Cours 6 : ORM

Mardi 8 Décembre 2020

Antoine Flotte (aflotte@excilys.com)

Excilys
Développeurs de passion

SOMMAIRE

ORM ?

ORM ?

Object Relational Mapping

Fait une correspondance entre la partie Base de Données et la partie applicatif.

Avantages

On peut remplacer les requêtes SQL par du code Java, permettant une meilleur cohérence dans le code.

On peut mapper les résultats des requêtes directement en objet Java de façon plus ou moins automatisés.

Diminue la répétabilité du code

Inconvenients

Vous n’avez pas la main sur les requêtes SQL qui sont donc peu optimisé quand il y a beaucoup de jointure (vous pouvez toujours rentrez des requêtes SQL à votre ORM)

Vous devez bien définir vos modèles, évitez que vos requêtes ne charge des objets trop complet.

Exemples

Hibernate

Java Persistance Api

Spring Data JPA (code très épuré, idéal pour les POC, peu optimisé donc jamais dans un vrai projet)

En Java

Eager / Lazzy

Sur les jointures, Lazzy signifie que l'attribut sera à null tant que vous n’en avez pas besoin.

Avec Eager, il est directement chargé.

Modèle

Avec un ORM vos DAO vous renvoient un modèle et non un Pojo, puisque les ORM sont orienté objet (et les Pojo sont orienté base de donnée).

Il faut donc préciser des informations supplémentaire dans vos modèles.

Sur le modèle :

```
@Entity
```

Sur l'id de votre classe :

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

Sur vos jointures :

```
@ManyToOne @OneToMany(fetch = FetchType.LAZY)
@JoinColumn(name = "id_realisateur")
```

En Java

Exemple de modèle

```java
@Entity
public class Film {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String titre;
    private int duration;
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "id_realisateur")
    private Realisateur realisateur;
    public Film(){
    }
}
```

Pour les DAO, c'est extrèmement dépendant de votre ORM et de votre façon de faire.

Vous pouvez rester sur des requêtes SQL (surtout pour optimiser les grosses requêtes), utiliser un langage de requêtage type HQL (" from Film f "), utiliser des fonctions qui décrivent les actions (.SELECT() .WHERE() …)

Spring JPA est le plus simple à prendre en main.

Créer une DAO que vous nommez FilmRepository (pour pas confondre avec l'autre).

C'est une interface :

```
import com.ensta.myfilmlist.model.Film;
import org.springframework.data.jpa.repository.JpaRepository;
public interface FilmRepository extends JpaRepository<Film, Integer> {
}
```

Dans votre service vous pouvez faire :

```
@Autowired
    private FilmRepository filmRepository;
```

Puis vous pourrez utiliser :

```
filmRepository.findAll();
```

```
filmRepository.save(FilmMapper.filmDTOToFilm(film));
```

Et bien d'autres fonctions toutes faites. C'est évidement peut optimisé mais idéal pour faire rapidement un projet test avant de se lancer dans un gros chantier.

En Java

Exemple de Service

```
@Service
public class FilmService {
    @Autowired
    private FilmRepository filmRepository;
    public FilmService() {
    }
    public List<FilmDTO> findAll() throws ServiceException {
        List<Film> listFilmModel;
        listFilmModel = filmRepository.findAll();
        return FilmMapper.listFilmToListFilmDTO(listFilmModel);
    }
}
```

En Java

```
2020-12-08 07:11:24.471  INFO 1371 --- [nio-8082-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 10 ms
Hibernate: select film0_.id as id1_0_, film0_.duration as duration2_0_, film0_.id_realisateur as id_reali4_0_, film0_.titre as titre3_0_ from
film film0_
Hibernate: select realisateu0_.id as id1_1_0_, realisateu0_.nom as nom2_1_0_, realisateu0_.prenom as prenom3_1_0_ from realisateur realisateu0
  where realisateu0_.id=?
```

Problème de N+1

Quand vous avez deux tables liés par une jointure, vous pouvez rencontrer le problème des N+1.

Il s'agit du fait que vous chargez un élément puis que vous décidiez de charger ses attributs, ce qui sera fait avec plusieurs requêtes (donc très coûteux).

Il existe des moyens de s'en prémunir, qui dépendent de votre ORM.

Problème de N+1

Il existe même sans ORM, vous l'avez remarqué sur le remplissage des Realisateur pour les films. Dans notre cas c'est limité à 10 (à cause de la pagination), donc peu d'impact.

Vous pouvez quand même soit avoir une fonction qui renvoie les Réalisateurs pour une liste d'id (avec une HashMap) donc un seul appel, et ensuite remplir les Réalisateurs grâce à leur id et cette HashMap

Une autre façon de faire, plus optimisé pour les gros projets est dans votre DAO d'utiliser le EntityManagerFactory.

```
    @Autowired
    EntityManagerFactory emf;
public List<Film> findAll() throws DaoException {
            EntityManager em = emf.createEntityManager();
            TypedQuery<Film> query = em.createQuery("from Film f", Film.class);
            List<Film> result = query.getResultList();
            return result;
        }
```

En Java

Avec Spring-Boot

```
    public void create(Film film) {
        EntityManager em = emf.createEntityManager();
        System.out.println(film.getRealisateur().getId());
        em.getTransaction().begin();
        em.persist(film);
        em.getTransaction().commit();
    }
    public void update(Film film) {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        em.merge(film);
        em.getTransaction().commit();
    }
```

```
    public void delete(long id) throws DaoException {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        em.remove(em.find(Film.class, id));
        em.getTransaction().commit();
}
```

Si vous souhaitez faire des requêtes, le langage de requêtage est ici le JPQL