Trinôme : Guerin Yoann
Mame Birame Sene
Ruffet Vincent

Architecture Logicielle Distribuée :

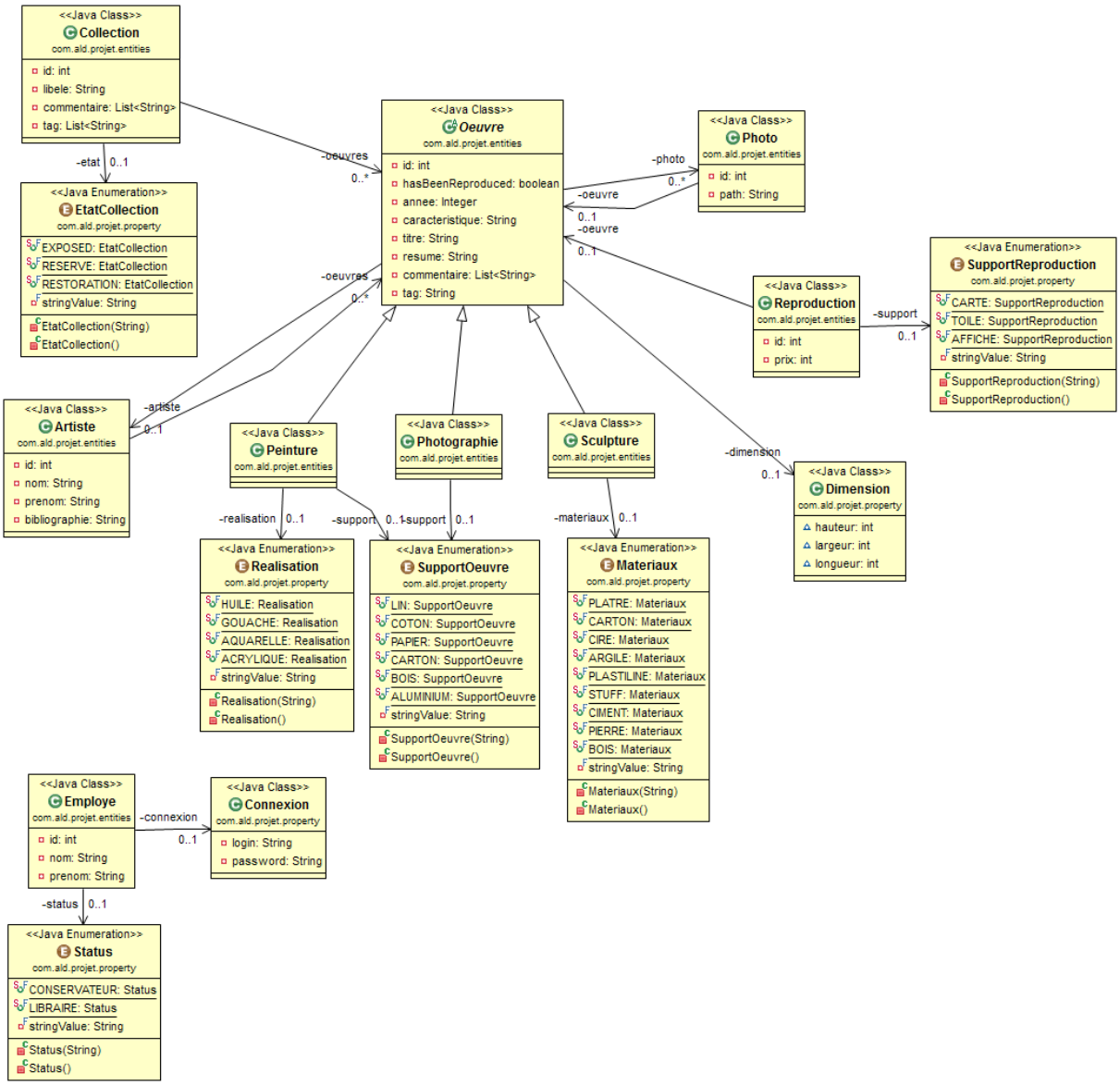
Outil d'aide à la gestion du patrimoine culturel d'un musée

Sommaire

Back-End

Technologies utilisées: Hibernate, RESTeasy, Jetty, Maven

Architecture



Implémentation Back-End

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.

Lorsqu'une œuvre est supprimée, il faut aussi la supprimer des collections qui l'utilisent. Etant donné que la relation est unidirectionnelle (l'œuvre n'a pas de champ collection) on ne peut pas utiliser simplement la suppression en cascade. Une table intermédiaire a été créée et contient les clés étrangères *Collection_id et Œuvre_id*. Comme cette table ne fait pas référence à une entité, il n'est pas possible de faire de requête dessus ➔ JPQL permet de faire des requêtes sur les objets.
Il faut donc récupérer toutes les collections qui la contiennent, supprimer les références dessus, mettre à jour les collections, et enfin supprimer l'œuvre.

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

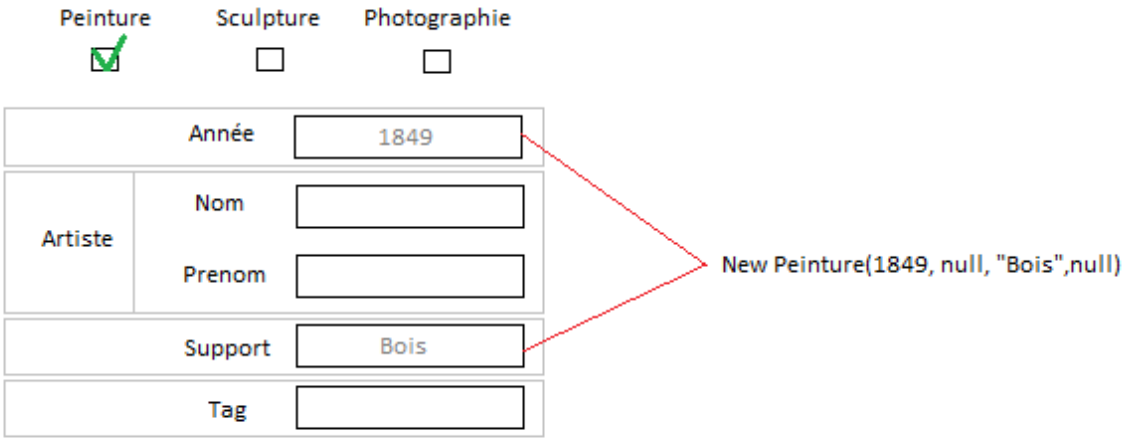
On ne veut pas créer de classe Commentaire juste parce qu'on en a plusieurs (List) or on ne peut pas le représenter en base.
Depuis JPA 2.0 c'est possible avec @ElementCollection, ça crée une table supplémentaire (ex : *collection_commentaire*) contenant l'id de la collection et le commentaire => on n'a pas eu besoin de créer une classe commentaire et faire une relation OneToMany (ManyToOne).

Œuvre /Photo

➔OneToMany bidirectionnel

Criteria

Pour former des requêtes dynamiques nous avons utilisé l'API criteria. L'utilisateur remplira un formulaire pour rechercher toutes les œuvres en fonction de certains critères.



Une œuvre rassemblant les champs remplis (année et support) est créée coté client. L'œuvre est ensuite associée au body de la requête POST pour consomme le web service voulu. L'œuvre n'est pas persistée coté serveur, elle sert juste d'intermédiaire pour construire la requête dynamique (remplir les différents prédicats).

Ex:

```
if(oeuvre.getAnnee() != null){
	Predicate annee = criteriaBuilder.equal(root.get("annee"), o.getAnnee());
	predicateList.add(annee);
}
```

Fetch

Les stratégies de téléchargement par défaut n'ont pas été changées dans l'ensemble, on est toujours en mode LAZY.

Explications de ce qu'il se passe avec le mode Lazy :

```
Artiste artiste = new Artiste ("puma", "guerin", "really good artiste");
```

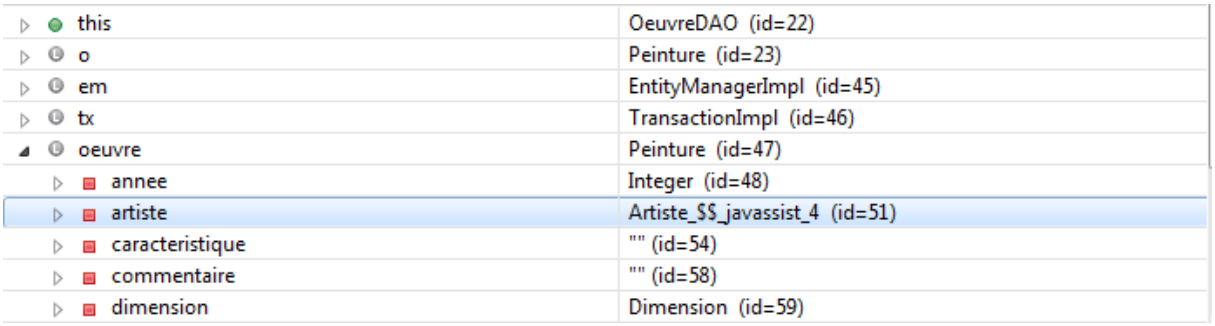
```
Peinture p = new Peinture (dimension, false, artiste, 2010, "", "bla", "bla", "",
"", SupportOeuvre.BOIS, Realisation.ACRYLIQUE);
oeuvreDAO.createOeuvre(p);
```

Pour rappel, la classe Œuvre a un attribut Artiste :

```
@ManyToOne
@JoinColumn(name="artiste_id")
private Artiste artiste;
```

➔Nous allons voir l'effet de FetchType.Lazy en mode debug

```
Oeuvre oeuvre =  em.find(Oeuvre.class,o.getId());
```



On a chargé tous les attributs de l'œuvre sauf l'artiste (qui contient lui-même une liste d'œuvre)

➔Il ne serait pas judicieux de télécharger toutes les autres œuvres de l'artiste afin d'avoir le graphe complet.

On voit artiste : Artiste_$$_javassist_4 (id=51) : il s'agit d'un proxy.

Les données sont téléchargées de manière transparente. A chaque fois qu'on voudra récupérer un élément de la liste une requête est générée sur la base de données.

RESTeasy (JAX-RS + JAXB)

Services

Les services sont déclarés dans une interface afin de séparer les annotations JAX-RS du code d'implémentation.

Voici les services REST que nous avons implémenté :

- Connexion
- Création d'objet (Œuvre/collection/Reproduction/photo/artiste/employé)
- Mise à jour d'un objet (Œuvre/Collection)
- Suppression d'un objet (Collection, Œuvre, Artiste)

- Affichage d'un ou plusieurs objets (toutes les collections, contenu d'une collection, toutes les œuvres, contenu d'une œuvre, tous les artistes, un artiste en particulier, toutes les œuvres d'un artiste)
- Affichage des œuvres en fonction de certains critères ➔criteria
- Affichage des différents services demandés pour le libraire :
    - Connaitre les œuvres qui n'ont pas de reproductions / qui ont une (des) reproduction(s)
    - Lister toutes les reproductions d'œuvre
    - Connaitre l'état des stocks pour les reproductions d'une œuvre
    - Fixer le prix d'une reproduction
    - Afficher toutes les reproductions des différentes œuvres constituant une collection
    - Afficher toutes les œuvres d'une collection qui n'ont encore jamais été reproduites

  ➔Pour effectuer ces requêtes nous avons utilisé le langage **JPQL** (jointures, sous-requêtes, existence, ..)

Etant donné que l'architecture REST n'utilise pas de WSDL pour décrire les services mis en place, c'est au développeur de fournir sa documentation pour les personnes souhaitant consommer son API.
Nous avons créé un wiki sur notre github pour détailler nos services :
https://github.com/isylhdin/ALD_Projet/wiki/Services-REST

JAXB

Nos classes utilisent JAXB pour le Marchalling/Unmarshalling, les objets sont rapatriés sur le client au format xml.

Lorsque des objets ont des relations bidirectionnelles et qu'ils sont demandés cotés client, le graphe d'objets induit un cycle infini (une œuvre a un artiste, un artiste a des œuvres, ses œuvres l'ont pour artiste, …). Nous avons voulu utiliser MOXy au début afin d'avoir des annotations que JAXB « vanilla » ne proposait pas, et garder une référence du champ qui créait le cycle infini.

Nous avons fini par utiliser simplement @XmlTranscient et le problème a été réglé.

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.
Les différents paramètres sont à spécifier dans le persistence.xml

Ex : crée 5 connexions dans le pool comme spécifié dans persistence.xml :

```
<property name="hibernate.c3p0.min_size" value="5" />
```

```
[main] DEBUG BasicResourcePool:404 - incremented pending_acquires: 1
[main] DEBUG BasicResourcePool:404 - incremented pending_acquires: 2
[main] DEBUG BasicResourcePool:404 - incremented pending_acquires: 3
[main] DEBUG BasicResourcePool:404 - incremented pending_acquires: 4
[main] DEBUG BasicResourcePool:404 - incremented pending_acquires: 5
[main] DEBUG BasicResourcePool:289 - com.mchange.v2.resourcepool.BasicResourcePool@10385cb0 config: [start -> 5; min
-> 5; max -> 20; inc -> 3; num_acq_attempts -> 30; acq_attempt_delay -> 1000; check_idle_resources_delay -> 3000000;
mox_resource_age -> 0; max_idle_time -> 300000; excess_max_idle_time -> 0; destroy_unreturned_resc_time -> 0;
expiration_enforcement_delay -> 75000; break_on_acquisition_failure -> false; debug_store_checkout_exceptions ->
false]
[main] DEBUG BasicResourcePool:538 - acquire test -- pool size: 0; target_pool_size: 5; desired target? 1
```

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).

Outils permettant de checker les requêtes HTTP : Firebug, RESTc

Implémentation Front-End

Piriti

Nous cherchions un framework de sérialisation/désérialisation utilisant le XML. En effet la plupart utilisent désormais le format JSON et ne proposent pas de format XML ➔RestyGWT, Javascript overlay types, AutoBean.

Piriti sert donc de mapping XML <-> Pojo, il est très facile à utiliser :

- Il suffit de définir une interface de type XmlReader`<T>` `et` XmlWriter`<T>` dans les nos classes d'entités.
- La sérialisation se fait en une ligne :
  ```
  String xml = Artiste.WRITER.toXml(instanceArtiste);
  ```

- La désérialisation est tout aussi simple
  ```
  Artiste artiste = Artiste.XML.read(reponseDuWebService);
  ```

Errai

Une autre solution un peu plus lourde consistait à utiliser Errai. Coté client il faut juste créer une interface des services que l'on veut proposer, et une classe contenant l'implémentation des services ➔ Les annotations JAX-RS sont séparées du code d'implémentation, ce n'est pas spécifique à Errai on peut aussi coder de cette façon pour rendre le code plus clair.

Le front end doit avoir une copie de cette interface, c'est-à-dire que si on ajoute/supprime un service ou qu'on en modifie la signature il faut répercuter les changements dans l'interface du front-end. Les appels aux services REST sont très simples à faire, le code est assez minimal.

Errai permet de sérialiser et désérialiser des POJO, il n'y a pas besoin de framework supplémentaire. Nous avons réussi à faire marcher le back-end avec Errai mais la version finale de notre projet ne l'utilise pas, nous utilisons l'API de GWT pour envoyer des requêtes HTTP (RequestBuilder) + Piriti

Problèmes rencontrés

Same Origin Policy

C'est l'erreur qui nous a fait perdre le plus de temps dans le développement de l'application. En pensant avoir résolu ce problème nous en avons créé une autre plus difficile à interpréter.

Les browsers web implémentent un model de sécurité appelé « Same Origin Policy ». Un code javascript qui s'exécute sur une page web peut ne pas interagir avec des ressources qui ne proviennent pas du même site web. Bien qu'elle empêche de se faire voler des informations par des hackers, elle rend la tache plus difficile au développeur.

Ceci n'est pas spécifique à GWT, ça s'applique à toute application en AJAX.
Voici comment ça affecte GWT :
Les appels http sont limités à des url pointant sur des ressources hébergées sur le même serveur qui contient la page hôte. Cela signifie qu'il n'est pas possible de faire de requête de type AJAX sur un autre serveur, ou sur un même serveur avec des ports différents. Dans notre cas, Jetty tourne sur le port 8080 et GWT (qui utilise lui-même Jetty) sur le port 8888 ➔ crée une « Same Origin Policy ».

Nous avons donc mis en place un proxy coté client pour régler le problème.

Modification du content-type des requêtes

En utilisant un proxy il était possible d'effectuer des requêtes GET sans problème. Les ennuis sont apparus avec les POST contenant des objets sérialisés.

Alors qu'on envoie du XML depuis le client avec un POST (la sérialisation marche comme il faut), le serveur reçoit du x-www-form-urlencoded.