Day20: JPA Mappings, Association-Mapping, Has-A Mapping, IS-A mapping

Association mappings

Association mappings are one of the key features of JPA and Hibernate. They model the relationship between two database tables as attributes in your domain model. That allows you to easily navigate the associations in your domain model and JPQL.

JPA and Hibernate support the same associations as you know from your relational database model. You can use:

- one-to-one associations,
- many-to-one associations and
- many-to-many associations.

You can map each of them as a uni- or bidirectional association. That means you can either model them as an attribute on only one of the associated entities or on both. That has no impact on your database mapping, but it defines in which direction you can use the relationship in your domain model and JPQL or Criteria queries. I will explain that in more detail in the first example.

One-to-Many/Many-to-One

We'll get started with the *One-to-Many* and *Many-to-One* relationships, which are closely related. You could go ahead and say that they're the opposite sides of the same coin.

As its name implies, it's a relationship that links *one* entity to *many* other entities.

In our example, this would be a `Teacher` and their `Courses`. A teacher can give multiple courses, but a course is given by only one teacher (that's the *Many-to-One* perspective - many courses to one teacher).

Another example could be on social media - a photo can have many comments, but each of those comments belongs to that one photo.

```
@Entity
public class Teacher {
    private String firstName;
    private String lastName;
}

@Entity
public class Course {
    private String title;
}
```

Now, the fields of the `Teacher` class should include a list of courses. Since we'd like to map this relationship in a database, which can't include a list of entities within another entity - we'll annotate it with a `@OneToMany` annotation:

```
@OneToMany
private List<Course> courses;
```

We've used a `List` as the field type here, but we could've gone for a `Set` or a `Map`

How does JPA reflect this relationship in the database? Generally, for this type of relationship, we must use a foreign key in a table.

JPA does this for us, given our input on how it should handle the relationship. This is done via the `@JoinColumn` annotation:

```
@OneToMany
@JoinColumn(name = "TEACHER_ID", referencedColumnName = "ID")
private List<Course> courses;
```

Using this annotation will tell JPA that the `COURSE` table must have a foreign key column `TEACHER_ID` that references the `TEACHER` table's `ID` column.

Let's add some data to those tables:

```
insert into TEACHER(ID, LASTNAME, FIRSTNAME) values(1, 'Doe', 'Jane');

insert into COURSE(ID, TEACHER_ID, TITLE) values(1, 1, 'Java 101');
```

```
insert into COURSE(ID, TEACHER_ID, TITLE) values(2, 1, 'SQL 101');
insert into COURSE(ID, TEACHER_ID, TITLE) values(3, 1, 'JPA 101');
```

And now let's check if the relationship works as expected:

```
Teacher foundTeacher = entityManager.find(Teacher.class, 1L);

assertThat(foundTeacher.id()).isEqualTo(1L);
assertThat(foundTeacher.lastName()).isEqualTo("Doe");
assertThat(foundTeacher.firstName()).isEqualTo("Jane");
assertThat(foundTeacher.courses())
        .extracting(Course::title)
        .containsExactly("Java 101", "SQL 101", "JPA 101");
```

We can see that the teacher's courses are gathered automatically, when we retrieve the `Teacher` instance.

The `assertThat` is one of the JUnit methods from the Assert object that can be used to check if a specific value match to an expected one.

JUnit is an open source framework designed for the purpose of writing and running tests in the Java programming language.

It primarily accepts 2 parameters. First one if the actual value and the second is a matcher object. It will then try to compare this two and returns a boolean result if its a match or not.

Owning Side and Bidirectionality

In the previous example, the `Teacher` class is called the *owning side* of the *One-To-Many* relationship. This is because it defines the join column between the two tables.

The `Course` is called the *referencing side* in that relationship.

We could've made `Course` the owning side of the relationship by mapping the `Teacher` field with `@ManyToOne` in the `Course` class instead:

```
@ManyToOne
@JoinColumn(name = "TEACHER_ID", referencedColumnName = "ID")
private Teacher teacher;
```

There's no need to have a list of courses in the `Teacher` class now. The relationship would've worked the opposite way:

```
Course foundCourse = entityManager.find(Course.class, 1L);

assertThat(foundCourse.id()).isEqualTo(1L);
assertThat(foundCourse.title()).isEqualTo("Java 101");
assertThat(foundCourse.teacher().lastName()).isEqualTo("Doe");
assertThat(foundCourse.teacher().firstName()).isEqualTo("Jane");
```

This time, we used the `@ManyToOne` annotation, in the same way we used `@OneToMany`.

Note: It's a good practice to put the owning side of a relationship in the class/table where the foreign key will be held.

So, in our case this second version of the code is better. But, what if we still want our `Teacher` class to offer access to its `Course` list?

We can do that by defining a bidirectional relationship:

```
@Entity
public class Teacher {
// ...@OneToMany(mappedBy = "teacher")
    private List<Course> courses;
}

@Entity
public class Course {
// ...

    @ManyToOne
    @JoinColumn(name = "TEACHER_ID", referencedColumnName = "ID")
    private Teacher teacher;
}
```

We keep our `@ManyToOne` mapping on the `Course` entity. However, we also map a list of `Course`s to the `Teacher` entity.

What's important to note here is the use of the `mappedBy` flag in the `@OneToMany` annotation on the *referencing side*.

Without it, we wouldn't have a two-way relationship. We'd have two one-way relationships. Both entities would be mapping foreign keys for the other entity.

With it, we're telling JPA that the field is already *mapped by* another entity. It's mapped by the `teacher` field of the `Course` entity.

Eager vs Lazy Loading

Another thing worth noting is *eager* and *lazy* loading. With all our relationships mapped, it's wise to avoid impacting the software's memory by putting too many entities in it if unnecessary.

Lazy loading is the phenomenon that happens for loading and accessing the related entities. This means that the relationship won't be loaded right away, but only when and if actually needed. However, eager loading is referred to the practice of force-loading all these relations beforehand.

Imagine that `Course` is a heavy object, and we load all `Teacher` objects from the database for some operation. We don't need to retrieve or use the courses for this operation, but they're still being loaded alongside the `Teacher` objects.

This can be devastating for the application's performance.

Thankfully, JPA thought ahead and made *One-to-Many* relationships load *lazily* by default.

> This means that the relationship won't be loaded right away, but only when and if actually needed.

In our example, that would mean until we call on the `Teacher#courses` method, the courses are not being fetched from the database.

By contrast, *Many-to-One* relationships are *eager* by default, meaning the relationship is loaded at the same time the entity is.

We can change these characteristics by setting the `fetch` argument of both annotations:

```
@OneToMany(mappedBy = "teacher", fetch = FetchType.EAGER)
private List<Course> courses;

@ManyToOne(fetch = FetchType.LAZY)
private Teacher teacher;
```

That would inverse the way it worked initially. Courses would be loaded eagerly, as soon as we load a `Teacher` object. By contrast, the `teacher` wouldn't be loaded when we fetch `courses` if it's unneeded at the time.

Optionality

> A relationship may be optional or mandatory.

Considering the *One-to-Many* side - it is always optional, and we can't do anything about it. The *Many-to-One* side, on the other hand, offers us the option of making it *mandatory*.

By default, the relationship is optional, meaning we can save a `Course` without assigning it a teacher:

```
Course course = new Course("C# 101");
entityManager.persist(course);
```

Now, let's make this relationship mandatory. To do that, we'll use the `optional` argument of the `@ManyToOne` annotation and set it to `false` (it's `true` by default):

```
@ManyToOne(optional = false)
@JoinColumn(name = "TEACHER_ID", referencedColumnName = "ID")
private Teacher teacher;
```

Thus, we can no longer save a course without assigning a teacher to it:

```
Course course = new Course("C# 101");
assertThrows(Exception.class, () -> entityManager.persist(course));
```

But if we give it a teacher, it works fine again:

```
Teacher teacher = new Teacher();
teacher.setLastName("Doe");
teacher.setFirstName("Will");

Course course = new Course("C# 101");
course.setTeacher(teacher);

entityManager.persist(course);
```

Well, at least, it would seem so. If we had run the code, an exception would've been thrown:

```
javax.persistence.PersistenceException: org.hibernate.PersistentObjectException: detached
 entity passed to persist: com.fdpro.clients.stackabuse.jpa.domain.Course
```

Why is this? We've set a valid `Teacher` object in the `Course` object we're trying to persist. However, we haven't persisted the `Teacher` object *before* trying to persist the `Course` object.

Thus, the `Teacher` object isn't a *managed entity*. Let's fix that and try again:

```
Teacher teacher = new Teacher();
teacher.setLastName("Doe");
teacher.setFirstName("Will");
entityManager.persist(teacher);

Course course = new Course("C# 101");
course.setTeacher(teacher);

entityManager.persist(course);
entityManager.flush();
```

Running this code will persist both entities and perserve the relationship between them.

Cascading Operations

However, we could've done another thing - we could've *cascaded*, and thus propagated the persistence of the `Teacher` object when we persist the `Course` object.

This makes more sense and works the way we'd expect it to like in the first example which threw an exception.

To do this, we'll modify the `cascade` flag of the annotation:

```
@ManyToOne(optional = false, cascade = CascadeType.PERSIST)
@JoinColumn(name = "TEACHER_ID", referencedColumnName = "ID")
private Teacher teacher;
```

This way, Hibernate knows to persist the needed object in this relationship as well.

There are multiple types of cascading operations: `PERSIST`, `MERGE`, `REMOVE`, `REFRESH`, `DETACH`, and `ALL` (that combines all the previous ones).

We can also put the cascade argument on the *One-to-Many* side of the relationship, so that operations are cascaded from teachers to their courses as well.

One-to-One

This time, instead of having a relationship between one entity on one side and a bunch of entities on the other, we'll have a maximum of one entity on each side.

This is, for example, the relationship between a `Course` and its `CourseMaterial`. Let's first map `CourseMaterial`, which we haven't done yet:

```
@Entity
public class CourseMaterial {
    @Id
    private Long id;
    private String url;
}
```

The annotation for mapping a single entity to a single other entity is, unshockingly, `@OneToOne`.

Before setting it up in our model, let's remember that a relationship has an owning side - preferably the side which will hold the foreign key in the database.

In our example, that would be `CourseMaterial` as it makes sense that it references a `Course` (though we could go the other way around):

```
@OneToOne(optional = false)
@JoinColumn(name = "COURSE_ID", referencedColumnName = "ID")
private Course course;
```

There is no point in having material without a course to encompass it. That's why the relationship is not `optional` in that direction.

Speaking of direction, let's make the relationship bidirectional, so we can access the material of a course if it has one. In the `Course` class, let's add:

```
@OneToOne(mappedBy = "course")
private CourseMaterial material;
```

Here, we're telling Hibernate that the material within a `Course` is already mapped by the `course` field of the `CourseMaterial` entity.

Also, there's no `optional` attribute here as it's `true` by default, and we could imagine a course without material (from a very lazy teacher).

In addition to making the relationship bidirectional, we could also add cascading operations or make entities load eagerly or lazily.

Many-to-Many

Effectively, in a database, a *Many-to-Many* relationship involves a middle table referencing **both** other tables.

Luckily for us, JPA does most of the work, we just have to throw a few annotations out there, and it handles the rest for us.

So, for our example, the *Many-to-Many* relationship will be the one between `Student` and `Course` instances as a student can attend multiple courses, and a course can be followed by multiple students.

In order to map a *Many-to-Many* relationship we'll use the `@ManyToMany` annotation. However, this time around, we'll also be using a `@JoinTable` annotation to set up the table that represents the relationship:

```
@ManyToMany
@JoinTable(
  name = "STUDENTS_COURSES",
  joinColumns = @JoinColumn(name = "COURSE_ID", referencedColumnName = "ID"),
  inverseJoinColumns = @JoinColumn(name = "STUDENT_ID", referencedColumnName = "ID")
)
private List<Student> students;
```

First of all, we must give the table a name. We've chosen it to be `STUDENTS_COURSES`.

After that, we'll need to tell Hibernate which columns to join in order to populate `STUDENTS_COURSES`. The first parameter, `joinColumns` defines how to configure the join column (foreign key) of the owning side of the relationship in the table. In this case, the owning side is a `Course`.

On the other hand, the `inverseJoinColumns` parameter does the same, but for the referencing side (`Student`).

Let's set up a data set with students and courses:

```
Student johnDoe = new Student();
johnDoe.setFirstName("John");
johnDoe.setLastName("Doe");
johnDoe.setBirthDateAsLocalDate(LocalDate.of(2000, FEBRUARY, 18));
johnDoe.setGender(MALE);
johnDoe.setWantsNewsletter(true);
johnDoe.setAddress(new Address("Baker Street", "221B", "London"));
entityManager.persist(johnDoe);

Student willDoe = new Student();
willDoe.setFirstName("Will");
willDoe.setLastName("Doe");
willDoe.setBirthDateAsLocalDate(LocalDate.of(2001, APRIL, 4));
willDoe.setGender(MALE);
willDoe.setWantsNewsletter(false);
willDoe.setAddress(new Address("Washington Avenue", "23", "Oxford"));
entityManager.persist(willDoe);

Teacher teacher = new Teacher();
teacher.setFirstName("Jane");
teacher.setLastName("Doe");
entityManager.persist(teacher);

Course javaCourse = new Course("Java 101");
javaCourse.setTeacher(teacher);
entityManager.persist(javaCourse);
```

```
Course sqlCourse = new Course("SQL 101");
sqlCourse.setTeacher(teacher);
entityManager.persist(sqlCourse);
```

Of course, this won't work out of the box. We'll have to add a method that allows us to add students to a course. Let's modify the `Course` class a bit:

```
public class Course {

    private List<Student> students = new ArrayList<>();

    public void addStudent(Student student) {
        this.students.add(student);
    }
}
```

Now, we can complete our dataset:

```
Course javaCourse = new Course("Java 101");
javaCourse.setTeacher(teacher);
javaCourse.addStudent(johnDoe);
javaCourse.addStudent(willDoe);
entityManager.persist(javaCourse);

Course sqlCourse = new Course("SQL 101");
sqlCourse.setTeacher(teacher);
sqlCourse.addStudent(johnDoe);
entityManager.persist(sqlCourse);
```

Once this code has ran, it'll persist our `Course`, `Teacher` and `Student` instances as well as their relationships. For example, let's retrieve a student from a persisted course and check if everything's fine:

```
Course courseWithMultipleStudents = entityManager.find(Course.class, 1L);

assertThat(courseWithMultipleStudents).isNotNull();
assertThat(courseWithMultipleStudents.students())
  .hasSize(2)
  .extracting(Student::firstName)
  .containsExactly("John", "Will");
```

Of course, we can still map the relationship as bidirectional the same way we did for the previous relationships.

We can also cascade operations as well as define if entities should load lazily or eagerly (*Many-to-Many* relationships are lazy by default).

Inheritance Mapping

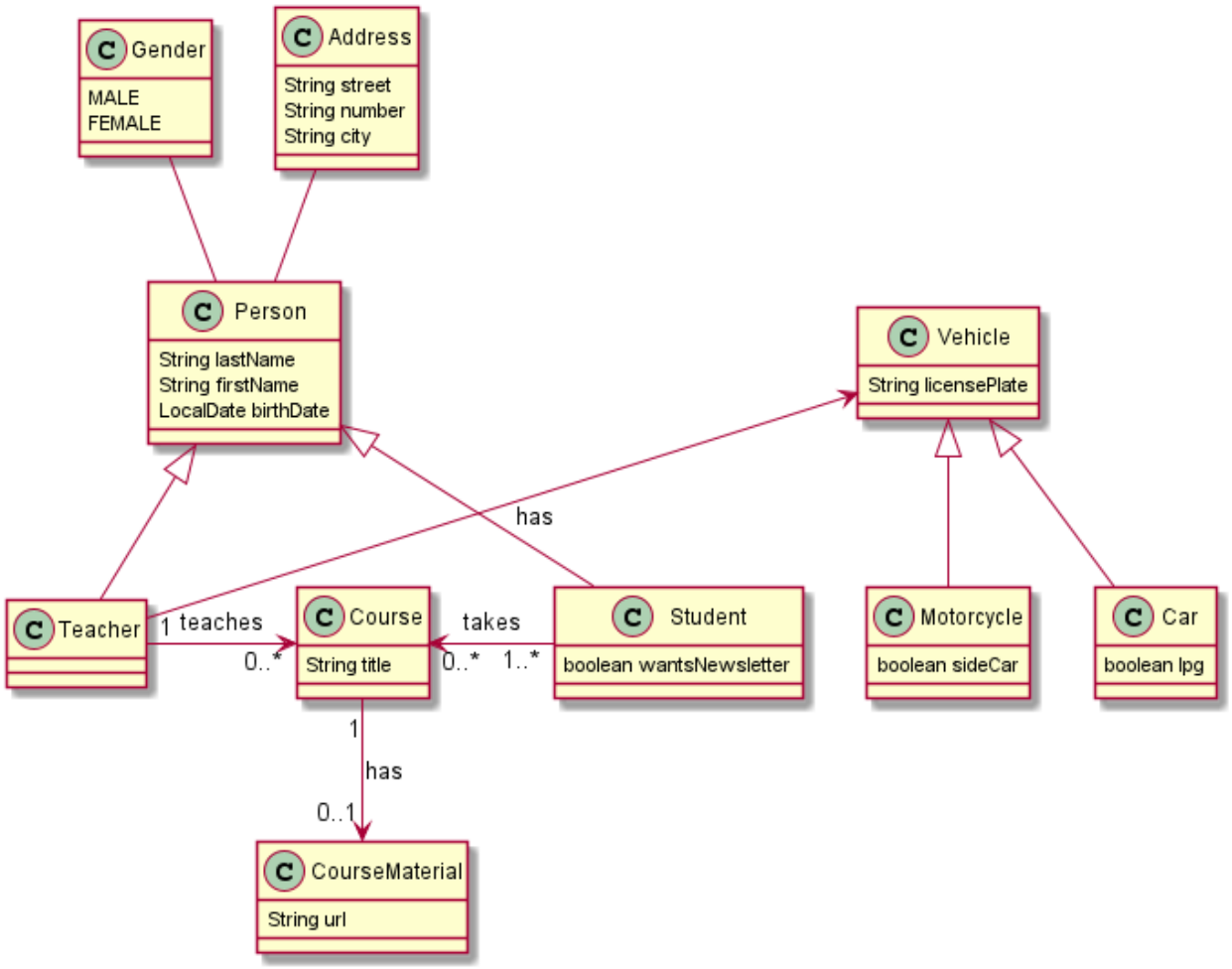
You can also map hierarchies of classes.

The idea here is to handle mapping of hierarchies of classes. JPA offers multiple strategies to achieve that, and we'll go through each of them:

- **Mapped Superclass**
- **Single Table**
- **One Table Per (Concrete) Class**
- **Joined Table**

Domain Model

First of all, let's add some inheritance in our domain model:



As we can see, we introduced the `Person` class, which is a super class of both `Teacher` and `Student` and holds names and birthdate as well as address and gender.

In addition to that, we added the `Vehicle` hierarchy to manage teacher's vehicles for parking management.

They can be `Car` or `Motorcycle`. Each vehicle has a license plate, but a car can run on LPG (which is forbidden on certain levels of the parking) and motorcycles can have a sidecar (which requires the parking space of a car).

Mapped Super-Class

Let's start with a simple one, the mapped superclass approach. **A mapped superclass is a class that's not an entity but one that contains mappings.** It's the same principle as embedded classes, but applied to inheritance.

So, let's say we want to map our new classes to handle teacher's parking in the school, we would first define the `Vehicle` class, annotated with `@MappedSuperclass`:

```
@MappedSuperclass
public class Vehicle {
    @Id
    private String licensePlate;
}
```

It only contains the identifier, annotated with `@Id`, which is the license plate of the vehicle.

Now, we want to map our two entities: `Car` and `Motorcycle`. Both will extend from `Vehicle` and inherit the `licensePlate`:

```
@Entity
class Car extends Vehicle {
    private boolean runOnLpg;
}

@Entity
class Motorcycle extends Vehicle {
    private boolean hasSideCar;
}
```

Okay, we've defined out entities now, and they inherit from `Vehicle`. Though, what happens on the database side? JPA generates these table definitions:

```
create table Car (licensePlate varchar(255) not null, runOnLpg boolean not null, primary k
ey (licensePlate))
create table Motorcycle (licensePlate varchar(255) not null, hasSideCar boolean not null,
 primary key (licensePlate))
```

Each entity has its own table, both with a `licensePlate` column, which is also the primary key of these tables. *There is no* `Vehicle` *table*. The `@MappedSuperclass` is not an entity. In fact, a class cannot have the `@Entity` and `@MappedSuperclass` annotations applied to it.

What are the consequences of `Vehicle` not being an entity? Well, we can't search for a `Vehicle` using the `EntityManager`.

Let's add some cars and a motorcycle:

```
insert into CAR(LICENSEPLATE, RUNONLPG) values('1 - ABC - 123', '1');
insert into CAR(LICENSEPLATE, RUNONLPG) values('2 - BCD - 234', '0');
insert into MOTORCYCLE(LICENSEPLATE, HASSIDECAR) values('M - ABC - 123', '0');
```

Intuitively, you might want to search for a `Vehicle` with the license plate `1 - ABC - 123`:

```
assertThrows(Exception.class, () -> entityManager.find(Vehicle.class, "1 - ABC - 123"));
```

And this will throw an exception. There are no persisted `Vehicle` entities. There are persisted `Car` entities though. Let's search for a `Car` with that license plate:

```
Car foundCar = entityManager.find(Car.class, "1 - ABC - 123");

assertThat(foundCar).isNotNull();
assertThat(foundCar.licensePlate()).isEqualTo("1 - ABC - 123");
assertThat(foundCar.runOnLpg()).isTrue();
```

Single Table Strategy

Let's now move on to the *Single Table Strategy*. This strategy allows us to map all the entities of a class hierarchy to the same database table.

If we reuse our parking example, that would mean cars and motorcycles would all be saved in a `VEHICLE` table.

In order to set up this strategy we'll need a few new annotations to help us define this relationship:

- `@Inheritance` - which defines the inheritance strategy and is used for all strategies except for mapped superclasses.
- `@DiscriminatorColumn` - which defines a column whose purpose will be to determine which entity is saved in a given database row. We'll mark this as `TYPE`, denoting the vehicle type.
- `@DiscriminatorValue` - which defines the value of the discriminator column for a given entity - so, whether this given entity is a `Car` or `Motorcycle`.

This time around, the `Vehicle` is a manged JPA `@Entity`, since we're saving it into a table. Let's also add the `@Inheritance` and `@DiscriminatorColumn` annotations to it:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE")
public class Vehicle {
    @Id
    private String licensePlate;
}
```

The `@Inheritance` annotation accepts a `strategy` flag, which we've set to `InheritanceType.SINGLE_TABLE`. This lets JPA know that we've opted for the Single Table approach. This type is also the default type, so even if we hadn't specified any strategies, it'd still be `SINGLE_TABLE`.

We also set our discriminator column name to be `TYPE` (the default being `DTYPE`). Now, when JPA generates tables, it'll look like:

```
create table Vehicle (TYPE varchar(31) not null, licensePlate varchar(255) not null, hasSi
deCar boolean, runOnLpg boolean, primary key (licensePlate))
```

That has a few consequences:

- **Fields for both `Car` and `Motorcycle` are stored in the same table**, which can become messy if we have a lot of fields.
- **All subclass' fields have to be nullable** (cause a `Car` can't have values for a `Motorcycle` fields, and vice-versa), which means less validation on the database level.

That being said, let's map our `Car` and `Motorcycle` now:

```
@Entity
@DiscriminatorValue("C")
class Car extends Vehicle {
    private boolean runOnLpg;
}

@Entity
@DiscriminatorValue("M")
class Motorcycle extends Vehicle {
```

```
    private boolean hasSideCar;
}
```

Here, we're defining the values of the discriminator column for our entities. We chose `C` for cars and `M` for motorcycles. By default, JPA is uses name of the entities. In our case, `Car` and `Motorcycle`, respectively.

Now, let's add some vehicles and take a look at how the `EntityManager` deals with them:

```
insert into VEHICLE(LICENSEPLATE, TYPE, RUNONLPG, HASSIDECAR) values('1 - ABC - 123', 'C',
'1', null);
insert into VEHICLE(LICENSEPLATE, TYPE, RUNONLPG, HASSIDECAR) values('2 - BCD - 234', 'C',
'0', null);
insert into VEHICLE(LICENSEPLATE, TYPE, RUNONLPG, HASSIDECAR) values('M - ABC - 123', 'M',
null, '0');
```

On one end, we can retrieve each `Car` or `Motorcycle` entity:

```
Car foundCar = entityManager.find(Car.class, "1 - ABC - 123");

assertThat(foundCar).isNotNull();
assertThat(foundCar.licensePlate()).isEqualTo("1 - ABC - 123");
assertThat(foundCar.runOnLpg()).isTrue();
```

But, since `Vehicle` is also an entity, we can also retrieve entities as their superclass - `Vehicle`:

```
Vehicle foundCar = entityManager.find(Vehicle.class, "1 - ABC - 123");

assertThat(foundCar).isNotNull();
assertThat(foundCar.licensePlate()).isEqualTo("1 - ABC - 123");
```

In fact, we can even save a `Vehicle` entity which is neither a `Car` nor a `Motorcycle`:

```
Vehicle vehicle = new Vehicle();
vehicle.setLicensePlate("T - ABC - 123");

entityManager.persist(vehicle);
```

Which translates into the following SQL query:

```
insert into Vehicle (TYPE, licensePlate) values ('Vehicle', ?)
```

Although we might not want that to happen - we must use the `@Entity` annotation on `Vehicle` with this strategy.

If you'd like to disable this feature, a simple option is to make the `Vehicle` class `abstract`, preventing anyone from instantiating it. If it's non-instantiable, it can't be saved as an entity, even though it's annotated as one.

One Table Per Class Strategy

The next strategy is called *One Table Per Class*, which, as the name implies, **creates one table per class in the hierarchy**.

Though, we could've used the term *"Concrete Class"* instead, since it doesn't create tables for abstract classes.

This approach looks a lot like the Mapped Superclass approach - the only difference being that the *superclass is an entity as well*.

To let JPA know we'd like to apply this strategy, we'll set the `InheritanceType` to `TABLE_PER_CLASS` in our `@Inheritance` annotation:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Vehicle {
    @Id
    private String licensePlate;
}
```

Our `Car` and `Motorcycle` classes just have to be mapped using `@Entity` and we're done. The table definitions are the same as with mapped superclass, plus a `VEHICLE` table (because it's a concrete class).

But, what differs from mapped superlcass is that we can search for a `Vehicle` entity, as well as a `Car` or `Motorcycle` entity:

```
Vehicle foundVehicle = entityManager.find(Vehicle.class, "1 - ABC - 123");
Car foundCar = entityManager.find(Car.class, "1 - ABC - 123");
```

```
assertThat(foundVehicle).isNotNull();
assertThat(foundVehicle.licensePlate()).isEqualTo("1 - ABC - 123");
assertThat(foundCar).isNotNull();
assertThat(foundCar.licensePlate()).isEqualTo("1 - ABC - 123");
assertThat(foundCar.runOnLpg()).isTrue();
```

Joined Table Strategy

Finally, there's the *Joined Table* strategy. It creates one table per entity, and keeps each column where it naturally belongs.

Let's take our `Person`/`Student`/`Teacher` hierarchy. If we implement it using the joined table strategy, we will end up with three tables:

```
create table Person (id bigint not null, city varchar(255), number varchar(255), street va
rchar(255), birthDate date, FIRST_NAME varchar(255), gender varchar(255), lastName varchar
(255), primary key (id))
create table STUD (wantsNewsletter boolean not null, id bigint not null, primary key (id))
create table Teacher (id bigint not null, primary key (id))
```

The first one, `PERSON`, gets the columns for all the fields in the `Person` entity, while the others are only getting columns for their own fields, plus the `id` that links the tables together.

When searching for a student, JPA will issue an SQL query with a join between `STUD` and `PERSON` tables in order to retrieve all the data of the student.

To map this hierarchy, we'll use the `InheritanceType.JOINED` strategy, in the `@Inheritance` annotation:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String lastName;

    @Column(name = "FIRST_NAME")
    private String firstName;

    private LocalDate birthDate;
```

```
    @Enumerated(EnumType.STRING)
    private Student.Gender gender;

    @Embedded
    private Address address;
}
```

Our other entities are just mapped using `@Entity`:

```
@Entity
public class Student extends Person {
    @Id
    private Long id;
    private boolean wantsNewsletter;
    private Gender gender;
}
```

And:

```
@Entity
public class Teacher extends Person {
    @Id
    private Long id;
```

Let's also define the ENUM we've used in the `Student` class:

```
enum GENDER {
MALE, FEMALE
}
```

There we go, we can fetch `Person`, `Student` and `Teacher` entities as well as save them using `EntityManager.persist()`.

Again, if we want to avoid creating `Person` entities we must make it `abstract`.

Inheritance vs. Composition for Plain Java Classes

In most cases, you should prefer composition when you design plain Java classes. **More on this in the unit 5.**

The general recommendation to use composition over inheritance is also valid for entity classes.

Using Composition with JPA and Hibernate

JPA and Hibernate support 2 basic ways to implement composition:

1. You can reference another entity in your composition by modelling an associatoin to it. Each entity will be mapped to its database table and can be loaded independently. A typical example is a *Person* entity which has a one-to-many association to an *Address* entity.
2. You can also use an embeddable to include its attributes and their mapping information into your entity mapping. The attributes are mapped to the same table as the other attributes of the entity. The embeddable can’t exist on its own and has no persistent identity. You can use it to define a composition of address attributes which become part of the *Person* entity.

Composition via association mapping

For a lot of developers, this is the most natural way to use composition in entity mappings. It uses concepts that are well-established and easy to use in both worlds: the database, and the Java application.

Associations between database records are very common and easy to model. You can:

- store the primary key of the associated record in one of the fields of your database record to model a one-to-many association. A typical example is a record in the *Book* table which contains the primary key of a record in the *Publisher* table. This enables you to store a reference to the publisher who published the book.
- introduce an association table to model a many-to-many relationship. Each record in this table stores the primary key of the associated record. E.g., a *BookAuthor* table stores the primary keys of records in the *Author* and *Book* tables to persist which authors wrote a specific book.

One of the main benefits of JPA and Hibernate is that you can easily map these associations in your entities. You just need an attribute of the type of the associated

entity or a collection of the associated entities and a few annotation.We will see a quick example.

The following code snippets show the mapping of a many-to-many association between the *Book* and the *Author* entity.In this example, the *Book* entity owns the association and specifies the join table with its foreign key columns. As you can see, the mapping is relatively simple. You just need an attribute of type *Set<Author>*, a *@ManyToMany* annotation which specifies the type of relationship and a *@JoinTable* annotation to define the association table.

```
1  @Entity
2  public class Book {
3  @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
4  private Long id;
5  @ManyToMany
6  @JoinTable(name = "book_author",
7  joinColumns = { @JoinColumn(name = "book_id") },
   inverseJoinColumns = { @JoinColumn(name = "author_id") })
8  private Set authors = new HashSet();
9  ...
10 }
11
12
13
14
15
```

Modelling the other end of the association is even simpler. The *books* attribute with its *@ManyToMany* annotation just references the association mapping defined on the *Book* entity.

```
1  @Entity
2  public class Author {
   @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @ManyToMany(mappedBy = "authors")
    private Set<Book> books = new HashSet<Book>();
    ...
}
```

You can use a similar approach to model one-to-one, one-to-many, and many-to-one associations.

Composition with embeddables

Embeddables are another option to use composition when implementing your entities. They enable you to define a reusable set of attributes with mapping annotations, the embeddable becomes part of the entity and has no persistent identity on its own.

Let's take a look at an example.

The 3 attributes of the *Address* class store simple address information.
The *@Embeddable* annotation tells Hibernate and any other JPA implementation that this class and its mapping annotations can be embedded into an entity. In this example, we rely on JPA's default mappings and don't provide any mapping information.

```
@Embeddable
public class Address {
    private String street;
    private String city;
    private String postalCode;
    ...
}
```

```
7
8
9
```

After you have defined your embeddable, you can use it as the type of an entity attribute. You just need to annotate it with *@Embedded*, and your persistence provider will include the attributes and mapping information of your embeddable in the entity.

So, in this example, the attributes *street*, *city* and *postalCode* of the embeddable *Address* will be mapped to columns of the *Author* table.

```
1  @Entity
2  public class Author implements Serializable {
3      @Id
       @GeneratedValue(strategy = GenerationType.AUTO)
4      private Long id;
5      @Embedded
6      private Address address;
7      ...
8  }
9
10
11
12
```

If you want to use multiple attributes of the same embeddable type, you need to override the column mappings of the attributes of the embeddable. You can do that with a collection of *@AttributeOverride* annotations, the *@AttributeOverride* annotation is repeatable, and you no longer need to wrap it in an *@AttributeOverrides* annotation.

```
1  @Entity
2  public class Author implements Serializable {
3      @Id
       @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    private Long id;
    @Embedded
    private Address privateAddress;
    @Embedded
    @AttributeOverride(
    name = "street",
    column = @Column( name = "business_street" )
    )
    @AttributeOverride(
    name = "city",
    column = @Column( name = "business_city" )
    )
    @AttributeOverride(
    name = "postalCode",
    column = @Column( name = "business_postcalcode" )
    )
    private Address businessAddress;
    ...
}
```

References:

https://stackabuse.com/a-guide-to-jpa-with-hibernate-relationship-mapping/

https://thorben-janssen.com/composition-vs-inheritance-jpa-hibernate/