Crémetz Olivier
Lemrabet Youness

GIS 2

RAPPORT D'ARCHITECTURES LOGICIELLES

« GESTION DE MUSEES VIRTUELS »

Polytech'Lille – Département Génie Informatique et Statistiques
Année 2005-2006

SOMMAIRE

INTRODUCTION

Dans le cadre du projet d'architectures logicielles de 2ème année, nous devons concevoir et implémenter une application client-serveur de type 3-tiers permettant de gérer virtuellement des musées.
Nous devons concevoir et réaliser une application J2EE qui administre et permet la consultation d'oeuvres d'art déjà référencées et accessibles sur internet.
Afin de faciliter la gestion de ces données, nous avons développé une application J2EE accessible en mode client distant et en site internet en utilisant Postgres et le langage JAVA(JSP, EJB 3.0). Nous déploierons notre application sur un serveur JBoss

Vous trouverez notre site sur le serveur de Polytech'Lille à cette adresse :
http://weppes.studserv.deule.net:8080/ocremetzMusee/

Nous exposerons dans un premier temps les éléments de conception de cette application. Puis, nous présenterons un guide d'utilisation de l'application..

I) OBJET DE L'ETUDE

Le but de ce projet est de réaliser une application J2EE suivant des critères stricts, c'est à dire de l'idée à la réalisation.
De ce fait, nous allons apprendre à gérer de nombreux paramètres :
Le premier est évident et découle de la signification même du nom commun projet : se projeter dans le futur. Donc il faudra anticiper et savoir par avance ce que l'on sera capable de faire.
Le deuxième est d'apprendre à gérer le temps : calendrier de remise du tp.
Le troisième est d'apprendre à travailler de manière collective : intégrer un travail réalisé individuellement dans un projet collectif.
Donc l'intérêt de notre projet est clair :

1) Ouverture à la programmation d'application J2EE
2) Découverte d'une situation de travail analogue à celle rencontrée en entreprise
3) Pouvoir se servir de notre propre programme

II) ANALYSE ET CAHIER DES CHARGES

Pour cette application nous nous intéressons à deux aspects majeurs :

- La gestion des oeuvres et artistes
- La gestion des musées virtuels

A) Gestion des oeuvres et artistes

L'application doit permettre la consultation d'oeuvres d'art déjà référencées et accessibles sur internet. L'oeuvre est identifiée par son url et caractérisée par un titre et une année de création.
L'oeuvre est réalisée obligatoirement par un seul artiste. Un artiste est lui même caractérisé par son nom et son prénom. Il sera identifié par son nom.
Une oeuvre doit pouvoir être consultée, ajoutée, supprimée et éditée (uniquement par son titre et son année de création).
On doit prévoir le seulement l'ajout et la consultation des artistes.
On doit pouvoir consulter toutes les oeuvres et toutes les oeuvres pour un artiste donné.

Les images seront consultées seulement sur la partie WEB JSP à l'aide de la balise HTML img. On prévoira la création des artistes et des oeuvres à l'aide d'un programme Java. Le reste des fonctionnalités seront développées uniquement par des pages web.

B) Gestion des musées virtuels

L'application J2EE doit permettre de construire des musées virtuels. Un musée est identifié par un nom de thème. Chaque musée permettra de recenser les oeuvres d'un thème quelconque. Un oeuvre pourra être partagée par plusieurs musées virtuels. L'application J2EE web devra permettre de créer un musée, d'affecter une oeuvre à un musée et consulter un musée donnée.

III) CONCEPTION DE LA SOLUTION

A) Les principales entités

En analysant le sujet donné, nous avons pu discerner trois entités principales composées de différents attributs. Ces composants vont nous permettre de mettre en oeuvre un mécanisme de relation entre les objets et de persistance sur une base de données postgres.

Pour faire le lien entre les classes et la base de données, nous utilisons les annotations EJB 3.0 suivantes :

@Table (name= nomTable) : pour définir le nom de la table associée
@Entity (access=AccessType.PROPERTY) : pour déclarer le composant persistant. On accède aux champs suivant les propriétés get et set.
@Id : pour définir la clé primaire.
@Column(length=100): pour définir le taille de chaque champs. (On peut aussi déclarer un nom de colonne)

Pour chacune de ces classes, nous avons mis en place des méthodes get et set afin de pouvoir accéder et modifier les données. Ces champs seront déclarés privés et ces méthodes seront publiques.

1) L'entité Oeuvre

Une oeuvre est caractérisée par son titre et une année de création. Elle est identifiée par l'url de l'image de l'oeuvre. L'entité conservera un lien sur l'entité Artiste et la collection de tous les musées auquel cette oeuvre est affectée.

2) L'entité Artiste

Un Artiste est caractérisé par son nom et son prénom. Il est identifiée par son nom qui sera unique. 2 artistes n'auront donc pas le même nom.
L'artiste possède la collection de toutes ses oeuvres.

3) L'entité Musée

Un musée est identifié par son nom de thème qui sera unique. 2 musées n'auront donc pas le même thème. Le musée possédera la collection d'oeuvres qui lui seront affectées.

4) *Mise en oeuvre des relations*

La programmation de ces relations nécessitent les annotations EJB 3.0 suivantes
Pour relier une Oeuvre à un Artiste, nous avons la relation suivante : « Une oeuvre a été crée par un seul artiste »
Ce mécanisme est représentée par l'annotation suivante :

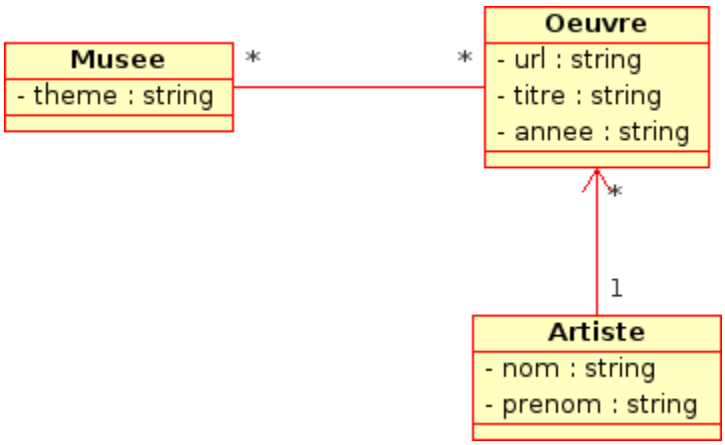
- Dans la classe OeuvreBean :
  L'oeuvre fait référence à un ArtisteBean donné :
  ```
  private ArtisteBean artiste;
  ```
  La relation est modélisée de la façon suivante :
  ```
  @ManyToOne(optional=false) @JoinColumn(name="ref_artiste")
  public ArtisteBean getArtiste()
  ```
  Dans la base de données, la table ocremetz_oeuvre aura une colonne nommée ref_artiste qui contiendra la clé primaire sur l'artiste donné. L'oeuvre doit obligatoirement appartenir à un artiste donné (optional=false).

- Dans la classe ArtisteBean :
  L'artiste conserve la collection des Oeuvres qu'il a créés.
  ```
  private Collection<OeuvreBean> oeuvres;
  ```
  La relation est modélisée de la façon suivante :
  ```
  @OneToMany(mappedBy="artiste",cascade=CascadeType.ALL)
  public Collection<OeuvreBean> getOeuvres()
  ```
  Si l'artiste est supprimé, les oeuvres associées seront aussi enlevée de la base grâce au type CascadeType.ALL.

La Relation entre un Musée et une Oeuvre est représentée comme ceci : « Une oeuvre peut être partagée par zéro ou plusieurs musées Virtuels et un musée peut contenir zéro ou plusieurs Oeuvres »
La programmation de cette relation consiste à déclarer une association multiple entre Oeuvre et Musée.

- Dans la classe MuseeBean :
  On ajoute la collection des Oeuvres comme ceci:
  ```
  private ArtisteBean artiste;
  ```
  Puis pour la relation multiple :
  ```
  @ManyToMany(fetch = FetchType.EAGER,cascade=CascadeType.ALL)
  public Collection<OeuvreBean> getOeuvres()
  ```
- Dans la classe OeuvreBean :
  On ajoute la collection des Musées auquel l'oeuvre appartient.
  ```
  private Collection<MuseeBean> musees ;
  ```
  Puis pour la relation multiple :
  ```
  @ManyToMany(fetch = FetchType.EAGER,mappedBy="oeuvres")
  public Collection<MuseeBean> getMusees()
  ```

5) *Schéma UML de L'application*



B) Conception de l'application: composants session

1) Les différents composants

Afin de gérer les composants entités, nous mettons en place des composants session sans état notés **@Stateless**.
La gestion de l'application consiste à implémenter les opérations d'ajout, modification et suppression en se servant des méthodes set et get des classes entités.
Pour chaque entités, nous avons défini une interface GestionX (X représentant Musee, Artiste ou Oeuvre) permettant de définir les opérations à implémenter. Puis pour chaque entité, nous définissons une interface local GestionXLocal notée par **@Local** et une interface GestionXRemote annotée par **@Remote** pour accéder à l'objet à distance qui étendent l'interface GestionX. Les interfaces locale et distante possèdent les mêmes opérations d'où la création de l'interface GestionX.
Enfin la classe GestionXBean notée **@Stateless** permet d'implémenter les opérations de l'interface GestionX.
Pour chaque classes GestionXBean, on utilise un objet EntityManager qui permet de faire le lien entre les objets entités et les tables en mettant à jour la base.
Pour les classes GestionXBean, voici les opérations implémentées :

a) composant GestionOeuvreBean

Ce composant permet de créer, supprimer une Oeuvre. Il offre aussi la possibilité de modifier le titre et l'année d'une oeuvre, récupérer une oeuvre particulier par son url, la liste des oeuvres de l'application et la liste des oeuvres pour un nom d'artiste donné.

b) composant GestionArtisteBean

Ce composant permet de créer un artiste ou récupérer la liste des artistes de l'application ou un artiste donné par son nom.

c) composant GestionMuseeBean

Sur ce composants nous pouvons créer un musée, récupérer un musée par son thème ou la liste des musées de l'application, ajouter ou supprimer une oeuvre à un musée

2) La gestion des exceptions

Afin de permettre la cohérence de l'application, nous avons mis en place des exceptions :
Lorsque nous ajoutons un artiste, une exception de type ArtisteDejaExistantException si l'artiste est déjà présent dans la base
Nous faisons de même pour les Oeuvres avec OeuvreDejaExistanteException et pour les Musées avec MuseeDejaExistantException.
Lorsqu'on récupère un Artiste à partir de son nom ou lors de l'affectation d'un artiste à une oeuvre ou encore lorsqu'on souhaite afficher toutes les oeuvres d'un artiste particulier et si il n'est pas inscrit dans la base, on lance une exception de type ArtisteIntrouvableException.
Lorsqu'on désire supprimer, éditer ou récupérer une oeuvre particulière ou pour l'affecter à un musée et si elle n'existe pas, on déclenche un exception de type OeuvreIntrouvableException.
Lorsqu'on récupère un musée par son nom ou lorsqu'on affecte ou supprime une oeuvre à un musée, on déclenche l'exception MuseeIntrouvableException.

3) Les composants web-jsp

Les composants sont déployés sur le même serveur que celui contenant la base de données. Ils accèdent aux composants session en local et récupèrent donc les interfaces locales pour lancer les opérations. Les fichiers correspondants aux composants web JSP sont inscrits dans le fichier **web.xml**. Grâce à un mécanisme de mapping, on peut faire le lien entre les pages WEB et les composants JSP permettant de réaliser les traitements

Pour les traitements nous avons un formulaire associé. Ce tableaux résume les fichiers utilisés :

Formulaire	Opération	Fonction de la page
ajout_artiste.html	ajouter_artiste.jsp	ajoute un artiste à la base
ajout_musee.html	ajouter_musee.jsp	ajoute un musée à la base
ajout_oeuvre.jsp	ajouter_oeuvre.jsp	ajoute une oeuvre à la base
modif_oeuvre.jsp	modifier_oeuvre.jsp	Modifie les champs titre et année d'une oeuvre donnée
suppr_oeuvre.jsp	supprimer_oeuvre.jsp	Supprime une oeuvre de la base
affecte_musee.jsp	affecter_musee.jsp	ajoute ou supprime l'affectation d'une oeuvre à un musée
liste_OeuvreArtiste.jsp	lister_OeuvreArtiste.jsp	Affiche la liste des oeuvres d'un artiste
	lister_oeuvres.jsp	Affiche la liste des oeuvres de la base
	lister_artiste.jsp	Affiche la liste des artistes contenus dans la base
consult_musee.jsp	consulter_musee.jsp	Affiche les oeuvres d'un musée donné

Le pages sont organisées suivant une frame html. Sur le côté gauche nous retrouvons le menu de l'application (menu.html) et sur le côté droit, les formulaires de saisie et les résultats des traitements.

Afin de contrôler les champs de saisie sont remplis, nous avons mis en place des fonctions javascript de vérification.
Pour le thème graphique de l'application, nous avons créer une feuille de style CSS. Cela permettra au futur programmateur de changer comme il le souhaite l'aspect de l'application sans changer le code des fichiers JSP.

4) La classe MainOeuvre

Cette classe accède à distance aux composants sessions. Elle récupère les interfaces distantes notées @Remote des objets sessions pour effectuer les traitement. Nous traiterons ici l'ajout d'ouevres, d'artistes et de Musées.

IV) GUIDE D'UTILISATION

L'application est déjà déployée et la base créée et peuplée.
Nous vous avons envoyé une archive **jar** contenant l'application. Pour décompresser l'archive il faut utiliser la commande

```
jar -xvf ocremetzMusee.jar
```

A) Compilation et déploiement de l'application

Nous nous servons ici d'un fichier **build.xml** et de l'outil Ant. Cela permet d'automatiser les tâches de compilation et de de déploiement de l'application. Pour utiliser les commandes suivantes, il faut se placer dans le répertoire contenant le fichier build.xml.
Pour compiler l'application : **ant**
Pour deployer l'application sur le serveur : **ant deploy**
Pour retirer l'application du serveur on lance la commande : **ant clean**
Les tables de l'application devront être supprimée de postgres avec les commandes suivantes :

```
drop table ocremetz_musee cascade;
drop table ocremetz_artiste cascade;
drop table ocremetz_oeuvre cascade;
drop table ocremetz_musee_ocremetz_oeuvre cascade;
```

B) Lancement de l'application client

Pour compiler l'application distante, la commande à lancer est : **ant compile-client**
Il faut déployer l'application avec **ant-deploy** ou **ant deploy-earWithoutWar**
La commande **ant run-client** permet de compiler et de lancer l'application MainOeuvre.java
L'application sert à charger la base de données avec des exemples.

C) ***Application Web-JSP***

Pour accéder à l'application, il suffit alors de se connecter à l'adresse suivante:

http://weppes.studserv.deule.net:8080/ocremetzMusee/

Il n'y a aucune recommandation particulière. Les champs de saisie doivent être obligatoirement remplis sinon un message d'erreur s'affichera.

CONCLUSION

La réalisation de ce projet nécessite de bonnes bases en conception de systèmes d'information, base de données ainsi qu'en programmation java J2EE. Ce projet nous a donc permis de prendre conscience de l'importance de l'organisation ainsi que de la répartition des tâches dans le cas de la réalisation d'un projet informatique.