JPA – Java Persistence API

Table of Contents

[1 Introduction to JPA 2](#_Toc90035018)

[1.1 Why learn JPA? 2](#_Toc90035019)

[1.2 Book Recommendations 2](#_Toc90035020)

[1.3 Why JPA? 2](#_Toc90035021)

[1.4 Hibernate vs JPA 3](#_Toc90035022)

[1.5 How JPA works and its advantages 4](#_Toc90035023)

[1.6 Setup JPA 4](#_Toc90035024)

[1.6.1 Configure the database connection and JPA’s behavior 4](#_Toc90035025)

[1.6.2 hibernate.hbm2ddl.auto 6](#_Toc90035026)

[1.6.3 Hibernate.hbm2ddl.auto in Production? -> NOPE 6](#_Toc90035027)

[1.6.4 Schema vs Database (in MySQL) 6](#_Toc90035028)

[2 JPA Annotations 7](#_Toc90035029)

[2.1.1 @Column 7](#_Toc90035030)

[2.1.2 @Basic 7](#_Toc90035031)

[2.1.3 @Transient 7](#_Toc90035032)

[2.1.4 @Temporal 8](#_Toc90035033)

[2.1.5 Enumerations 8](#_Toc90035034)

[2.1.6 @Id 8](#_Toc90035035)

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

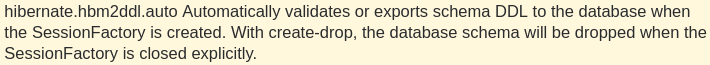
Table

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
* GenerationType.AUTO:
* GenerationType.AUTO:
* GenerationType.AUTO:
* GenerationType.AUTO:

**Create Data**

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

**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.

1. Reading Data

* When you read you can have but do not need a transaction.

We can however

Text

Description automatically generated



Asd

TODO

* SessionFactory/Session vs EntityManagerFactory/EntityManager
* Connection Pool Text