



# Framework Spring

## Partie Modèle

TP n°1

—

LP DAWIN | Module JEE

Année 2022 – 2023

H. Berger | M.A. Tessier

# Sommaire

<b>SOMMAIRE.....</b>	<b>0</b>
<b>1 INTRODUCTION.....</b>	<b>0</b>
<b>2 OUTILS UTILISES.....</b>	<b>0</b>
<b>3 APPLICATION JEE - SPRING FRAMEWORK PETCLINIC .....</b>	<b>0</b>
3.1 PREREQUIS.....	1
3.1.1 Récupération du code source et création de votre projet dans GitLab .....	1
3.1.2 Get Started.....	1
3.2 FONCTIONNALITES DE L'APPLICATION PETCLINIC .....	2
3.2.1 Accueil.....	2
3.2.2 Menu des propriétaires.....	2
3.2.3 Menu des vétérinaires.....	6
<b>4 SPRING FRAMEWORK – PARTIE MODELE .....</b>	<b>0</b>
4.1 ARCHITECTURE D'UNE APPLICATION SPRING .....	0
4.2 MODELE DE L'APPLICATION PETCLINIC.....	0
4.2.1 Diagramme de classe.....	1
4.2.2 Utilisation de l'héritage :.....	1
4.2.3 Navigabilité :.....	1
4.2.4 Annotations JPA pour la persistance :.....	2
4.3 AJOUT DE L'ENTITY MEMO AU MODEL.....	4
4.3.1 Diagramme de classes mis à jour .....	8
4.4 AJOUT DE L'ENTITY OPERATION AU MODEL .....	12
4.4.1 Exercice 2 (à rendre) : Gestion des opérations .....	12
4.5 LES TESTS UNITAIRES .....	12
4.5.1 Exercice 3 (pas à rendre) Test unitaire classe Owner .....	14
4.5.2 Exercice 4 (à rendre) Tests unitaires des classes Pet et Vet.....	14

# 1 Introduction

Nous allons étudier lors de ces 4 séances de TP une application appelée **Petclinic**.

Cette application de démonstration est développée à partir du **framework** Spring.

L'objectif de ces séances est de découvrir et comprendre les concepts mis en œuvre dans les applications Web développées en Java, d'apprivoiser le framework Spring et de développer de nouvelles fonctionnalités dans une application existante, tel que vous serez amené à le faire dans le monde de l'entreprise ou du logiciel libre par exemple.

Ce framework Java Open Source est largement utilisé en entreprise. Il permet de couvrir de nombreux cas d'usages. Il est facile à prendre en main, robuste, documenté et dispose d'une grande communauté d'utilisateurs.

## 2 Outils utilisés

La liste ci-dessous contient les outils utilisés dans le cadre des Travaux Pratiques. Veuillez à vous assurer que vous les avez à votre disposition sur votre machine :

- Un outil de gestion de version : [Git](#)
- Un IDE moderne pour ceux qui le souhaitent tels que [IntelliJ IDEA](#) / [Eclipse](#) / [VSCode](#) ou un éditeur de texte pour les minimalistes !
- Java Development Kit Java (JDK) version 8, 11 ou 17 en version open Source : par exemple: <https://adoptium.net>
- Un outil de build de projet Java : [Maven 3.3+](#)
- Un conteneur de servlet (serveur d'application Java) tels que [Tomcat 9 ou 10](#) ou [Jetty 9 ou 10](#)

Nous allons étudier une application de démonstration du framework [Spring](#): il s'agit de l'application [Spring Framework Petclinic](#)

**Attention** : il existe plusieurs versions de cette application. Nous allons travailler avec la version framework.

## 3 Application JEE - Spring Framework Petclinic

L'application Petclinic, une application de gestion d'une clinique vétérinaire.

Nous verrons comment elle est structurée et nous la compléterons.

L'application est construite à l'aide de l'outil **Maven**, il s'agit d'un outil Open Source qui permet de construire, déployer, exécuter des tests, générer de la documentation pour des projets développés en Java et notamment les projets **JEE**.

La configuration d'un projet Maven est définie dans le fichier **pom.xml** (Project Object Model)

Le fichier pom.xml indique notamment l'identité du projet, les dépendances (bibliothèques) utilisées dans le projet ainsi que les plugins.

Documentation rapide : <https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>

Les principales commandes maven sont les suivantes (en anglais)

- **clean**: cleans up artifacts created by prior builds
- **compile**: compile the source code of the project
- **test**: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **package**: take the compiled code and package it in its distributable format, such as a JAR.
- **verify**: run any checks to verify the package is valid and meets quality criteria
- **install**: install the package into the local repository, for use as a dependency in other projects locally

## 3.1 Prérequis

### 3.1.1 Récupération du code source et création de votre projet dans GitLab

Cloner le dépôt GIT en local

```
git clone https://github.com/spring-petclinic/spring-framework-petclinic.git
```

Supprimer le répertoire **.git** du projet cloné

Créer un dépôt petclinic spécifique au TD de JEE sur votre compte git puis ajouter le code source du projet petclinic que vous venez de cloner.

Commit et Push du code source sur votre dépôt Git.

Attribuer les droits en lecture sur votre dépôt GIT à votre enseignant (Hélène BERGER ou Marc-Antoine TESSIER)

### 3.1.2 Get Started

Avant de débiter, prenez le temps de lire le fichier **Readme.md** du projet (il s'agit d'une bonne pratique) il contient le plus souvent de précieuses informations et vous évitera de perdre du temps.

Construire l'application puis déployez l'application avec **maven** et le plugin jetty (la commande se trouve dans le fichier **readme**)

L'application doit être accessible sur votre poste local normalement via l'URL suivante <http://localhost:8080/> (aussi précisé dans le **readme**)

Importer le projet dans votre IDE

## 3.2 Fonctionnalités de l'application petclinic

### 3.2.1 Accueil

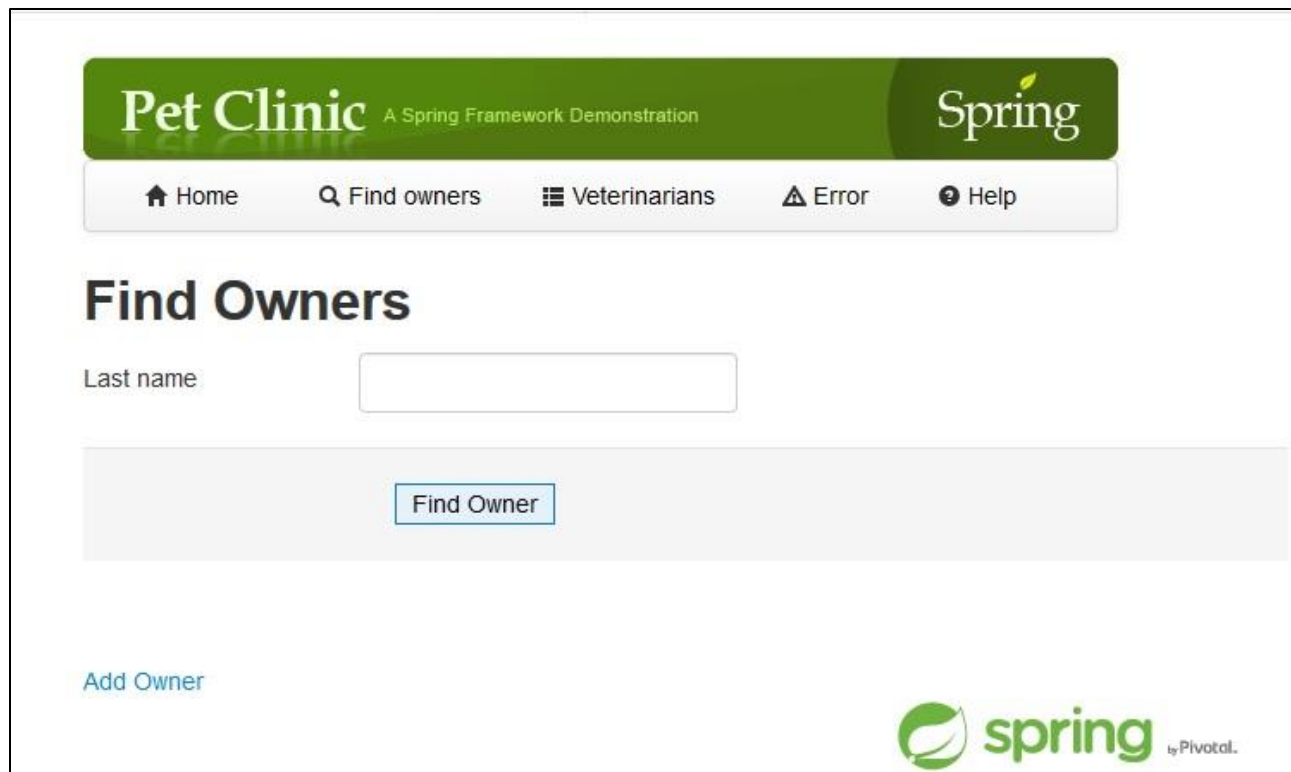
La page d'accueil permet d'accéder au menu principal.



Page d'accueil de l'application

### 3.2.2 Menu des propriétaires

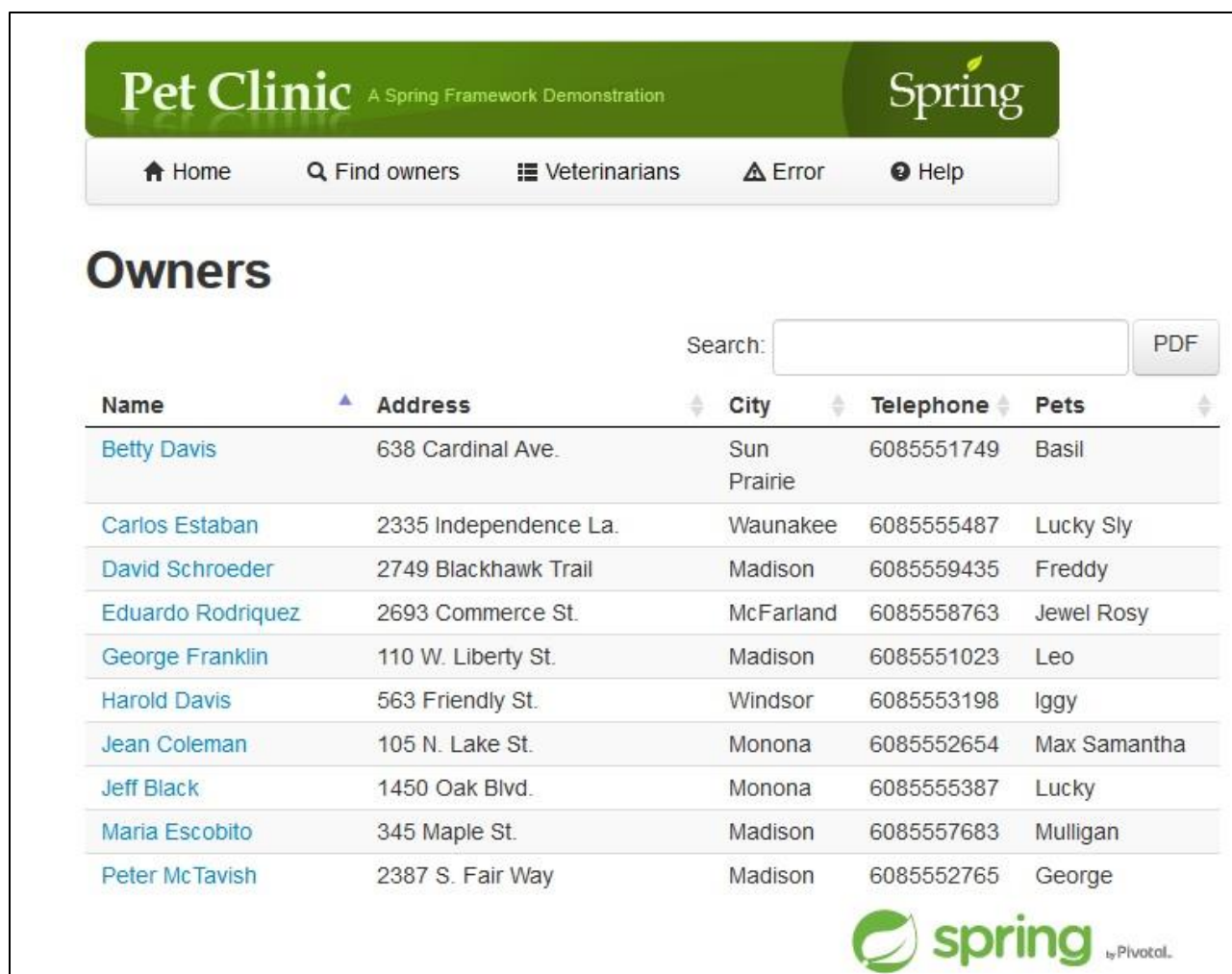
Aller dans le menu **Find owners** et cliquer sur **Find owner** sans rien saisir dans le champ **Last name** :



The screenshot displays the 'Pet Clinic' application, a Spring Framework demonstration. The header features the 'Pet Clinic' logo and the text 'A Spring Framework Demonstration' on the left, and the 'Spring' logo on the right. Below the header is a navigation bar with links: Home (house icon), Find owners (magnifying glass icon), Veterinarians (list icon), Error (warning triangle icon), and Help (person icon). The main content area is titled 'Find Owners'. It contains a text input field labeled 'Last name' and a 'Find Owner' button. At the bottom left, there is a link 'Add Owner'. At the bottom right, the 'spring' logo and 'by Pivotal.' are displayed.

Page de recherche des propriétaires

La liste complète des propriétaires (**owners**) apparaît avec leur adresse et leurs animaux domestiques (**pets**). Il s'agit d'une liste de liens :



The screenshot displays the 'Pet Clinic' application, a Spring Framework demonstration. The header features the 'Pet Clinic' logo and the Spring logo. A navigation bar includes links for Home, Find owners, Veterinarians, Error, and Help. The main section is titled 'Owners' and contains a search bar with a 'PDF' button. Below the search bar is a table listing ten owners with their details.

Name	Address	City	Telephone	Pets
Betty Davis	638 Cardinal Ave.	Sun Prairie	6085551749	Basil
Carlos Estaban	2335 Independence La.	Waunakee	6085555487	Lucky Sly
David Schroeder	2749 Blackhawk Trail	Madison	6085559435	Freddy
Eduardo Rodriquez	2693 Commerce St.	McFarland	6085558763	Jewel Rosy
George Franklin	110 W. Liberty St.	Madison	6085551023	Leo
Harold Davis	563 Friendly St.	Windsor	6085553198	Iggy
Jean Coleman	105 N. Lake St.	Monona	6085552654	Max Samantha
Jeff Black	1450 Oak Blvd.	Monona	6085555387	Lucky
Maria Escobito	345 Maple St.	Madison	6085557683	Mulligan
Peter McTavish	2387 S. Fair Way	Madison	6085552765	George

The Spring logo and 'by Pivotal.' are visible at the bottom right of the interface.

Page de résultat des propriétaires après une recherche

En cliquant sur un propriétaire, le détail des informations de ce propriétaire s'affiche avec un bouton **Edit Owner** pour pouvoir modifier les informations de celui-ci et un bouton **Add New Pet** pour pouvoir lui ajouter un animal domestique.

Pour chaque animal domestique, la liste des visites auprès d'un des vétérinaires de la clinique est affichée.

Le lien **Edit Pet** permet de modifier l'animal et le lien **Add Visit** de planifier une nouvelle visite.

## Owner Information

Name	Carlos Estaban
Address	2335 Independence La.
City	Waunakee
Telephone	6085555487
<a href="#">Edit Owner</a> <a href="#">Add New Pet</a>	

## Pets and Visits

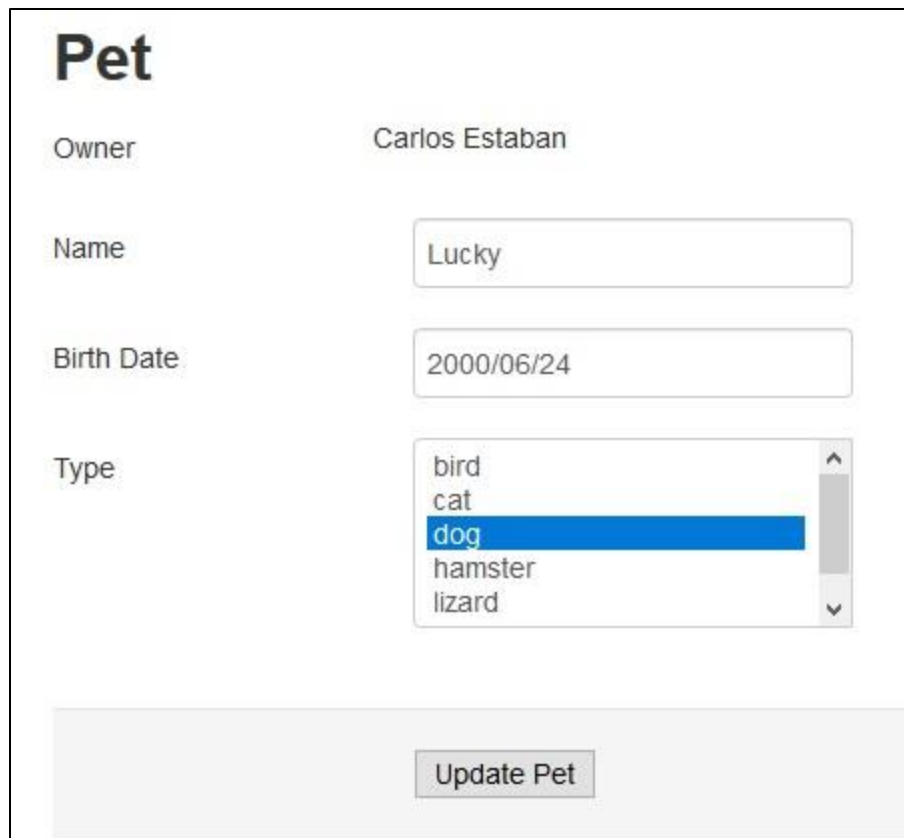
Name	Visit Date	Description
Lucky		
Birth Date	2000-06-24	<a href="#">Edit Pet</a> <a href="#">Add Visit</a>
Type	dog	

Name	Visit Date	Description
Sly		
Birth Date	2002-06-08	<a href="#">Edit Pet</a> <a href="#">Add Visit</a>
Type	cat	

Page de détail d'un propriétaire

En cliquant sur **Edit Pet**, on voit le détail de l'animal domestique et on peut le modifier.  
Chaque animal domestique a un type.





**Pet**

Owner Carlos Estaban

Name Lucky

Birth Date 2000/06/24

Type

- bird
- cat
- dog
- hamster
- lizard

Update Pet

Page de mise à jour d'un animal domestique

### 3.2.3 Menu des vétérinaires

Le menu **Veterinarians** permet d'afficher la liste des vétérinaires en indiquant pour chacun sa spécialité.

## Veterinarians

Search: 

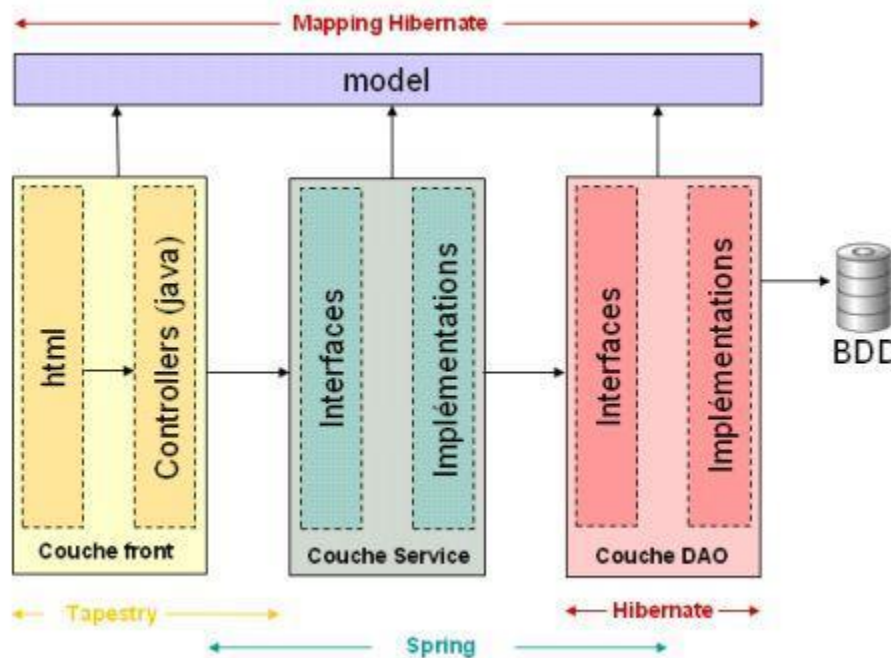
Name	Specialties
Helen Leary	radiology
Henry Stevens	radiology
James Carter	none
Linda Douglas	dentistry surgery
Rafael Ortega	surgery
Sharon Jenkins	none

Page de liste des vétérinaires

## 4 Spring Framework – Partie modèle

### 4.1 Architecture d'une application Spring

Spring est un framework permettant de développer une application web d'entreprise. Il respecte l'architecture MVC en étant structuré de la manière suivante :



Modèle d'Architecture en couches

### 4.2 Modèle de l'application petclinic

Dans ce premier TP, nous allons travailler sur la couche du Modèle (**model**)

Cette couche est transversale car elle est utilisée par les trois couches **front**, **service** et **DAO** pour Data Access Object de l'architecture d'une application développée avec le framework Spring.

Le modèle contient tous les objets manipulés dans l'application qui vont permettre de véhiculer les informations entre les différentes couches. Les classes de ces objets sont appelées des Entités (**Entity**).

Elles contiennent des attributs avec des getters et des setters.

Elles vont contenir également des annotations **JPA** permettant de gérer la persistance des objets.

**JPA** est une API (Application Programming Interface), autrement dit un standard utilisé dans les applications **JEE** pour gérer la persistance des objets. Cette API est basée sur **Hibernate** (framework de persistance).

### 4.2.1 Diagramme de classe

Voici le diagramme de classes du Model de l'application petclinic :

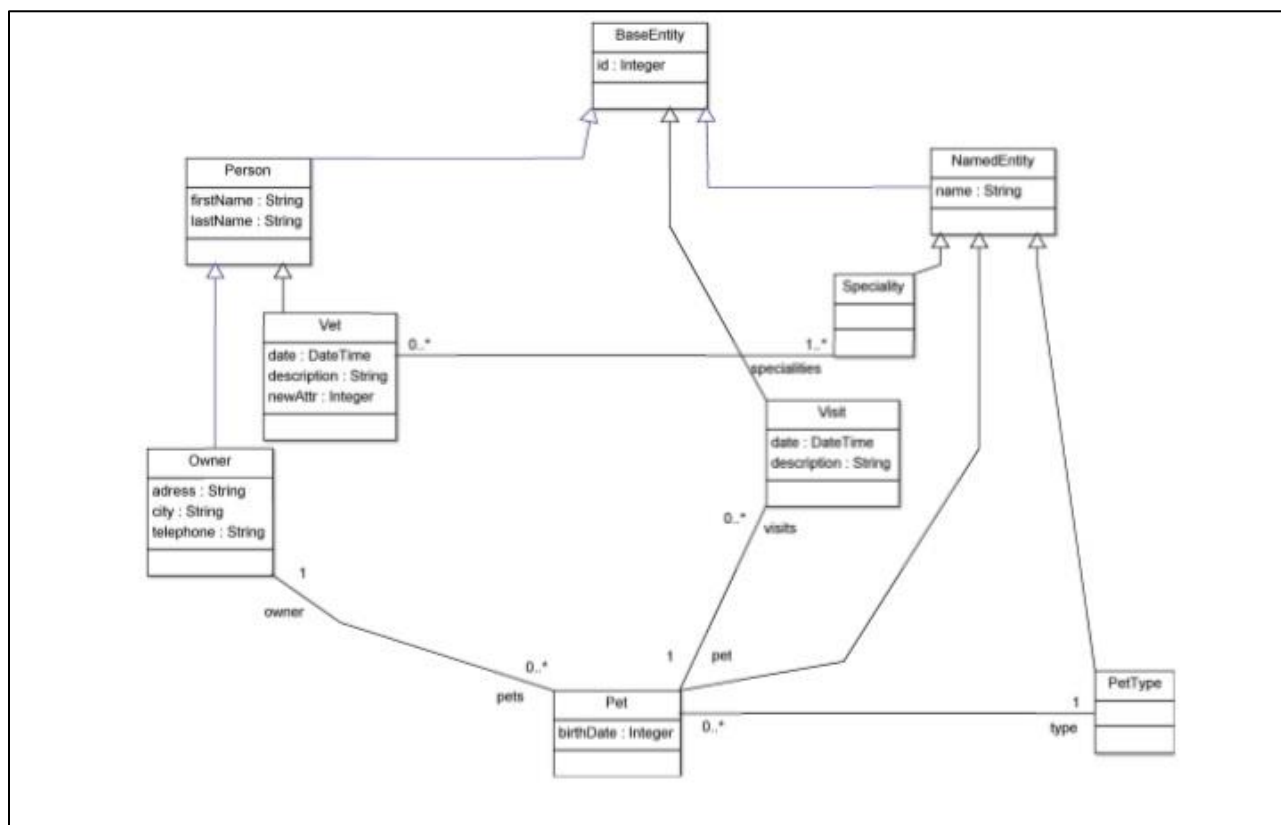


Diagramme de classe original de la couche model

### 4.2.2 Utilisation de l'héritage :

On remarque que toutes les classes héritent (directement ou indirectement) de la super classe **BaseEntity** qui contient juste un id comme attribut.

Certaines classes héritent de la classe **Person** qui contient en plus les attributs **firstName** et **lastName** : les classes **Vet** et **Owner**.

D'autres classes héritent de la classe **NamedEntity** qui contient l'attribut **name** : les classes **Speciality**, **Pet** et **PetType**.

La classe **Visit** hérite directement de la classe **BaseEntity**.

### 4.2.3 Navigabilité :

On remarque que plusieurs navigabilités entre les classes sont indiquées. On entend par navigabilité le fait qu'une entité est liée à une autre entité via une relation (jointure en SQL)

Par exemple, entre les classes **Owner** et **Pet** : Un propriétaire peut posséder 0 ou plusieurs animaux :

- **owner** est indiqué sur le lien du côté **Owner** ce qui va se traduire par la présence de l'attribut **owner** dans la classe **Pet** (aller vérifier dans le code de la classe dans le projet : <src/main/java/org/springframework/samples/petclinic/model>).
- **pets** est indiqué sur le lien côté **Pet** ce qui va se traduire par la présence de l'attribut **pets** dans la classe **Owner**, qui contient la liste des animaux domestiques (**pets**) du propriétaire (**owner**).

#### 4.2.3.1 Exercice 1 (pas à rendre)

Trouver les autres navigabilités présentes et vérifier la présence des attributs correspondant dans les différentes classes du Model concernées

### 4.2.4 Annotations JPA pour la persistance :

Observons maintenant les annotations JPA, utilisées pour la persistance :

Dans la classe **Visit** par exemple :

```

33 @Entity
34 @Table(name = "visits")
35 public class Visit extends BaseEntity {
36
37     /**
38      * Holds value of property date.
39      */
40     @Column(name = "visit_date")
41     @DateTimeFormat(pattern = "yyyy/MM/dd")
42     private LocalDate date;
43
44     /**
45      * Holds value of property description.
46      */
47     @NotEmpty
48     @Column(name = "description")
49     private String description;
50
51     /**
52      * Holds value of property pet.
53      */
54     @ManyToOne
55     @JoinColumn(name = "pet_id")
56     private Pet pet;
57

```

Classe Visit

#### 4.2.4.1 Signification des annotations utilisées :

**@Entity** → indique qu'il s'agit d'une entité à mapper

**@Table(name = "visits")** → table de la base de donnée utilisée pour le mapping de cette classe (table «visits » , voir ci-dessous le modèle de la base de données).

**@Column(name = "visit\_date")** → nom de colonne dans la table correspondant à cet attribut

**@DateTimeFormat(pattern = "yyyy/MM/dd")** → indique le format de date

**@NotEmpty** → indique que l'attribut ne doit pas être vide

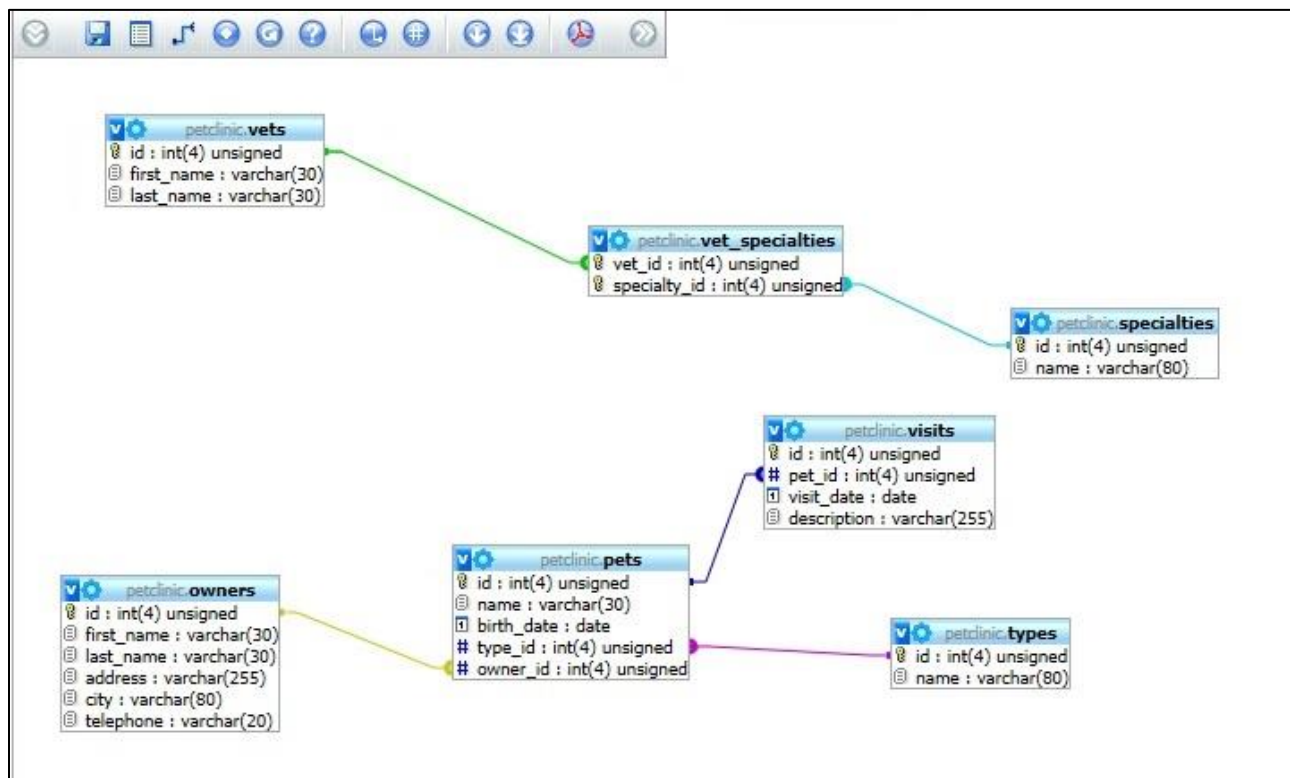
**@Column(name = "description")** → nom de colonne dans la table correspondant à cet attribut

**@ManyToOne** → signifie « **plusieurs** Visit (classe courante d'abord) pour **un** Pet »

(cohérent avec la multiplicité 0.\* côté Visit sur le lien Pet-Visit: « un animal domestique peut avoir 0 ou plusieurs visites »)

**@JoinColumn(name = "pet\_id")** → il faut indiquer la clé étrangère de la table visits

#### 4.2.4.2 Modèle de la base de données :



Et dans la classe Pet :

```

45 @Entity
46 @Table(name = "pets")
47 public class Pet extends NamedEntity {
48
49     @Column(name = "birth_date")
50     @DateTimeFormat(pattern = "yyyy/MM/dd")
51     private LocalDate birthDate;
52
53     @ManyToOne
54     @JoinColumn(name = "type_id")
55     private PetType type;
56
57     @ManyToOne
58     @JoinColumn(name = "owner_id")
59     private Owner owner;
60
61     @OneToMany(cascade = CascadeType.ALL, mappedBy = "pet", fetch = FetchType.EAGER)
62     private Set<Visit> visits;
63

```

Classe Pet

#### 4.2.4.3 Signification des annotations utilisées :

**@OneToMany(cascade = CascadeType.ALL, mappedBy = "pet", fetch = FetchType.EAGER) private Set<Visit> visits;** → signifie « **un** Pet (classe courante d'abord) peut avoir **plusieurs** Visits »

(cohérent avec la multiplicité 0.\* côté Visit sur le lien Pet-Visit : « un animal domestique peut avoir 0 ou plusieurs visites »)

**mappedBy = "pet"** → ATTENTION : il faut bien indiquer l'attribut dû à la navigabilité qui est dans la classe liée (ici dans la classe Visit)

## 4.3 Ajout de l'Entity Memo au Model

Nous allons compléter l'application avec la gestion de mémos : désormais, chaque vétérinaire pourra saisir des mémos dans l'application et il sera possible d'afficher la liste des mémos de chaque vétérinaire.

Chaque mémo aura :

- un id
- une date (**memo\_date**)
- une description

Un mémo appartenant à un vétérinaire, il y aura une clé étrangère **vet\_id** dans la table memos).

Pour cela, nous allons d'abord modifier la base de données en ajoutant la table **memos**.

## Exercice

### Mettre à jour la base de données

Compléter les fichiers **schema.sql** et **data.sql**.

Celui-ci est utilisé pour créer le modèle de données lors du premier démarrage de l'application, il est situé dans l'arborescence du projet dans **src/main/resources/db/h2**

Ajouter avec les instruction de SQL suivantes permettant de créer la table **memos**:

```
CREATE TABLE memos (
  id          INTEGER IDENTITY PRIMARY KEY,
  vet_id      INTEGER NOT NULL,
  memo_date   DATE,
  description  VARCHAR(255)
);
ALTER TABLE memos ADD CONSTRAINT fk_memos_vets FOREIGN KEY (vet_id) REFERENCES vets (id);
CREATE INDEX memos_vet_id ON memos (vet_id);
```

Ajouter l'instruction de suppression de la table en début de fichier:

```
DROP TABLE memos IF EXISTS;
```

Puis, pour obtenir un jeu de données, compléter le fichier **data.sql** avec les instructions d'insertion suivantes :

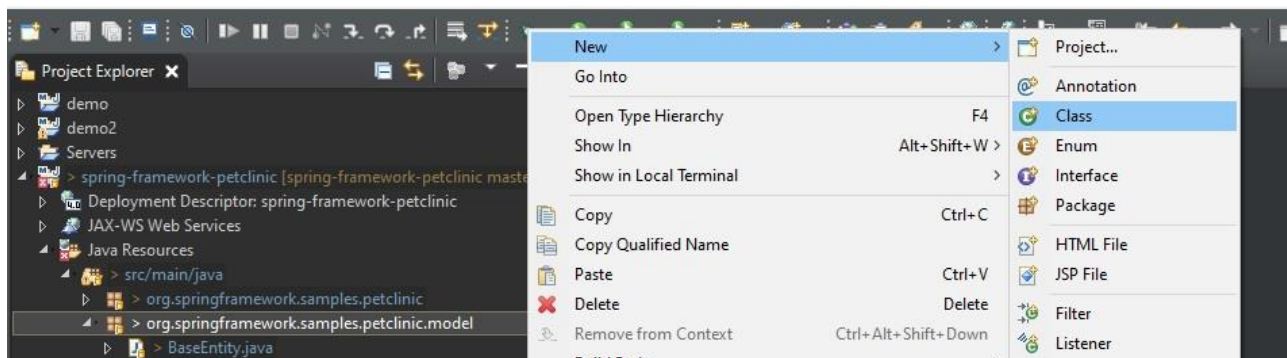
```
INSERT INTO memos VALUES (1, 1, '2018-10-07', 'planifier operations');
INSERT INTO memos VALUES (2, 1, '2018-10-09', 'organiser visites');
INSERT INTO memos VALUES (3, 1, '2018-10-09', 'contacter labo');
INSERT INTO memos VALUES (4, 3, '2018-10-05', 'planifier réunion');
INSERT INTO memos VALUES (5, 3, '2018-10-07', 'prévoir congés');
INSERT INTO memos VALUES (6, 3, '2018-10-07', 'commander matériel');
```

Nous devons maintenant compléter le modèle :

### Compléter le Modèle :

Ajouter une nouvelle entité **Memo** (classe java) dans le package **model**.

Exemple avec l'IDE Eclipse :



Ajout d'une classe Java dans le package org.springframework.samples.petclinic.model

Copier le contenu suivant dans la classe **Memo.java** que vous venez de créer :



```

package org.springframework.samples.petclinic.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import javax.validation.constraints.NotEmpty;
import org.springframework.format.annotation.DateTimeFormat;
import java.time.LocalDate;

/**
 * Simple JavaBean domain object representing a memo.
 *
 * @author Helena Berger
 */
@Entity
@Table(name = "memos")
public class Memo extends BaseEntity {

    /**
     * Holds value of property date.
     */
    @Column(name = "memo_date")
    @DateTimeFormat(pattern = "yyyy/MM/dd")
    private LocalDate date;

    /**
     * Holds value of property description.
     */
    @NotEmpty
    @Column(name = "description")
    private String description;

    /**
     * Holds value of property vet.
     */
    @ManyToOne
    @JoinColumn(name = "vet_id")
    private Vet vet;

    /**
     * Creates a new instance of Memo for the current date
     */
    public Memo() {
        this.date = LocalDate.now();
    }

    /**
     * Getter for property date.
     */
}

```

```
    * @return Value of property date.
    */
    public LocalDate getDate() {
        return this.date;
    }

    /**
     * Setter for property date.
     *
     * @param date New value of property date.
     */
    public void setDate(LocalDate date) {
        this.date = date;
    }

    /**
     * Getter for property description.
     *
     * @return Value of property description.
     */
    public String getDescription() {
        return this.description;
    }

    /**
     * Setter for property description.
     *
     * @param description New value of property description.
     */
    public void setDescription(String description) {
        this.description = description;
    }

    /**
     * Getter for property vet.
     *
     * @return Value of property vet.
     */
    public Vet getVet() {
        return this.vet;
    }

    /**
     * Setter for property vet.
     *
     * @param vet New value of property vet.
     */
    public void setVet(Vet vet) {
        this.vet = vet;
    }
}
```

Faire les ajouts nécessaires dans la classe **Vet.java** pour permettre la navigabilité entre l'entité mémo (**memo**) et l'entité vétérinaire (**vet**) à partir du code source suivant :

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "vet")
private Set<Memo> memos;

protected Set<Memo> getMemosInternal() {
    if (this.memos == null) {
        this.memos = new HashSet<>();
    }
    return this.memos;
}

protected void setMemosInternal(Set<Memo> memos) {
    this.memos = memos;
}

public List<Memo> getMemos() {
    List<Memo> sortedMemos = new ArrayList<>(getMemosInternal());
    PropertyComparator.sort(sortedMemos, new MutableSortDefinition("date", true,
true));
    return Collections.unmodifiableList(sortedMemos);
}

public void addMemo(Memo memo) {
    getMemosInternal().add(memo);
    memo.setVet(this);
}

public int getNrOfMemos() {
    return getMemosInternal().size();
}
```

**! ATTENTION :** il s'agit d'ajouts dans la classe **Vet**, ne pas écraser le contenu de la classe.

Ajouter les imports nécessaires :

```
import javax.persistence.OneToMany;
import javax.persistence.CascadeType;
```

### 4.3.1 Diagramme de classes mis à jour

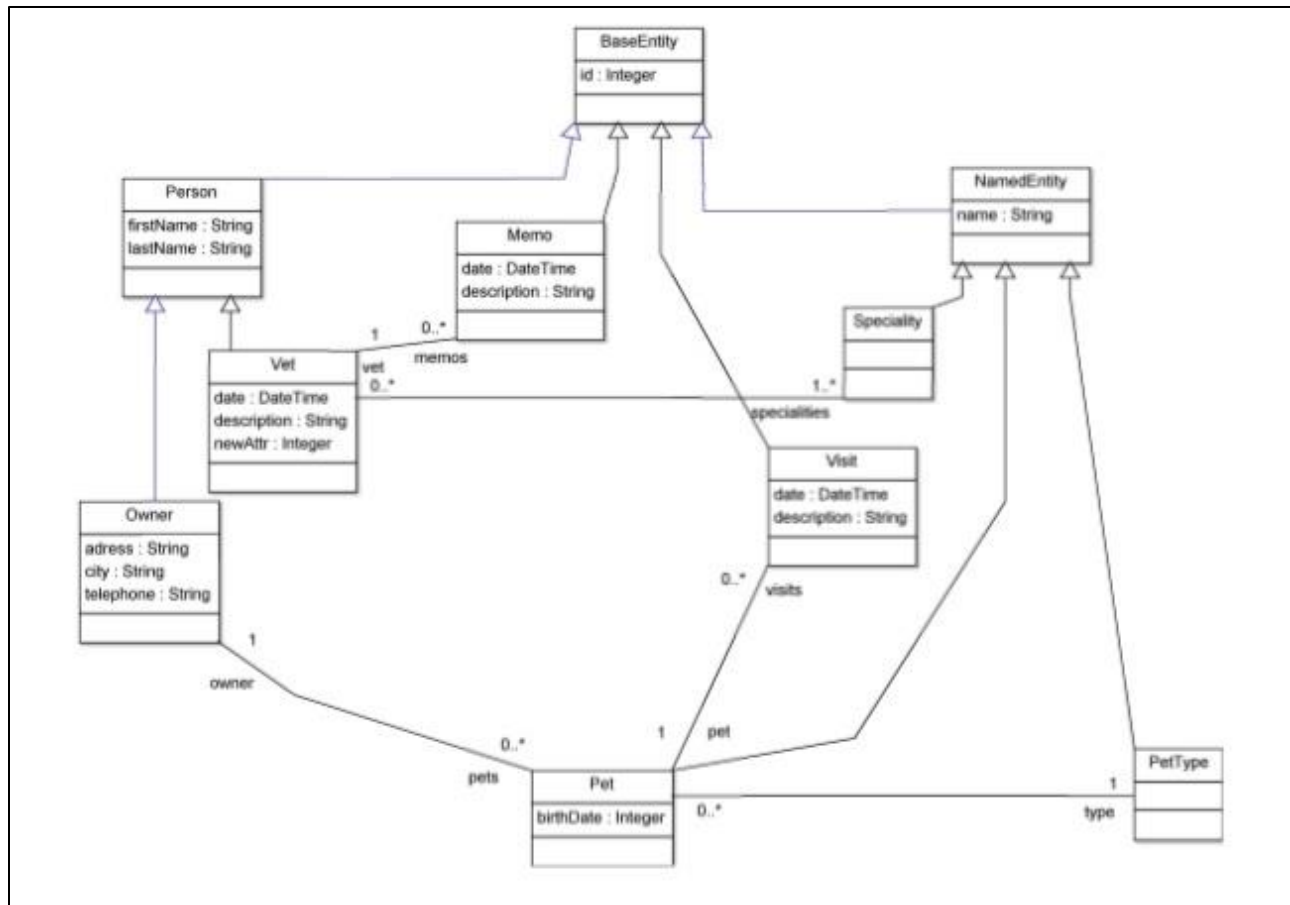


Diagramme de classe mis à jour

Observons le code des classes Memo et Vet :

```

Memo.java X
35 @Entity
36 @Table(name = "memos")
37 public class Memo extends BaseEntity {
38
39     /**
40      * Holds value of property date.
41      */
42     @Column(name = "memo_date")
43     @DateTimeFormat(pattern = "yyyy/MM/dd")
44     private LocalDate date;
45
46     /**
47      * Holds value of property description.
48      */
49     @NotEmpty
50     @Column(name = "description")
51     private String description;
52
53     /**
54      * Holds value of property vet.
55      */
56     @ManyToOne
57     @JoinColumn(name = "vet_id")
58     private Vet vet;
59

```

Classe Memo.java

On retrouve les mêmes annotations que précédemment :

**@Entity, @Table, @Column, @NotEmpty...** → Bien vérifier la correspondance entre les noms de tables et les noms de colonnes.

**@DateTimeFormat** pour le format de date.

L'attribut **vet** de type **Vet** a bien été ajouté pour gérer la navigabilité :

Un mémo appartient à un seul vétérinaire et on souhaite avoir accès à l'objet **Vet** correspondant à chaque objet **Memo**.

**@ManyToOne** → Signifie « Plusieurs mémos (classe courante d'abord) pour un vétérinaire »

**@JoinColumn (name= « vet\_id »)**

- A mettre absolument avec **@ManyToOne** (si on a **@OneToMany** de l'autre côté)
- Pour indiquer la clé étrangère de la table memos sur laquelle se fera la jointure entre les tables memos et vets

Allez observer l'attribut memos dans la classe Vet :

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "vet")
private Set<Memo> memos;
```

**@OneToMany** → Signifie « Un vétérinaire (classe courante d'abord) peut avoir plusieurs mémos »

**mappedBy= « vet »** → Attribut vet de la classe Memo (dû à la navigabilité) à indiquer côté **@OneToMany**

On remarque aussi les méthodes suivantes, semblables à celles présentes dans la classe Pet pour les visites :

```
83 protected Set<Memo> getMemosInternal() {
84     if (this.memos == null) {
85         this.memos = new HashSet<>();
86     }
87     return this.memos;
88 }
89
90 protected void setMemosInternal(Set<Memo> memos) {
91     this.memos = memos;
92 }
93
94 public List<Memo> getMemos() {
95     List<Memo> sortedMemos = new ArrayList<>(getMemosInternal());
96     PropertyComparator.sort(sortedMemos, new MutableSortDefinition("date", true, true));
97     return Collections.unmodifiableList(sortedMemos);
98 }
99
100 public void addMemo(Memo memo) {
101     getMemosInternal().add(memo);
102     memo.setVet(this);
103 }
104
105 public int getNrOfMemos() {
106     return getMemosInternal().size();
107 }
```

Les méthodes **protected setMemosInternal()** et **getMemosInternal()** peuvent être appelées uniquement à l'intérieur de la classe ou dans les classes filles. Il s'agit de getter/setter à visibilité interne.

Le setter initialise l'attribut memos s'il est nul.

Les méthodes **getMemos()** et **addMemo()** sont publiques et peuvent être appelées à l'extérieur de la classe.

La méthode **getMemos()** est en quelque sorte un *getter* qui retourne la liste des mémos triés par date.

Compiler et exécuter l'application en ligne de commande ou via un plugin de l'IDE

Le code doit compiler sans erreur et l'application doit se lancer sans erreur

Vérifier dans les traces applicatives (logs)

Cependant, nous verrons plus loin comment effectuer des tests unitaires pour les classes du Model.

## 4.4 Ajout de l'Entity Operation au Model

### 4.4.1 Exercice 2 (à rendre) : Gestion des opérations

Compléter le diagramme de classe afin de prendre en compte la nouvelle règle de gestion suivante concernant les « **opérations** »:

Le diagramme de classe est à rendre au format papier ou en ligne via un éditeur de modèle de base de donnée:

- basé sur plant UML <https://plantuml-editor.kkeisuke.com>
- format propriétaire : <https://dbdiagram.io/>

Faire apparaître uniquement les nouvelles classes et celles qui sont liées en indiquant bien les multiplicités et la navigabilité

Désormais l'application va gérer les opérations sur les animaux domestique:

- Une opération concerne un animal et est réalisée par un seul vétérinaire.
- Un animal peut subir plusieurs opérations et un vétérinaire peut réaliser plusieurs opérations.
- Chaque opération aura :
  - Un id
  - Une description (**description**)
  - Une date (**operation\_date**)

Compléter la base de données et le modèle afin de prendre en compte la nouvelle règle de gestion concernant les « **opérations** ».

Fichiers à mettre à jour et à pousser dans votre dépôt GIT : schema.sql, data.sql, ainsi que les nouveaux fichiers du modèle et fichiers mis à jour dans le modèle (**remarque** : bien implémenter toutes les navigabilités possibles).

Compiler et exécuter l'application en ligne de commande ou via un plugin de l'IDE

Le code doit compiler sans erreur et l'application doit se lancer sans erreur

## 4.5 Les tests unitaires

Nous allons maintenant réaliser des tests unitaires.

Les tests unitaires permettent de tester les méthodes des différentes classes.

Créer une classe de test **OwnerTests.java** dans le package de model de test du projet : *src/test/java/org/springframework/samples/petclinic/model*

Puis copier le contenu suivant :

```
package org.springframework.samples.petclinic.model;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;
import org.springframework.transaction.annotation.Transactional;

class OwnerTest {

    @Test
    @Transactional
    public void testHasPet() {
        Owner owner = new Owner();
        Pet fido = new Pet();
        fido.setName("Fido");
        assertNull(owner.getPet("Fido"));
        assertNull(owner.getPet("fido"));
        owner.addPet(fido);
        assertEquals(fido, owner.getPet("Fido"));
        assertEquals(fido, owner.getPet("fido"));
    }

}
```

Ces tests unitaires sont réalisés à l'aide du framework **Junit** (voir les import).

Chaque test doit être précédé de l'annotation **@Test**.

L'annotation **@Transactional** indique que l'on est en mode transactionnel (nécessaire si le code écrit génère des requêtes de mise à jour en base de données).

Le test unitaire **testHasPet()** permet de tester la méthode **addPet()** de la classe **Owner**.

Exécuter ce test unitaire dans l'IDE.

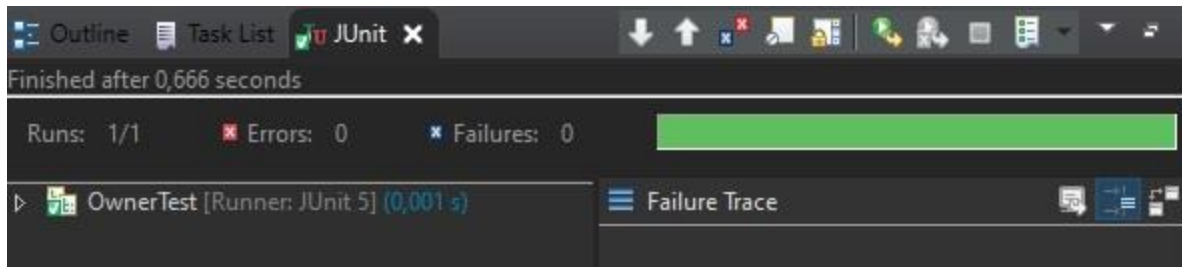
Exemple ci-dessous avec Eclipse :

*clic droit sur le fichier OwnerTest puis Run As JUnit Test :*



La barre verte indiquant que le test a été passé avec succès doit s'afficher :





Ou bien en ligne de commande :

```
mvn clean test -Dtest=OwnerTest#testHasPet
```

#### 4.5.1 Exercice 3 (pas à rendre) Test unitaire classe Owner

Commenter une à une les lignes de code du test unitaire **testHasPet()**

- Ligne 13
- Ligne 14
- Ligne 15
- Ligne 16
- Ligne 17
- Ligne 18
- Ligne 19
- Ligne 20

#### 4.5.2 Exercice 4 (à rendre) Tests unitaires des classes Pet et Vet

De la même manière que précédemment, écrire et exécuter les tests unitaires suivants:

**testHasVisit() et testHasOperation() de la classe Pet.**  
**testHasMemo() et testHasOperation() de la classe Vet**

Pensez à ajouter les fichiers **PetTests.java** et **VetTests.java** dans votre dépôt via des Commit et Push régulier.

A la fin du TD déposez un TAG GIT sur votre commit pour indiquer le travail réalisé