

# **Application Web Dynamique la persistance des données**



**Daniel Hagimont**

**<https://www.google.fr/search?q=daniel+hagimont+home+page>**

1

Nous attaquons la partie persistance du cours.

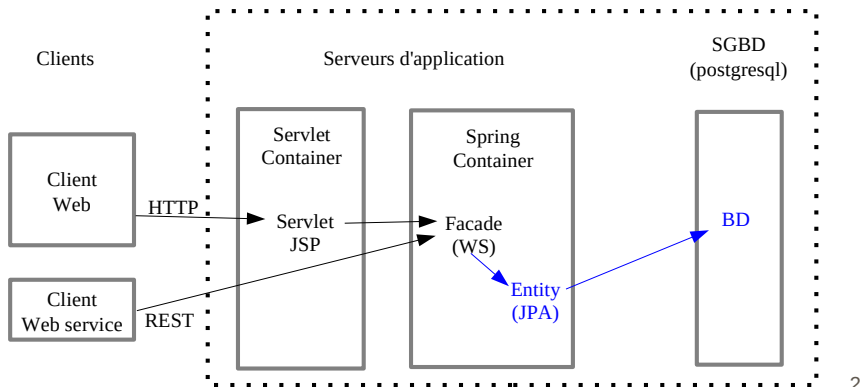
Le but est de simplifier le stockage des données dans une base de données (persistance).

Dans l'exemple des comptes bancaires, cela revient à supprimer la table stockant les comptes dans la Facade, et à gérer ces comptes dans une base de données.

Les comptes seront stockés dans une base de données, bien qu'on les manipule comme des objets Java.

## Objectif

- Gérer facilement les données dans une BD relationnelle
- Notion d'ORM (Object Relational Mapping)
  - Permet de gérer des collections d'objets dans la BD
  - JPA (Java Persistence API) : instantiation dans le monde Java



Cette technologie permettant de gérer sous forme d'objets des données stockées dans une BD relationnelle est appelée ORM (Object Relational Mapping).

JPA est une instantiation dans le monde Java.

A noter que ce soit Spring-JPA ou les EJB, ils reposent tous les deux sur Hibernate.

## Entity beans (JPA)

- Représentent des objets dans la base de donnée
  - Une classe correspond à une table
  - Une instance correspond à une ligne dans cette table
  - Un champ correspond à une colonne de la table
- Programmés comme des POJO (Plain Old Java Object)
  - Pas d'héritage ou d'implantations spéciales
- Ce sont des Java Beans
  - Possèdent un constructeur sans arguments
  - Possèdent des setters / getters (convention de nommage)

3

La gestion des données repose sur les entity beans.

Les entity beans sont des classes Java qui représentent des objets dans la base de données.

Chaque entity (par exemple la classe Compte) correspond à une table dans la base de données.

Une instance (par exemple un compte) correspond à une ligne dans cette table.

Un champ (par exemple le champs nom de la classe Compte) correspond à une colonne de la table.

La programmation des entity beans est dite POJO (Plain Old Java Object) car la classe d'un entity bean est un objet Java très simple (pas d'héritage, d'implémentation ou d'exceptions à ajouter).

Mais ce sont des Java Beans :

- ils possèdent un constructeur sans argument (il y en a un par défaut si on ne définit pas de constructeur supplémentaire)
- ils possèdent des setters et getter (attention, la convention des setters/getters doit être respectée, le mieux est donc de les générer avec vscode)

## Entity beans

- La classe doit être annotée avec `@Entity`
- Un champ annoté avec `@Id` est la clé primaire qui sert d'identifiant unique dans la table
  - Cette clé primaire peut être un type primitif ou de type objet
- Les autres champs peuvent être associés aux colonnes de la table
- Les champs sont utilisables au travers des accesseurs (getters/setters)
- On peut décrire l'association
  - Entre le nom de la classe et le nom de la table
  - Entre le nom d'un champ et le nom d'une colonne
  - Mêmes noms par défaut
- La base de donnée peut être pré-existante ou générée

4

Pour qu'une simple classe Java (un Java Bean) deviennent un entity bean, il faut lui ajouter quelques annotations. Chaque annotation dans vscode nécessite un import.

La classe doit être annotée avec `@Entity` (voir exemple 2 slides plus loin). La classe correspond à une table dans la BD.

Un champ doit être annoté avec `@Id`. Cela permet de dire que ce champ est la clé primaire de l'entity bean, donc de la table dans la base de donnée. La clé primaire est le champ (ou la colonne dans la BD) qui identifie de façon unique une instance (ou la ligne dans la table). Cette clé primaire peut être complexe (de type objet), mais en général, on utilise un entier (ou un Long en Spring), on appelle ce champs "id" et on demande à la BD de générer ce numéro unique automatiquement.

Les autres champs (sans annotations) correspondent à des colonnes dans la table.

Tous les champs de l'entity bean doivent être utilisés avec les accesseurs (setters/getters). Ceci est important, car Spring instrumente ces accesseurs (rajoute du code dedans).

Par défaut, la table dans la BD aura le même nom que la classe, et chaque colonne dans la table aura le même nom que le champ dans l'entity bean. C'est le cas en particulier lorsque les tables de la BD n'existent pas et sont générées depuis les entity beans (c'est Spring/Hibernate qui crée ces tables dans la BD).

Cependant, il est possible de décrire ce que deviennent ces noms (classe et variables de l'entity) dans la BD (table et colonnes). Ceci est intéressant lorsque l'on réutilise une BD existante (donc quand on n'a pas le choix de ces noms dans les tables).

## Annotation de mapping entre le bean et la table

- **@Table**
  - Préciser le nom de la table associée à une classe. Par défaut, c'est le nom de la classe.
- **@Column**
  - Préciser le nom de la colonne associée à un champ. Par défaut, c'est le nom du champ.
- **@Id**
  - Déclarer un champ (donc une colonne) comme clé primaire de la table.
- **@GeneratedValue**
  - Demander la génération automatique de la clé primaire par la BD (tag utilisé avec @Id).
- **@Transient**
  - Demander à ne pas tenir compte d'un champ (par défaut tous les champs sont associés à des colonnes)

5

Voici les annotations pour décrire le nom des tables et colonnes dans la BD :

- `@Table(name="account")`

- `@Column (name="lastname")`

Mais si on n'utilise pas ces deux précédentes annotations, les noms des tables et colonnes seront les mêmes que les noms des classes et des champs.

`@Id` permet de définir la clé primaire. Juste en dessous, on peut mettre une annotation `@GeneratedValue` pour dire que la BD doit générer la clé primaire (un numéro unique). Un exemple est donné sur le slide suivant.

`@Transient` permet de dire qu'un champ de l'entity bean n'a pas d'existence dans la BD. Attention, par défaut, tous les champs ont une correspondance dans la BD.

## Entity Compte

```
@Entity
public class Compte {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int num;
    private String nom;
    private int solde;

    public Compte() {}

    public Compte(String nom, int solde) {    // num est généré
        this.nom = nom; this.solde = solde;
    }

    // constructors, setters and getters
}
```

6

Un simple entity bean Compte avec les annotations @Entity et @Id

On est obligé de définir un constructeur vide, car on a défini un constructeur avec des paramètres (sans ça, il y a un constructeur vide par défaut).

Notez que le constructeur avec des paramètres n'a pas besoin d'initialiser la clé primaire (id), car elle est générée (@GeneratedValue).

## Créer un Repository par entity

- Le repository permet de gérer la persistance (il fournit les méthodes CRUD (Create, Read, Update, Delete))

```
public interface CompteRepository extends JpaRepository<Compte, Long> {  
  
    // Compte save(Compte c) ;  
  
    // Optional<Compte> findById(Long id);  
  
    // Collection<Compte> findAll();  
  
    // void delete(Compte c) ;  
  
    // long count() ;  
  
    @Query("SELECT c FROM Compte c WHERE c.solde < :montant")  
    Collection<Compte> findLower(int montant);  
}
```

Méthodes héritées

7

En Spring, il faut créer une interface Repository par entity bean.

Cette interface définit les méthodes CRUD permettant de gérer les instances de l'entity bean.

Ici, le Repository de Compte hérite de JpaRepository. Long est le type de la clé primaire.

Des méthodes sont implicitement définies pour :

- sauver dans la BD une instance ou des modifications sur une instance (save)
- retrouver une instance à partir de sa clé primaire (findById). A noter que cette méthode retourne un Optional<Compte>, ce qui signifie que la valeur retournée peut être "empty".
- retrouver toutes les instances (findAll)
- détruire une instance (delete)
- récupérer le nombre d'instances (count)
- et il y en a d'autres

Notons que l'on peut définir de nouvelles méthodes, notamment à partir de requêtes à la BD (on peut utiliser un langage proche de SQL, JPQL (Java Persistence Query Language, ou bien écrire des native queries en SQL).

## Facade

```
@RestController
public class Facade {

    @Autowired
    CompteRepository cr;

    @PostMapping("/addcompte")
    public void addCompte(Compte c) {
        cr.save(c);
    }

    @GetMapping("/consultercomptes")
    public Collection<Compte> consulterComptes() {
        return cr.findAll();
    }

    @GetMapping("/consultercompte")
    public Compte consulterCompte(@RequestParam("num") Long num) throws RuntimeException {
        Optional<Compte> c = cr.findById(num);
        if (! c.isPresent()) throw new RuntimeException("Compte introuvable");
        return c.get();
    }
}
```

8

Avec l'entity `Compte` défini précédemment, on peut écrire la Facade comme ci-dessus.

Notons que la Hashtable a disparu, car les instances sont stockées dans la BD.

`@Autowired` est une injection de dépendence qui initialise la variable `cr` qui fournit le moyen de gérer les instances persistentes de `Compte`.

On voit que le Repository est utilisé pour :

- persister une instance : `cr.save(c)`
- retrouver toutes les instances : `cr.findAll()`
- retrouver une instance : `cr.findById(nim)`



## Facade

```
@PostMapping("/debit")
public void debit(@RequestParam("num") Long num, @RequestParam("montant") int montant)
    throws RuntimeException {

    Compte c = consulterCompte(num);
    if (c.getSolde() < montant) throw new RuntimeException("Solde insuffisant");
    c.setSolde(c.getSolde() - montant);
    cr.save(c);
}

@PostMapping("/credit")
public void credit(@RequestParam("num") Long num, @RequestParam("montant") int montant) {
    Compte c = consulterCompte(num);
    c.setSolde(c.getSolde() + montant);
    cr.save(c);
}

@PostConstruct
public void populate() {
    cr.save(new Compte(null, "dan", 2000));
    cr.save(new Compte(null, "alain", 4000));
    cr.save(new Compte(null, "luc", 6000));
}
```

9

A la fin de la Facade, pour pouvoir débbugger, on définit une méthode qui crée quelques instance dans la BD.

Cette méthode est annotée avec `@PostConstruct`, ce qui signifie qu'elle est appelée après initialisation de la Facade.

## Developpement

- Lorsque vous créez votre projet Spring sous vscode
  - Spring initializ : Create a Maven Project
    - Ajouter les dépendances
      - [Spring Data JPA](#)
      - [HyperSQL Database](#)
- Vous devez ajouter un fichier [application.properties](#) dans [src/main/resources](#)

```
spring.application.name=bank-spring-jpa
spring.datasource.url=jdbc:hsqldb:hsql://localhost/xd
spring.datasource.driverClassName=org.hsqldb.jdbc.JDBC
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.platform=hsqldb
spring.datasource.hikari.enabled=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.HSQLDialect
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.use_sql_comments=true
```

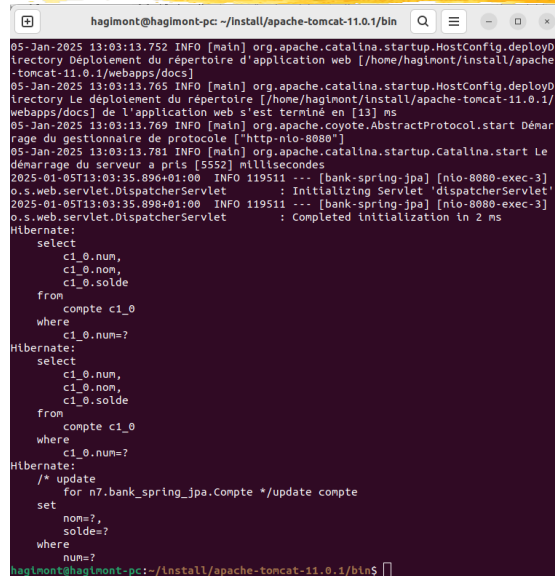
10

Pour pouvoir utiliser JPA dans votre projet Spring avec vscode, lorsque vous créez votre projet, il faut ajouter 2 dépendances :

- Spring Data JPA : ajouter JPA à votre projet
- HyperSQL Database : permet de se connecter à une BD HSQLDB

Il faut aussi ajouter un fichier `application.properties` dans le répertoire `src/main/resources`. Ce fichier indique les données de connexion au serveur de BD.

## Exécution avec JPA



```
hagimont@hagimont-pc: ~/install/apache-tomcat-11.0.1/bin
05-Jan-2025 13:03:13.752 INFO [main] org.apache.catalina.startup.HostConfig.deploy0
Directory Déploiement du répertoire d'application web [/home/hagimont/install/apache
-tomcat-11.0.1/webapps/docs]
05-Jan-2025 13:03:13.765 INFO [main] org.apache.catalina.startup.HostConfig.deploy0
Directory Le déploiement du répertoire [/home/hagimont/install/apache-tomcat-11.0.1/
webapps/docs] de l'application web s'est terminé en [13] ms
05-Jan-2025 13:03:13.769 INFO [main] org.apache.coyote.AbstractProtocol.start Démarr
age du gestionnaire de protocole ["http-nio-8080"]
05-Jan-2025 13:03:13.781 INFO [main] org.apache.catalina.startup.Catalina.start Le
démarrage du serveur a pris [5552] millisecondes
2025-01-05T13:03:35.896+01:00 INFO 119511 --- [bank-spring-jpa] [nio-8080-exec-3]
o.s.web.servlet.DispatcherServlet      : Initializing Servlet 'dispatcherServlet'
2025-01-05T13:03:35.898+01:00 INFO 119511 --- [bank-spring-jpa] [nio-8080-exec-3]
o.s.web.servlet.DispatcherServlet      : Completed initialization in 2 ms
Hibernate:
    select
      c1_0.num,
      c1_0.non,
      c1_0.solde
    from
      compte c1_0
    where
      c1_0.num=?
Hibernate:
    select
      c1_0.num,
      c1_0.non,
      c1_0.solde
    from
      compte c1_0
    where
      c1_0.num=?
Hibernate:
    /* update
      for n7.bank_spring_jpa.Compte */update compte
    set
      non=?,
      solde=?
    where
      num=?
hagimont@hagimont-pc:~/install/apache-tomcat-11.0.1/bin$
```

11

Dans la console, on peut observer que lorsqu'on exécute l'application, la création d'un entity ou sa modification se traduisent par des requêtes SQL automatiquement effectuées par le container Spring/Hibernate.

NB :

- si vous lancer votre service Spring avec `./mvnw spring-boot:run`, vous verrez ces traces dans la console
- si vous avez exporté votre service Spring comme un war dans Tomcat, il faut aller regarder dans les logs (catalina.out).

## Gérer les associations entre les Entity

- @OneToMany
- @ManyToOne
- @OneToOne
- @ManyToMany

12

Jusque là, nous sommes capables depuis la Facade de créer des entity beans qui sont gérés automatiquement dans la BD.

A noter que des instances d'entity bean contiennent des champs dont le format est fixe : des entiers, réels, chaînes de caractères. Cela correspond à ce que l'on trouve dans les BD : des colonnes qui sont de taille fixe. Les chaînes de caractère sont tronquées en fonction de la taille définie dans la BD.

On ne peut donc pas avoir une liste chaînée d'objets Java dans un entity : on ne saurait pas les stocker dans une BD relationnelle dont les colonnes sont de taille fixe.

La seule façon d'avoir une liste dans un entity est d'avoir une relation (ou association) avec un autre entity et cette relation peut être multiple. Et la BD saura comment implanter cette relation.

Par exemple, un entity Client peut faire référence à une liste d'entity Compte. Dans la BD, un compte contiendra l'index du client qui le référence. C'est la façon de gérer des listes dans les BD.

Une relation entre 2 entity peut être de type :

- OneToMany
- ManyToOne
- OneToOne
- ManyToMany

Nous allons voir ces différents cas et voir à quoi ils correspondent dans la BD.

## @OneToMany (unidirectionnel)

pk=primary key  
fk=foreign key

- Un Entity Client possède un champ Collection de références vers des instances d'un autre Entity Compte
- Un compte appartient à un seul client (sinon, ManyToMany)
  - La BD est sensée être configurée pour vérifier l'unicité

```
@Entity
public class Client {
    @Id
    @GeneratedValue(strategy=
        GenerationType.IDENTITY)
    Long id;
    String nom;

    @OneToMany
    Collection<Compte> comptes;
```

*Peut être implanté par une fk ou une table de jointure (avec unicité pk Compte)*

```
@Entity
public class Compte {
    @Id
    @GeneratedValue(strategy=
        GenerationType.IDENTITY)
    Long id;
    int solde;
```

```
public void ajoutCompte(Long cli_id, Compte c) {
    compte_rep.save(c);
    Client cli = client_rep.findById(cli_id).get();
    cli.getComptes().add(c);
    client_rep.save(cli);
}
```

13

On considère ici une relation OneToMany entre un entity Client et un entity Compte. Cela implique que l'entity Client inclut un champ de type Collection<Compte> contenant les pointeurs vers les entity Compte qu'il référence. Cette relation est unidirectionnelle, cela signifie que l'entity Compte n'inclut pas de champ pointant vers l'entity Client. Cependant, un entity Compte ne peut être pointé que par un seul entity Client, sinon ce serait une relation ManyToMany.

On voit dans le code de l'entity Client qu'il possède bien une Collection<Compte> (comptes) contenant les références vers les entity Compte. L'annotation @OneToMany indique à Hibernate qu'il faut gérer cette relation. Notons qu'il n'est pas nécessaire d'instancier une liste (Hibernate le fera). L'entity Compte ne contient pas de relation inverse (on est unidirectionnel). Un exemple de code de la Facade est donné avec la méthode ajoutCompte(). On rend le compte persistant avec la méthode save(), puis à partir de la clé primaire du client, on retrouve l'entity Client, et on lui ajoute (dans sa collection) la référence au compte. Et on sauve la modification.

La BD est gérée automatiquement. En général, une telle relation est implantée avec une foreign key (cli-fk) dans la table Compte. Par exemple, si un client cli référence 2 comptes c1 et c2, on aura dans la BD :

Client		Compte		
id	nom	id	solde	cli-fk
cli	durant	c1	10000	cli
		c2	5000	cli

La BD peut aussi gérer cette relation avec une table de jointure (cela dépend des BD), mais la pk-compte doit être unique pour que ce soit bien une relation OneToMany

Client		Compte		ClientCompte	
id	nom	id	solde	pk-client	pk-compte
cli	durant	c1	10000	cli	c1
		c2	5000	cli	c2

## @ManyToOne (unidirectionnel)

- Un Entity Compte possède un champ référence vers une instance d'un autre Entity Client

```
@Entity
public class Client {
    @Id
    @GeneratedValue(strategy=
        GenerationType.IDENTITY)

    Long id;
    String nom;
}
```

```
public void ajoutCompte(Long cli_id, Compte c) {
    Client cli = client_rep.findById(cli_id).get();
    c.setOwner(cli);
    compte_rep.save(c);
}
```

```
@Entity
public class Compte {
    @Id
    @GeneratedValue(strategy=
        GenerationType.IDENTITY)

    Long id;
    int solde;

    @ManyToOne
    Client owner;
}
```

*Implanté par une fk*

14

On considère ici une relation ManyToOne entre un entity Compte et un entity Client. C'est la relation inverse de la précédente. Cela implique que l'entity Compte inclut un champ de type Client contenant un pointeur vers l'entity Client qu'il référence. Cette relation est unidirectionnelle, cela signifie que l'entity Client n'inclut pas de Collection<Compte>.

On voit dans le code de l'entity Compte qu'il possède bien une référence vers un entity Client. L'annotation @ManyToOne indique qu'il faut gérer cette relation. Un exemple de code de la Facade est donné avec la méthode ajoutCompte(). A partir de la clé primaire du client, on retrouve l'entity Client, et on affecte dans le compte la référence au client. Puis on rend le compte persistant avec la méthode save().

La BD est gérée automatiquement. Une telle relation est implantée avec une foreign key (cli-fk) dans la table Compte. Par exemple, si 2 comptes c1 et c2 référencent un client cli, on aura dans la BD :

Client		Compte		
id	nom	id	solde	cli-fk
cli	durant	c1	10000	cli
		c2	5000	cli

## @OneToMany / @ManyToOne (bidirectionnel)

- Permet la navigation dans les 2 sens

```
@Entity
public class Client {

    @Id
    @GeneratedValue(strategy=
        GenerationType.IDENTITY)
    Long id;
    String nom;

    @OneToMany(mappedBy="owner")
    Collection<Compte> comptes;
```

```
@Entity
public class Compte {

    @Id
    @GeneratedValue(strategy=
        GenerationType.IDENTITY)
    Long id;
    int solde;

    @ManyToOne
    Client owner;
```

*Implanté par une fk*

15

Si on combine les deux slides précédents, on obtient une relation bidirectionnelle. Cela signifie que si un client cli référence 2 comptes c1 et c2, alors :

- cli aura dans sa collection les 2 références aux 2 comptes c1 et c2
- c1 et c2 auront leur champ owner pointant vers cli

Le paramètre mappedBy de l'annotation OneToMany indique l'association entre les 2 annotations (OneToMany et ManyToOne). Sans ce paramètre mappedBy, on aurait alors 2 relations unidirectionnelles qui permettent d'avoir :

- cli qui référence c1 et c2
- c1 et c2 qui référencent un autre client que cli

Le mappedBy est donc obligatoire pour avoir une relation bidirectionnelle.

## @OneToMany / @ManyToOne (bidirectionnel)

- Le mappedBy indique le champ correspondant à la relation dans l'autre Entity
- On dit que le coté opposé au mappedBy (Compte) est le porteur de la relation
- On peut le faire dans l'autre sens, mais ce serait inefficace
- La mise à jour
  - Doit être faite du coté du porteur de la relation (Compte)
  - Elle est propagée de l'autre coté par le container (dans la liste de l'Entity Client)

```
public void ajoutCompte(Long cli_id, Compte c) {  
    Client cli = client_rep.findById(cli_id).get();  
    c.setOwner(cli);  
    compte_rep.save(c);  
}
```

16

Comme dit précédemment, le mappedBy indique l'association entre les 2 annotations dans le cas d'une relation bidirectionnelle.

Le coté opposé au mappedBy (le champ owner dans l'exemple précédent) est appelé le porteur de la relation. Lorsqu'on veut faire une mise à jour de la relation (par exemple ajouter un compte au client), on doit réaliser la mise à jour du coté du porteur de la relation bidirectionnelle. Donc ici, on affecte le client au compte avec setOwner(). Cette modification est propagée automatiquement de l'autre coté de la relation (dans la liste de l'entity Client).



## @OneToOne

### ■ Implantation

- B1 doit pointer sur un B2 qui n'est pointé que par B1
- Deux implantations possible : fk dans B1 ou B2
- La BD est sensée être configurée pour vérifier l'unicité de la fk

### ■ Exemple

- Le schéma ci-dessous est utilisé pour bidirectionnel (unicité fk coté Compte)

```
// Client
@OneToOne(mappedBy="owner")
Compte compte;
```

```
// Compte
@OneToOne
Client owner;
```

```
public void changerCompte(Long cli_id, Compte c) {
    Client cli = client_rep.findById(cli_id).get();
    Compte old = cli.getCompte();
    if (old != null) {
        old.setOwner(null);
        compte_rep.save(old);
    }
    c.setOwner(cli);
    compte_rep.save(c);
}
```

17

Une relation OneToOne permet à un entity B1 de pointer vers un entity B2 qui ne peut être pointé que par B1 (sinon, c'est du ManyToOne).

Cette relation peut être implantée avec une foreign key dans B1 ou B2 (cela dépend de la BD). Pour implanter exactement du OneToOne, la BD doit être configurée pour vérifier que la foreign key est unique.

Cette relation peut être unidirectionnelle ou bidirectionnelle (il faut alors respecter les règles du mappedBy énoncées précédemment).

Dans l'exemple donné ici, on a une OneToOne bidirectionnelle et on affecte bien du coté du porteur de la relation.

## @ManyToMany

### ■ Unidirectionnel

- Comme un @OneToMany unidirectionnel (mais pas unicité du référençant)
- Forcément une table de jointure
- @ManyToMany des 2 cotés sans mappedBy correspond a 2 relations indépendantes (2 tables de jointure)

### ■ Bidirectionnel

- @ManyToMany des deux cotés
- Un porteur de l'association avec mappedBy
- Une mise à jour est effectuée du côté du porteur et propagée à l'autre côté

18

Une relation ManyToMany, c'est quand un entity B1 référence une collection d'entity B2, qui peuvent être référencés depuis plusieurs entity.

Notons qu'une ManyToMany en unidirectionnel, c'est comme une OneToMany en unidirectionnel, mais sans la contrainte que l'entity pointé ait un unique référençant.

Dans la BD, une ManyToMany est forcément implantée avec une table de jointure.

Client		Compte		ClientCompte	
id	nom	id	solde	pk-client	pk-compte
cli1	durant	c1	10000	cli1	c1
cli2	dupont	c2	5000	cli1	c2
				cli2	c2

On voit dans l'exemple ci-dessus que le client cli1 a 2 comptes (c1 et c2) et que le compte c2 a 2 clients (cli1 et cli2).

Si on gère une ManyToMany en bidirectionnel, on applique la même règle du mappedBy que précédemment. Il y a un porteur de la relation et l'affectation doit se faire du côté du porteur (et elle est propagée de l'autre côté) .

## @ManyToMany

```
// Client
@ManyToMany
Collection<Compte> comptes;
```

```
// Compte
@ManyToMany(mappedBy="comptes")
Collection<Client> owners;
```

```
public void partagerCompte(Long cli_id, Long co_id)
{
    Client cli = client_rep.findById(cli_id).get();
    Compte co = compte_rep.findById(co_id).get() ;
    cli.getComptes().add(co);
    client_rep.save(cli) ;
}
```

- Le compte apparaît dans la liste *comptes* du client
- Le client apparaît dans la liste *owners* du compte

19

Dans cet exemple, on a une ManyToMany bidirectionnelle entre les entity Client et Compte. Le porteur de la relation est l'entity Client, donc la mise à jour doit être faite de ce côté là. La méthode de la Facade partagerCompte() réalise bien l'affectation du côté de l'entity Client.

Suite à cette mise à jour, les 2 listes de chaque côté sont mises à jour (et la table de jointure dans la BD est aussi mise à jour).

## Cascades / Fetch

- JPA définit les opérations sur les entity :
  - PERSIST : quand un objet est rendu persistant
  - MERGE : quand un objet est mis à jour
  - REMOVE : quand un objet est détruit
  - ...
- On peut contrôler la propagation de ces opérations sur les Entity référencés

```
// Client
@OneToMany(cascade=CascadeType.REMOVE)
Collection<Compte> comptes;
```

20

JPA définit les opérations suivantes sur les entity :

PERSIST : quand un entity est rendu persistant dans la BD avec un appel à save().

MERGE : quand un objet est mis à jour avec un appel à save()

REMOVE : quand un objet est détruit, par exemple avec un appel à delete().

Lorsqu'on a une relation entre 2 entity, on peut spécifier qu'une telle opération doit être propagée (cascade) en suivant la relation. Sur l'exemple donné ici, il y a une relation OneToMany entre le client et les comptes. La destruction du client sera propagée aux comptes qui seront également détruits.

## Cascades / Fetch

- Que se passe t-il lorsqu'un Entity est sérialisé (sorti du container), pour ses références aux autres Entity

```
// Client
@OneToMany(mappedBy="owner")
Collection<Compte> comptes;
```

- Par défaut les objets sont chargés à la demande depuis la BD
  - Donc si on retourne un client à une JSP, la liste des comptes sera vide ! (si on n'y a pas accédé avant)

- On peut contrôler le chargement des Entity

```
// Client
@OneToMany(mappedBy="owner", fetch=FetchType.EAGER)
Collection<Compte> comptes;
```

- Un client sérialisé aura une liste cohérente de comptes
- Par défaut FetchType.LAZY (chargement paresseux lors de l'utilisation)

21

Un autre type de propagation est très utilisée dans le cas suivant. Supposons qu'il y a une relation OneToMany entre Client et Compte et supposons qu'une méthode de la Facade retourne un entity client, qui est sérialisé et retourné à la servlet. La servlet passe ensuite cet objet client à une JSP pour affichage. Imaginons que la JSP affiche le client et sa liste des comptes.

Le problème est que les entity Compte dans cette liste sont chargés à la demande dans le container Spring (depuis la BD). Si ils n'ont pas été accédés dans le container Spring, ils n'ont pas été chargés et la liste des comptes dans l'objet client est vide. En spécifiant le paramètre fetch=EAGER pour l'annotation, cela implique que si un entity client est chargé depuis la BD, alors les comptes de ce client seront chargés en même temps. Ainsi, si on sort un entity client du container (par exemple pour l'envoyer à la servlet et à la JSP), il aura une liste de comptes cohérente.

NB : on observe souvent une exception dans la JSP en raison du fait que les entity référencés par une relation n'ont pas été chargés (parce que ce fetch=EAGER) a été oublié.

## Attention – erreurs fréquentes

- On ne gère pas (dans des entity) des collections d'objets sans que ce soit des références à des entity (annotées)
- Il faut toujours utiliser les setter/getter pour affecter des champs (notamment pour le maintien des relations bidirectionnelles)
- Il faut faire attention au FetchType.EAGER lorsqu'on veut envoyer une collection d'objets vers une JSP

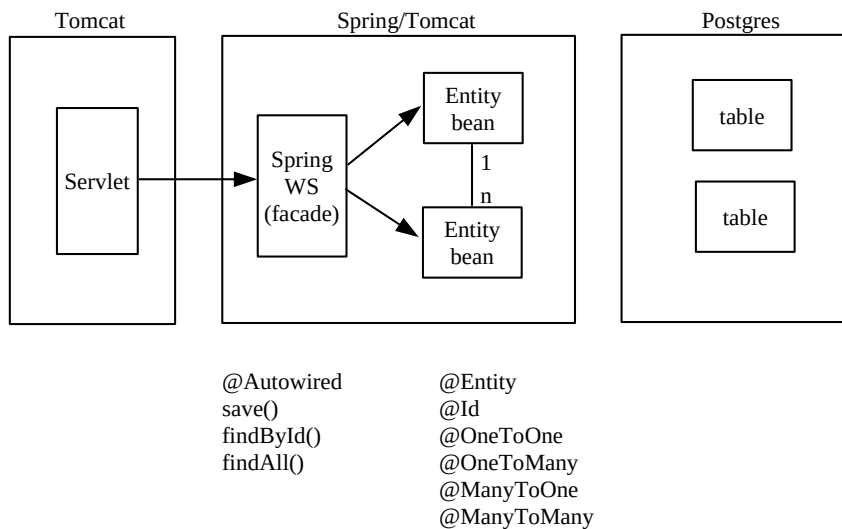
22

On ne peut pas gérer des collections (dont la taille est variable) dans un entity, car la BD ne saurait pas comment les stocker dans une table. Les collections dans des entity pointent vers d'autres entity et ça la BD sait le gérer (ce sont des relation entre des tables).

Il faut toujours utiliser les setters/getters des entity pour modifier leurs champs. Ces setters/getters ont pu être instrumentés (code injecté) par Spring. Notamment, pour les relations bidirectionnelles, la mise à jour des deux cotés est assurée par l'utilisation des setters/getters.

Lorsqu'un entity A fait référence à un (ou une collection de) entity B et lorsqu'on envoie une référence à A à une JSP, si on n'a pas paramétré la relation entre A et B avec FetchType.EAGER, seul l'entity A sera envoyé à la JSP (et pas l'entity B), ou du moins cela dépend de si B a été accédé avant (c'est aléatoire).

## L'essentiel

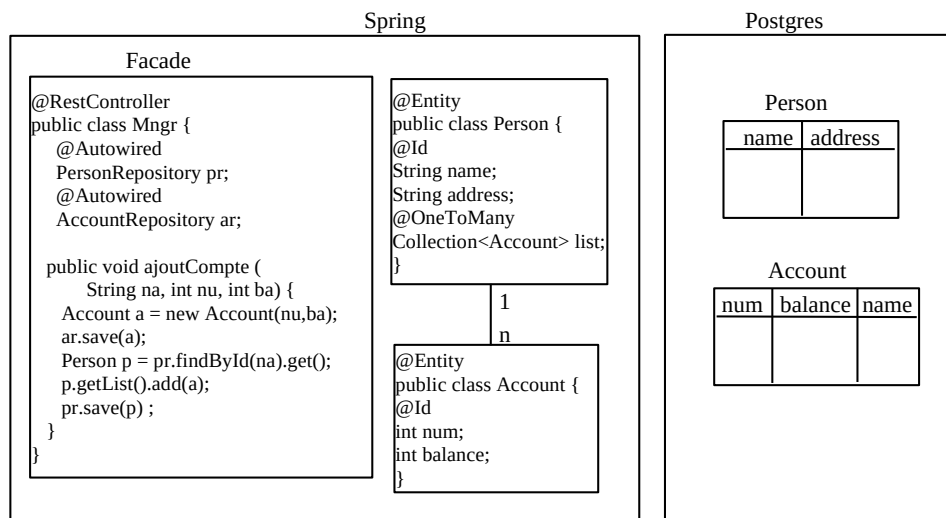


23

Voici un schéma qui résume les principales annotations et interfaces qui peuvent être utilisées.

- 1) la servlet peut faire appel à la Facade en appelant son API REST.
- 2) La définition de la Facade (Spring WS) peut définir des Repository pour les Entity
- 3) la Facade peut créer une instance d'entity et la rendre persistante (avec `save()`), retrouver une instance d'entity à partir de sa clé primaire (avec `findById()`) ou retrouver une collection d'instance d'entity (avec `findAll()`).
- 4) on peut définir des entity (avec `@Entity`) qui ont une clé primaire (avec `@Id`) et qui ont des relations avec les autres entity (avec `@OneToOne` par exemple). Ces entity sont implicitement gérés dans la base de donnée, bien qu'on les manipule sous la forme d'objets Java très simples.

## L'essentiel



24

Un dernier schéma qui synthétise un petit exemple.

On a deux entity Person et Account. Person a pour clé primaire name. Cette clé primaire n'est pas générée par la BD (pas de @GeneratedValue). Person contient une relation qui est une collection de Account (@OneToMany unidirectionnelle). Account a pour clé primaire num (pas généré par la BD également).

Dans la Facade, la méthode ajoutCompte() prend en paramètre le nom (na) de la personne à qui on ajoute ce compte (le nom est la clé primaire pour Person), le numéro du compte à créer (aussi la clé primaire pour Compte), et la balance pour le compte. Pour ajouter le compte, on instancie un objet Compte, on le rend persistant (save()). Puis on recherche la personne de clé primaire na (findById()) et enfin on ajoute le compte dans la liste de la personne.

On peut voir dans la BD l'implantation de la relation OneToMany avec une foreign key dans la table Account.



## Conclusion



- JPA facilite la gestion des données dans la BD
- JPA offre aussi des services pour
  - Les transactions
  - La sécurité
  - Le synchronisation

25

JPA offre également de nombreux services pas détaillés ici, notamment :

- les transactions : on peut annoter des méthodes avec `@Transactional`. Ce n'est alors plus la peine de faire des `save()`. Toutes les modifications sont journalisées, puis reportées sur la BD lorsque (et seulement si) la transaction est validée.
- la sécurité : il est possible de définir sans effort une politique de contrôle d'accès aux méthodes de la Facade.
- la synchronisation