Practice 3: JPA Java Persistence API, Hibernate, Part II

1. Open the project lab3 in IntelliJ IDE: File – New Project from Existing Sources. Add H2 and MySql run configurations. Remember if H2 profile is running, http://localhost:8080/h2-console is available.

-Dspring.profiles.active=H2

Cascade Types [1][2]

Info Cascade types specify how entity state changes are propagated from Parent to Child entities.

JPA cascade types:

ALL	propagates all operations from a parent to a child entity.
PERSIST	propagates persist operation **[3]**. Persist operation is used only for new entities (TRANSIENT entities) for which there is no associated record in the database. - - *SQL insert statements*
MERGE	propagates merge operation **[3]**. Merge operation is used only for DETACHED entities, to reattach entities in the context and perform update to the associated database records. - - *SQL update statements*
REMOVE	propagates remove/delete operations - - *SQL delete statements*
REFRESH	if parent entity is re-read from the database, child entity is also re-read form the database
DETACH	if parent entity is removed from the context, child entity is also removed from the context.

Hibernate cascade types

LOCK	if parent entity is attached to the context, child entity is also attached to the context.
REPLICATE	used with multiple data sources, when parent entity is replicated, child record is also replicated
SAVE_UPDATE	same as PERSIST + MERGE

2. Modify class Participant adding **cascade = CascadeType.All** to the annotation @OneToMany.

```
public class Participant {
    //....
    @OneToMany(mappedBy = "seller", cascade = CascadeType.ALL)
    private List<Product> products;
    //...
}
```

3. Add in src/test/java/com/awbd/lab3 a new test class:

```
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace =
AutoConfigureTestDatabase.Replace.NONE)
@ActiveProfiles("mysql")
@Rollback(false)
public class CascadeTypesTest {

    @Autowired
    private EntityManager entityManager;

    @Test
    public void saveParticipant() {

        Participant participant = new Participant();
        participant.setFirstName("Will");
        participant.setLastName("Snow");

        Product product = new Product();
        product.setName("Impression, Sunrise");
        product.setReservePrice(300D);
        product.setCode("PMON");
        product.setSeller(participant);

        participant.setProducts(Arrays.asList(product));

        entityManager.persist(participant);
        entityManager.flush();
        entityManager.clear();

    }
}
```

After running the test add **@Ignore** annotation to saveParticipant method.

```
@Ignore
@Test
public void saveParticipant() {

}
```

EntityManager [4]

Info

void	**persist(Object entity)**	Make an instance managed and persistent.
void	**flush()**	Synchronize the persistence context to the underlying database.
void	**clear()**	Clear the persistence context, causing all managed entities to become detached.
<T> T	**merge(T entity)**	Merge the state of the given entity into the current persistence context.

4. Modify the script **data_mysql.sql** and rerun the application with MySql profile configuration.

```
delete from product_category;
delete from info;
delete from product;
delete from category;
delete from participant;

insert into category(id, name) values(1, 'paintings');
insert into category(id, name) values(2, 'sculptures');


insert into product (id, name, code, reserve_price) values (1, 'The Card
Players', 'PCEZ', 250);
insert into info(id, product_id, description) values (1, 1, 'Painting by
Cezanne');
insert into product_category (product_id, category_id) values(1,1);
insert into participant(id, first_name, last_name) values (1, 'Will',
'Snow');

insert into product (id, name, code, reserve_price, seller_id)
values (2, 'Impression, Sunrise', 'PMON', 300, 1);
insert into info(id, product_id, description) values (2, 2, 'Painting by
monet');
insert into product_category (product_id, category_id) values (2, 1);

insert into product (id, name, code, reserve_price, seller_id)
values (3, 'Ballon Dog' , 'SJEF', 200, 1);
insert into info (id, product_id, description)
values (3, 3 , 'Sculpture by Jeff Koons');
insert into product_category (product_id, category_id) values (3, 2);
```

5. Add a test that modifies a participant name and the currency or the products he offers.

```
@RunWith(SpringRunner.class)
@DataJpaTest
//...
public class CascadeTypesTest {

    @Autowired
    private EntityManager entityManager;

    @Ignore
    @Test
    public void saveParticipant() {
        //...
    }

    @Test
    public void updateParticipant(){
        Product product = entityManager.find(Product.class, 2L);
        Participant participant = product.getSeller();
        participant.setFirstName("Wiilliam");
        participant.getProducts().forEach(prod ->
                        {prod.setCurrency(Currency.USD);});
        entityManager.merge(participant);
        entityManager.flush();
    }
}
```

6. Modify **@ManyToMany** relationship product-category in class Product:

```
@ManyToMany(mappedBy = "products",
        cascade = {CascadeType.PERSIST, CascadeType.MERGE})
private List<Category> categories;
```

and in class Category:

```
@ManyToMany(
cascade = {CascadeType.PERSIST, CascadeType.MERGE})
@JoinTable(name = "product_category",
        joinColumns = @JoinColumn(name = "category_id",
referencedColumnName = "id"),
        inverseJoinColumns = @JoinColumn(name = "product_id",
                referencedColumnName = "id"))
private List<Product> products;
```

Test behavior with **CascadeType.*REMOVE***.
More examples may be found in [**2**]. CascadeType.All is not recomanded for @ManyToMany.
Also, in practice @ManyToMany relationships are replaced by two @OneToMany relationships.

7. Modify **@OneToOne** relationship product-info in class Product:

```
@OneToOne(mappedBy = "product",
        cascade = CascadeType.ALL, orphanRemoval = true)
private Info info;
```

orphanRemoval [5]

Info If OrphanRemoval attribute is set to true, a remove entity state transition is triggered for the child entity, when it is no longer referenced by its parent entity.

8. Add a test that removes info for a product.

```
@RunWith(SpringRunner.class)
@DataJpaTest
//...
public class CascadeTypesTest {

    @Autowired
    private EntityManager entityManager;

    @Ignore
    @Test
    public void saveParticipant() {
        //...
    }

    @Test
    public void updateParticipant(){
        //...

    @Test
    public void orphanRemoval(){
        Product product = entityManager.find(Product.class, 1L);
        product.setInfo(null);
        entityManager.persist(product);
        entityManager.flush();
}
```

9. Add in pom.xml the Maven dependency for junit test with paramters.

```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-params</artifactId>
   <scope>test</scope>
</dependency>
```

10. Add the *id* of the product as parameter to *orphanRemoval* test. *orphanRemoval* method will be executed twice, with arguments 2 and 3.

```
@ParameterizedTest
@ValueSource(longs = {2, 3})
public void orphanRemoval(long id){
    Product product = entityManager.find(Product.class, id);
    product.setInfo(null);
    entityManager.persist(product);
    entityManager.flush();
}
```

Info **Parameterized tests [6]**

Parametrized tests are available in Junit 5 as a new feature that allows executing a test method multiple times with different parameters.

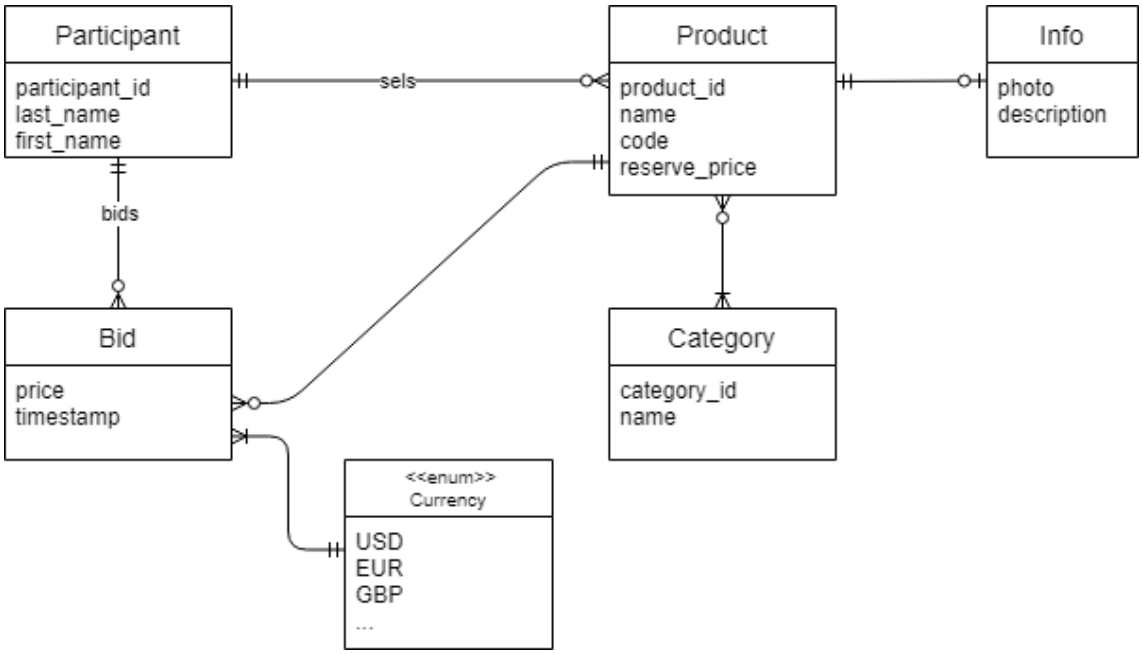
We can pass argument sources with
@ValueSource - - simple values. We can pass only one argument to the test method each time.

@EnumSource - - We can pass all elements of an enumeration or filter values using *names* attribute and regular expressions.

@CsvSource or @CsvFileSource - - pairs of (actual, expected) values.

11. Add in src/main/java/awbd/lab3 a new package, **repositories.**

12. Add in src/main/java/awbd/lab3/ repositories CrudRepository/PagingAndSortingRepository implementation for each entity: **ParticipantRepository, ProductRepository, CategoryRepository, BidRepository.**




```
package com.awbd.lab3.repositories;

import com.awbd.lab3.domain.Bid;
import org.springframework.data.jpa.repository.JpaRepository;

public interface BidRepository extends JpaRepository<Bid, Long> {
}
```

```
package com.awbd.lab3.repositories;

import com.awbd.lab3.domain.Category;
import org.springframework.data.repository.CrudRepository;

public interface CatagoryRepository extends CrudRepository<Category,
Long> {
}
```

```
package com.awbd.lab3.repositories;

import com.awbd.lab3.domain.Participant;
import org.springframework.data.repository.CrudRepository;

public interface ParticipantRepository extends
CrudRepository<Participant, Long> {
}
```

```
import com.awbd.lab3.domain.Product;
import org.springframework.data.repository.PagingAndSortingRepository;

public interface ProductRepository extends
PagingAndSortingRepository<Product, Long> {
}
```

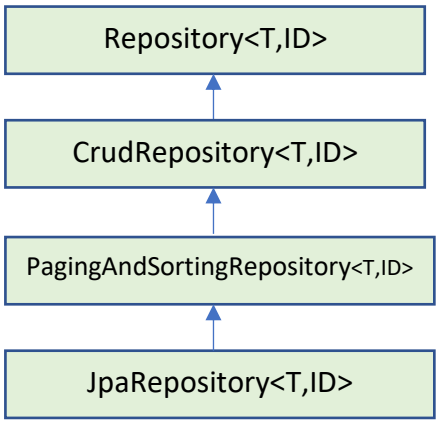
Spring Repositories [7][9]

All repositories extend generic interface **org.springframework.data.repository.Repository**.

Info **Repository** interface has two types arguments: domain class and ID type of the domain class.

CrudRepository provides CRUD functionality:

<S extends T>	S **save**(S entity);
Optional<T>	**findById**(ID primaryKey);
Iterable<T>	**findAll**();
Long	**count**();
Void	**delete**(T entity);
etc.	



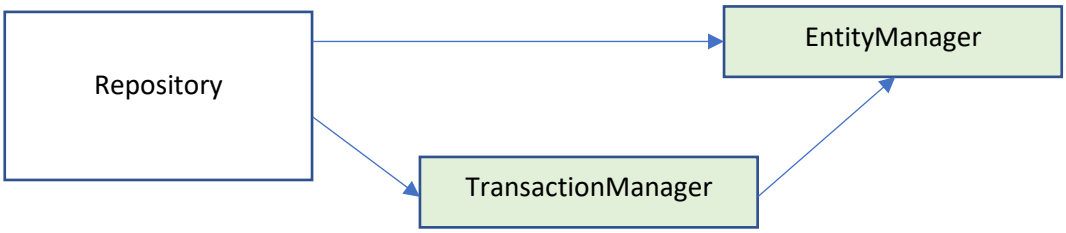
PagingAndSortingRepository provides methods to do pagination and sort records.

Iterable<T>	**findAll**(Sort sort);
Page<T>	**findAll**(Pageable pageable);

JpaRepository provides methods to manage persitance context (flush/delete records etc.)

void	**deleteInBatch**(Iterable<T> entities)
<S extends T>	**saveAndFlush**(S entity)
etc.	

A repository is wired to an EntityManager and to a TransactionManager **[8]**



13. Add in src/test/java/com/awbd/lab3 a new package, **repositories.**

Add a new test class, ParticipantRepositoryTest in package com.awbd.lab3.repositories, run test and check MySql database.

14.

```
package com.awbd.lab3.repositories;

import com.awbd.lab3.domain.Participant;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.annotation.Rollback;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace =
AutoConfigureTestDatabase.Replace.NONE)
@ActiveProfiles("mysql")
@Rollback(false)
public class ParticipantRepositoryTest {

    @Autowired
    private ParticipantRepository participantRepository;

    @Test
    public void addParticipant() {
        Participant participant = new Participant();
        participant.setFirstName("Jhon");
        participant.setLastName("Adam");
        participantRepository.save(participant);
    }
}
```

```
select * from participant
```

Info

Finder Methods [10][11][12]

Interfaces extending CrudRepository may include finder methods with the following naming convention:

findByAttributeKeywordAttribute

Keyword is one of the following:
And, Or, Like, IsNot, OrderBy, GreaterThan, IsNull, StartingWith etc.

Examples:

findByName(String name)	- - WHERE name = name.
findByNameAndDescription(String name, String desc)	- - WHERE name = name or description = desc
findByNameLike(String name)	- - WHERE name LIKE 'name%'.
findByValueGraterThan(Double val)	- - WHERE values > val
findByNameOrderByNameDesc(String name)	- - ORDER BY name DESC

SpringData JPA will automatically generate implementations for these methods.

15. Add finder method in ParticipantRepository class:

```
package com.awbd.lab3.repositories;

import com.awbd.lab3.domain.Participant;
import org.springframework.data.repository.CrudRepository;

import java.util.List;

public interface ParticipantRepository extends
CrudRepository<Participant, Long> {
    List<Participant> findByLastNameLike(String lastName);
    List<Participant> findByIdIn(List<Long> ids);
}
```

16. Add in pom.xml maven dependency to use Assertions in Junit tests:

```
<dependency>
   <groupId>org.assertj</groupId>
   <artifactId>assertj-core</artifactId>
   <version>3.11.1</version>
</dependency>
```

17. Add a new test class in package src/test/java/com/awbd/lab3/repositories.

```
package com.awbd.lab3.repositories;

import com.awbd.lab3.domain.Participant;
import lombok.extern.slf4j.Slf4j;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDataba
se;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;
import java.util.List;
import static org.junit.Assert.assertFalse;

@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace =
AutoConfigureTestDatabase.Replace.NONE)
@ActiveProfiles("mysql")
@Slf4j
public class FinderParticipantTest {

    @Autowired
    private ParticipantRepository participantRepository;

    @Test
    public void findByName() {
        List<Participant> participants =
participantRepository.findByLastNameLike("%no%");
        assertFalse(participants.isEmpty());
        log.info("findByLastNameLike ...");
        participants.forEach(participant ->
log.info(participant.getLastName()));
    }
}
```

18. Add test findByIds:

```
@Test
public void findByIds() {
    List<Participant> participants =
participantRepository.findByIdIn(Arrays.asList(1L,2L));
    assertFalse(participants.isEmpty());
    log.info("findByIds ...");
    participants.forEach(participant ->
log.info(participant.getLastName()));
}
```

19. Add findBySeller method in ProductRepository class:

```
package com.awbd.lab3.repositories;

import com.awbd.lab3.domain.Product;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.PagingAndSortingRepository;

import java.util.List;

public interface ProductRepository extends
PagingAndSortingRepository<Product, Long> {
    @Query("select p from Product p where p.seller.id = ?1")
    List<Product> findBySeller(Long sellerId);
}
```

Info **@Query annotation [13][14]**

If we want to define a custom SQL query to be executed by the CRUD repository, we may use @Query annotation.

Queries are written in **JPQL** or in **native sql**, adding attribute native: *Query* (, ***native = true***).
JPQL is an object-oriented query language. It uses the entity objects to define operations on the database records. JPQL queries are transformed to SQL.

There are two ways of transferring parameters to queries:

Indexed Query Parameters
Spring Data will pass method parameters to the query in the same order they appear in the method declaration:

```
@Query("select p from Product p where p.seller.firstName = ?1 and
p.seller.lastName = ?2")
List<Product> findBySellerName(String sellerFirstName, String
sellerLastName);
```

Named Parameters

We use the @Param annotation in the method declaration to match parameters defined by name in JPQL with parameters from the method declaration:

```
@Query("select p from Product p where p.seller.firstName = :firstName and
p.seller.lastName = :lastName")
List<Product> findBySellerName(@Param("firstName") String sellerFirstName,
@Param("lastName") String sellerLastName);
```

20. Add a new test class, ProductRepositoryTest in package com.awbd.lab3.repositories:

```
package com.awbd.lab3.repositories;

import com.awbd.lab3.domain.Product;
import lombok.extern.slf4j.Slf4j;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDataba
se;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.annotation.Rollback;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

import static org.junit.Assert.assertTrue;

@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace =
AutoConfigureTestDatabase.Replace.NONE)
@ActiveProfiles("mysql")
@Slf4j
public class ProductRepositoryTest {

    @Autowired
    private ProductRepository productRepository;

    @Test
    public void findProducts() {
        List<Product> products = productRepository.findBySeller(1L);
        assertTrue(products.size() >= 2);
        log.info("findBySeller ...");
        products.forEach(product -> log.info(product.getName()));
    }

    @Test
    public void findPage(){
       Pageable firstPage = PageRequest.of(0, 2);
       Page<Product> allProducts = productRepository.findAll(firstPage);
       Assert.assertTrue(allProducts.getNumberOfElements() == 2);
    }
}
```

21. Add method findBySellerName in ProductRepository and create a test method in class ProductRepositoryTest to use Named Parameters.

22. In the following steps we will create a MySql Docker image, run sprin-boot application docker image in a separate container and link it to MySql Docker image.

In PowerSwell download Docker image **[15].** You should find mysql image with docker images command.

Create a docker network boot-mysql. **[16]**

```
>>  docker pull mysql

>>  docker images

>>  docker network create boot-mysql

>>  docker network ls
```

23. Instantiate the image providing a name for the container, user and password, a root password, and a default database name.

```
>>  docker run --name mysql_awbd --network boot-mysql -e
MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=awbd -e  MYSQL_PASSWORD=awbd -e
MYSQL_USER=awbd mysql
```

24. Change **spring.datasource.url** in **application-mysql.properties** by replacing localhost with the container name:

```
spring.datasource.url=jdbc:mysql://mysql_awbd:3306/awbd

spring.datasource.username=awbd

spring.datasource.password=awbd

spring.datasource.platform=mysql

spring.jpa.hibernate.ddl-auto=create

spring.datasource.initialization-mode = always
```

25. Build the application .jar. We may use a maven configuration and skip tests. Add maven configuration and run:

```
clean install -Dmaven.test.skip=true
```

26. Create a docker file for the project.

```
FROM openjdk:11-oracle
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

27. Run in PowerShell (in the directory where a docker file is present) docker build and create a docker image *laborator3*. Check all the available images using docker images.

```
>> docker build -t laborator3 .

>> docker images
```

28. Run docker image laborator3 in network **boot_mysql**. Check `spring.jpa.hibernate.ddl-auto=create`

```
>> docker run --name lab3 --network boot-mysql -p 8080:8080 laborator3
```

29. Connect to mysql container and select all rows from participant table.

```
>> docker ps -a
>> docker exec -it [container_id] bash
>> mysql -u root -p
>> mysql> use awbd
>> mysql> select * from participant
```

B

[1] https://www.baeldung.com/jpa-cascade-types

[2] https://vladmihalcea.com/a-beginners-guide-to-jpa-and-hibernate-cascade-types/

[3] https://vladmihalcea.com/jpa-persist-and-merge/

[4] https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html

[5] https://vladmihalcea.com/orphanremoval-jpa-hibernate/

[6] https://www.baeldung.com/parameterized-tests-junit-5

[7] https://docs.spring.io/spring-data/jpa/docs/2.3.0.RELEASE/reference/html/#repositories

[8] https://docs.spring.io/spring-data/jpa/docs/2.3.0.RELEASE/reference/html/#jpa.java-config

[9] https://www.baeldung.com/spring-data-repositories

[10] https://www.baeldung.com/spring-data-derived-queries

[11] https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods

[12] https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repository-query-keywords

[13] https://www.baeldung.com/spring-data-jpa-query

[14] https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.at-query

[15] https://hub.docker.com/_/mysql

[16] https://docs.docker.com/network/bridge/