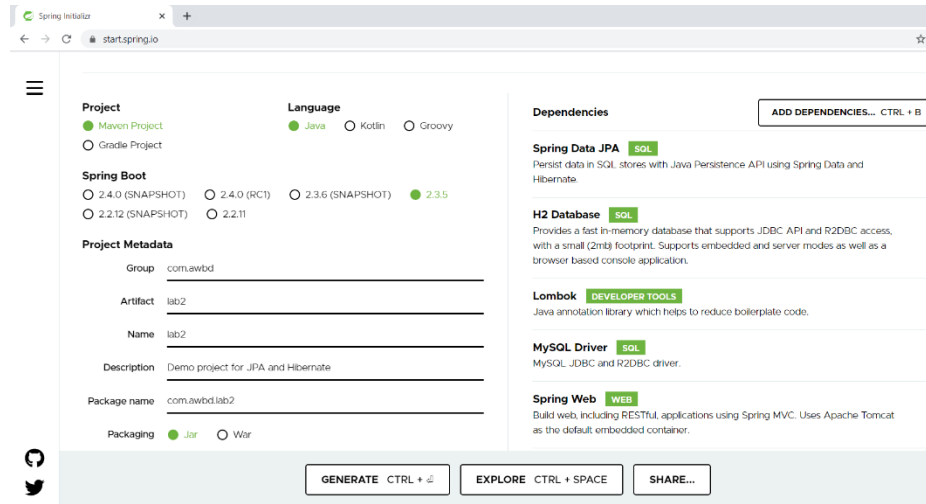


## Practice 3: JPA Java Persistence API, Hibernate, Part I

1.

Use **Spring initializr** to generate a maven project with dependencies: Spring Data JPA, H2 Database, Lombok, MySQL Driver, Spring Web or open LAB3\_START and skip to step 4.

<https://start.spring.io/>



2.

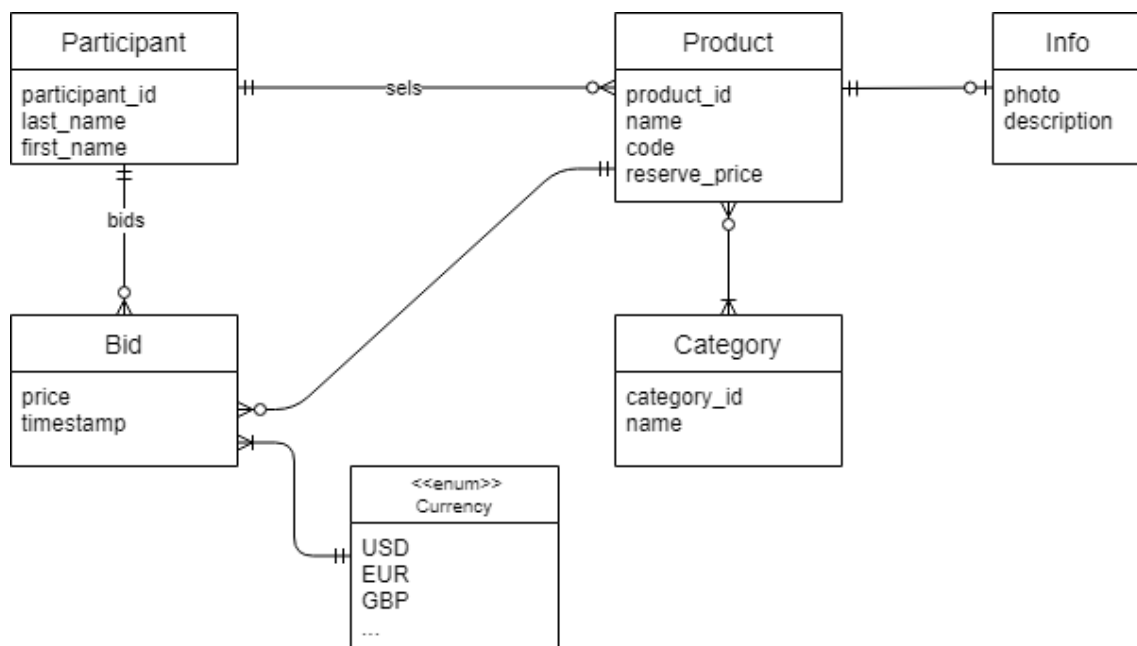
Open the project in IntelliJ IDE: File – New Project from Existing Sources. Check java.version in pom.xml file.

3.

Add in src/main/java/com/awbd/lab3 a new package, **domain**.

Add enumeration awbd.com.lab3.domain.**Currency** and classes:

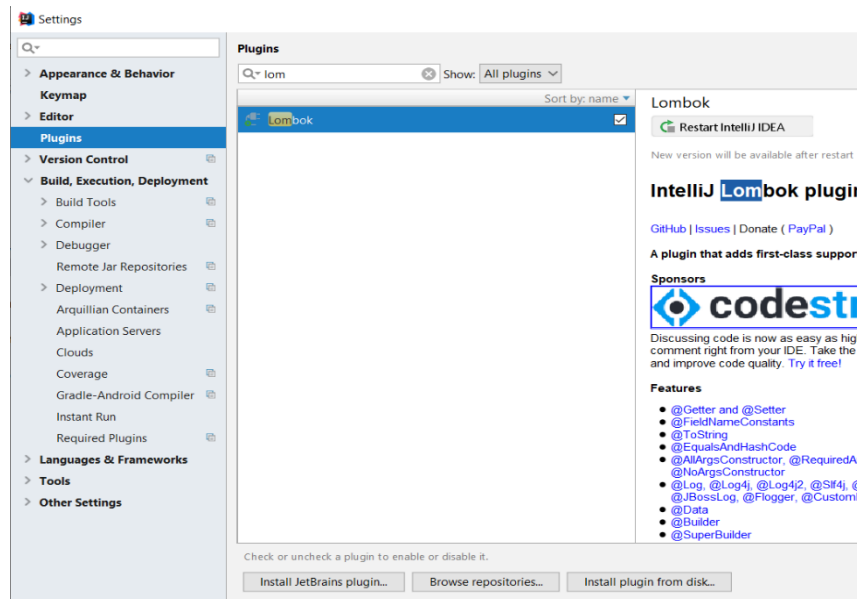
awbd.com.lab3.domain.**Participant**, awbd.com.lab3.domain.**Product**, awbd.com.lab3.domain.**Info**, awbd.com.lab3.domain.**Category**, awbd.com.lab3.domain.**Bid**, awbd.com.lab3.domain.**Info**



In the next steps we will use project Lombok to add **POJO** (plain old java object) basic methods (getters, setters, toString, equals, hashCode and constructors), then we will add **JPA** annotation and transform POJOs into entities, i.e. objects that maps tables from the database.

4.

Add in IntelliJ Lombok plugin, from File-Settings, Plugins and restart IntelliJ.



5.

Annotate all classes with Lombok.Data

```
package com.awbd.lab3.domain;

import lombok.Data;

@Data
public class Category {

    private Long Id;
    private String name;

}
```

6.

Try Refactor – Delombok to see the equivalent Java Code:

```
package com.awbd.lab3.domain;

public class Category {

    private Long Id;
    private String name;

    public Category() {}

    public Long getId() {...}
    public String getName() {...}
    public void setId(Long Id) {...}
    public void setName(String name) {...}
    public boolean equals(final Object o) {...}
    protected boolean canEqual(final Object other) {...}
    public int hashCode() {...}
    public String toString() {...}

}
```

## Lombok “Spicing up your Java”

### Info

Use annotations for code generation. [1]

It may be plugged in into editors (IntelliJ, Eclipse, NetBeans etc.) and automatically inject code that is immediately available.

Reduce boilerplate code (code that is repeated many times in the application).

### Lombok annotations

**@Getter** and **@Setter** generates getters and setters for a field.

If the class already contains a getter method for the field annotated with **@Getter**, the annotation is ignored.

**@ToString**(callSuper=true,exclude="someExcludedFields")

generates toString method, some fields may be excluded from the output, also the output return by the method toString of a superclass may be included.

**@EqualsAndHashCode**(callSuper=true,exclude={"someExcludedFields"})

Generates equals and hashCode methods. By default all non-static, non-transient fields are considered.

**@Data** is the same as using **@EqualsAndHashCode**, **@Getter**, **@Setter**, **@ToString**. It also adds a constructor taking as arguments all **@NonNull** and final fields.

**@NoArgsConstructor** generates a constructor with no arguments.

**@RequiredArgsConstructor** adds a constructor for each **@NonNull** or not initialized final field.

**@Builder** implements builder pattern. If class Participant is annotated **@Builder** we may use Participant.builder().lastName("Adam").firstName("John").build();

### @Log

Creates private static final java.util.logging.Logger log =  
java.util.logging.Logger.getLogger(LogExample.class.getName());

### @Slf4j

Creates private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExample.class);

Maven dependency:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
```

7.

Enable H2 database console and configure the datasource in the **application.properties** file:

```
spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

H2 in-memory RDBMS (relational database management system) [3], can be embedded in Java Applications. Support standard SQL and JDBC API,

Info

In-memory databases rely on main memory for data storage, in contrast to databases that store data on disk or SSDs, hence in-memory databases are faster than traditional obtaining minimal response time by eliminating the need to access disks.

Maven dependency:

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

If H2 console is enabled, setting the property `spring.h2.console.enabled = true`, we may access the url: <http://localhost:8080/h2-console>

If property `spring.datasource.url=jdbc:h2:mem:testdb` is set, a database named `testdb` will be embedded in the application, notice also the properties setting the driver and credentials.

8.

Run the application and test the H2 console.

<http://localhost:8080/h2-console>

If no classes are annotated with entity, no tables are created in the database, we only have *information* `INFORMATION_SCHEMA` labels and `USERS`.

9.

Annotate all classes with `@Entity` and key attributes with `@Id` and `@GeneratedValue`. Re-run the application and check that tables CATEGORY, PRODUCT, BID, PARTICIPANT and INFO are created in the H2 database.

```
package com.awbd.lab3.domain;

import lombok.Data;
import javax.persistence.Id;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;

@Entity
@Data
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

}
```

### @Entity

Info

**JPA Entities** are POJOs representing data that can be persisted to the database. An entity represents a table stored in a database. Every instance of an entity i.e. every instance of a class annotated with `@Entity` represents a row in the table.

### @Id

All entities must have a primary key. The field annotated with `@Id` represents the primary key.

For each primary key it is mandatory to define a generation strategy. There are four possible generation strategies and also one may define a custom strategy [4]:

<b>GenerationType.AUTO</b>	Spring chooses strategy.
<b>GenerationType.IDENTITY</b>	auto-incremented value.
<b>GenerationType.SEQUENCE</b>	uses a sequence if sequences are supported by the database (for example for Oracle database).
<b>GenerationType.TABLE</b>	uses a table to store generated values.

For the last two generation strategies one must also specify a generator (sequence or table):

```
@GeneratedValue(strategy = GenerationType.TABLE, generator = "table-generator")
@TableGenerator(name = "table-generator", ...
)
```

For more info about `@Column`, `@Table`, `@Transient` annotations see [5].

10.

Rename the file application.properties, **application-h2.properties**. Create in src/main/resources two more properties files

**application-mysql.properties.**

```
spring.datasource.url=jdbc:mysql://localhost:3306/awbd
spring.datasource.username=awbd
spring.datasource.password=awbd
spring.datasource.platform=mysql
```

**application.properties**

```
spring.profiles.active=mysql
spring.jpa.show-sql=true
```

11.

Re-run the application. All sql commands are showed in the application log (spring.jpa.show-sql=true). The active profile is now mysql. The file application-**mysql**.properties will be used to set the datasource properties. We will connect to a mysql database, awbd, running on localhost, with user awbd. Notice that any changes you make to the tables will be lost if we restart the application. The database is recreated any time the application restart. To change this behavior set the following property in **application-mysql.properties**:

```
...
spring.jpa.hibernate.ddl-auto=validate
```

#### Database initialization

Info

spring.jpa.hibernate.ddl-auto [6]

**none**

**create**

tables are dropped and recreated, a table is created for each class annotated @Entity

**create-drop**

schema is dropped and recreated, a table is created for each class annotated @Entity, used for tests. This is the default value for embedded databases.

**validate**

application start if all tables corresponding to entities exists, along with the column correspond to entities fields

**update**

Hibernates updates schema if tables differ from entities specifications.

Scripts **import.sql** (Hibernate option) or [ **schema.sql** and **data.sql** (Spring Boot option)] may be used to create (LDD) and initialize (LMD) the database.

12.

Add a category in awbd.category table, in mysql database, restart the application and check awbd.category rows.

```
insert into awbd.category values(1, 'paintings');

SELECT * FROM awbd.category;
```

13.

Add in src\main\resources a file **import.sql**. This DML script is executed when the active profile is H2, and may be used to add test data in the schema when **spring.jpa.hibernate.dli-auto = create** | **create-drop** is used. [6]

```
insert into category(name) values('paintings');

insert into category(name) values('sculptures');
```

14.

Re-run the application with VM option -Dspring.profiles.active=H2. Check in H2 console that table category contains two rows.

**-Dspring.profiles.active=H2**

15.

Add in src\main\resources a file **data-mysql.sql**. This DML script is executed when the active profile is mysql, if spring.datasource.initialization-mode is set always.

**application-mysql.properties:**

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

**data-mysql.sql:**

```
delete from category;  
insert into category(name) values('paintings');  
insert into category(name) values('sculptures');
```

In the next steps we will create the relations between entities.

**Relationships between entities [7]:**

## Info

**@OneToOne** In a RDBMS, a one-to-one relationship links two tables based on a **FK** column. The child table Foreign Key references the Primary Key of the parent table row. Each row in the child table is link to exactly one row in the parent table, in other words, each instance of the child @Entity is linked to exactly one instance of the parent @Entity.

OneToOne relationships can be either **unidirectional** or **bidirectional**.

For instance, unidirectional relationship *product – info* means that product entity will provide access to info entity, and we will be able to get info about the product (photo, description etc.) but info entity we will not provide access to the associated product.

In the associated tables in the RDBMS we will add info\_id column in product table but we will not add product\_id column in info table.

**@OneToMany** one entity is associated with one or more entities stored in a collection of type List or Set, Map, SortedSet, SortedMap etc. The foreign key is added in the table corresponding to “many”.

**@ManyToOne** is the opposite relationship of @OneToMany.

**@JoinColumn** defines the foreign key. In @Entity Bid we have:

```
@ManyToOne  
@JoinColumn(name="participant_id")  
private Participant participant;
```

The attribute **mappedBy** defines the corresponding field in the opposite relationship (@ManyToOne relationship) in @Entity Participant we have:

```
@OneToMany(mappedBy = "participant")  
private List<Bid> bids;
```

## Info

**@ManyToMany** in RDBMS is defined by an association table. For instance the relationship product-category is defined by the table `product_category` with columns: `product_id`, `category_id`.

**@JoinTable** defines the association table. In **@Entity Category** we have:

```
@JoinTable(name = "product_category",
joinColumns = @JoinColumn(name = "category_id", referencedColumnName = "id"),
inverseJoinColumns = @JoinColumn(name = "product_id",
referencedColumnName = "id"))
private List<Product> products;
```

The attribute **mappedBy** defines the corresponding field in the opposite (**@ManyToMany** relationship) in **@Entity Product** we have:

```
@ManyToMany(mappedBy = "products")
private List<Category> categories;
```

## 16.

Add **@OneToOne** relationship between product and info entities:

In Product entity add field:

```
@OneToOne
private Info info;
```

In Info entity add field:

```
@OneToOne
private Product product;
```

Change `spring.jpa.hibernate.ddl-auto` property in `application-mysql.properties` and re-run the application. Check that column **product\_id** and **info\_id** are added in tables **info** and **product**:

```
spring.jpa.hibernate.ddl-auto=update
```

## 17.

Add **@OneToMany** and **@ManyToOne** relationships between entities **Participant** and **Product**.

In Participant entity add:

```
@OneToMany(mappedBy = "seller")
private List<Product> products;
```

In Product entity add:

```
@ManyToOne
private Participant seller;
```

Re-run the application. Check that column **seller\_id** is added in table **product**, in MySQL database.



18.

Add @OneToMany and @ManyToOne relationships between entities: **Participant** – **Bid** and **Bid-Product**.

Add in Participant entity:

```
@OneToMany(mappedBy = "bidder")
private List<Bid> bids;
```

Add in Bid entity:

```
@ManyToOne
private Participant bidder;
```

Add in Product entity:

```
@OneToMany(mappedBy = "product")
private List<Bid> bids;
```

Add in Bid entity:

```
@ManyToOne
private Product product;
```

Re-run the application and check that two foreign keys are added in tables **bid**: **participant\_id** and **product\_id**, in MySQL database.

19.

Add @ManyToMany relationship Product-Category:

In entity Product add:

```
@ManyToMany(mappedBy = "products")
private List<Category> categories;
```

In entity Category add:

```
@ManyToMany
@JoinTable(name = "product_category",
joinColumns = @JoinColumn(name="category_id", referencedColumnName =
"id"),
inverseJoinColumns
= @JoinColumn(name="product_id", referencedColumnName="id"))
private List<Product> products;
```

Re-run the application and check the existence of table **product\_catgory**, in MySQL database.

20.

Populate table product with one line and re-run:

H2 profile import.sql:

```
insert into category(name) values ('paintings');
insert into category(name) values ('sculptures');

insert into product (name, code, reserve_price) values ('The Card
Players', 'PCEZ', 250);

insert into product_category values (1,1);
```

MySql profile data-mysql.sql

```
delete from product_category;
delete from product;
delete from category;

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 product_category values (1,1);
```

21.

Create a test class, **EntityManagerTest** in src\test\java\com\awbd\lab3

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

    @Autowired
    private EntityManager entityManager;

    @Test
    public void findProduct() {

        Product productFound = entityManager.find(Product.class, 1L);

        assertEquals(productFound.getCode(), "PCEZ");
    }
}
```

Info

Spring Testing [8]

**spring-boot-starter-test** imports useful modules and libraries for testing spring applications:

**spring-boot-test** module, with core items

**spring-boot-test-autoconfigure** module, providing support for auto-configuration

**Junit** [9]

**AssertJ** library to write assertions for tests[10]

**Mockito** mocking framework [11] etc.

## Info

**spring-boot-starter-test** dependency is added in .pom when we generate a project with Spring initializr

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

### @RunWith(SpringRunner.class)

Runs test with SpringRunner so that Spring annotations will be interpreted with SpringTestContext Framework [12]. For a better understanding of runner see [13].

### @DataJpaTest [14]

@DataJpaTest can be used to test JPA applications. By default, it will configure an in-memory embedded database (replacing any explicit or usually auto-configured DataSource), scan for @Entity classes and configure Spring Data JPA repositories. Regular @Component beans will not be loaded into the ApplicationContext.

By default, tests annotated with @DataJpaTest are transactional and rollback at the end of each test. @Rollback(false) overrides this setting.

The @AutoConfigureTestDatabase annotation can be used to override settings for default DataSource configuration.

## Info

### Persistence Context

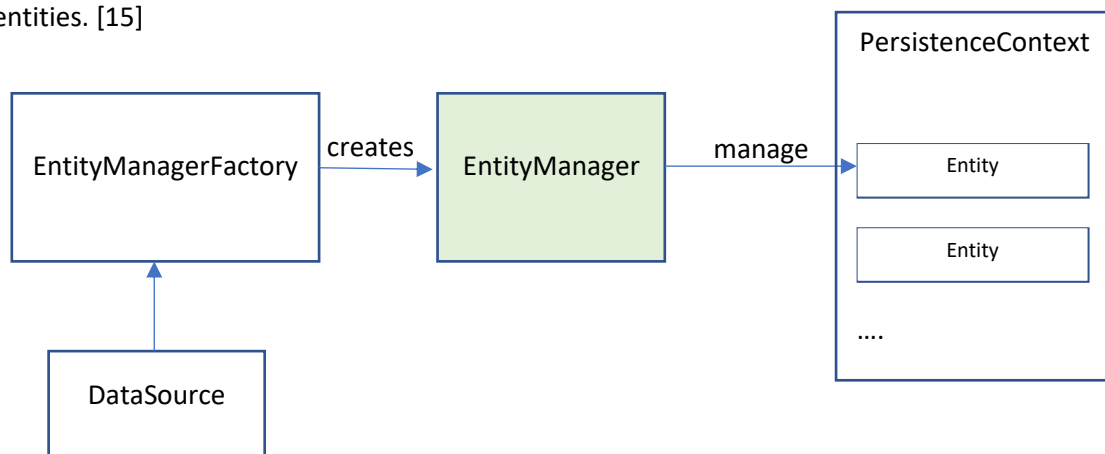
A persistence context is a set of entity instances. An EntityManager instance is associated with a persistence context. The entity instances and their lifecycle is managed by a particular entity manager.

### EntityManagerFactory

An entity manager factory provides EntityManager instances connected to the same datasource.

### EntityManager

An EntityManager instance is used to create and remove persistent entity instances and to query for entities. [15]



22.

Add filed currency in entity Product and Bid:

```
@Enumerated(value = EnumType.ORDINAL)
private Currency currency;
```

23.

Add tests in class EntityManagerTest

```
@Test
public void updateProduct() {

    Product productFound = entityManager.find(Product.class, 1L);
    productFound.setCurrency(Currency.EUR);

    entityManager.persist(productFound);
    entityManager.flush();

}
```

```
@Test
public void findCurrency() {

    Product productFound = entityManager.find(Product.class, 1L);
    assertEquals(productFound.getCurrency(), Currency.EUR);

}
```

The column currency will be set 1, the order number of currency *EUR* in the enumeration, counting from 0.

24.

Drop column currency and recreate the schema so that @Enumerated values will store string instead of integers.

```
alter table bid drop column currency;
alter table product drop column currency;
```

Change filed currency in entity Product and add field currency in entity Bid:

```
@Enumerated(value = EnumType.STRING)
private Currency currency;
```

## B

- [1] <https://projectlombok.org/>
- [2] <https://www.artima.com/lejava/articles/equality.html>
- [3] <https://www.h2database.com/html/main.html>
- [4] <https://www.baeldung.com/hibernate-identifiers>
- [5] <https://www.baeldung.com/jpa-entities>
- [6] <https://docs.spring.io/spring-boot/docs/1.1.0.M1/reference/html/howto-database-initialization.html>
- [7] <https://www.baeldung.com/spring-data-rest-relationships>
- [8] <https://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/reference/html/boot-features-testing.html>
- [9] <https://junit.org/junit5/docs/current/user-guide/>
- [10] <https://joel-costigliola.github.io/assertj/>
- [11] <https://site.mockito.org/>
- [12] <https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#testcontext-framework>
- [13] <https://www.logicbig.com/tutorials/unit-testing/junit/runner.html>
- [14] <https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/orm/jpa/DataJpaTest.html>
- [15] <https://www.baeldung.com/hibernate-entitymanager>
- [16] <https://www.baeldung.com/jpa-persisting-enums-in-jpa>