EAF Summary

Persönliche Zusammenfassung im Modul eaf HS17. Irrtum vorbehalten. Ausführungen beziehen sich vorwiegen auf das Spring Framework.

Dependency Injection

DI setzt sich aus 3 Teilen zusammen, der Komponente, dem Schema und der Infrastruktur.

Komponente

Spring Beans sind normale Java-Objekte (POJOs). Wird dieses durch einen Spring Container verwaltet, wird es zum Spring Bean.

Schema

Bauanleitung wie Beans zu einem Gesamtsystem zusammengebaut werden. In Spring XML, Annotations, Java oder CoC.

Infrastruktur

Der Spring Container enthält und verwaltet den Lebenszyklus und die Konfiguration von Java-Objekten.

Varianten der Konfiguration

XML (Standard)

Welche POJOs als Beans aufgenommen werden sollen wird mittels XML definiert. Ebenso wird das Wiring zwischen den Beans konfiguriert.

```
<context:property-placeholder location="classpath:application.properties" />

<bean id="renderer" class="ch.fhnw.edu.eaf.springioc.renderer.StandardOutRenderer">
    <property name="messageProvider" ref="provider" />
</bean>

<bean id="provider"
class="ch.fhnw.edu.eaf.springioc.provider.ExternalizedConstructorMessageProvider">
    <constructor-arg name="message" value="${hello.message}" />
</bean>
```

Annotations

Beans werden mit `@Component` registriert. Abhängigkeiten zwischen Beans müssen mittels `@Autowiring` eingesteckt werden. Damit das Spring Framework die Beans automatisch erkennt, muss `@ComponentScan` aktiviert werden.

```
<context:component-scan base-package="ch.fhnw.edu.eaf.app.domain" />
```

Context in einem JUnit Test definieren:

```
@ContextConfiguration(locations = {"/spring/annotationTest.xml"})
```

Java Configuration

Konfigurationen können ebenso auch in einer Java-Klasse gelöst werden. Hierzu ein Bespiel.

```
@Configuration
@ComponentScan(basePackages = {"package1", "package2", …})
@PropertySource("application.properties")
public class AppConfig {
   @Bean
   public Foo foo() {
      return new Foo(bar()); // inject Bar as dependency
   }
   @Bean
   public Bar bar() {
      return new Bar();
   }
}
```

Context in einem JUnit Test definieren:

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {AppConfig.class})
```

Convention over Configuration (CoC)

Spring bringt bereits viele vordefinierte Konventionen mit, es kann also auf einiges an Konfiguration verzichtet werden – solange die Konventionen eingehalten werden.

```
@SpringBootConfiguration
@ComponentScan(basePackages = {"ch.fhnw.edu.eaf.app"})
@SpringBootTest(classes = {AppConfigCoC.class})
```

Dependeny Injection Pitfalls

Folgend sind klassische Fehler im Umgang mit DI und Spring aufgelistet. 1. Einem Bean-Property wird mittels Setter-Injection ein Wert zugewiesen. Das Feld in der Klasse ist jedoch als private deklariert. Lösung: Constructor Injection 2. Mehrere Beans mit identischem Identifier definiert = Exception (id = Unique!). Weitere Definition des selben Beans ohne id = keine Exception. 3. Ein Bean kann auch «By-Class» referenziert werden. Wenn das Bean nicht eindeutig definiert ist, siehe (2),

wird eine Exception geworfen. 4. Zirkuläre Abhängigkeit A > B > C > A - Setter Injection: Kein Problem, es werden erst alle Instanzen erzeugt, danach werden die Abhängigkeiten aufgelöst. - Constructor Injection: Führt zu einer Exception, Instanzen stehen bei Auflösung der Abhängigkeiten noch nicht zur Verfügung.

DAO Pattern

Jegliche Datenbankzugriffe werden über DAO-Objekte, auch Repository genannt, abgewickelt. Jedes DAO ist für CRUD Operationen auf eine Entität zuständig. Transaktionen, Sessions oder Verbindungen werden nicht durch das DAO verwaltet. Mit dem DAO Pattern wird der Zugriff auf den Datenbank-Layer vom Rest der Anwendung (Business Logik) gekapselt. Ebenso wird die Wart- und Testbarkeit optimiert.

Template

```
public interface Repository<T, ID extends Serializable> {
    T findOne(ID id);
    List<T> findAll();
    T save(T t);    // used for create and update
    void delete(ID id);
    void delete(T entity);
    booleanexists(ID id);
    long count();
}
```

Service Layer

Core API durch welche andere Teile der Anwendung kommunizieren (Façade Pattern). Hier wird üblicherweise die Business Logik implementiert, welche dann über DAOs auf den Persistance-Layer zugreift.

Persistance

Plain JDBC

Die Implementierung eines DAOs ist mittels JDBC höchst umständlich. In jedem DAO müssen Verbindung und Exceptions verwaltet sowie eine Menge an Boiler-Plate Code geschrieben werden. Spring bietet hier Template Klassen an, welche diese Arbeiten bereits zum grossen Teil erledigen.

JDBC Template Pattern

Reduziert redundaten Code, verwaltet Ressourcen (Verbindungen), lässt sich einfach Injecten und vereinheitlicht das Exception handling. Konkret kann entweder ein `JdbcTemplate` oder ein `NamedParameterJdbcTemplate` verwendet werden. Letzteres ermöglicht SQL Anfragen welche benannte Parameter wie z.B. `:id` enthalten.

Methoden

- execute
- query
- update
- batchUpdate

Beispiel

```
@Override
public List<Movie> findAll() {
    return template.query(
        "select * from Movies",
        (rs, row) -> createMovie(rs) // callback method returns Movie instance per
row
    );
}
```

Java Persistance API

JPA übernimmt das Verwalten des Persistierens sowie das Mapping von Objekten gegenüber dem Persistance Layer (ORM).

Entity Manager

Wie der Name bereits impliziert, verwaltet der EM die Entitäten und ermöglicht Zugriff via find, persist, update und remove auf diese (ähnlich wie bei einem Repository). Der Lifecycle der Entiät wird durch den EM kontrolliert. Konfiguriert werden Entitäten über `@Entity` Annotationen oder XML Dateien.

Beispiel

```
@Entity
public class Movie {
    @Id
    private Long id;
}

public class MovieRepository {
    @PersistenceContext
    private EntityManagerem;
    public void saveNewMovie(String title, Date date) {
        Movie m = new Movie(title, date);
        em.persist(m);
    }
}
```

Methoden

- **persist** Macht Objekt managed und persistiert
- **remove** Löscht Instanz von der Datenbank

- **find** Findet Entität bei PK/ID
- **merge** Merges gegebene Entität mit dem Persistance Context, neue Instanz wird returnt
- **refresh** Entität neu von der Datenbank laden, Änderungen werden verworfen
- **flush** Schreibt den Persistance Context auf die Datenbank
- **contains** prüft ob Entität zu dem Context gehört (nicht ob diese auf der DB vorhanden ist)
- **clear** Context bereinigen, alle managed Objects werden detachted

Entity Bean Lifecycle

- New (Transient)
- Managed (Persistent)
- Detached
- Removed

Persistance Context

Dies ist eine Set von verwalteten Objekten welche durch den Entity Manager verwaltet werden. Eine Persistance Unit ist z.B. eine definierte Datenbank, diese können wie folgt auf einem EM definiert werden.

```
@PersistenceContext(name="movierental") // easy switching between persistance units
```

Manuelle Transaktion

```
em.getTransaction().begin();
...
em.getTransaction().commit();
```

Entity Annotations

Folgend die wichtigsten Annotationen für Entitätsklassen im JPA/Hibernate Framework.

`@Entity` = markiert POJO als Entität, somit managed durch Entity Manager
`@Table(name="MyTableName")` = definiert manuell Namen der Tabelle
`@Id` = markiert Feld als PK
`@GeneratedValue(strategy=GenerationType.IDENTITY)` = definiert wie Id Wert generiert werden soll
`@Column(name="MyColumn")` = manuelle Definition des Spaltennamens
`@Basic` = markiert Feld das persistiert werden soll
`@Enumerated` = definiert wie Enumeration persistiert werden soll (EnumType.ORDINAL oder EnumType.STRING)
`@Lob` = markiert Feld als large object, also BLOB Feld
`@Transient` = Markiert Feld das nicht persistiert werden soll
`@Temporal` = Markiert Datumsfelder, lässt Format/Präzision definieren

Bei `@Entity`, `@Table` sowie `@Column` ist der Name jeweils der UQN des annotierten Felds/Klasse.

Primary Key generation

Folgende Möglichkeiten zur Generierung von PKs sind vorhanden.
- AUTO (JPA/Hibernate wählt einen basiernd auf unterliegender DB) - Assigned (Applikation regelt Generierung/Zuweisung selbst) - Identity (Klassisches Auto-Increment) - Sequence (Generator wie UUID in MSSQL, ORACLE) - Table (PKs werden in seperater Tabelle geführt)

Queries

In JPA können Queries natürlich auch selbst geschrieben werden, dies in sogn. JPQL. Bespiel:

```
TypedQuery<Movie> q = em.createQuery("SELECT m FROM Movie m WHERE m.title= :title",
Movie.class);
q.setParameter("title", title);
List<Movie> movies = q.getResultList();
// q.getSingleResult() ==> would return one result
```

Typed Named Queries

Queries können auf der Entitätsklasse vordefiniert werden, quasi als Variable.

```
@NamedQueries({
    @NamedQuery(name="movie.all", query="SELECT m from Movie m"),
    @NamedQuery(name="movie.byTitle", query="select m from Movie m where m.title =
:title")
})
class Movie {...}

// Verwendung
TypedQuery<Movie> q = em.createNamedQuery("movie.byTitle", Movie.class);
```

Interessantes Query

```
SELECT NEW ch.fhnw.edu.Person(c.name,c.prename) FROM Customer c
```

Pageing

```
query.setFirstResult(20);   // start at position 20
query.setMaxResults(10);    // take 10 entries
```

Joins

- Bei ManyToOne und OneToOne werden Joins implizit gemacht

```
SELECT c.name, c.address.city FROM Customer c
```

- Über mehrere Many Beziehungen hinwegg (User->Rentals->Movie) braucht es explizite Joins!

```
-- INNER JOIN
SELECT r.movie.title from User u inner join u.rentals r where r.id = 10
-- LEFT OUTER JOIN
SELECT u.name, r from User u leftjoin u.rentals r
-- LAZY LOADING JOIN
SELECT u from User u left join fetch u.rentals
```

OrderBy

- `@OrderBy` = by primary key
- `@OrderBy("name")` = by name ascending)
- `@OrderBy("name DESC")` = by name descending)
- `@OrderBy` = by primary key
- `@OrderBy("city ASC, name ASC")` = by phonebook order

Criteria API

Das Schreiben von JSQL Queries kann zu verdeckten Fehlern führen. Mit der Creteria API lassen sich Queries mittels OOP Builder schreiben.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Movie> cq = cb.createQuery(Movie.class);
// SELECT m FROM Movie m WHERE m.title= :title
Root<Movie> m = cq.from(Movie.class);
cq.select(m);
cq.where(cb.equal(m.get("title"), title));
return em.createQuery(cq).getResultList();
```

Meta data class

Um bei der Creteria API auf Strings verzichten zu können (oben `m.get("title")`), kann eine meta data Klasse erstellt werden.

```
@StaticMetamodel(Movie.class)
public abstract class Movie_ {
public static volatile SingularAttribute<Movie, Boolean> rented;
...
cq.where(cb.equal(m.get(Movie_.title), title));
```

Inner Join mit Creteria API

```
CriteriaBuildercb= em.getCriteriaBuilder();
CriteriaQuery<Object[]> query= cb.createQuery(Object[].class);
Root<User> user= query.from(User.class);
Join<User, Rental> rental = user.join(User_.rentals);
```

```
query.select(cb.array(
    user.get(User_.lastName),
    rental.get(Rental_.movie).get(Movie_.title)
));
List<Object[]> result= em.createQuery(query).getResultList();
for(Object[] res: result) {
    …
}
```

Flush Mode

- AUTO (Änderungen werden **vor** einem Query geflusht)
- COMMIT (Änderungen werden **nur** explizit - `em.flush()` - geflusht)

Associations

Beziehungen zwischen Entitäten können mittels JPA/Hibernate genau so definiert und benutzt werden. Beziehungen können unidirectional oder bidirectional sein. Für bidirektionale Beziehungen muss eine Seite mit `mappedBy="FIELD_NAME"` markiert werden.

Collection type

Werden Collections in Beziehungen verwendet, so muss der Typ des Ziels immer bekannt gemacht werden.

```
// manual definition
@OneToMany(targetEntity=Order.class)
public Collection orders;

// Explicit
@OneToMany
Collection<Order> orders;
```

Cascading

Folgende Aktionen können mittels Cascading auch auf zugeordneten Entitäten ausgeführt werden. - PERSIST - REMOVE - Nur bei `@OneToOne` und `@OneToMany` verwenden! - REFRESH - MERGE - DETACH - ALL

Fetch Types

- EAGER
  - Abhängigkeiten werden beim Laden des ROOT Objekts aufgelöst
  - default für `@OneToOne` und `@ManyToOne`
- LAZY
  - Abhängigkeiten werden bei Bedarf aufgelöst
  - default für `@OneToMany` und `@ManyToMany`

@OneToOne

```
// optional=false defines NOT NULL = TRUE
@OneToOne(optional=false, cascade={CascadeType.PERSIST, CascadeType.REMOVE})
private Address address; // unidirectional

@OneToOne(mappedBy="address")
private Person person; // makes it bidirectional
```

@ManyToOne / @OneToMany

Bidirectional Owner: Many Seite (mappedBy immer auf `@OneToMany`)
Tipp: Immer aus Sicht der Klasse beurteilen. Ein User hat viele Rentals - one user to many rentals - `@OneToMany`

```
@Entity
public class Rental {
    @ManyToOne // This is the owner of the relationship
    @JoinColumn(name="USER_FK") // optional
    private User user;
}

@Entity
public class User {
    @OneToMany(mappedBy="user")
    private Collection<Rental> rentals;
}
```

ManyToMany

- Jede Seite kann als Owner definiert werden.
- Wird über eine Mapping-Tabelle realisiert (n:n)
- Name der Mapping-Tabelle über `@JoinTable` definierbar
- Name der Spalten definierbar

```
@JoinTable(
    name = "ENROLLMENTS",
    joinColumns = @JoinColumn(name = "student"),
    inverseJoinColumns = @JoinColumn(name = "module")
)
@ManyToMany
private List<Module> modules = new LinkedList<>();
```

Bidirektionale Beziehungen

Wichtig bei genannten Beziehungen ist, dass es jeweils eine sogn. "Owner" Seite gibt.
Bei der Speicherung wird lediglich der State des Owners betrachtet - jener der Inversen Seite wird ignoriert.

Orphan removal

Identisch zu Cascade.REMOVE, jedoch werden auch nicht mehr refernzierte Objekte gelöscht. Wird z.B. einem Feld null oder eine andere Instanz zugewiesen, so wird dies normalerweise nicht gelöscht auf der Datenbank. Mittels `@OneToXXX(orphanRemoval=true)` schon!

Inheritance

Alle Klassen in der Vererbungshirarchie müssen mit `@Entity` annotiert werden. Auf der ROOT Klasse kann das Verhalten beim persistieren gesteuert werden. Hierzu wird über die Annotation `@Inheritance` ein InheritanceType angegeben.

SINGLE_TABLE (DEFAULT)

`@Inheritance(strategy=InheritanceType.SINGLE_TABLE)`
- Alle Attribute in einer Tabelle - Typ mittels `@DiscriminatorColumn("name_of_type", DiscriminatorType type)` - Value mittels `@DiscriminatorValue("value_of_type")` - Neue Felder in Sub-Klassen müssen Nullable sein - optional = false nur mittles manuellen SQLs constraints möglich - FKs können nur auf Basisklasse zeigen

JOINED

`@Inheritance(strategy=InheritanceType.JOINED)`
- Pro Klasse wird eine Tabelle angelegt (BASE-PK joined)
- Vorteile: Normalized, NOT NULL möglich, FKs zu Subklassen möglich - Nachteil: Zugriffe müssen über mehrere Tabellen gehen

TABLE_PER_CLASS

`@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)` - Eine Tabelle pro nicht abstrakte Klasse - Tabelle enthält jeweils Felder der Basisklasse(n) sowie der abgeleiteten Klasse - Vorteile: NOT NULL möglich, FKs zu Subklassen möglich - Nachteil: ID-Generator nicht nutzbar, Polymorphe Abfragen müssen mehrere Tabellen anfragen

@MappedSuperclass

Diese Annotation signalisiert, dass diese Klasse eine Superklasse ist und nicht direkt in der DB abgebildet werden soll.
Erben `@Entity` Klassen jedoch von dieser, werden deren Properties übernommen und persistiert.

Automatic JPA Repositories

Bei der "manuelle" Implementierung von JPA Repositories fällt auf, dass diese oft gleich aussehen (Boilderplate) und jeweils direkt mit dem Entity Manager interagieren. Die Code-Stellen lässt sich mit Spring Data bequem automatisch generieren.

JPA Repository

Das Interface `JpaRepository` bietet CRUD sowie Paging und Sorting Methoden an. Die Implementation ist Teil von Spring Boot. In Repositories welche Spring findet, bzw. wo konfiguriert, können die generierten Methoden direkt "gratis" verwendet werden.

Konfiguration

```
<jpa:repositories base-package="ch.fhnw.edu.rental.repository" />
```

```
@EnableJpaRepositories("ch.fhnw.edu.rental.repository")
@EnableJpaRepositories(basePackageClasses=RentalRepository.class)

rentalRepository.findAll(); // gratis!
```

Query Methods (MAGIC)

In einem Interface, welches von JpaRepository erbt, können weitere Methoden definiert werden. Über den Namen können automatisch Queries generiert werden.

```
public interface MovieRepository extends JpaRepository<Movie, Long> {
    // NOTATION: find[Entity]By…
    List<Movie> findMovieByTitleIgnoringCase(String title);
    // ==> where upper(m.title) = upper(?1)
}
```

Mögliche Keywords

- AND / OR
  - findByLastnameAndFirstname / findByLastnameOrFirstname
  - `where x.lastname = ?1 and (or) x.firstname = ?2`
- Is, Equals
  - findByFirstnameIs/ findByFirstnameEquals/ findByFirstname
  - `where x.firstname = ?1`
- Between
  - findByStartDateBetween
  - `where x.startDate between ?1 and ?2`
- LessThan, GreaterThan
  - findByAgeLessThan/ findByAgeGreaterThan
  - `where x.age < ?1 / … where x.age > ?1`
- After, Before
  - findByStartDateAfter/ findByStartDateBefore
  - `where x.startDate > ?1 / … where x.startDate < ?1`
- IsNull, IsNotNull, NotNull
  - findByAgeIsNull/ findByAge[Is]NotNull
  - `where x.age is null / … where x.age not null`

- Like / NotLike
  - findByFirstnameLike/ findByFirstnameNotLike
  - `where x.firstname like ?1/ … where x.firstname not like ?1`
- OrderBy
  - findByAgeOrderByLastnameDesc
  - `where x.age = ?1 order by x.lastname desc`
- True / False
  - findByActiveTrue()/ findByActiveFalse()
  - `where x.active = true/ … where x.active = false`
- In / NotIn
  - findByAge[Not]In(Collection ages)
  - `where x.age in ?1/ … where x.age not in ?1`
- Not
  - findByLastnameNot
  - `where x.lastname <> ?1`
- IgnoreCase
  - findByFirstnameIgnoreCase
  - `where UPPER(x.firstname) = UPPER(?1)`
- Limit
  - findFirstByAddressCityByNameAsc
  - findFirst10ByLastnameAsc

Zugriff auf Properties

```
// Entities can be passed as parameters
// Attributes over ManyToOne / OneToOne associations can be accessed
List<Rental> findByMovieTitleContains(String title);
List<Rental> findByMoviePriceCategoryIs(PriceCategory pc);
```

Named Queries

```
// @NamedQuery(name = "Movie.byTitle", query = "SELECT m FROM Movie m WHERE m.title
=":title"
List<Movie> byTitle(@Param("title") String title);

// Explicit Query specification using @Query
@Query("select m from Movie m where UPPER(m.title) = UPPER(:title)")
List<Movie> findMovieByTitle(@Param("title") String title);
```

DTOs

Werden `@Entity` Entitäten als Resultat zurückgegeben und weiterverwendet treten folgende Probleme auf: - Lazy loading Exceptions wenn Felder nicht zugreifbar sind weil **DETACHED** - Solche "accessors" welche Exceptions werfen verletzen den "Contract"

Verhindern der Exceptions

- FetchType.EAGER
- keep session / persistence context open
- JPQL: fetch join
- Entity Graphs

Entity Graphs

```
@NamedEntityGraphs({
    @NamedEntityGraph(name="previewCustomerEntityGraph",
        attributeNodes= {
            @NamedAttributeNode("name"),
            @NamedAttributeNode("age") }),
    @NamedEntityGraph(name="fullCustomerEntityGraph",
        attributeNodes= {
            @NamedAttributeNode("name"),
            @NamedAttributeNode("age"),
            @NamedAttributeNode("address"),
            @NamedAttributeNode("orders") })
    // optional (sample)
    subgraphs={} // Allows to define subgraphsfor the associated objects
})
@Entity
public class Customer { ... }

// Passing the entity graph as a property
// If a fetch graph is used, only the attributes specified by the entity graph will
// be treated as FetchType.EAGER. All other attributes will be lazy
Map<String, Object> props = new HashMap<>();
props.put("javax.persistence.fetchgraph",
em.getEntityGraph("fullCustomerEntityGraph"));
Customer c = em.find(Customer.class, 1, props);

// Passing the entity graph as a hint
// If a load graph is used, all attributes that are not specified
// by the entity graph will keep their default fetch type
EntityGraph<?> eg= em.getEntityGraph("previewEmailEntityGraph");
List<Customer> customers = em.createNamedQuery("user.findByName",
Customer.class)
                .setParameter("name", name)
                .setHint("javax.persistence.loadgraph", eg)
                .getResultList();
```

DTO Sample

```
public class UserDto implements Serializable{
    private Long id;
    private String lastName;
    private String firstName;
    private List<Long> rentalIds; // allows to access rentals on demand
    public UserDto(Long id, String lastName, String firstName, List<Long> rentalIds)
{
```

```
        this.id = id;
        this.lastName= name;
        this.firstName= firstName;
        this.rentalIds= rentalIds;
    }
}

// in repository
public UserDtogetUserDataById(Long id) {
    TypedQuery<UserDTO> q = em.createNamedQuery("User.dataById", UserDTO.class);
    q.setParameter("id", id);
    UserDTO dto = q.getSingleResult();
    TypedQuery<Long> q2 = em.createNamedQuery("User.rentalsById",Long.class);
    q2.setParameter("id", id);
    dto.setRentalIds(q2.getResultList());
    return dto;
}

// custom named query
@NamedQuery(name="User.dataById", query="SELECT NEW ch.fhnw.eaf.UserDto(u.id,
u.name, u.firstName) FROM User u WHERE u.id = :id"),
@NamedQuery(name="User.rentalsById", query="SELECT r.id FROM User u, IN(u.rentals)
r WHERE u.id = :id")

// mapper
@Mapper(componentModel="spring")
public interface MovieMapper{
    @Mapping(source = "rentals", target = "rentalIds")
    UserDtouserToUserDto(User user);
    default Long rentalToLong(Rental r) {
        return r.getId();
    }
}

@Autowired
MovieMappermapper;

public UserDto getUserDataById(Long id) {
    return mapper.userToUserDto(userRepo.findOne(id));
}
```

JOOQ

```
@Autowired
private DSLContextdsl;

@Override
public List<User> findAll() {
    List<User> users = new ArrayList<>();
    Result<Record> result = dsl.select().from(USERS).fetch();
    for (Record r : result) { users.add(getUserEntity(r)); }
    return users;
}

@Override
public User save(User user) {
```

```
    if(user.getId() == null) {
        UsersRecorduserRecord= dsl.insertInto(USERS)
        .set(USERS.USER_NAME, user.getLastName())
        .set(USERS.USER_FIRSTNAME, user.getFirstName())
        .set(USERS.USER_EMAIL, user.getEmail())
        .returning(USERS.USER_ID)
        .fetchOne();
        user.setId(userRecord.getUserId());
        } else {
        dsl.update(MOVIES).set(USERS.USER_NAME, user.getLastName())
        .set(USERS.USER_FIRSTNAME, user.getFirstName())
        .set(USERS.USER_EMAIL, user.getEmail())
        .where(MOVIES.MOVIE_ID.eq(user.getId()))
        .execute();
    } return user;
}
```