Lab 4 - Domain modelling and Java Persistence API

SENG301 Software Engineering II

Neville Churcher, Morgan English, Fabian Gilson, Karl Moore, Jack Patterson, Saskia van der Peet

16th March 2023

Learning objectives

This lab will help students to transpose conceptual domain entities (e.g., classes in a UML Class Diagram) and deal with persisting these domain entities into a database using the *Java Persistence API* (JPA). JPA abstracts some database-specific aspects for developers to create, retrieve or modify objects stored in a database without dealing with implementation-specific query language (e.g., SQL). By the end of this lab, students will:

- extend a domain model using the UML 2.5.1 Class Diagram[1];
- create a Java project using `gradle` dependency management (see lab 1 for a full recap);
- transpose these conceptual objects and their relationships into JPA `entity` classes; and
- retrieve, insert and delete these objects from an in-memory database.

This lab is a mix of a step-by-step tutorial with lab exercises to get hands-on experience with domain modelling in object-oriented programming together with the Java Persistence API.

1 Introduction to domain

You want to create a simple **Yu-Gi-Oh! Clone App** similar to the well-known card game *Yu-Gi-Oh!* The app needs to allow users to create an account, build a deck of cards, and save it. In future labs we will look at the mechanics of battling between decks. Figure 1 depicts the conceptual domain model of this app using UML Class Diagram notation. `Players` can have any number of `Decks` made up of at least one `Card`.

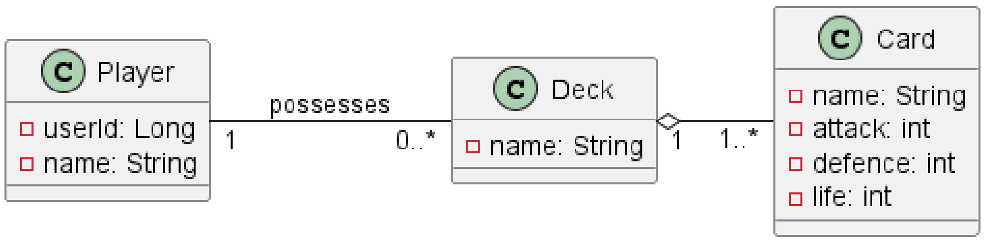

Figure 1: Simplified domain model of a Card Battler Clone App

We will build off this domain model in this lab as well as the following ones too.

[1] `https://www.omg.org/spec/UML/About-UML/`

2 Setting up your project (in IntelliJ IDEA)

The first exercise in this lab is to create a new Gradle project `uc-seng301-cardbattler-lab4`. For information on how to do this please refer back to `lab 1`.

Listing 1 shows the generated `build.gradle` file where dependencies and building instructions are defined. However, you will notice that **lines 18-22** have been added into below listing and you are asked to do the same in your project.

```
1  plugins {
2      // Apply the application plugin to add support for building a CLI application in Java.
3      id 'application'
4  }
5
6  repositories {
7      // Use Maven Central for resolving dependencies.
8      mavenCentral()
9  }
10
11 dependencies {
12     // Use JUnit Jupiter for testing.
13     testImplementation 'org.junit.jupiter:junit-jupiter:5.8.1'
14
15     // This dependency is used by the application.
16     implementation 'com.google.guava:guava:30.1.1-jre'
17
18     // use hibernate to persist (java) domain entities for us, aka JPA implementation
19     implementation 'org.hibernate:hibernate-core:6.1.7.Final'
20
21     // use a in-memory database to store entities (can be substituted with any database)
22     implementation 'com.h2database:h2:2.1.214'
23 }
24
25 application {
26     // Define the main class for the application.
27     mainClass = 'uc.seng301.cardbattler.lab4.App'
28 }
29
30 tasks.named('test') {
31     // Use JUnit Platform for unit tests.
32     useJUnitPlatform()
33 }
```

Listing 1: Additional configuration settings in `build.gradle` file.

In **line 19**, we declare the dependency to *Hibernate* which is the JPA implementation we use in SENG301/302. This will enable you to create special Java objects that will be persisted in a database relatively transparently.

In **line 22**, we add the dependency to *h2*, an *in-memory* database that will be used to persist the data. In-memory databases are application-context-dependent, so when the Java application stops, the database is cleared automatically. This is useful for *zero-configuration* local development or automated testing as these in-memory database behave similarly to SQL (relational) databases, especially when used with a JPA framework.

2.1 Additional details on `gradle` project structure

At the root of your `uc-seng301-cardbattler-lab4` folder, you have a few files created for you:

`.gitignore` predefined set of file names that will not be pushed to the version control system.

`gradlew(.bat)` a wrapper script that executes the `build.gradle` file and can bootstrap (i.e. install) the latest `gradle` tool for you if it's not installed on your machine. Keeping this file with your sources make it easier to develop and deploy anywhere at the moment the .

`settings.gradle` additional settings for `gradle` (you can ignore this file for now).

You may take a look into the project structure under the `app/src` folder. In a nutshell, the main folders are:

`main` contains the sources of your project
- **`java`** java source files
- **`resources`** application configuration files (e.g., *Hibernate*, *h2*)

`test` contains all test sources
- **`java`** java test sources
- **`resources`** test configuration files in later labs (e.g., *Hibernate*, *h2*)

`build.gradle` the dependencies configuration file

2.2 Run the default application

To make sure you have set up everything properly, you can run `$ ./gradlew run` from the `uc-seng301-card battler-lab4` folder to execute the programme. Your command line should output a similar trace as shown in Listing 2.

```
$ ./gradlew run

> Task :run
Hello world.

BUILD SUCCESSFUL in 987ms
3 actionable tasks: 3 executed
```

Listing 2: First run of generated skeleton application.

As you can see, `gradle` knows about how to start your application because of the `application` plugin (see line 3 in Listing 1) and the reference to the main class (i.e. the one containing a `main` method, see line 29 in Listing 1.

3 Create the `model` classes

3.1 Create a package that will host your domain entities

You need to implement our domain model into your Java application. To this end, you will create one Java class per class in the UML diagram depicted in Figure 1. A common convention is to put these domain objects into a `model` package that you will place under `src/main/java/uc/seng301/cardbattler/lab4/`.

3.2 Create the domain entities as plain old java objects

When your package is ready, you will create a `Player` java class with its attributes, as visible in Listing 3.

```java
package uc.seng301.cardbattler.lab4.model;

import java.util.List;

public class Player {
    private Long playerId;
    private String name;
    private List<Deck> decks;

  /* getters and setters omitted */
}
```

Listing 3: Plain old java object representing a `Player`.

You need to implement the other class of our domain in a similar way (i.e. for the `Deck` and `Card` classes). Note the `playerId` technical attribute. You will need to create a similar technical id for the other classes.

3.3 Create your first `@Entity` class

Now you have created the three classes under the `model` package, you can convert these plain java objects into JPA ones with dedicated *annotations*[2]. These annotations are used to declare a persistence object (i.e. database table), reproduce associations between classes at the database levels (i.e. foreign keys) as well as to specify technical constraints (e.g., ids or primary keys). The full updated `Player` class is visible in Listing 4. We have left out from this listing:

- the import section composed of `java.util` and `jakarta.persistence` elements (use the auto-import feature of your IDE wisely);
- all *getters/setters* for the class fields.

```java
 1 package uc.seng301.cardbattler.lab4.model;
 2 @Entity
 3 @Table(name = "player")
 4 public class Player {
 5
 6     @Id
 7     @GeneratedValue(strategy = GenerationType.AUTO)
 8     @Column(name = "id_player")
 9     private Long playerId;
10
11     @OneToMany(fetch = FetchType.EAGER)
12     @JoinColumn(name = "id_player")
13     private List<Deck> decks;
14
15     private String name;
16
17     public Player() {
18         // a (public) constructor is needed by JPA
19     }
20
21     @Override
22     public String toString() {
23         StringBuilder sb = new StringBuilder();
24         sb.append(String.format("Player (%d): %s%n\tDecks:%n", playerId, name));
25         for (Deck deck : decks) {
26             sb.append(String.format("\t\tDeck (%d): %s%n\t\t\tCards:%n", deck.getDeckId(), deck.getName()));
27             for (Card card : deck.getCards()) {
28                 sb.append(
29                         String.format("\t\t\t\tCard (%d): %s -- Attack: %d, Defence: %d, Life: %d%n", card.getCardId(),
30                                 card.getName(), card.getAttack(), card.getDefence(), card.getLife()));
31             }
32         }
33         return sb.toString();
34     }
35
36     /* Getters and setters omitted */
37 }
```

Listing 4: `Player` entity class (without imports).

line 2-3 The first thing to do is to tell the compiler that this class is a *Java Persistence API* (JPA) object by annotating the class name by `@Entity`. By doing so, your object will inherit from persistence-related behavioural properties (e.g., ability to be saved and retrieved from a database) and will be mapped to a database table almost automatically for you. We also specify a specific name for the table (i.e. `player` in lowercase letters).

lines 6-9 You need to tell which attribute will be the identifying column (i.e. primary key in terms of relational database). This is done by using the `@Id` annotation to the *field* being the unique identifier for that object. You also tell that this `@Id` is generated automatically for that table (`@GeneratedValue` with `AUTO` type)[3]. You

[2]An annotation is a metadata that is used by a (pre-) compiler or at runtime to give more information to the compiler (e.g., methods overides or warning suppressions) or to enable additional logic (e.g., framework features, generation of getters/setters).

[3]We usually recommend to use `IDENTITY` to have separate ids per table instead of unique ids accross all identifiable elements in table (i.e. global id). We use the `AUTO` strategy here for compatibility with *sqlite* that is used in the extension Section 5.

also rename this id column `id_player` in the database to comply to SQL naming convention (i.e. you override the attribute name by another name, but only inside the database table).

lines 11-13 You can link classes to one another with special JPA mechanisms that reflect relations in a relational database and correspond to the associations in the UML Class Diagram in Figure 1. This means that when a `Player` is retrieved from the database, its corresponding `Decks` will also be fetched at the same time (`EAGER` type). Because one `Player` can have many `Decks`, you declare the link as a `@OneToMany` (i.e. many `Decks` can belong to one `Player`) and use the `Players`'s `id_player` as the reference (i.e. foreign key), actually referring to the database column's name instead of the Java field name (since you will rename the `Player` attribute in the `Deck` class to `id_player` later in Section 3.4).

lines 17-19 The JPA framework (i.e. *Hibernate*) needs a *no-args* constructor that must be package-private at least.

lines 21-28 To easily see the content of a player we override the `toString()` method and return the relevant information for the player and it's attributes. Not the usage of the `format` method[4]. Also, note that if you implement a similar `toString` method for other classes, do not create a *circular dependency* between the `toString` methods, e.g., do not call the `getPlayer().toString()` method from the `Deck.toString()` method.

After copying all these annotations, *IntelliJ* may show errors on the column names (e.g., next to the `@JoinColmn` annotation) saying that there is no column with that name. This is expected as you did not define the link to a database yet (that will be done in Section 3.5).

3.4 Transforming the other two classes into `@Entity` classes

In a similar fashion as above, you are required to adapt the `Deck` and `Card` classes to make them JPA entities. This is left to you as an exercise. Note that, because this time, the `Player` has many `Decks`, you will define a `@ManyToOne` relationship in the `Deck` class. A similar relationship exists between `Deck` and `Card` where a `Deck` aggregates many `Cards`.

3.5 Register the `@Entity` classes

In order for the JPA engine to know what classes are to be mapped to database tables as well as to define the connection to a database (you use an *in-memory* database here), we need to create a configuration file to be placed under the `resources` folder with a predefined name for *Hibernate* to pick it up with little programming effort. To this end, right click on the `resources` folder, select *"New > File"* and name your file `hibernate.cfg.xml` (including the extension). You can copy the following configuration[5]:

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN" "http://www.hibernate.org/dtd/
    hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>
        <property name="connection.driver_class">org.h2.Driver</property>
        <property name="connection.url">jdbc:h2:mem:test</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"/>
        <!-- Set this to "true" to print all executed SQL to stdout -->
        <property name="show_sql">false</property>
        <!-- generate the schema at startup -->
        <property name="hibernate.hbm2ddl.auto">create</property>
        <!-- specify the classes that are mapped to database tables -->
        <mapping class="uc.seng301.cardbattler.lab4.model.Player"/>
        <mapping class="uc.seng301.cardbattler.lab4.model.Deck"/>
        <mapping class="uc.seng301.cardbattler.lab4.model.Card"/>
    </session-factory>
</hibernate-configuration>
```

Listing 5: *Hibernate* configuration file.

[4] See `https://www.javatpoint.com/java-string-format` for more details on the keywords used.
[5] Please note that due to the line numbers in this handout, you will need to remove them after copy-pasting.

lines 5-9 database dialect, driver (*h2* here, but it can be almost any kind of database, see Section 5), url to the database[6] and connection credentials.

line 11 for debugging purpose, *Hibernate* can be asked to print the generated SQL statements to the standard output (or a logging facility, if one is declared; we will address logging in lab 6).

line 13 *Hibernate* is asked to generate the database schema for you, i.e. everytime the application starts, a new schema is created[7]

lines 15-17 list of entities this database will map the tables to, i.e. all classes you declared with `@Entity` annotations.

4 Create, update, delete `@Entity`

Now that you have your domain entities ready, you can update your `main` method in the `App` class to interact with the database. For this lab, you will simply write example code inside the `main` method, but you will learn about proper design and architectural patterns later in the course.

Note that when starting your programme with `$ ./gradlew run`, *Hibernate* normally outputs some information about its startup, however we can disable most of these logs with the code in Listing 6 (making sure to place it at the very top of your `main` function).

```java
java.util.logging.Logger.getLogger("org.hibernate").setLevel(Level.SEVERE);
```

Listing 6: Java code to hide Hibernate logs from the console.

4.1 Initial setup and database session factory

Your first step is to tell your java programme that you are using a database that has been configured in the `xml` file reproduced in Listing 5. You can create a *no-args* constructor for your `App` class with the content shown in Listing 7 (note that all imports are from `org.hibernate`):

```java
    private static SessionFactory sessionFactory;

    public App() {
        // this will load the config file (hibernate.cfg.xml in resources folder)
        Configuration configuration = new Configuration();
        configuration.configure();
        sessionFactory = configuration.buildSessionFactory();
    }
```

Listing 7: `App` constructor with no arguments with *Hibernate* setup.

A `SessionFactory` is needed to access the database. It is your proxy to manipulate objects that are persisted through *Hibernate*, e.g., saving or querying objects.

4.2 Save `@Entity` objects

In order to save an `@Entity`, you need to open a *transaction*, *persist* the object and *commit* your transaction. You can think of a *transaction* as a communication window between your Java application and the database. This is useful to ensure that, when performing (multiple) action/s with the database, either they are all executed or none of these actions are executed, i.e. the database's integrity is preserved and no partial changes are applied to it.

Assuming you have instantiated a `Player` named `player`, a `Deck` named `deck`, and `Cards` named `goliath` and `minotaur` the following Listing 8 shows the transaction code, i.e how everything is persisted into the database in

[6]A connection url to a database from a Java application must follow a predefined format. `jdbc` is the type of connector used, `h2` is the type of database, `mem` is a special keyword from *h2* to use the memory instead of a file (e.g., *sqlite* or a database (e.g., *mariadb*) and `test` is the name of the database schema you use.

[7]You should avoid using this in a production environment as it would either crash or corrupt your database. See `https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/appendices/Configurations.html`

one transaction such that either all statements in the transaction are executed or none (see the error handling part and more particularly line 16 where you *"rollback"* the database to the state it was before starting the transaction in case of failure).

```java
1   Transaction transaction = null;
2   Long playerId = -1L;
3   try (Session session = sessionFactory.openSession()) {
4       System.out.println("Persisting a player with deck of cards");
5       transaction = session.beginTransaction();
6       session.persist(player);
7       playerId = player.getPlayerId();
8       session.persist(deck);
9       session.persist(goliath);
10      session.persist(minotaur);
11      // persist into the database
12      transaction.commit();
13  } catch (HibernateException e) {
14      System.err.println("Unable to open session or transaction");
15      e.printStackTrace();
16      if (transaction != null) {
17          transaction.rollback();
18      }
19  }
```

Listing 8: Creating a transaction to save `@Entity` objects.

4.3 Retrieve objects

In order to retrieve saved objects, you can query your database. *Hibernate* uses a special SQL *"dialect"*, named *HQL*[8] that allows you to refer to tables and columns using the names you used for your `@Entity` classes and attributes instead of the actual column names. Similarly to Listing 8, you need to open a `session` to query the database, as shown in Listing 9.

```java
try (Session session = sessionFactory.openSession()) {
    System.out.println("Retrieving player with id: " + playerId);
    // note that HQL queries use the Java attributes names, i.e. playerId
    // (attribute) and not id_player (table column)
    Player retrievedPlayer = session.createQuery("FROM Player WHERE playerId =" + playerId, Player.class)
            .uniqueResult();
    System.out.println(retrievedPlayer);
} catch (HibernateException e) {
    System.err.println("Unable to open session or transaction");
    e.printStackTrace();
}
```

Listing 9: Retrieve an `@Entity` object using HQL.

5 Going further

For the quickest amongst you, here are a few extension tasks you should look at.

5.1 Use a *sqlite* database file

You can easily switch to a (persisting) database by updating your `build.gradle` file with additional libraries. We show how to use *sqlite*[9] in Listing 10 (note that only the additional dependencies are shown):

[8] See `https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html`
[9] *sqlite* is a lightweight file-based database that requires close to zero-configuration. See `https://www.sqlite.org/index.html`

```
dependencies {
    // Sqlite as persistent DB
    implementation 'org.xerial:sqlite-jdbc:3.40.1.0'
    implementation 'org.hibernate.orm:hibernate-community-dialects:6.1.7.Final'
}
```

Listing 10: Excerpt from `build.gradle` file to use *sqlite*.

You need to tell *Hibernate* you want to use a *sqlite* database file instead of *h2*, as visible in the updated `xml` configuration file in Listing 11. See that only the top lines need to be updated, i.e. the link and configuration to the database.

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN" "http://www.hibernate.org/dtd/
       hibernate-configuration-3.0.dtd">
3  <hibernate-configuration>
4      <session-factory>
5          <!-- sqlite properties -->
6          <property name="hibernate.dialect">org.hibernate.community.dialect.SQLiteDialect</property>
7          <property name="connection.url">jdbc:sqlite:lab4.sqlite</property>
8          <property name="connection.driver_class">org.sqlite.JDBC</property>
9          <!-- Set this to "true" to print all executed SQL to stdout -->
10         <property name="show_sql">false</property>
11         <!-- generate the schema at startup -->
12         <property name="hibernate.hbm2ddl.auto">create</property>
13         <!-- specify the classes that are mapped to database tables -->
14         <mapping class="uc.seng301.cardbattler.lab4.model.Player"/>
15         <mapping class="uc.seng301.cardbattler.lab4.model.Deck"/>
16         <mapping class="uc.seng301.cardbattler.lab4.model.Card"/>
17     </session-factory>
18 </hibernate-configuration>
```

Listing 11: Updated `hibernate.cfg.xml` to use *sqlite*.

Per configuration above, a file named `lab4.sqlite` will be created at the root of your project the first time you `$ ./gradlew run` your project. After having it created for the first time, you can deactivate the auto-creation (line 12) to keep using the same data across subsequent runs of your programme.

To know more about how to use *sqlite* databases inside *IntelliJ*, refer to `https://www.jetbrains.com/help/idea/connecting-to-a-database.html#connect-to-sqlite-database`

5.2 Understand `@Id` generation and `cascade` clauses

To learn more about the automatic generation of ids in *hibernate* see:

- `https://docs.jboss.org/hibernate/orm/5.0/mappingGuide/en-US/html/ch06.html`
- `https://www.baeldung.com/hibernate-identifiers`

You may be interested to know how JPA can manage CRUD operations on related `@Entity` objects (to be used wisely): `https://www.baeldung.com/jpa-cascade-types`.