Java Persistence API

- **Spring Data Repositories**
- **Open Session In View**
- **Data Transfer Objects**
- **Summary**

Automatic JPA Repositories

- **JPA repository implementations**

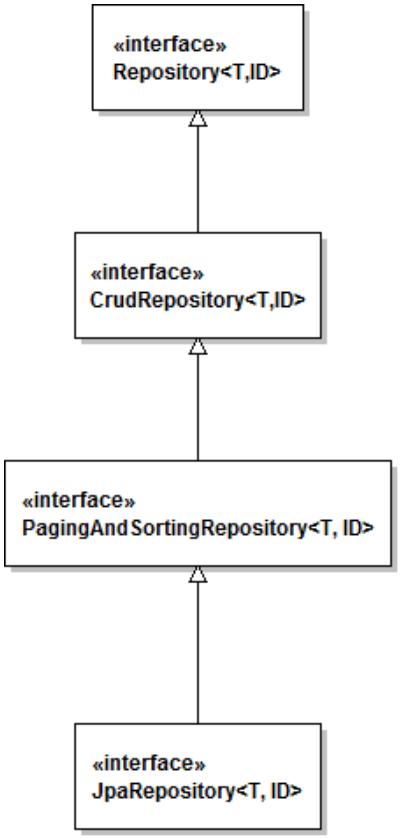
```
public Optional<Movie> findById(Long id) {
   return Optional.ofNullable(em.find(Movie.class, id));
}

public List<Movie> findAll() {
   return em.createQuery("SELECT m FROM Movie m", Movie.class)
                                                  .getResultList();
}
```

  - Methods interact directly with the entity manager
  - Methods look boilerplate-ish (always the same code)

=> Spring Data support automatic generation of this code

Automatic JPA Repositories

- **Repository-Interfaces**
  - Repository<T, ID> *marker interface*
  - CrudRepository<T, ID> *generic CRUD operations*
    - <S extends T> S save(S entity)
    - Optional<T> findById(ID id);
    - Iterable<T> findAll();
    - boolean existsById(ID id);
    - long count();
    - void deleteById(ID id);
    - void delete(T entity);
    - void deleteAll();
  - PagingAndSortingRepository<T, ID>
    - Iterable<T> findAll(Sort sort);
    - Page<T> findAll(Pageable pageable);
  - JpaRepository<T, ID>



JpaRepository<T, ID>

```
JpaRepository<T, ID> - org.springframework.data.jpa.repository
  findAll() : List<T> - org.springframework.data.jpa.repository.JpaRepository
  findAll(Sort) : List<T> - org.springframework.data.jpa.repository.JpaRepository
  findAllById(Iterable<ID>) : List<T> - org.springframework.data.jpa.repository.JpaRepository
  saveAll(Iterable<S>) <S extends T> : List<S> - org.springframework.data.jpa.repository.JpaRepository
  flush() : void - org.springframework.data.jpa.repository.JpaRepository
  saveAndFlush(S) <S extends T> : S - org.springframework.data.jpa.repository.JpaRepository
  deleteInBatch(Iterable<T>) : void - org.springframework.data.jpa.repository.JpaRepository
  deleteAllInBatch() : void - org.springframework.data.jpa.repository.JpaRepository
  getOne(ID) : T - org.springframework.data.jpa.repository.JpaRepository
  findAll(Example<S>) <S extends T> : List<S> - org.springframework.data.jpa.repository.JpaRepository
  findAll(Example<S>, Sort) <S extends T> : List<S> - org.springframework.data.jpa.repository.JpaRepository
  findAll(Sort) : Iterable<T> - org.springframework.data.repository.PagingAndSortingRepository
  findAll(Pageable) : Page<T> - org.springframework.data.repository.PagingAndSortingRepository
  save(S) <S extends T> : S - org.springframework.data.repository.CrudRepository
  saveAll(Iterable<S>) <S extends T> : Iterable<S> - org.springframework.data.repository.CrudRepository
  findById(ID) : Optional<T> - org.springframework.data.repository.CrudRepository
  existsById(ID) : boolean - org.springframework.data.repository.CrudRepository
  findAll() : Iterable<T> - org.springframework.data.repository.CrudRepository
  findAllById(Iterable<ID>) : Iterable<T> - org.springframework.data.repository.CrudRepository
  count() : long - org.springframework.data.repository.CrudRepository
  deleteById(ID) : void - org.springframework.data.repository.CrudRepository
  delete(T) : void - org.springframework.data.repository.CrudRepository
  deleteAll(Iterable<? extends T>) : void - org.springframework.data.repository.CrudRepository
  deleteAll() : void - org.springframework.data.repository.CrudRepository
  findOne(Example<S>) <S extends T> : Optional<S> - org.springframework.data.repository.query.QueryByExampleExecutor
  findAll(Example<S>) <S extends T> : Iterable<S> - org.springframework.data.repository.query.QueryByExampleExecutor
  findAll(Example<S>, Sort) <S extends T> : Iterable<S> - org.springframework.data.repository.query.QueryByExampleExecutor
  findAll(Example<S>, Pageable) <S extends T> : Page<S> - org.springframework.data.repository.query.QueryByExampleExecutor
  count(Example<S>) <S extends T> : long - org.springframework.data.repository.query.QueryByExampleExecutor
  exists(Example<S>) <S extends T> : boolean - org.springframework.data.repository.query.QueryByExampleExecutor

Press 'Ctrl+O' to hide inherited members
```

Automatic JPA Repositories

- **Define an Interface as an extension of a repository interface**

```
public interface RentalRepository extends JpaRepository<Rental,Long>{
}
```

- **Implementation is provided by Spring**

Spring Data JPA

Spring Data module for JPA repositories.

License	Apache 2.0
Tags	data spring persistence jpa

  – Part of `org.springframework.boot:spring-boot-starter-data-jpa`

Automatic JPA Repositories

- **Configuration of Spring Data JPA with Spring Boot**
  - By default, Spring Boot will enable JPA repository support and look in the package (and its subpackages) where `@SpringBootApplication` is located (actually where the `@EnableAutoConfiguration` is located)
  - If the JPA repository interfaces are located in other packages, they can be referenced using an `@EnableJpaRepositories` annotation

```
@EnableJpaRepositories("ch.fhnw.edu.rental.repository")
```

    - Scans all packages below the specified package

```
@EnableJpaRepositories(basePackageClasses=RentalRepository.class)
```

    - Type-safe variant to specify base packages to scan for repository interfaces

Automatic JPA Repositories

- **Use the injected implementation**

```
@Transactional
public class RentalServiceImpl implements RentalService {

  @Autowired
  private RentalRepository rentalRepository;

  @Override
  public List<Rental> getAllRentals() throws RentalServiceException {
    return rentalRepository.findAll();
  }

  @Override
  public Rental getRentalById(Long id) {
    return rentalRepo.findById(id).orElse(null);
  }
  …
```

These methods are provided "for free"

Automatic JPA Repositories

- **Implementation is based on class SimpleJpaRepository**

```
public <S extends T> S save(S entity) {
   if (entityInformation.isNew(entity)) {
      em.persist(entity); return entity;
   } else {
      return em.merge(entity);
   }
}

public void delete(T entity) {
   if (entityInformation.isNew(entity)) { return; }
   Class<?> type = ProxyUtils.getUserClass(entity);
   T existing = (T) em.find(type, entityInformation.getId(entity));
   // if the entity to be deleted doesn't exist, delete is a NOOP
   if (existing == null) { return; }
   em.remove(em.contains(entity) ? entity : em.merge(entity));
}
```

Implementation invokes `persist` or `merge` (relevant for cascade annotations)

delete can be invoked on detached instances!

Spring Data Repository DSL

- **Defining Query Methods**

```
public interface MovieRepository extends JpaRepository<Movie, Long> {
   List<Movie> findMovieByTitleIgnoringCase(String title);
}
```

  - The query method is generated based on the method name
    - `find[Entity]By…`
    - => method name is a simple DSL for defining queries
    - Implementation is based on dynamic proxy classes
  - The method name could also refer to a named query
  - The query could also be marked with a `@Query` annotation

Spring Data Repository DSL

- **Supported 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, LessThanEqual, GreaterThanEqual
    - `findByAgeLessThan` / `findByAgeGreaterThan`
    - … where x.age < ?1 / … where x.age > ?1

Spring Data Repository DSL

- **Supported keywords**
  - After, Before
    - `findByStartDateAfter` / `findByStartDateBefore`
    - … where x.startDate > ?1 / … where x.startDate < ?1
  - IsNull, Null, IsNotNull, NotNull
    - `findByAge[Is]Null` / `findByAge[Is]NotNull`
    - … where x.age is null / … where x.age not null
  - Like / NotLike (regular expression must be passed as argument)
    - `findByFirstnameLike` / `findByFirstnameNotLike`
    - … where x.firstname like ?1/ … where x.firstname not like ?1
  - StartingWith / EndingWith
    - `findByNameStartingWith` / `findByNameEndingWith`
    - … where x.name like ?1 (parameter bound with appended / prepended %)

Spring Data Repository DSL

- **Supported keywords**
  - Containing
    - `findByNameContaining`
    - … where x.firstname like ?1 (parameter bound wrapped in %)
  - IgnoreCase
    - `findByFirstnameIgnoreCase`
    - … where UPPER(x.firstname) = UPPER(?1)
  - True / False
    - `findByActiveTrue()` / `findByActiveFalse()`
    - … where x.active = true / … where x.active = false
  - Not
    - `findByLastnameNot`
    - … where x.lastname <> ?1

Spring Data Repository DSL

- **Supported keywords**
  - In / NotIn
    - `findByAge[Not]In(Collection<Age> ages)`
    - … where x.age in ?1 / … where x.age not in ?1
  - Distinct
    - `find`**`Distinct`**`ByLastnameAndFirstname`
    - … SELECT DISTINCT … WHERE x.lastname = ?1 AND x.firstname = ?2
  - OrderBy
    - `findByAgeOrderByLastnameDesc`
    - … where x.age = ?1 order by x.lastname desc

Spring Data Repository DSL: Examples

- **MovieRepository**

```
public interface MovieRepository extends JpaRepository<Movie, Long> {
   List<Movie> findByTitle(String title);
}
```

- **User Repository**

```
public interface UserRepository extends JpaRepository<User, Long> {
   List<User> findByLastName(String lastName);
   List<User> findByFirstName(String firstName);
   List<User> findByEmail(String email);
}
```

- **PriceCategoryRepository**

```
public interface PriceCategoryRepository
                        extends JpaRepository<PriceCategory, Long> { }
```

Spring Data Repository DSL: Examples

- **UserRepository (additional query methods)**

```
public interface UserRepository extends JpaRepository<User, Long> {
   List<User> findByFirstNameAndLastName(String firstname,
                                         String lastname);

   // must be invoked with a pattern, e.g. "%er%"
   List<User> findByFirstNameLike(String firstname);

   List<User> findByFirstNameContaining(String firstname);

   List<User> findByFirstNameStartingWith(String firstname);

   List<User> findByAgeAndFirstNameLikeOrLastNameLike(
                  int age, String firstname, String lastname);
}
```

Evaluated as `(e.age == age && e.firstName = firstname) || e.lastName = lastname`

Spring Data Repository DSL: Examples

- **Example of Repository methods**

```
public interface ContactRepository
                                extends JpaRepository<Contact, Long> {
   List<Contact> findByFirstNameStartingWithOrLastNameStartingWith
   OrderByLastNameAscFirstNameAsc(String prefixFN, String prefixLN);
}
```

  – Finds all customers whose first or last name starts with the given term (has to be specified twice) ordered ascending by last and first name.
    – Spring Data, Petri Kainulainen, PACKT Publishing 2012

```
List<Lent> findAllByUserFirstNameContainsOrUserLastNameContainsOrUser
BookBookTitleContainsOrUserBookBookDescriptionContainsOrUserBookBookP
ublisherContainsOrUserBookBookAuthorContains(String userFirstName,
String userLastName, String userBookBookTitle, String
userBookBookDescription, String userBookBookPublisher, String
userBookBookAuthor);
```

Spring Data Repository DSL: Examples

- **DSL allows to navigate over properties**

```
public interface RentalRepository
                          extends JpaRepository<Rental, Long> {
   List<Rental> findByUser(User user);                          // (1)

   List<Rental> findByMovieTitleContains(String title);        // (2)

   List<Rental> findByMoviePriceCategoryIs(PriceCategory pc);  // (3)
}
```

  - Entities can be passed as parameters
    - (1) and (3)
  - Attributes can be accessed over ManyToOne / OneToOne associations
    - (2) and (3)

Spring Data Repository NamedQueries

- **Define Named Query**

```
@NamedQuery(name = "Movie.byTitle",
            query = "SELECT m FROM Movie m WHERE m.title = ?1"),
```

```
public interface MovieRepository extends JpaRepository<Movie, Long> {
   List<Movie> byTitle(String title);
}
```

  - Name of the configured domain class followed by the method name
  - Parameters may be named

```
@NamedQuery(name = "Movie.byTitle",
            query = "SELECT m FROM Movie m WHERE m.title =":title"),
```

```
public interface MovieRepository extends JpaRepository<Movie, Long> {
   List<Movie> byTitle(@Param("title") String title);
}
```

Spring Data Repository Queries

- **Explicit Query specification using @Query**

```
public interface MovieRepository extends JpaRepository<Movie, Long> {
   @Query("SELECT m FROM Movie m WHERE UPPER(m.title) = UPPER(?1)")
   List<Movie> findMovieByTitle(String title);
}
```

  - Is used if otherwise the method name would be very very long
  - Needs to be used to specify joins
  - Can be used to specify modifying queries (UPDATE …)
- **Named parameters could also be used**

```
public interface MovieRepository extends JpaRepository<Movie, Long> {
   @Query("SELECT m FROM Movie m "
                          + "WHERE UPPER(m.title) = UPPER(:title)")
   List<Movie> findMovieByTitle(@Param("title") String title);
}
```

Spring Data Repository Queries

- **Example from UserRepository**

```
public interface UserRepository extends JpaRepository<User, Long> {
   @Query("SELECT u from USER u where"
      + " (u.firstName like :name or u.lastName like :name)"
      + " and u.age = :age")
   List<User> getByAgeAndNameLike(
                  @Param("age") int age,
                  @Param("firstname") String name);
}
```

  – A && (B || C) ⇔ (A && B) || (A && C) (=> disjunctive normal form)

```
List<User> findByAgeAndFirstNameLikeOrAgeAndLastNameLike(
          int age1, String firstname, int age2, String lastname);
```

Spring Data Repository Queries

- **Example from Lab of last week**

```
public interface UserRepository extends JpaRepository<User, Long> {
   @Query("SELECT u FROM User u JOIN u.rentals r WHERE r.movie = :m")
   List<User> query(@Param("m") Movie movie);
}
```

  - Method name can freely be chosen (query is not the best name…)
  - Use:

```
Movie movie = movieRepo.findById(1L).get();
List<User> res = userRepo.query(movie);
if(res.isEmpty()) {
   System.out.println("nicht ausgeliehen");
} else {
   System.out.println("ausgeliehen an " + res.get(0).getEmail());
}
```

Spring Data Repository Specifications

- **Specifications**
  - Spring Data JPA allows to define the where clause using the criteria API as a predicate
- **Interface JpaSpecificationExecutor<T>**

```
JpaSpecificationExecutor<T>
    findOne(@Nullable Specification<T>) : Optional<T>
    findAll(@Nullable Specification<T>) : List<T>
    findAll(@Nullable Specification<T>, Pageable) : Page<T>
    findAll(@Nullable Specification<T>, Sort) : List<T>
    count(@Nullable Specification<T>) : long
```

- **Specification interface**

```
public interface Specification<T> {
   Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
                                          CriteriaBuilder builder);
}
```

Spring Data Repository Specifications

- **Example**

```
public interface MovieRepository extends JpaRepository<Movie, Long>,
                                         JpaSpecificationExecutor<Movie> {

   default List<Movie> findByTitle(String title) {
      Specification<Movie> specification = (root, query, builder)-> {
         return builder.equal(root.get(Movie_.title), title);
      };
      return findAll(specification);
   }

}
```

  - In this example using specifications makes no sense, but
  - the power of specifications really shines when you combine them to create new Specification objects

Spring Data Repository Specifications

- **Example**

```
static Specification<User> byAge(int age) {
   return (root, query, builder) ->
      builder.equal(root.get(User_.age), age);
}
static Specification<User> firstNameContains(String name) {
   return (root, query, builder) ->
      builder.like(root.get(User_.firstName), "%"+name+"%");
}
static Specification<User> lastNameContains(String name) {
   return (root, query, builder) ->
      builder.like(root.get(User_.lastName), "%"+name+"%");
}
default List<User> findByAgeAndNameContains(int age, String name) {
   return findAll(byAge(email)
      .and(firstNameContains(name).or(lastNameContains(name))));
}
```

Automatic JPA Repositories

- **Additional Features**
  - Support of Paging and Slicing
  - Support of limiting the result size of a query
    - `findFirstByAddressCityByNameAsc`
    - `findFirst10ByLastnameAsc`

- **Reference**
  - https://spring.io/projects/spring-data-jpa
  - https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

Java Persistence API

- Spring Data Repositories
- Open Session In View
- Data Transfer Objects
- Summary

Open Session In View

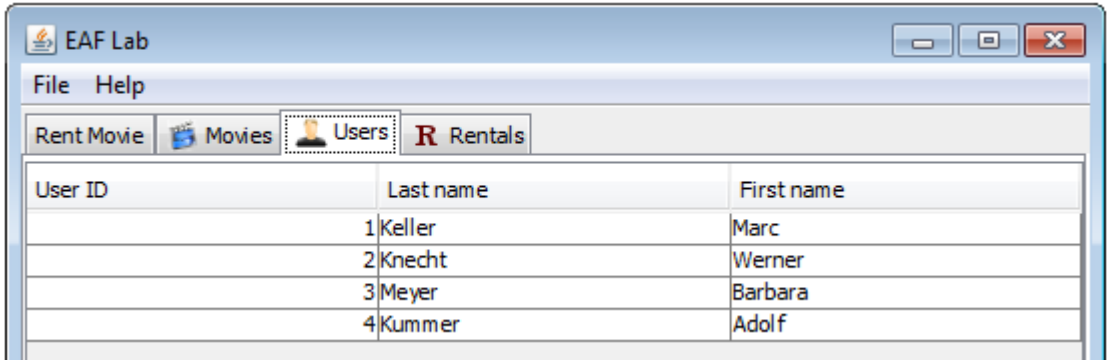
- **Warning upon starting the server**

```
2021-10-10 10:42:03.423  WARN 54256 --- [           main]
JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is
enabled by default. Therefore, database queries may be performed
during view rendering. Explicitly configure spring.jpa.open-in-view
to disable this warning
```

- **By Default, OSIV is enabled (with a warning)**
  - Persistence Context is bound to the calling thread, i.e. lazy loaded references may be accessed & reloaded in the view/controller layer
  - Support for the programmer, but bad idea from a DB point of view
    - Access outside of transactions is executed in auto-commit mode, i.e. each statement must flush the transaction log to disk
    - Changes are not automatically persisted (outside of transactions)
    - Danger: N+1 Problem

Open Session In View

- **N+1 Problem**




```
Hibernate: select user0_.user_id as user_id1_3_, user0_.user_email as user_ema2_3_, user0_.user_fi
Hibernate: select rentals0_.user_id as user_id5_2_0_, rentals0_.rental_id as rental_i1_2_0_, renta
Hibernate: select rentals0_.user_id as user_id5_2_0_, rentals0_.rental_id as rental_i1_2_0_, renta
Hibernate: select rentals0_.user_id as user_id5_2_0_, rentals0_.rental_id as rental_i1_2_0_, renta
Hibernate: select rentals0_.user_id as user_id5_2_0_, rentals0_.rental_id as rental_i1_2_0_, renta
```

=> For every User the Rentals are accessed (in separate transactions)!

Open Session In View: Problem 1

- **Let us disable OSIV**

```
spring.jpa.open-in-view=false
```

  – Error which happens on serialization of the entities into JSON (as the rentals cannot be accessed outside of the persistence context)

Open Session In View: Problem 1

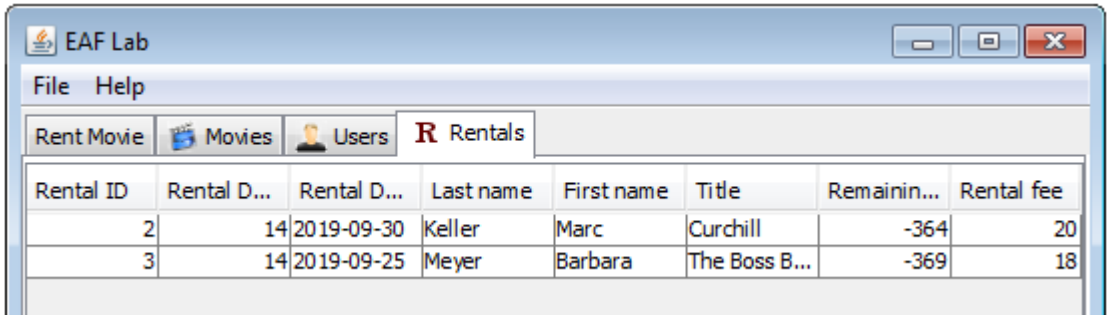
- **Solution: Eager Loading**

```
@Entity
@Table(name = "USERS")
public class User {
   @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "USER_ID")
   private Long id;

   @OneToMany(mappedBy = "user",
      cascade = CascadeType.REMOVE,
      fetch = FetchType.EAGER)
   private List<Rental> rentals;

   protected User() { }
```

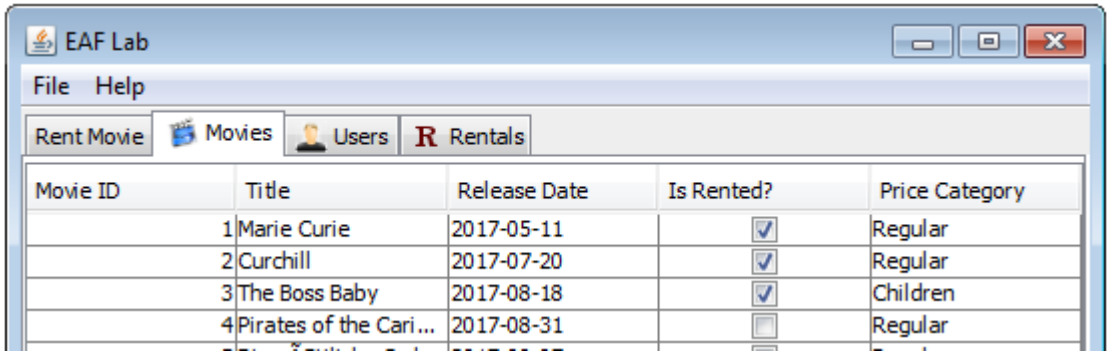
Prevents too many DB accesses outside of TX

Open Session In View: Problem 2

- **Let us disable OSIV**
  - If a rental is deleted…



  - …the corresponding movie is still marked as rented



Open Session In View: Problem 2

- **RentalServiceImpl.deleteRental**

```
@Override
public void deleteRental(Rental rental) {
   if (rental == null) {
      throw new RuntimeException("'rental' parameter is not set!");
   }

   rental.getUser().getRentals().remove(rental); // consistency only
   rental.getMovie().setRented(false);

   rentalRepo.delete(rental);
}
```

But the flag remains set in the GUI. Why???

  - Invoked from RentalController

```
@DeleteMapping(path = "/rentals/{id}")
public void deleteRental(@PathVariable Long id) {
   rentalService.deleteRental(rentalService.getRentalById(id));
}
```

Open Session In View: Problem 2

```
@Override
public void deleteRental(Rental rental) {
   if (rental == null) {
      throw new RuntimeException("'rental' parameter is not set!");
   }

   // The problem is that the rental entity is not managed
   // => add detached rental object to persistence context
   rental = rentalRepo.save(rental);

   // user and movie are managed as well
   // as they are accessed over the
   // managed entity rental.
   rental.getUser().getRentals().remove(rental);
   rental.getMovie().setRented(false);

   rentalRepo.delete(rental);
}
```

or `em.merge` if entity manager is accessible

Java Persistence API

- Spring Data Repositories
- Open Session In View
- Data Transfer Objects
- Summary

JPA Entities

- **Detached Entity objects as DTOs**
  - JPA developers may recommend to use entity (or domain) objects as result types in service methods

```
public interface UserService {
   User getUserById(Long id) throws RentalServiceException;
   ...
```

  - Problems:
    - Lazy load exceptions are thrown if "not-loaded" fields are accessed, e.g. `getRentals` on a `User`-Instance (=> `LazyInitializationException`)
    - Having an accessor which does throw an exception is contract violating

LazyInitializationException: Solutions

- **Declare association as fetch = FetchType.*EAGER***
  - Referenced objects are *always* loaded
- **OpenSessionInView / ExtendedPersistenceContext**
  - => keep session / persistence context open
    - Default in Spring-Boot if JPA and Web is used
  - No exceptions anymore, but database access outside of transactions
- **Load the associated objects in the service if needed / on demand**
  - => Loading Methods:
    - JPQL: fetch join
    - Manually loading the entities
      - `user.getRentals().size();` // typical solution
      - `user.getRentals().forEach(r -> {});`
      - `Hibernate.initialize(u.getRentals());` // works for Hibernate
    - Entity Graphs

All or nothing solutions

Load/Fetch Entity Graphs

```
@NamedEntityGraph(name="previewCustomerEntityGraph",
     attributeNodes = {
        @NamedAttributeNode("name"),
        @NamedAttributeNode("age") })
@NamedEntityGraph(name="fullCustomerEntityGraph",
     attributeNodes = {
        @NamedAttributeNode("name"),
        @NamedAttributeNode("age"),
        @NamedAttributeNode("address"),
        @NamedAttributeNode("orders") })
@Entity
public class Customer { ... }
```

Load/Fetch Entity Graphs

```
@NamedEntityGraph(
   name = "customersWithOrderId",
   attributeNodes = {
       @NamedAttributeNode(value = "name"),
       @NamedAttributeNode(value = "orders", subgraph = "ordersGraph")
   },
   subgraphs = {
      @NamedSubgraph(
         name = "ordersGraph",
         attributeNodes = {
            @NamedAttributeNode(value = "id"),
            @NamedAttributeNode(value = "campaignId")
         }
      )
   }
)
```

Allows to define subgraphs for the associated objects

Load/Fetch Entity Graphs

- **Passing the entity graph as a property**

```
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**

```
EntityGraph<?> eg = em.getEntityGraph("previewCustomerEntityGraph");
List<Customer> customers = em.createNamedQuery("user.findByName",
                                               Customer.class)
        .setParameter("name", name)
        .setHint("javax.persistence.loadgraph", eg)
        .getResultList();
```

Load/Fetch Entity Graphs

- **Load graph**
  - `javax.persistence.loadgraph`
  - If a load graph is used, all attributes that are not specified by the entity graph will keep their default fetch type

- **Fetch graph**
  - `javax.persistence.fetchgraph`
  - 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

Updating Entities

- **Updating Entities**
  - Similar problem for updating operations (even worse…)

```
public interface UserService {
   void saveOrUpdateUser(User user)
                         throws RentalServiceException;
   ...
```

  - Questions:
    - Is only the User updated, or also the rentals which are stored in the user?
    - Are changes made on the movies referenced over rentals stored as well? (comparable to `CascadeType.MERGE`)
    - Are rentals which are no longer in the list of rentals removed from the DB? (comparable to `orphanRemoval=true`)

Data Transfer Objects

- **Data Transfer Objects**
  - Are used to transfer data across layers of your application
  - Only the data needed by the requesting layer is passed, i.e. not all properties need to be defined
    - For different use cases different DTOs could be defined
  - No Lazy Loading Exception surprises
  - Clients are independent of ORM technology used

Sample: User DTO

```
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;
   }

   // Getter
   // Setter
}
```

Serializable only if needed

Remark: The entities used in microservices will also contain the primary keys to the other entities

DTO creation in Service (Java Implementation)

```
public UserDto getUserDataById(Long id) {
   User u = userRepo.findById(id).get();

   List<Long> rentalIds = new ArrayList<>();
   for(Rental r : u.getRentals()) {
      rentalIds.add(r.getId());
   }

   return new UserDTO(u.getId(), u.getName(), u.getFirstName(),
                                                      rentalIds);
}
```

- Must be executed within a transaction
  - Otherwise a lazy loading exception would be thrown

DTO creation in Service (JPA implementation)

```
public UserDto getUserDataById(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;
}
```

```
@NamedQuery(name="User.dataById", query=
   "SELECT NEW ch.fhnw.edu.rental.dtos.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, JOIN u.rentals r WHERE u.id = :id")
```

DTO creation in Service (Mapper)

- **Mapper**
  - Mappers recursively copy data form Java Bean to Java Bean
    => can be used to copy DTOs
  - MapStruct ist an annotation processor for generating type-safe, performant and dependency-free bean mapping code
    - Supports mapping of arbitrary deep object graphs (including collections)
    - Provides automatic type conversions and customer-provided conversions
    - http://mapstruct.org/

DTO creation in Service (Mapper)

- **Mapper Interface**

```
@Mapper(componentModel="spring")
public interface MovieMapper {

   @Mapping(source = "rentals", target = "rentalIds")
   UserDto userToUserDto(User user);

   default Long rentalToLong(Rental r) {
      return r.getId();
   }

}
```

  - Name mappings can be specified using annotations
  - Default method is a custom mapper (e.g. Rental -> Long)

DTO creation in Service (Mapper)

- **Mapper Use**

```
@Autowired
MovieMapper mapper;

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

DTO creation in Service (Mapper)

- **Gradle Integration**

```
dependencies {
 // MapStruct
 implementation("org.mapstruct:mapstruct:1.4.2.Final")
 annotationProcessor("org.mapstruct:mapstruct-processor:1.4.2.Final")
}

tasks.withType(JavaCompile) {
   options.annotationProcessorGeneratedSourcesDirectory =
      file("$buildDir/generated/mapstruct")
}

sourceSets {
   main { java { srcDirs += ["$buildDir/generated/mapstruct"] } }
}
```

  – gradle build then generates the mapper implementations

Data Transfer Object: Pros / Cons

- **Con: Code Duplication**
  - In particular when DTOs have the same fields as domain objects
- **Con: Code to copy attributes back and forth**
  - MapStruct / Orika / JPA

- **Pro: Lazy Loading Problem**
  - You are not caught by a *Lazy Loading Exception*
    - neither on client side
    - nor upon serialization
- **Pro: Triggers Design**
  - Forces you to think about the interface of the remote service façades
  - Information from multiple domain objects can be combined into one DTO
  - Ideal class to add JSON annotations

Java Persistence API

- **Spring Data Repositories**
- **Worksheet: Find JPA Errors**
- **Data Transfer Objects**
- **Summary**

JPA Recommendations

- **Avoid bidirectional associations**
  - Problem with owner / opposite side
  - If you need them, add the consistency ensuring code in the service classes, not in the entities themselves
  - Try to avoid cycles in general
  - If you have to navigate in both directions, provide association in one direction and finder for the other direction
    - either

```
class User {
   @OneToMany
   List<Rental> rentals;
   …
}

findUserByRental(Rental r)
```

    - or

```
class Rental {
   @ManyToOne
   User user;
   …
}

findRentalsByUser(User r)
```

JPA Recommendations

- **Collection fields**
  - Instead of a setters, provide add and remove methods

```
public void add(E obj) {
   Assert.notNull(obj);
   collection.add(obj);
}
```

  - Return a unmodifiable instance with the getter

```
public List<E> getReferences() {
   return Collections.unmodifiableList(coll);
}
```

JPA Recommendations

- **Make SQL great again**
  - In order to apply JPA correctly, you should have profound knowledge in SQL (JPA/Hibernate will not release you from any DB knowledge)
  - The database model should be designed up front (SQL first)

  - High-Performance Hibernate: Devoxx 2016
    https://www.youtube.com/watch?v=BTdTEe9QL5k
  - JOOQ: Get Back In Control of Your SQL
    https://www.jooq.org/