Rapport de Projet JEE

ALLOUCH EVAN

Table des matières

1. Introduction

Dans le cadre du module JEE, j'ai développé une application web de gestion RH.
L'idée de départ du sujet est de permettre à une entreprise de gérer :

- Ses employés,
- Ses départements,
- Ses projets,
- Et les fiches de paie.

Par manque de temps, je me suis concentré sur un noyau fonctionnel réaliste autour des ressources humaines : la gestion des employés, des départements et des fiches de paie. Mon objectif était d'avoir une application qui tourne vraiment sur Tomcat, avec un CRUD complet sur les employés, plutôt que d'essayer de tout faire à moitié.

L'application a été réalisée en Java / JEE avec :

- Servlets,
- JSP,
- Un pattern MVC simple (model / dao / controller / views),
- Et un stockage des données en mémoire (DAO mémoire).

2. Besoins et périmètre choisi

2.1. Ce que demande le sujet

Le sujet complet prévoit beaucoup de choses :

- Gestion avancée des employés (CRUD, recherche, affectation aux projets, etc.) ;
- Gestion des départements ;
- Gestion des projets (création, suivi, affectation d'employés) ;
- Gestion des fiches de paie (calcul du net, consultation, impression) ;
- Éventuellement une partie sécurité (rôles, authentification) ;
- Plus tard : une version avec base de données (JDBC / Hibernate) puis avec Spring Boot.

2.2. Ce que j'ai réellement implémenté

- **Employés**
  - Liste des employés ;
  - Ajout d'un employé ;
  - Modification d'un employé ;
  - Suppression d'un employé ;
  - Gestion des champs : id, nom, prénom, poste, salaire de base, département.
- **Départements**
  - Classe `Département` ;
  - DAO en mémoire avec quelques départements (Informatique, RH, Finance…)
  - Page JSP qui liste les départements.
- **Fiches de paie**
  - Classe `FichePaie` (salaire de base, prime, déduction) ;
  - Calcul du salaire net dans le code : `net = base + prime - déduction` ;
  - DAO mémoire avec quelques fiches d'exemple ;
  - Page JSP qui affiche les fiches de paie et le salaire net.
- **Non réalisé (faute de temps)**
  - Gestion des projets ;
  - Connexion réelle à une base de données (tout est en mémoire) ;
  - Version Spring Boot de l'application ;
  - Authentification et rôles.

3. Conception

3.1. Modèle de données

Je suis parti d'un modèle de données assez classique pour une appli RH :

- **Employé**
  - `Id`
  - `Nom`
  - `Prénom`
  - `Poste`
  - `salaireBase`
  - `Département` (stocké ici sous forme de String)
- **Département**
  - `Id`
  - `Nom`
- **FichePaie**
  - `Id`
  - `Employé` (ou au minimum un lien vers un employé)
  - `Mois`
  - `salaireBase`
  - `Prime`
  - `Déduction`
  - Méthode `getSalaireNet()` qui fait le calcul.

En base de données, on aurait des relations 1–N entre Département et Employés, et 1–N entre Employé et FichePaie. Dans ma version, je simule tout ça en mémoire avec des listes Java.

3.2. Architecture MVC

J'ai organisé le projet en trois couches principales :

- **`model`** : classes métier (`Employe`, `Departement`, `FichePaie`, …)
- **`dao`** : interfaces et implémentations mémoire (pattern DAO)
- **`controller`** : servlets qui jouent le rôle de contrôleurs
- **`webapp`** : JSP pour l'affichage

Schéma logique de la manière dont ça fonctionne :

1. L'utilisateur clique sur un lien dans le navigateur (par ex. `/employes`).
2. L'URL arrive sur une Servlet (contrôleur).
3. La Servlet appelle un DAO pour récupérer ou modifier des données.
4. Le contrôleur met les données dans la requête (`request.setAttribute(...)`).
5. La requête est transférée vers une JSP, qui se charge de l'affichage HTML.

4. Implémentation des principales fonctionnalités

4.1. Gestion des employés

4.1.1. Modèle et DAO

Dans le package `model`, la classe `Employe` contient :

- `int id`
- `String nom`
- `String prenom`
- `String poste`
- `double salaireBase`
- `String departement`

Dans le package `dao` :

- `EmployeDao` définit les opérations :
  - `List<Employe> findAll()`
  - `Employe findById(int id)`
  - `void add(Employe e)`
  - `void deleteById(int id)`
- `EmployeDaoMemoire` implémente cette interface en utilisant une `List<Employe>` statique.
  Les trois premiers employés sont initialisés en dur dans un bloc `static`.
  Quand on ajoute un employé, l'ID est calculé comme "dernier ID + 1".

4.1.2. Liste des employés

La servlet `ListeEmployesServlet` :

- est mappée sur l'URL `/employes` ;
- appelle `employeDao.findAll()` ;
- met le résultat dans `request.setAttribute("listeEmployes", liste)` ;
- envoie vers `listeEmployes.jsp`.

La JSP `listeEmployes.jsp` :

- importe la classe `Employe` ;
- récupère la liste depuis la requête ;
- parcourt la liste et affiche un tableau avec :
  - id, nom, prénom, poste, salaire de base, département ;
  - actions "Modifier" et "Supprimer" sur chaque ligne.

4.1.3. Ajout d'un employé

- JSP : `ajoutEmploye.jsp`
  Contient un formulaire HTML classique avec les champs nom, prénom, poste, salaire de base, département.
  Le formulaire envoie les données en POST vers l'URL `/ajout-employe`.

- Servlet : `AjoutEmployeServlet`
  - lit les paramètres du formulaire ;
  - crée un objet `Employe` (avec un id provisoire) ;
  - appelle `employeDao.add(...)` ;
  - redirige vers `/employes`.

4.1.4. Modification d'un employé

- Lien "Modifier" dans `listeEmployes.jsp` pointe vers `/edit-employe?id=...`
- Servlet `EditEmployeServlet` :
  - récupère l'`id` dans les paramètres ;
  - appelle `employeDao.findById(id)` ;
  - met l'objet `Employe` dans la requête (`"employe"`) ;
  - forward vers `editEmploye.jsp`.
- JSP `editEmploye.jsp` :
  - pré-remplit le formulaire avec les valeurs de l'employé ;
  - envoie en POST vers `/update-employe`.
- Servlet `UpdateEmployeServlet` :
  - lit les nouvelles valeurs ;
  - met à jour les infos de l'employé dans la liste (dans ma version, je supprime puis je réinsère l'employé avec les nouvelles données) ;
  - redirige vers `/employes`.

4.1.5. Suppression d'un employé

- Lien "Supprimer" dans la JSP envoie vers `/supprimer-employe?id=...`
- La servlet `SupprimerEmployeServlet` lit l'id et appelle `deleteById(id)`.
- L'utilisateur est redirigé sur la liste actualisée.

4.2. Gestion des départements

La gestion des départements est plus simple : pour l'instant, je n'ai fait que l'affichage d'une liste.

- Classe `Departement` (id, nom).
- DAO mémoire `DepartementDaoMemoire` avec une liste statique de 3 départements.
- Servlet `ListeDepartementsServlet` mappée sur `/departements`.
- JSP `listeDepartements.jsp` qui affiche la liste dans un tableau.

L'objectif était surtout de montrer que l'application ne se limite pas aux employés, et prépare l'intégration de relations entre employés et départements.

4.3. Gestion des fiches de paie

4.3.1. Modèle

La classe `FichePaie` contient :

- `id`

- un lien vers un `Employe` (ou au moins ses infos de base),
- `mois`,
- `salaireBase`,
- `prime`,
- `deduction`.

La méthode :

```
public double getSalaireNet() {
    return salaireBase + prime - deduction;
}
```

fait le calcul demandé dans l'énoncé.

4.3.2. DAO et affichage

- `FichePaieDaoMemoire` :
  contient une liste statique de quelques fiches pour deux employés, avec des primes et des déductions.
- `ListeFichesPaieServlet` :
  récupère la liste, la pose dans la requête et forward vers `listeFichesPaie.jsp`.
- `listeFichesPaie.jsp` :
  affiche un tableau avec : id, nom de l'employé, mois, salaire de base, prime, déduction, salaire net.

5. Outils et environnement

- **IDE :** Eclipse IDE for Enterprise Java and Web Developers
- **Serveur d'application :** Apache Tomcat 10
- **Langage :** Java (Jakarta EE)
- **Technos côté serveur :** Servlets, JSP
- **Pattern :** MVC + DAO
- **Gestion de version :** Git + GitHub (dépôt contenant tout le projet)
- **Navigateurs utilisés pour les tests :** Firefox / Chrome

6. Limites actuelles et pistes d'amélioration

6.1. Limites

- Les données sont stockées en mémoire (lists Java) :
  si le serveur redémarre, tout est perdu.
  C'est suffisant pour une démo, mais pas pour une vraie appli.
- La partie Projets n'est pas implémentée :
  pas de création de projet, pas d'affectation d'employés à un projet.
- Il n'y a pas de sécurité :
  aucune authentification, pas de rôles (admin, chef de projet, etc.).
- La partie Spring Boot + JPA demandée dans le sujet n'a pas été faite, pour ne pas casser la version JEE qui fonctionne déjà.

6.2. Améliorations possibles

Si je devais continuer le projet, je ferais :

1. **Passage à une base de données relationnelle**
   - création des tables EMPLOYE, DEPARTEMENT, FICHE_PAIE, PROJET ;
   - remplacement des DAO mémoire par des DAO JDBC ou JPA.
2. **Utilisation d'Hibernate / JPA**
   - annotations `@Entity`, `@Id`, `@ManyToOne`, etc. ;
   - persistence.xml ;
   - vrais `EntityManager` pour la persistance.
3. **Version Spring Boot**
   - projet Spring Boot avec Spring Web + Spring Data JPA ;
   - controllers Spring à la place des servlets ;
   - possibilité d'ajouter une API REST pour les fiches de paie, par exemple.
4. **Gestion des projets**
   - création / modification / suppression des projets ;
   - ajout d'un écran pour affecter plusieurs employés à un projet.
5. **Authentification et rôles**
   - au minimum, un login / mot de passe pour l'admin ;
   - plus tard, gestion fine des droits par rôle.

7. Conclusion

Ce projet m'a permis de prendre en main concrètement le développement d'une application web en JEE :

- mise en place d'un projet dynamique sous Eclipse ;
- configuration de Tomcat ;
- utilisation de Servlets et JSP dans une architecture MVC ;
- mise en œuvre d'un CRUD complet sur une entité métier réelle (Employé) ;
- intégration d'autres entités (Département, Fiche de paie) pour montrer une vision plus globale de la gestion RH.

Même si tout le sujet n'est pas couvert (base de données, projets, Spring Boot), l'application livrée fonctionne et forme une base propre sur laquelle on peut facilement brancher une couche de persistance et ajouter des fonctionnalités.