Framework Spring Partie Repository et Service

TP n°2

LP DAWIN | Module JEE

Année 2022 – 2023

H. Berger | M.A. Tessier

Sommaire

1 Introduction

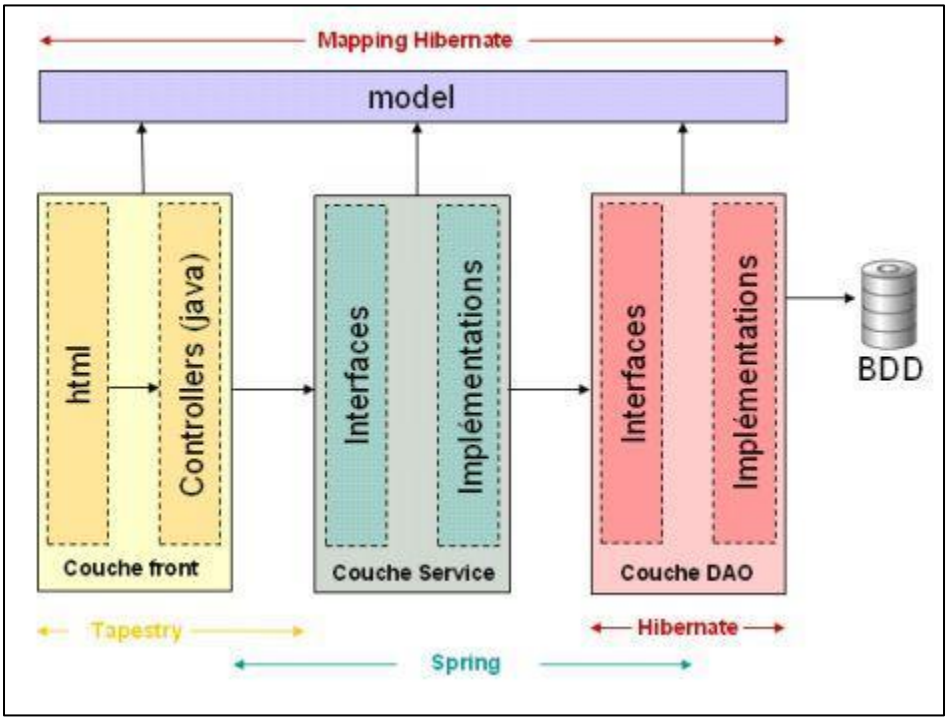
Lors du premier TP, le modèle de donnée JAVA (partie **model**) de l'application a été étudié.

Pour rappel, le modèle est la représentation **JAVA** du modèle de donnée de la base en SQL. Les classes qui décrivent les tables et les relations entres elles utilisent des annotations de la spécification **JPA** (Java Persistence API)

La manipulation des objets du modèle pour la lecture et l'écriture est réalisée par la couche de plus bas niveau appelée DAO pour Data Access Object,les classes associées sont appelées **Repository**

La couche intermédiaire de traitement des données est la couche **Service**, c'est elle qui contient toute la logique métier de l'application.

Nous allons étudier dans ce TP ces 2 couches de l'architecture type du framework Spring :



Architecture en couche typique Spring

2 La couche DAO

La couche **DAO** (Data Access Object) est la couche d'accès aux données.

Elle contient toutes les méthodes **CRUD** (Create Read Update Delete) permettant de :

- Mettre à jour la base de données (à l'aide de requêtes **SQL** de type *insert*, *update* et *delete*)
- Interroger la base de données (à l'aide de requêtes **SQL** de type *select*)

Cette couche utilise les objets du model (vu dans le TP1) soit pour :

- Récupérer les informations à mettre à jour en base de données
- Stocker les informations récupérées en base de données

La couche DAO dans Spring est représentée par des classes appelées **Repository**.

2.1 Utilisation d'interface de Repository

2.1.1 Utilisation d'interface :

On remarque que les couches **DAO** et **Service** de l'architecture Spring utilisent toutes les 2 des interfaces JAVA.

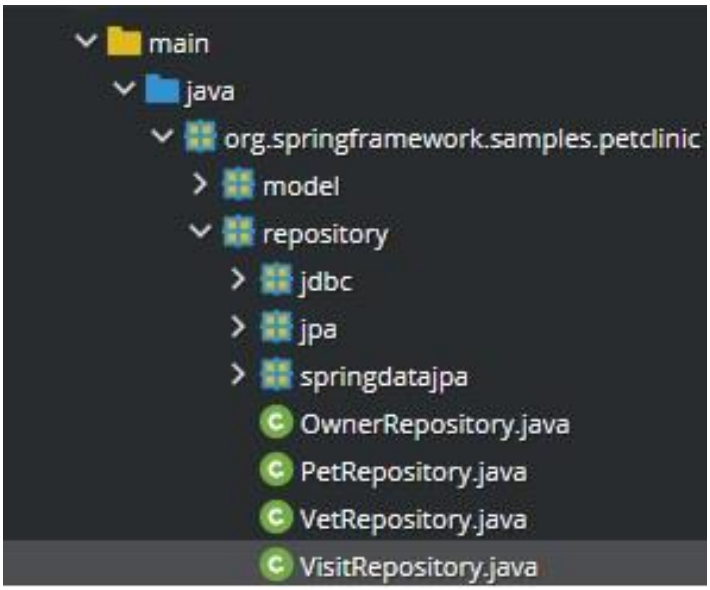
Rappel de la notion d'interface : une interface contient une liste de signatures de méthodes sans l'implémentation (sans le code) de celles-ci.

Lorsque la couche service utilise par exemple des objets de la couche DAO, elle va utiliser les méthodes des interfaces fournies par la couche DAO sans avoir à se préoccuper de la manière dont sont implémentés les méthodes de ces interfaces.

Ce procédé permet aussi de proposer plusieurs implémentations possibles. Ceci est le cas par exemple pour la couche DAO :

Chaque interface de Repository propose 3 implémentations différentes :

- Implémentation à l'aide de l'API **JDBC**
- Implémentation à l'aide de l'API **JPA**, qui est le standard utilisé en développement JEE pour gérer la persistance
- Implémentation spécifique à Spring avec les **Spring Data** pour simplifier l'écriture avec des annotations spécifiques au framework Spring.



Regardons par exemple le contenu de l'interface OwnerRepository.java :

```
OwnerRepository ×
47
48  public interface OwnerRepository {
49
50      /**
51       * Retrieve <code>Owner</code>s from the data store by last name, returning all own
52       * with the given name.
53       *
54       * @param lastName Value to search for
55       * @return a <code>Collection</code> of matching <code>Owner</code>s (or an empty <
56       *         found)
57       */
58      Collection<Owner> findByLastName(String lastName) throws DataAccessException;
59
60      /**
61       * Retrieve an <code>Owner</code> from the data store by id.
62       *
63       * @param id the id to search for
64       * @return the <code>Owner</code> if found
65       * @throws org.springframework.dao.DataRetrievalFailureException
66       *           if not found
67       */
68      Owner findById(int id) throws DataAccessException;
69
70
71      /**
72       * Save an <code>Owner</code> to the data store, either inserting or updating it.
73       *
74       * @param owner the <code>Owner</code> to save
75       * @see BaseEntity#isNew
76       */
77      void save(Owner owner) throws DataAccessException;
```

Cette interface contient 3 méthodes :

- La méthode findByLastName() qui va permettre de récupérer une liste d'objets Owner correspondant à un nom passé en paramètre.

- La méthode findById() qui va permettre de récupérer un objet Owner correspondant à un id de Owner passé en paramètre.
- La méthode save() qui va permettre de sauvegarder un objet Owner en base de données

2.2 Les 3 implémentations possible des Repository – IoC – Injection de dépendance

Aller observer les 3 implémentations possibles de cette interface :

- **Jdbc**OwnerRepositoryImpl.java dans le package **jdbc**
- **Jpa**OwnerRepositoryImpl.java dans le package **jpa**
- **SpringData**OwnerRepositoryImpl.java dans le package **springdatajpa**

2.2.1 Les éléments importants à retenir :

Dans les 3 implémentations, l'annotation **@Repository** est placée au-dessus de la déclaration de la classe.

```
@Repository
public class JpaOwnerRepositoryImpl implements OwnerRepository {
```

2.2.2 Implémentation JDBC

Dans l'implémentation **JDBC**, aller observer les **requêtes SQL** utilisant les noms de tables et de de colonnes de la base de données.

```
public Owner findById(int id) throws DataAccessException {
    Owner owner;
    try {
        Map<String, Object> params = new HashMap<String, Object>();
        params.put("id", id);
        owner = this.namedParameterJdbcTemplate.queryForObject(
                "SELECT id, first_name, last_name, address, city, telephone FROM owners WHERE id= :id",
                params,
```

2.2.3 Implémentation JPA

Dans l'implémentation **JPA**, aller observer les requêtes écrites en **langage JPQL** qui utilisent des noms de classes, d'objets et d'attributs.

```
public Owner findById(int id) {
    // using 'join fetch' because a single query should load both owners and pets
    // using 'left join fetch' because it might happen that an owner does not have pets yet
    Query query = this.em.createQuery("SELECT owner FROM Owner owner left join fetch owner.pets WHERE owner.id =:id");
    query.setParameter("id", id);
    return (Owner) query.getSingleResult();
}
```

On remarque l'utilisation de requêtes préparées.

Pour la méthode **save()**, les méthodes **merge()** et **persist()** sont utilisées :

```
public void save(Owner owner) {
	if (owner.getId() == null) {
		this.em.persist(owner);
	}
	else {
		this.em.merge(owner);
	}

}
```

La méthode **merge()** permet de mettre à jour les données de l'objet owner passé en paramètres si il est déjà existant en base.

La méthode **persist()** permet de rendre l'objet owner persistant en base si il n'existe pas encore dans la base.

L'implémentation JPA utilise un attribut de type **EntityManager** précédé de l'annotation **@PersistenceContext :**

```
@Repository
public class JpaOwnerRepositoryImpl implements OwnerRepository {

    @PersistenceContext
    private EntityManager em;
```

IMPORTANT : la notion d'injection de dépendance et d'inversion de contrôle et rôle de **l'EntityManager**

L'objet **em** est le manager d'entités du modèle. Il permet de gérer la persistance des objets du Model en base de données. Cet objet va être auto-injecté automatiquement (créé par le conteneur de l'application).

Ceci permet d'utiliser **l'inversion de contrôle (IoC),** encore appelée **injection de dépendance** : plutôt que ça soit la classe de Repository qui crée l'objet EntityManager en appelant le constructeur de la classe, c'est le conteneur de l'application qui crée l'objet (ici l'EntityManager). Ainsi, il n'y a pas à se soucier de la manière dont la classe EntityManager est implémentée.

Ceci permet de garantir l'indépendance des composants développés. Lorsqu'une classe dépend d'une autre classe (c'est-à-dire utilise un objet d'une autre classe comme ici l'EntityManager), cette dépendance (l'objet) est injecté automatiquement.

2.2.4 Implémentation SpringData

Dans l'implémentation avec les **SpringData**, on remarque que la **requête JPQL** est stockée dans une annotation spécifique : **l'annotation @Query.**

```
@Override
@Query("SELECT owner FROM Owner owner left join fetch owner.pets WHERE owner.id =:id")
public Owner findById(@Param("id") int id);
```

Dans ce TP, nous allons gérer seulement l'implémentation JPA.

2.3 Le Repository MemoRepository et son implémentation JPA

Dans le package **repository**, créer l'interface **MemoRepository.java**

Ajouter le code de cette interface ci-dessous :

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

import org.springframework.dao.DataAccessException;
import org.springframework.samples.petclinic.model.BaseEntity;
import org.springframework.samples.petclinic.model.Memo;

import java.util.List;

/**
 * Repository class for <code>Memo</code> domain objects
 */
public interface MemoRepository {

    /**
     * Save a <code>Memo</code> to the data store, either inserting or updating it.
     *
     * @param memo the <code>Memo</code> to save
     * @see BaseEntity#isNew
     */
    void save(Memo memo) throws DataAccessException;

    /**
     * Retrieves a list of <code>Memo</code> associated to a vet
     *
     * @param vetId the <code>Vet</code> identifier
     * @return List of of <code>Memo</code>
     */
    List<Memo> findByVetId(Integer vetId);
}
```

Dans le package **jpa**, créer la classe d'implémentation JPA de l'interface MemoRepository : **JpaMemoRepositoryImpl.java** puis ajouter le code ci-dessous :

```
package org.springframework.samples.petclinic.repository.jpa;

import org.springframework.samples.petclinic.model.Memo;
import org.springframework.samples.petclinic.repository.MemoRepository;
import org.springframework.stereotype.Repository;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import java.util.List;

/**
 * JPA implementation of the {@link MemoRepository} interface.
```

```
 */
@Repository
public class JpaMemoRepositoryImpl implements MemoRepository {

    @PersistenceContext
    private EntityManager em;


    @Override
    public void save(final Memo memo) {
        if (memo.getId() == null) {
            this.em.persist(memo);
        } else {
            this.em.merge(memo);
        }
    }

    @Override
    @SuppressWarnings("unchecked")
    public List<Memo> findByVetId(final Integer vetId) {
        Query query = this.em.createQuery("SELECT m FROM Memo m where m.vet.id=
:id");
        query.setParameter("id", vetId);
        return query.getResultList();
    }

}
```

Observer le contenu de ces 2 fichiers.

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)

Nous pourrons tester ce code plus tard dans le TP, lorsque nous écrirons les tests unitaires de la couche service.

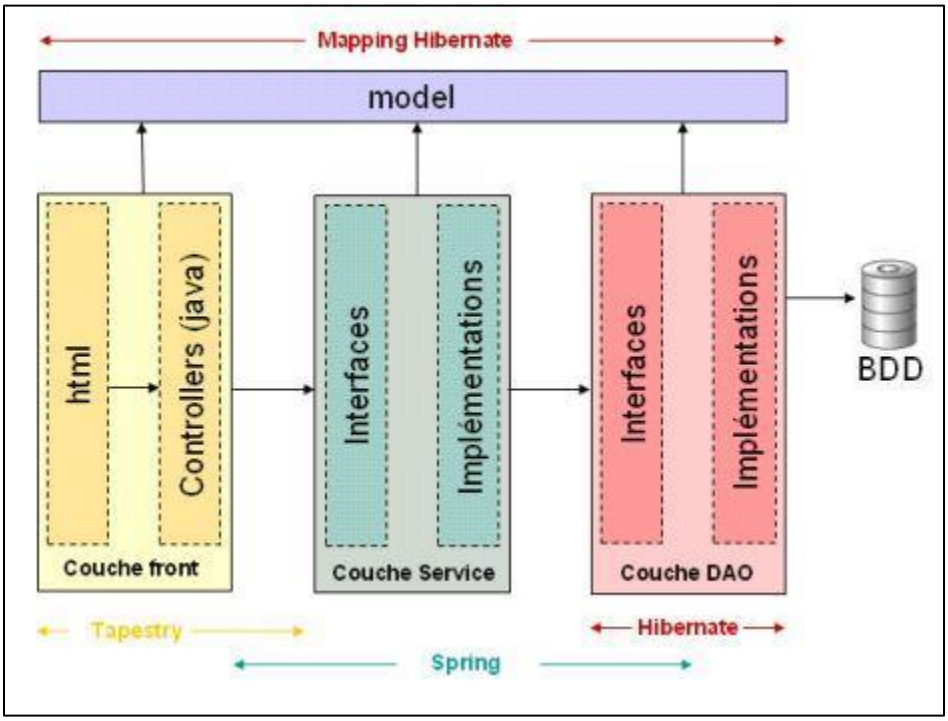
2.4 Le Repository OperationRepository et son implémentation JPA

2.4.1 Exercice 1 (à rendre)

Ajouter le Repository **OperationRepository** ainsi que la classe d'implémentation JPA : **JpaOperationRepositoryImpl**.

L'interface doit déclarer 3 méthodes :
- Enregistrer l'entité Operation
- Lister les opérations associées à un vétérinaire
- Lister les opérations associées à un animal

3 La couche Service



La couche service permet de lister l'ensemble des services métier mis à disposition par notre application, c'est-à-dire l'ensemble des méthodes qui pourront être invoquées depuis l'extérieur.

Dans notre cas, ces méthodes seront appelées par la couche front mais elles pourraient l'être depuis une autre application.

3.1 L'interface de la couche service

Comme pour la couche DAO, cette couche met en œuvre le principe d'abstraction en utilisant une interface, de telle sorte que la couche front n'aura pas à se soucier de la manière dont est implémentée cette interface.

La couche front va juste appeler les méthodes fournies par l'interface de la couche Service.

La couche Service contient la logique métier avec tous les traitements nécessaires.

Elle permet aussi de séparer la couche Contrôleur de la couche d'accès aux données.

Elle utilise également les objets du modèle, les entities.

Observons l'interface ClinicService.java (dans **src/main/java/org/springframework/samples/petclinic /service**) :

```java
public interface ClinicService {

    Collection<PetType> findPetTypes() throws DataAccessException;

    Owner findOwnerById(int id) throws DataAccessException;

    Pet findPetById(int id) throws DataAccessException;

    void savePet(Pet pet) throws DataAccessException;

    void saveVisit(Visit visit) throws DataAccessException;

    Collection<Vet> findVets() throws DataAccessException;

    void saveOwner(Owner owner) throws DataAccessException;

    Collection<Owner> findOwnerByLastName(String lastName) throws DataAccessException;

}
```

Cette interface liste l'ensemble des services fournis par l'application Pet Clinic.

Cette application offre la possibilité de :

- Lister les types d'animaux de la clinique
- Trouver un animal à partir de son id
- Enregistrer un animal
- Enregistrer une visite
- Lister tous les vétérinaires
- Enregistrer un propriétaire
- Trouver des propriétaires à partir d'un nom

3.2 L'implémentation de la couche service

Regardons maintenant dans la classe **ClinicServiceImpl** comment sont implémentées les méthodes de l'interface **ClinicService** :

```java
@Service
public class ClinicServiceImpl implements ClinicService {

    private PetRepository petRepository;
    private VetRepository vetRepository;
    private OwnerRepository ownerRepository;
    private VisitRepository visitRepository;

    @Autowired
    public ClinicServiceImpl(PetRepository petRepository,
    VetRepository vetRepository, OwnerRepository ownerRepository,
    VisitRepository visitRepository) {
        this.petRepository = petRepository;
        this.vetRepository = vetRepository;
        this.ownerRepository = ownerRepository;
        this.visitRepository = visitRepository;
    }

    @Override
    @Transactional(readOnly = true)
    public Collection<PetType> findPetTypes() throws DataAccessException {
        return petRepository.findPetTypes();
    }
```

L'annotation **@Service** est placée au-dessus de la définition de la classe.

On remarque que la classe contient des attributs permettant de stocker les différents objets de Repository : **petRepository, vetRepository, ownerRepository** et **visitRepository.**

Ceci va permettre à la couche service d'utiliser les Repository et d'appeler leurs méthodes.

Les attributs sont du type des interfaces de Repository. Ainsi, le développeur n'a pas à se soucier de la manière dont celles-ci sont implémentées.

L'annotation **@Autowired** placée au-dessus du constructeur signifie que les objets **petRepository, vetRepository, ownerRepository** et **visitRepository** (passés en paramètres) vont être auto-injectés par le conteneur web.

Ici aussi, le principe IoC (d'inversion de contrôle) / injection de dépendance va être mis en œuvre.

La méthode **findPetTypes()** est précédée de l'annotation **@Transactionnal (readOnly=true).**

Elle indique que le mode transactionnel est utilisé mais en lecture seulement (pas de mises à jour de la base de données).

On voit que la méthode **findPetTypes()** du repository **petRepository** est appelée.

La couche service se base donc sur la couche repository.

Observons maintenant l'implémentation de la méthode **saveVisit()** :

```
@Override
@Transactional
public void saveVisit(Visit visit) throws DataAccessException {
    visitRepository.save(visit);
}
```

L'annotation **@Transactionnal** est utilisée (sans **readOnly= true** cette fois-ci puisqu'il y aura une mise à jour de la base de données).

Cette méthode appelle la méthode **save()** du repository **visitRepository** en lui passant en paramètre l'objet du Model (objet **visit**) à sauvegarder en base.

3.3 La gestion des memos et des opérations dans la couche service

Ajouter la méthode suivante dans l'interface **ClinicService** :

```
void saveMemo(Memo memo);
```

Ajouter son implémentation dans la classe **ClinicServiceImpl** :
Ajouter l'attribut **memoRepository** :

```
private MemoRepository memoRepository;
```

Modifier le constructeur :

```
@Autowired
    public ClinicServiceImpl(PetRepository petRepository, VetRepository
vetRepository, OwnerRepository ownerRepository, VisitRepository visitRepository,
MemoRepository memoRepository) {
        this.petRepository = petRepository;
        this.vetRepository = vetRepository;
        this.ownerRepository = ownerRepository;
        this.visitRepository = visitRepository;
        this.memoRepository=memoRepository;
    }
```

Ajouter la méthode **saveMemo**() :

```
@Override
@Transactional
    public void saveMemo(Memo memo) {
        memoRepository.save(memo);
}
```

3.3.1 Exercice 2 (à rendre)

Ajouter la méthode **saveOperation**() dans l'interface **ClinicService** et faites les ajouts nécessaires dans **ClinicServiceImpl**.

3.4 Les tests unitaires de la couche service

Pour gérer les tests unitaires de la couche service, la classe abstraite AbstractClinicServiceTests.java est utilisée (dans **src/test/java/org/springframework/samples/petclinic/service**).

Les 3 classes **ClinicServiceJdbcTests**, **ClinicServiceJpaTests** et **ClinicServiceSpringDataTests** héritent de cette classe abstraite.

Ceci permet d'exécuter les tests unitaires de la couche service avec les 3 implémentations possibles des Repository.

Nous allons juste tester l'implémentation **JPA**.

Exécuter les tests unitaires de la classe **ClinicServiceJpaTests** (clic droit sur la classe puis « Run Test », « Run Junit Test »).

3.4.1 Exercice 3 (à rendre)

En vous inspirant du test « **shouldAddNewVisitForPet** » (dans **AbstractClinicServiceTests**.java) ajouter les test **shouldAddNewMemoForVet**() et **shouldAddNewOperationForPetAndVet**() dans **AbstractClinicServiceTests**.java et vérifier que les tests unitaires fonctionnent.