Cours Inf3522 - Développement d'Applications N tier

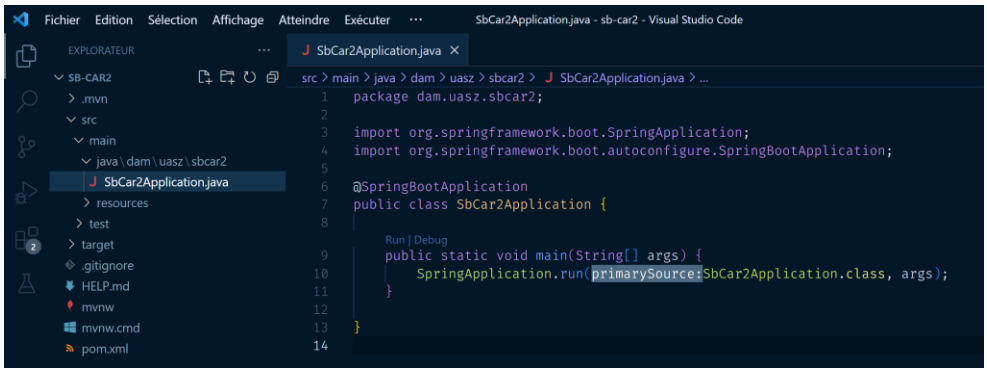
Lab_3 : JPA pour créer et accéder à une BD

Exercice 1

Créer une nouvelle application *Spring Boot* nommée **sb-car-2,** en prenant le soin d'ajouter les dépendances pour l'ORM JPA et la BD runtime H2.

spring init --dependencies=web,devtools,jpa,h2 --build=maven sb-car-2

Ouvrir le code dans vs code et vérifier que les dépendances sont bien présentes dans le fichier **pom.xml** et que l'on a l'arborescence ci-dessous :



I. Les concepts de bases : **ORM, JPA,** et **Hibernate**

- **ORM** : Object-Relational Mapping ou « *Mapping Objet-Relationnel* » permet de faire le lien entre les objets de votre application et les données stockées dans une base de données relationnelle. Il permet donc de travailler avec des données en utilisant des objets et des méthodes plutôt que des requêtes **SQL**.
- **Java Persistence API (JPA)** est un ORM et fournit une correspondance objet-relationnel pour les développeurs Java.
- **Hibernate** est l'implémentation JPA basée sur Java la plus populaire et est utilisée par défaut dans Spring Boot. Hibernate est un produit mature et est largement utilisé dans des applications à grande échelle.

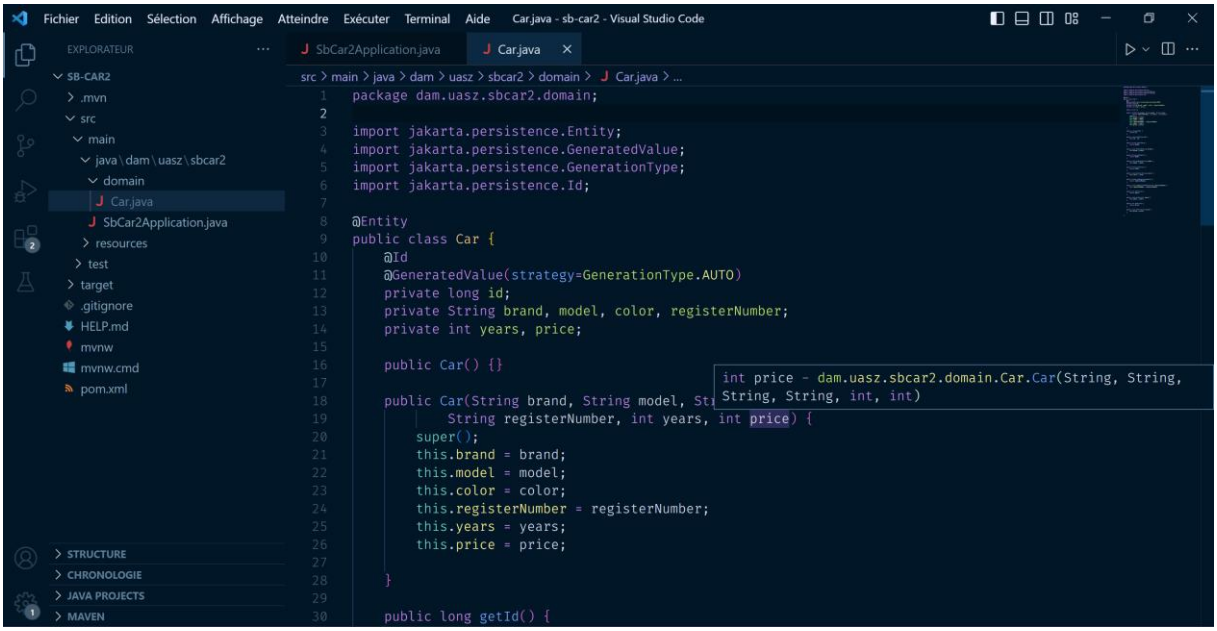
II. Création des classes d'entité

En **Java Persistence API (JPA),** une classe d'entité est une simple classe Java qui représente une table de base de données et est annotée avec l'annotation **@Entity**. Une classe d'entité doit suivre les conventions de nommage **JavaBean** et avoir des champs privés avec des méthodes getter et setter publiques.

Lorsque l'application est initialisée, **JPA** crée une table correspondante dans la base de données avec le même nom que la classe d'entité. Cependant, si vous souhaitez utiliser un nom différent pour la table dans la base de données, vous pouvez utiliser l'annotation **@Table** dans votre classe d'entité et spécifier le nom souhaité comme valeur pour l'annotation.

Une classe d'entité peut également avoir d'autres annotations pour fournir des informations supplémentaires sur l'entité, telles que l'annotation **@Id** pour indiquer le champ de clé primaire, **@Column** pour spécifier le nom et les attributs de la colonne, et **@GeneratedValue** pour indiquer comment la valeur de clé primaire est générée.

Nous allons créer une classe d'entité nommée Car dans le package **dam.uasz.sbcar2.domain.** On aura l'arborescence ci-dessous :



Faudra ajouter tous les getters et setters. Puis on lance l'application pour constater que notre application s'est connectée à une BD (regarder aux lignes où on a HiakiPool-1) dont l'url est *url=jdbc:h2:mem:25bae819-1912-415b-a5b3-941b8d2bc99a user=SA*.

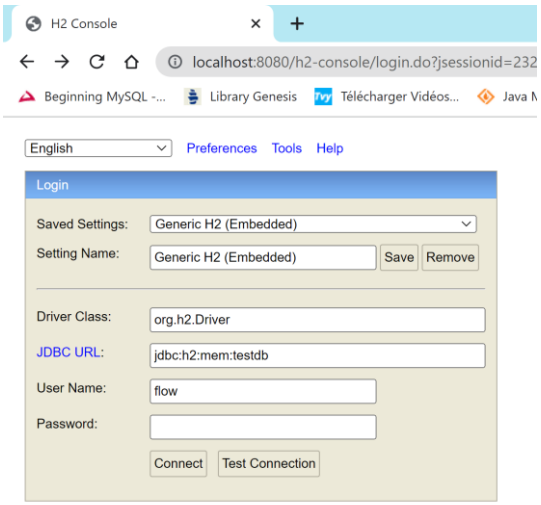
Nous n'avons écrit aucun code pour créer une BD et créer l'utilisateur associé, mais spring boot le fait pour nous grâce notamment à l'auto configuration et le principe du « *convention over configuration* ».

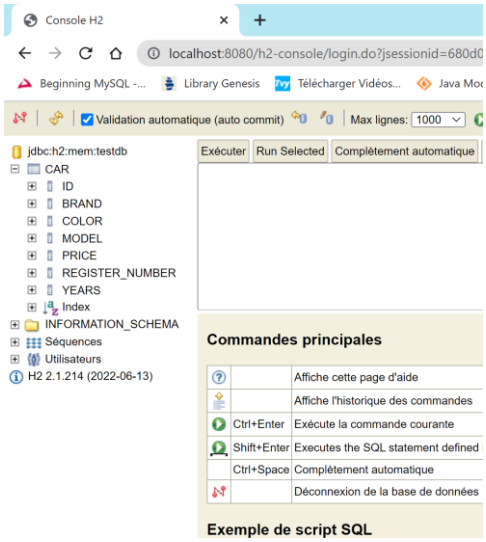
Il nous est possible également de modifier ces données par défaut en agissant sur le fichier **application.properties** dans le dossier **resource** en mettant notamment les information ci-dessous.

```
application.properties × Car.java
1 spring.datasource.url=jdbc:h2:mem:testdb
2 spring.jpa.show-sql=true
3 spring.datasource.username=flow
4
```

On relance l'application pour constater que les noms sont devenus maintenant plus compréhensibles. La ligne 2 ajoutée dans le fichier ci-dessus permet d'afficher les requêtes SQL exécutées.

La BD est accessible à l'url http://localhost:8080/h2-console. Nous constatons que notre table est bien présente. On peut également spécifier le mot de passe dans **application.properties** comme on a fait pour le « *username* ».

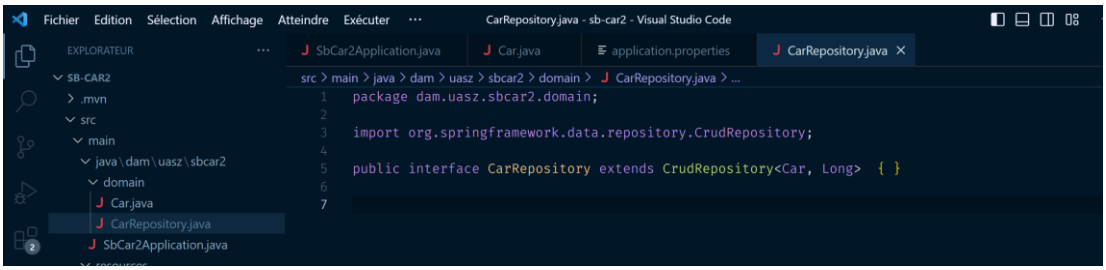




III. Creation des repositories CRUD

Spring Boot Data JPA fournit une interface **CrudRepository** pour les opérations CRUD. Elle fournit des fonctionnalités CRUD à notre classe d'entité.

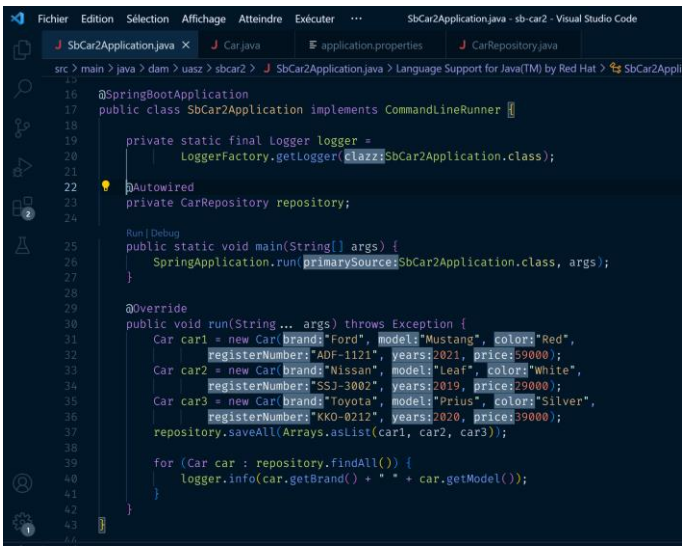
Nous allons créer un nouveau fichier dans le package **domain**. Il s'agit d'une interface **CarRepository** qui étend **CrudRepository**. Pour le moment nous n'ajouterons pas de nouvelles méthodes mais on peut faire des opérations CRUD de base.



Les arguments de type **<Car, Long>** définissent que ceci est le répertoire pour la classe d'entité Car et que le type du champ **ID est Long**. L'interface **CrudRepository** fournit plusieurs méthodes CRUD que nous pouvons maintenant commencer à utiliser. Le tableau suivant répertorie les méthodes les plus couramment utilisées.

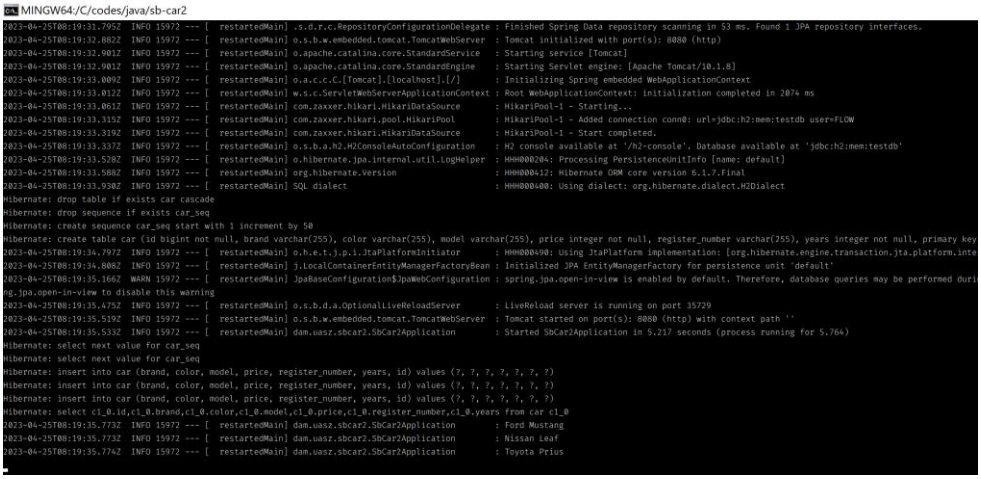
Method	Description
`long count()`	Returns the number of entities
`Iterable<T> findAll()`	Returns all items of a given type
`Optional<T> findById(ID Id)`	Returns one item by ID
`void delete(T entity)`	Deletes an entity
`void deleteAll()`	Deletes all the entities in the repository
`<S extends T> save(S entity)`	Saves an entity
`List<S> saveAll(Iterable<S> entities)`	Saves multiple entities

Maintenant, on peut ajouter des données de départ à notre base de données runtime H2. Pour cela, nous utiliserons l'interface **CommandLineRunner** de Spring Boot. L'interface **CommandLineRunner** nous permet d'exécuter du code supplémentaire avant que l'application ne soit entièrement démarrée. Par conséquent, c'est un bon point pour ajouter des données de démonstration à votre base de données. La classe principale de votre application Spring Boot implémente l'interface **CommandLineRunner**. Le code de notre classe principale ressemblera à ceci :

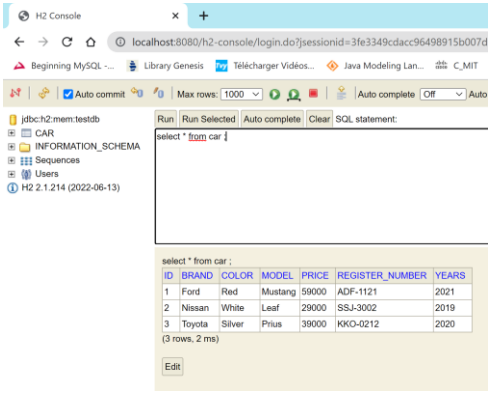


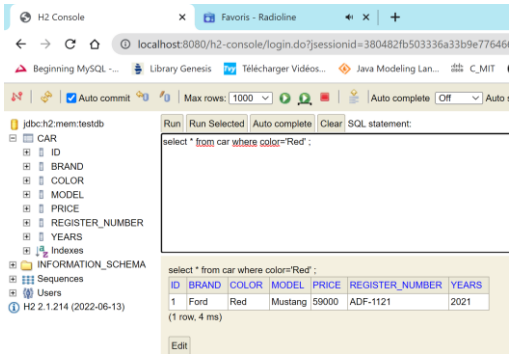
Nous pouvons constater que les requêtes sql d'insertion sont affichées dans la console. Nous avons un **logger** pour voir les trafics dans la console. A noter que l'on peut choisir les types de log à afficher dans le fichier **application.properties** en ajoutant la ligne

logging.level.root=INFO. La valeur par défaut c'est INFO, la valeur mais on peut mettre **ERREUR, WARN ou DEBUG,** ce dernier donne plus de détails.



On peut ouvrir notre interface dans H2 afin de constater si nos données sont insérées.





Il est également possible d'ajouter ses propres requêtes dans le « repository »[1].

```java
import java.util.List;

import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;

public interface CarRepository extends CrudRepository<Car, Long>  {
    // Fetch cars by brand
    List<Car> findByBrand(@Param("brand") String brand);

    // Fetch cars by color
    List<Car> findByColor(@Param("color") String color);

    // Fetch cars by brand and model
    List<Car> findByBrandAndModel(String brand, String model);

    // Fetch cars by brand or color
    List<Car> findByBrandOrColor(String brand, String color);

    // Fetch cars by model using SQL
    @Query("select c from Car c where c.model = ?1")
    List<Car> findByModel(String model);
}

```

[1] L'intérêt de cette partie dans le lab4

IV. Ajout de relations entre tables

Considérons ce modèle ci-dessous :

- Une voiture a un seul propriétaire qui peut avoir plusieurs voitures.

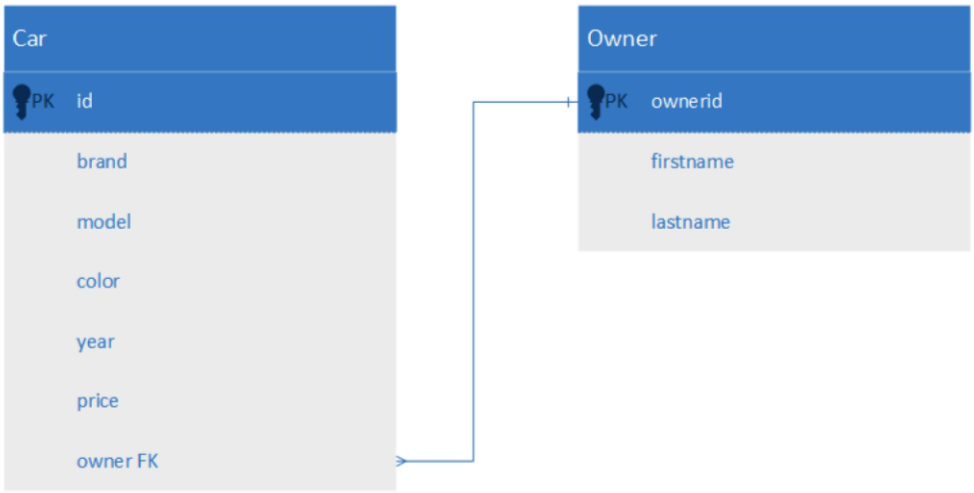
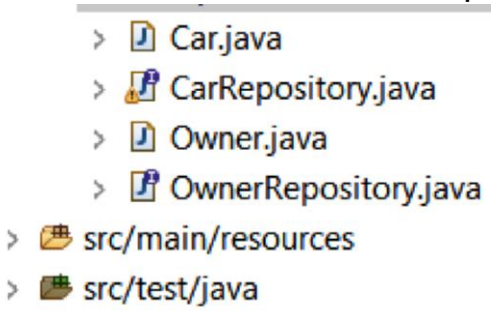
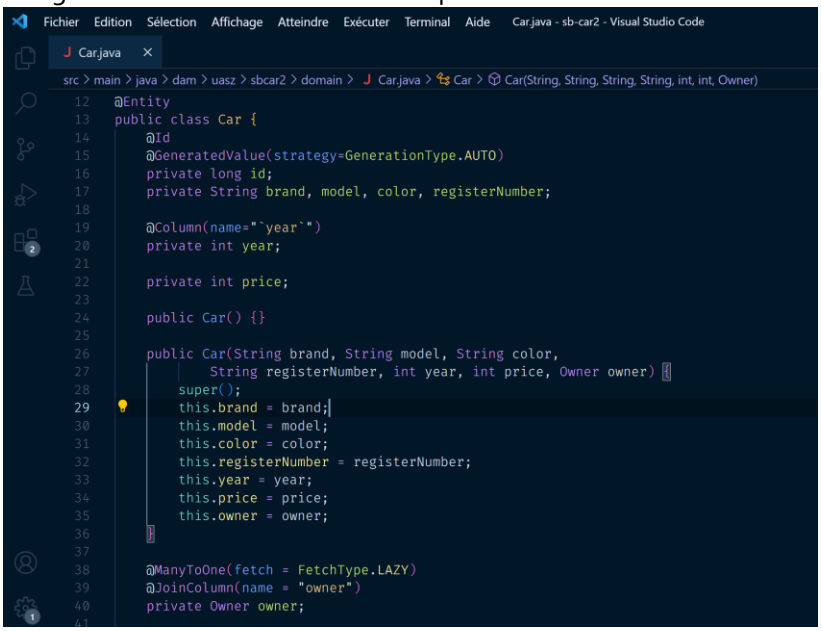


Figure 3.12 – OneToMany relationship

Créer la classe d'entité **Owner** et son **repository** correspondant dans le package **domain**.



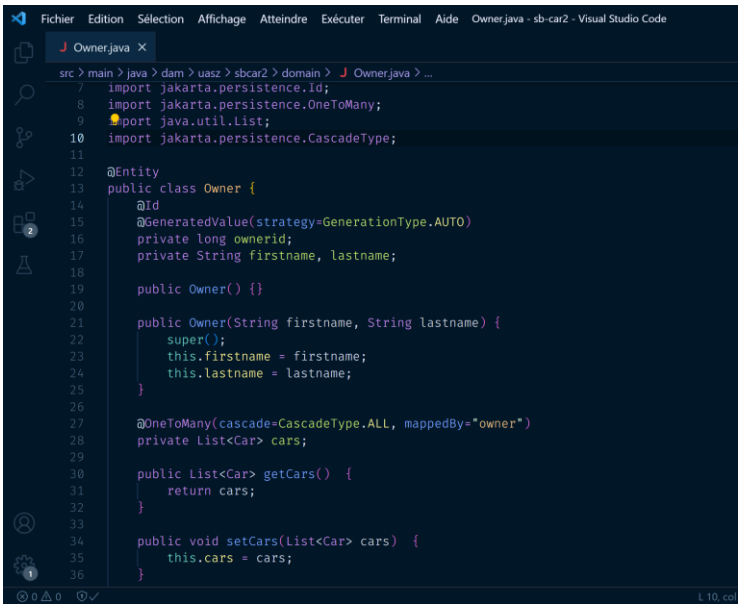
Le code de la classe **Car** sera revue comme suit en ajoutant la colonne **owner** : la clé étrangère. Le mot clé *year* est un mot clé dans sql, peut donc engendrer quelques erreurs. Il faut ajouter des getters et setters à la classe **Car** pour le nouvel attribut **owner**.



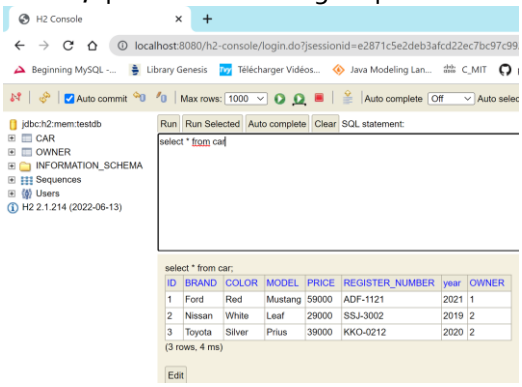
La relation un-à-plusieurs peut être ajoutée en utilisant les annotations **@ManyToOne** et **@OneToMany**. Dans la classe d'entité **Car,** qui contient une clé étrangère, vous devez définir

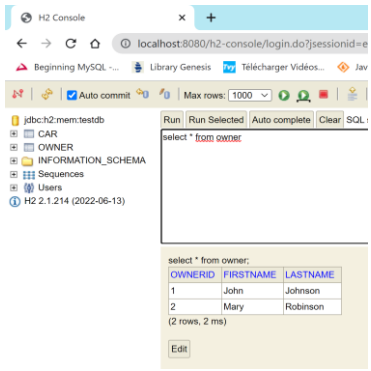
la relation avec l'annotation **@ManyToOne**. Vous devriez également ajouter les accesseurs pour le champ **owner**. Il est recommandé d'utiliser **FetchType.LAZY** pour toutes les associations. Pour les relations de type "**toMany**", c'est la valeur par défaut, mais pour les relations de type "**toOne**", vous devriez la définir. **FetchType** définit la stratégie de récupération des données à partir de la base de données. La valeur peut être soit **EAGER** soit **LAZY**. Dans notre cas, la stratégie "**lazy**" signifie que lorsque le propriétaire est récupéré de la base de données, toutes les voitures associées au propriétaire seront récupérées lorsque cela est nécessaire. "**Eager**" signifie que les voitures seront récupérées immédiatement par le propriétaire.

Le code de la classe **Owner** est présentée ci-dessous. Faudra ajouter les getters et setters pour **firstname** et **lastname**. Dans l'entité **Car**, la relation est définie avec l'annotation **@OneToMany**. Le type de champ est **List<Car>** car le propriétaire peut avoir plusieurs voitures.



L'annotation **@OneToMany** a deux attributs que nous utilisons. L'attribut **cascade** définit comment la cascade affecte les entités en cas de suppressions ou de mises à jour. Le paramètre **ALL** signifie que toutes les opérations sont en cascade. Par exemple, si le propriétaire est supprimé, les voitures qui sont liées à ce propriétaire sont également supprimées. Le paramètre **mappedBy="owner"** nous indique que la classe **Car** a le champ **owner**, qui est la clé étrangère pour cette relation.





Le code de la classe principale est donné ci-dessous :

```java
package com.dam.uasz.sbcar2;

import java.util.Arrays;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import com.dam.uasz.sbcar2.domain.*;

@SpringBootApplication
public class SbCar2Application implements CommandLineRunner {

    private static final Logger logger =
            LoggerFactory.getLogger(SbCar2Application.class);

            @Autowired
            private CarRepository repository;


            @Autowired
            private OwnerRepository orepository;


            public static void main(String[] args) {
                SpringApplication.run(SbCar2Application.class, args);
            }
```

```java
            public static void main(String[] args) {
                SpringApplication.run(SbCar2Application.class, args);
            }

            @Override
            public void run(String... args) throws Exception {
                // Add owner objects and save these to db
                Owner owner1 = new Owner("John" , "Johnson");
                Owner owner2 = new  Owner("Mary" , "Robinson");
                orepository.saveAll(Arrays.asList(owner1, owner2));

                // Add car object and link to owners and save these to db
                Car car1 = new Car("Ford", "Mustang", "Red",
                        "ADF-1121", 2021, 59000, owner1);
                Car car2 = new Car("Nissan", "Leaf", "White",
                        "SSJ-3002", 2019, 29000, owner2);
                Car car3 = new Car("Toyota", "Prius", "Silver",
                        "KKO-0212", 2020, 39000, owner2);
                repository.saveAll(Arrays.asList(car1, car2, car3));

                for (Car car : repository.findAll()) {
                    logger.info(car.getBrand() + " " + car.getModel());
                }
            }
}
```

V. Utilisation d'une BD relationnelle

Pour la suite, nous allons utiliser une BD **MariaDB** à la place de notre BD **in-memory H2**. **MariaDB** est une base de données relationnelle open source largement utilisée. Il faudra l'installer dans la suite de ce cours. Il peut venir avec un serveur comme *xampp* ou vous pouvez le télécharger et l'installer.

Les tables de la base de données sont toujours créées automatiquement par **JPA**. Cependant, avant de lancer notre application, nous devons créer une base de données pour celle-ci. Un autre SGBD relationnelle fera l'affaire.

```
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> create database cardb ;
Query OK, 1 row affected (0.001 sec)

MariaDB [(none)]>
```

Dans Spring Boot, ajoutez une dépendance cliente Java **MariaDB** au fichier **pom.xml** et supprimez la dépendance H2 car nous n'en avons plus besoin :

```xml
    <dependency>
        <groupId>org.mariadb.jdbc</groupId>
        <artifactId>mariadb-java-client</artifactId>
    </dependency>
    <dependency>
       <groupId>mysql</groupId>
       <artifactId>mysql-connector-java</artifactId>
       <version>5.1.36</version>
    </dependency>
</dependencies>
```

Ajouter ces infos de config dans **application.properties** :

SbCar2Application.java | Car.java | CarRepository.java | sb-car-2/pom.xml | application.proper

```
spring.datasource.url=jdbc:mariadb://localhost:3306/cardb
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
```

Le paramètre **spring.jpa.generate-ddl** définit si **JPA** doit initialiser la base de données (*true/false*). Les valeurs possibles sont *none*, *validate*, *update*, *create* et *create-drop*. La valeur par défaut dépend de votre base de données. Si vous utilisez une base de données intégrée comme **H2**, la valeur par défaut est *create-drop*, sinon, la valeur par défaut est *none*. *create-drop* signifie que la base de données est créée lorsque l'application démarre et supprimée lorsque l'application s'arrête. La valeur *create* ne crée la base de données que lorsque l'application démarre. La valeur *update* crée la base de données et met à jour le schéma si celui-ci a changé.

Si on redémarre le serveur, nous voyons que la BD est peuplée par les données initiales.

XAMPP for Windows - mysql -u root

```
MariaDB [cardb]> show tables ;
+-----------------+
| Tables_in_cardb |
+-----------------+
| car             |
| car_seq         |
| owner           |
| owner_seq       |
+-----------------+
4 rows in set (0.001 sec)

MariaDB [cardb]> select * from car ;
+----+--------+--------+---------+-------+-----------------+------+-------+
| id | brand  | color  | model   | price | register_number | year | owner |
+----+--------+--------+---------+-------+-----------------+------+-------+
|  1 | Ford   | Red    | Mustang | 59000 | ADF-1121        | 2021 |     1 |
|  2 | Nissan | White  | Leaf    | 29000 | SSJ-3002        | 2019 |     2 |
|  3 | Toyota | Silver | Prius   | 39000 | KKO-0212        | 2020 |     2 |
+----+--------+--------+---------+-------+-----------------+------+-------+
3 rows in set (0.001 sec)

MariaDB [cardb]>
```

Exercice 2

NB : Pour faire cet exercice, il vous est demandé de cloner l'exercice précédent. Le code de l'exercice 1 sera utilisé dans le prochain lab.

On considère maintenant qu'une voiture peut avoir plusieurs propriétaires. Essayer d'adapter les codes pour gérer les relations de plusieurs à plusieurs.

Ci-dessous, on constate qu'une table supplémentaire est automatiquement ajoutée.

La table est appelée car_owner.

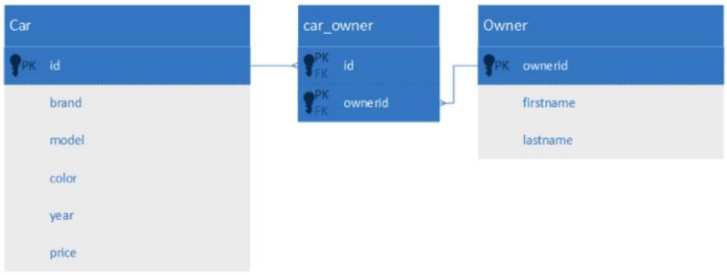


Figure 3.18 – ManyToMany relationship

La table d'association est un type particulier de table qui gère la relation de plusieurs à plusieurs entre deux tables. La table d'association est définie en utilisant l'annotation **@JoinTable**. Avec cette annotation, nous pouvons définir le nom de la table de liaison et les colonnes de liaison.

Hint ☺

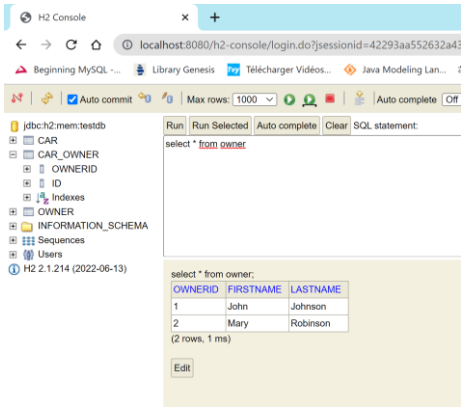
Les classes entités sont modifiées comme suit :

Car.java × CarRepository.java Owner.java OwnerRepository.java SbCar1Application.java

```java
    private int price;

    public Car() {}

    public Car(String brand, String model, String color,
            String registerNumber, int year, int price, Set<Owner> owners) {
        super();
        this.brand = brand;
        this.model = model;
        this.color = color;
        this.registerNumber = registerNumber;
        this.year = year;
        this.price = price;
        this.owners = owners;
    }

    @ManyToMany(mappedBy="cars")
    private Set<Owner> owners = new HashSet<Owner>();
    public Set<Owner> getOwners() {
        return owners;
    }

    public void setOwner(Set<Owner> owners)  {
        this.owners = owners;
    }
```

```java

@Entity
public class Owner {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long ownerid;
    private String firstname, lastname;

    public Owner() {}

    public Owner(String firstname, String lastname) {
        super();
        this.firstname = firstname;
        this.lastname = lastname;
    }

    @ManyToMany(cascade=CascadeType.PERSIST)
    @JoinTable(name="car_owner", joinColumns = { @JoinColumn(name="ownerid") },
    inverseJoinColumns = { @JoinColumn(name="id") })
    private Set<Car> cars = new HashSet<Car>();

    public Set<Car> getCars() {
        return cars;
    }
```

```
sb-car-1 - SbCar1Application [Spring Boot App] C:\Users\DELL\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe  (25 avr. 2023, 23:45:30) [pi
Hibernate: drop sequence if exists owner_seq
Hibernate: create sequence car_seq start with 1 increment by 50
Hibernate: create sequence owner_seq start with 1 increment by 50
Hibernate: create table car (id bigint not null, brand varchar(255), color varchar(255), model varchar(255), price integer not null,
Hibernate: create table car_owner (ownerid bigint not null, id bigint not null, primary key (ownerid, id))
Hibernate: create table owner (ownerid bigint not null, firstname varchar(255), lastname varchar(255), primary key (ownerid))
Hibernate: alter table if exists car_owner add constraint FKofvh7bce0mwfwfud5r1k21w3p foreign key (id) references car
Hibernate: alter table if exists car_owner add constraint FKrd5fu5fk3bil77mf9m6viqixx foreign key (ownerid) references owner
2023-04-25T23:45:37.275Z  INFO 12572 --- [  restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000490: Using JtaPlatform im
2023-04-25T23:45:37.290Z  INFO 12572 --- [  restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFa
2023-04-25T23:45:37.991Z  WARN 12572 --- [  restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enab
2023-04-25T23:45:38.387Z  INFO 12572 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on
2023-04-25T23:45:38.441Z  INFO 12572 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080
2023-04-25T23:45:38.459Z  INFO 12572 --- [  restartedMain] com.dam.uasz.sbcar1.SbCar1Application    : Started SbCar1Application in 5.
Hibernate: select next value for owner_seq
Hibernate: select next value for owner_seq
Hibernate: insert into owner (firstname, lastname, ownerid) values (?, ?, ?)
Hibernate: insert into owner (firstname, lastname, ownerid) values (?, ?, ?)
Hibernate: select next value for car_seq
Hibernate: select next value for car_seq
Hibernate: insert into car (brand, color, model, price, register_number, "year", id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into car (brand, color, model, price, register_number, "year", id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: insert into car (brand, color, model, price, register_number, "year", id) values (?, ?, ?, ?, ?, ?, ?)
Hibernate: select c1_0.id,c1_0.brand,c1_0.color,c1_0.model,c1_0.price,c1_0.register_number,c1_0."year" from car c1_0
```



Finir l'exercice2 ☹