Architecture des applications Web - introduction

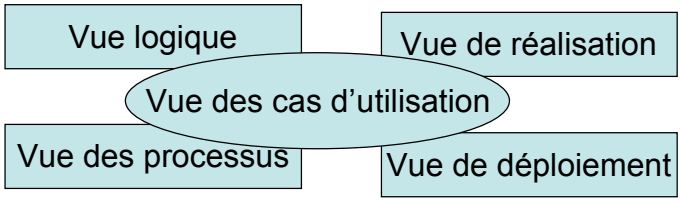
Architecture des applications Web
Introduction

Plan

1. Architecture en couches
2. Architecture MVC
3. Client riche avec AJAX
4. Persistance : HIBERNATE
5. Transaction avec JDBC : le pourquoi du comment

Architecture logicielle

Elle décrit la structure générale du logiciel en constituants de haut niveau, ainsi que l'interaction entre ces éléments



Le modèle des 4+1 vue de Philippe Kruchten

Architecture en couche

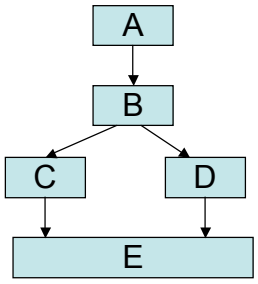
Le logiciel peut être organisé en couche : une couche regroupe un ensemble de classes et propose un ensemble cohérent de services à travers une interface.

Les couches sont ordonnées : les couches de plus haut niveau peuvent accéder à des couches de plus bas niveau mais pas l'inverse.

Dans une architecture *étanche* , un couche de niveau *n* n'accède que à la couche de niveau *n-1*

Architecture en couche

Exemple



A : couche de présentation
B : couche « application », réalisant la médiation entre l'interface graphique et les couches C et D.
C et D : 2 couches « domaine », (« métier ») contenant les principales classes du domaine d'applications, conçues pour être indépendante de l'interface graphique et de l'infrastructure
E : couche infrastructure contenant les services techniques bas niveau

Avantage d'une architecture en couche

1. Maintenance
   - Le système est plus facilement modifiable. Une modification de couche n'affecte pas les couches inférieures. Une modification de couche qui ne change pas d'interface n'affecte pas les couches supérieures
2. Réutilisation
   - Des éléments de chaque couche peuvent être réutilisés. Par exemple, les couches « métier » peuvent être communes à plusieurs applications
3. Portabilité
   - On confine les éléments qui dépendent du système aux basses couches. Suivant le système d'exploitation, on change les couches basses, mais les couches hautes restent identiques

Inconvénients d'une architecture en couche

Si on a un grand nombre de couches, et si en plus ces couches sont étanches, l'appel de fonction de bas niveau est moins efficace, puisqu'il faut traverser toutes les couches pour parvenir à ces fonctions.

Il faut donc trouver un compromis entre une bonne encapsulation et une bonne efficacité.

Architecture Modèle-Vue-Contrôleur

Fréquemment utilisée pour les IHM. Introduite par SmallTalk en 1980, elle très largement répandue aujourd'hui : Struts, Spring, Zend, etc.

Le logiciel est découpé en 3 parties :

- Le modèle : les données métier.
- La vue : traitement des sorties.
- Le contrôleur : traitement des interactions (entrées).

Entrées --> Traitement --> Sorties

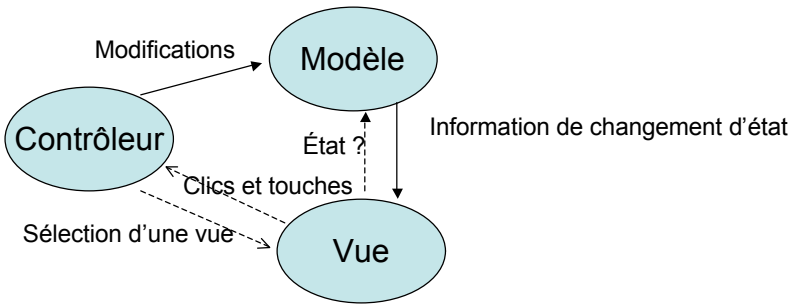
Contrôleur --> Modèle --> Vue

Architecture MVC

1. Modèle
   - comporte les classes principales correspondant aux différentes fonctionnalités de l'application : données et traitements. Indépendamment des parties « vue » et « contrôleur », elle effectue des actions en réponse aux demandes de l'utilisateur (par l'intermédiaire de la partie contrôleur), et informe la partie vue des changements d'états du modèle pour sa mise à jour.
2. Vue
   - Comporte les classes relatives à l'interface graphique (ce que l'utilisateur voit). Cette partie est passive : elle est informée des changements d'états du modèle et se met à jour lors de ces changements.
3. Contrôleur
   - Récupère les actions de l'utilisateur (clics de souris, touches clavier) et associe ces évènements à des actions qui modifient le modèle.

Interactions MVC

1. Le contrôleur, en fonction des évènements qu'il reçoit de l'utilisateur, effectue des modifications du modèle.
2. Ces modifications du modèle sont ensuite transmises à l'interface graphique.



Avantages d'une architecture MVC

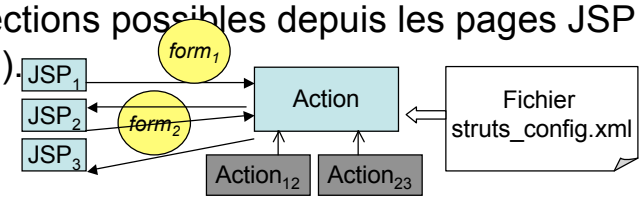
1. Vues multiples
   - Gestion et affichage de plusieurs vues du même modèle possible et facile
2. Portabilité
   - Le portage de l'interface sur d'autres plateformes est possible, sans avoir à modifier le noyau de l'application
3. Evolution
   - L'amélioration de l'interface graphique (ajout de lignes dans un menu, ou de boutons) est plus facile. De plus, ces modifications peuvent se faire lors de l'exécution du logiciel. --> personnalisation des pages Web.

Exemple avec Struts

1. Le contrôleur : *`Action`*
   - Toutes les classes du contrôleur étendent *`Action`*, et implémentent cette méthode:

```
public ActionForward perform(ActionMapping mapping,
    ActionForm form, HttpServletRequest request,
    HttpServletResponse response)
```

   - Le contrôleur est piloté via un fichier de configuration qui déclare toutes les indirections possibles depuis les pages JSP (VUE).



Struts : configuration du contrôleur

1. Le fichier struts_config.xml

```
  <!-- Check whether a session already exists for a incoming request -->
<action path="/tiles/checkSession »
type="com.eloquant.oss.workflow.actions.CheckSessionAction »
      scope="request" validate="false">
<forward   name="noSession"   path="site.index.page" />
</action>

  <!-- Create a subscriber -->
<action path="/createSubscriber »
type="com.eloquant.oss.workflow.actions.CreateSubscriberAction"
name="addSubscriberForm" scope="request" validate="true"
      input="site.add.Subscriber.page">

<forward name="success" path="site.add.SSC.page"/>
<forward name="cancel" path="site.display.SO.page"/></action>
```

Next page (any case)

URL de la page

Formulaire à valider

Class contrôleur à appeler

From Page

Next page if form is validated

Next page if form is canceled

Exemple d'action 1/2

```
public ActionForward perform(ActionMapping mapping, ActionForm form, HttpServletRequest
      request, HttpServletResponse response)throws IOException, ServletException{
// Few object declaration
      GlobalInfo globalinfo = null;
      COD cod = null;
// Prepare an ActionError object to store errors if any
      ActionErrors errors = new ActionErrors();
// Get the session object associated with this request
      HttpSession session = request.getSession();
//-----------------------------------------------------------------
// Check That the session still exists
//-----------------------------------------------------------------
if (session != null){// Get the globalnfo object attached to this session
      globalinfo = (GlobalInfo)session.getAttribute ("globalinfo");
      // Get the cod object attached to this session
      cod = (COD)session.getAttribute ("cod");}
else{  errors.add(ActionErrors.GLOBAL_ERROR,
          new ActionError("error.session.has.expired"));
      saveErrors(request, errors);
      return (mapping.findForward("sessionExpired"));}
//-----------------------------------------------------------------
// Check if the 'Cancel' button was pressed. If so get back to the previous page
//-----------------------------------------------------------------
if (isCancelled (request)){
      return (mapping.findForward("cancel"));}
```

Récupère des BEANS (DTO) de session

Utilisation d'un forward configuré de façon globale à toutes les pages

Utilisation d'un forward spécifique en cas de cancel pour cette action

Exemple d'action 2/2

```
// Process the action, provided that necessary session beans are not null
//-------------------------------------------------------------
if (globalinfo!=null && cod!=null){
// Get the parameters
String destObject = request.getParameter("destObject");
String childName = ((AddSubscriberForm) form).getName();
boolean addSubMonitorUser = ((AddSubscriberForm)form).getAddSubscriberMonitorUser();

//do the job
…      //There is some calls to MODEL classes, to record changes,
       //(create a new Subscriber Object in DB)
//if you got somme errors then
errors.add(ActionErrors.GLOBAL_ERROR,new ActionError("error.object.creation"));

//-------------------------------------------------------------
// Report any errors we have discovered back to the original form
//-------------------------------------------------------------
if (!errors.empty()){
       saveErrors(request, errors);
       return (new ActionForward(mapping.getInput()));}
else {
       return (mapping.findForward("success")); }
```

Récupère le bean form lié à cette action, et les paramètres qu'il porte

Renvoi sur la page *from* (formulaire initial de saisie)

Renvoi sur la page *next* (cas de succès)

Concevoir un CLIENT RICHE avec AJAX

AJAX : Asynchronous JavaScript And XML

Enrichir HTML avec :

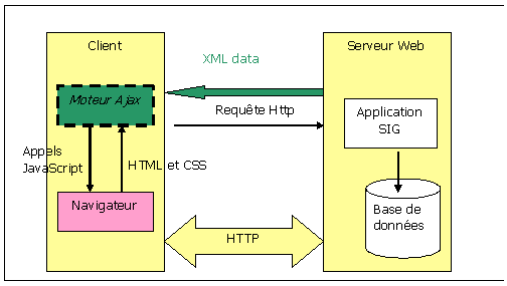
- des feuilles de style CSS pour présenter l'information
- JavaScript et le modèle objet DOM pour effectuer des calculs
- XML et XSLT pour la structuration des données
- l'objet **XMLHttpRequest** pour échanger les données de manière asynchrone avec le serveur Web. Supporté nativement par les navigateurs connus : Mozilla, InternetExplorer, Safari, Opera,…, recommandé par W3C depuis Février 2007

XMLHttpRequest permet aux scripts de réaliser l'ensemble des opérations d'un client http :

- créer des requêtes HTTP (type configurable : GET, POST),
- envoyer des données avec
- traiter leur résultat de façon asynchrone.

AJAX

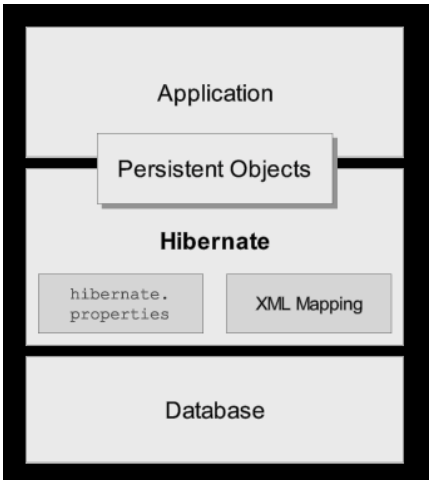
1. Avantage
   - allège le volume de données qui transitent entre le client et le serveur
   - une partie des traitements sur les données se font coté client.
2. Inconvénient
   - le temps de chargement de la première page : bibliothèque Ajax (500 Ko par exemple).



2010 - C. Plumejeaud LP essig - 17/30

HIBERNATE : faciliter la persistance

Hibernate fournit de nombreuses interfaces d'extensions optionnelles permettant de personnaliser la couche persistance http://www.hibernate.org/

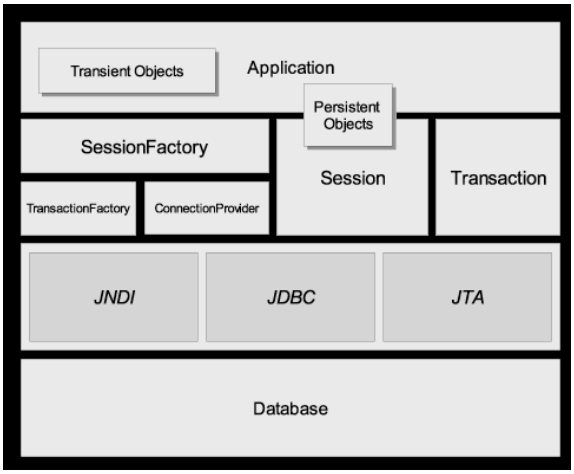


Hibernate est un **outil de mapping objet/relationnel pour le monde Java**. Le terme mapping objet/relationnel (ORM) décrit la technique consistant à faire le lien entre la représentation objet des données et sa représentation relationnelle basée sur un schéma SQL.
Non seulement, Hibernate s'occupe du transfert des classes Java dans les tables de la base de données (et des types de données Java dans les types de données SQL), mais il permet de requêter les données et propose des moyens de les récupérer. Il peut donc réduire de manière significative le temps de développement qui aurait été autrement perdu dans une manipulation manuelle des données via SQL et JDBC.

2010 - C. Plumejeaud LP essig - 18/30

Isoler la couche Métier des données avec Hibernate

Exemple d'architecture



Objets JAVA

O1 : Personne
- Name : Smith
- Address : Lyon
- Phone : 0476083321

Tables relationnelles

Id	Name	adress	phone
01	Smith	Lyon	0476083321

2010 - C. Plumejeaud LP essig - 19/30

Isoler la couche Métier des données avec Hibernate

SessionFactory (org.hibernate.SessionFactory)
Un cache threadsafe (immuable) des mappings vers une (et une seule) base de données. Une factory (fabrique) de Session et un client de ConnectionProvider.

Session (org.hibernate.Session)
Un objet mono-threadé, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance. Encapsule une connexion JDBC. Factory (fabrique) des objets Transaction. Contient un cache (de premier niveau) des objets persistants, ce cache est obligatoire. Il est utilisé lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant.

Objets et Collections persistants
Objets mono-threadés à vie courte contenant l'état de persistance et la fonction métier. Ceux-ci sont en général les objets de type **JavaBean** (ou POJOs) ; la seule particularité est qu'ils sont **associés avec une (et une seule) Session**. Dès que la Session est fermée, ils seront détachés et libres d'être utilisés par n'importe laquelle des couches de l'application (ie. de et vers la présentation en tant que Data Transfer Objects - DTO : objet de transfert de données).

Objets et collections transients
Instances de classes persistantes qui ne sont actuellement pas associées à une Session. Elles ont pu être instanciées par l'application et ne pas avoir (encore) été persistées ou elle ont pu être instanciées par une Session fermée.

Transaction (org.hibernate.Transaction)
(Optionnel) Un objet mono-threadé à vie courte utilisé par l'application pour définir une **unité de travail atomique**. Abstrait l'application des transactions sous-jacentes qu'elles soient JDBC, JTA ou CORBA. Une Session peut fournir plusieurs Transactions dans certains cas.

ConnectionProvider (org.hibernate.connection.ConnectionProvider)
(Optionnel) Une fabrique de **(pool de) connexions JDBC**. Abstrait l'application de la Datasource ou du DriverManager sous-jacent.

TransactionFactory (org.hibernate.TransactionFactory)
(Optionnel) Une fabrique d'instances de Transaction.

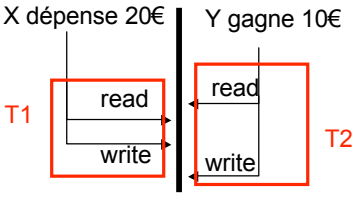
Transaction

1. Mais Pourquoi ?
2. Et comment avec JDBC ?

Les accès concurrents sur une BD

Le problème

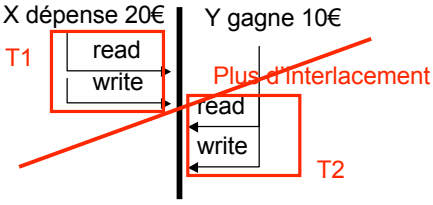
MonCompte = 10€



Read
–Pour X, MonCompte = 10€
–Pour Y, MonCompte = 10€
Write
–Pour X, MonCompte = -10€
–Pour Y, MonCompte = 20€

L'idéal

MonCompte = 10€



Read
–Pour X, MonCompte = 10€
–Pour Y, MonCompte = -10€
Write
–Pour X, MonCompte = -10€
–Pour Y, MonCompte = 0€

La transaction

Elle définit une suite d'opérations (read/write) sur la base de données par un acteur.

Elle commence par START et se termine par COMMIT (valide le bloc d'instructions)

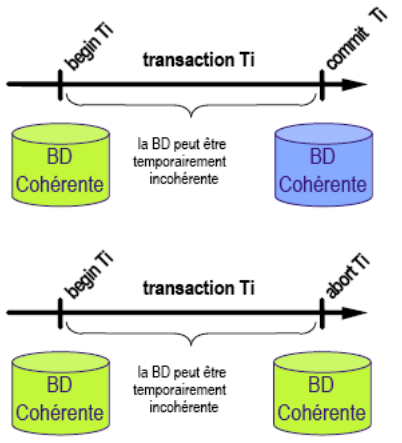
ou ROLLBACK (annule les instructions)

Elle est doit être ACID

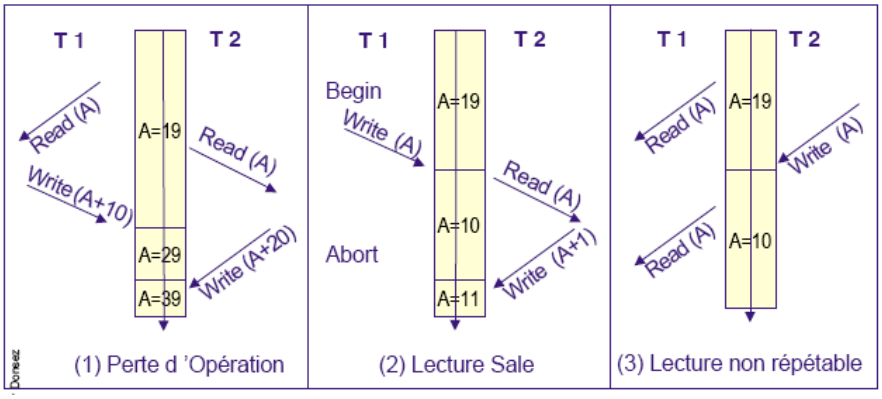
ACID

Propriétés ACID

- **A**tomicité
  - tout ou rien
- **C**onsistance
  - cohérence sémantique
- **I**solation
  - pas de propagation de résultats non validés
- **D**urabilité
  - persistance des effets validés



Problèmes de concurrence



2010 - C. Plumejeaud LP essig - 25/30

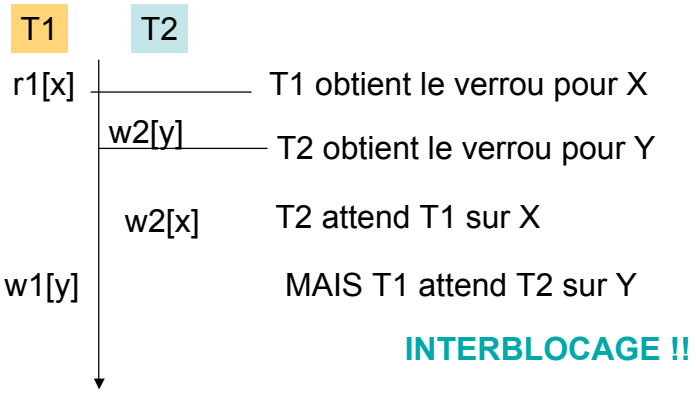
Solutions

1. Contrôler la concurrence par des algorithme qui réordonne les opérations de chaque transaction de façon à garantir l'ACIDité : on vise la SÉRIALISABILITÉ
   - Retarder certaines opérations
   - Respecter l'ordre des opérations de chaque transaction
2. Usage de VERROUS sur les variables, mais avec risque d'interblocage

2010 - C. Plumejeaud LP essig - 26/30

Verrou et interblocage

1. T1 : r1[x] w1[y] c1
2. T2 : w2[y] w2[x] c2



Plusieurs degrés d'isolation

1. La cohérence forte (sérialisable) n'est pas toujours souhaitable car le blocage des transaction limite leur débit (rapidité).
2. Contrôle SQL du niveau de

Niveaux d'Isolation	Lecture Sale	Lecture Non Répétable	Fantômes
READ_UNCOMMITED	oui	oui	oui
READ_COMMITED	non	oui	oui
REPEATABLE_READ	non	non	oui
SERIALIZABLE	non	non	non

La transaction en JDBC

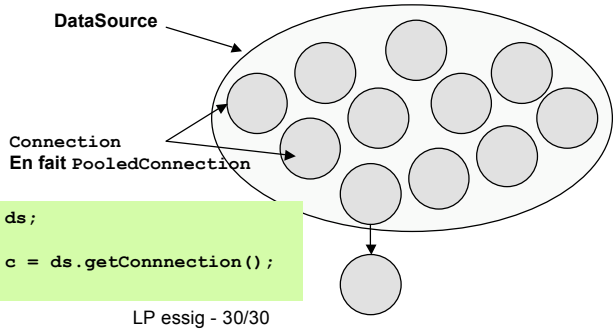
```
//commencer la transaction
connexion.setAutoCommit (false)
      // do the job
//si OK
      connexion.commit()
//sinon
      connexion.rollback()
```

Attention : les connections utilisant les transactions ne peuvent pas être partagées par plusieurs threads…

Mais créer une connection pour chaque requête (thread) est coûteux…

Usage d'un groupe de connexions

Les *pool de connexions* (***DataSource***) sont le moyen de créer un certain nombre de connections à l'avance, qui pourront être utilisées par chaque transaction, avant d'être relâchées.




```
DataSource ds;
...
Connection c = ds.getConnnection();
...
```