Choix technologiques et Framework

Pour la réalisation de l'application nous avons choisi les technologies suivantes :

HSQL : La base de données est ainsi plus facile à stocker en mémoire et donc facilite le déploiement de l'application.

Criteria API: Pour la construction de requêtes plus facilement

PrimeFace : Fournit des balises supplémentaires qui sont des outils améliorant le design et l'ergonomie de l'application.

Hibernate : L'une des implementations de JPA les plus pratiques surtout avec les annotation de Filtre qui permet minimiser le travail sur les requêtes

Structure de Projet

Parmi les contraintes fonctionnelles qui sont imposées, la performance de l'application est l'une des plus importantes, et c'est pour cela que nous avons privilégié l'utilisation des Relations entre les entités en mode LAZY, c'est à dire que la récupération d'une entité ne signifie pas que ses relations seront récupérés c qui permet d'optimiser les temps de réponses de l'application.

Une autre méthode qui permet de faciliter l'utilisation de l'application et la rendre plus ergonomique et aussi plus performante c'est l'utilisation des requêtes AJAX pour la saisie de données ou les formulaires en général.

Les Entités

Dans le projet nous avons 2 entités : Personne, Activite et une enumeration NatureActivité

L'entité Personne est l'entité principale de l'application, vu que c'est elle qui contient les données principales qui serviront pour les filtres

```
@Id
@GeneratedValue()
Long idPerson;

@Column
@NotNull
String nom;

@Column
@NotNull
String prenom;

@Column
String siteweb;

@Column
String email;

@Column
String password;

@Column
@Past
Date dateNaissance;

// @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
@OneToMany(fetch = FetchType.LAZY, mappedBy = "referant")
Set<Personne> cooptations = new HashSet<Personne>();

@ManyToOne()
Personne referant;

@OneToMany(targetEntity = Activite.class, cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
private List<Activite> activites = new ArrayList<Activite>();
```

La classe Personne est reliée :

- a la classe Activite par une relation @OneToMany et la relation Inverse dans Activite
- avec elle même avec une relation @OneToMany pour gérer les cooptations et sa relation inverse @ManyToOne dans la même table

Pour les filtres nous avons utilisé les annotations d'hibernate :

```
@NamedQueries({ @NamedQuery(name = "findPersonne", query = "Select p from Personne p where p.idPerson= :value "),
        @NamedQuery(name = "findAllPersonnes", query = "Select p from Personne p "),
        @NamedQuery(name = "findAllPersonnesIn", query = "Select distinct p from Personne p where p.idPerson in (:liste)"),
        @NamedQuery(name = "findPersonsByLastName", query = "Select p from Personne p where p.nom= :nom"),
        @NamedQuery(name = "findPersonByFirstName", query = "Select p from Personne p where p.prenom= :prenom"),
        @NamedQuery(name = "findPersonneByEmail", query = "Select p from Personne p where p.email= :email"),
        @NamedQuery(name = "countAllPersonnes", query = "Select count(p) from Personne p ")

})
@Entity
@FilterDefs({ @FilterDef(name = "nom", parameters = { @ParamDef(name = "value", type = "string") }),
        @FilterDef(name = "prenom", parameters = { @ParamDef(name = "value", type = "string") }),
        @FilterDef(name = "email", parameters = { @ParamDef(name = "value", type = "string") }),

})
@Filters({ @Filter(name = "nom", condition = "lower(nom) like :value"),
        @Filter(name = "prenom", condition = "lower(prenom) like :value"),
        @Filter(name = "email", condition = "lower(email) like  :value")

})
```

Les requêtes ont été faites de sorte que les jointures sont toujours réalisées à la fin pour permettre de gagner du temps.

Les DAO

pour chaque Entité, nous avons prévu une Interface d'accès à ces Données pour réaliser les opérations relative à la base de données dans le package *dao*

- *IPersonneDao*
- *IActiviteDao*

ainsi que leur implémentation dans le package *impl*

Les Services

pour les services nous avons utilisé deux EJB

GuestService

un EJB Stateless qui se charge de realiser les operations qui nécessite pas que l'utilisateur soit connecté , ces opérations sont :

- la recherche
  - recherche de toutes les personnes
  - recherche de personnes en fonction de critères cumulables sur le nom, prénom de la personne, et le titre ou la description d'une activité
- l'inscription
- le comptages de personnes
- la génération aléatoire de personnes
- la récupération des CVs et cooptations

UserService

un EJB stateful et sessionScoped et contient aussi la personne actuellement connecté ; il gère les fonctionnalités :

- d'authentification
- de déconnexion
- de création / mise à jours / suppression des informations personnelles

- de création / mise à jours / suppression du CV
    - de création / mise à jours / suppression des personnes cooptés

Les deux services seront injectés dans les Managed Bean que nous avons préférés appeler Views dans le package' *views.*

Les vues

pour chaque page JSF nous avons utilisé des managed beans en portée vue(ViewScoped) pour gérer les interactions de l'utilisateur avec l'interface

- LoginView : la fenêtre de l'authentification
- SignupView : la fenêtre de l'inscription
- SearchView : l'affichage de la page de recherche
- UserSpaceView l'espace personnel de l'utilisateur :
- NavBarView : pour mettre à jour la barre de navigation selon l'etat de l'utilisateur

L'optimisation de l'application

La partie qui ralentit l'application était la page de recherche vu la quantité de données qu'elle charge en mémoire.
nous avons utilisé un DataTable de primefaces en mode LazyLoading, le LazyLoading est un mécanisme qui permet à ce tableau de récupérer que la page à afficher pour ne pas trop encombrer la RAM.

Dans notre implémentation de ce tableau nous avons privilégié la récupération par partie depuis la base de données, et ne pas tout charger en mémoire la première fois.C'est la partie la plus importante de notre application qui lui permet de supporter une quantité de données supérieur à 100000 personnes avec en moyenne 2 activités par personnes

La classe *LazyPersonneDataModel* qui est injectée dans le Datatable de la page de recherche aura comme dépendance injecté le GuestService pour avoir accès au données

```
// On injecte nous meme le service GuestService vu que l'annotation @Inject ne
// marche pas ici
public LazyPersonneDataModel(GuestService guestService) {
    this.guestService = guestService;
    otherFilters = new HashMap<String, String>();
    personneFilters = new HashMap<String, String>();

}
```

La fonction principale de cette classe est la fonction load qui permet de faire appels au GuestService avec les filtres saisis par l'utilisateur et aussi d'éviter les requêtes inutile a la base de données en gardant trace de l'intervalle de données charge dans le tampon data.

Dans cette fonction nous faisons appels a deux requêtes a travers le Guest Service

- La première c'est pour charger 5 page de données correspondantes aux filtres

```
data = new ArrayList<>();

// On recupere les donnees filtrees
data = guestService.filterPersonnes(personneFilters, otherFilters, first, pageSize);
```

:

- et la deuxième pour compter le nombre de personnes qui correspondent aux filtres en total vu que la première ne récupère que 5 pages

```
// On fait une autre requete pour compter le nombre de donnees en totales vu qu'on recupere que 5 pages
int dataSize = guestService.countAllPersonne(personneFilters, otherFilters);
this.setRowCount(dataSize);
```

Les deux requêtes ont été optimisés pour pouvoir s'exécuter en un temps raisonnable pour une grande quantité de données

1- La requête de recuperation de donnees :

Cette requête nous a permis d'optimiser les recherches, au lieu de faire une jointure puis faire le tri ; on passe par la table Activite et on applique les filtres d'activités et on récupère les Ids de personnes a qui appartiennent ces activités ensuite on applique les filtres sur ces personnes pour avoir le resultat finale, cette requête nous a permis de supporter plus de 10k personnes .

```
@Override
public List<Personne> getFilteredData(Map<String,String> filters, Map<String, String> activiteFilters,int first,int pageSize) {
    // On recupere la session hibernate pour appliquer les filtres predifinies
    org.hibernate.Session session =em.unwrap(Session.class);
    // Appliquer les filtres sur la table personnes
    applyFilter(session, filters);
    // Si on a pas de filtres sur les activites on fait une requete select all basique sur la table personne
    // Sinon on passe pas la table activite pour faire plus rapide et evite de faire une jointure couteuse
    if(!activiteFilters.isEmpty()) {
        // Recuperer une projection de la table activite vers IdPerson on appliquant les filtres sur les activites
        List<Long> a = this.activiteDao.getAssociatedPersonne((activiteFilters));
        // Si aucune acitvite ne satisfait les criteres on retourne null
        if(a.size()==0) return null;
        // Sinon on recupere toutes les pêrsonnes qui ont leur id dans la liste precedentes
        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Personne> cq = cb.createQuery(Personne.class);
        Root<Personne> root = cq.from(Personne.class);
        CriteriaQuery<Personne> vrai = cq.where(root.get("idPerson").in(a));
        TypedQuery<Personne> allQuery = em.createQuery(vrai);
        List<Personne> resultat = allQuery.getResultList();

        // On desactive les filtres Hibernates
        disableFilters(session, filters);
        return resultat;
    }else {

        TypedQuery<Personne> query = em.createNamedQuery("findAllPersonnes",Personne.class);
        List<Personne> resultList = query.getResultList();
        disableFilters(session, filters);

        return resultList;

    }


}
```

2- La requête pour compter le nombre de personnes

Pour calculer le nombre de personnes dans la table nous avons choisi d'utiliser la même méthode que pour la récupération de données, on passe par la table activites en premier.

```java
    @Override
    public int countAllPersonne(Map<String, String> filters, Map<String, String> activiteFilters) {
        // On applique les filtres de personnes dans la session
        org.hibernate.Session session =em.unwrap(Session.class);
        System.out.println("activite Filter vide == "+activiteFilters.isEmpty());
        applyFilter(session, filters);

// Si on a pas de filtres sur les activites on fait une requete count basique sur la table personne
// Sinon on passe pas la table activite pour faire plus rapide et evite de faire une jointure couteuse
        if(!activiteFilters.isEmpty()) {
            // Recuperer les ids de personnes ayant des activites qui match les criteres
            List<Long> liste = activiteDao.getAssociatedPersonne(activiteFilters);
            // Si aucune activite on retourne zero
            if(liste.size()==0) return 0;
            // Sinon on fait une requete sur la table Personne pour appliquer les filtres sur la table personne
            CriteriaBuilder cb = em.getCriteriaBuilder();
            CriteriaQuery<Personne> cq = cb.createQuery(Personne.class);
            CriteriaQuery<Long> vrai = cb.createQuery(Long.class);
            Root<Personne> root = vrai.from(Personne.class);
            vrai.select(cb.count(root));
            vrai.where(root.get("idPerson").in(liste)) ;
            TypedQuery<Long> allQuery = em.createQuery(vrai);
            Long x = allQuery.getSingleResult();
            disableFilters(session,filters);
            return x.intValue();
        }
        // On fait un count sur la table personne
        TypedQuery<Long> query = em.createNamedQuery("countAllPersonnes",Long.class);
        int intValue = query.getSingleResult().intValue();
        disableFilters(session, filters);
        return intValue;

    }
```

Tests

Avant propos
Ce document de tests a été développé par les étudiants de Master M2 Ingénierie de logiciel et de données d'Aix Marseille université.

Objet
Ce document spécifie l'ensemble des cas de test de l'application de gestion de CV pour les services des EJBs et des CRUDs de nos entités.

Responsabilités
Les responsables de ce document sont tous les membres du projet.

Éléments génériques
Nom de l'application : Gestion des contacts
Date de création du cahier : 13/12/2020
Auteurs de création du cahier : Victor IMBERT
Date de passage des tests : 13/12/2020
Nom des personnes qui ont effectué les tests : Victor, Djamel
Un indicateur du taux de réussite du cahier de tests : 0%

Descriptions des tests:

Pour garantir la fiabilité et la qualité logicielle de l'application, on a mis des tests unitaires avec Junit5. Le package "test.gestiont.test" : contient l'ensemble des tests.

CAHIER DE TESTS

Nom du test	**testLoginSuccess**
Description	On teste si un utilisateur arrive à s'authentifier
Résultats attendus	L'utilisateur est authentifié. La fonction "authentify" renvoie true
Résultat obtenu	Succès

Nom du test	**testLoginFaillure**

Description	On teste si un utilisateur n'arrive pas à s'authentifier avec des identifiants incorrects
Résultats attendus	L'utilisateur n'est pas authentifié. La fonction "authentify" renvoie false
Résultat obtenu	Succès

Nom du test	**testCRUDActivitesFromPerson()**
Description	Un utilisateur logué va utilisant les opérations CRUD des activités
Résultats attendus	pour chaque opération crud effectué, on a un trueAssert confirmant l'opération
Résultat obtenu	Succès

Nom du test	**testgetInfosWhenLogged**
Description	Un utilisateur s'authentifie et essaye de récupérer ses informations personnelles.
Résultats attendus	L'utilisateur récupère bien ses infos personnelles
Résultat obtenu	Succès

Nom du test	**testsetInfosWhenNotLogged**
Description	Un utilisateur essaie de récupérer des infos personnelles sans être logué
Résultats attendus	L'opération échoue. On attrape une exception du type "AccessInterditException"
Résultat obtenu	Succès

Nom du test	**testgetCooptations**
Description	Un utilisateur coopte une personne, et regarde si elle est bien enregistré

Résultats attendus	La personne cooptée est bien présente
Résultat obtenu	Succès

Nom du test	**testgetInfosAfterLogout**
Description	On se login et on se déconnecte ensuite
Résultats attendus	Quand on voudra faire une action qui requiert d'être authentifié, on attrape une exception du type "AccessInterditException"
Résultat obtenu	Echec : Can not call EJB Stateful Bean Remove Method without scoped @Dependent. Found scope: @SessionScoped

Test de GuestService

Nom du test	**testgetInfosAfterLogout**
Description	On se login et on se déconnecte ensuite
Résultats attendus	Quand on voudra faire une action qui requiert d'être authentifié, on attrape une exception du type "AccessInterditException"
Résultat obtenu	Echec : Can not call EJB Stateful Bean Remove Method without scoped @Dependent. Found scope: @SessionScoped