JPA – Java Persistence API

Table of Contents

[1 Introduction to JPA 3](#_Toc91445339)

[1.1 Why learn JPA? 3](#_Toc91445340)

[1.2 Book Recommendations 3](#_Toc91445341)

[1.3 Why JPA? 3](#_Toc91445342)

[1.4 Hibernate vs JPA 4](#_Toc91445343)

[1.5 How JPA works and its advantages 5](#_Toc91445344)

[1.6 Setup JPA 5](#_Toc91445345)

[1.6.1 Configure the database connection and JPA’s behavior 5](#_Toc91445346)

[1.6.2 hibernate.hbm2ddl.auto 7](#_Toc91445347)

[1.6.3 Hibernate.hbm2ddl.auto in Production? -> NOPE 7](#_Toc91445348)

[1.6.4 Schema vs Database (in MySQL) 7](#_Toc91445349)

[2 JPA Annotations 8](#_Toc91445350)

[2.1.1 @Column 8](#_Toc91445351)

[2.1.2 @Basic 8](#_Toc91445352)

[2.1.3 @Transient 8](#_Toc91445353)

[2.1.4 @Lob 8](#_Toc91445354)

[2.1.5 @Temporal 9](#_Toc91445355)

[2.1.6 @Enumerated 9](#_Toc91445356)

[2.1.7 @Id 10](#_Toc91445357)

[2.1.8 @GeneratedValue 10](#_Toc91445358)

[2.1.9 hibernate\_sequence Table 11](#_Toc91445359)

[2.1.10 Natural Keys vs Surrogate Keys 11](#_Toc91445360)

[2.1.11 JoinColumn 11](#_Toc91445361)

[3 CRUD Operations 12](#_Toc91445362)

[3.1 Create Data 12](#_Toc91445363)

[3.2 Update Data 12](#_Toc91445364)

[3.3 Reading Data 12](#_Toc91445365)

[3.4 Deleting Data 12](#_Toc91445366)

[4 Relational Concepts for Mapping 13](#_Toc91445367)

[4.1 Uni-directional and Bi-directional Relationships 13](#_Toc91445368)

[4.2 One to One 14](#_Toc91445369)

[4.3 One to Many – Many to One 15](#_Toc91445370)

[4.4 Many to Many 17](#_Toc91445371)

[4.4.1 Customizing the Join Table 18](#_Toc91445372)

[4.5 Updating a Many to Many Relationship 18](#_Toc91445373)

[4.5.1 Update works only if the owner is updated 18](#_Toc91445374)

[4.6 Deletion and Cascading 19](#_Toc91445375)

[4.7 Owner side and XXX side 19](#_Toc91445376)

[4.8 Why Have Cyclical Relationships? 20](#_Toc91445377)

[4.9 Fetch Types - Lazy and Eager Initialization 22](#_Toc91445378)

[4.10 Proxy Objects 23](#_Toc91445379)

1. Introduction to JPA
   1. Why learn JPA?

It is widely used in the Java Enterprise world. Most of the Java enterprise apps use relational databases and JPA makes it easy to work with such databases. An alternative is only using JDBC but it is a low level DB connection API and does not have many features like JPA does.

JPA can show in various different types of Java applications:

* **Spring Boot application with a DB**: the most common way to build large scale Java applications
* **Java / Jakarta EE application with DB**: also widely used in the enterprise applications, especially the old ones
* **Microservices (each service with a DB)**: each microservice might have a DB and could use JPA
* **Desktop and mobile apps**: not widely used but JPA can still be used for such cases because probably your desktop or mobile app would send requests to a server.
  1. Book Recommendations

In addition to this course, there are also a couple useful books:

* **Pro JPA 2 in Java EE 8**: Covers JPA in its essence, a reference book.
* **Java Persistence with Hibernate**
  1. Why JPA?

What’s the point? Well, the point is **ORM (Object Relational Mapping)**. To build a java app with a database, you need:

1. Diagram, schematic

   Description automatically generated**JDBC drivers** to connect to the database
2. **Data access services** to access the data in the database and read, write data using the JDBC API.
3. The data “container” will be the **data access objects (DAO’s)**, basically data instances.
4. The **business logic / business server** will call the **data services** to perform the required operations on the database

JPA is concerned with the data service part of your application. But why? This picture was working for many years just fine. So what’s the problem?

* Diagram

  Description automatically generated with low confidenceThe problem is the **paradigm mismatch**. Class vs database table.
* A table would be corresponding to a class. Each column in the table are fields in the class and each row is an instance of the class.
* This is called an **impedance mismatch**. Because in an RDB, the emphasis is always on the relationships between tables but classes does not have that.

Lets say we want to **read/fetch** a bunch of Employees.

1. Prepare a SQL query: SELECT \* FROM Employee WHERE …
2. Run the query: once you run it you will get a ResultSet. You can’t use the result set directly in your app. In your java app you want a collection of Employees.
3. You will loop through each row in the ResultSet and
4. create a new Employee instance per row.
5. Extract each column and assign it to the fields of your instance.
6. Once the mapping is done, put the instance into a collection
7. But you also have to handle the exceptions or edge cases and decide a fallback value.

This was just a typical **Read** operation, the **R** of **CRUD**. This is already a lot and the other operations would be even more complex when more tables, more relationships comes into play.

* Object Relational Mapping, mapping objects to rows has to be done with every operation, so JPA does this for you. With the Java persistence API you can define the relationships and all the mapping logic.
  1. Hibernate vs JPA

Oversimplifying it, one is an **interface/specification (JPA)** and one is an **implementation (Hibernate)**. Hibernate is an open-source ORM framework. So, it know what to do with the tables and classes.

* Diagram

  Description automatically generatedHibernate existed even before JPA was created.

The steps to add **Hibernate** to a java project**:**

1. you would get the library, add it to the classpath and then
2. you would map your classes to the database tables
3. map your member variables to column names
4. then use Hibernates API to do CRUD operations on the java object instances

So, Hibernate would sit between JDBC and your application. Hibernate would create all the SQL queries which does the job for you.

These mappings in step 2 and 3 can happen in 2 ways:

* XML configuration
* Java annotations (preffered)

Hibernate is one of the multiple ORM solutions, the most popular. So, when a framework gets too popular in the Java community, people will fear depending on one framework too much. What happens when one day Hibernate disappears and they are forced to use a different framework? They then need to adjust the existing code which is really painful.

* To avoid that, the community creates an interface/standard. They try to standardize the API’s which are used by the most common frameworks in the Java space. Java EE and Jakarta EE is also examples for that.
* So, JPA is a standardization of ORM API’s

**How does Hibernate/JPA relate to JDBC ?**

Hibernate (or another JPA provider) actually uses JDBC for all database communications. Hibernate is another layer of abstraction on top of JDBC. So when our application uses the Hibernate framework, it will store and retrieve objects using Hibernate API. In the background Hibernate does all of the low level JDBC work, submitting the SQL queries and so on. But in the background everything goes through the JDBC API.

* 1. Diagram

     Description automatically generatedHow JPA works and its advantages

The steps to add JPA to a java project:

1. You need to add a JPA implementation to your java project. There are also other JPA implementations, but yes, the most popular is still Hibernate.
2. Now in your code you would call the JPA API’s and not Hibernate API’s

What’s the problem with using just SQL vs using JPA?

* **Productivity**: Lesser code to write, lesser errors, less time to maintain
* **Database independence**: some things are depended on the database that you use. With JPA, it handles those differences for you if you want to switch the database.
* **Caching and performance**: JPA does caching in some scenarios which can improve the performance.
* But be aware of the performance drawbacks that JPA can have in certain situations
  1. Setup JPA

Typically, JPA is used with Spring or Java EE projects but the problem is that those hide a lot of details of JPA.

* So, we will setup a barebone Java project with only JPA to better understand it.

Graphical user interface, text, application, email

Description automatically generatedSteps:

1. **Create a java project**: a simple maven project
2. **Add JPA libraries**: we will use Hibernate as our JPA implementation and we also require the JDBC driver for the database we will use
3. **Configure the database connection and JPA’s behavior**
4. **Do entity relation mapping**
5. **Use JPA API to persist an entity instance**
   * 1. Configure the database connection and JPA’s behavior

JPA requires you to specify a **persistence context**. A persistence context contains information about:

* What is the database?
* Where is it located?
* Credentials?
* How should it connect and behave when connection to the database?

Chart

Description automatically generated

I created a persistence.xml in the classpath. It probably don’t have to be in META-INF but I guess it is the convention.

Graphical user interface, text, application, email

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

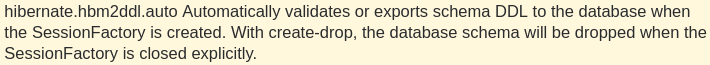
Table

Description automatically generated

Text

Description automatically generated

* Hibernate **WON’T** create databases for you, only the tables. You need to do that manually beforehand
* The order of the columns will be different if we use hbm2dll.auto. If we create it manually then we can decide which order we want. **create table user (1.column, 2.column, ...)**
* **dialect**: SQL is a standart, however each database has different implementations. A dialect is a SQL grammar that the JPA implementation uses when connecting to the database. There are slight differences in the way that SQL queries are written because there are certain operations that can only be done in certain databases. Therefore we need to explicitly tell which dialect we want to use.
* **show\_sql**: When hibernate executes, it will also print the SQL it sends to the database. Useful for development and testing, to see what hibernate is actually sending.
* **hibernate.hbm2ddl.auto:** definesthe default behaviour of hibernate
  + 1. hibernate.hbm2ddl.auto
* none: does nothing, same as not defining this property.
* create: create the schema. If the same schema already exists, it will be dropped. So, the tables with the same name would be deleted each time your app starts.
* update: if schema is not yet present, it will be created. If it is already present it will try to update the schema with the new changes. It will attempt to add new columns, constraints, etc. but will never remove a column or constraint that may have existed previously but no longer exists. It is not recommended to use this in prod.
* validate: validates the schema. Makes no changes in the DB.
* create-drop: same as create but it will also drop the schama (meaning dropping the tables) only when the sessionFactory (in pure Hibernate) or entityManagerFactory (JPA) **is explicitly closed**, meaning if entityManagerFactory.close() is called before your app exits. Otherwise, the schema won’t be removed.  
  This is useful for testing purposes, for unit and integration tests.

****

* + 1. Hibernate.hbm2ddl.auto in Production? -> NOPE

Text

Description automatically generated

* + 1. Schema vs Database (in MySQL)

Graphical user interface, text, application, email

Description automatically generated

* Schema and a database is different in almost every other database
* A schema is a part of the database.   
  Usually a schema is a collection of tables and a Database is a collection of schemas

1. JPA Annotations

An “Entity Class” is just a fancy name for a Java class that is mapped to a database table with the help of annotations.

1. **Mapping a class to a database table** is done by using the **@Entity** and **@Table** annotations. @Entity lets it know that this is an entity that we will map to a database table. @Table(name = “...”) we give the actual name of the database table where we want to map. The @Table annotation can also take a schema or a category as a parameter.
2. **Mapping fields to database columns** is done with **@Column** annotation.
   * 1. @Column

The field name and column names can be different of course, that is why we use the name = “...” . But if they are the same there is no need to use the **@Column** annotation.

* Annotations can either be written on top of the fields, in that situation it picks up the values of the fields or on top of the getter methods of the fields. Then they will picks the return values of the getter methods.
* **@Column(unique = true)**: Can be used when you want to make sure your values in a column are unique. You will get an exception when trying to insert a duplicate value. In the DDL, it does **add constraint … unique**.  
  You can also set many other options here, like the length of the column and so on. But notice that these options are for the **DDL**, meaning these are only useful when you let JPA create your tables. In production applications you would not use such options because we would define our database/schema in a different way, for example with **Liquibase** and there you would define such constraints.
* Most of the primitive Java types, and for example String will be mapped to the appropriate database types automatically by JPA. String -> varchar, int -> number and so on. But there are of course other object types which JPA cannot map automatically. For example a Date or a Calendar instance.
  + 1. @Basic

**@Basic**: (Default) Hibernate knows to treat this field as a default. Using this annotation makes only sense if we also use the properties of it. If we won’t annotate fields then Hibernate still persists every field and creates a column for each.

* + 1. @Transient

**@Transient**: Hibernate won’t automatically persist this field. Can be used for static fields or for fields we don’t want to save. There is also a **transient** keyword in Java which is used to avoid serialization, so the field won’t be serialized. This can also be used but if you only want to tell JPA (and not to Java) to avoid this field, then use the annotation.

* + 1. @Lob

**@Lob**: saves the data in BLOB or CLOB. If data is text and is not enough to save in VARCHAR, then that data should be saved in CLOB. In case of double byte character large data is saved in BLOB data type.

**TODO** – other annotations

* + 1. @Temporal

**@Temporal (value)**: Used for storing Java objects like Date/Calendar etc. It was mandatory before Java 8. Since Java 8, the mapping between the Java 8 Date/Calendar classes and the SQL types is implicit, there is no need to specify the @Temporal annotation. For example, **Date** and **LocalDateTime** will be mapped to **2021-12-10 12:22:03** in the database by default.

You will get an exception if you try to use it with a different type:

* ***@Temporal*** *should only be set on a* ***java.util.Date*** *or* ***java.util.Calendar*** *property*

Text

Description automatically generated with medium confidence

* But Date should not be used anymore. It is the old Java API, it is not thread safe, you can difficultly handle time zoning, and on the top of all, it is poorly designed: one simple uniformity is that months start from 1 while days start from 0
  + 1. @Enumerated

Java enums will get an implicit **ordinal** value, which is an integer and denotes the order in which they are defined.

* Even though if we define a value in our enum, the ordinal value will be saved in the database by default.  
  This can be BAD because if the order changes, you will have a problem.

Graphical user interface, text, application, email

Description automatically generatedGraphical user interface, application, Teams

Description automatically generatedA picture containing text

Description automatically generated



You can use the **@Enumerated** to save the enum values as strings or with its ordinal value.   
The default is saving the ordinal.

* If you save the ordinal, you can change the names of the enums but not the position
* If you save the string: you can’t change the names but can change the positions of the values
  + 1. @Id

**@Id** defines the primary key. Can be used with primitive types, byte, char, long and also with their wrapper types.

* When you mark a field/column as a primary key, the database automatically adds a constraint that the values should be unique.
* You can also have a **String** (**varchar**) for a primary key but that has performance implications
* Using **float or double** as a primary key is not recommended because of the precision problem. You might get different values for your id for the same object.
* Text

  Description automatically generatedUse **Long** as the primary key.



* + 1. @GeneratedValue

If we want to automatically generate the primary key value, we can add the @GeneratedValue annotation. We can tell the JPA provider how we want to generate our primary. If none is defined, AUTO is the default one. You can also define your own generation strategy.

**How to choose a strategy?**

Basically, you have 2 major choices:

* You can generate the identifier yourself, in which case you can use an assigned identifier
* You can use the @GeneratedValue annotation and the JPA provider will assign the ID for you (generated identifier)

For the identifiers you have 2 options:

1. UUID identifier
2. Numerical identifiers

This is a huge debate to dive into but the **TLDR;** is that if you don’t need globally unique identifiers, if you won’t be merging 2 databases or have distributed systems then use auto increment and avoid UUID’s as primary keys.

Generation strategies:

* **AUTO**: Pick an appropriate strategy for the particular database
* **IDENTITY**: Insert a row without specifying a value for the ID. After inserting the row, ask the database for the last generated ID. Is only a good choice when you cannot use SEQUENCE (e.g. MySQL [not sure what this means]) because it disables JDBC batch updates.
* **SEQUENCE**: (preferred) Before inserting a new row, ask the database for the next sequence value, then insert this row with the returned sequence value as ID. It is the preferred option, especially when used with an identifier optimizer like **pooled** or **pooled-lo**. Creates a hibernate\_sequence table in the database which stores the next id to be used when a new row will be inserted.  
    
  The number of queries is thus the same in both cases for IDENTITY and SEQUENCE. **But**, Hibernate uses by default a strategy that is more efficient for the sequence generator, which it can’t use for IDENTITY. Therefore SEQUENCE is optimized much better.
* **TABLE**: Try to avoid this strategy! Scales poorly.
  + 1. hibernate\_sequence Table

If you have multiple **@Id** properties which uses the SEQUENCE strategy, only one hibernate\_sequence table will be created and each new row in those separate table will get the next sequence. So, by default, all such ID’s will get unique values.

* In the following example, both table share the same sequence counter

Text

Description automatically generatedText, letter

Description automatically generated

Text

Description automatically generatedTable

Description automatically generatedTable

Description automatically generated

* + 1. Natural Keys vs Surrogate Keys
* **Natural Key (Natürlicher Primärschlüssel**): Values that are naturally unique. They contain real data and that real data can also be used to uniquely identify each row. For example: the phone number, social security number etc.
* **Surrogate Key (Künstlicher Primärschlüssel)**: When there are no natural key candidates, a made up key is used and its only purpose it to uniquely identify the rows. For example: employee\_id.

If we work with surrogate keys, we can let JPA do the generation for us like we saw it in **@GeneratedValue**.

* + 1. JoinColumn
* **@JoinColumn** is similar to **@Column** where you can change the properties of a column. The name is Join Column because you tend to do JOINS on that foreign key column.
* It is not needed for mappings.**???**. If you don’t give a name, hibernate will automatically generate the foreign key column name.

1. CRUD Operations
   1. Create Data

an **entityManager.persist(employee)**,

* You need to start a transaction and then commit that transaction
  1. Update Data

When you do an **entityManager.persist(employee)**, if the primary key of the employee already exists, it will do an update with the value you provided.

**TODO** persistent, transient and detached

* **TODO** If you auto generate the ID and set the id yourself and try to update, you will get a “detached entity passed to persist” error.
  1. Reading Data

We can use the **find** method to fetch an entity with a given primary key as an Object

* **entityManager.find(Employee.class, 1L);**
* if the primary key does not exist, you will get a **null** back.
* When you read data from the database you do not need to create/start a transaction.
  1. Deleting Data

First we need to **find()** and then **remove()** the instance that you fetched. This unfortunately is necessary.

**TODO**, more about entity lifecycle later

1. Relational Concepts for Mapping
   1. Uni-directional and Bi-directional Relationships

**TODO**

* 1. One to One

Normally one to one relationships are not common, and are generally avoided in database design, because all fields can be put in either table. In our example we could put all the access card properties in the Employee class. But it depends on the use case. If we do a lot of processing using only the AccessCard, or if AccessCard has too many fields, then it could be useful to put it in a separate class like here.

* **One-to-one**: Use a foreign key to the referenced table. Hibernate automatically creates an ID field and makes it a foreign key.
* In the example below we have a **one-way relationship** only the Employee knows about the AccessCard.
* We could’ve put the **@OneToOne** also to the AccessCard or we can put it in both classes, if it is in both then we would have a **two-way relationship**. Hibernate then would create a foreign key in AccessCard, referencing the primary key of Employee.
* **TODO** @JoinColumn(name = “bla\_id”) in the old document, didn’t understand what that means

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated with medium confidence

Text

Description automatically generatedText

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

* 1. One to Many – Many to One

Simply put, one-to-many mapping means that one row in a table is mapped to multiple rows in another table. We will use our User-Vehicle example.

* Since a user can have many vehicles, we will define a collection of Vehicles on the Java side and annotate it with **@OneToMany** in order to tell that this maps to multiple other columns.

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text

Description automatically generated

Table

Description automatically generatedGraphical user interface, table

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generated

If we implement it like this, then **JPA** **creates 3 tables**:

Timeline

Description automatically generatedAn additional User\_Vehicle table is created.

This table is actually unnecessary, since we have a 1 to n relationship, this can/should be done in 2 tables.

But fetching the user works fine, it also fetches the vehicles:





**The proper way:**

**TODO**: <https://medium.com/@rajibrath20/the-best-way-to-map-a-onetomany-relationship-with-jpa-and-hibernate-dbbf6dba00d3>

* In a **@OneToOne** bidirectional relationship, the owner is the side which has the foreign key (and **@JoinColumn**)
* **Owning side**: The entity which contains the foreign key
* **Inverse side**: Should contain mappedBy.

The mappedBy element designates the property or field in the entity that is the owner of the relationship.

Hibernate **one to many** mapping solutions:

1. **One to many mapping with foreign key association**: One is to have a foreign key column in Vehicle table i.e. user\_id. This column will refer to primary key of User table. This way no two vehicle can be associated with multiple users.
2. **One to many mapping with join table**: Second approach is to have a common join table let’s say User\_Vehicle, This table will have two column i.e. vehicle\_id which will be foreign key referring to primary key in Vehicle table and similarly user\_id which will be foreign key referring to primary key of User table.

**TODO shouldn’t this be bidirectional?** If @OneToMany/@ManyToOne doesn’t have a mirroring @ManyToOne/@OneToMany association respectively on the child side then, the @OneToMany/@ManyToOne association is **unidirectional**.

MappedBy - @OneToOne, it prevents multiple SQL operations, see video

* Who is the owner of the relationship?
  1. Many to Many

Displays different cases: <https://www.baeldung.com/jpa-many-to-many>

In **one-to-many** relationships we could put the foreign key to the many side, in a column called the **join column**. This is called so because it is used in the JOIN operations.

In many-to-many relationships this is not possible, therefore we create a 3rd table which contains the primary keys of both tables. This table is called a **join table**.

Putting just the **@ManyToMany** annotation on both sides **will produce 2 extra join tables**, instead of 1 like we expect. This is because JPA treats both **@ManyToMany** relationships as separate relationships. This is logical because you might have multiple **@ManyToMany** annotations with the same type in your class and maybe only one is the relevant one.

* We need to tell JPA that they are referencing each other, they are the same relationship. This is done again with **mappedBy**. Only one side can have a mappedBy, else it will throw an error.
* For **@ManyToMany** bidirectional relationships, either side may be the owning side.
* **@ManyToMany** is also **LAZY** by default. Be careful when you want to switch to EAGER. It might have a cascading effect and you might fetch too many unnecessary records.

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Table

Description automatically generatedA picture containing table

Description automatically generated

Table

Description automatically generated

* + 1. Customizing the Join Table

Just like we can customize columns or join columns, we can also customize join tables.

* The **@JoinColumn** and **@JoinTable** annotations are not required. JPA will generate the table and column names for us. However, the strategy JPA uses won't always match the naming conventions we use. So, we need the possibility to configure table and column names.
* The **@JoinColumn** and **@JoinTable** annotations must be on the other side of the mappedBy. Because the mappedBy annotations are used in the referencing (inverse) side, the other side is then the owner of the relationship.
* Be careful not to switch the names of the foreign key columns by accident.

Table

Description automatically generatedGraphical user interface, text, application

Description automatically generated



Diagram

Description automatically generated

* 1. Updating a Many to Many Relationship

We just need to update our objects on the Java side, the rest JPA will handle for us. Here we first fetched and then updated both User and Fruit.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application, Teams

Description automatically generated

* + 1. Update works only if the owner is updated

Note that the row (4, 2) is not always inserted to the database. This is **probably because the Fruit owns the relationship** and if the fruit updates, the changes are also persisted but if User is updated, the changes are not persisted in the database.

* User has the annotation **@ManyToMany(mappedBy = "users")** which makes it the reference side and therefore Fruit is the owner side.

Text

Description automatically generated with medium confidenceText

Description automatically generated with low confidence

* 1. Deletion and Cascading
* When it comes to cascading a deletion, the owner side and the non-over side behaves differently  
  <https://stackoverflow.com/questions/1082095/how-to-remove-entity-with-manytomany-relationship-in-jpa-and-corresponding-join>
* Cannot delete or update a parent row: a foreign key constraint fails (`barebone-jpa-app-database`.`my\_super\_table`, CONSTRAINT `FKkakq9xxno5l2bvxr9ldp09xt3` FOREIGN KEY (`**user\_id**`) REFERENCES `**User**` (`**userId**`))
  1. Bidirectional and Unidirectional Relationships

<https://stackoverflow.com/questions/2749689/what-is-the-owning-side-in-an-orm-mapping>

* **Unidirectional** is a relation where one side does not know about the relation.
* **Bidirectional** relationship is when both sides know about each other. Meaning you can access the other side without explicit queries.

In the **database side, there is no such concept as bidirectional relationships**. Everything is unidirectional (in the database), referenced with foreign keys. Therefore, the database model will always be the same for the following examples.

* + 1. One To One
    2. One To Many
    3. Many to Many

Here we have a **unidirectional** relationship. The User “knows” about “Fruit”, but “Fruit” does not know about the User.

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text

Description automatically generated

This is a **unidirectional** relationship where each side has a reference to (“knows”) each other.

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Both produce the same database model/data:

Graphical user interface, application, table, Teams

Description automatically generatedTable

Description automatically generatedTable

Description automatically generated

* 1. Owner side and Inverse (Reference) Side

The difference comes when we want to …

For example cascade delete or UPDATE. Does not work for the inverse side?

* 1. Why Have Cyclical Relationships?

Graphical user interface, text, application

Description automatically generatedText, application

Description automatically generated

Table

Description automatically generatedTable

Description automatically generated

In this example:

* Vehicle is the **owner** of the relationship (Vehicle table contains the foreign key).
* There is no reference to Vehicle in the User side, on the database.



**So why are we doing this?**

Each individual Vehicle has already a reference to a User instance.

If we remove the can.setVehicles() and then do a fetch of user can, do a can.getVehicles(), we are still getting all the vehicles. So, why do we also set the vehicles for can?

* The answer is we want the relationships and our model to be consistent at every step of the way. Since Hibernate does a lot of caching and we can’t be sure when the entities we want to save are actually saved in the database, we always want to have a consistent model of our relationships.  
    
  If we didn’t set the vehicles explicitly, then there are certain places in our code where we get a null value if we access the list. Imagine having a lot of code, so our goal is to have the same model at every place in our code.
  1. Fetch Types - Lazy and Eager Initialization

**Fetch strategy**: Lets say our our employee has 100 addresses. If we just want the employee information without the 100 addresses, we can tell hiberante NOT to fetch all the addresses.

This is called **Lazy Initialization** in Hibernate. Meaning you do not initialize the entire object. You only initialize the first level member variables of the object, then you initialize the list only when you try to access it.   
The opposite of that is **Eager** **Initialization**.

However the moment when that field is accessed hibernate goes and does another query to get all the addresses.

**Why?** If we are just interested about certain data, hibernate shouldn’t get all the data with every single operation. Those data should be fetched only when we try to access it. It takes a lot of resources.

* **Eager Loading** is a design pattern in which data initialization occurs on the spot
* **Lazy Loading** is a design pattern which is used to defer initialization of an object as long as it’s possible

These are the default behaviors when nothing was set:

* **@OneToOne** -> **EAGER**
* **@OneToMany** -> **LAZY**
* **@ManyToOne** -> **LAZY**
* **@ManyToMany** -> **LAZY**

**Example:**

Here we can see when it’s lazy, hibernate will fetch it just before it is accessed with result.getAccessCard().

A picture containing text

Description automatically generated



Graphical user interface, text, email

Description automatically generated

* 1. Proxy Objects

**TODO**

* Are proxy objects only used when we use Lazy initialization?
* What are some of the problems with proxy objects

**TODO** – mappedBy: <https://www.youtube.com/watch?v=JbdGtsvw0o0&list=PLqq-6Pq4lTTb3J4IxKEg80LFTUbzm2p7V&index=26>

**TODO** - @ElementCollection is the same as @OneToMany??? <https://stackoverflow.com/questions/8969059/difference-between-onetomany-and-elementcollection/8969169>

TODO

* SessionFactory/Session vs EntityManagerFactory/EntityManager
* Connection Pool Text from the old document
* <https://stackoverflow.com/questions/2441598/detached-entity-passed-to-persist-error-with-jpa-ejb-code>  
  I got this error when I tried to update an entry by setting its ID manually. Document this
* TODO are transactions happening asyncly? When I do is sequentially, in the main method, delete does not work because the data is not there yet?
* Hibernate\_sequence for each table? <https://youtu.be/vT-0xALFwUA?list=PLqq-6Pq4lTTb3J4IxKEg80LFTUbzm2p7V&t=438> This is why **nfm\_api** app defined names for it