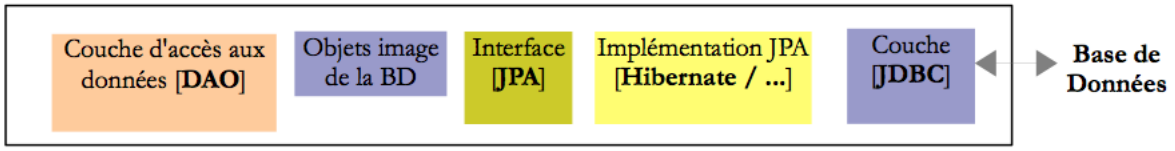
Objectif **:** Base de données MySQL et Java Persistance API (JPA)

1 Objectifs

Le travail à faire est de se familiariser avec JPA et d'implémenter des requêtes JPQL (Java Persistence Query Language – langage de requêtes de la couche JPA) indiquées en fin de document, dans le cas de la base [dbrdvmedecins] fournie (script sql). Ce travail correspond à la problématique d'interaction avec une base de données en manipulant des objets en Java. Dans l'architecture en couche, ceci est requis par la couche DAO, qui dialogue avec l'implémentation de la spécifique JPA (hibernate dans notre cas). Cette implémentation accède à la base en utilisant JDBC (Java DataBase Connectivity).



2 Projet « mv_rdvmedecins »

2.1 Mise en place du projet et lien avec la base de données

- Démarrer WAMP, puis importer et exécuter le script [dbrdvmedecin.sql] fourni.
- Créer le projet netbeans [mv_rdvmedecins] de type « Maven / JavaApplication ».
- Connexion à la « base de données » depuis netbeans : « services » → « bouton droit sur base de données / nouvelle connexion » → pilote « MySQL (Connector / J Driver) → définir «base=dbrdvmedecins / localhost / 3306 / root / root » (vérifier depuis WAMP votre configuration: url, port, nom, mot de passe). Vous devez (sous netbeans/services) voir apparaître une « connexion » du type (exemple): « jdbc:mysql://localhost:3306/dbrdvmedecins » . Vérifier ensuite que vous pouvez ensuite consulter les tables depuis netbeans.
- Créer une unité de persistance associée au projet. « Bouton droit sur projet » → « new persistence unit », en sélectionnant la base considérée, et l'implémentation JPA (ici « hibernate »), enfin « none » sur « table generation strategy ». A noter que la « table generation strategy » peut être « drop and create » (efface et réinitialise les tables), « create » (créé les tables mais lève une erreur si elles existent déjà) ou « None » : ne fait rien. On remarque que la dépendance à « org.hibernate » est automatiquement ajoutée (voir « pom.xml »). On remarque également qu'un fichier « persistance.xml » est créé (il sera utilisé par le programme java pour accéder à la base).

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence … >
  <persistence-unit name="istia_mv_rdvmedecins_jar_1.0-SNAPSHOTPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/dbrdvmedecins?zeroDateTimeBehavior=convertToNull"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password" value="root"/>
      <property name="hibernate.cache.provider_class" value="org.hibernate.cache.NoCacheProvider"/>
      <property name="hibernate.show_sql" value="True"/>
      <property name="hibernate.format_sql" value="True"/>
    </properties>
  </persistence-unit>
</persistence>
```

Fichier « persistance.xml »

- Ajouter la dépendance au pilote JDBC de MySQL : « bouton droit sur dependencies », de manière à compléter le fichier maven « pom.xml » avec la dépendance suivante (vous pouvez aussi éditer « manuellement » ce « pom.xml »):

```
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.39</version>
</dependency>
```

« pom.xml » avec la dépendance au pilote JDBC [<property name = "javax. Persistence. jdbc.driver" value="com.mysql.jdbc.Driver"/>] dans le fichier « persistance.xml ». Attention à bien respecter la version car la plus récente est incompatible avec MySQL5 !

- Créer enfin les entités JPA : classes images des tables de la base, générées automatiquement depuis Netbeans : « bouton droit sur le projet » → « New / Persistance / entity classes from database », avec la configuration suivante :
  - renommer les classes « au singulier » (raison pratique de lisibilité)
  - décocher « related tables » et « Generate ... » (« tous les generate »)
  - choisir java.util.List pour les collections,
  - choisir un nom de package pertinent dans lequel les classes vont être générées : par exemple « jpa »

2.2 *Programme simple de test*

Créer le programme « simple » suivant pour tester que tout fonctionne ("persistence_unit_name" à remplacer en fonction du nom l'unité de persistance déclaré dans votre « persistence.xml »).

```
package rdvmedecins.console; //depend de votre projet
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class MainJpql {
public static void main(String[] args) {
// EntityManagerFactory
EntityManagerFactory emf = Persistence.createEntityManagerFactory("persistence_unit_name");
// entityManager
EntityManager em = emf.createEntityManager();
// Requete simple
Query        qr = em.createQuery("select c from Client c");
try
{   for (Object o : qr.getResultList())    { System.out.println(o);  }
}
catch (Exception e)
{    System.out.println("L'exception suivante s'est produite : " + e); }
// on vide le contexte de persistance
em.clear();
// fermeture des ressources
em.close();
emf.close();
}}
```

Programme « MainJpql.java » de test.

Ce programme exécute la requête JPQL [select c from Clients c]. Cette requête permet de récupérer chaque ligne de la table « Clients » sous forme d'un objet de type « Client », et de l'afficher. Avant d'exécuter ce programme, pensez à correctement configurer le nom de l'unité de persistance (ligne [Persistence.createEntityManagerFactory("persistence_unit_name")]), celui-ci correspondant au nom indiqué dans le fichier « persistance.xml » (<persistence-unit name="…" >).

Travail à faire : on pourra surcharger (@Override) la méthode « public String toString() {} » de la classe « Client » pour personnaliser l'affichage, afin, notamment, d'afficher la valeur de certains attributs (e.g. le nom), afin d'avoir la sortie console suivant :

```
Client[1,Mr,Jules,MARTIN]
Client[2,Mme,Christine,GERMAN]
...
```

2.3 Annotations: quelques explications

(https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html)

```
6. @Entity
7. @Table(name = "medecins")
8. public class Medecin implements Serializable
…
11. @Id
12. @GeneratedValue(strategy = GenerationType.IDENTITY)
13. @Column(name = "ID")
14. private Long id;
15.
16. @Column(name = "TITRE")
17. private String titre;
18.
19. @Column(name = "NOM")
20. private String nom;
...
```

- @Entity fait de la classe [Medecin], une entité JPA, i.e.. une classe liée à une table de BD via l'API JPA
- Les classes générées sont « Serializables » (i.e. implémentent [Serializable]) : nécessaire dans les applications client / serveur, pour échanger les entités sous forme de « chaîne de caractères »
- @Id indique que le champ annoté est associé à la clé primaire de la table. La couche [JPA] va générer la clé primaire des lignes qu'elle insèrera dans la table [Medecins]. Ici la stratégie GenerationType.IDENTITY indique que la couche JPA va utiliser le mode auto_increment de la table MySQL (valeur unique générée automatiquement à chaque nouvelle insertion dans une table)
- @Column(nam = "…") associent les attributs aux colonnes des tables
- Voir ci-dessous : la table [creneaux] a une clé étrangère sur la table [medecins]. Un créneau appartient à un médecin. Inversement, un médecin a plusieurs créneaux qui lui sont associés. On a donc une relation un (médecin) à plusieurs (créneaux), une relation qualifiée par l'annotation @OneToMany par JPA (ligne 28). Le champ de la ligne 26 contiendra

tous les créneaux du médecin. Ceci sans programmation. Pour comprendre totalement la ligne 28, il nous faudra présenter la classe [Creneau] ; la table [CRENEAUX] avait une clé étrangère vers la table [MEDECINS] : un créneau est associé à un médecin. Plusieurs créneaux peuvent être associés au même médecin. On a une relation de la table [CRENEAUX] vers la table [MEDECINS] qui est qualifiée de plusieurs (créneaux) à un (médecin). C'est l'annotation @ManyToOne de la ligne 35 qui sert à qualifier la clé étrangère, la ligne 34 avec l'annotation @JoinColumn précise la relation de clé étrangère : la colonne [ID_MEDECIN] de la table [CRENEAUX] est clé étrangère sur la colonne [ID] de la table [MEDECINS]. Dans l'entité [Medecin], l'attribut cascade=CascadeType.ALL fixe le comportement de l'entité [Medecin] vis à vis de l'entité [Creneau] :

- si on insère une nouvelle entité [Medecin] dans la base, alors les entités [Creneau] du champ de la ligne 2 doivent être insérées elles-aussi,
- si on modifie une entité [Medecin] dans la base, alors les entités [Creneau] du champ de la ligne 2 doivent être modifiées elles-aussi,
- si on supprime une entité [Medecin] dans la base, alors les entités [Creneau] du champ de la ligne 2 doivent être supprimées elles-aussi.

```
//Extrait de la classe [Medecin]
28. @OneToMany(cascade = CascadeType.ALL, mappedBy = "idMedecin")
29. private List<Creneau> creneauList;
//Extrait de la classe [Creneau]
31. @OneToMany(cascade = CascadeType.ALL, mappedBy = "idCreneau")
32. private List<Rv> rvList;
33.
34. @JoinColumn(name = "ID_MEDECIN", referencedColumnName = "ID")
35. @ManyToOne(optional = false)
36. private Medecin idMedecin;
```

2.4 Travail à faire : requêtes JPQL

Donner les requêtes JPQL (Java Persistence Query Language) permettant d'obtenir les informations suivantes :

- Question 1 : liste des médecins dans l'orde décroissant de leurs noms
- Question 2 : liste des médecins dont titre='Mr'
- Question 3 : liste des créneaux horaires de Mme Pelissier
- Question 4 : liste des Rv pris dans l'ordre croissant des jours
- Question 5 : liste des clients (nom) ayant pris RV avec Mme PELISSIER le 23/08/2006
- Question 6 : nombre de clients de Mme PELISSIER le 24/08/2006
- Question 7 : les clients n'ayant pas pris de Rdv
- Question 8 : les médecins n'ayant pas de Rdv

Testez les requêtes en utilisant le programme « MainJPQL ». Dans certains cas, il y aura des jointures (simple ou multiple). Pour la documentation :

- Tutoriels sur internet, par exemple http://blog.paumard.org/cours/jpa/chap06-requetes-exemple.html
- Spécifications officielles: http://docs.oracle.com/html/E13946_04/ejb3_langref.html
- Très bon rappel pédagogique sur les jointures: http://www.epershand.net/developpement/mysql-bdd/comprendre-jointures-inner-left-right-join-mysql

2.5 Adaption des entités produites automatiquement (optionnel)

La génération automatique des entités JPA nous permet d'obtenir une base de travail qu'il faut parfois ajuster.

- Ecrire des méthodes [toString] plus explicites que celles générées,
- Supprimer les relations @OneToMany inverses des relations @ManyToOne. Elles ne sont pas indispensables et elles amènent parfois des complications de programmation (dépendances croisées). Par exemple, les « médecins » et « clients » n'ont pas besoin de savoir à quels « rendez-vous » ils sont associés. Seuls les « rendez-vous » ont besoin de connaître les « médecins » et « clients » (clé étrangère).
- Les entités [Medecin] et [Client] sont analogues. On va les faire dériver d'une classe [Personne] que vous allez ajouter (factorisation du code de [Medecin] et [Client] dans [Personne]), en suivant le schéma suivant. Le plus simple est de copier la classe [Client] (ou [Medecin]) en [Personne], et de simplement replacer les annotations « @Entity et @Table(...) » par « @MappedSuperclass ». Simplifier ensuite les classes [Medecin] et [Client] comme illustré ci-dessous : les méthode [hashCode] [equals] et [toString] sont conservées à l'identique.

```
6. @MappedSuperclass
7. public class Personne implements Serializable {
```

```
8. private static final long serialVersionUID = 1L;
9. @Id
10. @GeneratedValue(strategy = GenerationType.IDENTITY)
11. @Column(name = "ID")
12. private Long id;
13.
14. @Basic(optional = false)
15. @Column(name = "TITRE")
16. private String titre;
…
```

```
6. @Entity
7. @Table(name = "clients")
8. @NamedQueries({@NamedQuery(name = "Client.findAll", query = "SELECT c FROM Client c")})
9. public class Client extends Personne implements Serializable {
10. private static final long serialVersionUID = 1L;
11. // constructeurs
12. public Client() { super(); }
14. }
15.
16. public Client(Long id) {  super(id); }
19.
20. public Client(Long id, String titre, String nom, int version, String prenom) { super(id, titre, nom, version, prenom); }
24. @Override
25. public int hashCode() { … }
28.
29. @Override
30. public boolean equals(Object object) { … }
33.
34. @Override
35. public String toString() { return String.format("Client[%s,%s,%s,%s]", getId(), getTitre(), getPrenom(), getNom()); }
38.}
```

3 *Projet « pam_jpa_hibernate »*

3.1 Etape 1 : test préliminaire

Travail à faire : créer les entités (classes) [Cotisation] [Employe] [Indemnite] associées aux tables générées par [dbpam.sql], en suivant la procédure précédente (**ne pas oublier la dépendance à « mysql-connector-java » dans le pom.xml**). Pour tester le bon fonctionnement, écrire un programme « Main » (voir ci-après) qui créé les objets java de type [Employe] disponible dans la base de données et les affiche en mode console (personnaliser la méthode toString()).

Extrait du main

```
List< Object > os = em.createQuery("select e from Employe e").getResultList();
for(Object o : os) { System.out.println(o); }
```

Avec la méthode « toString() » suivante pour [Employe] :

```
@Override
public String toString() { return "jpa.Employe[ nom=" + getNom() + " indemnite=" + getIndemniteId()+ " ]";  }
```

Structure du code Java de test dont l'exécution produit :

```
jpa.Employe[ nom=Jouveinal indemnite=jpa.Indemnite[ id=2Base heures=2.1 ] ]
jpa.Employe[ nom=Laverti indemnite=jpa.Indemnite[ id=1Base heures=1.93 ] ]
```

Quelques commentaires et ajustements du code:

- pour simplifier, penser à éliminer les relations @OneToMany
- on souhaite que l'attribut « securité social » [ss] de [Employe] soit non nul et unique : ceci se traduit par @Column(name = "SS",nullable=false,unique=true). Remarque : on pourra consulter en ligne toutes les options possibles (https://docs.oracle.com/javaee/7/api/javax/persistence/Column.html)

3.2 Etape 2 : test sur la « table generation strategy »

Travail à faire : Configurer cette stratégie à « drop and create » (supprimer et créer) au niveau du fichier de configuration « persistence.xml » (peut se faire en mode graphique avec netbeans, en cliquant sur le fichier → « conception »):

```
<property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
```

Re-exécuter le programme précédent et vérifier que les tables sont vides (vidées et recréées) : depuis WAMP → afficher le contenu des tables (vide), la sortie console n'affiche aucun employé

Remarque : pour revenir à l'état « normal » avec les tables initiales, il faut re-importer les tables depuis WAMP et configurer la stratégie de création des tables à « none ».

3.3 Etape 3 : « LAZY » ou « EAGER »

Ceci correspond à la configuration :

```
@ManyToOne(optional = false,fetch=...)
private Indemnite indemniteId;
```

L'attribut fetch définit la stratégie de recherche du champ [Indemnite indemniteId] de la classe [Employe] :

- FetchType.LAZY : lorsqu'un employé est cherché, l'indemnité qui lui correspond n'est pas ramenée. Elle le sera lorsque le champ [Employe.indemniteId] sera référencé pour la première fois. Dans notre cas, ceci se fait lors du System.println(…) qui invoque [Employe.getIndemniteId()]. Avec Hibernate, ce n'est possible que si le contexte de persistance est ouvert. L'intérêt de cette stratégie est d'éviter de ramener trop de données de la base à un moment donné.
- FetchType.EAGER : lorsqu'un employé est cherché, l'indemnité qui lui correspond est ramenée. C'est le mode par défaut lorsqu'aucun mode n'est précisé.

Travail à faire : vérifier le comportement en modifiant en utilisant le débugger et en consultant l'attribut « indemnite » des employés récupérés par une requête JPQL : "select e from Employe e". Dans le cas d'une configuration EAGER : l'indemnité est bien de type « Indemnite ». Dans le cas d'une configuration LAZY : l'attribut « indemnité » est de type « Indemnite_$$_.. ». Cet objet n'est pas l'indemnité mais un objet « cachant » l'accès ultérieur à l'indemnité

[0]	Employe	#2739
id	Long	1
adresse	String	"5 rue des oiseaux"
cp	String	"49203"
nom	String	"Jouveinal"
prenom	String	"Marie"
ss	String	"254104940426058"
version	int	1
ville	String	"St Corentin"
indemnite	Indemnite_$$_jvst298_1	#2756
indemniteId	Long	2

Mode « LAZY »

[0]	Employe	#2741
id	Long	1
adresse	String	"5 rue des oiseaux"
cp	String	"49203"
nom	String	"Jouveinal"
prenom	String	"Marie"
ss	String	"254104940426058"
version	int	1
ville	String	"St Corentin"
indemnite	Indemnite	#2758
indemniteId	Long	2

Mode « EAGER »

Dans le cas du mode « LAZY », on peut récupérer l'indemnité de plusieurs :

- Soit lors de l'accès à l'employé pour une jointure explicite ("select e from Employe e left join fetch e.indemnite")
- Soit par le getter : employe.getIndemnite(). Cependant, le contexte de persistance ne doit pas avoir été fermé.
- Soit par une seconde requête de type "select i from Indemnite i where i.id=:key". Il faut avoir la « clé » primaire de l'indemnité : on peut ajouter un nouvel attribut à la classe EMPLOYE, stockant la clé primaire de l'indemnité (clé étrangère dans employé), permettant ainsi une récupération ultérieure (même après la fermeture de la session) de l'indemnité.

Travail à faire : Tester cette dernière solution (ajout d'un attribut supplémentaire à « employé »). Ce nouvel attribut sera « indemniteId » (voir ci-dessous). Tester ensuite de récupérer et afficher (Main) les indemnités de chaque employé par des requêtes de type "select i from Indemnite i where i.id=:key" (attention à ouvrir un contexte de persistance : e.g. EntityManager em2 = emf.createEntityManager();)

```
// indemnités
@JoinColumn(name = "INDEMNITE_ID", referencedColumnName = "ID")
@ManyToOne(optional = false, fetch = FetchType.LAZY)
private Indemnite indemnite;
// clé étrangère
@Column(name = "INDEMNITE_ID", updatable = false, insertable = false, nullable = false)
private Long indemniteId;
…
// toString
@Override
public String toString() { return "jpa.Employe[ nom=" + getNom() + " indemnite id=" + getIndemniteId()+ " ]";}
```

Adaptation de la classe « Employe » : pensez à adapater les getters/setters en conséquence (e.g. getIndemniteId() retourne [indemniteId] et non plus [indemnite]

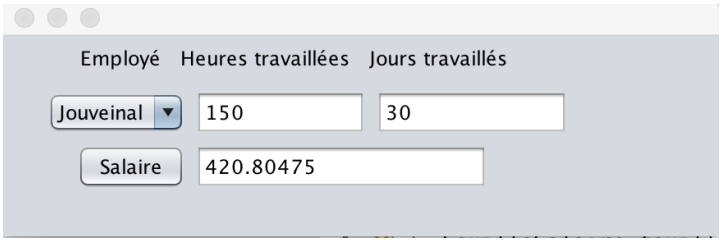
Nous avons rajouté un champ [indemniteId] pour récupérer la valeur de la colonne [INDEMNITE_ID] de la table [EMPLOYES] qui est une clé étrangère sur la colonne [INDEMNITES.ID]. L'annotation [@Column] a des attributs pas encore rencontrés : updatable et insertable. Si on ne les met pas, on a une erreur à l'exécution qui dit qu'il y a un conflit entre les champs [indemniteId] et [indemnite]. Avant l'insertion du nouveau champ [indemniteId], lorsqu'on persistait une entité [Employe] dans la table [EMPLOYES], la colonne [INDEMNITES_ID] de cette table était initialisée avec la valeur du champ [Employe.indemnite.id] (clé primaire stockée dans l'objet indemnité). Avec l'insertion du nouveau champ [indemniteId], la colonne [INDEMNITES_ID] peut aussi être mise à jour par la valeur de ce champ. Il y a donc un conflit. On le lève en indiquant par [insertable=false] que la colonne [INDEMNITES_ID] ne doit pas être mise à jour par la valeur du champ [indemniteId] lors d'une opération INSERT. Le même raisonnement s'applique lorsqu'il y a modification d'une entité [Employe] qui va se traduire par une opération SQL UPDATE sur la table [EMPLOYES]. On indique par [updatable=false] que la colonne [INDEMNITES_ID] ne doit pas être mise à jour par la valeur du champ [indemniteId] lors d'une opération UPDATE. L'ajout d'un champ pour la clé étrangère [INDEMNITES_ID] est rendue nécessaire par le mode LAZY appliqué au champ [Indemnite indemnite]. En effet, lorsqu'on récupère des employés par une opération JPQL [select e from Employe e], nous récupérons des employés sans leurs indemnités. Avec le champ [long indemniteId] qui représente la clé primaire de l'indemnité de l'employé, il devient possible de faire une requête pour obtenir l'indemnité complète d'un employé lorsqu'on en a besoin.

Objectif **:** Application « PAM » Architecture en couche utilisant les frameworks spring, hibernate.

1. Objectif

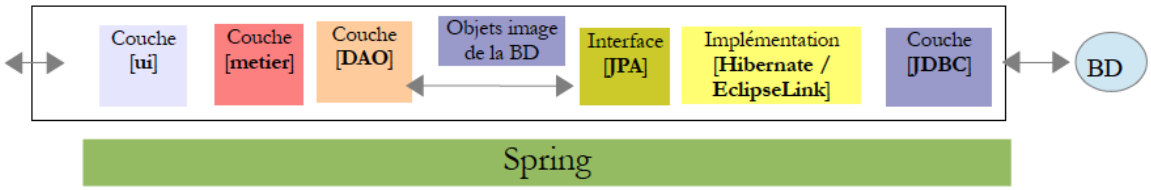
1.1 *Objectif fonctionnel : application console et graphique (avec swing)*

La version « console » consiste en un programme simple « Main » calculant la feuille de salaire dans la sortie console. La version graphique considérée ici sera réalisée avec le framework « swing ».



1.2 *Architecture et organisation des développements*

L'architecture considérée est la suivante (avec hibernate), [ui] correspondant à la console ou à l'interface graphique.



Cette architecture sera implémentée pas à pas, en commençant par les entités associées à l'interface JPA, puis la couche DAO, ensuite la couche métier et enfin les deux variantes de la couche UI.

Le diagramme UML ci-dessous fournit une vue générale de cette architecture en terme de packages, classes et dépendances : chaque couche communique avec les autres (adjacentes) au travers d'interface.

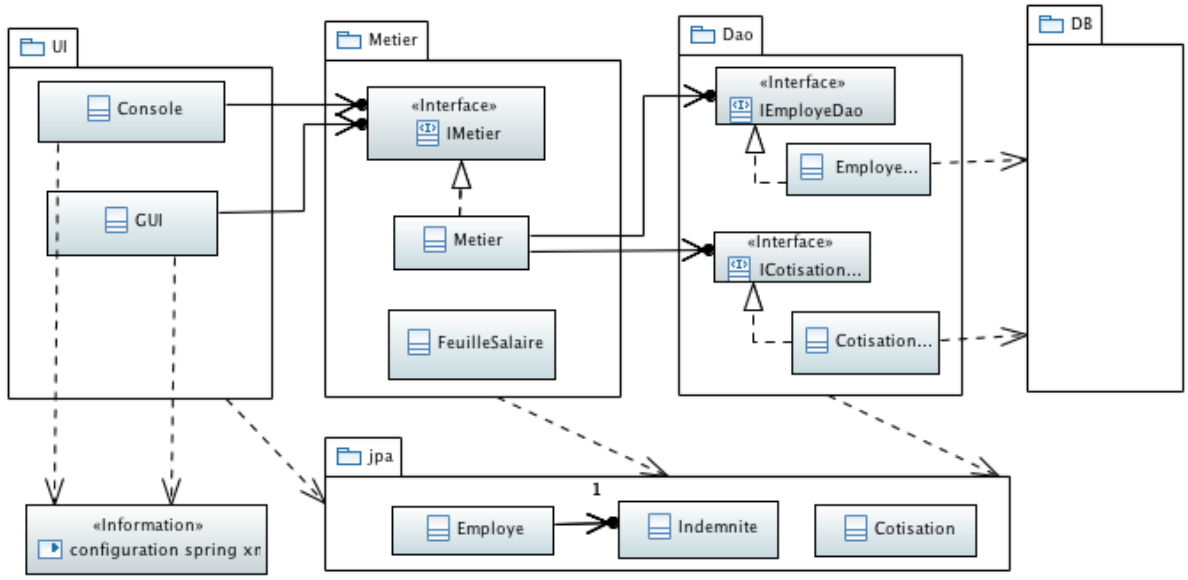


Illustration 1: Vue globale de l'architecture

1.3 *Eléments fournis : Base de données + Projet netbeans (caneva)*

Le fichier [dbpam.sql] fourni permet de créer les tables dans la base MySQL, en utilisant WAMP (importer puis exécuter). On fournit également un caneva de départ [projet mv-pam-spring-hibernate], qu'il faudra compléter.

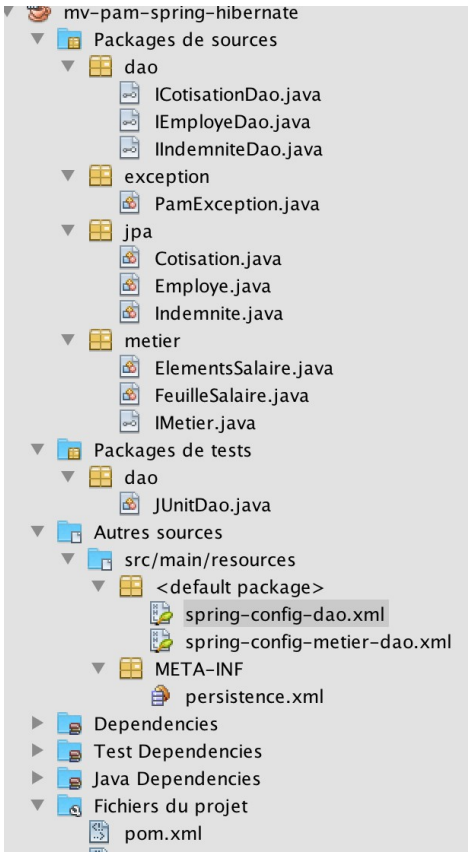
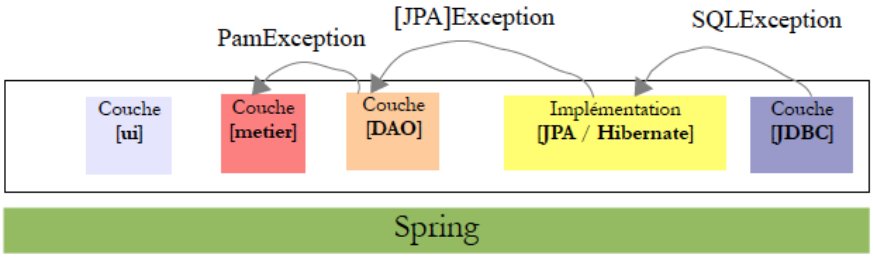


Illustration 2: projet fourni

Quelques commentaires :

- Le fichier pom.xml gère toutes les dépendances aux librairies [hibernate], [spring…], [swing],… etc
- L'exception [PamException] permet de gérer les exceptions comme schématisé par la figure suivante (voir chapitre 4.7.4 de [JEETahe]). Définir sa propre exception de type [RuntimeException] permet d'éviter que le compilateur nous oblige à la déclarer dans la signature des méthodes (ce qui pose des problèmes avec les « beans »).



- Chaque couche correspond à un package spécifique ([metier] et [dao] dans les diagrammes UML), les interfaces étant décrites ci-après.
- Le fichier [spring-config- .xml] (2 versions fournies) configure le projet, incluant notamment l'instanciation de certains objets (les beans liés aux classes), la configuration JPA. A noter que le fichier [persistance.xml] reste nécessaire mais limité en particulier à la déclaration des classes [employe,cotisation,indemnite] associés aux tables de la base de données. Ce fichier [persistance.xml] ne configure plus la connexion avec la base de données (ceci étant intégré au fichier de configuration [spring…-.xml]).
- Le package [jpa] avec les classes [employe,cotisation,indemnite] est fournie, configurées en mode [LAZY]. Nous n'aurons pas à y toucher.

2. *Implémentation de couche DAO*

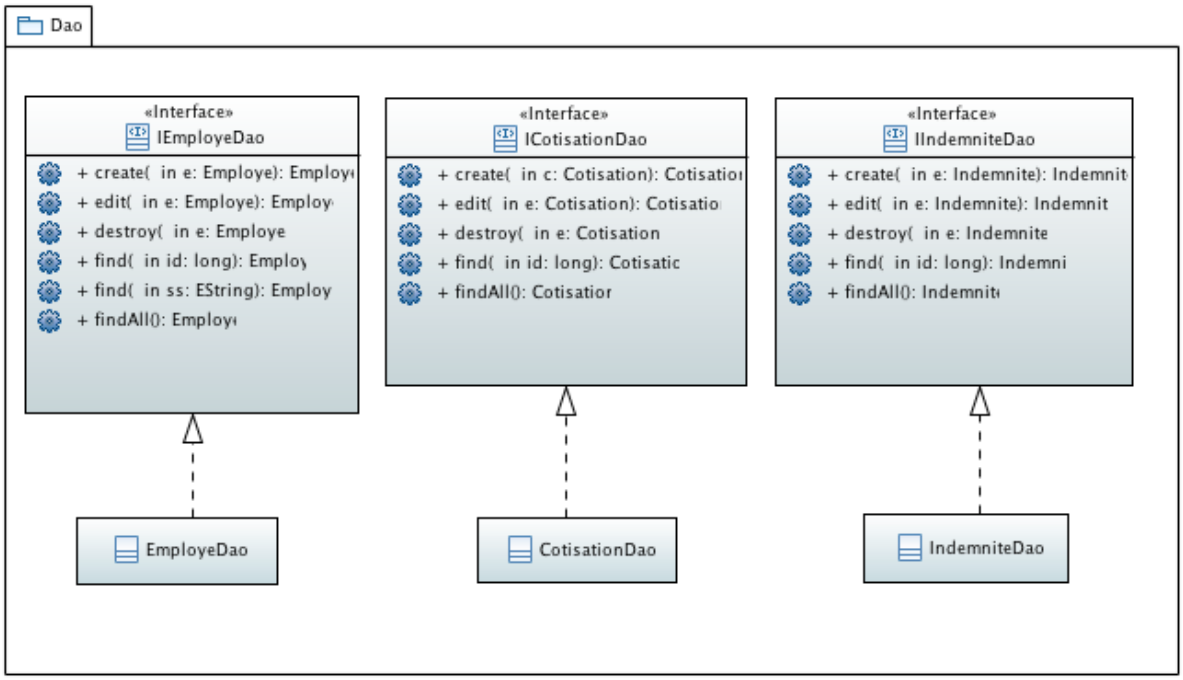


Illustration 3: Diagramme de classe pour la couche DAO

La couche DAO permet de cacher les détails de l'accès à la base de données (par exemple les requêtes JPQL).

Objectif (Lisez attentivement les explications ci-après avant de commencer !!!): L'objectif est d'écrire les classes [CotisationDao, IndemniteDao, EmployeDao] d'implémentation des interfaces [ICotisationDao, IIndemniteDao, IEmployeDao]. Chaque méthode de classe interceptera une éventuelle exception et l'encapsulera dans une exception de type [PamException] avec un code d'erreur propre à l'exception interceptée. Pensez à tester toutes les méthodes depuis le « Main », incluant également la création d'objet et leur sauvegarde dans la base de données (par exemple, la création d'une nouvelle cotisation, qui doit se traduire par l'ajout d'une nouvelle ligne à la table [Cotisations], visible avec WAMP). Attention : n'éditer pas les champs des entités JPA annotés [@Version] → non éditables car permettent le suivi des modifications (géré au niveau de la base – un peu au sens des outils de versionning de type svn git).

Test préliminaire : vérifier que vous parvenez à exécuter le programme principal [MainDao] fourni (vérifier par exemple que le mot de passe déclaré dans les fichiers de configuration [spring.[..].xml] est conforme):

```
//Extrait de [MainDao]
...
ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config-metier-dao.xml");
// CotisationDao
ICotisationDao cotisationDao = (ICotisationDao) ctx.getBean("cotisationDao");
// Recuperation des cotisations en utilisant la couche DAO
List<Cotisation> cotisations = cotisationDao.findAll();
// Affichage des cotisations
for ( Cotisation c : cotisations) { System.out.println("Cotisation: "+c); }
…
//Exemple de sortie console attendue
Cotisation: Cotisations=[id=1, version=1, csgrds=3.49, csgd=6.15, secu=9.39, retraite=7.88]
```

Extrait de [MainDao] et exemple de sortie console attendu

2.1 Explications et recommandations

Méthodes des interface de la couche DAO (cf diagramme UML « couche Dao ») : exemple de ICotisationDao

- La méthode [findAll] retourne toutes les cotisations présentes dans la base de données.
- La méthode [find(id)] retourne la cotisation correspondant la clé primaire [id].
- La méthode [destroy] efface la cotisation (passée en paramètre) de la base de données.
- La méthode [create] ajoute une ligne de la table [COTISATIONS] de la base de données.
- La méthode [edit] met à jour une ligne de la table [COTISATIONS] de la base de données.

Remarque : on constate que les méthodes [create] et [edit] retournent des objets. A priori ceci n'est pas nécessaire car le code qui invoque (code « appelant » - voir ci-dessous) cette API fournit les entités en paramètre, et n'a donc, a priori pas besoin de les récupérer car ce code « appelant » a déjà une référence sur les objets : si l'objet est modifié lors de l'écriture en base de données (e.g. affectation de clé primaire ou modification de la version), le code appelant verra cette modification car il a une référence sur l'objet, et il n'est pas nécessaire de « changer de référence ». Ceci est néanmoins une bonne pratique dans le cas d'un échange de données entre 2 JVM ou par le réseaux : la référence ne sera pas affectée lors de la mise à jour, il est donc requis que la variable ([c] dans notre cas) fasse référence à l'objet retourné (e.g. [c=dao.create(c)]):

```
Cotisation c=new Cotisation(null, 8.15,3.49,7.88,9.39,  1) ;

c=cotisationdao.create(c) ;
```

Exemple de code appelant

Code et annotations : prenons l'exemple de la classe CotisationDao est fournie, avec seulement une méthode implémentée ([findAll()]), utilisée dans le [MainDao] :

```
import org.springframework.transaction.annotation.Transactional;
 @Transactional
 public class CotisationDao implements ICotisationDao
{
        @PersistenceContext
        private EntityManager em;
        // constructeur
        public CotisationDao() {}
        // créer une cotisation
        public List<Employe> findAll()
        {
                try {
                        List<Cotisation> l = em.createQuery("select c from Cotisation c").getResultList();
                        return l;
                    }
                catch (Throwable th) { throw new PamException(th,11); }
}
        ....
}
```

Extrait de la classe [CotisationDao] founie (partiellement implémentée)

- l'annotation Spring @Transactional indique à Spring que chaque méthode doit se dérouler dans une transaction ;
- l'annotation Spring @PersistenceContext injecte dans le champ [em] le gestionnaire du contexte de persistance de la couche JPA. On utilise l'EntityManager pour faire des opérations de persistance (persist, merge, remove, createQuery), depuis les méthodes [create], [edit], [find(..)], [findAll]… Parce que la méthode se déroule dans une transaction, on est assuré qu'à la fin de la méthode, les modifications apportées au contexte de persistance seront synchronisées avec la base de données ; Pour implémenter les interfaces Dao, pensez à lire la documentation : [http://docs.oracle.com/javaee/6/api/javax/persistence/EntityManager.html]).
- on fera en sorte que le code de chaque méthode soit entouré d'un try / catch arrêtant toute éventuelle exception. On encapsulera celle-ci dans un type [PamException].

Fichier de configuration « spring[…].xml » :

- L'instance « cotisationDao » est une implémentation de l'interface « ICotisationDao », dont le [Main] ne dépend pas : [import dao. ICotisationDao ;] et non [import dao. CotisationDao ;]. L'implémentation utilisée est déclarée dans le fichier de configuration [spring-….xml] : [<bean id="cotisationDao" class="dao.CotisationDao"/>].
- L'idée de ce type d'architecture est qu'une couche dépend d'une autre couche à travers l'interface et non l'implémentation : le choix de l'implémentation est définie dans un fichier de configuration et non dans du code « en dur ».

2.2 Travail à faire :

Travail à faire (1) : Implémenter les méthodes de la classe CotisationDao (seule [findAll]) est correctement implémentée. Pour les différentes méthode ([create], [edit], [find(..)], [findAll]), on utilisera l'EntityManager. Plus précisément, on utilisera les méthodes [persist], [merge], [remove] et enfin [createQuery] (requête JPQL). Pensez à lire la documentation :

[http://docs.oracle.com/javaee/6/api/javax/persistence/EntityManager.html]).

Remarque : pour effacer une cotisation de la table il faut (dans la méthode [destroy]) d'abord synchroniser l'objet (méthode merge de [EntityManager]).

Pour tester le bon fonctionnement, vous pourrez, depuis le [MainDao] exécuter des opérations de création, modification, effacement et consulter l'effet sur la base (WAMP). Par exemple :

//Code commenté extrait de [MainDao] Cotisation c = new Cotisation(null, 8.15,3.49,7.88,9.39, 1); c = cotisationDao.create(c);
Test de l'API de CotisationDao

Travail à faire (2) : Implémenter [IndemniteDao], en vous inspirant de CotisationDao, sans la tester. Remarque : attention à la colonne [INDICE] qui est non nulle et unique (erreur si vous essayer de créer une nouvelle indemnité d'indice déjà existant).

Travail à faire (3) : Implémenter [EmployeDao]. Dans ce cas, on remarque une nouvelle méthode à implémenter : [find(ss)] qui retourne l'employé dont le numéro de sécurité social est [ss]. A noter le cas particulier du mode [LAZY] (passer en mode « LAZY » : [@ManyToOne(optional = false,fetch=FetchType.LAZY)] à l'attribut [private Indemnite indemnite] de [Employe]:

- la méthode [employeDao::findAll()] retourne les employés **sans** les indemnités : dans notre cas, ceci permet de consulter la liste des employés de la table [EMPLOYES] sans rapatrier toutes les informations « annexes » (indemnité de la table [INDEMNITES]), et ainsi optimiser l'application.
- les méthodes [employeDao::find(String ss)] et [employeDao::find(Long id)] retourne les employés **avec** les indemnités (requête JPQL avec jointure) : ceci permettra de calculer la feuille de salaire, qui requiert en particulier les indemnités.

3. *Implémentation de la couche Metier*

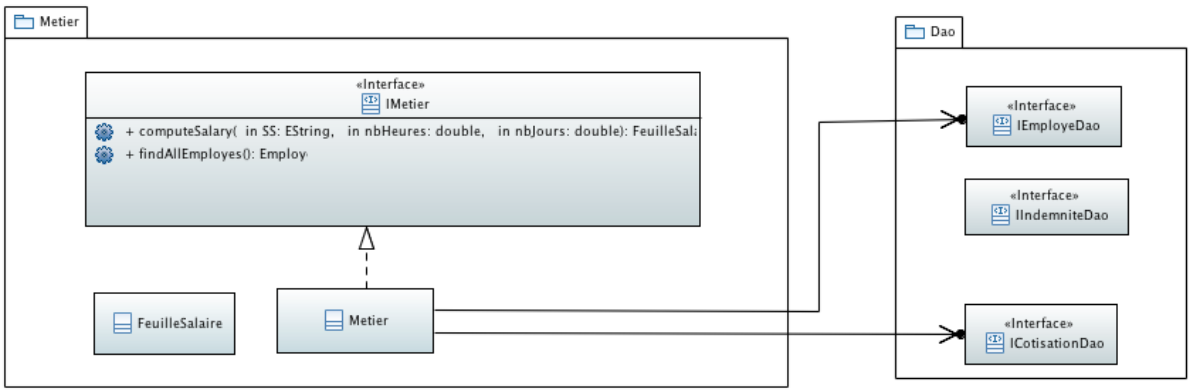


Illustration 4: Diagramme de classes de la couche "Métier"

L'objectif est d'implémenter l'interface de [IMetier] : il s'agit de la classe [Metier] dont la structure est la suivante (voir également le diagramme UML « métier »):

```
@Transactional
public class Metier implements IMetier {
// références sur la couche [dao]: par configuration (properties) du bean "metier" dans spring-...-config.xml
private ICotisationDao cotisationDao = null;
private IEmployeDao employeDao = null;
// obtenir la feuille de salaire
@Override
public FeuilleSalaire calculerFeuilleSalaire(String SS, double nbHeuresTravaillees, int nbJoursTravailles)
{
try        {   // à coder }
catch (Throwable th) {   throw new PamException("Employe de numero SS "+ SS + " not found",101);  }
}
// liste des employés
@Override
public List<Employe> findAllEmployes() { // à coder }
// getters et setters (requis pour renseigner « automatiquement » les attributs privés)
public void setCotisationDao(ICotisationDao c) {this.cotisationDao=c;}
public void setEmployeDao(IEmployeDao e) {this.employeDao=e;}
...
}
```

Structure de la classe [Metier] à implémenter

Remarque : les attributs [cotisationDao] et [employeDao] sont assignés de manière « transparente » (« automatiquement ») à l'instanciation de l'objet de type [Metier], car cette classe est déclarée dans le fichier [spring….xml] sous forme d'une « bean » avec les propriétés « property » appropriée :

```
<!-- couches applicatives metier -->
<bean id="metier" class="metier.Metier">
   <property name="employeDao" ref="employeDao"/>
   <property name="cotisationDao" ref="cotisationDao"/>
</bean>
```

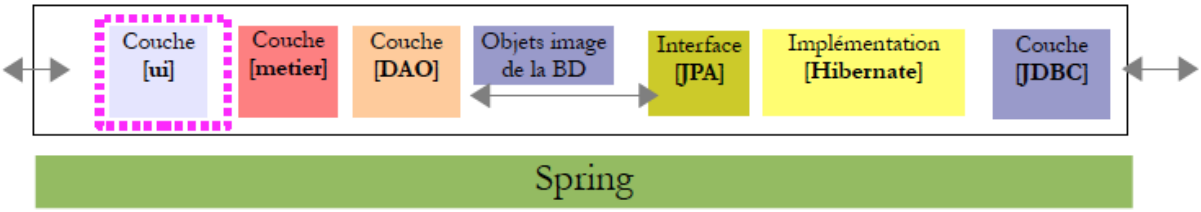
Extrait du fichier de configuration [spring…. xml]

Travail à faire : écrire le code de les méthodes [findAllEmployes] et [calculerFeuilleSalaire] de la classe [Metier]. [calculerFeuilleSalaire] retourne un objet de type [FeuilleSalaire] (code fourni) qui stocke simplement les différentes informations liées au salaire (l'employé, les cotisations, les éléments de salaire tels que le salaire de base, le salaire net, les indemnité de repas,...etc). Pour rapidement tester votre code, faire un [MainMetier] (package ui.console) de la forme suivante :

```
// instanciation couche [metier]
ApplicationContext ctx = new ClassPathXmlApplicationContext("...");
IMetier metier = (IMetier) ctx.getBean("metier");
// calcul de la feuille de salaire
feuilleSalaire=metier.calculerFeuilleSalaire("254104940426058", 150, 3);
```

Extrait du [MainMetier] de test : le salaire net retourné dans ce cas est **280.3685** euros

4. Couche UI de l'application en mode graphique



Travail à faire : implémenter la classe [ui.swing.PamSwing] de manière à obtenir (au moins) l'interface graphique (voir section 1.2). Remarque : un caneva de départ est fourni (obtenu, depuis netbeans, en faisant : [NewFile]→[Swing GUI Form]-→ [JFrame Form]). Le diagramme UML ci-après donne une vue partielle de la classe : vous devez, en particulier implémenter la méthode [doMyInit], qui vous permettra d'initialiser les éléments graphiques (e.g. ComboxBox), qui auront été placé « graphiquement » depuis netbeans. [doMyInit] initialisera également l'application (« xml » invoqué depuis le [MainMetier]).

Documentation : chapitre 4.11 de [JEETahe] & https://netbeans.org/kb/docs/java/quickstart-gui.html

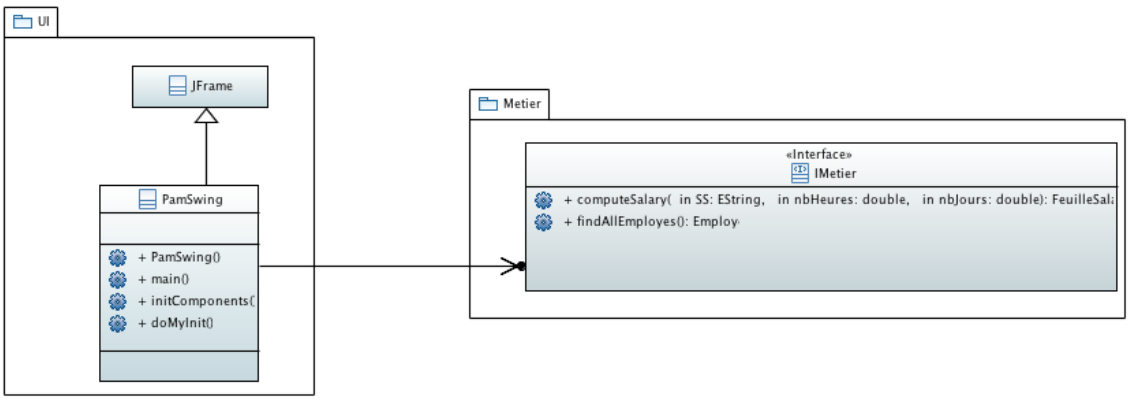


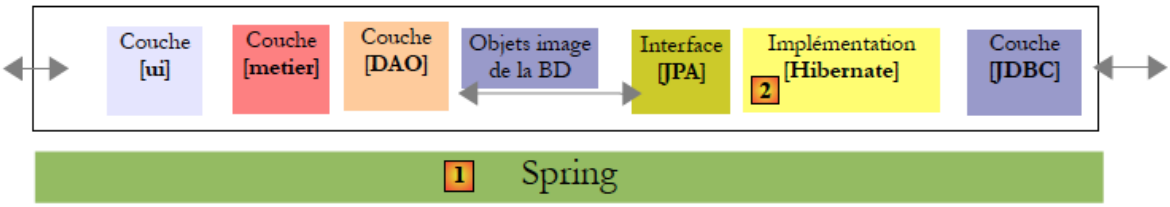
Illustration 5: Diagramme de classe pour "GUI"

***Objectif* : Application « PAM » : Portage de [spring, hibernate] vers [openEJB, EclipseLink]**

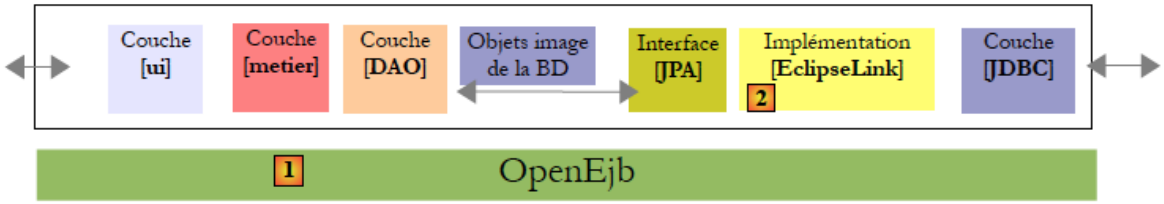
1. Objectif

Il s'agit de porter l'application précédente [spring, JPA/Hibernate] vers [openEJB,JPA/EclipseLink]. Au terme du portage, les « codes » suivants devront fonctionner (après une éventuelle adaptation) : ui.console.Main et ui.swing.PamSwing

L'implémentation actuelle est :



Les deux implémentations à construire avec OpenEJB / EclipseLink :



Version 1 : cas d'une couche [ui] qui est un client local de [metier]

2. *Projet caneva fourni : explications*

2.1 *Test préliminaire :*

Test préliminaire : Exécuter le programme principal [MainExemple] et vérifier vous parvenez à instancier un objet implémentant l'interface Iexemple, et accédant à la base de données (configuration conf/openejb.conf)

In methode maMethode ... Test acces table cotisations: ID=1, CSGD=6.15
[MainExemple] fourni : sortie attendue

2.2 *Fichiers de configuration*

Fichier [POM.xml] Librairies nécessaires, principalement « org.apache.openejb » et « eclipseLink »

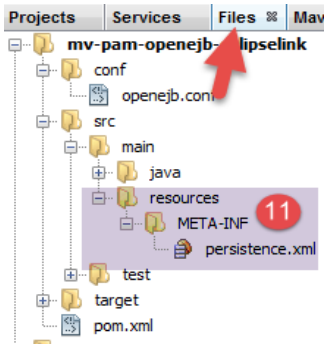
Fichier [persistance.xml] Le fichier « persistance.xml » est le suivant :

<persistence-unit name="..." transaction-type="**JTA**"> … </persistence-unit>
Les transactions dans un conteneur EJB sont de type JTA (Java Transaction API), et non plus de type RESOURCE_LOCAL (Spring).

Fichier de configuration [conf/conf/openejb.conf] : Les caractéristiques JDBC de la source de données JTA utilisée par le conteneur OpenEJB seront précisées par le fichier de configuration suivant :

```
<?xml version="1.0"?>
<openejb>
   <Resource id="Default JDBC Database">
            JdbcDriver com.mysql.jdbc.Driver
            JdbcUrl jdbc:mysql://localhost:3306/dbpam
            UserName root
            Password root
   </Resource>
</openejb>
```

Fichier « openejb.conf »



Contrairement à [spring], les « beans » ne sont pas déclarées dans un fichier de configuration (c'était le cas avec spring, à travers le fichier [spring-config.xml] : avec openEJB, tout se fait au niveau du code

2.3 *Les entités sérializables*

Les couches [ui] et [metier] vont s'échanger des objets. Dans la pratique, ces deux couches sont dans deux JVM différentes. Si la couche [ui] veut passer un objet à la couche [métier], elle ne peut pas passer la référence de cet objet. La couche [métier] ne peut en effet référencer des objets qui ne sont pas dans sa JVM. La couche [ui] va alors passer la valeur de l'objet et non sa référence. On appelle cela la sérialisation de l'objet. La couche [métier] va recevoir cette valeur et va créer un nouvel objet à partir d'elle (« désérialisation » de l'objet). Les couches [ui] et [métier] ont alors 2 objets identiques, chacun dans sa JVM.

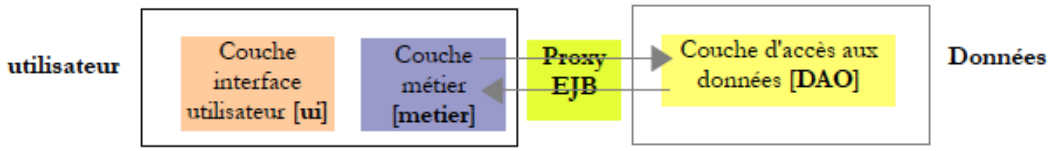
Dans notre exemple, les types suivants peuvent être échangés entre les couches [ui] et [metier] : **[Employe, Cotisation, Indemnite, FeuilleSalaire, ElementsSalaire, PamException]**. Ces classes doivent pouvoir être sérialisées. Cela est obtenu par la simple déclaration (il n'y a rien d'autre à faire): **« public [Classe] extends ... implements Serializable »**

2.4 *La classe [PamException]*

La classe [PamException] reste ce qu'elle était à un détail près :

```
@ApplicationException(rollback=true)
public class PamException extends RuntimeException implements Serializable
```

Modification à apporter à la classe [PamException]



Lorsque la couche [metier] appelle une méthode M de la couche [DAO], cet appel est intercepté par le conteneur EJB. Tout se passe comme s'il y avait une classe intermédiaire entre la couche [metier] et la couche [DAO], appelée ici[Proxy EJB], interceptant tous les appels vers la couche [DAO]. Lorsque l'appel à la méthode M de la couche [DAO] est interceptée, le Proxy EJB démarre une transaction puis passe la main à la méthode M de la couche [DAO] qui s'exécute alors dans cette transaction. La méthode M se termine avec ou sans exception.

- si la méthode M se termine sans exception, l'exécution revient au proxy EJB qui termine la transaction en la validant par un commit . Le flux d'exécution est ensuite rendu à la méthode appelante de la couche [metier] ;
- si la méthode M se termine avec exception, l'exécution revient au proxy EJB qui termine la transaction en l'invalidant par un rollback . De plus il encapsule cette exception dans un type EJBException . Le flux d'exécution est ensuite rendu à la méthode appelante de la couche [metier] qui reçoit donc une EJBException. L'annotation @ApplicationException empêche cette encapsulation. La couche [metier] recevra donc une PamException. De plus, l'attribut rollback=true indique au proxy EJB que lorsqu'il reçoit une PamException , il doit invalider la transaction ;

2.5 EJB Local et distant : cas de IExemple

Les classes implémentant les couches ([DAO] et [Métier] dans notre cas) deviennent des EJB. L'EJB va implémenter cette même interface sous deux formes différentes : une locale et une distante. L'interface locale peut être utilisée par un client s'exécutant dans la même JVM, l'interface distante par un client s'exécutant dans une autre JVM. Voir ci-dessous pour la déclaration d'un EJB en « local » ([ExampleA]) et « remote » ([ExampleB]):

`import javax.ejb.Local;`
`@Local`
`public interface IExampleA { }`
	`import javax.ejb.Local;`
	`@Remote`
	`public interface IExampleB { }`

`@Stateless`
`@TransactionAttribute(TransactionAttributeType.REQUIRED)`
`public class ExampleA implements IExampleA { }`
	`@Stateless`
	`@TransactionAttribute(TransactionAttributeType.REQUIRED)`
	`public class ExampleB implements IExampleB { }`

Déclaration EJB local	Déclaration EJB remote

- l'annotation @Local déclare une interface locale pour l'EJB qui l'implémentera
- l'annotation @Remote déclare une interface distante pour l'EJB qui l'implémentera.
- l'annotation @Stateless fait de la classe [CotisationDao] un EJB (à la fois local et distant) et l''annotation @TransactionAttribute qui fait que chaque méthode de la classe s'exécutera au sein d'une transaction.

Les EJB peuvent être récupérés depuis avec un code du type (voir [MainExemple] fourni) :

```
Properties properties = new Properties();
properties.setProperty(Context.INITIAL_CONTEXT_FACTORY, "org.apache.openejb.client.LocalInitialContextFactory");
InitialContext initialContext = new InitialContext(properties);
```

```
IExempleA a = (IExempleA) initialContext.lookup("ExempleALocal");
….
```

Exemple de code (similaire au cas Spring) permettant de récupérer l'EJB « local »

Remarque : En exécutant ce code, on peut voir dans la sortie console les informations suivantes :

INFOS - Jndi(name=**ExempleALocal**) --> Ejb(deployment-id=**ExempleA**)

INFOS - Jndi(name=global/classpath.ear/pam_opene/Exemple!exemple.IExempleALocal) --> Ejb(deployment-id=Exemple)

Le JNDI (Java Naming and Directory Interface) : Cette API fournit une interface unique pour utiliser différents services de nommages ou d'annuaires et définit une API standard pour permettre l'accès à ces services. Dans notre cas, cela permet d'instancier (« récupérer ») les classes embarquées dans notre « EJB » à partir d'une chaîne de caractère (sorte d'annuaire qui associe un nom à une classe). Pour plus d'informations, on pourra se référer à :

https://www.jmdoudoux.fr/java/dej/chap-jndi.htm ou http://docs.oracle.com/javase/jndi/tutorial/ ou

http://www.oracle.com/technetwork/java/overview-142035.html

Remarque : on peut également implémenter les deux interfaces [Local] et [Remote] comme illustré par l'exemple fourni pour [ExampleLR], par héritage en diamant.

3. *Travail à faire : portage de la couche DAO en « Local »*

- Reprendre les contenus des packages [jpa] (copier-coller depuis le projet [spring-hibernate]), et rendez les classes sérializables (voir explications précédentes).
- Reprendre les contenus des packages [dao] (copier-coller depuis le projet [spring-hibernate]), et adapter les classes.
- Ecrire un programme [MainDaoLocal] affichant les contenus des différentes tables [cotisations, employes et indemnité].

4. *Travail à faire : portage de la couche Metier en « remote »*

Le principe est le même que pour la couche dao. La seule différence est que la classe [Metier] a deux attributs sur les classes de la couche dao (ici couche « remote »):

```
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class Metier implements IMetier
{
@EJB
private ICotisationDao cotisationDao = null;
@EJB
private IEmployeDao employeDao = null;
...
```

L'EJB [Metier] de la couche metier implémente les interfaces « locale » et « distante »

Au final, lorsque l'EJB [Metier] sera instancié, les attributs seront initialisés avec des références sur les interfaces locales des deux EJB de la couche [DAO]. On fait donc l'hypothèse que les couches [metier] et [DAO] s'exécuteront dans la même JVM.

Travail à faire : Implémenter la couche metier. Pour cela, commencer par copier le package [metier] de la version « spring » de l'application, puis adapter les classes.

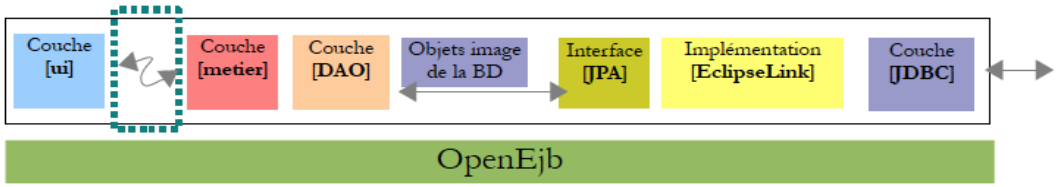
5. *Interface ui graphique avec Swing*

Travail à faire *:* porter la classe [ui.swing.PamSwing] de l'application « spring ».

Objectif **:** Application « PAM » **:** Portage sur un serveur d'applications

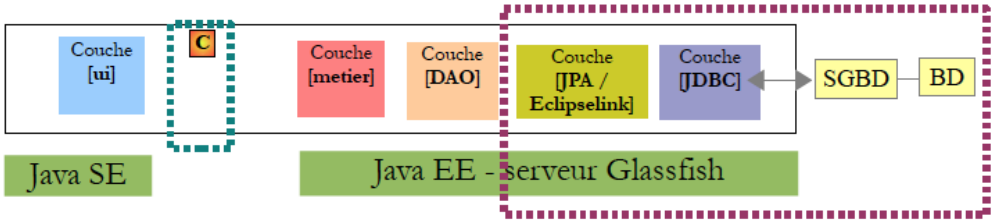
1. Objectif

On se propose de placer les EJB des couches [metier] et [DAO] dans le conteneur d'un serveur d'applications Glassfish. Dans l'application actuelle (voir figure), la couche [ui] utilise l'interface distante de la couche [metier] (nous avons également testé la version « locale »).



Application PAM actuelle

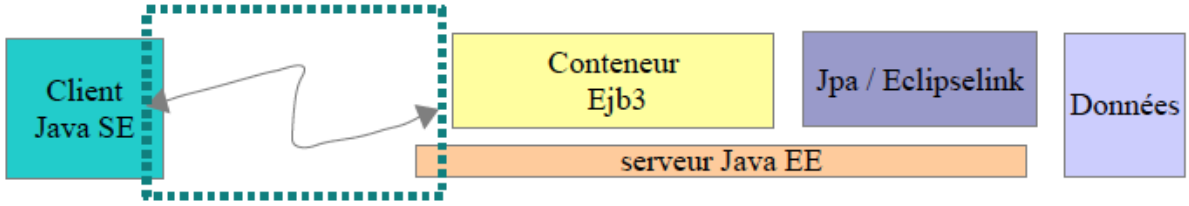
Dans ce mode distant, la couche [ui] était cliente de la couche [metier], couche implémentée par des EJB. Pour fonctionner en mode client / serveur, dans lequel le client et le serveur s'exécutent dans deux JVM différentes :



Architecture attendue

- la couche [ui] s'exécutera dans un environnement Java SE (Standard Edition)
- les couches [metier, DAO, JPA] s'exécuteront dans un environnement Java EE, sur un serveur Glassfish ;
- le client communiquera avec le serveur via un réseau tcp-ip avec le protocole RMI (Remote Method Invocation), utilisable uniquement entre deux applications Java. Le client et le serveur s'échangent des objets sérialisés pour communiquer et non des références d'objets.

la partie serveur qui sera hébergée par le conteneur EJB3 du serveur Glassfish :



Il s'agit de faire un portage vers le serveur Glassfish de ce qui a déjà été fait et testé avec le conteneur OpenEJB. C'est là l'intérêt de OpenEJB et en général des conteneurs EJB embarqués : ils nous permettent de tester l'application dans un environnement d'exécution simplifié. Lorsque l'application a été testée, il ne reste plus qu'à la porter sur un serveur cible

L'objectif sera de :

- Porter l'application PAM précédente : cela consiste en un nouveau projet type « EJB », le « copié-collé » des package [metier] [dao] et [jpa] du projet PAM du TD précédent.
- Implémenter le « client » dans un autre projet (2 types de client dans notre cas)

2. Exemple minimaliste fourni

Un exemple d'application « simple » est fourni :

- Script sql : permet de créer et peupler une base de données
- Projet « ejb-persons » : module EJB à déployer sur « glassfish » (il suffit de l'exécuter → déploiement)
- Projet « client » : projet maven/java application. Exécuter le programme [MainRemoteDao], la sortie attendue est :

```
jpa.Person[ id=1, nom=Lemonnie,age=38.0 ]
jpa.Person[ id=2, nom=Fernandi,age=42.0 ]
```

Sortie console résultant de l'exécution de MainRemoteDao

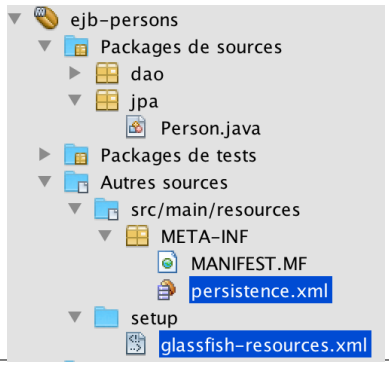
3. Partie serveur

3.1 *Préparation du projet*

- Créer un projet **[pam-ejb-glassfish]** de type « **Maven / EJB module** », en sélectionnant [JEE7] et [Glassfish].
  - On notera (POM.xml) que le packaging est de type ejb (et non plus jar).
  - On notera la dépendance [java-ee] (« POM.xml »). L'attribut [provided] signifie que la dépendance [java-ee] ne sera pas incluse dans l'archive jar du projet. A l'exécution, elle sera trouvée dans les archives du serveur Glassfish.
  - Vérifier que votre [POM.xml] est similaire au [POM.xml] de l'exemple fourni (sauf nom du projet !), en particulier les numéros de version des dépendances (e.g. eclipseLink).
- Configuration de la couche persistance / accès à la base de données : fichiers [persistence.xml] et [**glassfish-resources.xml**] (spécifique au déploiement sur le serveur glassfish). Ajouter ces deux fichiers à votre projet, en vous inspirant de l'exemple fourni qu'il faudra adapter (notamment connection à la base de données). En particulier, le fichier de configuration **glassfish-resources.xml** définit la source de données exploitée par la couche JPA avec la source JDBC gérée par le serveur Glassfish. Ces deux fichiers sont de la forme :

//glassfish-resources.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 3.1
Resource Definitions//EN" "http://glassfish.org/dtds/glassfish-resources_1_5.dtd">
<resources>
  <jdbc-resource enabled="true" jndi-name="jdbc/dbpam" object-type="user" pool-
name="connectionPool">
    <description/>
  </jdbc-resource>
  <jdbc-connection-pool … >
    <property name="URL" value="jdbc:mysql://localhost:3306/dbpam"/>
    <property name="User" value="root"/>
    <property name="Password" value="root"/>
  </jdbc-connection-pool>
</resources>
```



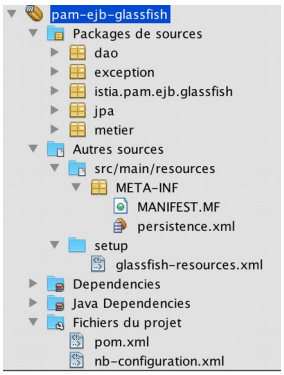
//persistence.xml

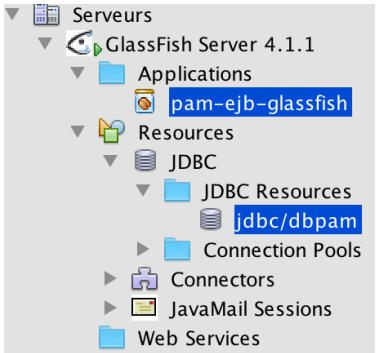
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence ...">
<persistence-unit name="istia_pam-ejb-glassfish_ejb_1.0-SNAPSHOTPU" transaction-type="JTA">
        <jta-data-source>jdbc/dbpam</jta-data-source>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
                <property name="eclipselink.weaving" value="static"/>
                <property name="eclipselink.weaving.lazy" value="true"/>
                <property name="eclipselink.weaving.internal" value="true"/>
                <property name="eclipselink.logging.level" value="FINE"/>
        </properties>
</persistence-unit>
</persistence>
```

« glassfish-resources.xml » : le nom JNDI de la ressource JDBC créée est utilisé dans le fichier [persistence.xml] pour désigner la source de données que l'implémentation JPA doit utiliser ;

3.2 *Intégration des couches [JPA] [Dao] [Metier] et de l'exception [PamException]*

Il s'agit de faire une simple copie des packages JPA/Exception/DAO/Metier depuis le projet du TD précédent vers ce nouveau projet → il ne doit pas y avoir d'erreur. Le contenu du projet doit maintenant être le suivant :





3.3 *Déploiement sur le serveur glassfish*

- On démarre le serveur « glassfish » depuis netbeans : « services → serveurs → glassfish server → bouton droit → start
- On déploie le projet [pam-ejb-glassfish] : « nettoyer compiler » puis « exécuter » → [pam-ejb-glassfish] doit apparaître dans « services / serveurs / GlassFish Server / Applications » (voir figure ci-dessus).

Lors du déploiement, la sortie console doit afficher des informations du type :

Infos: Portable JNDI names for EJB Metier: [java:global/pam-ejb-glassfish/Metier]
Extrait de la sortie console : on observe que la couche métier est accessible. **Note :** les noms portables JNDI des EJB déployés sont reconnu par tous les serveurs Java EE (supérieur à la version 5).

4. Client version 1 [ClientRMI1] – en vous inspirant de [ClientPersonRMI1]

Le client utilisera la couche [Metier] distante (accessible via le serveur Glassfish), ainsi que les entités [PamException], [Employe], [Cotisation] et [Indemnite]. Après création du projet de type « Maven / Java Application » : [ClientConsole1]

- **Travail à faire :** reprendre le fichier POM.xml de l'exemple fourni, gérant 3 dépendances:
  - Dépendance au module EJB qui sera déployé sur le serveur : **adapter le POM.xml afin de remplacer la dépendance au projet EJB [ejb-persons] par la dépendance au projet [pam-ejb-glassfish].** Vous pouvez procéder depuis netbeans par [bouton droit sur « dependencies » → « add dependancies » → « projets ouverts »]
  - Dépendance à EclipseLink.
  - Dépendance nécessaire à tous les clients Glassfish (Artefact « gf-client »): requis pour l'accès aux EJB via leurs noms JNDI. A noter que la balise [repository] permet de spécifier d'autres dépôt pour gérer les dépendances: dans notre cas, sans spécifier ce nouveau dépôt, une dépendance vers EclipseLink nécessitée par la dépendance à [gf-client] ne serait pas trouvée dans le dépôt [http://maven.glassfish.org/content/groups/glassfish], mais dans des dépôts spécialisés.
- **Travail à faire :** Créer un programme [MainMetier] utilisant la couche [metier], permettant d'instancier un objet [Metier] et calculer le salaire d'un employé (voir exemple fourni).

```
public static void main(String[] args) throws NamingException
{
...
}
```

Extrait du [MainMetier] de test

Le [throws NamingException] permet de laisser remonter une exception de type [NamingException] sans l'intercepter, permettant de se focaliser sur d'autres exceptions, plus « claires ». En cas de problème, pensez à notamment vérifier que vos classes sont [Serializable] et pensez à regarder la sortie de log du serveur GlassFish.

- **Travail à faire :** implémenter dans ce projet le client graphique « swing », en adaptant la classe [PamSwing] codée lors d'un TD précédent. Pour cela, n'oubliez pas d'ajouter la dépendance à « swing » dans votre « pom.xml »

5. Client distant version 2 [ClientRMI2] - en vous inspirant de [ClientPersonRMI2] (optionnel)

Dans le cas précédent, on ne peut pas facilement configurer, à l'aide d'un fichier de configuration, les détails de la machine JNDI et le port d'écoute de celui-ci. Pour cette raison, nous proposons d'utiliser « spring », rendant plus explicite cette configuration. En vous inspirant de l'exemple [ClientPersonRMI2], créer votre projet « Java Application / Maven » et faites les ajustements suivants concernant

- Le fichier de configuration spring initial étant le suivant, pour récupérer l'objet de type IPersonDao

```
<!-- métier -->
<jee:jndi-lookup id="persondao" jndi-name="java:global/ejb-persons/PersonDao">
   <jee:environment> …    </jee:environment>
</jee:jndi-lookup>
```

Fichier spring-config-client.xml à adapter

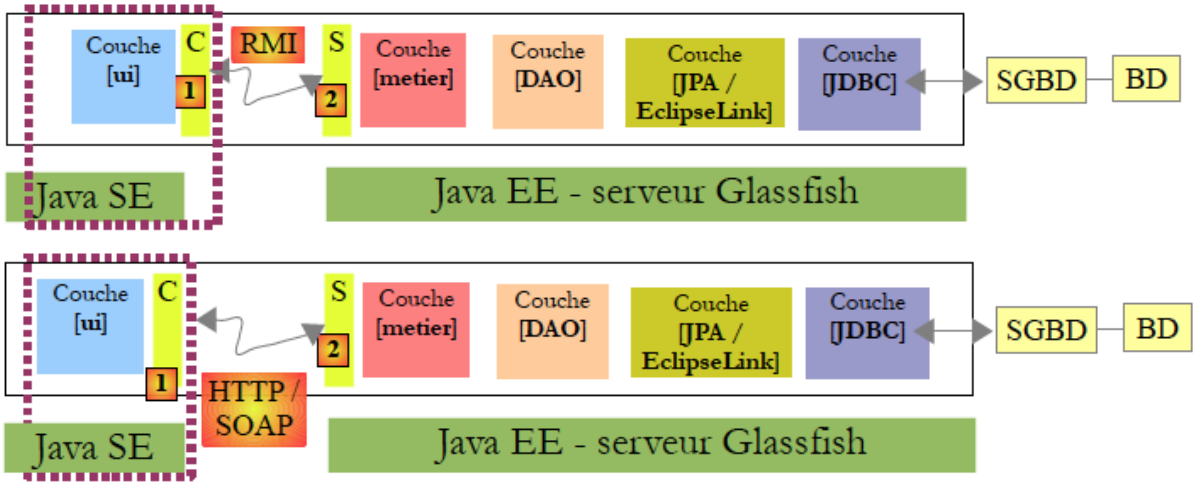
- POM.xml : ajoutez la dépendance au module EJB [pam-ejb-glassfish] (comme pour le premier client)
- Adapter le programme console pour calculer le salaire d'une assistante maternelle.

Travail à faire : implémenter dans ce projet le client graphique « swing », en adaptant la classe [PamSwing] codée lors d'un TD précédent. Pour cela, n'oubliez pas d'ajouter la dépendance à « swing » dans votre « pom.xml ».

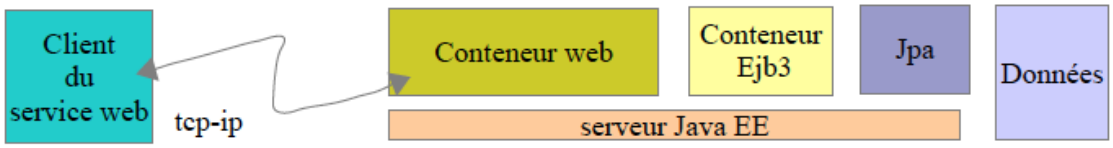
Objectif **:** Application « PAM » **:** client serveur dans un architecture de service web SOAP (chapitres 7.1 et 7.2 de [JEETahe])

1. Objectif

Nous allons maintenant remplacer la couche de communication [C, RMI, S] par une couche [C, HTTP / SOAP, S] :



Le protocole HTTP / SOAP a l'avantage sur le protocole RMI / EJB précédent d'être multi-plateformes. Ainsi le service web peut être écrit en Java et déployé sur le serveur Glassfish. Le client lui, pourrait être un client .NET ou PHP. Nous allons développer cette architecture selon le modes suivant : le service web sera assuré par une application web utilisant l'EJB [Metier] ; Dans ce cas un service web peut être implémenté par une classe annotée @WebService qui utilise un EJB :



Un exemple élémentaire complet est fourni :

- Importer la base dans wamp [dbpersons.sql] (TD5)
- Ouvrez le projet [ejb-persons] (à être exécuter dans « conteneur ejb3 » - voir figure – caneva TD5) : compilez-le
- Ouvrez le projet [webservice-persons] (à être exécuter dans « conteneur web » - voir figure) : compilez-le
- Ouvrez le projet [webapp-persons] : combine les 2 précédents projets. Ouvrir le « sous-projet » [webapp-persons / Modules / webapp-persons-ear] → compilez-le et exécutez-le (déploiement sur serveur glassfish) - « Hello world » doit apparaître dans votre navigateur.
- Ouvrez le projet [ClientWebAppPerson] : compilez-le et exécuter [MainRemoteDao] → les noms et âges des personnes doivent apparaître dans la sortie console : **[Lemonnie 38.0, Fernandi 42.0]**
- Testez le client Python ClientSOAP ([pip install zeep] au préalable, depuis « Anaconda prompt » en « admin »):

```
#zeep : l'objet client dispose automatiquement des methodes associees aux webservices wsdl [findAll()]
import zeep
client = zeep.Client("http://localhost:8080/webservice-persons/PersonWsEjbService?wsdl")
print(client.service.findAll())
```

2. Partie serveur : projet webservice

- Ne pas oublier de peupler la base de données [dbpam] (WAMP).
- Créer un nouveau projet de type [Maven / Web Application], avec le nom [pam-webservice-glassfish].
- Ajouter la dépendance (avec le **scope provided**) à [pam-ejb-glassfish] (TD5) : application PAM sous forme d'EJB. Ceci se fait par « bouton droit sur Dependencies / add dependency ». Cet EJB (« provided ») sera fourni par l'environnement d'exécution (i.e. sur le serveur glassfish). Ceci se traduit par l'ajout d'un nouvel élément au POM.xml :

```
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>pam-ejb-glassfish</artifactId>
```

```
        <version>${project.version}</version>
        <scope>provided</scope>
    </dependency>
```

- Ajouter la seule classe du projet [PamWsEjbMetier] (en faisant [add web service]), qui sera associée au webservice : [Packages de sources / pam.ws / PamWsEjbMetier]. Elle fait le lien entre le webservice et la couche [Metier] de l'EJB. **Vous pouvez vous inspirer du webservice de l'exemple fourni**. Il suffit que cette classe soit taguée @Webservice, avec deux méthodes [calculerFeuilleSalaire] et [findAllEmployes] ;

```
@WebService
public class PamWsEjbMetier {
@EJB
private IMetier metier;

public FeuilleSalaire calculerFeuilleSalaire(String SS, double nbHeuresTravailles, int nbJoursTravaills)
{  try{ return metier.calculerFeuilleSalaire(SS, nbHeuresTravailles, nbJoursTravaills); }
   catch (Exception ex) {
      System.err.println(String.format("L'erreur suivante s'est produite autre : %s", ex));
      return null; }
}

public List<Employe> findAllEmployes() {
   try{ return metier.findAllEmployes(); }
   catch (Exception ex) {
      System.err.println(String.format("L'erreur suivante s'est produite autre : %s", ex));
      return null; }
}}
```

Classe PamWsEjbMetier

L'annotation **@WebService** fait de la classe [Metier] un service web. Un service web expose des méthodes à ses clients. Celles-ci peuvent être annotées par l'attribut **@WebMethod**, mais c'est facultatif. Par défaut, les méthodes publiques d'une classe annotée par [**@WebService**] deviennent automatiquement des méthodes du service web. Il est important que les getters et setters soient supprimés sinon ils seront exposés dans le service web et cela cause des erreurs de sécurité : on pourrait obtenir/modifier les accès à la couche [dao] et donc aux données.

3. Partie serveur : projet webservice + projet ejb = projet entreprise application

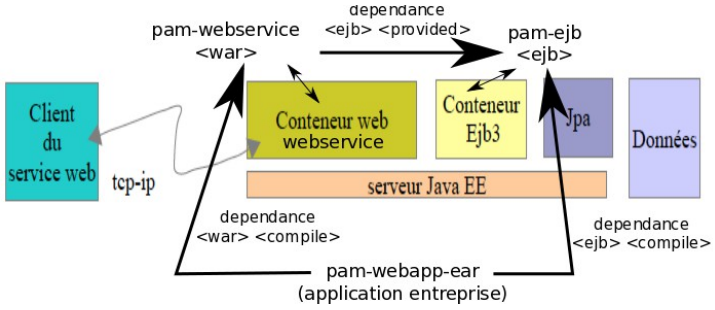
Pour déployer le service web sur le serveur Glassfish, il nous faut à la fois déployer :

- le module web [pam-webservice-glassfish] dans le conteneur web du serveur ;
- le module EJB [pam-ejb-glassfish] dans le conteneur EJB du serveur ;

Pour cela, nous avons besoin de créer une application de type [Enterprise Application] qui va déployer les deux modules en même temps. Pour ce faire, il faut que les deux projets soient chargés dans Netbeans. Ceci fait, nous créons un nouveau projet de type [Maven / Enterprise Application], que l'on appelera [pam-webapp] : **on décoche les options [Create EJB Module] et [Create Web App Module] : ils sont déjà créés et chargés dans netbeans.** Cela créé 2 projets dans netbeans [pam-webapp] et [pam-webapp-ear] (en pratique, le répertoire [pam-webapp-ear] est inclus dans le répertoire [pam-webapp]).

On ajoute à [pam-webapp-ear] :

- la dépendance au module web [pam-webservice-glassfish], en précisant le type [war] et le « scope » à « compile »
- la dépendance au module ejb [pam-ejb-glassfish], en précisant le type [ejb] et le « scope » à « compile »



Démarrer le serveur « glassfish », compiler [pam-webapp-ear] et exécuter-le (entraîne de déploiement) : votre navigateur doit afficher la page [http://localhost:8080/pam-web-ws-ejb-glassfish/] avec « Hello world ». Remarque : en cas de problème, vous pouvez forcer la compilation des projets « war » et « ejb ».

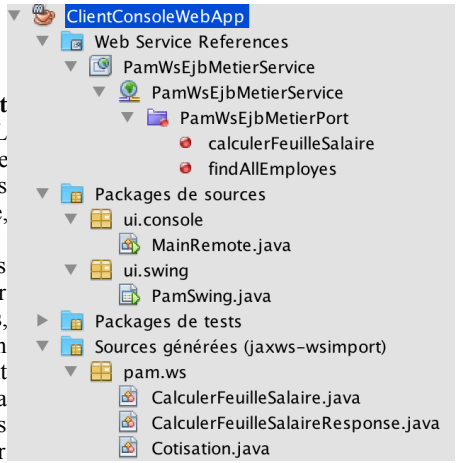
Problème du type « [2] EJBs ..etc. » : cela peut venir des dépendances : Attention à ce que le projet « webservice » ait une dépendance de type « provided » et non « compile » vers pam-ejb-glassfish, sinon vous aurez un « double » déploiement du « pam-ejb-glassfish ».

4. Test du webservice

Test 1: Après avoir déployé de le projet [pam-webapp-ear], vous pouvez tester votre webservice : projet [pam-webservice-glassfish] / bouton droit sur [Web Services / PamWsEjbMetier] / Tester webservice. Le navigateur doit afficher la page : [http://localhost:8080/MetierService/Metier?Tester]

Test 2 : en cliquant sur [findAllEmployes], vous devez récupérer les informations affichées (table [Employes] de la base MySQL) au format « xml » directement dans la page du navigateur. De même, vous pouvez calculer la feuille de salaire en entrant les bons paramètres. Si ces tests échouent, la suite ne fonctionnera pas. **Cause possible de problème : il faut avoir, avoir les getters/setters sur la classe [FeuilleSalaire].**

5. Partie cliente [ClientWebApp]



Structure du projet client

- Créer le projet [ClientWSEJB] de type « Maven / Java Application»
- Créer un « webservice client » du service web déployé : **« bouton droit sur le projet / new / webservice / webservice client ».** Pour le « WSDL and client location », on choisira [Project] et le webservice [PamWsEjbMetier] du projet [pam-webservice-glassfish]. Nous retrouvons des classes qui ont été déployées côté serveur : Indemnite, Cotisation , Employe , FeuilleSalaire, PamWsEjbMetier . [PamWsEjbMetier] est le service web et les autres classes sont des classes nécessaires à ce service. On pourra avoir la curiosité de consulter leur code. On verra que la définition des classes qui, instanciées, représentent des objets manipulés par le service, consiste en la définition des champs de la classe et de leurs accesseurs ainsi qu'à l'ajout d'annotations permettant la sérialisation de la classe en flux XML. La classe Metier est devenue une interface avec les deux méthodes @WebMethod. Chacune de celles-ci donne naissance à deux classes, par exemple [CalculerFeuilleSalaire.java] et [CalculerFeuilleSalaireResponse.java], où l'une encapsule l'appel à la méthode et l'autre son résultat. Enfin, la classe [PamWsEjbMetierService] est la classe qui permet au client d'avoir une référence sur le service web Metier distant :

```
@WebEndpoint(name = "PamWsEjbMetierPort")
public PamWsEjbMetier getPamWsEjbMetierPort() {
    return super.getPort(new QName("http://ws.pam/", "PamWsEjbMetierPort"), PamWsEjbMetier.class);
}
```

Extrait de la classe [MetierService] automatiquement générée

- En s'inspirant de l'exemple fourni, créer le programme console [MainRemote] dans le package [ui.console], où l'on récupérera l'instance [Metier] de la manière suivante (voir exemple fourni) :

```
PamWsEjbMetierService metier_service=new PamWsEjbMetierService();
PamWsEjbMetier metier=metier_service.getPamWsEjbMetierPort();
```

Code permettant de récupérer l'instance de [Metier] pour calculer la feuille de salaire.

- En vous inspirant du cas « console » et des TDs précédents pour l'utilisation de l'interface graphique avec swing, créer une interface graphique [PamSwing] dans le package [ui.swing]

6. Partie cliente [Python] : au moins en mode console (voir exemple fourni)

Objectif **:** Bases en JSF & application à l'application PAM

1 Généralités

Documentation en ligne : API : https://javaserverfaces.java.net/ ; Tutoriels : https://www.tutorialspoint.com/jsf/ http://www.coreservlets.com/JSF-Tutorial/jsf2/

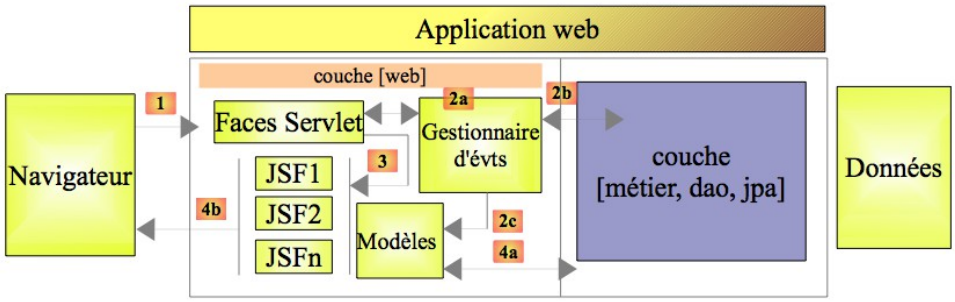


Illustration 1: Architecture détaillée de la couche web/JSF

Cette architecture implémente le *Design Pattern* **MVC** (Modèle, Vue, Contrôleur) :

- le contrôleur **C** est la servlet [javax.faces.webapp.FacesServlet] (bibliothèque [JSF-api.jar])
- les vues **V** sont implémentées par des pages JSF ;
- les modèles **M** et les **gestionnaires d'événements** sont implémentés par des classes Java ("backing beans") ;

Si la demande d'un client est faite avec un **GET**, les deux étapes suivantes sont exécutées :

1. **demande** - le client navigateur fait une demande au contrôleur [**Faces Servlet**]. Celui-ci voit passer toutes les demandes des clients. C'est la porte d'entrée de l'application. C'est le **C** de MVC ;
2. **réponse-** le contrôleur **C** demande à la page JSF choisie de s'afficher. C'est la vue, le **V** de MVC . La page JSF utilise un modèle **M** pour initialiser les parties dynamiques de la réponse qu'elle doit envoyer au client. Ce modèle est une classe Java qui peut faire appel à la couche [métier] [4a] pour fournir à la vue **V** les données requises

Si la demande est faite avec un **POST**, deux étapes supplémentaires s'insèrent entre la demande et la réponse :

1. **demande** -le client navigateur fait une demande au contrôleur [**FacesServlet**];
2. **traitement**- le contrôleur **C** traite cette demande POST accompagnée de données qu'il faut traiter. Pour ce faire, le contrôleur se fait aider par des gestionnaires d'événements spécifiques à l'application écrite [2a]. Ces gestionnaires peuvent avoir besoin de la couche métier [2b]. Le gestionnaire de l'événement peut être amené à mettre à jour certains modèles M [2c]. Le gestionnaire d'événement rend au contrôleur [Faces Servlet] un résultat de type chaîne de caractères appelée **clé de navigation ;**
3. **navigation**-le contrôleur choisit la page JSF(=vue) à envoyer au client (en fonction de la clé retournée)
4. **réponse** - la page JSF choisie va envoyer la réponse au client. Elle utilise son modèle **M** pour initialiser ses parties dynamiques. Ce modèle peut lui aussi faire appel à la couche [métier] [4a] pour fournir à la page JSF les données dont elle a besoin ;

2 Projet [mv-jsf01] fourni: Notions élémentaires et internationalisation

2.1 Fichier « web.xml »

```
7. <servlet>
8.          <servlet-name>Faces Servlet</servlet-name>
9.          <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10.         <load-on-startup>1</load-on-startup>
11. </servlet>
12. <servlet-mapping>
13.         <servlet-name>Faces Servlet</servlet-name> <url-pattern>/faces/*</url-pattern>
15. </servlet-mapping>
16. <session-config> <session-timeout> 30 </session-timeout> </session-config>
21. <welcome-file-list>
22.         <welcome-file>faces/index.xhtml</welcome-file>
23. </welcome-file-list>
   <!-- page d'exception -->
   <error-page>
      <error-code>500</error-code> <location>/faces/exception.xhtml</location> </error-page>
   <error-page>
```

```
    <exception-type>java.lang.Exception</exception-type> <location>/faces/exception.xhtml</location>
  </error-page>
```

Fichier web.xml

- La classe « servlet » [FacesServlet] est le contrôleur C du modèle MVC. La balise <servlet-mapping> sert à associer une servlet à une URL demandée par le navigateur client. Ici, il est indiqué que les URL de la forme [/faces/*] doivent être traitées par la servlet de nom [Faces Servlet]. Les URL des clients traitées par la servlet [Faces Servlet] auront donc la forme [http://machine:port/nom_projet/faces/*].
- <session-config … /> : durée d'une session. Ici, si un client C ne fait pas de nouvelle demande pendant 30 mn, sa session est détruite et les informations qu'elle contenait, perdues. Au prochain accès, une nouvelle session démarrera.
- La page d'accueil. Lorsque le client demande l'URL [http://machine:port/nom_projet], c'est l'URL [http://machine:port/nom_projet/index.xhtml] qui lui sera servie.
- Le fichier [exception.xhtml] gère les erreurs. **Pour tester la levée d'exception** : « corrompre votre fichier index.html » (par exemple en enlevant un « > » à une balise), compiler et exécuter le projet → votre navigateur doit instantanément afficher le message d'erreur (code 500) décrit dans exception.xhtml.

2.2 Fichier « page JSF » [index.xhtml]

```
10. <f:view … >
11. <h :head>  … </h :head>
14. <h :body> <h:form > ….  <h:outputText value="Saisir un nombre:"/> … </h:form> </h :body>
17. </f:view>
```

Fichier index.xhtml

- Les balises préfixées par **h sont des balises HTML et celles préfixées par f sont des balises propres à JSF**. La balise <f:view> sert à délimiter le code que le moteur JSF doit traiter, celui où apparaîssent les balises <f:xx>. La balise <h:form > introduit un formulaire, généralement constitué de balises de champs de saisie (texte, boutons radio, cases à cocher, listes d'éléments, ...) et balises de validation du formulaire (boutons, liens).
- La feuille de style (dossier [resources]) est définie à l'intérieur de la balise HTML <head >, par <h:outputStylesheet .../> Exemple d'utilisation (voir code): <h:panelGrid columnClasses="col1,col2" columns="2" > (tableau à trois colonnes, où le style de colonnes 1 et 2 seront col1 et col2). L'image de fond est définie par <h:body style="background-image: ….>

2.3 Lien entre page JSF et classe Java

```
    <h:form id="formulaire">
     <h:panelGrid columns="2" columnClasses="col1,col2">
     <!-- LIGNE 1 -->
     <h:outputText value="Saisir un nombre:"/>
     <h:inputText id="saisie1" value="#{form.myAttrib}"  required="true" requiredMessage="Requis" converterMessage="Pas Entier"/>
     <!-- LIGNE 2 -->
     <h:message for="saisie1" styleClass="error"/>
     <h:outputText value=""/> < !-- Ne sert a rien -->
     <!-- LIGNE 2 -->
     <h:outputText value="Nombre saisie:"/>
     <h:outputText value="#{form.myAttrib}"/>
     <!-- LIGNE 3 -->
     <h:commandLink type="submit()" value="SUBMIT"/>
     <h:commandButton value="INCREMENT" onclick='clean()' immediate='true' action="#{form.increment}"/>
     </h:panelGrid>
    </h:form>
```

```
@ManagedBean
@SessionScoped
public class Form implements Serializable
{
  private Integer myAttrib;
  public Form() {myAttrib=10 ;}
  public Integer getMyAttrib()   {   return myAttrib; }
  public void setMyAttrib(Integer h) {   myAttrib=h;}
  public String increment() {this.myAttrib+=10;return null;}
}
```

Extrait du fichier index.xhtml (en haut) et de la classe [Form] associée (en bas)

- L'extrait du fichier index.xhtml donne des exemples de composants graphiques en lien avec le « monde » Java : <h:inputText > pour la saisie, <h:command... /> pour l'exécution de commandes,… interagissant avec la classe [Form].
- La classe [Form] doit avoir les propriétés suivantes :

- @ManagedBean fait de la classe [Form] un bean reconnu par JSF. Son nom peut être fixé par @ManagedBean(name= "xx ") . **Par défaut, le nom de la classe est utilisé (premier caractère en minuscule): e.g. #{form...}.** ManagedBean appartient à javax.faces.bean.. et non javax.annotations....
- L'annotation SessionScoped fixe la portée du bean. Il y en a plusieurs, dont :
  - **RequestScoped** : la durée de vie du bean est celle du cycle demande navigateur / réponse serveur. L'objet sera à nouveau instancié (donc les données perdues!) pour traiter une nouvelle requête,
  - **SessionScoped** : la durée de vie du bean est celle de la session d'un client particulier. Le bean est créé initialement lors d'une des requêtes du client, et mémorise en général des données propres à un client donné.
  - **ApplicationScoped** : la durée de vie du bean est celle de l'application elle-même, souvent partagé par tous les clients de l'application. Il est en général initialisé au début de l'application.
- [Form] implémente l'interface [Serializable] : obligatoire pour la portée Session (e.g. sérialisation dans des fichiers)
- La classe [Form] doit avoir des getters/setters pour les attributs utilisés par la page JSF, de la forme [get]/[set] suivi du **nom de l'attribut avec la première lettre en majuscule (voir exemple)**

- Lien entre le fichier JSP et la classe Java, par l'exemple:
  - L'instruction [<h:inputText id="saisie1" value="#{form.myAttrib}"] signifie que la saisie sera affectée (lors du POST) à l'attribut [myAttrib] d'un objet de type [Form]. Ici, on impose qu'une valeur soit saisie avec l'option [required="true"] (sinon un message d'erreur est affiché). On personnalise également le message associé à une erreur de conversion (attribut attendu de type [Integer] ici). L'affichage de ces messsages est effectué dans une balise [<h:message />]. Remarque : on peut contraindre davantage la saisie (e.g. intervalle de valeur autorisée : <f:validateLongRange minimum="1" maximum="10" />). Bien que non illustré ici, on peut également valider des contraintes complexes avec du code Java côté serveur (e.g. concernant simultanéement plusieurs champs).
  - [<h:outputText value="#{form.myAttrib}"/>] permet d'afficher la valeur de l'attribut [myAttrib] (getter requis)
  - [<h:commandButton value="INCREMENT" ... action="#{form.increment}"/>] invoquera la méthode [increment], qui va incrémenter la valeur de [myAttrib], la nouvelle valeur étant affiché dans la page du navigateur, cette page étant retourné par le contrôleur. Cette méthode rendre un résultat de type String qui pourrait être le nom d'un page à afficher (généralement [return null] pour ne pas changer de page).
  - [<h:commandLink type="submit()" value="..."/>], en précisant le type [submit] permet de déclencher le POST.

- Remarque générale : En fonction de la requête, le contrôleur analyse la page JSP à afficher (e.g. mise à jour en fonction du code Java associé), avant de finalement retourner une page web « classique » purement html/javascript. Vous pouvez consulter le code source de la page depuis le navigateur pour constater la différence entre la page finale et la page JSP.

- Certaines instructions dans le fichier JSP peuvent ne pas avoir de lien avec des classes java.
  - Par exemple, dans le fichier index.xhtml, l'instruction [<h:commandLink value="#{msg['welcome.page1']}" action="page1"/>] permet de tout de suite passer à la page [page1.xhtml].
  - Par exemple, on peut déclarer, dans la page JSP du code javascript à exécuter, par exemple :

```
  <script type="text/javascript">
    function clean(){
     document.forms['formulaire'].elements['formulaire:saisie1'].value=""; document.forms['formulaire'].submit(); }
  </script>
….
<h:inputText id="saisie1" value="#{form.myAttrib}"  … />
….
<h:commandButton value="INCREMENT" onclick='clean()' immediate='true' action="#{form.increment}"/>
```

Exemple d'usage du javascript : sur "onclick", la fonction [clean] est appelée pour reinitialiser (vider) le champ de saisie [saisie1]. A noter que la méthode [increment] sera également invoquée. L'instruction [immediate='true'] permet d'éviter le processus de validation qui échouerait car on vide le champ [saisie1] alors que celui est requis [required="true"].

2.4 Internationalisation (i18n – car 18 lettres entre « I » et « n »)

```
6. <f:view locale="#{changeLocale.locale}">
7.          <head> <title><h:outputText value="#{msg['welcome.titre']}" /></title> </head>
11.         ...
13. </f:view>
```

Fichier « Index.xhtml »

- La langue est associé la variable [locale], dont la valeur est géré par la classe [ChangeLocale]. En fonction cette valeur, les textes affichés seront différents (en anglais, en français), ceci étant géré par des fichiers (un fichier par langue) et la variable [msg] (voir l'exemple <h:outputText /> avec #{msg['welcome.titre']}). [msg] est associé à un fichier de messages. Ce dernier doit être déclaré dans le fichier de configuration [faces-config.xml]:

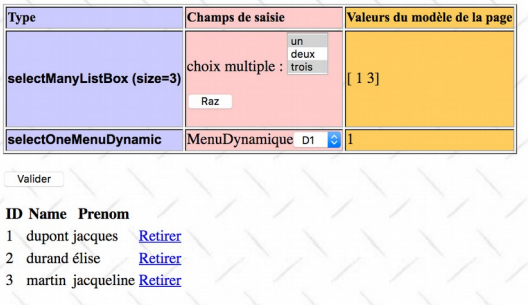
```
11. <application>
12.         <resource-bundle>
13.                 <base-name>messages</base-name>    <var>msg</var>
17.         </resource-bundle>
18. </application>
```

Fichier « faces-config », où <application > sert à configurer l'application JSF

- Dans le cas de [#{msg['welcome.titre']}], le système ira chercher la chaine de caractère associée à "welcome.titre" dans le fichier "messages[_CodeLangue][_CodePays].properties", où [_CodeLangue][_CodePays] est défini par la variable [locale] (« fr » ou « en » dans notre cas). Il peut exister plusieurs fichiers de messages (typiquement placés dans [src / main / resources] du projet) : messages_fr.properties (français), messages_en.properties (anglais) et messages.properties (par défaut). Ici, si [locale] vaut "fr" : "#{msg['welcome.titre']}" vaudra "Tutoriel JSF (JavaServer Faces)"

welcome.titre=JSF (JavaServer Faces) Tutorial welcome.langue1=French welcome.langue2=English	welcome.titre=Tutoriel JSF (JavaServer Faces) welcome.langue1=Français welcome.langue2=Anglais
messages_en.properties	messages_fr.properties

3 Projet [mv-jsf02] fourni : Quelques composants graphiques avancés.



Balise <h:selectManyListBox> :

```
2. <h:outputText value="selectManyListBox (size=3)" styleClass="info"/>
3. <h:panelGroup>
4. <h:outputText value="#{msg['form.selectManyListBoxPrompt']}"/>
5. <h:selectManyListbox id="selectManyListBox" value="#{form.selectManyListBox}"
size="3">
6. <f:selectItem itemValue="1" itemLabel="un"/>
….
10. <f:selectItem itemValue="5" itemLabel="cinq"/>
11. </h:selectManyListbox>
12. <p><input type="button" value="…" onclick="this.form['formulaire:selectManyListBox'].selectedIndex=-1;" /></p>
```

Cet élément est associé au tableau de chaine de caractère de [Form] : la méthode invoquée ([getSelectManyListBoxValue()]) ne retourne pas un tableau, mais une seule chaine qui concatène toutes les chaines du tableau.

```
1. private String[] selectManyListBox=new String[]{"1","3"};
2. ...
5. public String getSelectManyListBoxValue(){
10. String value="[";
11. for(String chaine : chaines){ value+=" "+chaine;}
14. return value+"]";
15. }
```

Extrait de Form.java

Lorsqu'on clique sur le bouton [Raz], le code Javascript de l'attribut [onclick] s'exécute. Il permet de modifier directement dans la page côté navigateur (sans « POST » au serveur) l'état du « ManyListBox » : la valeur -1 à l'attribut selectedIndex a pour effet de désélectionner tous les éléments de la liste s'il y en avait.

Balise <h:selectOneMenu> avec <f:selectItems> (listes dynamiques): Les éléments de la liste sont ici générés dynamiquement par du code Java, et non « en dur » comme précédemment

```
    <h:panelGroup>
      <h:outputText value="..."/>
      <h:selectOneMenu id="selectOneMenuDynamic" value="#{form.selectOneMenuDynamic}">
        <f:selectItems value="#{form.selectOneMenuDynamicItems}"/>
      </h:selectOneMenu>
    </h:panelGroup>
    <h:outputText value="#{form.selectOneMenuDynamic}"/>
```

```
import javax.faces.model.SelectItem;
...
```

```
public SelectItem[] getSelectOneMenuDynamicItems() {
    SelectItem[] items=new SelectItem[3];
    items[0]=new SelectItem("1","D1");
    items[1]=new SelectItem("2","D2");
    items[2]=new SelectItem("3","D3");
    return items;  }
```

Le contenu de la liste déroulante est dynamiquement fournie par la méthode [**getSelectOneMenuDynamicItems**] de Form

Balise <h:dataTable>:

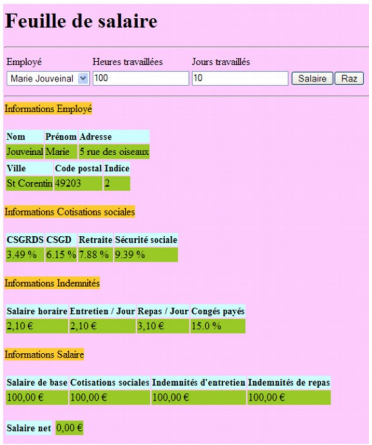
```
1. <h:dataTable value="#{table.personnes}" var="personne" … >
2. <h:column>
3. <f:facet name="header"> <h:outputText value="Id"/> </f:facet>
6. <h:outputText value="#{personne.id}"/>
7. </h:column>
8. …
4. <h:commandLink value="Retirer" action="#{table .retirerPersonne}">
5.          <f:setPropertyActionListener target="#{table .personneId}" value="#{personne.id}"/>
6. </h:commandLink>
9. </h:dataTable>
```

- Le bean [Table] comprend une liste de personne [table.personnes] que nous affichons ligne par ligne : l'attribut var="personne" fixe le nom de la variable représentant la personne courante à l'intérieur de la balise <h:datatable >.
- Lorsque le lien [Retirer] est cliqué, la méthode [Form].retirerPersonne : la personne associée à l'identifiant [personne.id] va être retiré, en affectant préalablement la valeur [personne.id] (« value ») à l'attribut [personneId] de [Table] (« target »). Cette affectation préalable est géré par le [<f:setPropertyActionListener … />] va être exécutée. A noter que [Table] est de portée « session » pour que cette liste de personne vive au fil des requêtes

```
@SessionScoped.
11. public class Table {
14. private List<Personne> personnes;
15. private int personneId;
18. public Form() { // initialisation de la liste des personnes }
26. public String retirerPersonne() { /* On retire la personne dont l'identifiant vaut [personneId] */ return null}
43. }
```

4 Objectif

L'objectif est d'implémenter de l'application web « PAM » avec JSF (voir copies d'écran attendues)



Note: un projet « JSF » peut être créé depuis netbeans: « Maven / Web Application » [mv-pam-jsf2-simulation], avec le framework JavaServer Pages [Properties → Framework → Add JavaServer Pages]. Après compilation et exécution, consulter, avec votre navigateur, la page : http://localhost:8080/mv-pam-jsf2-simulation/ pour vérifier que tout fonctionne. **Dans notre cas, un caneva est fourni, que l'on peut directement compiler et exécuter (déploiement sur serveur glassfish fourni).**

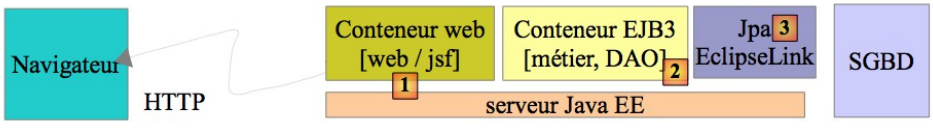


Illustration 2: Architecture considérée

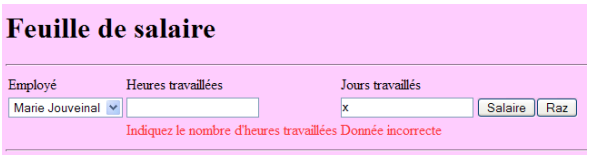
5 Implémentation de la couche [ui] JSF avec une couche métier « simulée »

5.1 *Travail à faire : étape 1*

Implémenter le fonctionnalité permettant de passer d'un mode « français » à un mode « anglais ».

5.2 *Travail à faire : étape 2*

Construire le formulaire [index.xhtml] et son modèle [Form.java] pour obtenir la page suivante, avec les composants JSF du type [<h:selectOneMenu>, <h:inputText>, <h:commandButton>] :



Validité de la saisie : exemple d'erreur (champ non rempli, ou donnée incorrecte)

- Les objets affichés par le combo auront pour attribut itemValue , le n° SS de l'employé et pour attribut itemLabel, une chaîne constituée du prénom et du nom de l'employé ;
- Les boutons [Salaire] et [Raz] seront pour l'instant inactifs
- La validité des saisies sera vérifiée

Pensez à utiliser le « logger » ([java.utils.logging.logger]) pour suivre les échanges entre le navigateur et le serveur

5.3 Travail à faire : étape 3

Travail à faire : compléter le projet pour obtenir la page suivante une fois que le bouton [Salaire] a été cliqué. Le bouton [Salaire] sera connecté à la méthode [calculerSalaire] de [Form], qui invoquera la méthode [calculerFeuilleSalaire] de la couche [métier]. Dans la classe [Form], la feuille de salaire sera représentée par : private FeuilleSalaire feuilleSalaire; (avec méthodes get/set).

Pour l'affichage, on pourra écrire des expressions comme : <h:outputText value="#{form.feuilleSalaire.employe.nom}"/>, se traduisant par l'appel à [form].getFeuilleSalaire().getEmploye().getNom() (attention à ce que ces méthodes existent!).

L'affichage des « informations » se fera à l'aide d'une balise « subview » dont on contrôlera la visibilité (i.e. [rendered]) grâce à un nouvel de la classe [Form] (« private boolean viewInfosIsRendered; ») :

```
1. <f:subview id="viewInfos" rendered="#{form.viewInfosIsRendered}">
2. ... la partie du formulaire qu'on veut pouvoir ne pas afficher
3. </f:subview>
```

5.4 Travail à faire : Etape 4 : Gérer le bouton [Raz], ramenant le formulaire dans l'état initial

- Réinitialiser les champs de saisie avec un code javascript (onclick="raz")
- Cacher la partie « salaire » en mettant l'attribut [viewInfosIsRendered] à « False » (méthode [Form::raz])

```
<h:commandButton id="raz" onclick="raz()" value="..." immediate="true" action="#{form.raz}"/>
```

6 [Optionnel] Intégration de la couche « ejb » réelle (couche métier réelle – non simulée)

Nous commençons par adapter la couche web, en intégrant de la couche « ejb ». Pensez à travailler sur une copie du projet web (e.g. [pam-jsf2] et supprimer les paquetages [exception, metier, JPA] « simulés » et ajouter ensuite aux dépendances du projet web le projet du serveur EJB construit précédemment (TD5 : pam-ejb-glassfish): « Add dependancy » → dépendance de type « **ejb** » avec le scope « **provided** » (car fourni ultérieurement au projet web par son environnement). Adapter également la classe [Form] pour référencer la couche [metier] réelle (« EJB local » car exécution dans la même JVM). **Note :** pour éviter les erreurs de compilation, pensez à « encadrer », dans la classe [Form], les appels aux méthodes de [metier] par un [try {} catch(Exception ex) {}].

```
<dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>pam-ejb-glassfish</artifactId>
        <version>${project.version}</version>
```

```
...
public class Form {
@EJB
private IMetierLocal metier; // couche métier réelle
```

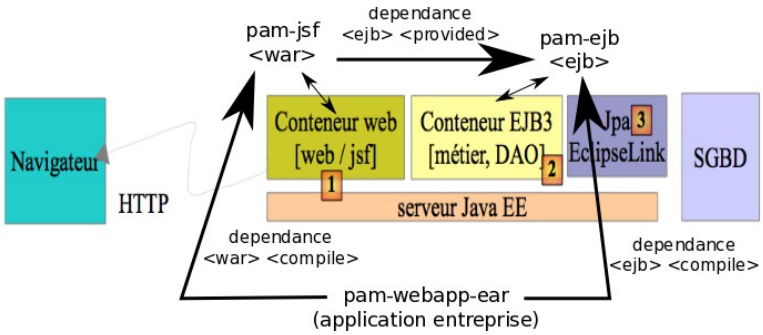
```
            <scope>provided</scope>                    …
            <type>ejb</type>
        </dependency>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-web-api</artifactId>
            <version>7.0</version>
            <scope>provided</scope>
        </dependency>
```

Forme du fichier « pom.xml »

Adaptation du bean [Form.java]

Comme pour le TD avec le « webservice », créez un nouveau projet Netbeans de type « Maven / Entreprise Application » [pam-webapp], en décochant les options « create web..» et « create ejb » (fournis ensuite) :

- Deux projets Maven ont été créés. Le projet d'entreprise est celui qui a le suffixe **ear**. L'autre projet est un projet parent.
- Nous ajoutons le module web et le module EJB au projet d'entreprise (ajout de dépendance):
  - ajout du projet EJB [pam-ejb-glassfish], de type *ejb (scope : compile)*
  - ajout du projet web [pam-jsf2], de type *war (scope : compile)*



Vérifiez que la base MySQL [dbpam] existe et est remplie, et déployez l'application d'entreprise [pam-webapp-ear] :

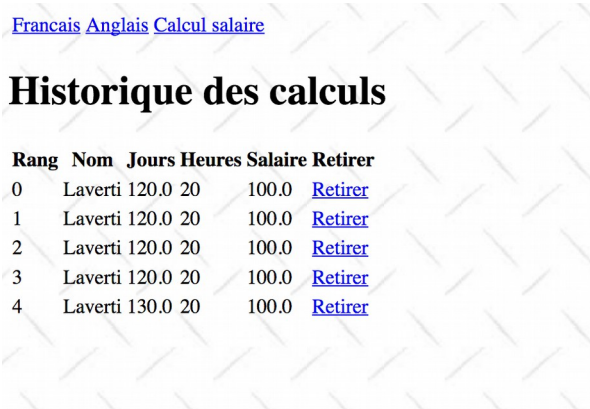
- faire un [Clean and Build] sur les projets EJB [pam-ejb-glassfish], Web [pam-jsf2] et EAR [pam-webapp-ear] ;
- Exécuter le projet d'entreprise (projet « ear ») : la page [http://localhost:8080/pam-jsf2/faces/index.xhtml] doit s'afficher.

Objectif : Application web et utilisation du framework JEE Java Server Pages : application multi-pages avec « sessionScoped »

1 Objectif

L'objectif est de développer d'étendre l'application en ajoutant l'historique des salaires calculés. On souhaite également que cet historique soit éditable (on souhaite pouvoir retirer des éléments de l'historique).

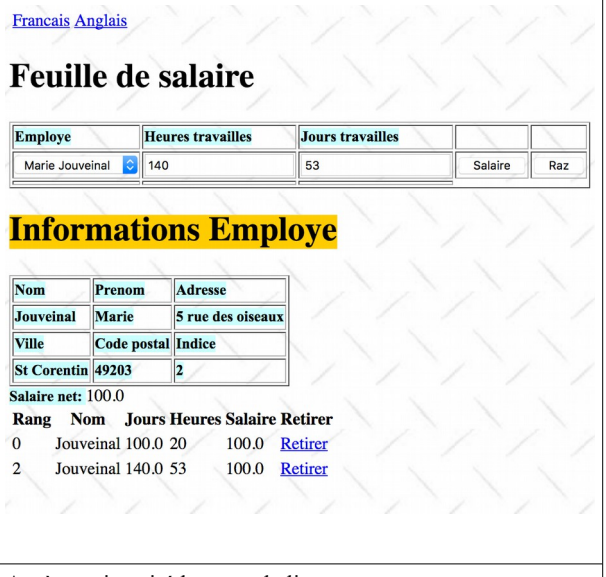
Vue principale pour le calcul des salaires



Vue secondaire pour l'historique

2 Etape 1 : Affichage de l'historique dans la page courante.

Affichage de l'historique après 3 calculs de feuilles de salaire



Après avoir retiré la seconde ligne

- On ajoute la classe [HistoryElement] au même package que [Form] : il stockera les attributs [rank, name, jours, heures et salaireNet] requis pour l'historisation. Rank correspond au rang de l'élément dans la liste (0, 1, 2, ...)
- Form aura un nouvel attribut [List<HistoryElement> history= new ArrayList<>() ;] avec le getter associé. History se verra ajouter un nouvel 'HistoryElement' à chaque calcul de salaire (méthode [calculerSalaire]).
- Changer l'annotation « scope » de [Form] en [@SessionScoped] En effet, celui-ci ne doit pas être de type [@RequestScoped], sinon l'objet de type [Form] sera instancié à chaque requête, perdant ainsi l'historique. **Attention à bien utiliser le [SessionScoped] du package [import javax.faces.bean.SessionScoped]**

- Ajouter au [index.xhtml] le tableau de l'historique, de type [h:dataTable], en associant la « valeur » à l'historique ([value="#{form.history}"]). Dans un premier temps, pour tester que tout fonctionne, ne considérer qu'une seule colonne, affichant par exemple la valeur du rang pour chaque opération. Pour implémenter cette étape, il faut lire l'exemple [mv-jsf04] fourni.
- Ajouter la fonctionnalité d'édition de la table (i.e. de l'historique) avec la colonne [retirer], qui permet de retirer un élément de l'historique (et donc de la table qui sera ensuite re-afficher). On utilisera un élément du type (voir l'exemple [mv-jsf04] fourni):

```
<h:column>
    <h:commandLink value="Retirer" action="#{form.removeElement}">
      <f:setPropertyActionListener target="#{form.attribut}" value="#{value}"/>
    </h:commandLink>
</h:column>
```

Cette [commandLink] permet d'exécuter l'action (méthode) [form.removeElement]. Cette méthode n'aura pas de paramètre. Cependant, on a besoin de passer à [form] l'identifiant de l'objet à retirer de l'historique : ici, ce sera le [rank]. C'est là que le [f:setPropertyActionListener] intervient. On va considérer un nouvel attribut ([form.attribut] dans l'exemple ci-dessus) de [form] que [form.removeElement] utilisera. La valeur de cet attribut correspondra à [value="#{value}"], où « value » est la valeur que l'on souhaite affecter au nouvel attribut. Dans notre cas, on pourra utiliser le [rank] associé [HistoryElement].

3 Etape 2 : Affichage dans une autre page

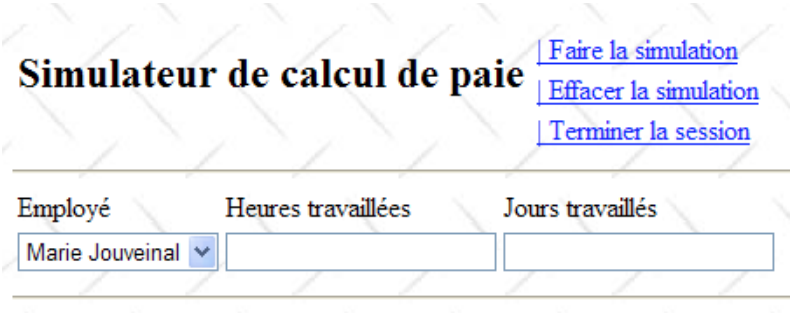
On souhaite que le calcul de feuille de salaire et l'affichage de l'historique se fassent dans deux pages distinctes (voir ci-dessous). Il s'agit simplement d'ajouter deux liens ([<h:commandLink />]) permettant de naviguer entre les deux pages (voir exemple [mv-jsf01])

Objectif : Application web et utilisation du framework JEE Java Server Pages : application multi-pages / multi-vues

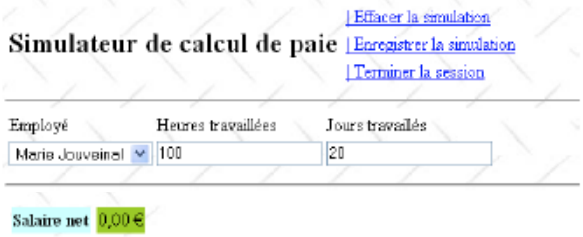
1 Objectif

L'objectif est de développer un simulateur de calcul de paie, impliquant plusieurs vues. Les différentes vues présentées à l'utilisateur seront les suivantes :

- la vue [accueil.xhtml] qui présente le formulaire de simulation :

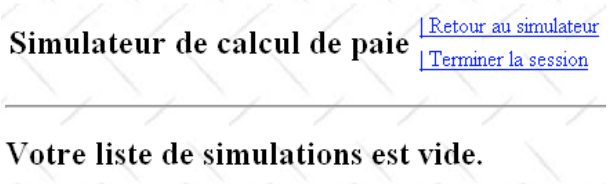


- la vue [simulation.xhtml] utilisée pour afficher le résultat détaillé de la simulation :

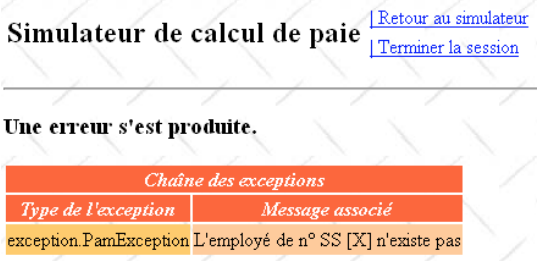


- la vue [simulations.xhtml] (simulation**S** avec un « S » !) qui donne la liste des simulations faites par le client

- la vue [simulationsVides.xhtml] qui indique que le client n'a pas ou plus de simulations :

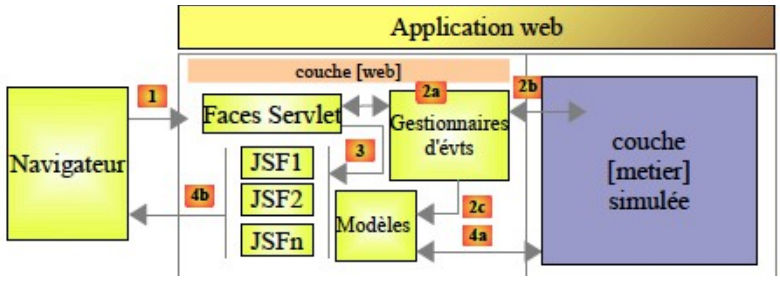


- la vue [erreurs.xhtml] qui indique une ou plusieurs erreurs :



2 Architecture

Nous revenons ici à l'architecture initiale où la couche [métier] était simulée. Nous savons désormais que celle-ci peut être aisément remplacée par la couche [métier] réelle. La couche [métier] simulée facilite les tests.



Une application JSF est de type MVC (Modèle Vue Contrôleur) :

- la servlet [Faces Servlet] est le contrôleur générique fourni par JSF. Ce contrôleur est étendu par les gestionnaires d'événements spécifiques à l'application. Les gestionnaires d'événements rencontrés jusqu'ici étaient des méthodes des classes servant de modèles aux pages JSF ;
- les pages JSF envoient les réponses au navigateur client. Ce sont les vues de l'application ;

Dans l'architecture JSF, le passage d'une page JSFi à une page JSFj peut être problématique :

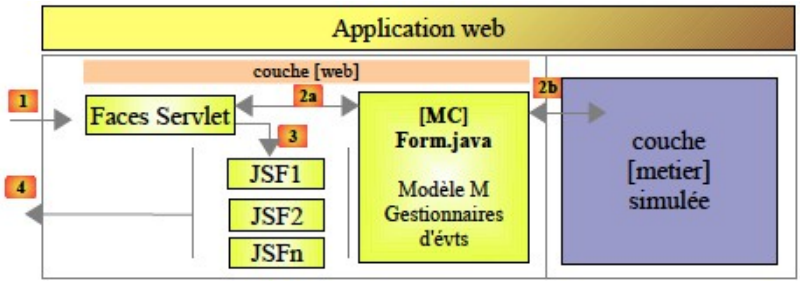
- la page JSFi a été affichée. A partir de cette page, l'utilisateur provoque un POST par un événement quelconque [1]
- en JSF, ce POST sera traité [2a,2b] en général par une méthode C du modèle Mi de la page JSFi . On peut dire que la méthode C est un contrôleur secondaire ;
- si à l'issue de cette méthode, la page JSFj doit être affichée, le contrôleur C doit :

  1) mettre à jour [2c] le modèle Mj de la page JSFj ;

  2) rendre [2a] au contrôleur principal, la clé de navigation qui permettra l'affichage de la page JSFj ;

  L'étape 1 nécessite que le modèle Mi de la page JSFi ait une référence sur modèle Mj de la page JSFj . Cela complique un peu les choses rendant les modèles Mi dépendant les uns des autres. En effet, le gestionnaire C du modèle Mi qui met à jour le modèle Mj doit connaître celui-ci. Si on est amené à changer le modèle Mj , on sera alors amené à changer le gestionnaire C du modèle Mi. Il existe un cas où la dépendance des modèles entre-eux peut être évitée : celui où il y a un unique modèle M qui sert à toutes les pages JSF. Cette architecture n'est utilisable que dans les applications n'ayant que quelques vues mais elle se révèle alors très simple d'usage. C'est celle que nous utilisons maintenant.

Dans ce contexte, l'architecture de l'application est la suivante :



3 Le projet NetBeans fourni : couche métier simulée

Pour implémenter cette application, plusieurs éléments doivent être programmés et/ou configurés (projet initial fourni):

- Les fichiers de configuration [faces-config.xml] et [web.xml]. Ils sont identiques aux applications précédentes à l'exception de la page d'accueil « faces/accueil.xhtml ».
- les dépendances du projet.
- La feuille de style et l'image de fond des vues,
- les couches « basses » de l'application : dans notre cas (pour la version « simulée »), il s'agit des packages précédemment développés [jpa], [metier] et [exception].
- le fichier des messages pour l'internationalisation de l'application
- Les pages JSF
- Les classes de la couche [web] (voir ci-après) dans les packages [web.entities] [web.beans.*]

3.1 Pages JSF

La page d'accueil est la page [accueil.xhtml]: cette page s'affiche à l'intérieur de la page [layout.xhtml] la place du fragment nommé part1. Dans ce fragment, on affiche la page [saisie.xhtml].

L'entête [entete.xhtml] est intégrée par le « template » [layout.xhtml] : elle affiche un titre général ainsi que les différents liens (en haut à droite) correspondant aux différentes actions. A noter que ces liens (<h:commandLink >) sont affichés (« rendered=... ») selon la vue rendue : tous les liens possibles sont déclarés mais seulement certains sont affichés, ceci étant géré par le bean « SessionData » (voir ci-après). Plus précisément :

- les six liens correspondant aux six actions que peut faire l'utilisateur. Ces liens sont contrôlés (attribut rendered) par des booléens du bean SessionData.
- Un clic sur le lien [Effacer la simulation] provoque l'exécution de la fonction Javascript raz . Celle-ci a été définie dans le modèle [layout.xhtml],
- l'attribut « immediate=true » fait que la validité des données n'est pas vérifiée avant exécution de la méthode (par exemple [Form.enregistrerSimulation] de l'entête « entete.xhtml »). C'est voulu. On peut vouloir enregistrer la dernière simulation faite il y a une minute, même si les données saisies depuis dans le formulaire (mais pas encore validées) ne sont pas valides. Il est fait de même pour les actions [Voir les simulations], [Effacer la simulation], [Retour au simulateur] et [Terminer la session].

3.2 Couche métier « simulée »

On reprend la classe métier précédente : la seule différence est que l'on ajoute « temporairement » un employé qui n'existe pas, afin de tester la levée d'erreur.

3.3 Le bean « ApplicationData »

Le bean ApplicationData sera de portée application et sert à deux choses :

- maintenir une référence sur la couche [métier],
- définir un « logger » qui pourra être utilisé par les autres beans pour faire des logs sur la console de Glassfish.

Quelques remarques techniques (voir code fourni) :

- l'annotation @Named fait de la classe un bean managé. On notera qu'à la différence du projet précédent, on n'a pas utilisé l'annotation @ManagedBean. La raison en est que la référence de cette classe doit être injectée dans une autre classe à l'aide de l'annotation @Inject et que celle-ci n'injecte que des classes annotées @Named
- l'annotation @ApplicationScoped fait de la classe, un objet de portée application. On notera que la classe de l'annotation est [javax.enterprise.context.ApplicationScoped] et non [javax.faces.bean.ApplicationScoped] comme dans les beans du projet précédent.

3.4 Le bean « SessionData »

Le bean SessionData a les caractéristiques suivantes :

- La classe SessionData est un bean managé (@Named) qui pourra être injecté dans d'autres beans managés,
- Elle est de portée session (@SessionScoped),
- Une référence sur le bean ApplicationData lui est injecté (@Inject),
- Gère les données de l'application qui doivent être maintenues au fil des sessions : la locale (la langue), l'état des liens (affichés ou non) dans le navigateur, selon le mode (i.e. la vue) dans lequel l'utilisateur se trouve.
- Gère la liste des simulations faites par l'utilisateur, ainsi que le n° de la dernière simulation enregistrée et la dernière simulation qui a été faite,

A noter que la méthode [init] est exécutée après instanciation de la classe (@PostConstruct). Ici, elle n'est utilisée que pour laisser une trace de son exécution. On doit pouvoir vérifier qu'elle n'est exécutée qu'une fois par utilisateur puisque la classe est de portée session. Cette méthode utilise le [logger] défini dans la classe ApplicationData (C'est pour cette raison qu'on avait besoin d'injecter une référence sur ce bean).

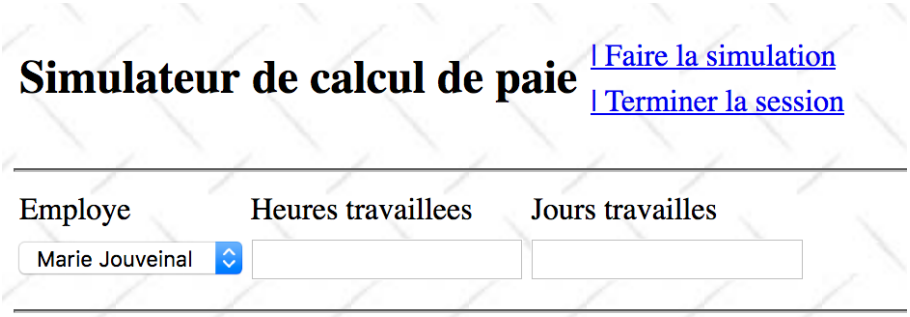
3.5 Le bean « Form »

Le bean Form est caractérisé par :

- la classe est un bean managé (@Named),
- de portée requête (@RequestScoped), donc de durée de vie limitée à la requête
- injection d'une référence sur le bean de portée application ApplicationData ,
- injection d'une référence sur le bean de portée session SessionData.

4 Travail à faire

4.1 Page de connexion



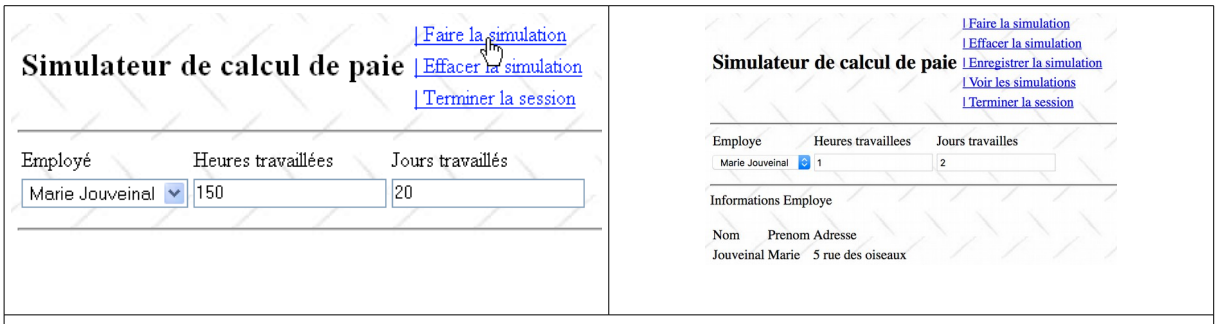
Page attendue lors de la connexion

Travail à faire : Compléter le programme pour que la liste des éléments du combo des employés soit fournie par une méthode du bean [Form]. Les éléments affichés dans le combo auront leur propriété **itemValue** égale au n° SS d'un employé et la propriété **itemLabel** sera une chaîne formée du prénom et du nom de celui-ci.

Travail à faire : Initialisez correctement le bean [SessionData] pour que, lors de la requête GET initiale faite au formulaire, le menu de l'entête soit celui montré ci-dessus.

4.2 Faire une simulation

Travail à faire : Ecrire le code la méthode [faireSimulation] de la classe [Form]. Ceci inclut le calcul de la feuille de salaire ainsi que la création d'une simulation qui sera enregistrée dans le bean SessionData : cela concerne l'attribut « simulation » du « SessionData ». Cet attribut correspond à la simulation « courante », qui sera ultérieurement ajouté à la liste des simulations de « SessionData » au moyen du lien [Enregistrer la simulation].



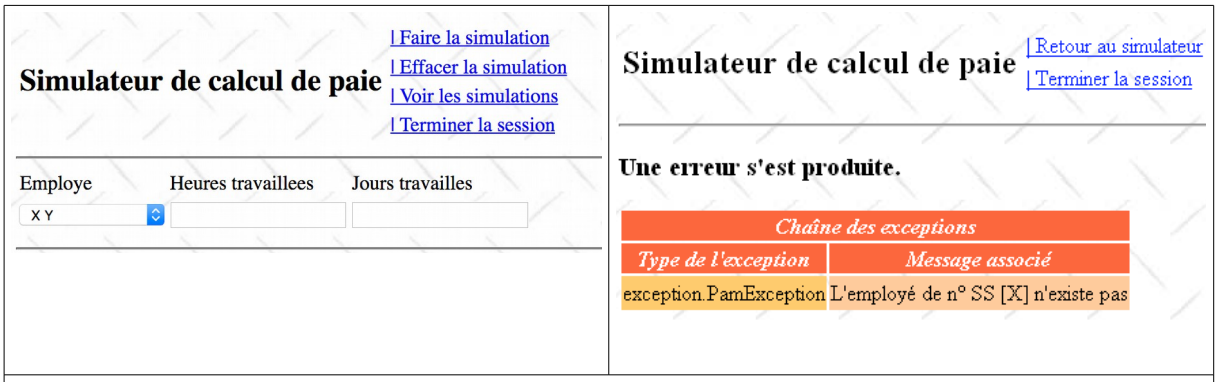
Simulation : données entrées (à gauche) et résultat attendu (à droite)

La structure de la méthode [faireSimulation] est la suivante :

```
// action du menu
public String faireSimulation(){
// on calcule la feuille de salaire
feuilleSalaire= …
// on calcule la simulation
...
// on met . jour le menu
. ...
// on rend la vue simulation
return "simulation";
}
```

On constate que la méthode [faireSimulation] doit retourner le nom de la page qui doit être affichée, à savoir ["simulation"] dans notre cas. Ainsi, de manière générale, l'instruction **[<h:commandLink … action="#{monBean.method}" />]** associée à la méthode [method] de la classe MonBean **[public String method() { return "page1" ;}]**, est équivalente à l'instruction **[<h:commandLink ... action="page1" />]**. Cette technique permet à la fois d'exécuter un code java [MonBean::method()] tout en changeant la page rendue dans le navigateur.

4.3 La page d'erreur



Page d'erreur : l'employé sélectionné (à gauche) n'existe pas, ce qui conduit à l'affichage d'une page d'erreur (à droite)

Travail à faire : Compléter la méthode [faireSimulation] afin que lors d'une exception, elle fasse afficher la vue [vueErreur] (voir figure).

Démarche : afin de tester la levée d'erreur, sélectionnez un employé « erroné » (XYZ) ajouté en dur dans la méthode « findAllEmployes » de la classe Metier.

La structure de la méthode [faireSimulation] intégrant l'erreur est la suivante :

```
// action du menu
2. public String faireSimulation(){
3. try{
4. // on calcule la feuille de salaire
5. feuilleSalaire= ...
6. // on met . jour le menu
7. ...
8. // on rend la vue simulation
9. return "simulation";
10. }catch(Throwable th){
11. // on vide la liste des erreurs pr.c.dentes
12. ...
13. // on cree la nouvelle liste des erreurs
14. ...
15. // on affiche la vue vueErreur
16. ...
17. // on met . jour le menu
18. ...
19. // on affiche la vue erreur
20. return "erreurs";
21. }
22.}
```

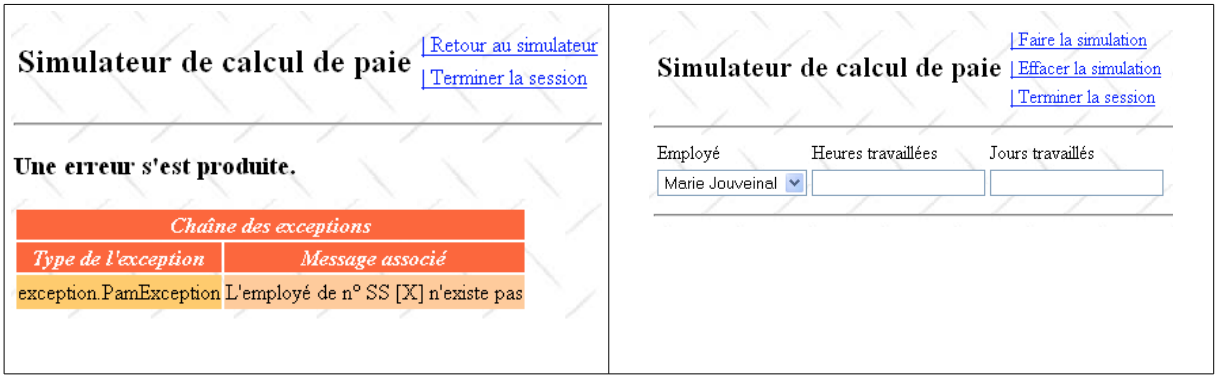
La liste des erreurs est mémorisée dans la classe [Form] par l'attribut private List<Erreur> erreurs=new ArrayList<Erreur>();

Cette liste d'erreur est construite en ajoutant des objets de type [Erreur] (classe fournie) à la liste : chaque erreur est construite avec le nom de la classe de l'exception (e.g. th.getClass().getName()) et le message associé (e.g. th.getMessage()). On ajoute itérativement les causes de l'exception (e.g. [cause.getCause()]) jusqu'à ce que la cause soit nulle (i.e. [cause.getCause() == null]).

Note: documentation sur [Throwable] en ligne : [https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html].

4.4 Action [retour simulateur]

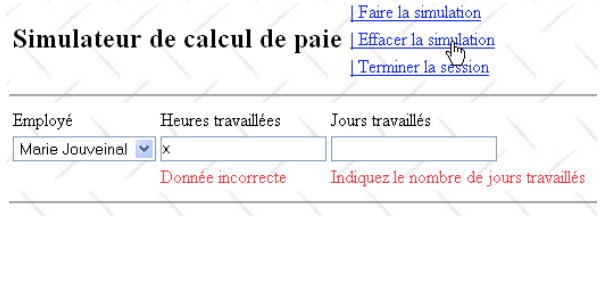
L'action [retourSimulateur] associée au lien permet à l'utilisateur de revenir de la vue [vueSimulations] à la vue[vueSaisies] :

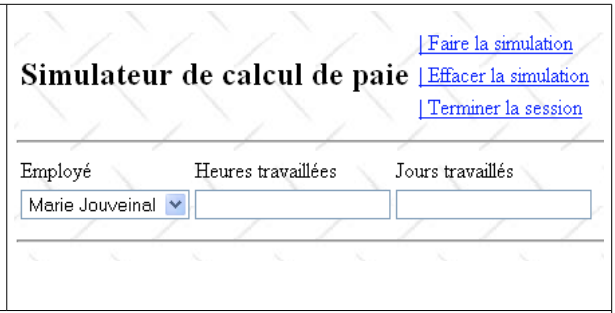


Travail à faire : écrire la méthode [retourSimulateur] de la classe [Form]. Le formulaire de saisie présenté doit être vide comme ci-dessus.

4.5 Action [effacer simulation]

L'action [effacerSimulation] permet à l'utilisateur de retrouver un formulaire vide :





Effacer une simulation conduit à la page de droite.

Un clic sur le lien [EffacerSimulation] provoque d'abord l'appel de la fonction Javascript raz(). Cette méthode est définie dans la page [layout.xhtml] :

- les valeurs postées sont des valeurs valides, c.a.d. qu'elles passeront les tests de validation des champs de saisie heuresTravaillees et joursTravailles ;
- la fonction raz ne poste pas le formulaire. En effet, celui-ci va être posté par le lien cmdEffacerSimulation . Ce post se fera après exécution de la fonction Javascript raz ;

Au cours du post , les valeurs postées vont suivre un cheminement normal : validation puis affectation aux champs du modèle. Ceux-ci sont les suivants dans la classe [Form] :

```
// le mod.le des vues
private String comboEmployesValue;
private String heuresTravaill.es;
private String joursTravaill.s;
```

Ces trois champs vont recevoir les trois valeurs postées {"0","0","0"}. Une fois cette affectation opérée, la méthode effacerSimulation va être exécutée.

Travail à effectuer : écrire la méthode [effacerSimulation] de la classe [Form]. On fera en sorte que :

- seule la zone des saisies soit affichée,
- le combo soit positionné sur son 1er élément,
- les zones de saisie heuresTravaillees et joursTravailles affichent des chaînes vides.
- La liste des commandes (entête) soit réduite à [Faire simulation] [Effacer simulation] [Terminer Session]

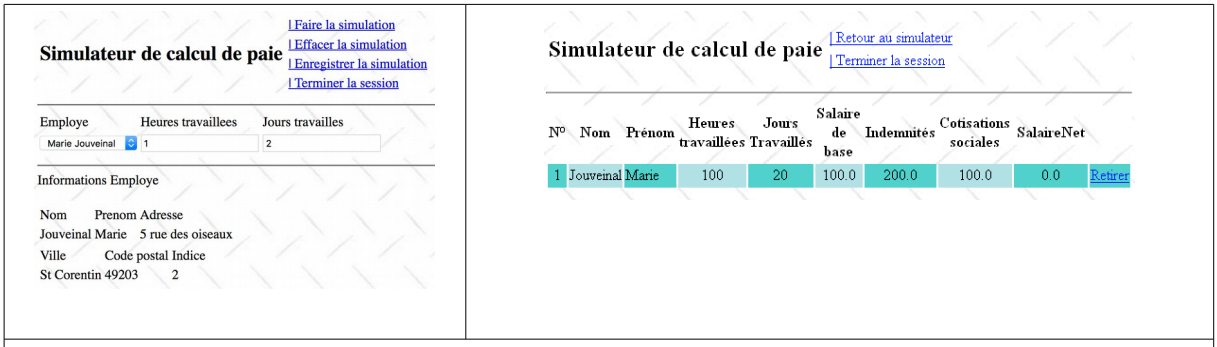
4.6 Action [enregistrer simulation]

L'action [enregistrerSimulation] associée au lien permet d'enregistrer la simulation courante dans une liste de simulations maintenue dans la classe [SessionData] :

```
private List<Simulation> simulations=new ArrayList<Simulation>();
```

La classe Simulation permet de mémoriser une simulation faite par l'utilisateur :

- le n° de la simulation (nombre incrémenté à chaque nouvel enregistrement (SessionData)),
- la feuille de salaire qui a été calculée,
- le nombre d'heures travaillées,
- le nombre de jours travaillés.



Enregistrement d'une simulation : la page affichée devient celle des simulations enregistrées

La méthode [enregistrerSimulation] peut procéder ainsi :

- récupérer le n° de la dernière simulation dans le bean [SessionData] et l'incrémenter,
- ajouter la nouvelle simulation à la liste des simulations maintenue par la classe [SessionData],
- faire afficher le tableau des simulations

L'affichage du tableau des simulations est géré par le fichier [simulations.xhtml] fourni, dans lequel on peut voir que :

- la page [simulations.xhtml] s'insère à l'intérieur de la page [layout.xhtml], à la place du fragment nommé part1
- la balise <h:dataTable > utilise le champ #{sessionData.simulations} comme source de données, c.a.d. le champ « private List<Simulation> simulations »

Quelques précisions :

- l'attribut var="simulation" fixe le nom de la variable représentant la simulation courante à l'intérieur de la balise <h:datatable > ;
- l'attribut headerClass="simulationsHeaders" fixe le style des titres des colonnes du tableau ;
- l'attribut columnClasses="...." fixe le style de chacune des colonnes du tableau ;

Le code JSF de la colonne Nom est le suivant :

```
1. <h:column>
2. <f:facet name="header">
3. <h:outputText value="#{msg['simulations.headers.nom']}"/>
4. </f:facet>
5. <h:outputText value="#{simulation.feuilleSalaire.employe.nom}"/>
6. </h:column>
```

- lignes 2-4 : la balise <f:facet name="header" > définit le titre de la colonne ;
- ligne 5 : le nom de l'employé est écrit :

simulation correspond à l'attribut var de la balise <h:dataTable ...> : <h:dataTable value ="#{sessionData.simulations}" var ="simulation" ...>

- simulation désigne la simulation courante de la liste des simulations : d'abord la 1ère, puis la 2ème, ... ;
- simulation.feuilleSalaire fait référence au champ feuilleSalaire de la simulation courante ;
- simulation.feuilleSalaire.employe fait référence au champ employe du champ feuilleSalaire ;
- simulation.feuilleSalaire.employe.nom fait référence au champ nom du champ employe ;

La même technique est répétée pour toutes les colonnes du tableau. Il y a une difficulté pour la colonne Indemnités qui est générée avec le code suivant :

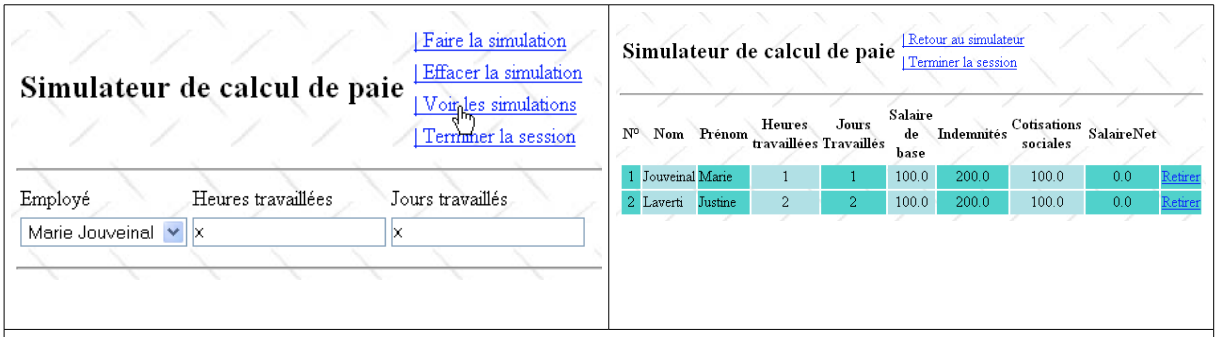
```
1. <h:column>
2. <f:facet name="header">
3. <h:outputText value="#{msg['simulations.headers.indemnites']}"/>
4. </f:facet>
5. <h:outputText value="#{simulation.indemnites}"/>
6. </h:column>
```

Ligne 5, on affiche la valeur de simulation.indemnites . Or la classe Simulation n'a pas de champ indemnites . Il faut se rappeler ici que le champ indemnites n'est pas utilisé directement mais via la méthode simulation.getIndemnites() . Il suffit donc que cette méthode existe. Le champ indemnites peut lui ne pas exister. La méthode getIndemnites doit rendre le total des indemnités de l'employé. Cela nécessite un calcul intermédiaire car ce total n'est pas disponible directement dans la feuille de salaire.

Travail à faire : écrire la méthode [enregistrerSimulation] de la classe [Form]. Vérifier que le retour au simulateur fonctionne dans le cas de la fonctionnalité d'affichage de l'erreur.

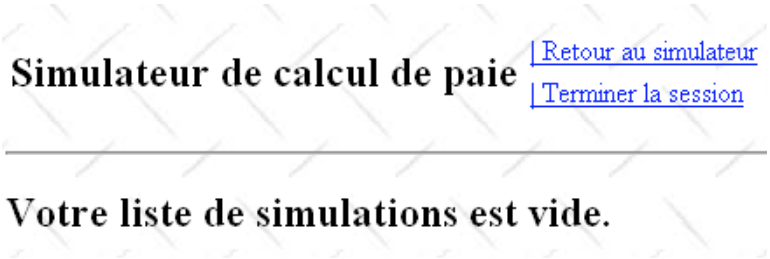
4.7 Action [voir simulations]

L'action [voirSimulations] associée au lien permet à l'utilisateur d'avoir le tableau des simulations, ceci quelque soit l'état de ses saisies :



Fonctionnalité d'affichage des simulations (à droite) depuis le simulateur (à gauche)

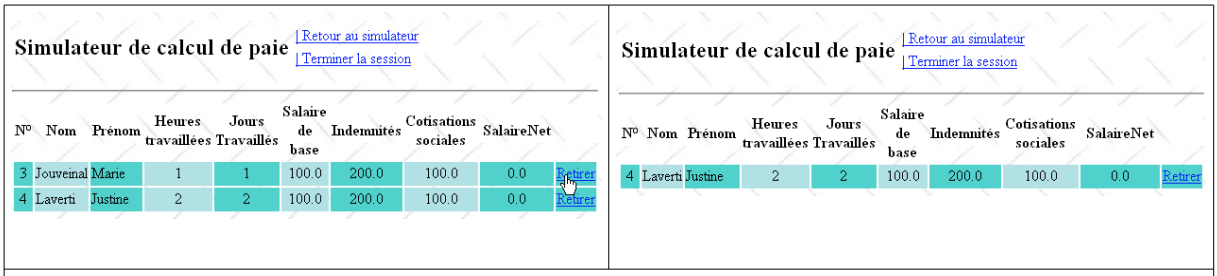
On fera en sorte que si la liste des simulations est vide, la vue affichée soit [vueSimulationsVides] (voir fichier simulationsVides.xhtml):



Travail à faire : écrire la méthode [voirSimulations] de la classe [Form].

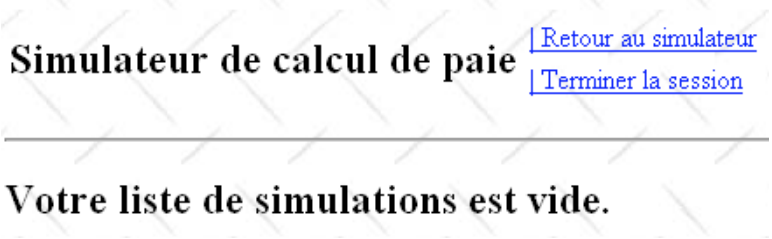
4.8 Action [retirer simulation]

L'utilisateur peut retirer des simulations de sa liste :



Avant (à gauche) et après (à droite) retrait d'une simulation

Si ci-dessus, on retire la dernière simulation, on obtiendra le résultat suivant :



Le code JSF de la colonne [Retirer] du tableau des simulations est le suivant :

```
1. <h:dataTable value="#{form.simulations}" var="simulation"
2. headerClass="simulationsHeaders"
```

```
columnClasses="simuNum,simuNom,simuPrenom,simuHT,simuJT,simuSalaireBase,simuIndemnites,simuCotisationsSociales,simuSalaire
Net">
3. ...
4. <h:column>
5. <h:commandLink value="Retirer" action="#{form.retirerSimulation}">
6.              <f:setPropertyActionListener target="#{form.numSimulationToDelete}" value="#{simulation.num}"/>
7. </h:commandLink>
8. </h:column>
9. </h:dataTable>
```

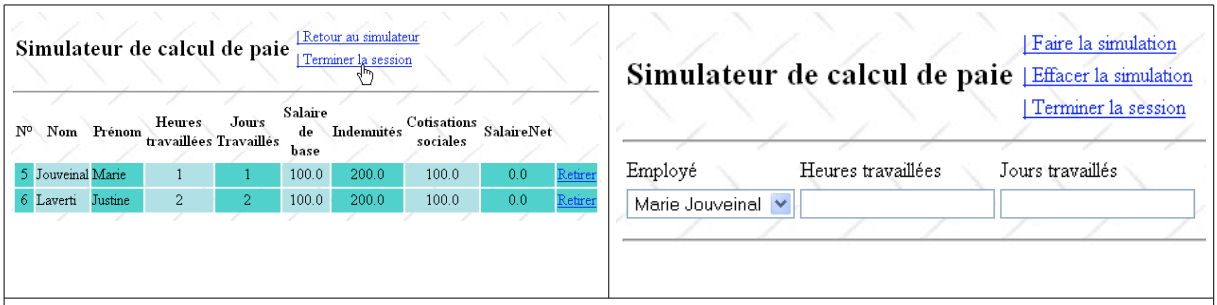
le lien [Retirer] (ligne 5) est associé à la méthode [retirerSimulation] de la classe [Form]. Cette méthode a besoin de connaître le n° de la simulation à retirer. Celui-ci lui est fourni par la balise <f:setPropertyActionListener >. Cette balise a deux attributs **target** et **value** : l'attribut target désigne un champ du modèle auquel la valeur de l'attribut value sera affectée. Ici le n° de la simulation à retirer #{simulation.num} sera affectée au champ numSimulationToDelete de la classe [Form] (« private Integer numSimulationToDelete; »).

Lorsque la méthode [retirerSimulation] de la classe [Form] s'exécutera, elle pourra utiliser la valeur qui aura été stockée auparavant dans le champ numSimulationToDelete, désigne le numéro de la simulation à retirer (chaque objet « simulation » embarque son numéro).

Travail à faire : écrire la méthode [retirerSimulation] de la classe [Form].

4.9 Action [terminer session]

L'action [terminerSession] associée au lien permet à l'utilisateur d'abandonner sa session et de revenir au formulaire de saisies vide. Si l'utilisateur avait une liste de simulations, celle-ci est vidée. Par ailleurs, la numérotation des simulations repart à 1.

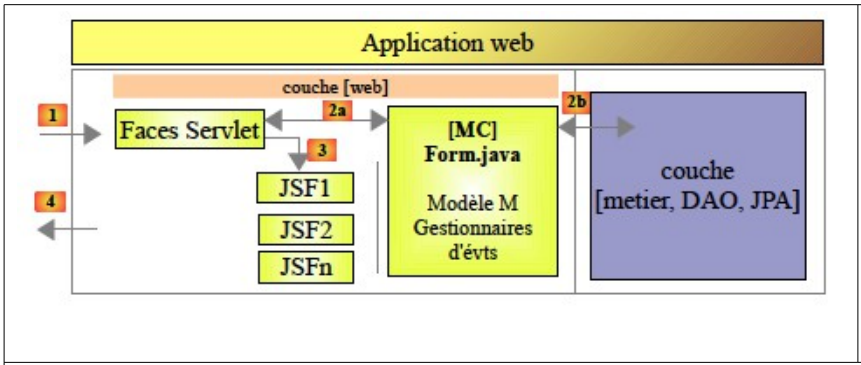


En terminant la session, on revient au formulaire de saisie (à droite), sachant que la liste de simulations (à gauche) est vidée.

Travail à faire : écrire la méthode [terminerSession] de la classe [Form].

5 Optionnel : intégration de la couche web dans une architecture 3 couches JSF / EJB

Travail à faire : remplacer la couche [métier] simulée, par les couches [métier, DAO, JPA] implémentées par des EJB (voir TD précédent)



Architecture intégrant l'EJB (à droite : les projets netbeans à créer)

Note : il faudra modifier le code du bean [ApplicationData.java] (voir TD précédent):

```
1. @Named
2. @ApplicationScoped
3. public class ApplicationData implements Serializable {
4.
5. // couche metier
6. @EJB
7. private IMetierLocal metier;
8. ...
```