Prüfungscode

# Woche 1

## XML Konfiguration

|  |
| --- |
| <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-4.3.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-4.3.xsd"**>   <**context:property-placeholder location="classpath:application.properties"**/>   <**bean id="renderer1" class="ch.fhnw.edu.eaf.springioc.renderer.StandardOutRenderer"**>  <**property name="messageProvider" ref="provider1"**/>  </**bean**>   <**bean id="renderer2" class="ch.fhnw.edu.eaf.springioc.renderer.StandardOutRenderer"**>  <**property name="messageProvider" ref="provider2"**/>  </**bean**>   <**bean id="renderer3" class="ch.fhnw.edu.eaf.springioc.renderer.StandardOutRenderer"**>  <**property name="messageProvider" ref="provider3"**/>  </**bean**>   <**bean id="provider1" class="ch.fhnw.edu.eaf.springioc.providers.GrueziProvider"**/>   <**bean id="provider2" class="ch.fhnw.edu.eaf.springioc.providers.HelloWorldProvider"**/>   <**bean id="provider3" class="ch.fhnw.edu.eaf.springioc.providers.ExternalizedHelloWorldMessageProvider"**>  <**constructor-arg index="0" value="${helloworld.message}"**/>  </**bean**> </**beans**> |

## Dumme BeanFactory mit XML

|  |
| --- |
| BeanFactory beanFactory = **new** XmlBeanFactory(**new** ClassPathResource(**"/config.xml"**)); MessageRenderer messageRenderer = (MessageRenderer) beanFactory.getBean(**"renderer"**); messageRenderer.render(); |

## Intelligenter ApplicationContext

|  |
| --- |
| ApplicationContext context = **new** ClassPathXmlApplicationContext(**new** String[]{**"config.xml"**});  MessageRenderer messageRenderer1 = (MessageRenderer) context.getBean(**"renderer1"**); messageRenderer1.render(); |

# Woche 2

## Spring Boot mit XML

Mit @ContextConfiguration(location=””) im Unit Test

Ohne irgendwelche Annotationen im Source Code

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.1.xsd"**>   *<!-- load property resources -->* <**context:property-placeholder location="classpath:application.properties"**/>   <**bean id="renderer" class="ch.fhnw.edu.eaf.app.domain.impl.StandardOutRenderer"**>  <**property name="messageProvider" ref="provider"**/>  </**bean**>   <**bean id="provider" class="ch.fhnw.edu.eaf.app.domain.impl.ExternalizedHelloWorldMessageProvider"**>  <**property name="message" value="${helloworld.message}"**/>  </**bean**> </**beans**> |

## Spring Boot mit Annotation-based Configuration

Mit @ContextConfiguration(location=””) im Unit Test

Mit @Component, @Autowired und @Value Annotationen im Source Code

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.1.xsd"**>  *<!-- load property re<sources -->* <**context:property-placeholder location="classpath:application.properties"**/>   <**context:component-scan base-package="ch.fhnw.edu.eaf.app.domain"**/> </**beans**> |

## Spring Boot mit Java-based Configuration

Mit @ContextConfiguration(classes=””) im Unit Test

Kein XML mehr 🡪 Das restliche XML wird durch eine Java Klasse ersetzt

|  |
| --- |
| @Configuration @ComponentScan @PropertySource(**"classpath:application.properties"**) **public class** BeanConfiguration { } |

## Spring Boot mit Convention over Configuration

Mit @SpringBootTest(classes=””) im Unit Test

Einführen von @SpringBootApplication (Resultiert in @SpringBootConfiguration und @ComponentScan)

# Woche 3

## Spring JdbcTemplate

|  |
| --- |
| @Autowired **private** JdbcTemplate **template**;  **public** Optional<Movie> findById(Long id) {  Map<String, Object> res = **template**.queryForMap(  **"select \* from MOVIES where MOVIE\_ID = ?"**, id);  **long** priceCategoryId= (Long)res.get(**"PRICECATEGORY\_FK"**);  Movie m = **new** Movie(  (Long)res.get(**"MOVIE\_ID"**),  (String)res.get(**"MOVIE\_TITLE"**),  ((java.sql.Date)res.get(**"MOVIE\_RELEASEDATE"**)).toLocalDate(),  (Boolean)res.get(**"MOVIE\_RENTED"**),  priceCategoryRepo.findById(priceCategoryId).get());  **return** Optional.of(m); } |

## Spring NamedParameterJdbcTemplate

|  |
| --- |
| @Autowired **private** NamedParameterJdbcTemplate **namedTemplate**;  **public** Optional<Movie> findById(Long id) {  String query = **"select \* from MOVIES where MOVIE\_ID = :id"**;  Map<String, Long> params= **new** HashMap<>();  params.put(**"id"**, id);  Map<String, Object> res = **namedTemplate**.queryForMap(query,params);  **long** priceCategoryId= (Long) res.get(**"PRICECATEGORY\_FK"**);  Movie m = **new** Movie(  (Long) res.get(**"MOVIE\_ID"**),  (String) res.get(**"MOVIE\_TITLE"**),  ((java.sql.Date) res.get(**"MOVIE\_RELEASEDATE"**)).toLocalDate(),  (Boolean) res.get(**"MOVIE\_RENTED"**),  priceCategoryRepo.findById(priceCategoryId).get());  **return** Optional.of(m); } |

## Spring DAO Repository User

|  |
| --- |
| **package** ch.fhnw.edu.rental.persistence.jdbc;  **import** ch.fhnw.edu.rental.model.Rental; **import** ch.fhnw.edu.rental.model.User; **import** ch.fhnw.edu.rental.persistence.UserRepository; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.dao.EmptyResultDataAccessException; **import** org.springframework.jdbc.core.JdbcTemplate; **import** org.springframework.jdbc.support.GeneratedKeyHolder; **import** org.springframework.jdbc.support.KeyHolder; **import** org.springframework.stereotype.Component;  **import** java.sql.PreparedStatement; **import** java.sql.ResultSet; **import** java.sql.SQLException; **import** java.sql.Statement; **import** java.util.List; **import** java.util.Optional;  @Component **public class** JdbcUserRepository **implements** UserRepository {   @Autowired  **private** JdbcRentalRepository **rentalRepository**;   @Autowired  **private** JdbcTemplate **template**;   @Override  **public** List<User> findByLastName(String lastName) {  String sql = **"SELECT** *\** **FROM USERS WHERE USER\_NAME = ?"**;  **return template**.query(sql, (rs, row) -> createUser(rs), lastName);  }   @Override  **public** List<User> findByFirstName(String firstName) {  String sql = **"SELECT** *\** **FROM USERS WHERE USER\_FIRSTNAME = ?"**;  **return template**.query(sql, (rs, row) -> createUser(rs), firstName);  }   @Override  **public** List<User> findByEmail(String email) {  String sql = **"SELECT** *\** **FROM USERS WHERE USER\_EMAIL = ?"**;  **return template**.query(sql, (rs, row) -> createUser(rs), email);  }   @Override  **public** Optional<User> findById(Long aLong) {  String sql = **"SELECT** *\** **FROM USERS WHERE USER\_ID = ? LIMIT 1"**;  **return** Optional.*of*(**template**.queryForObject(sql, **new** Object[]{aLong}, (rs, row) -> createUser(rs)));  }   @Override  **public** List<User> findAll() {  String sql = **"SELECT** *\** **FROM USERS"**;  **return template**.query(sql, (rs, row) -> createUser(rs));  }   @Override  **public** User save(User user) {  **if** (user.getId() == **null**) {  String sql = **"INSERT INTO USERS(USER\_EMAIL, USER\_FIRSTNAME, USER\_NAME) VALUES (?, ?, ?)"**;  KeyHolder keyHolder = **new** GeneratedKeyHolder();  **template**.update(connection -> {  **final** PreparedStatement statement = connection.prepareStatement(sql, Statement.***RETURN\_GENERATED\_KEYS***);  statement.setString(1, user.getEmail());  statement.setString(2, user.getFirstName());  statement.setString(3, user.getLastName());  **return** statement;  }, keyHolder);  user.setId(keyHolder.getKey().longValue());  } **else** {  String sql = **"UPDATE USERS SET USER\_EMAIL = ?, USER\_FIRSTNAME = ?, USER\_NAME = ?"**;  **template**.update(sql, **new** Object[]{user.getEmail(), user.getFirstName(), user.getLastName()});  }  **return** user;  }   @Override  **public void** deleteById(Long aLong) {  String sql = **"DELETE FROM USERS WHERE USER\_ID = ?"**;  **template**.update(sql, **new** Object[]{aLong});  }   @Override  **public void** delete(User entity) {  **for** (Rental r : entity.getRentals()) {  **rentalRepository**.delete(r);  }  deleteById(entity.getId());  entity.setId(**null**);  }   @Override  **public boolean** existsById(Long aLong) {  *// It's a bad idea to use count(\*) because it can result in a full table scan. Instead use the queryForObject exception handling* **try** {  findById(aLong);  **return true**;  } **catch** (EmptyResultDataAccessException exception) {  **return false**;  }  }   @Override  **public long** count() {  String sql = **"SELECT** *COUNT***(***\****) FROM USERS"**;  **return template**.queryForObject(sql, Long.**class**);  }   **private** User createUser(ResultSet resultSet) **throws** SQLException {  Long id = resultSet.getLong(**"USER\_ID"**);  User user = **new** User();  user.setId(id);   user.setLastName(resultSet.getString(**"USER\_NAME"**));  user.setFirstName(resultSet.getString(**"USER\_FIRSTNAME"**));  user.setEmail(resultSet.getString(**"USER\_EMAIL"**));  user.setRentals(**rentalRepository**.findByUser(user));  **return** user;  } } |

## Spring DAO Repository Rental

|  |
| --- |
| **package** ch.fhnw.edu.rental.persistence.jdbc;  **import** ch.fhnw.edu.rental.model.Movie; **import** ch.fhnw.edu.rental.model.Rental; **import** ch.fhnw.edu.rental.model.User; **import** ch.fhnw.edu.rental.persistence.RentalRepository; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.dao.EmptyResultDataAccessException; **import** org.springframework.jdbc.core.JdbcTemplate; **import** org.springframework.jdbc.support.GeneratedKeyHolder; **import** org.springframework.jdbc.support.KeyHolder; **import** org.springframework.stereotype.Component;  **import** java.sql.PreparedStatement; **import** java.sql.ResultSet; **import** java.sql.SQLException; **import** java.sql.Statement; **import** java.time.format.DateTimeFormatter; **import** java.util.HashMap; **import** java.util.List; **import** java.util.Optional;  @Component **public class** JdbcRentalRepository **implements** RentalRepository {   **private static final** DateTimeFormatter ***dateTimeFormatter*** = DateTimeFormatter.*ofPattern*(**"yyyy-MM-dd"**);   @Autowired  **private** JdbcUserRepository **userRepository**;   @Autowired  **private** JdbcMovieRepository **movieRepository**;   @Autowired  **private** JdbcTemplate **template**;   @Override  **public** List<Rental> findByUser(User user) {  String sql = **"SELECT** *\** **FROM RENTALS WHERE USER\_ID = ?"**;  **return template**.query(sql, (rs, row) -> createRentals(rs, user), user.getId());  }   @Override  **public** Optional<Rental> findById(Long aLong) {  String sql = **"SELECT** *\** **FROM RENTALS WHERE RENTAL\_ID = ? LIMIT 1"**;  **return** Optional.*of*(**template**.queryForObject(sql, **new** Object[]{aLong}, (rs, row) -> createRentals(rs)));  }   @Override  **public** List<Rental> findAll() {  String sql = **"SELECT** *\** **FROM RENTALS"**;  **return template**.query(sql, (rs, row) -> createRentals(rs));  }   @Override  **public** Rental save(Rental rental) {  **if** (rental.getId() == **null**) {  String sql = **"INSERT INTO RENTALS(RENTAL\_RENTALDATE, RENTAL\_RENTALDAYS, USER\_ID, MOVIE\_ID) VALUES (?, ?, ?, ?)"**;  KeyHolder keyHolder = **new** GeneratedKeyHolder();  **template**.update(connection -> {  **final** PreparedStatement statement = connection.prepareStatement(sql, Statement.***RETURN\_GENERATED\_KEYS***);  statement.setString(1, ***dateTimeFormatter***.format(rental.getRentalDate()));  statement.setInt(2, rental.getRentalDays());  statement.setLong(3, rental.getUser().getId());  statement.setLong(4, rental.getMovie().getId());  **return** statement;  }, keyHolder);  rental.setId(keyHolder.getKey().longValue());  } **else** {  String sql = **"UPDATE RENTALS SET RENTAL\_RENTALDATE = ?, RENTAL\_RENTALDAYS = ?, USER\_ID = ?, MOVIE\_ID = ?"**;  **template**.update(sql, **new** Object[]{***dateTimeFormatter***.format(rental.getRentalDate()), rental.getRentalDays(), rental.getUser().getId(), rental.getMovie().getId()});  }  **return** rental;  }   @Override  **public void** deleteById(Long aLong) {  String sql = **"DELETE FROM RENTALS WHERE RENTAL\_ID = ?"**;  **template**.update(sql, **new** Object[]{aLong});  }   @Override  **public void** delete(Rental entity) {  deleteById(entity.getId());  entity.setId(**null**);  }   @Override  **public boolean** existsById(Long aLong) {  *// It's a bad idea to use count(\*) because it can result in a full table scan. Instead use the queryForObject exception handling* **try** {  findById(aLong);  **return true**;  } **catch** (EmptyResultDataAccessException exception) {  **return false**;  }  }   @Override  **public long** count() {  String sql = **"SELECT** *COUNT***(***\****) FROM RENTALS"**;  **return template**.queryForObject(sql, Long.**class**);  }   **private** Rental createRentals(ResultSet resultSet) **throws** SQLException {  Long id = resultSet.getLong(**"USER\_ID"**);  Optional<User> user = **userRepository**.findById(id);  **if** (!user.isPresent()) {  **throw new** IllegalStateException(**"Unable to find the user in the database - the reference integrity was violated!"**);  }  *//return createRentals(resultSet, user.get());* **for** (Rental rental : user.get().getRentals()) {  **if** (rental.getId().equals(id)) {  **return** rental;  }  }  **throw new** RuntimeException(**"Inconsistent user"**);  }   **private** Rental createRentals(ResultSet resultSet, User user) **throws** SQLException {  Long id = resultSet.getLong(**"RENTAL\_ID"**);  Rental rental = **new** Rental();  rental.setId(id);   Optional<Movie> movie = **movieRepository**.findById(resultSet.getLong(**"MOVIE\_ID"**));  **if** (!movie.isPresent()) {  **throw new** IllegalStateException(**"Unable to find the movie in the database - the reference integrity was violated!"**);  }   rental.setRentalDays(resultSet.getInt(**"RENTAL\_RENTALDAYS"**));  rental.setUser(user);  rental.setMovie(movie.get());  **return** rental;  } } |

## Spring DAO Repository Categories

|  |
| --- |
| **package** ch.fhnw.edu.rental.persistence.jdbc;  **import** ch.fhnw.edu.rental.model.PriceCategory; **import** ch.fhnw.edu.rental.model.PriceCategoryChildren; **import** ch.fhnw.edu.rental.model.PriceCategoryNewRelease; **import** ch.fhnw.edu.rental.model.PriceCategoryRegular; **import** ch.fhnw.edu.rental.persistence.PriceCategoryRepository; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.dao.EmptyResultDataAccessException; **import** org.springframework.jdbc.core.JdbcTemplate; **import** org.springframework.stereotype.Component;  **import** java.security.InvalidParameterException; **import** java.sql.ResultSet; **import** java.sql.SQLException; **import** java.util.List; **import** java.util.Optional;  @Component **public class** JdbcPriceCategoryRepository **implements** PriceCategoryRepository {   @Autowired  **private** JdbcTemplate **template**;   @Override  **public** Optional<PriceCategory> findById(Long aLong) {  String sql = **"SELECT** *\** **FROM PRICECATEGORIES WHERE PRICECATEGORY\_ID = ? LIMIT 1"**;  **return** Optional.*of*(**template**.queryForObject(sql, **new** Object[]{aLong}, (rs, row) -> createPriceCategory(rs)));  }   @Override  **public** List<PriceCategory> findAll() {  String sql = **"SELECT** *\** **FROM PRICECATEGORIES"**;  **return template**.query(sql, (rs, row) -> createPriceCategory(rs));  }   @Override  **public** PriceCategory save(PriceCategory priceCategory) {  **throw new** UnsupportedOperationException(**"You are not allowed to save or create static database entries!"**);  }   @Override  **public void** deleteById(Long aLong) {  **throw new** UnsupportedOperationException(**"You are not allowed to delete static database entries!"**);  }   @Override  **public void** delete(PriceCategory entity) {  **throw new** UnsupportedOperationException(**"You are not allowed to delete static database entries!"**);   }   @Override  **public boolean** existsById(Long aLong) {  *// It's a bad idea to use count(\*) because it can result in a full table scan. Instead use the queryForObject exception handling* **try** {  findById(aLong);  **return true**;  } **catch** (EmptyResultDataAccessException exception) {  **return false**;  }  }   @Override  **public long** count() {  String sql = **"SELECT** *COUNT***(***\****) FROM PRICECATEGORIES"**;  **return template**.queryForObject(sql, Long.**class**);  }   **private** PriceCategory createPriceCategory(ResultSet resultSet) **throws** SQLException {  PriceCategory priceCategory;   **switch** (resultSet.getString(**"PRICECATEGORY\_TYPE"**)) {  **case "Regular"**:  priceCategory = **new** PriceCategoryRegular();  **break**;  **case "Children"**:  priceCategory = **new** PriceCategoryChildren();  **break**;  **case "NewRelease"**:  priceCategory = **new** PriceCategoryNewRelease();  **break**;  **default**:  **throw new** InvalidParameterException(**"Invalid price category type!"**);  }   priceCategory.setId(resultSet.getLong(**"PRICECATEGORY\_ID"**));  **return** priceCategory;  } } |

## Alternatives Auslesen des Primary Key

|  |
| --- |
| SimpleJdbcInsertinsert = **new** SimpleJdbcInsert(**template**.getDataSource())  .withTableName(**"MOVIES"**)  .usingGeneratedKeyColumns(**"MOVIE\_ID"**); Map<String, Object> parameters = **new** HashMap<>(); parameters.put(**"MOVIE\_TITLE"**, movie.getTitle()); parameters.put(**"MOVIE\_RELEASEDATE"**, movie.getReleaseDate()); parameters.put(**"MOVIE\_RENTED"**, movie.isRented()); parameters.put(**"PRICECATEGORY\_FK"**, movie.getPriceCategory().getId()); Number id = insert.executeAndReturnKey(parameters); movie.setId(id.longValue()); |

# Woche 4

## JPA Relationen Rental

Many Rentals to One User

|  |
| --- |
| @Entity **public class** Rental {  @Id  **private** intid;  @ManyToOne*// Rental is the owner of the relationship* @JoinColumn(name = **"USER\_FK"**) *// optional* **private** User **user**; } |

## JPA Relationen User

One User To Many Rentals

|  |
| --- |
| @Entity **public class** User {  @OneToMany(mappedBy=**"user"**)**private** Collection<Rental> **rentals**;  *// this is the inverse side of the relationship*  **public** Collection<Rental> getRentals() {  **return rentals**;  }  **public void** setRentals(Collection<Rental> rentals) {  **this**.**rentals**= rentals;  } } |

## JPA Discriminators

In der Basisklasse

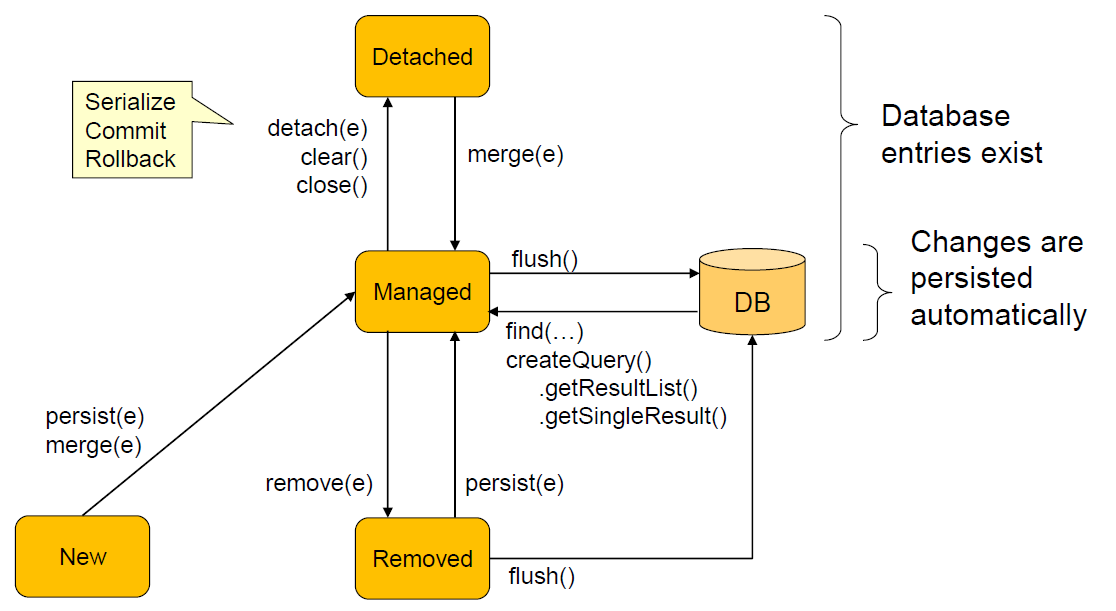
|  |
| --- |
| @Entity @Table(name = **"PRICECATEGORIES"**) @DiscriminatorColumn(name=**"PRICECATEGORY\_TYPE"**) **public abstract class** PriceCategory {   @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  @Column(name = **"PRICECATEGORY\_ID"**)  **private** Long **id**; } |

In der vererbten Klasse

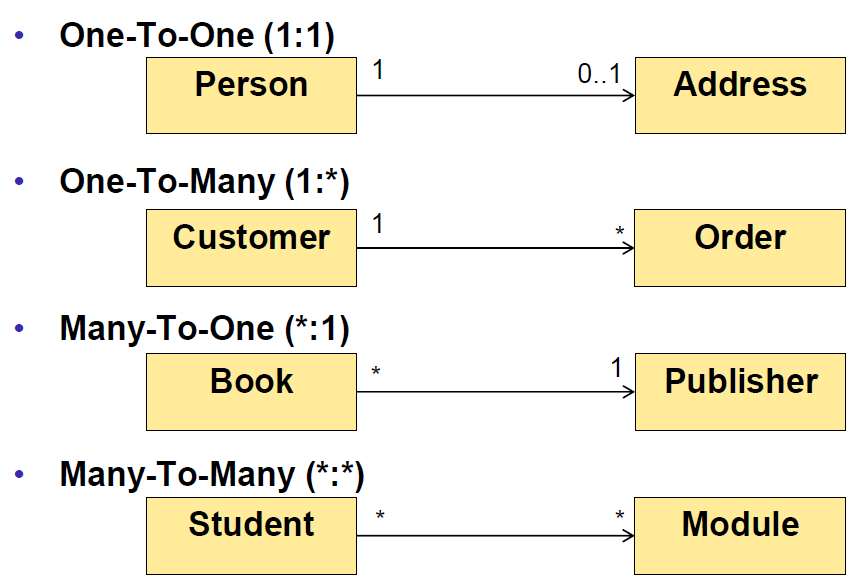
|  |
| --- |
| @Entity @DiscriminatorValue(**"Children"**) **public class** PriceCategoryChildren **extends** PriceCategory { } |

# Woche 5

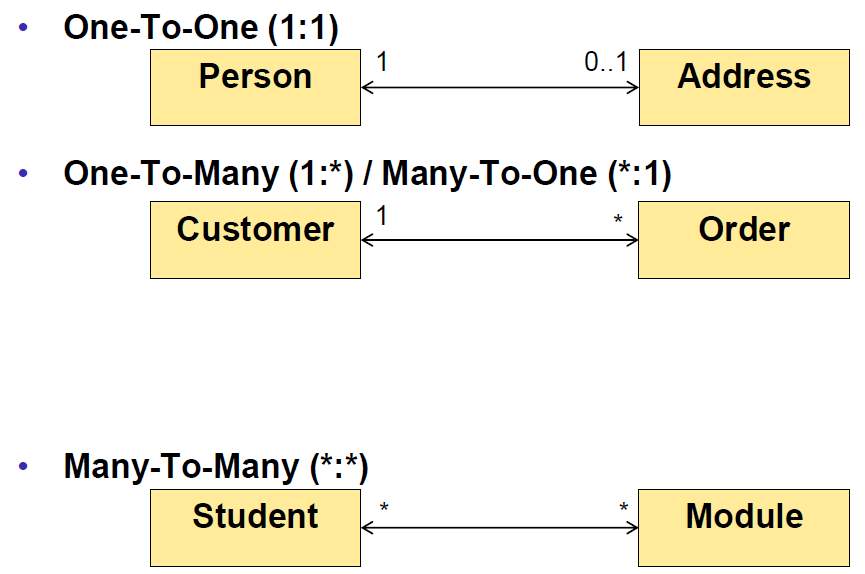
## Entity Bean Lifecycle



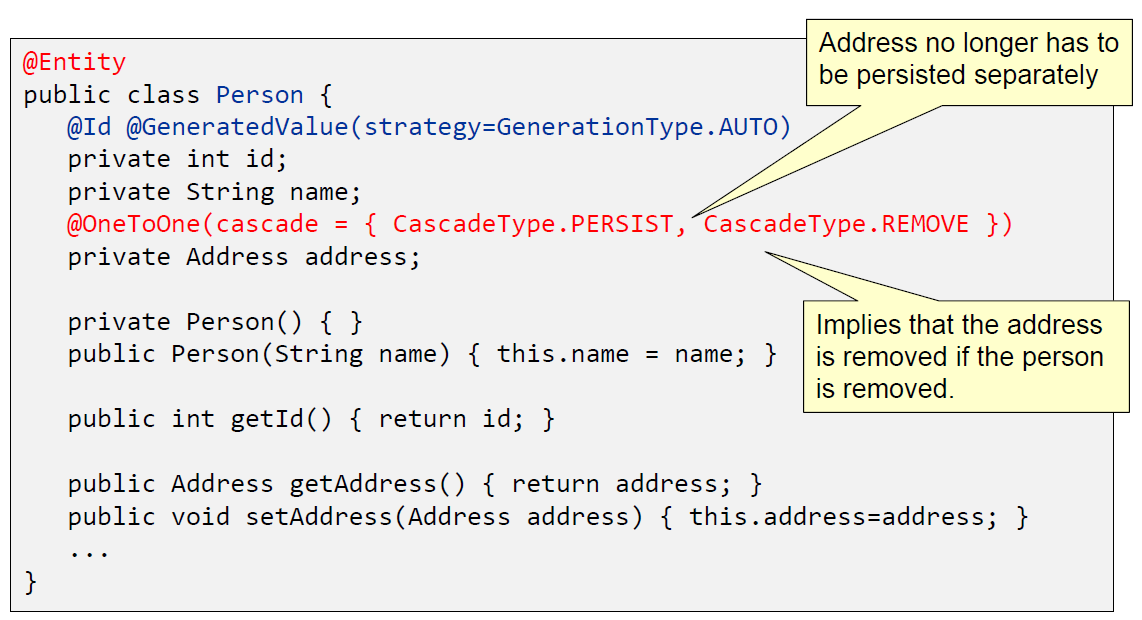
## Unidirektionale Relationen



## Bidirektionale Relationen

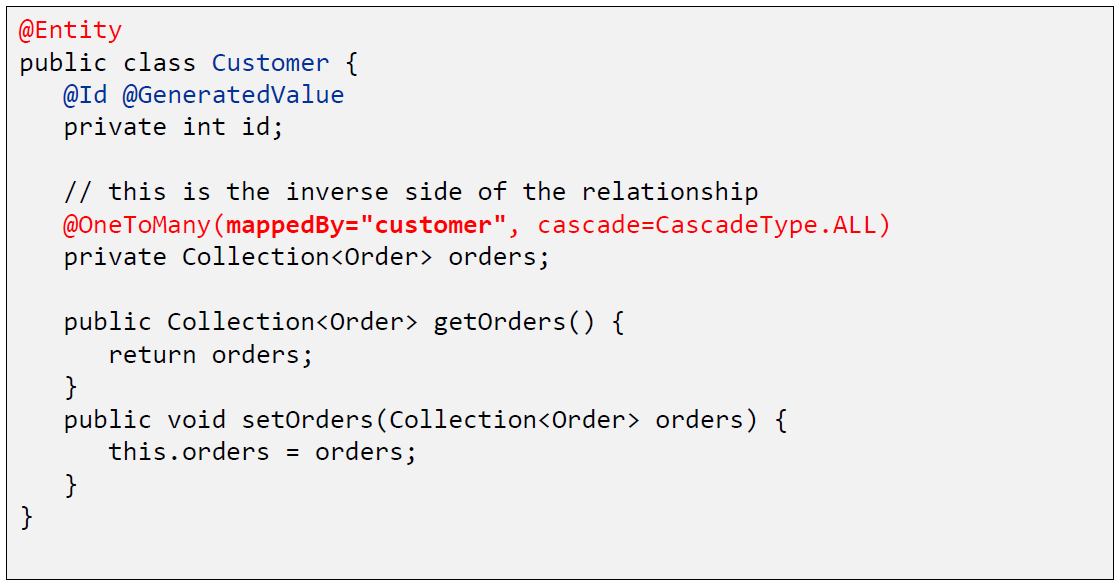


## Cascading Operations

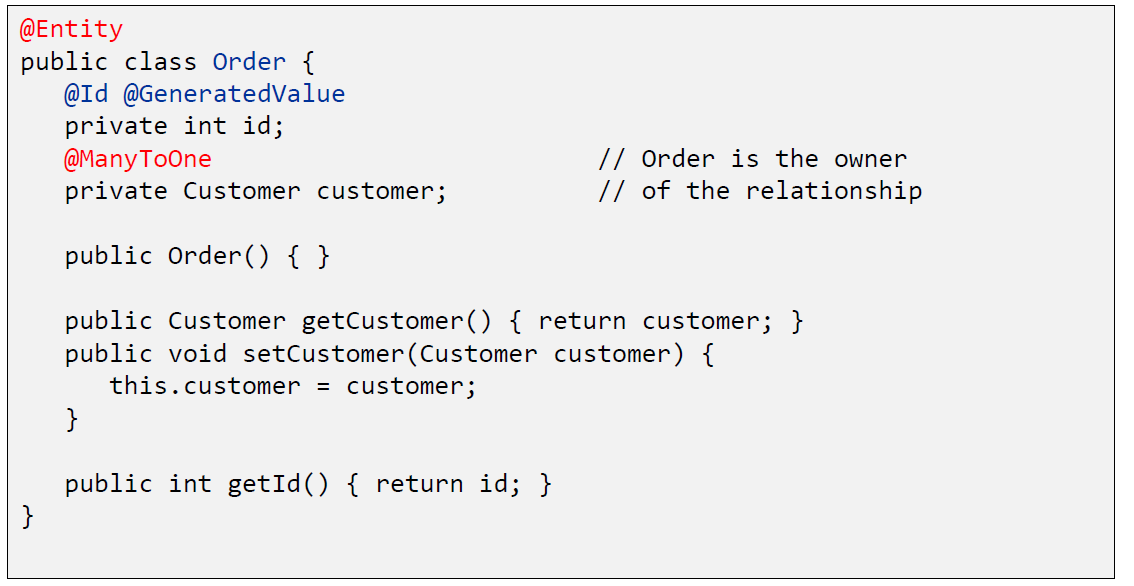


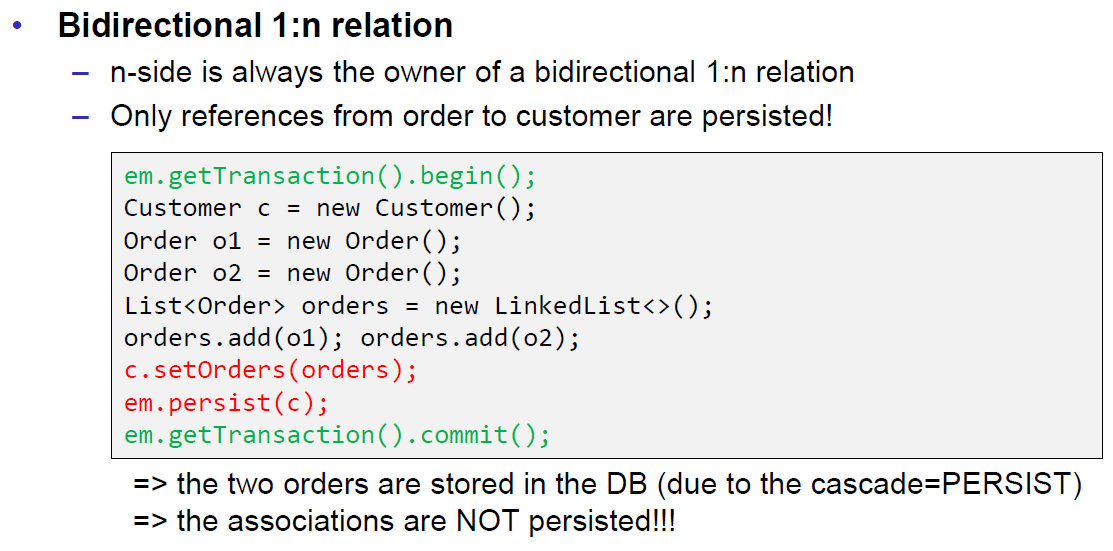
## OneToMany Bidirectional (Doesn’t feel natural)

Customer (Inverse)

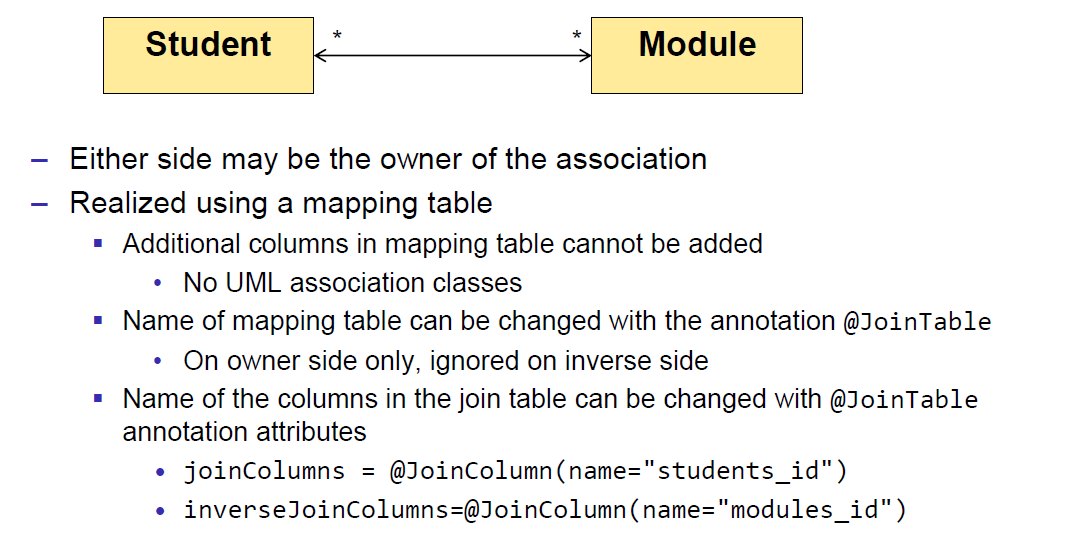


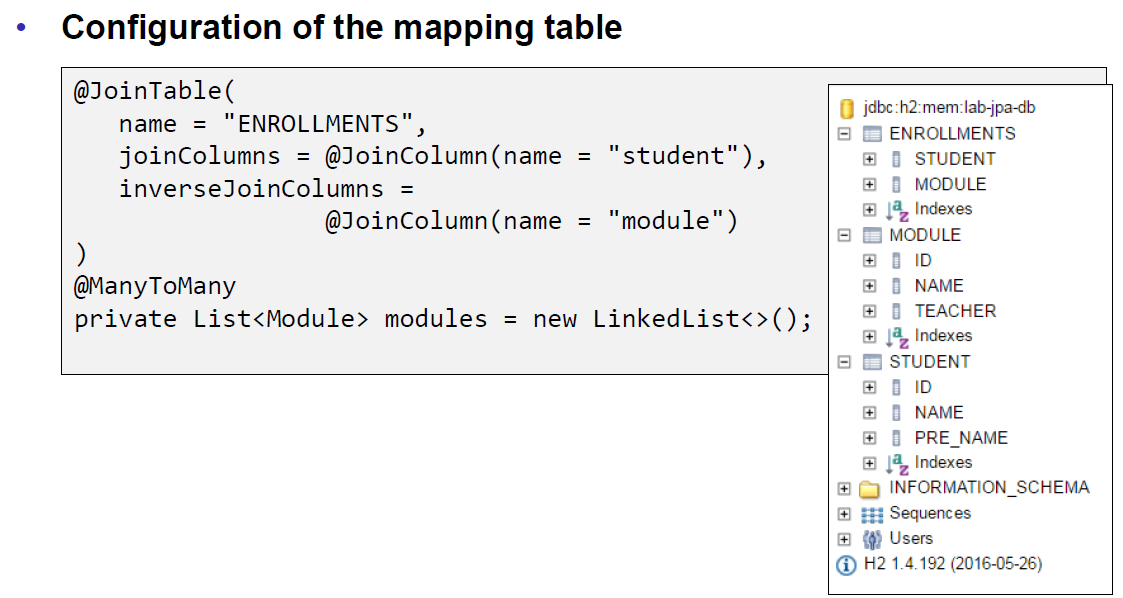
Order (Owner)



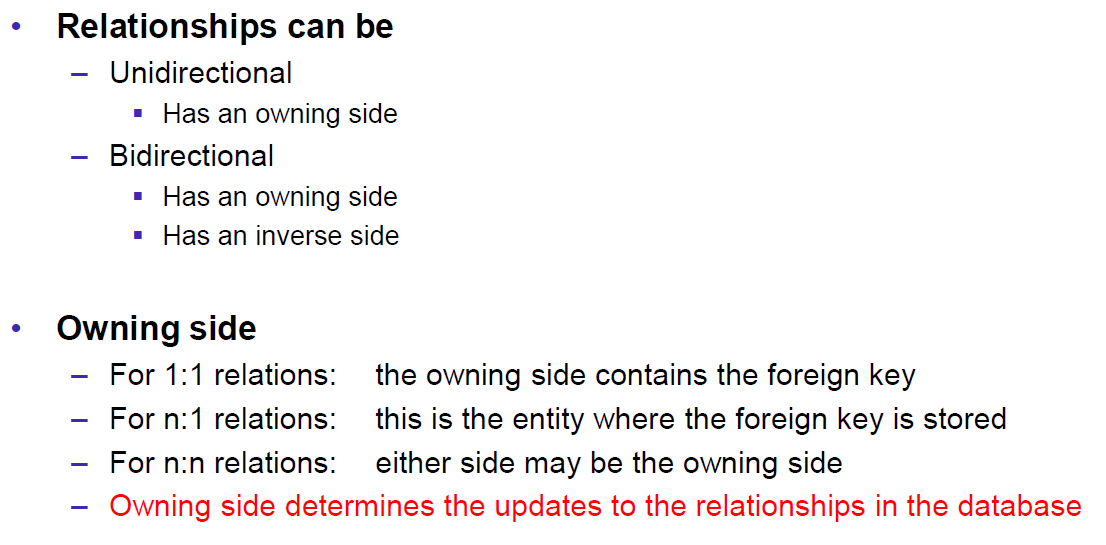


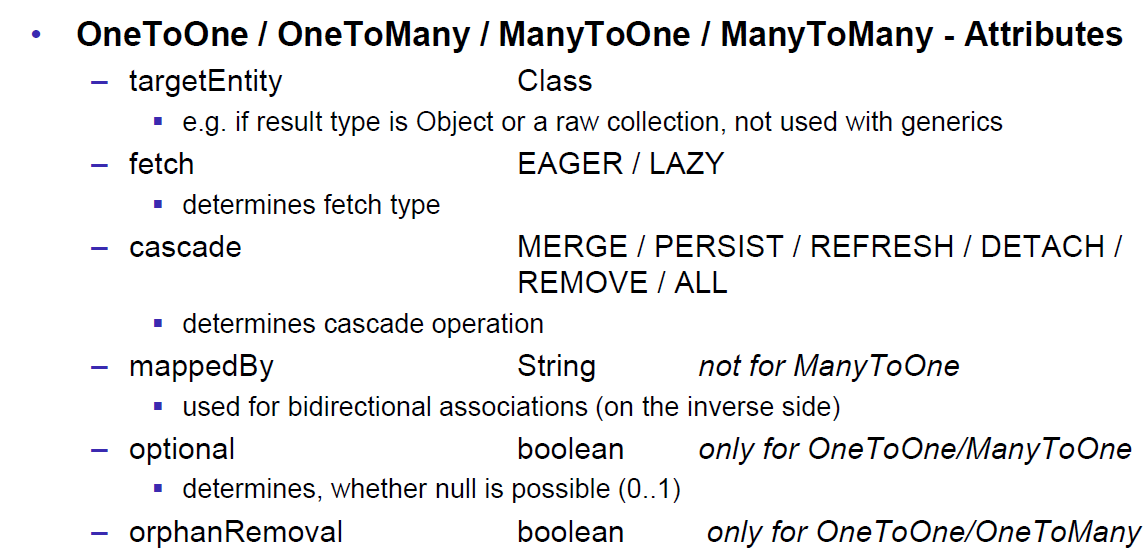
## ManyToMany Bidirectional





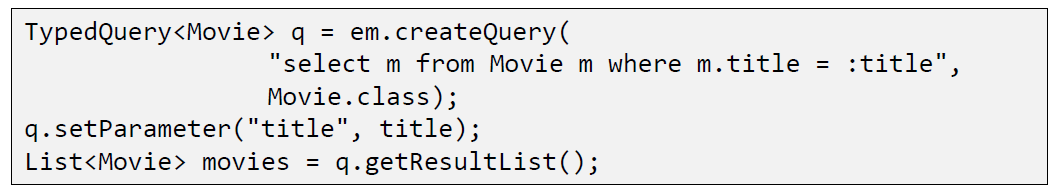
## Summary



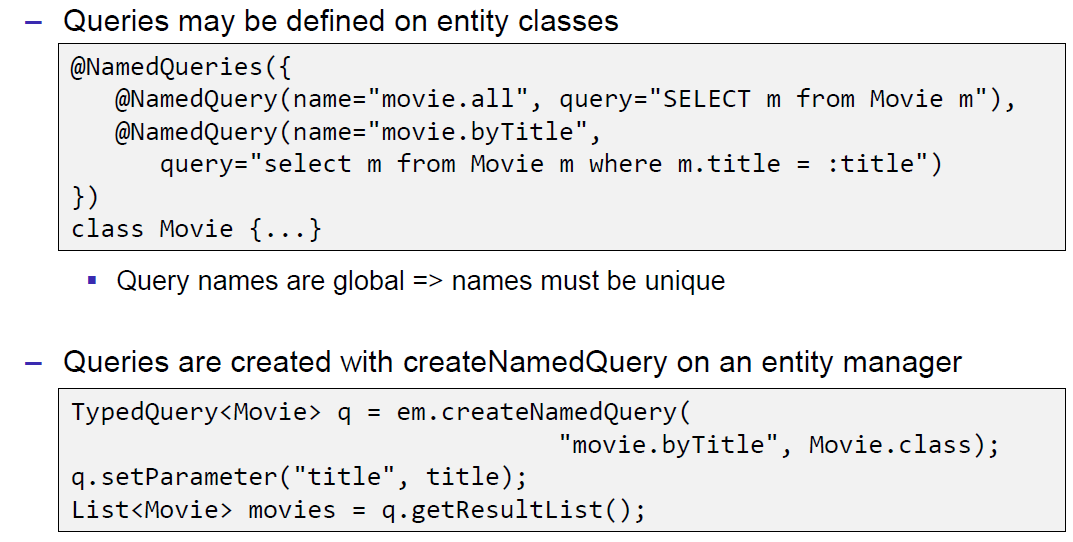


# Woche 6

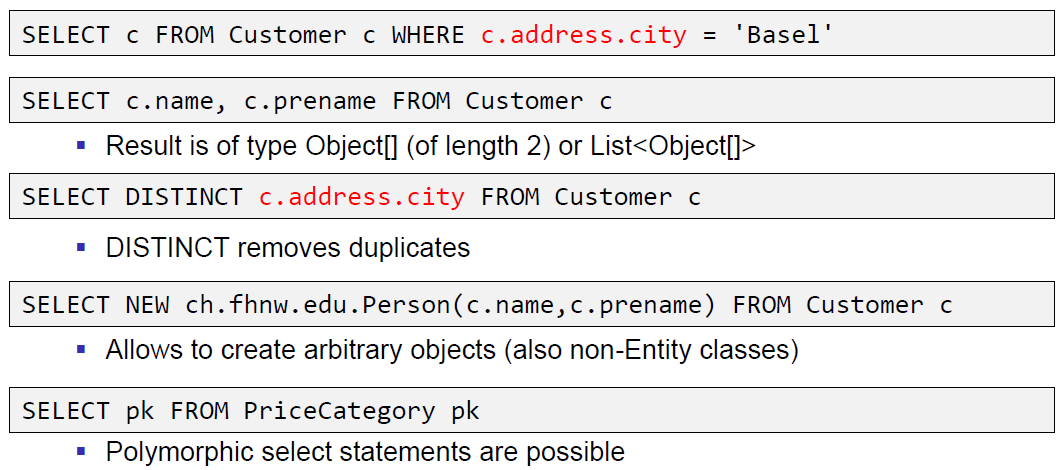
## Typed Queries



## Named Queries



## Query Beispiele



# Woche 7

## Entität User

|  |
| --- |
| @Entity @Table(name = **"USERS"**) **public class** User {   @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  @Column(name = **"USER\_ID"**)  **private** Long **id**;   @Column(name = **"USER\_NAME"**)  **private** String **lastName**;   @Column(name = **"USER\_FIRSTNAME"**)  **private** String **firstName**;   @Column(name = **"USER\_EMAIL"**)  **private** String **email**;   @OneToMany(mappedBy = **"user"**, cascade = {CascadeType.***REMOVE***}, fetch = FetchType.***EAGER***)  **private** List<Rental> **rentals**; } |

## Entität Rental

|  |
| --- |
| @Entity @Table(name = **"RENTALS"**) **public class** Rental {   @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  @Column(name = **"RENTAL\_ID"**)  **private** Long **id**;   @OneToOne  @JoinColumn(name = **"MOVIE\_ID"**)  **private** Movie **movie**;   @ManyToOne  @JoinColumn(name = **"USER\_ID"**)  **private** User **user**;   @Column(name = **"RENTAL\_RENTALDATE"**)  **private** LocalDate **rentalDate**;   @Column(name = **"RENTAL\_RENTALDAYS"**)  **private int rentalDays**; } |

## Entität Movie

|  |
| --- |
| @Entity @Table(name = **"MOVIES"**) **public class** Movie {   @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  @Column(name = **"MOVIE\_ID"**)  **private** Long **id**;   @Column(name = **"MOVIE\_TITLE"**)  **private** String **title**;   @Column(name = **"MOVIE\_RELEASEDATE"**)  **private** LocalDate **releaseDate**;   @Column(name = **"MOVIE\_RENTED"**)  **private boolean rented**;   @OneToOne  @JoinColumn(name = **"PRICECATEGORY\_FK"**)  **private** PriceCategory **priceCategory**; } |

## Entität PriceCategory

|  |
| --- |
| @Entity @Table(name = **"PRICECATEGORIES"**) @DiscriminatorColumn(name=**"PRICECATEGORY\_TYPE"**) **public abstract class** PriceCategory {   @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  @Column(name = **"PRICECATEGORY\_ID"**)  **private** Long **id**; } |

## Entität PriceCategoryChildren

|  |
| --- |
| @Entity @DiscriminatorValue(**"Children"**) **public class** PriceCategoryChildren **extends** PriceCategory { } |

## Generisches JPA Repository

|  |
| --- |
| **public abstract class** JpaBaseRepository<T> **implements** Repository<T, Long> {   **private final** EntityManager **entityManager**;   **private final** Class<T> **entityClass**;   **public** JpaBaseRepository(EntityManager entityManager, Class<T> entityClass) {  **this**.**entityManager** = entityManager;  **this**.**entityClass** = entityClass;  }   **protected final** EntityManager getEm() {  **return entityManager**;  }   @Override  **public** Optional<T> findById(Long id) {  **return** Optional.*ofNullable*(getEm().find(**entityClass**, id));  }   @Override  **public** List<T> findAll() {  TypedQuery<T> query = getEm().createQuery(**"SELECT item FROM "** + **entityClass**.getSimpleName() + **" item"**, **entityClass**);  **return** query.getResultList();  }   @Override  **public** T save(T t) {  **return** getEm().merge(t);  }   @Override  **public void** deleteById(Long aLong) {  getEm().remove(getEm().getReference(**entityClass**, aLong));  }   @Override  **public void** delete(T entity) {  getEm().remove(getEm().merge(entity));  }   @Override  **public boolean** existsById(Long aLong) {  **return** findById(aLong).isPresent();  }   @Override  **public long** count() {  CriteriaBuilder builder = **entityManager**.getCriteriaBuilder();  CriteriaQuery<Long> query = builder.createQuery(Long.**class**);  query.select(builder.count(query.from(**entityClass**)));  **return entityManager**.createQuery(query).getSingleResult();  } } |

## User JPA Repository

|  |
| --- |
| @Repository **public class** JpaUserRepository **extends** JpaBaseRepository<User> **implements** UserRepository {   @Autowired  **public** JpaUserRepository(EntityManager entityManager) {  **super**(entityManager, User.**class**);  }   @Override  **public** List<User> findByLastName(String lastName) {  CriteriaBuilder builder = getEm().getCriteriaBuilder();  CriteriaQuery<User> query = builder.createQuery(User.**class**);  Root<User> root = query.from(User.**class**);  query.select(root).where(builder.equal(root.get(**"lastName"**), lastName));  **return** getEm().createQuery(query).getResultList();  }   @Override  **public** List<User> findByFirstName(String firstName) {  CriteriaBuilder builder = getEm().getCriteriaBuilder();  CriteriaQuery<User> query = builder.createQuery(User.**class**);  Root<User> root = query.from(User.**class**);  query.select(root).where(builder.equal(root.get(**"firstName"**), firstName));  **return** getEm().createQuery(query).getResultList();  }   @Override  **public** List<User> findByEmail(String email) {  CriteriaBuilder builder = getEm().getCriteriaBuilder();  CriteriaQuery<User> query = builder.createQuery(User.**class**);  Root<User> root = query.from(User.**class**);  query.select(root).where(builder.equal(root.get(**"email"**), email));  **return** getEm().createQuery(query).getResultList();  } } |

# Woche 8, 9 und 10

## Registry Application

|  |
| --- |
| */\* # Set the port (Default!) server.port=8761 # Do not register itself eureka.client.register-with-eureka=false eureka.client.fetch-registry=false # No cluster eureka.server.enable-self-preservation=false  \*/* @SpringBootApplication @EnableEurekaServer **public class** RegistryApplication {   **public static void** main(String[] args) {  SpringApplication.*run*(RegistryApplication.**class**, args);  } } |

## Rental Application

|  |
| --- |
| */\* spring.application.name=rentalmanagement  \*/* @SpringBootApplication @EnableDiscoveryClient @EnableCaching **public class** RentalmgmtApplication {   **public static void** main(String[] args) {  SpringApplication.*run*(RentalmgmtApplication.**class**, args);  }    @Bean  @LoadBalanced  **public** RestTemplate restTemplate() {  **return new** RestTemplate();  } } |

## Entität Rental

|  |
| --- |
|  |

## DTO RentalDTO

|  |
| --- |
|  |

## DTO UserDTO

|  |
| --- |
|  |

## DTO MovieDTO

|  |
| --- |
|  |

## DTO PriceCategoryDTO

|  |
| --- |
|  |

## Controller Users

|  |
| --- |
|  |

## Controller Movies

|  |
| --- |
|  |

## Controller Rentals

|  |
| --- |
| @RestController @RequestMapping(**"/rentals"**) **public class** RentalController {  **private** Logger **log** = LoggerFactory.*getLogger*(**this**.getClass());   @Value(value = **"${microservice.usermanagement:usermanagement}"**)  **private** String **userService**;   @Value(value = **"${microservice.moviemanagement:moviemanagement}"**)  **private** String **movieService**;   @Autowired  **private** RestTemplate **restTemplate**;   @Autowired  **private** RentalRepository **rentalRepository**;   @GetMapping  **public** ResponseEntity<List<RentalDTO>> findAll() {  List<RentalDTO> dtos = **new** ArrayList<RentalDTO>();  Sort sort = **new** Sort(Direction.***ASC***, **"id"**);  List<Rental> rentals = **rentalRepository**.findAll(sort);  rentals.forEach(rental -> dtos.add(createRentalDTO(rental)));  **log**.debug(**"Found "** + rentals.size() + **" rentals"**);  **return new** ResponseEntity<List<RentalDTO>>(dtos, HttpStatus.***OK***);  }   @Cacheable(value = **"rentals"**, key = **"#id"**)  @GetMapping(value = **"/{id}"**)  **public** ResponseEntity<RentalDTO> findById(@PathVariable Long id) {  Optional<Rental> rentalOptional = **rentalRepository**.findById(id);  **if** (rentalOptional.isPresent()) {  Rental rental = rentalOptional.get();  RentalDTO rentalDTO = createRentalDTO(rental);  **log**.debug(**"Found rental with id="** + rental.getId());  **return new** ResponseEntity<RentalDTO>(rentalDTO, HttpStatus.***OK***);  }  **return new** ResponseEntity<RentalDTO>(HttpStatus.***NOT\_FOUND***);  }   @PostMapping  **public** ResponseEntity<Rental> create(@Valid @RequestBody Rental rental, BindingResult result) {  **if** (result.hasErrors()) {  **return new** ResponseEntity<Rental>(HttpStatus.***PRECONDITION\_FAILED***);  }  rental = **rentalRepository**.save(rental);  **log**.debug(**"Created rental with id="** + rental.getId());  **return new** ResponseEntity<Rental>(rental, HttpStatus.***CREATED***);  }   @CacheEvict(value = **"rentals"**, key = **"#id"**, beforeInvocation = **true**)  @PutMapping(value = **"/{id}"**)  **public** ResponseEntity<Rental> update(@RequestBody Rental newRental, @PathVariable Long id) {  Optional<Rental> rentalOptional = **rentalRepository**.findById(id);  **if** (rentalOptional.isPresent()) {  Rental rental = rentalOptional.get();  rental.setRentalDate(newRental.getRentalDate());  rental.setRentalDays(newRental.getRentalDays());  **rentalRepository**.save(rental);  **log**.debug(**"Updated rental with id="** + rental.getId());  **return new** ResponseEntity<Rental>(rental, HttpStatus.***OK***);  }  **return new** ResponseEntity<Rental>(HttpStatus.***NOT\_FOUND***);  }   @CacheEvict(value = **"rentals"**, key = **"#id"**)  @DeleteMapping(value = **"/{id}"**)  **public** ResponseEntity<String> delete(@PathVariable Long id) {  Optional<Rental> rentalOptional = **rentalRepository**.findById(id);  **if** (rentalOptional.isPresent()) {  Rental rental = rentalOptional.get();  **rentalRepository**.delete(rental);  **log**.debug(**"Deleted rental with id="** + id);  **return new** ResponseEntity<String>(HttpStatus.***OK***);  }  **return new** ResponseEntity<String>(HttpStatus.***NOT\_FOUND***);  }   **private** RentalDTO createRentalDTO(Rental rental) {  RentalDTO rentalDTO = **new** RentalDTO();  rentalDTO.setId(rental.getId());  rentalDTO.setRentalDate(rental.getRentalDate());  rentalDTO.setRentalDays(rental.getRentalDays());  UserDTO userDTO = getUserForRental(rental);  MovieDTO movieDTO = getMovieForRental(rental);  rentalDTO.setUser(userDTO);  rentalDTO.setMovie(movieDTO);  **return** rentalDTO;  }   **private** MovieDTO getMovieForRental(Rental rental) {  String url = **"http://"** + **movieService** + **"/movies/"** + rental.getMovieId();  **log**.debug(**"using url: "** + url);  MovieDTO dto = **restTemplate**.getForObject(url, MovieDTO.**class**);  **return** dto;  }   **private** UserDTO getUserForRental(Rental rental) {  String url = **"http://"** + **userService** + **"/users/"** + rental.getUserId();  **log**.debug(**"using url: "** + url);  **try** {  UserDTO dto = **restTemplate**.getForObject(url, UserDTO.**class**);  **return** dto;  } **catch** (Exception e) {  UserDTO dto = **new** UserDTO();  dto.setId(rental.getUserId());  **return** dto;  }  } } |