Trinôme : Guerin Yoann

Mame Birame Sene

Ruffet Vincent

# Architecture Logicielle Distribuée :

Outil d’aide à la gestion du patrimoine culturel d’un musée

Sommaire

[Architecture Logicielle Distribuée : 1](#_Toc347332956)

[Architecture 3](#_Toc347332957)

[Implémentation 4](#_Toc347332958)

[JPA 4](#_Toc347332959)

[Hibernate 4](#_Toc347332960)

[Filter 4](#_Toc347332961)

[Stratégies de mapping objet-relationnel 4](#_Toc347332962)

[Cardinalités 5](#_Toc347332963)

[RESTeasy (JAX-RS + JAXB) 6](#_Toc347332964)

[Services 6](#_Toc347332965)

[JAXB 6](#_Toc347332966)

[Bases de données 6](#_Toc347332967)

[MySQL 6](#_Toc347332968)

[Profils 6](#_Toc347332969)

[C3p0 +JNDI 6](#_Toc347332970)

[Tests unitaires 6](#_Toc347332971)

Back-End

Technologies utilisées: Hibernate, RESTeasy, Jetty, Maven

# Architecture

# architecture.png

# Implémentation

## JPA

### Hibernate

L’ORM que nous avons choisi d’utiliser est Hibernate.

Pour persister les données, on peut soit utiliser les EntityManager (avec persist()), soit utiliser une session (qui est spécifique à  hibernate) et faire un save().

Il peut être judicieux d'utiliser l'API JPA (standard JEE 🡺augmente la portabilité du projet. ex : si on change de framework de persistance, on n'a pas de changement à faire) plutôt que celle d'hibernate.

### Filter

Quand l'application est déployée, l'entityManagerFactory est crée (un seul pour l'application entière), et est fermé quand l'application est "détruite".

Chaque thread qui appelle JPAUtil pour avoir un nouvel entityManager en a un nouveau qui dure le temps de la requête HTTP : Un entityManager par requête HTTP.

Quand la requête est terminée (close()), l'entityManager est retourné au pool pour être réutilisé plus tard.

Le filter pemet de gérer plusieurs utilisateurs en même temps sur notre application.

### Stratégies de mapping objet-relationnel

Pour passer du modèle objet au modèle relationnel nous avons du choisir entre différentes stratégies afin de pouvoir conserver la notion d’héritage dans le modèle objet et qu’on ne perde pas la notion en base.

#### Œuvre -> Peinture / Sculpture / Photographie

Nous avons choisi de prendre la stratégie **SINGLE TABLE** du fait que les insertions et requêtes de sélection ne sont pas couteuses. Quelques attributs des classes filles (Photographie, peinture, ..) sont rajoutées dans la classe mère (ils sont peu nombreux), quelques attributs sont spécifiques à  une classe fille ce qui implique qu'ils ne seront pas renseignés pour certaines classes concrètes.

Nous n'avons pas choisi la méthode **TABLE\_PER\_CLASS** car cela impliquerait de devoir dupliquer les informations de la classe mère dans chacune de ses classes concrètes.

Au niveau des requêtes de sélection sur la classe mère cela les alourdit en créant des jointures pour accéder aux classes concrètes (ex : le conservateur veut connaitre toutes les œuvres d'une collection 🡺 Comme on a persisté que les classes concrètes, on est obligé de faire une jointure entre toutes les classes concrètes).

Nous n'avons également pas choisi la méthode **JOINED** bien qu'elle soit proche du modèle objet car cela alourdit les insertions en base (2 requêtes par insertion, insérer d'abord les informations dans la classe mère puis dans la classe fille) et les requêtes.

#### Employé / artiste

Nous avions choisi de créer un classe abstraite *AgentMusée* et les classes concrètes conservateur / libraire. Notre stratégie était **TABLE PER CLASS** afin de rentrer en base les classes concrètes. En effet nous n'avions pas besoin d'effectuer de requêtes (jointures) pour récolter des informations sur un ensemble de personnes. Un employé n’a pas à effectuer de requête sur ses collègues.

Finalement nous avons décidé de ne faire qu’une classe Employé (avec un statu (enum) qui peut être conservateur/libraire).

🡺C'est plus pratique pour le web service de connexion, il retourne directement le statu pour savoir à quelle section on peut accéder sur le client.

### Cardinalités

#### Œuvre et reproduction :

🡺 OneToOne unidirectionnel

Une reproduction connait l'œuvre qu'elle reproduit, tandis qu'une œuvre n'est pas consciente de ses reproductions.

Une œuvre peut avoir plusieurs reproductions, une reproduction ne reproduit qu'une œuvre.

#### Collection et oeuvre :

🡺 n:m ManyToMany unidirectionnel

Une œuvre ne sait pas qu'elle est dans une collection, une collection connait toutes les œuvres qu'elle contient.

Une collection possède une ou plusieurs œuvres, une œuvre peut appartenir a plusieurs collections.

#### Oeuvre et artiste :

🡺 1:n bidirectionnel

Une œuvre sait par quel artiste elle a été faite, un artiste a conscience de ses œuvres (liste d'œuvres).

Une œuvre possède une clé étrangère sur Artiste (JoinColumn)

Une œuvre est faite par un artiste, un artiste peut faire plusieurs œuvres.

La suppression d’un artiste entraine la suppression de ses œuvres : orphanRemoval=**true.**

De plus nous avons choisi d’utiliser une HashSet pour stocker les œuvres de l’artiste afin qu’il n’y ait pas de doublons.

#### Commentaires/tags et Œuvre/Collection

On ne veut pas créer de classe Commentaire juste parcequ'on en a plusieurs (List) or on ne peut pas le représenter en base.

Depuis JPA 2.0 c'est possible avec @ElementCollection, ça crée une table supplémentaire (ex : collection\_commentaire) contenant l'id de la collection et le commentaire => on n’a pas eu besoin de créer une classe commentaire et faire une relation OneToMany (ManyToOne).

#### Œuvre /Photo

🡺OneToMany bidirectionnel

## RESTeasy (JAX-RS + JAXB)

### Services

### JAXB

//parler de moxy + éviter les cycles infinis

## Bases de données

### MySQL

Nous utilisons deux bases de données : Une base de prod et une base de tests. Il s’agit de deux bases MySQL.

Pour savoir comment les utiliser voir : <https://github.com/isylhdin/ALD_Projet/wiki/Utiliser-une-base-de-données>.

Les bases doivent être créées par l’utilisateur et être vides. Lorsque Jetty est lancé alors les tables seront créées automatiquement grâce aux annotations de JPA.

### Profils

Pour pouvoir switcher entre nos deux bases nous avons mis en place un système de profils.

Voir : <https://github.com/isylhdin/ALD_Projet/wiki/Environnements-de-prod-et-tests>

### C3p0 +JNDI

Les pools de connexion sont bons pour les performances, ça évite à l’application de créer une connexion à chaque fois qu’on interagit avec la base de données et minimise le coût d’ouverture/fermeture des connexions.

Hibernate fournit un pool de connexion interne mais il n’est pas approprié pour la production.

## Tests unitaires

Nos DAO ont été testé un à un avec la base de tests.

Au fur et à mesure de notre avancement nous avons rencontré un problème.

Alors qu’avant un entityManagerFactory + un entityManager étaient crées à chaque opération sur nos objets, on n'avait pas de problème (même si ce n'était pas du tout optimisé de faire comme ça) avec notre base de prod et base de test.

Désormais, avec la mise en place du Filter c'est dans la classe JPAFilter qu'on distribue des entityManagers.

C'est-à-dire qui si on ne passe pas par des requêtes il est impossible d’obtenir un entitymanager.

Il a donc été nécessaire de tester nos DAO à travers des requêtes http. Dans nos tests unitaires nous testons donc aussi nos services REST (ça englobe tout).

Pour avoir un client http, nous avons utilisé le client RESTeasy et marshallé nos données en XML.

Front-End

Technologies utilisées : GWT, Piriti (essai avec Restlet et Errai mais nous ne l’avons pas intégré à notre solution finale).