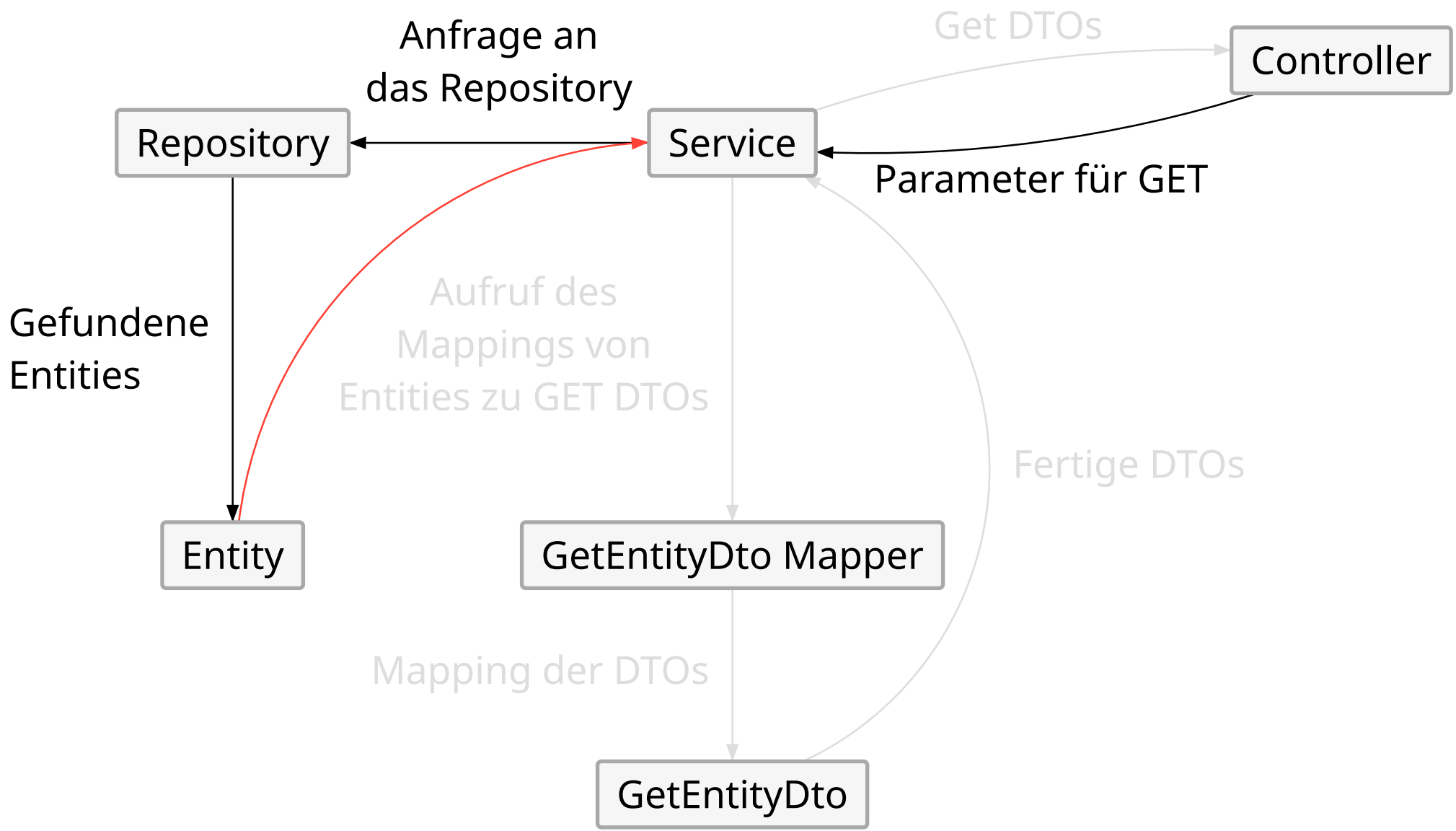


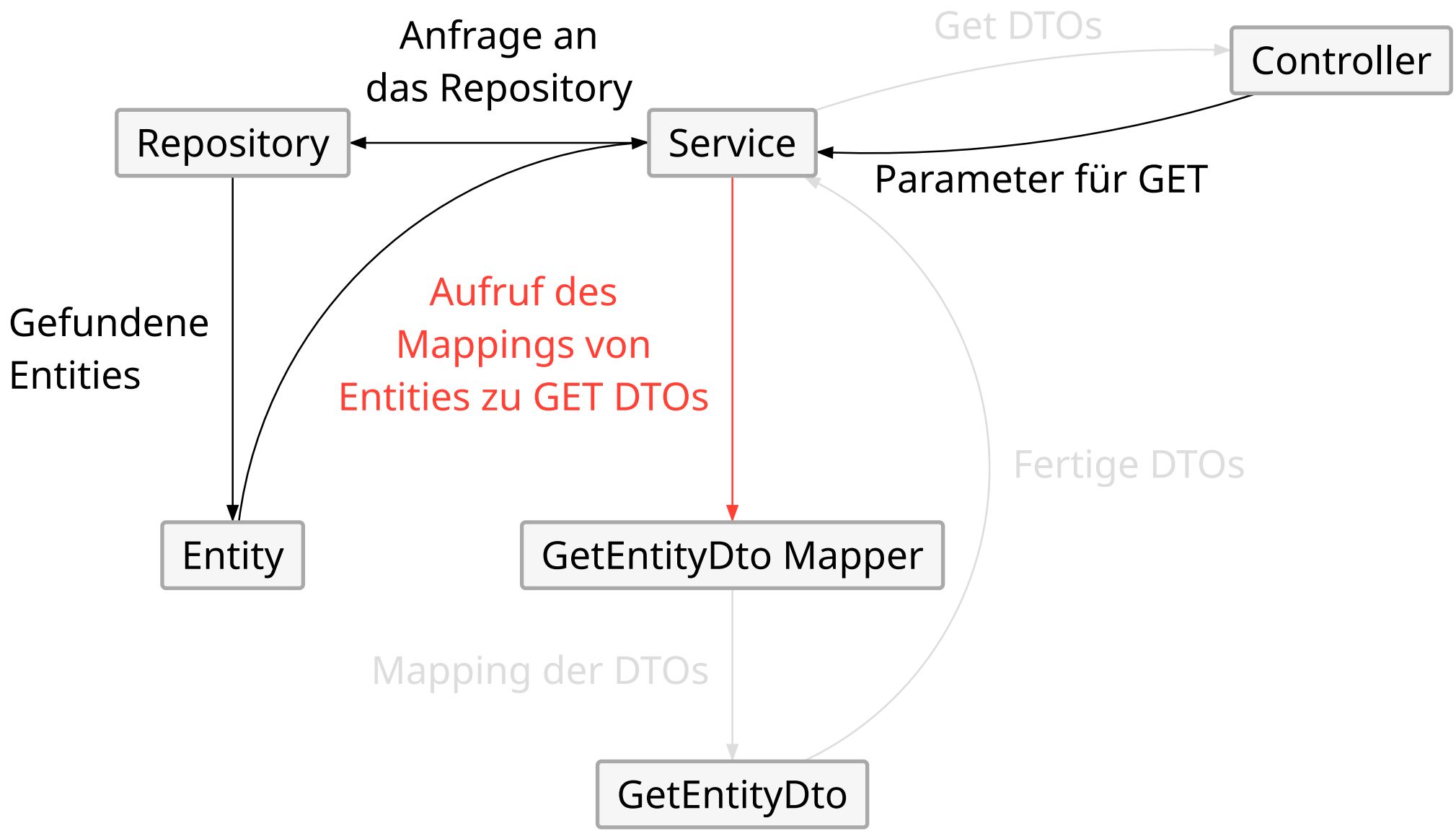
Komponenten und Ablauf einer GET Request

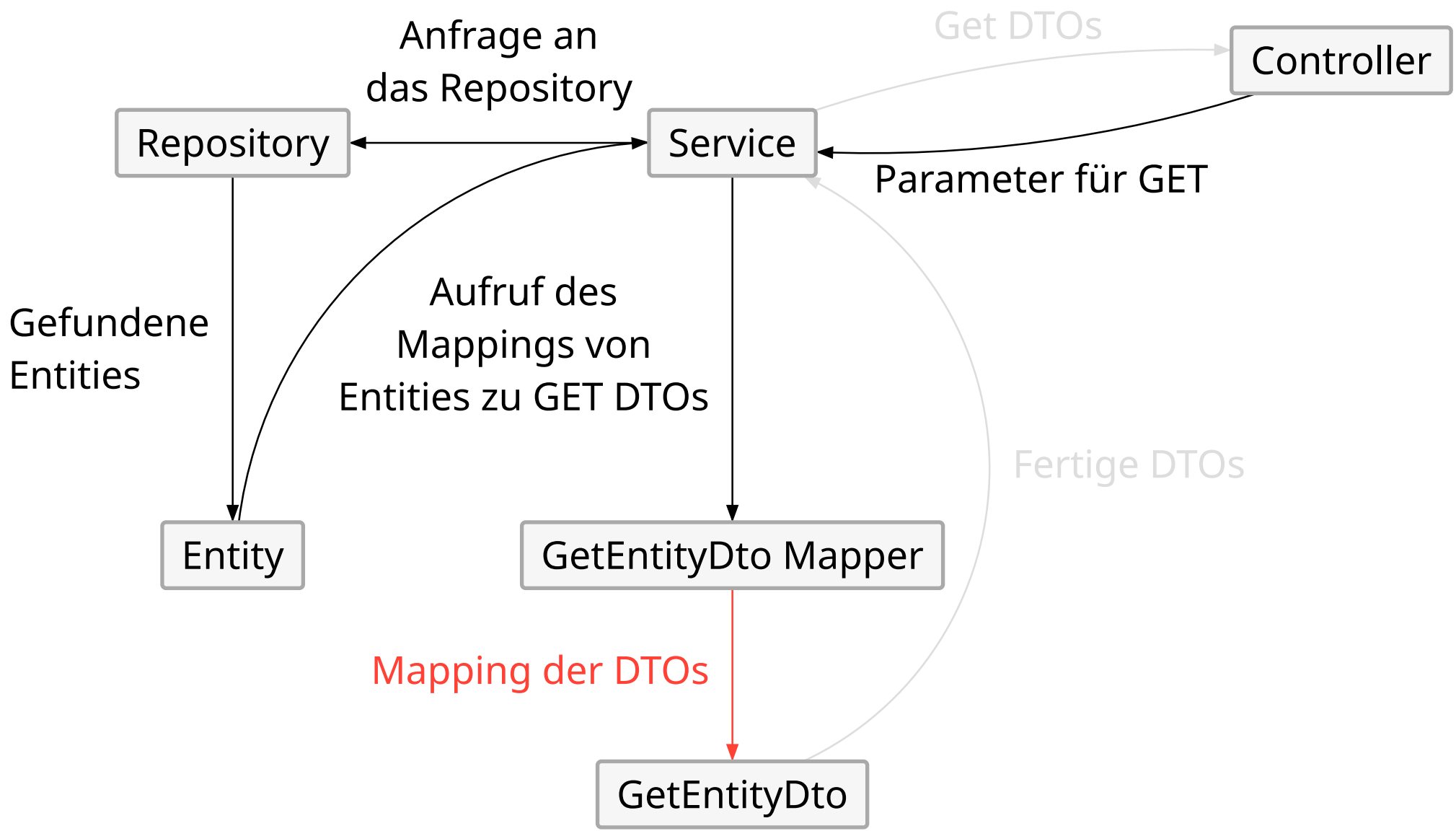


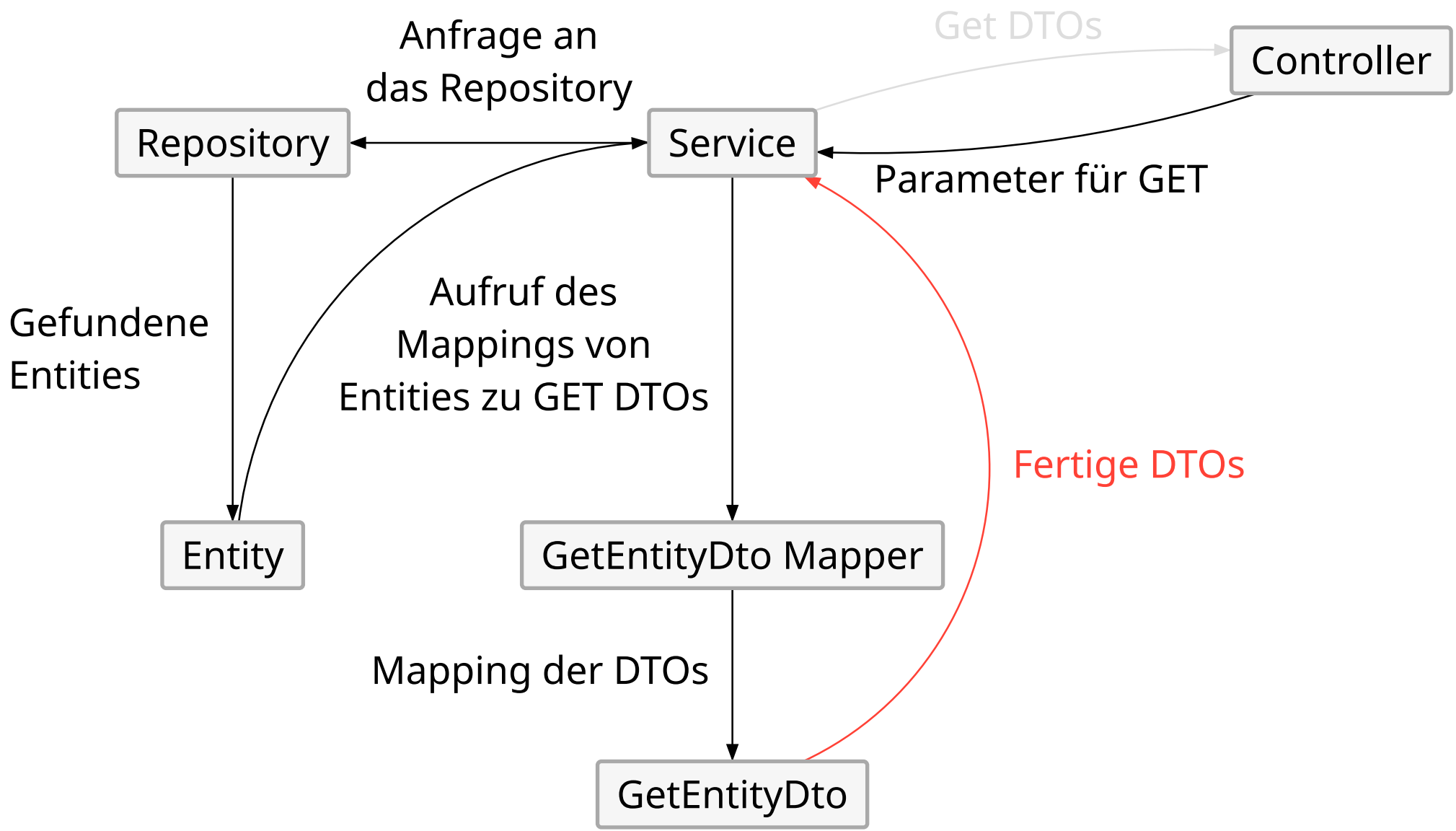


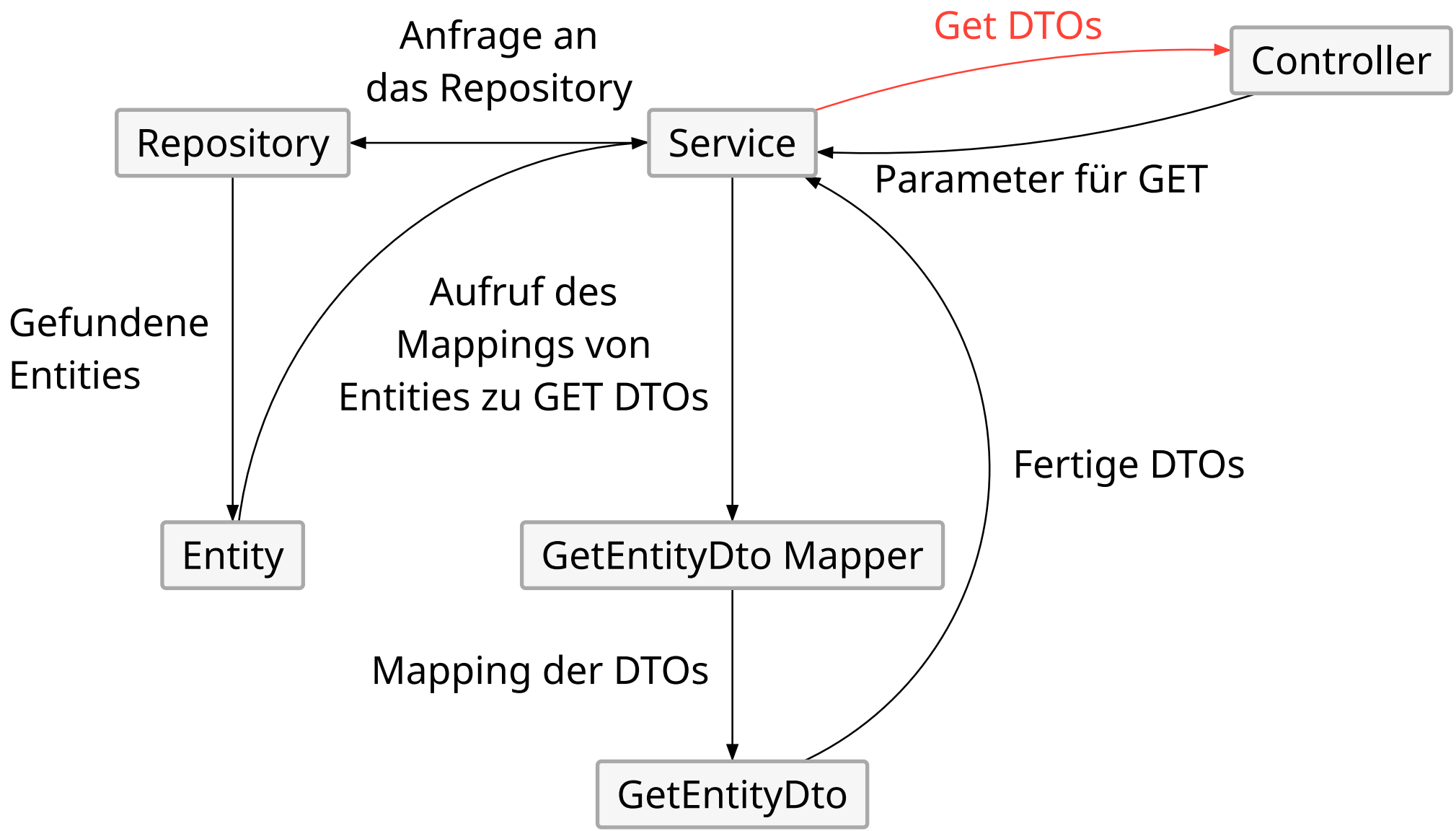












Komponenten

Entities

Repositories

Controllers

Services

DTOs

Mapper

Entities

Repositories

Controllers

Services

DTOs

Mapper

Entity

```
1  @Entity(name = "entityName")
2  @Table(name = "tableName")
3  class Entity {
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      val id: Long? = null,
7      // ...
8  }
```

[◀ Kotlin](#)

@Entity Annotation

- Klasse wird zu einer Entity, wenn sie mit @Entity annotated ist
- Optionaler Name kann angegeben werden
- Name wird in Queries verwendet
- Wenn kein Name vorhanden ist, wird der Klassenname verwendet

```
1 @Entity(name = "student")
```

◀ Kotlin

@Table Annotation

- Erlaubt definierung der Tabelle in der Datenbank
- Name und Schema können hier festgelegt werden
- Wenn @Table nicht vorhanden ist, wird eine Tabelle mit dem Entity Namen erstellt

```
1 @Table(name="STUDENT", schema="SCHOOL")
```

◀ Kotlin

@Id Annotation

- Jede Entity benötigt einen Primary Key
- Datenfeld mit @Id wird zum Primary Key
- Die Id wird automatisch generiert
- Strategie zur Generierung kann mit @GeneratedValue festgelegt werden
 - AUTO
 - TABLE
 - SEQUENCE
 - IDENTITY
- Mit AUTO wird sich eine der anderen Strategien automatisch ausgewählt

@Id Annotation

Id einer Entity muss als **null** definiert sein

Hinweis 1

Eine Id muss nullable sein und als default Wert null besitzen. Wenn das nicht der Fall ist, wird Spring beim Erstellen einer neuen Entity einen Error werfen. Grund hierfür ist, dass eine definierte Id Spring davon ausgehen lässt, dass es diesen Eintrag bereits in der Datenbank gibt. Wenn die Id null ist, weiß Spring, dass es sich hier um einen neuen Eintrag handelt.

@Id Annotation

```
1  @Id
2  @GeneratedValue(strategy = GenerationType.AUTO)
3  val id: Long? = null
```

◀ Kotlin

@Column Annotation

- Genauere Beschreibung von Datenfeldern
- Folgende Informationen können angegeben werden
 - name: Name des Columns in der Datenbank
 - length: Maximale Länge des Datenfeldes. Beispiel: Maximale Anzahl an Zeichen in einem String
 - nullable: Definiert, ob das Datenfeld null sein darf oder immer einen Wert besitzen muss
 - unique: Definiert, ob dieses Datenfeld ein unique Key sein soll

```
1 @Column(name = "dataField", nullable = false)
2 val dataField: Long = 0,
```

◀ Kotlin

Referenzen auf andere Tables

- Relationen zwischen Entities
- Beispiel: User Entities und Roles Entities. Jedem User werden Roles zugewiesen
- @OneToMany, @ManyToOne, @ManyToMany, @OneToOne

@OneToMany und @ManyToOne

- Ein User kann eine Role besitzen, aber eine Role kann auf mehrere User zugewiesen werden
- User hier das Child und Role der Parent
- Child ist der Besitzer der Relation
- ID von der Role wird als Foreign Key im User gespeichert


```
1  @Entity(name = "user")
2  @Table(name = "user")
3  class User (
4
5      @ManyToOne
6      @JsonBackReference
7      @JoinColumn(name = "role_id", referencedColumnName = "id")
8      var role: Role? = null
9
10 )
```

[◀ Kotlin](#)

```
1  @Entity(name = "role")
```

◀ Kotlin

```
2  @Table(name = "role")
```

```
3  class Role(
```

```
4
```

```
5      @OneToMany(
```

```
6          mappedBy = "role"
```

```
7      )
```

```
8      var users: List<User> = emptyList()
```

```
9
```

```
10 )
```

@OneToMany und @ManyToOne

- Liste mit Usern wird mit @OneToMany annotated
- mappedBy gibt Variable an, die in der User Klasse genutzt wird
- mappedBy zeigt, dass hier keine Foreign Keys verwaltet werden

Weitere Optionen:

- cascade: Operationen angeben, die alle Child Objects betreffen. Wenn keine angegeben sind, werden Child Objects nicht gelöscht, wenn der Parent gelöscht wird. Kann zu Problemen mit übrigen Foreign Keys führen
- orphanRemoval: Automatische Lösung von Child Objects, wenn sie von ihrem Parent getrennt werden.

```
1  @OneToMany(  
2      mappedBy = "role"  
3  )  
4  var users: List<User> = emptyList()
```

◀ Kotlin

@OneToMany und @ManyToOne

- role Variable in User Klasse wird mit @ManyToOne annotated
- @ManyToOne erschafft eine bidirektionale Relation
- Auch in User Klasse können Daten über Role abgerufen werden
- @JoinColumn auf der besitzenden Seite der Relation
- referencedColumnName nicht zwingend nötig. Primary Key kann automatisch gesucht werden

```
1  @ManyToOne
2  @JsonBackReference
3  @JoinColumn(name = "role_id", referencedColumnName = "id")
4  var role: Role? = null
```

◀ Kotlin

@OneToOne

- Verbindung zwischen zwei einzelnen Entities
- @JoinColumn auf der besitzenden Seite der Relation
- Nicht besitzende Seite besitzt mappedBy Attribute in @OneToOne

```
1  class User (◀ Kotlin  
2      @OneToOne  
3      @JsonBackReference  
4      @JoinColumn(name = "address_id", referencedColumnName = "id")  
5      var address: Address? = null  
6  )
```

```
1  class Address (◀ Kotlin  
2      @OneToOne(mappedBy = "address")  
3      var user: User? = null  
4  )
```

@ManyToMany

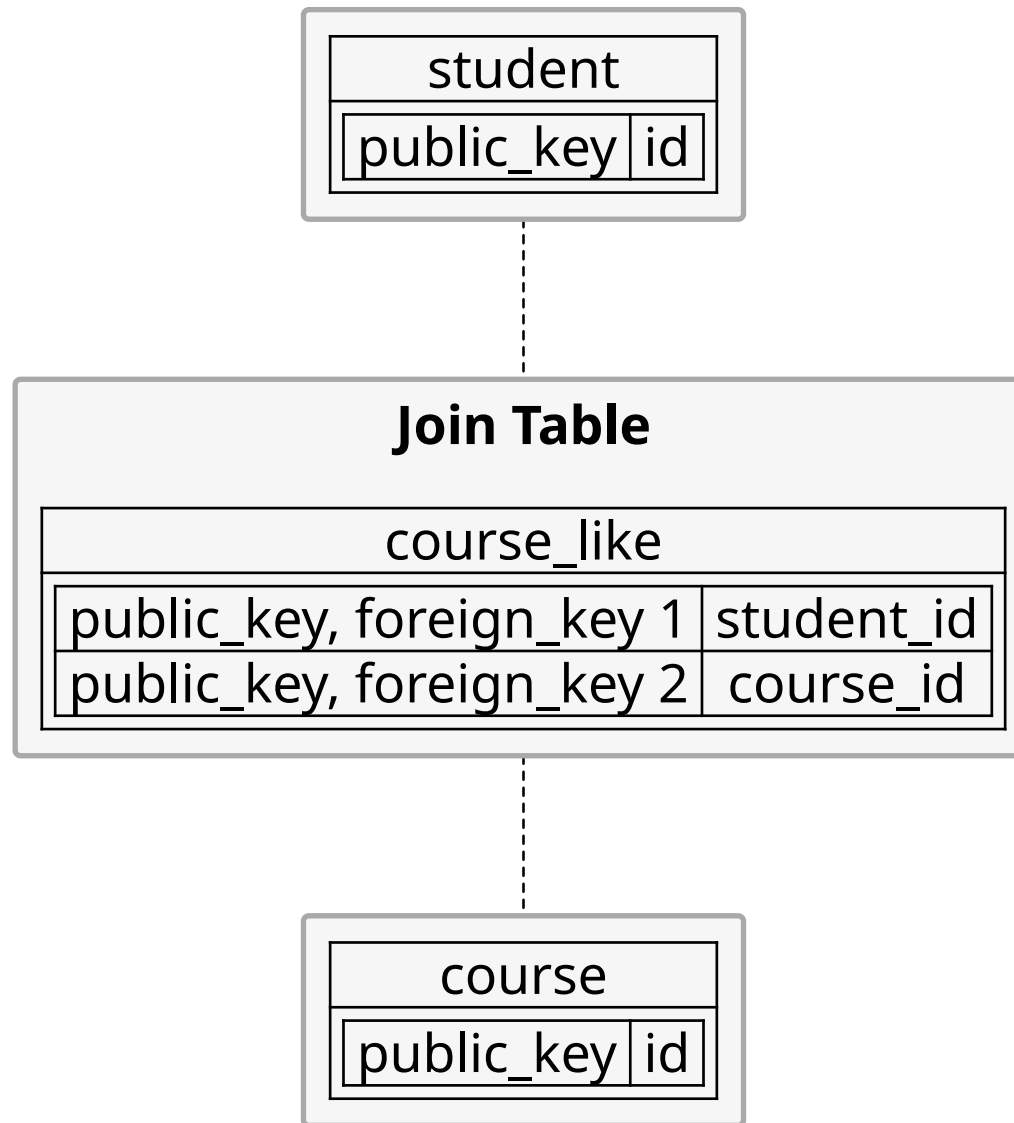


Figure 1: Aufbau einer Many to Many Relation mit einem Join Table

```
1 class Student (  
2     @ManyToMany  
3     @JoinTable(  
4         name = "course_like",  
5         joinColumns = @JoinColumn(name = "student_id"),  
6         inverseJoinColumns = @JoinColumn(name = "course_id")  
7     )  
8     val likedCourses: List<Course>  
9 )
```

◀ Kotlin

```
1 class Course (  
2     @ManyToMany(mappedBy = "likedCourses")  
3     val likes: List<Students>  
4 )
```

◀ Kotlin

@ManyToMany

- Bei Many to Many Beziehungen muss ein neuer Table angelegt werden
- Neuer Table enthält Foreign Keys aller Entities in der Relation
- Neuer Table ist ein Join Table
- Datenfelder auf beiden Seiten erhalten @ManyToMany Annotation
- Besitzende Seite enthält Definition für den Join Table mit @JoinTable
- Nicht besitzende Seite erhält mappedBy Attribut mit Variablennamen der besitzenden Seite

@ManyToMany - @JoinTable

- Enthält Name für den Join Table
- `joinColumns` enthält Verbindung zu der besitzenden Seite mit `@JoinColumn` und dem Namen der ID
- `inverseJoinColumns` definiert die nicht besitzende Seite der Relation mit `@JoinColumn` und dem Namen der ID
- `@JoinColumn` und `@JoinTable` sind nicht zwingend notwendig
- JPA kann Verbindungen selbst herstellen
- Namenskonventionen müssen dafür allerdings übereinstimmen und Strategie zur Verbindung kann nicht ausgesucht werden

Entities

Repositories

Controllers

Services


DTOs

Mapper

Repository

- Hauptinteraktionspunkt mit der Datenbank
- Wird erstellt mit @Repository Annotation in einem **Interface**
- Vererbung bestimmt Art des Repositories
- Unterschiedliche Repositories bringen unterschiedliche eingebaute Funktionen mit
- Repository benötigt Klasse mit ID Typ, die es verwalten soll

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



Operationen und Queries

- Einfach CRUD Operation werden direkt unterstützt
- Möglichkeiten, eigene SQL Queries zu verfassen oder generieren zu lassen:
 - Definierung einer neuen Methode im Interface
 - SQL Query bereitstellen über die @Query Annotation
 - Nutzung von Querydsl
 - Eigene Queries erstellen mit JPA Named Queries

Automatische Generierung von Queries

- Spring Data analysiert jede Methode in einem Repository
- Versucht aus Methodennamen eine Query zu erstellen
- Unterstützter Syntax ist dabei sehr breit (mehr im Seminar)

```
1  @Repository
2  interface UserInterface : JpaRepository<User, Long> {
3      fun findUserByName(name: String): MutableList<User>
4  }
```

◀ Kotlin

Manuelle Queries

- @Query Annotation erlaubt Erstellen von selbst geschriebenen Queries

```
1  @Query(  
2      "SELECT u FROM User u WHERE LOWER(u.name) = LOWER(:name)"  
3  )  
4  fun retrieveUserByName(  
5      @Param("name") name: String  
6  ): User
```

◀ Kotlin

Entities

Repositories

Controllers

Services

DTOs

Mapper

Controller

```
1  @RestController
2  @RequestMapping("/path/to/controller")
3  class Controller @Autowired constructor(
4      private val service: Service
5  ) {
6      @GetMapping
7      @ReponseStatus(HttpStatus.OK)
8      fun getEntities(): List<GetEntityDto> {
9
10     }
11 }
```

[◀ Kotlin](#)

Mappings

- @RequestMapping Annotation erlaubt das Mapping von Anfragen auf Methoden in der annotierten Klasse
- Unterschiedliche Parameter für die Zuordnung:
 - URL
 - Http Methode
 - Request Parameter
 - Header und Media Typen

```
1 @RequestMapping("/path/to/controller")
```

 Kotlin

HTTP Methoden Mappings

- Für HTTP Methoden gibt es Annotationen, die als Abkürzung von `@RequestMapping` dienen:
 - `@GetMapping`
 - `@PostMapping`
 - `@PutMapping`
 - `@DeleteMapping`
- Sie nutzen selbst `@RequestMapping`
- Annotationen existieren, da ein Controller die meisten seiner Methoden auf HTTP Methoden mappen sollte
- `@RequestMapping` würde jede beliebige HTTP Methode mappen

HTTP Methoden Mappings

Nutzen von mehreren @RequestMapping Annotationen

Hinweis 2

Es kann immer nur eine @RequestMapping an einem Element geben. Ein Element ist in diesem Fall eine Klasse, Methode oder ein Interface. Sollten dennoch mehrere Annotationen an einem Element vorhanden sein, wird nur die erste genutzt. Diese Regel gilt auch für Annotationen, die auf @RequestMapping basieren.

URI Patterns für Mappings

Beispiele:

- `"/resources/ima?e.png"` - Ein Zeichen wird abgeglichen
- `"/resources/*.png"` - Eine beliebige Anzahl an Zeichen wird abgeglichen
- `"/resources/**"` - Mehrere Pfad Segmente werden abgeglichen
- `"/projects/{project}/versions"` - Pfad Element wird abgeglichen und als Variable ausgelesen
- `"/projects/{project:[a-z]+}/versions"` - Pfad Element wird abgeglichen mit Regex und als Variable ausgelesen

Variablen in Pfaden

- Variablen können in Pfaden mit {} angegeben werden
- Mit @PathVariable können diese Variablen ausgelesen werden

```
1  @GetMapping("/owners/{ownerId}/pets/{petId}")
2  fun findPet(
3      @PathVariable ownerId: Long,
4      @PathVariable petId: Long
5  ): Pet {
6
7  }
```

◀ Kotlin

Variablen in Pfaden

- Pfad Variablen können bereits auf Klassenebene deklariert werden

```
1  @RestController
2  @RequestMapping("/owners/{ownerId}")
3  class OwnerController {
4      @GetMapping("/pets/{petId}")
5      fun findPet(
6          @PathVariable ownerId: Long,
7          @PathVariable petId: Long
8      ): Pet {
9
10     }
11 }
```

◀ Kotlin

Variablen in Pfaden

- Konvertierung zu den gewollten Typen findet automatisch statt
- Einfache Typen wie `int`, `long` oder `Date` werden unterstützt
- Eigene Konvertierungen können für komplexere Typen implementiert werden
- Wenn der Name im Pfad nicht mit dem Variablennamen übereinstimmt, kann ein Name angegeben werden

```
1 @PathVariable("customName")
```

◀ Kotlin

Entities

Repositories

Controllers

Services

DTOs

Mapper

Service

- Wird erstellt mit der @Service Annotation an einer Klasse
- Enthält Business Logik der Anwendung
- Typischerweise enthält der Service eine Funktion pro Controller Mapping
- Der Controller sollte nur diese Funktion aufrufen

```
1  @Service
2  class EntityService (
3      private val entityRepository: EntityRepository
4  ) {
5      // service functions
6  }
```

◀ Kotlin

Transactional Methoden

```
1  @Entity(name = "user")
2  @Table(name = "user")
3  class User(
4      @Id
5      @GeneratedValue(strategy = GenerationType.AUTO)
6      val id: Long? = null,
7      @OneToMany(
8          mappedBy = "user",
9          cascade = [CascadeType.REMOVE]
10     )
11     val roles: List<Role> = emptyList()
12 )
```

◀ Kotlin

Transactional Methoden

```
1  @Transactional
2  fun deleteUser(id: Long) {
3      val toDeleteUser = getUserById(id)
4      userRepository.delete(toDeleteUser)
5  }
```

◀ Kotlin

Transactional Methoden

- Manche Funktionen im Service werden mit `@Transactional` annotiert
- Annotation verlangt Nutzung des Spring Transaction Managements
- Sonst würde würden andere Persistence Teile der Anwendung die Transaktion durchführen
- `@Transactional` stellt sicher, dass alle Änderungen an der Datenbank erfolgreich waren, bevor die Änderungen eingefügt werden

Wann sollte `@Transactional` genutzt werden

- **Datenbank Operation mit mehreren Schritten**
 - Mehrere Schritte müssen Erfolgreich sein
 - Beispiel: Transaktion in einem Banksystem. Entferne Geld beim Sender und füge es beim Empfänger hinzu
- **Wenn die Operation kaskadierende Updates durchführt**
 - Operation beeinflusst Daten in anderen Tabellen
 - Beispiel: Löschen eines Users, der Konten besitzt, die auch gelöscht werden müssten
- **Wenn volatile Daten benutzt werden**
 - Fehler sind wahrscheinlich oder werden erwartet
- **Wenn bei Daten parallelität bzw. concurrency erwartet wird**
 - Mehrere Nutzer arbeiten parallel an Datenbank
 - `@Transactional` isoliert Operationen voneinander
 - Änderungen werden erst nach Erfolg sichtbar

Wann wird `@Transactional` nicht benötigt

- Updates die nur einen Schritt enthalten und nicht kaskadieren
- Wenn strikte konsistenz nicht benötigt wird
- Methoden, die keine Daten modifizieren

Entities

Repositories

Controllers

Services

DTOs

Mapper

DTO

- Bündelung mehrerer Datenfelder
- Inhalt Richtet sich nach Einsatzzweck
- Beispiel: DTO, welches zu einer GET Request gehört und die Daten enthält, die zurückgegeben werden sollen
- Verweise auf andere Tabellen können als ID gespeichert werden oder direkt mit allen Daten eingetragen werden

Beispiele

```
1  class Entity(  
2      val id: Long?,  
3      val name: String,  
4      val age: Integer  
5  )
```

◀ Kotlin

```
1  data class GetEntityDto (  
2      val id: Long,  
3      val name: String,  
4      val age: Integer  
5  )
```

◀ Kotlin

Beispiele

```
1  class Entity(  
2      val id: Long?,  
3      val name: String,  
4      val age: Integer  
5  )
```

◀ Kotlin

```
1  data class PostEntityDto (  
2      val name: String,  
3      val age: Integer  
4  )
```

◀ Kotlin

Entities

Repositories

Controllers

Services

DTOs

Mapper

Mapper

- Umwandlung von Entities in DTOs oder umgekehrt
- Meist trivial, da nur Kopieren von Werten stattfindet

```
1  class Entity(  
2      val id: Long?,  
3      val name: String,  
4      val age: Integer  
5  )
```

◀ Kotlin

```
1  fun convertEntityToGetEntityDto(entity: Entity):  
   GetEntityDto {  
2      return GetEntityDto(  
3          id = entity.id!!,  
4          name = entity.name,  
5          age = entity.age  
6      )  
7  }
```

◀ Kotlin



Mensa im Osten
Studentenwerk Dresden