CABANNES Jordy Lundi 7 décembre 2015

Bases de données II

Rapport TP II: Interfaçage de base de données avec Hibernate

Enseignant : Mr ADDA Mehdi

Sommaire

Partie I : Associations

1) Explication et justification de la raison d'être et la pertinence de chaque entité crée

Les entités suivantes : Adresse, Employé, PersonneContact, Entreprise, Projet, Telephone et Sevice ne sont pas de sentités issues d'associations. Elles ont chacune un identifiant qui est clé primaire au sein d'elle-même et non clé primaire dans une autre association comme c'est le cas par exemple pour les tables : localisation_service, telephone_personne_contact, telephone_employe, employe_affectation_projet, entreprise_faire_affaire.

Étant donné cette information, qui nous est confiée par la clé qui est jaune et non rouge dans le schéma de la base de données, Nous devons obligatoirement créer les entités : Adresse, Employé, PersonneContact, Entreprise, Projet, Telephone et Sevice qui sont les tables fondamentales de la base de données.

Néanmoins il reste d'après le schéma cinq tables à implémenter. Or dans le code Java nous constatons que certaines entités devant faire référence à ces tables n'ont pas été créées, c'est le cas pour les tables : localisation_service, telephone_personne_contact et telephone_employe. Ces tables sont issues d'association « Plusieurs-À-Plusieurs » entre :

- personne_contact et telephone pour telephone_personne_contact.
- telephone et employe pour telephone_employe.
- adresse et service pour localisation_service.

En fait ces tables ont été créées mais dans des entités de bases :

- telephone_personne_contact a été crée dans Telephone grâce au code suivant :

```
@ManyToMany
@JoinTable(name = "telephone_personne_contact",
            joinColumns        = { @JoinColumn(name = "telephone_id",
                                               referencedColumnName="id")
            },
            inverseJoinColumns = { @JoinColumn(name = "personne_conatct_id",
                                               referencedColumnName="id")
            })
private Collection<Personne_Contact> personne_contacts = new ArrayList<Personne_Contact>();
```

De plus il faut rajouter dans Personne_Contact le code suivant pour bien définie l'association :

```
@ManyToMany(mappedBy="personne_contacts", cascade=CascadeType.ALL)
private Collection<Telephone> telephones = new ArrayList<Telephone>();
```

Nous aurions bien évidemment pu créer la table « telephone_personne_contact» dans Personne_Contact, le résultat aurait été le même.

- telephone_employe a été créée dans Employe grâce au code suivant :

```
@ManyToMany
@JoinTable(name = "telephone_employe",
            joinColumns        = { @JoinColumn(name = "employe_id",
                                               referencedColumnName="id")
            },
            inverseJoinColumns = { @JoinColumn(name = "telephone_id",
                                               referencedColumnName="id")
            })
private Collection<Telephone> telephones = new ArrayList<Telephone>();
```

On rajoute le code suivant dans Telephone :

```
@ManyToMany(mappedBy="telephones", cascade=CascadeType.ALL)
private Collection<Employe> employes = new ArrayList<Employe>();
```

Nous aurions bien évidemment pu créer la table « telephone_employe» dans Telephone, le résultat aurait été le même.

- localisation_service a été créée dans Service grâce au code suivant :

```
@ManyToMany
@JoinTable(name = "localisation_service",
            joinColumns        = { @JoinColumn(name = "service_id",
                                               referencedColumnName="id")
            },
            inverseJoinColumns = { @JoinColumn(name = "adresse_id",
                                               referencedColumnName="id")
            })
private Collection<Adresse> adresses = new ArrayList<Adresse>();
```

On rajoute le code suivant dans Adresse :

```
@ManyToMany(mappedBy="adresses", cascade=CascadeType.ALL)
private Collection<Service> services = new ArrayList<Service>();
```

Nous aurions bien évidemment pu créer la table « localisation_service » dans Adresse, le résultat aurait été le même.

Ainsi il n'a pas été nécessaire de créer des entités pour avoir les trois tables précédentes dans la base de données.

Enfin il nous reste deux tables du schéma de la base de données, à savoir : employe_affectation_projet et entreprise_faire_affaire. Comme on peut le voir sur le schéma de la base de données, ces tables sont issues d'associations « Plusieurs-À-Plusieurs » étant donné la présence de la clé rouge (attribut clé primaire et étrangère). Cette fois-ci, ces deux tables contiennent des attributs en plus de leur clé, nous ne pouvons pas utiliser la même méthode que tout à l'heure.

Pour pouvoir avoir ces tables dans la base de données, j'ai créé une entité pour chacune de ces deux associations. Cependant comme on peut le voir dans le code suivant :

- d'une part pour employe_affectation_projet :

```
import java.io.Serializable;

@Entity
@Table(name="Employe_Affectation_Projet")
public class Employe_Affectation_Projet implements Serializable
{
    private static final long serialVersionUID = 1L;

        @Min(value=1)
        @EmbeddedId
        private Employe_Affectation_ProjetPK id;
```

- d'autre part pour entreprise_faire_affaire :

```
import java.io.Serializable;

@Entity
@Table(name="Entreprise_Faire_Affaire")
public class Entreprise_Faire_Affaire implements Serializable
{
    private static final long serialVersionUID = 1L;

    @Min(value=1)
    @EmbeddedId
    private Entreprise_Faire_AffairePK id;
```

J'ai utilisé l'annotation « @EmbeddedId ». J'ai créé une classe « Entreprise_Faire_AffairePK » et « Employe_Affectation_ProjetPK » qui ne sont pas des entités et qui contiennent les attributs des clés primaires de « Entreprise_Faire_Affaire » et « Employe_Affectation_Projet ». Ce sont des classes qui portent l'annotation « @Embeddable ».

```
import java.io.Serializable;

@Embeddable
public class Employe_Affectation_ProjetPK implements Serializable
{
    private static final long serialVersionUID = 1L;

    @Column(name="employe_id")
    private Integer employe_id;

    @Column(name="projet_numero")
    private Integer projet_numero;
```

```
import java.io.Serializable;

@Embeddable
public class Entreprise_Faire_AffairePK implements Serializable
{
    private static final long serialVersionUID = 1L;

    @Column(name="entreprise_id")
    private Integer entreprise_id;

    @Column(name="entreprise_partenaire_id")
    private Integer entreprise_partenaire_id;
```

Cela me permet d'avoir un seul attribut dans les entités « Employe_Affectation_Projet » et « Entreprise_Faire_Affaire » pour définir l'ID.

Enfin il faut relier ces deux tables aux entités de leurs associations « Plusieurs-À-Plusieurs » respectives grâce au code suivant :

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "entreprise_id", insertable = false, updatable = false)
private Entreprise entreprise;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "entreprise_partenaire_id", insertable = false, updatable = false)
private Entreprise entreprise_partenaire;
```

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "projet_numero", insertable = false, updatable = false)
private Projet projet;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "employe_id", insertable = false, updatable = false)
private Employe employe;
```

Pour finaliser ces associations on retrouve :

- Dans l'entité Projet :

```
@OneToMany(mappedBy="projet", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Employe_Affectation_Projet> employe_affectation_projet = new ArrayList<Employe_Affectation_Projet>();
```

- Dans l'entité Employe :

```
@OneToMany(mappedBy="employe")
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Employe_Affectation_Projet> employe_affectation_projet=new ArrayList<Employe_Affectation_Projet>();
```

- Dans l'entité Entreprise :

```
@OneToMany(mappedBy="entreprise", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Entreprise_Faire_Affaire> entreprise_faire_affaire=new ArrayList<Entreprise_Faire_Affaire>();

@OneToMany(mappedBy="entreprise_partenaire", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Entreprise_Faire_Affaire> entreprise_partenaire_faire_affaire = new ArrayList<Entreprise_Faire_Affaire>()
```

Ainsi toutes les tables de la base de données seront créées lors de l'exécution du programme. Nous avons pu voir qu'en fonction de la nature de l'existence d'une table (si elle est issue d'une association ou non) et des éléments qu'elle contient, il n'est pas toujours nécessaire de créer une entité.

Dans la partie qui suit nous allons voir comment les associations entre les différentes tables ont été implémentées.

2) Explication et justification de la raison d'être et la pertinence de chaque association

D'après le schéma de la base de données, nous pouvons voir qu'il y a des associations de différents types entre les tables. Nous allons commencer par traiter les associations de type « Un-À-Un », puis nous verrons les associations « Un-À-Plusieurs» et enfin nous reviendrons sur les associations « Plusieurs-À-Plusieurs ».

Association « Un-À-Un » :

Il y a des associations « Un-À-Un » entre les tables : « entreprise » et « employe », deux associations entre « employe » et « service ».

Pour l'association entre « entreprise » et « employe », l'id de « employe » fait référence à l'attribut « dg_id » dans la table « entreprise ». On doit donc implémenter l'association de la façon suivante :

- Dans l'entité Employe :

```
@OneToOne(mappedBy="employe")
private Entreprise entreprise;
```

- Dans l'entité Entreprise :

```
@OneToOne
@JoinColumn(name="dg_id")
private Employe employe;
```

« Entreprise » contient la colonne de jointure de l'association et on mappe « Employe » avec « Entreprise » via l'attribut « employe » définie dans « Entreprise ». On a ainsi implémenté l'association « Un-À-Un » entre « Entreprise » et « Employe ».

Pour la première association entre « service» et « employe », l'id de « employe » fait référence à l'attribut « directeur_id » dans la table « service». On doit donc implémenter l'association de la façon suivante :

- Dans l'entité Employe :

```
@OneToOne(mappedBy="employe1")
private Service service1;
```

- Dans l'entité Service:

```
@OneToOne
@JoinColumn(name="directeur_id")
private Employe employe1;
```

« Service» contient la colonne de jointure de l'association et on mappe « Employe » avec « Service» via l'attribut « employe1 » définie dans « Service». On a ainsi implémenté une des associations « Un-À-Un » entre « Service» et « Employe ».

Pour la seconde association entre « service» et « employe », l'id de « service» fait référence à l'attribut « service_id » dans la table « employe». On doit donc implémenter l'association de la façon suivante :

- Dans l'entité Employe :

```
@OneToOne
@JoinColumn(name="service_id")
private Service service2;
```

- Dans l'entité Service:

```
@OneToOne(mappedBy="service2")
private Employe employe2;
```

« Employe» contient la colonne de jointure de l'association et on mappe « Service» avec « Employe» via l'attribut « service2» définie dans « Employe». On a ainsi implémenté une des associations « Un-À-Un » entre « Service» et « Employe ».

Association « Un-À-Plusieurs » :

Il y a des associations « Un-À-Plusieurs » entre les tables : « adresse» et « personne_contact», « adresse» et « employe», « projet » et « service », « service » et « entreprise », « employe » et « employe ». Les autres associations « Un-À-Plusieurs » avec des traits pleins sont issues d'associations « Plusieurs-À-Plusieurs ».

Pour l'association entre « adresse» et « personne_contact», l'id de « adresse» fait référence à l'attribut « adresse_id » dans la table « personne_contact». Une instance de « personne_contact » ne peut avoir qu'une seule « adresse », et une instance de « adresse » peut être présente plusieurs fois dans l'ensemble des instances de « personne_contact ».

On implémente l'association de la façon suivante :

- Dans l'entité Adresse

```
@OneToMany(mappedBy="adresse", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Personne_Contact> listePersonnes = new ArrayList<Personne_Contact>();
```

- Dans l'entité Personne_Contact

```
@ManyToOne
@JoinColumn(name="adresse_id")
private Adresse adresse;
```

« Personne_Contact» contient la colonne de jointure de l'association et on mappe « Adresse» avec « Personne_Contact» via l'attribut « adresse» défini dans « Personne_Contact». L'entité « Adresse » contiendra une collection de « Personne_Contact » car chaque instance de « Adresse » est référencée par une ou plusieurs instances de « Personne_Contact ».

On a ainsi implémentée l'association « Un-À-Plusieurs » entre « Adresse» et « Personne_Contact».

Pour l'association entre « adresse» et « employe», l'id de « adresse» fait référence à l'attribut « adresse_id » dans la table « employe». Une instance de « employe» ne peut avoir qu'une seule « adresse », et une instance de « adresse » peut être présente plusieurs fois dans l'ensemble des instances de « employe».

On implémente l'association de la façon suivante :

- Dans l'entité Adresse

```
@OneToMany(mappedBy="adresse", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Employe> listeEmployes = new ArrayList<Employe>();
```

- Dans l'entité Employe

```
@ManyToOne
@JoinColumn(name="adresse_id")
private Adresse adresse;
```

« Employe» contient la colonne de jointure de l'association et on mappe « Adresse» avec « Employe » via l'attribut « adresse» défini dans « Employe». L'entité « Adresse » contiendra une collection de « Employe» car chaque instance de « Adresse » est référencée par une ou plusieurs instances de « Employe».

On a ainsi implémenté l'association « Un-À-Plusieurs » entre « Adresse» et « Employe».

Pour l'association entre « projet» et « service», l'id de « service» fait référence à l'attribut « service_id » dans la table « projet». Une instance de « projet» ne peut avoir qu'un seul « service», et une instance de « service» peut être présente plusieurs fois dans l'ensemble des instances de « projet».

On implémente l'association de la façon suivante :

- Dans l'entité Service

```
@OneToMany(mappedBy="service_id", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Projet> listeProjets = new ArrayList<Projet>();
```

- Dans l'entité Projet

```
@ManyToOne
@JoinColumn(name="service_id")
private Service service;
```

« Projet» contient la colonne de jointure de l'association et on mappe « Service» avec « Projet» via l'attribut « service» défini dans « Projet». L'entité « Service» contiendra une collection de « Projet» car chaque instance de « Service» est référencée par une ou plusieurs instances de « Projet».

On a ainsi implémenté l'association « Un-À-Plusieurs » entre « Service» et « Projet».

Pour l'association entre « service» et « entreprise», l'id de « entreprise» fait référence à l'attribut « entreprise_id » dans la table « service». Une instance de « service» ne peut avoir qu'une seule « entreprise», et une instance de « entreprise» peut être présente plusieurs fois dans l'ensemble des instances de « service».

On implémente l'association de la façon suivante :

- Dans l'entité Entreprise

```
@OneToMany(mappedBy="entreprise", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Service> services = new ArrayList<Service>();
```

- Dans l'entité Service

```
@ManyToOne
@JoinColumn(name="entreprise_id")
private Entreprise entreprise;
```

« Service» contient la colonne de jointure de l'association et on mappe « Entreprise» avec « Service» via l'attribut « entreprise» défini dans « Service». L'entité « Entreprise» contiendra une collection de « Service» car chaque instance de « Entreprise» est référencée par une ou plusieurs instances de « Service».

On a ainsi implémenté l'association « Un-À-Plusieurs » entre « Service» et « Entreprise».

Pour l'association entre « employe» et « employe», l'id de « employe» fait référence à l'attribut « superieur_id » dans la table « employe». Une instance de « employe» ne peut avoir qu'un seul « employe» superieur, et une instance de « employe»(superieur) peut être présente plusieurs fois dans l'ensemble des instances de « employe», car plusieurs employés ont le même supérieur.

On implémente l'association de la façon suivante :

- Dans l'entité Employe :

```
@OneToMany(mappedBy="employe", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Employe> listeEmployes = new ArrayList<Employe>();

@ManyToOne
@JoinColumn(name="superieur_id")
private Employe employe;
```

« Employe» contient la colonne de jointure de l'association et on mappe « Employe» avec « Employe» via l'attribut « employe» défini dans « Employe». L'entité « Employe» contiendra une collection de « Employe» car chaque instance de « Employe» (supérieur) est référencée par une ou plusieurs instances de « Employe».

On a ainsi implémenté l'association « Un-À-Plusieurs » entre « Employe» et « Employe».

Association « Plusieurs-À-Plusieurs » :

Il y a des associations « Plusieurs-À-Plusieurs » (traits pleins) entre les tables : « adresse » et « service », « personne_contact » et « telephone », « telephone » et « employe », « employe » et « projet », « entreprise » et « entreprise ».

On va distinguer les associations avec attributs et sans attributs.

Association sans attributs

Pour l'association entre « adresse » et « service », l'id de « service » et celui de « adresse » sont clés primaires dans « localisation_service ». Étant donné que cette association n'a pas d'attributs nous pouvons l'implémenter de la façon suivante.

- Dans l'entité Service :

```
@ManyToMany
@JoinTable(name = "localisation_service",
           joinColumns        = { @JoinColumn(name = "service_id",
                                              referencedColumnName="id")
           },
           inverseJoinColumns = { @JoinColumn(name = "adresse_id",
                                              referencedColumnName="id")
           })
private Collection<Adresse> adresses = new ArrayList<Adresse>();
```

- Dans l'entité Adresse :

```
@ManyToMany(mappedBy="adresses", cascade=CascadeType.ALL)
private Collection<Service> services = new ArrayList<Service>();
```

Nous aurions bien évidemment pu créer l'association « localisation_service» dans Adresse, le résultat aurait été le même.

Pour l'association entre « telephone» et « personne_contact», l'id de « telephone» et celui de « personne_contact» sont clés primaires dans « telephone_personne_contact ». Étant donné que cette association n'a pas d'attributs nous pouvons l'implémenter de la façon suivante.

- Dans l'entité Telephone:

```
@ManyToMany
@JoinTable(name = "telephone_personne_contact",
            joinColumns        = { @JoinColumn(name = "telephone_id",
                                               referencedColumnName="id")
            },
            inverseJoinColumns = { @JoinColumn(name = "personne_conatct_id",
                                               referencedColumnName="id")
            })
private Collection<Personne_Contact> personne_contacts = new ArrayList<Personne_Contact>();
```

- Dans l'entité Personne_Contact :

```
@ManyToMany(mappedBy="personne_contacts", cascade=CascadeType.ALL)
private Collection<Telephone> telephones = new ArrayList<Telephone>();
```

Nous aurions bien évidemment pu créer l'association « telephone_personne_contact» dans Personne_Contact, le résultat aurait été le même.

Pour l'association entre « telephone» et « employe», l'id de « telephone» et celui de « employe» sont clés primaires dans « telephone_employe ». Étant donné que cette association n'a pas d'attributs nous pouvons l'implémenter de la façon suivante.

- Dans l'entité Employe :

```
@ManyToMany
@JoinTable(name = "telephone_employe",
            joinColumns        = { @JoinColumn(name = "employe_id",
                                               referencedColumnName="id")
            },
            inverseJoinColumns = { @JoinColumn(name = "telephone_id",
                                               referencedColumnName="id")
            })
private Collection<Telephone> telephones = new ArrayList<Telephone>();
```

- Dans l'entité Telephone :

```
@ManyToMany(mappedBy="telephones", cascade=CascadeType.ALL)
private Collection<Employe> employes = new ArrayList<Employe>();
```

Nous aurions bien évidemment pu créer l'association « telephone_employe» dans Telephone, le résultat aurait été le même.

Association avec attributs

Pour l'association entre « employe» et « projet», l'id de « employe» et celui de « projet» sont clés primaires dans « employe_affectation_projet ». Étant donné que cette association a des attributs, nous devons découper cette association en deux relations « Un-À-Plusieurs » par le biais de la table « employe_affectation_projet ». Nous avons donc créé une entité « Employe_Affectation_Projet » comme cela était

expliqué dans la première sous-partie de cette partie. Grâce à des annotations « @OneToMany » dans « Projet »

```
@OneToMany(mappedBy="projet", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Employe_Affectation_Projet> employe_affectation_projet = new ArrayList<Employe_Affectation_Projet>();
```

et « Employe »

```
@OneToMany(mappedBy="employe")
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Employe_Affectation_Projet> employe_affectation_projet=new ArrayList<Employe_Affectation_Projet>();
```

On fait les liens entre ces deux entités et l'entité « Employe_Affectation_Projet » qui contient le code suivant :

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "projet_numero", insertable = false, updatable = false)
private Projet projet;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "employe_id", insertable = false, updatable = false)
private Employe employe;
```

Une instance de « Employe_Affectation_Projet » contient un numéro de projet et un employé, tandis qu'une instance de « Projet » ou de « Employe » peut être présente plusieurs fois dans « Employe_Affectation_Projet », d'où la présence de « Collection » dans « Projet » et « Employe ». On mappe les instances de « Projet» et « Employe » avec « Employe_Affectation_Projet» via les attributs « projet» et « employe» présents dans « Entreprise_Affectation_Projet» qui définissent les colonnes de jointures.

Pour l'association entre « entreprise» et « entreprise», l'id de « entreprise» et celui de « entreprise» sont clés primaires dans « entreprise_faire_affaire». Étant donné que cette association a des attributs, nous devons découper cette association en deux relations « Un-À-Plusieurs » par le biais de la table « entreprise_faire_affaire». Nous avons donc créé une entité « Entreprise_Faire_Affaire» comme cela était expliqué dans la première sous-partie de cette partie. Grâce à des annotations « @OneToMany » dans « Entreprise».

```
@OneToMany(mappedBy="entreprise", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Entreprise_Faire_Affaire> entreprise_faire_affaire=new ArrayList<Entreprise_Faire_Affaire>();

@OneToMany(mappedBy="entreprise_partenaire", cascade=CascadeType.ALL)
@OnDelete(action = OnDeleteAction.CASCADE)
private Collection<Entreprise_Faire_Affaire> entreprise_partenaire_faire_affaire = new ArrayList<Entreprise_Faire_Affaire>()
```

On fait les liens entre ces deux entités et l'entité « Entreprise_Faire_Affaire» qui contient le code suivant :

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "entreprise_id", insertable = false, updatable = false)
private Entreprise entreprise;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "entreprise_partenaire_id", insertable = false, updatable = false)
private Entreprise entreprise_partenaire;
```

Une instance de « Entreprise_Faire_Affaire» contient les id de deux entreprises qui font affaire, et une instance de « Entreprise » peut être présente plusieurs fois dans « Entreprise_Faire_Affaire», d'où la présence de « Collection » dans « Entreprise ».
On mappe les instances de « Entreprise » avec « Entreprise_Faire_Affaire » via les attributs « entreprise » et « entreprise_partenaire » présents dans « Entreprise_Faire_Affaire » qui définissent les colonnes de jointures.

Partie II : Hibernate Validator

1) Exploration et documentation des contraintes

Les exemples pour les contraintes sont pour la plupart issus du site suivant : http://jmdoudoux.developpez.com/cours/developpons/java/chap-validation_donnees.php

- **@DecimalMax**

Syntaxe : @DecimalMax(value=,inclusive=)

Il convient de l'utiliser pour les types de données suivants : BigDecimal, BigInteger, String, byte, short, int, long.

Cette contrainte permet de vérifier si la valeur de l'attribut est strictement inférieure au maximum spécifié si l'argument inclusive est égal à faux. Si inclusive est à vraie, il va vérifier si la valeur est inférieure ou égale au maximum. On spécifie le maximum sous forme d'une chaîne de caractères suivant le format d'un BigDecimal comme ci-après : BigDecimal bd = new BigDecimal("345.67886"); on mettrait value="345.67886" nombre avec 3 chiffres avant la virgule et 5 chiffres après la virgule. De plus si le paramètre est à null alors la contrainte est validée.

Exemple :

Si la valeur passée à l'attribut n'est pas de type String :

```
package com.jmdoudoux.test.validation;
import javax.validation.constraints.DecimalMin;

public class MonBean
{
	@DecimalMax(value="10.5")
	private int maValeur;

	public MonBean(int maValeur)
	{
		super();
		this.maValeur = maValeur;
	}

	 public int getMaValeur()
	{
		return maValeur;
	 }

	public void setMaValeur(int maValeur)
	{
		this.maValeur = maValeur;
	}
```

```
}
```

Si la valeur passée à l'attribut est de type String :

```
package com.jmdoudoux.test.validation;
import javax.validation.constraints.DecimalMin;

public class MonBean
{
	@DecimalMin(value="10.5")
	private String maValeur;

	public MonBean(String maValeur)
	{
		super();
		this.maValeur = maValeur;
	 }

	public String getMaValeur()
	{
		return maValeur;
	}

	public void setMaValeur(String maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

Si on affecte à maValeur un String il faut que ce String soit une valeur qui puisse être couverte en BigDecimal, par exemple il faut passer en paramètre une valeur du genre « 18.2 » à la place de « test » dans le code suivant :

```
package com.jmdoudoux.test.validation;
import java.util.Set;
import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationMonBean
{

	public static void main(String[] args)
	{
		MonBean monBean = new MonBean("test");
		...
	}
}
```

Si on passe « test » en paramètre on obtient le résultat suivant : Impossible de valider les données du bean : MonBean.maValeur doit être plus petit que 10.5.

- **@DecimalMin**

Syntaxe : @DecimalMin(value=,inclusive=)

Il convient de l'utiliser pour les types de données suivants : BigDecimal, BigInteger, String, byte, short, int, long.

Cette contrainte permet de vérifier si la valeur de l'attribut est strictement supérieure au minimum spécifié si l'argument inclusive est égal à faux. Si inclusive est à vraie, il va vérifier si la valeur est supérieure ou égale au minimum. On spécifie le minimum sous forme d'une chaîne de caractères suivant le format d'un BigDecimal comme ci-après : BigDecimal bd = new BigDecimal("345.67886"); on mettrait value="345.67886" nombre avec 3 chiffres avant la virgule et 5 chiffres après la virgule. De plus si le paramètre est à null alors la contrainte est validée.

Exemple :

Si la valeur passée à l'attribut n'est pas de type String :

```
package com.jmdoudoux.test.validation;
import javax.validation.constraints.DecimalMin;

public class MonBean
{
	@DecimalMin(value="10.5")
	private int maValeur;

	 public MonBean(int maValeur)
	{
		super();
		this.maValeur = maValeur;
	}

	public int getMaValeur()
	{
		return maValeur;
	 }

	public void setMaValeur(int maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

Si la valeur passée à l'attribut est de type String :

```
package com.jmdoudoux.test.validation;
```

```
import javax.validation.constraints.DecimalMin;

public class MonBean
{
		@DecimalMin(value="10.5")
		private String maValeur;

		public MonBean(String maValeur)
		{
			super();
			this.maValeur = maValeur;
		 }

		public String getMaValeur()
		{
			return maValeur;
		}

		public void setMaValeur(String maValeur)
		{
			this.maValeur = maValeur;
		}
}

package com.jmdoudoux.test.validation;
import java.util.Set;
import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationMonBean
{

		public static void main(String[] args)
		{
			MonBean monBean = new MonBean("test");
			...

		}
}
```

Si on passe « test » en paramètre on obtient le résultat : Impossible de valider les données du bean : MonBean.maValeur doit être plus grand que 10.5.

- **@Digits**

		Syntaxe : @Digits(integer=,fraction=)

Il convient de l'utiliser pour les types de données suivants : BigDecimal, BigInteger, String, byte, short, int, long.

Vérifie si la valeur affectée à l'attribut, ayant cette contrainte, comporte moins ou autant de chiffres avant la virgule que ce qui est spécifié pour le paramètre integer et moins ou autant de chiffres après la virgule (de valeurs décimales) que ce qui est spécifié pour fraction.

Exemple :

```
import javax.validation.constraints.Digits;

public class MonBean
{
	@Digits(integer=5, fraction=2)
	 private String maValeur;

	public MonBean(int maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

- **@Max**

Syntaxe : @Max(value=)

Il convient de l'utiliser pour les types de données suivants : BigDecimal, BigInteger, byte, short, int, long.

Vérifie si la valeur affectée à l'attribut ayant cette contrainte est inférieure ou égale à la valeur maximum spécifiée au niveau de l'argument value de la contrainte Max sous la forme d'un entier de type long. De plus si la valeur est à null alors la contrainte de @Max sera valide.

Exemple :

```
public class MonBean
{

	@Max(value=20)
	private int maValeur;

	public MonBean(int maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

- **@Min**

Syntaxe : @Min(value=)

Il convient de l'utiliser pour les types de données suivants : BigDecimal, BigInteger, byte, short, int, long.

Vérifie si la valeur affectée à l'attribut ayant cette contrainte est supérieure ou égale à la valeur minimum spécifiée au niveau de l'argument value de la contrainte Min sous la forme d'un entier de type long. De plus si la valeur est à null alors la contrainte de @Min sera valide.

Exemple :

```
import javax.validation.constraints.Min;

public class MonBean
{
	@Min(value=10)
	private int maValeur;

	public MonBean(int maValeur)
	{
		this.maValeur = maValeur;
	}
 }
```

- **@NULL**

Syntaxe : @NULL

Cette contrainte peut s'appliquer à tous les types.
Vérifie que le paramètre ayant cette contrainte soit à null.

Exemple :

```
import javax.validation.constraints.Min;

public class MonBean
{
	@NULL
	private int maValeur;

	public MonBean(int maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

On devra passer null en paramètre de MonBean() quand on instanciera un objet de type MonBean.

- **@Past**

Syntaxe : @Past

Cette contrainte peut s'appliquer aux types suivants : java.util.Date, java.util.Calendar, java.time.chrono.ChronoZonedDateTime, java.time.Instant, java.time.OffsetDateTime.

Vérifie que la date passée à l'attribut ayant l'un des types ci-dessus et ayant cette contrainte, soit inférieure à la date actuelle.

Exemple :

```
import java.util.Date;

import javax.validation.constraints.Past;

public class MonBean
{
	@Past
	private Date maValeur;

	public MonBean(Date maValeur)
	{
		super();
		this.maValeur = maValeur;
	}
}
```

- **@Future**

Syntaxe : @Future

Cette contrainte peut s'appliquer aux types suivants : java.util.Date, java.util.Calendar, java.time.chrono.ChronoZonedDateTime, java.time.Instant, java.time.OffsetDateTime.

Vérifie que la date passée à l'attribut ayant l'un des types ci-dessus et ayant cette contrainte, soit supérieure à la date actuelle.

Exemple :

```
import java.util.Date;

import javax.validation.constraints.Past;

public class MonBean
{
	@Future
	private Date maValeur;

	public MonBean(Date maValeur)
	{
		super();
		this.maValeur = maValeur;
	}
}
```

- **@Pattern**

Syntaxe : @Pattern(regex=,flag=)

Cette contrainte peut s'appliquer au type String.
Vérifie si la chaîne passée à l'attribut ayant cette contrainte respecte les conditions de l'expression régulière passée au paramètre regex. « La donnée est valide si sa valeur est null. Le format de l'expression régulière est celui utilisé par la classe java.util.regex.Pattern.L'attribut flags est un tableau de l'énumération Flag qui précise les options à utiliser par la classe Pattern. Les valeurs de l'énumération sont : UNIX_LINES, CASE_INSENSITIVE, COMMENTS, MULTILINE, DOTALL, UNICODE_CASE et CANON_EQ ».

UNIX_LINES : autorise le mode Unix lines et dans ce mode seulement \n comme élément pour terminer une ligne.
CASE_INSANSITIVE : la chaîne sera sensible à la casse si l'on utilise ce flag.
COMMENTS : permet de mettre des espaces et des commentaires dans l'expression régulière du pattern.
MULTILINE : permet de définir une expression régulière contenant plusieurs lignes. Les caractères ^ et $ définissent le début d'une nouvelle ligne.
DOTALL : si dans l'expression régulière on a un '.' alors grâce à ce mode le '.' peut correspondre à n'importe quel caractère, il n'est plus seulement considéré comme le caractère de ponctuation.
UNICODE_CASE : permet la casse unicode, c'est-à-dire qu'il permet de différencier des mots avec accents et sans accents, par exemple « lance » et « lancé » sont des mots différents grâce à ce mode.
CANON_EQ : autorise l'équivalence canonique qui « est une forme d'équivalence qui préserve visuellement et fonctionnellement les caractères équivalents. Par exemple le caractère ü est canoniquement équivalent à la séquence u suivi du tréma. »

Exemple :

```
package com.jmdoudoux.test.validation;
import java.util.Date;
import javax.validation.constraints.Pattern;

public class UtilisateurBean extends PersonneBean
{
	private String digiCode;

	public UtilisateurBean(String nom, String prenom, Date dateNaissance,
String digiCode)
	{
		super(nom, prenom, dateNaissance);
		this.digiCode = digiCode;
	}

	//Message correspond à ce qui s'affichera si le code n'a pas un format
convenable.
	@Pattern(regexp="\\d\\d\\d[A-F]",
	message="Le digicode doit contenir 3 chiffres et une lettre entre A et F")
```

```
	public String getDigiCode()
	{
		return digiCode;
	}

	public void setDigiCode(String digiCode)
	{
		this.digiCode = digiCode;
	}
}
```

- **@Size**

Syntaxe : @Size(min=, max=)

Elle peut s'appliquer aux types suivants : String, Collection, Map et tableau. On évalue la taille de chacun de ces types.

Vérifie que la taille de l'attribut ayant cette contrainte soit égale au min ou au max ou bien comprise entre le min et le max. Avec « min (valeur par défaut 0) et max (valeur par défaut Integer.MAX_VALUE) incluses ». Si l'attribut est à null la condition est aussi vérifiée.

Exemple :

```
import javax.validation.constraints.Size;

public class MonBean
{
	@Size(min=10, max=20)
	private String maValeur;

	public MonBean(String maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

- **@Valid**

Syntaxe : @Valid

S'applique à tout type non primitif.

Effectue récursivement une validation sur les objets qui sont associés. Si l'objet est une collection ou un tableau, les éléments sont validés récursivement. Si l'objet est du type Map la valeur de chaque élément est aussi validée récursivement.

Exemple :

```
import javax.validation.constraints.Valid;

public class MonBean
```

```
{
	@Valid
	private String maValeur;

	public MonBean(String maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

- **@Email**

	Syntaxe : @Email

	Elle convient pour les éléments de type String.
	Vérifie si la valeur affectée à l'attribut ayant cette contrainte est une adresse mail valide. On peut également appliquer une expression régulière avec la contrainte @email et un flag comme pour @pattern, afin de définir un type d'adresse mail spécifique que l'attribut doit recevoir lorsqu'il est initialisé.

Exemple :

```
import javax.validation.constraints.Size;

public class MonBean
{
	@Email
	private String maValeur;

	public MonBean(String maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

- **@Length**

	Syntaxe : @Length(min=, max=)

	Elle Convient pour les String.
	Vérifie si la longueur de la chaîne de caractère que l'on affecte à l'attribut ayant cette contrainte est supérieure ou égale aux min et inférieure ou égale au max.

Exemple :

```
import javax.validation.constraints.Size;

public class MonBean
{
	@Length(min=10, max=20)
	private String maValeur;
```

```
	public MonBean(String maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

- **@NotBlank**

Syntaxe : @NotBlank

Elle convient pour les String.
Vérifie si la chaîne de caractères que l'on affecte à l'attribut ayant cette contrainte n'est pas égale à null et que la taille de la chaîne rognée est supérieure à 0. La différence avec @NotEmpty est que cette contrainte ne peut être appliquée qu'à des Strings et les espaces sont ignorés.

Exemple :

```
import javax.validation.constraints.Size;

public class MonBean
{
	@NotBlank
	private String maValeur;

	public MonBean(String maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

On ne pourra pas instancier un objet de type MonBean avec une chaîne de caractères nulle.

- **@NotEmpty**

Syntaxe : @NotEmpty

Il convient de l'appliquer aux types : Strings, Collection, Map et tableau.
Vérifie que l'attribut n'est pas à null ou vide.

```
public class MonBean
{
	@NotEmpty
	private Collection<Integer> maValeur;
	public MonBean(String maValeur)
	{
		this.maValeur = new Collection<Integer>();
	}
```

```
}
```

- **@Range**

Syntaxe : @Range(min= , max=)

Il convient de l'appliquer aux éléments de type : BigDecimal, BigInteger, String, byte, short, int, long.

Vérifie si la valeur affectée à l'attribut ayant cette contrainte se situe entre le min et le max, elle peut également être égale au min ou au max.

Exemple :

```
public class MonBean
{
	@Range(min=5, max=20)
	private int maValeur;

	public MonBean(int maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

- **@URL**

Syntaxe : @URL(protocol=, host=, port=, regexp=, flags=)

Il convient de l'appliquer à des Strings.

Vérifie si la chaîne de caractères que l'on affecte à l'attribut ayant cette valeur est une URL valide selon la norme RFC2396. Si l'un des paramètres host ou port a une valeur spécifique, la chaîne de caractères doit comporter des éléments qui correspondent aux valeurs définies pour host et port. On peut aussi, c'est optionnel, utiliser une expression régulière avec un flag pour définir une adresse UL bien spécifique à laquelle la chaîne de caractères devra correspondre. La contrainte @URL utilise par défaut le constructeur de java.net.URL pour vérifier que la chaîne de caractères est une URL valide.

Exemple :

```
public class MonBean
{
	@URl(host=« 3306 », port=« 3308 », regexp=«^[http://www.] »)
	private String maValeur;

	public MonBean(String maValeur)
	{
		this.maValeur = maValeur;
	}
}
```

- **@ConstraintComposition**

Syntaxe : @ConstraintComposition

On applique à un attribut de n'importe quel type un ensemble de contraintes. La contrainte @ConstraintComposition va nous permettre de savoir comment sont liées les contraintes. @ConstraintComposition peut prendre en argument OR, AND et ALL_FALSE. Si on écrit @CompositionConstraint(AND) il faut que toutes les contraintes définies juste après soient vérifiées. Si on écrit @CompositionConstraint(OR) il suffit qu'une contrainte soit vérifiée pour que l'ensemble des contraintes après @CompositionConstraint soit correct. Enfin avec @CompositionConstraint(ALL_FALSE) si aucune contrainte n'est vérifiée alors l'ensemble des contraintes est juste.

Exemple :

```
public class Salaire
{
	@ConstraintComposition(AND)
	@Min(value=0)
	@Digits(integer=11, fraction=0)
	private Integer salaire_net;

	public Salaire(Integer monSalaire)
	{
		salaire_net=monSalaire;
	}
}
```

2) Création validateur personnalisé

Processus de création de @CodePostalValide

- Tout d'abord j'ai créé une classe de validation CodePostalValidator. Cette classe implémente l'interface ConstraintValidator avec pour annotation CodePostalValide et s'appliquera aux objets de type String car le champ codePostal est de type String. La classe CodePostalValidator contient une méthode (isValid) retournant un boolean : true si le codePostal n'est pas null et s'il respecte le format; false sinon.

```java
import javax.validation.ConstraintValidator;

public class CodePostalValidator implements ConstraintValidator<CodePostalValide, String> {

  @Override
  public void initialize(CodePostalValide constraintAnnotation)
  {}

  //Méthode qui permettra de vérifier si le format du code postal est correct
  @Override
  public boolean isValid(String value, ConstraintValidatorContext context)
  {
      boolean res=true;

      //On vérifie que le code postal n'est pas null et qu'il est de la forme LCL CLC avec entre LCL CLC la présence
      //d'un trait d'union, d'un espace ou non.
      if(value==null || !value.matches("^[A-Z]{1}[0-9]{1}[A-Z]{1}(-| ){1}[0-9]{1}[A-Z]{1}[0-9]{1}$"))
      {
          res=false;
      }
      return res;
  }

}
```

- Une fois que l'on a la classe de validation nous pouvons créer l'annotation CodePostalValide. Grâce à la contrainte : @Constraint(validatedBy ={CodePostalValidator.class}) on relie l'annotation à la classe contenant la méthode de validation. Avec la ligne suivante : @Target(ElementType.FIELD) nous spécifions que l'annotation CodePostalValide va s'appliquer sur les champs d'une classe. Avec la ligne suivante : @Retention(value = RetentionPolicy.RUNTIME) nous spécifions que la contrainte va s'appliquer lorsque l'on exécutera l'application. Enfin on définit dans la code de l'annotation le message qui va s'afficher si la contrainte n'est pas validée.

```java
import java.lang.annotation.Documented;

@Documented
//On relie l'annotation à la classe qui va permettre la validation de la donnée
@Constraint(validatedBy = {CodePostalValidator.class})
@Target(ElementType.FIELD)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface CodePostalValide {

    //Message que l'on renvoi si le format du code postal saisi n'est pas bon
    String message() default "Format de code postal invalide, doit être de la forme LCLCLC, "
            + "ou LCL-CLC ou LCL CLC avec C: chiffre et L : Lettre en majuscule.";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

Nous pouvons maintenant utiliser la contrainte @CodePostalValide en important le package validator.CodePostalValide.

3) Application des validateurs aux entités (voir entités dans projet Eclipse)

Partie III : Hibernate HQL et Query API

1) Exploration et documentation

a) La classe Query pour formuler des requêtes HQL

La classe Query permet de faire des requêtes HQL. On instancie un objet Query en faisant appel à Session.createQuery(), session que l'on crée avec Hibernate. entre parenthèses nous devons spécifier une requête qui reprend le langage SQL et qui peut utiliser des paramètres nommés et peut afficher des résultats d'une certaine façon grâce à la pagination.

Query est une interface qui ne contient donc que des méthodes. Vous pouvez retrouver l'ensemble de ces méthodes sur le site suivant :
https://docs.jboss.org/hibernate/orm/3.2/api/org/hibernate/Query.html bien que ce soit la version 3.2, je n'ai pas réussi à trouver des informations plus récentes sur Query avec hibernate 5.0.2.Final

Les requêtes Query sont exécutées avec les méthodes list() , scroll() et iterate() de l'interface Query
-list() : retourne les résultats de la requête sous forme d'objet List
-scroll(): retourne les résultats de la requête sous forme d'objet ScrollableResults.
-iterate() : retourne les résultats de la requête sous forme d'objet Iterator.

Les requêtes peuvent être exécutées de nouveau avec les méthodes de ci-dessus. Cependant la durée de vie de la requête est équivalente à celle de la session.

Dans le code de test.Main.java on importe les packages

```
import org.hibernate.Query; //Afin de pouvoir utiliser l'interface Query.
import org.hibernate.Session; //Afin de pouvoir utiliser l'objet Session et donc faire des requêtes.
import org.hibernate.SessionFactory; //Afin de créer une session.
```

On crée la requête et on l'exécute

```
Query query=session.createQuery("select * from Adresse");
List<Adresse> adresses1 = query.list();
```

b) Les clauses

La clause from permet de sélectionner une ou plusieurs tables auxquelles nous allons appliquer notre requête. Si nous écrivons seulement FROM maTable cela nous renverra toutes les instances de maTable. Nous pouvons écrire FROM maTable1, maTable2 qui aura pour effet de renvoyer toutes les instances des tables maTable1 et maTable2.

On va exécuter la requête sur la table Adresse

```
Query query=session.createQuery("select * from Adresse");
```

```
List<Adresse> adresses1 = query.list();
```

La clause as est utilisée avec la clause from. Elle permet de définir un alias : FROM maTable1 as tb1. Ainsi lorsque l'on fera référence à maTable1 pour les autres clauses de la requête on utilisera tb1,

Exemple : where tb1.name = « Jean » au lieu d'écrire maTable1.name=« Jean ».

La clause select permet de sélectionner les objets et propriétés qui seront retournés à la suite de l'exécution de la requête. Le select permet de retourner des propriétés de n'importe quel type de valeur. Si la requête renvoie plusieurs objets ou propriétés ces derniers sont contenus dans un tableau de type Object[] ou si on le souhaite sous forme d'une liste. Dans la clause select nous pouvons également utiliser la clause as lorsque l'on effectue une opération du genre sum, max, min , avg, count.

La clause where permet de spécifier des conditions sur les résultats que devra retourner la requête.

Exemple :

On va exécuter la requête sur la table Adresse et sur le premier élément de la table. On a renommé la table avec l'alias.

Cette requête va nous afficher toutes les propriétés du premier élément de la table Adresse.

Exemple :

```
//Utilisation d'un alias
Query query2=session.createQuery("select * from Adresse as adr where adr.id = 1");
List<Adresse> adresses2 = query2.list();
```

La clause order by permet de trier la liste de résultats retournée par la requête selon n'importe quelle propriété de classe ou des composants retournés. On peut utiliser les options asc pour ascendant et desc pour descendant pour l'ordre de tri suivant une propriété.

Exemple :

```
//Utilisation de order by
Query query3=session.createQuery("select * from Adresse as adr where adr.pays=France order by adr.code_postal");
List<Adresse> adresses3 = query3.list();
```

La clause group by permet de regrouper les résultats après une opération d'agrégation (avg, sum, min, max, count) selon une propriété ou des propriétés bien précises.

Exemple :

```
//Utilisation de group by et de count
Query query4=session.createQuery("select count(*) from Adresse as adr where adr.pays=France group by adr.code_postal");
List<Adresse> adresses4 = query4.list();
```

La clause update est utilisée pour mettre à jour une ou plusieurs propriétés appartenant à un ou plusieurs objets.

Exemple :

```
//Utilisation de update
Query query5=session.createQuery("update Adresse set numero=23 where adr.id=2");
List<Adresse> adresses5 = query5.list();
```

La clause delete permettra de supprimer un champ bien précis d'une table si l'on applique une condition avec la clause where.

Exemple :

```
//Utilisation de delete
Query query6=session.createQuery("delete from Adresse where adr.id=4");
List<Adresse> adresses6 = query6.list();
```

Les fonctions d'agrégations permettent d'effectuer des opérations sur les propriétés d'une table :
- min permet de retourner la valeur minimale pour une propriété sur l'ensemble des instances d'une table.
- max permet de retourner la valeur maximale pour une propriété sur l'ensemble des instances d'une table.
- avg permet de faire une moyenne pour une propriété sur l'ensemble des instances d'une table.
- sum permet d'additionner toutes les instances d'une propriété d'une table.
- count : - count (*) compte toutes les instances d'une ou plusieurs tables.
  - count (…) compte toutes les instances d'une table bien précise que l'on spécifie entre les parenthèses.
  - count (distinct ...) compte également toutes les instances mais évite de compter deux fois une même instance.

Exemples :

```
//Utilisation de sum
Query query7=session.createQuery("select sum(emp.salaire_net) from Employe as emp");
List<Employe> employes1 = query7.list();
```

```
//Utilisation de min
Query query8=session.createQuery("select min(emp.salaire_net) from Employe as emp");
List<Employe> employes2 = query8.list();

//Utilisation de max
Query query9=session.createQuery("select max(emp.salaire_net) from Employe as emp");
List<Employe> employes3 = query9.list();

//Utilisation de avg
Query query10=session.createQuery("select avg(emp.salaire_net) from Employe as emp");
List<Employe> employes4 = query10.list();
```

Pour count voir exemple de group by

Les sous-requêtes sont définies dans les clauses select ou where de la requête principale. Elles sont écrites entre parenthèses et consistent souvent en une opération d'agrégation.

Exemple :

```
//Utilisation des sous-requêtes
Query query11=session.createQuery("select emp.nom from Employe as emp where emp.salaire_net < (select avg(e.salaire_net) from Employe as e)");
List<Employe> employes5 = query11.list();
```

c) Paramètres nommés

Les paramètres nommés se trouvent dans les requêtes (query) que l'on peut écrire dans les classes JAVA avec Hibernate.

On leur affecte une certaine valeur selon le résultat que l'on souhaite obtenir à la suite de la requête. On peut ainsi créer des requêtes paramétrables. Pour affecter une valeur à ces paramètres nommés on utilise des méthodes de la classe Query. JDBC permet également d'utiliser des paramètres nommés du moins il en existe seulement un seul qui est '?'. Une requête peut contenir plusieurs paramètres nommés, il faut savoir que pour JDBC le premier paramètre nommé porte l'indice numéro 1, le second l'indice numéro 2 et ainsi de suite alors que pour Hibernate le premier paramètre nommé porte l'indice numéro 0, le second porte l'indice numéro 2 et ainsi de suite.

Nous ne pouvons pas mélanger les paramètres nommés de JDBC et de Hibernate.Avec Hibernate nous pouvons utiliser des paramètres nommés plus explicites qu'avec JDBC, par exemple pour un nom il y a le paramètre nommé « :name ». Les paramètres nommés sont également auto-documentés par leur nom.

Exemple :

```
Query query12=session.createQuery("select * from Employe as emp
```

```
where emp.prenom = :name");
query12.setString("name", "Marc");
List<Employe> employes6 = query12.list();
```

d) La pagination

La pagination permet de ne pas charger à la fois tous les résultats d'une requête afin de garder des performances convenables pour l'accès à la base de données et ne pas faire crasher son système s'il y a trop de données à gérer.

Il existe de méthode de l'interface Query pour la pagination :

- Query setFirstResult(int startPosition) : cette méthode prend un entier qui représente la première ligne pour l'affichage des résultats avec pagination.
- Query setMaxResults(int maxResult) : Cette méthode permet d'afficher au maximum maxResult résultats. Si maxResult est égal à 20 alors la requête affichera au maximum 20 résultats.

Exemple :

```
//Utilisation de la pagination
Query query15 = session.createQuery("select * from Employe");
query.setFirstResult(1);
query.setMaxResults(10);
List<Employe> employes9 = query15.list();
```

e) Itération sur les résultats retournés

On effectue une itération sur les résultats si l'on exécute la requête avec la méthode iterate() de l'interface Query. On peut également parcourir, à l'aide d'un itérateur, les résultats de la requête qui ont été retournés sous la forme d'une liste si on l'a exécutée avec la méthode list().

Selon les cas les performances seront meilleures si l'on utilise l'une ou l'autres des méthodes précédentes. Si les instances d'entités retournées par la requête sont déjà chargées dans la session ou le cache de second niveau alors il vaudra mieux utiliser iterate(). Sinon list() sera plus efficace, car si les instances ne sont pas chargées iterate() nécessitera plusieurs accès à la base de données pour exécuter une simple requête.

Exemple :

```
//Itération sur les résultats
Query query13=session.createQuery("select * from Employe as emp where
emp.salaire_net < (select avg(e.salaire_net) from Employe as e)");
Iterator<Employe> employes7 = query13.iterate();
while(employes7.hasNext())
{
        Employe emp = employes7.next();
        System.out.println("Nom" + emp.getNom() + ", Prénom : " +
emp.getPrenom());
}
```

f) Requêtes nommées (@NamedQuery)

L'annotation @NamedQuery permet de donner un nom à une requête et de la rappeler par ce nom. Elles sont compilées à l'exécution et stockées dans le cache. Lors de l'exécution, la syntaxe de la requête est validée. Les principaux avantages des requêtes nommées sont :

- la réutilisation dans plusieurs endroits. il suffit juste d'utiliser le nom donné à la requête pour la réutiliser.
- elles sont chargées au démarrage dès lors la réponse de l'application à la suite de la requête sera plus rapide mais le temps de chargement de l'application sera plus long
- « les requêtes sont regroupées avec le mapping ce qui permet de faciliter leur écriture ».

Exemple :

On crée la requête nommée dans l'entité Adresse

```
@Entity
@Table(name="Adresse")
@NamedQuery(name=Adresse.GET_ADRESSE_BY_ID,
query=Adresse.GET_ADRESSE_BY_ID_QUERY)
public class Adresse
{
        static final String GET_ADRESSE_BY_ID_QUERY = "from Adresse adr where
adr.id=:id";
   public static final String GET_ADRESSE_BY_ID = "GET_ADRESSE_BY_ID";

        @Id
        @GeneratedValue
        @Column(name="id")
        private Integer id;

        …
}
```

Pour le test :

```
//Utilisation des requêtes nommées
Query
query14=session.getNamedQuery(Adresse.GET_ADRESSE_BY_ID).setInteger("id",
1);
List<Employe> employes8 = query14.list();
session.close();
sf.close();
```

g) Différents types de jointures

Il existe 4 types de jointures :
- inner join : elle va comparer tous les objets comportant une concordance être les

deux tables qui caractérisent la jointure
- left outer join : elle va comparer tous les objets comportant une concordance être les deux tables qui caractérisent la jointure et renvoyer également tous les objets de la table de gauche l'association.
- right outer join : elle va comparer tous les objets comportant une concordance être les deux tables qui caractérisent la jointure et renvoyer également tous les objets de la table de droite l'association.
- full join : elle va renvoyer tous les objets des tables appartenant à l'association.

Exemple :

```
//Utlisation de la jointure innerjoin
Query query16 = session.createQuery("select * from Adresse as adr inner join
Employe as emp where adr.id=emp.adresse_id");
List<Adresse> adresse7 = query16.list();
```

2) Application des différents éléments (voir les exemples de Partie III première sous-partie)

3) Méthode implémentant les opérations CRUD (voir la classe Main)

Je ne savais pas comment utiliser la réflexivité, du coup j'ai défini les opérations CRUD pour chaque entité.

Bibliographie

http://jmdoudoux.developpez.com/cours/developpons/java/chap-validation_donnees.php

http://docs.jboss.org/hibernate/validator/5.2/reference/en-US/html/ch02.html

http://docs.jboss.org/hibernate/validator/5.2/reference/en-US/html/ch11.html

http://www.developpez.net/forums/d1161776/java/general-java/debuter/manipuler-bigdecimal/

https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

https://fr.wikipedia.org/wiki/%C3%89quivalence_Unicode#.C3.89quivalence_canonique

http://makble.com/the-dotall-mode-in-java-regular-expression

http://www.javabeat.net/bean-validation-java-ee-creating-custom-constraints-validations/

http://www.dil.univ-mrs.fr/~massat/docs/hibernate-2/reference/fr/html/manipulatingdata.html

https://docs.jboss.org/hibernate/stable/core.old/reference/fr/html/objectstate-querying.html

http://howtodoinjava.com/2013/07/03/hibernate-named-query-tutorial/

http://www.tutorialspoint.com/hibernate/hibernate_query_language.htm

https://docs.jboss.org/hibernate/orm/3.3/reference/fr/html/queryhql.html

http://www.trucsweb.com/tutoriels/asp/tw104/

http://www.tomsquest.com/blog/2010/03/jpa-les-illusions-sur-les-namedqueries/

Pour comprendre le second niveau de cache :
http://www.tutorialspoint.com/hibernate/hibernate_caching.htm

https://blog.axopen.com/2013/11/les-cles-primaires-composees-avec-hibernate-4/