

Conception d'applications internet

JDBC & Transactions

Olivier Cailloux

LAMSADE, Université Paris-Dauphine

21 janvier 2016

Était à faire

- Groupe sur [MyCourse](#) & serveur git : cf. dernière annonce

Avant le 21 janvier

- +^a Chaque binôme : *au moins* un servlet utile à votre projet
- + Chaque binôme : *au moins* un EJB utile à votre projet
- + Au moins un push par personne dans le projet

a. Le + indique que cet aspect intervient dans la note

Chaque commit doit indiquer clairement le ou les auteurs

- Utilisez `user.name`
- Indiquez dans le commentaire du commit le binôme éventuel

Évaluation

- Échelle : --, -, N, +, ++
- Une feuille d'évaluation par projet
- Note projet = note de chacun des membres *par défaut*

Java EE et la persistance : introduction

- BD : modèle relationnel typiquement
- JDBC : accès via Java, modèle relationnel
- JPA : accès via Java, modèle objet
- JPA implémente un *ORM* : Object-Relational Mapping

JPA

- Inconvénient : plus complexe (?) que JDBC
- Avec Java EE, JPA généralement utilisé
- JPA s'appuie sur JDBC

Dans ce cours : JDBC puis JPA

Modèle relationnel

- JDBC permet d'accéder aux BD
- Suit le modèle relationnel
- Élément central du modèle : la relation (une table, par exemple)
- Relations produites par des JOIN, SELECT, etc.

Forces du modèle relationnel

Modèle relationnel

- JDBC permet d'accéder aux BD
- Suit le modèle relationnel
- Élément central du modèle : la relation (une table, par exemple)
- Relations produites par des JOIN, SELECT, etc.

Forces du modèle relationnel

- Garanties théoriques (algèbre relationnelle)
- Efficace
- Standard de fait depuis ~1990
- Robuste : [Codd 1970](#), ANSI (puis ISO) SQL [1987](#) ; ... ; [2011](#)

(Bémol : nombreuses variations propriétaires)

Vue d'ensemble de l'API JDBC

- Fournisseur de SGBD implémente un pilote JDBC
- Obtenir une [Connection](#) (communique avec le pilote)
- `Connection` permet les *transactions*
- Transaction : ensemble atomique de « statements » SQL
- Gérer début et fin de transaction via la connexion
- Exécuter des statements SQL via cette connexion
- Par défaut, mode auto-commit : une transaction par stmt
- Via `Connection` : exécution requêtes, navigation de [ResultSets](#), ...
- Puis *fermer* la connexion

Cf. [tutoriel](#)

Instanciation

Approche 1 (Java SE, typiquement)

- Une classe fournisseur implémente `Driver`
- Développeur appelle `DriverManager`
- `DriverManager` trouve le pilote et l'instancie
- Exemple : `DriverManager.getConnection(url)`

Approche 2 (Java EE, typiquement)

- Une classe fournisseur implémente `DataSource`
- Source accessible via JNDI à un endroit convenu
- Développeur instancie `DataSource` par lookup JNDI puis appelle `source.getConnection(...)`

Injection de la DataSource

- Injection de ressources via `@Resource`
- Le conteneur va chercher la ressource via JNDI
- Nom JNDI par défaut selon type de la ressource
- Pour nous : `@Resource DataSource myDataSource;`

Statement et ResultSet

- Création d'un **Statement** (via `Connection`)
- Via `Statement` : exécution d'une commande SQL (`SELECT`, `UPDATE`...)
- Via `Statement` : paramétrisation possible (nb résultats max...)
- Obtention (si `SELECT`) d'un **ResultSet**
- `ResultSet` associé à une ligne courante ; initialement : avant la première
- Naviguer via `next()` aux lignes suivantes
- Invoquer `getInt(columnLabel)`, `getString(columnLabel)`...

PreparedStatement

- **PreparedStatement** : précompilé + paramétrisation facile
- La commande SQL contient des ?
- Invoquer `setInt`, `setString`... pour les paramètres

Exemple PreparedStatement

```
String s = "update USER set NAME = ? where ID = ?";
PreparedStatement stmt = con.prepareStatement(s);
stmt.setString(1, "NewName");
stmt.setInt(2, 1234);
boolean isResultSet = stmt.execute();
assert(!isResultSet);
assert(stmt.getUpdateCount() == 1);
```

Utiliser `PreparedStatement` pour éviter les attaques de type injection SQL !

Transactions

Par défaut, mode *auto commit* : une transaction par commande

Gestion de transactions explicite

- Invoquer `setAutoCommit` sur `Connection`
- Exécuter les commandes normalement
- Puis invoquer `commit` sur `Connection`
- Ou : `rollback`
- Voir aussi : `getTransactionIsolation`, `setTransactionIsolation`

Glassfish

- BD intégrée à Glassfish : [Derby](#)
- Il faut démarrer la BD : `asadmin start-database`
- [Manuel](#) SQL pour Derby

Eclipse

- Pour accéder à une BD, il faut un pilote
- Preferences / Data Management / Connectivity / Driver Definitions
- Créer une instance de pilote pour Derby
- Renseigner le pilote fourni avec glassfish (`glassfish4/javadb/lib/derbyclient.jar`)
- Connexions depuis vue *Data Source Explorer*
- Créer une connexion à BD Derby (via pilote créé)
- Le Schéma à utiliser est « APP »
- Envoi de commandes : SQL Scrapbook (avec complétion)
- Édition de données : depuis Data Source Explorer

Pattern DAO

Extraction des aspects propres à la persistance

- DAO ?

Pattern DAO

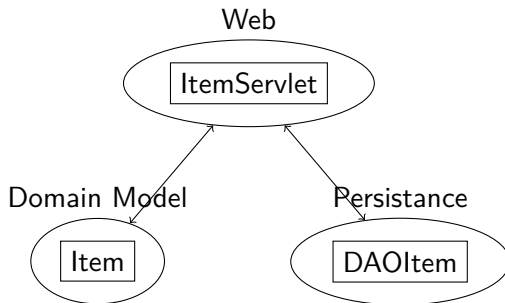
Extraction des aspects propres à la persistance

- DAO ? Data Access Object

Pattern DAO

Extraction des aspects propres à la persistance

- DAO ? Data Access Object
- « Modèle » découpé en aspects Persistance et Domain Model
- Domain Model : opérations logiques, connaissance métier
- Persistance : seule autorisée à communiquer avec la BD



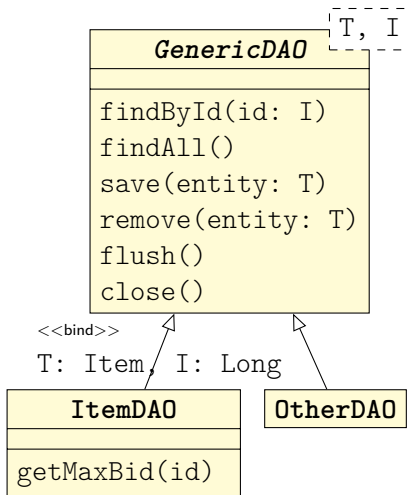
DAO : mise en œuvre

Classe parent

- abstraite
- générique :
 - T = type entité
 - I = type ID
- contient les méthodes *CRUD* : Create / Update / Read / Delete

Sous-classes

- contiennent les méthodes spécialisées



Inspiré par : Java persistence with hibernate

Autres remarques architecturales et références

- Alternative au DAO : Active Record
- Avec JPA, DAO peut être superflu

Références

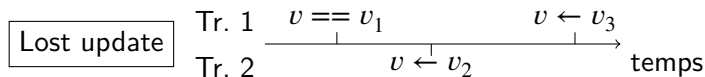
- Java Persistence with Hibernate
- Patterns of Enterprise Application Architecture

Nécessité des transactions atomiques

- Accès concurrents à DB : risques
- Si une transaction par statement ?

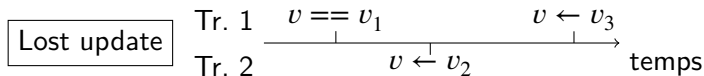
Nécessité des transactions atomiques

- Accès concurrents à DB : risques
- Si une transaction par statement ?



Nécessité des transactions atomiques

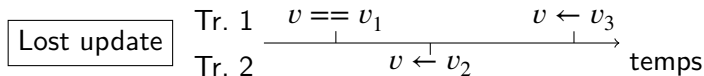
- Accès concurrents à DB : risques
- Si une transaction par statement ?



- Transaction atomique non triviale (couvrant un ensemble de statements) permet de lire-puis-écrire sans interruption
- Implémentation naïve : DB verrouillée pour un utilisateur pendant le temps de la transaction
- Problème ?

Nécessité des transactions atomiques

- Accès concurrents à DB : risques
- Si une transaction par statement ?

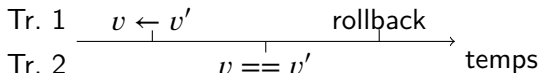


- Transaction atomique non triviale (couvrant un ensemble de statements) permet de lire-puis-écrire sans interruption
- Implémentation naïve : DB verrouillée pour un utilisateur pendant le temps de la transaction
- Problème ? Souvent trop peu efficace
- Protection : transaction terminée par *commit* ou *rollback*

Niveaux d'isolation

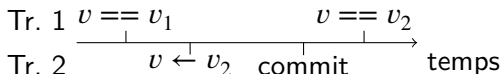
Read uncommitted (risques ↓)

Dirty read

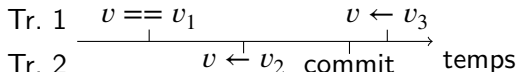


Read committed (protection ↑, risques ↓)

Non-repeatable rd

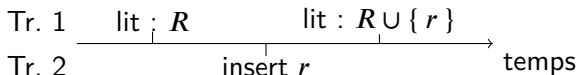


2nd lost update



Repeatable read (protection ↑, risques ↓)

Phantom



Serializable (protection ↑)

Niveaux d'isolation

- Quatre niveaux d'isolation standards (ANSI ; JDBC ; JTA)
(critiqués)
- Définis comme protection contre catégories de risques
- Risque défini comme : phénomène problématique
- SGBD configuré pour un niveau d'isolation donné
- Typiquement : Read committed
- Possible de se protéger contre certains risques au cas par cas

Protection contre 2nd lost update

- Optimiste : lire version lors lecture, check version lors écriture
- Pessimiste : verrouiller lors lecture

Exercices

- Créer (sur papier) une table pour un type de votre projet.
 - La créer dans votre BD via le SQL Scrapbook.
 - Tester des requêtes simple de création, sélection, effacement dans le SQL Scrapbook.
 - Permettre CR.D : Create, Retrieve, Delete *aussi simple que possible*, via un ou plusieurs servlets. (N'utilisez pas de paramètres complexes, ce n'est pas le but de cet exercice.)
 - Programmer une méthode qui transforme un attribut d'un objet. Par exemple, elle met le nom en majuscule s'il ne l'était pas (obligation d'utiliser Java, pas SQL : supposez que la transformation est trop complexe pour être exprimée en SQL).
- + Permettre l'application de cette méthode via un servlet. Votre servlet ne doit pas nécessairement accepter de paramètres. Attention à l'atomicité de la transaction !

À vous de jouer

Exercice à effectuer *avant* le 25 janvier, par chaque membre de chaque équipe.

- +¹ Un servlet qui applique une transformation quelconque sur un attribut d'un ou plusieurs objets stocké dans votre BD. Cf. **Exercices**.

Chaque commit doit indiquer clairement le ou les auteurs

- Utilisez `user.name`, **gardez toujours le même nom**
- Indiquez dans le commentaire du commit le binôme éventuel : « **reviewer: Machin** » avec Machin le `user.name`

1. Le + indique que cet aspect intervient dans la note

Licence

Cette présentation, et le code LaTeX associé, sont sous [licence MIT](#). Vous êtes libres de réutiliser des éléments de cette présentation, sous réserve de citer l'auteur.

Le travail réutilisé est à attribuer à [Olivier Cailloux](#), Université Paris-Dauphine.

(Ceci ne couvre pas les images incluses dans ce document, puisque je n'en suis généralement pas l'auteur.)