Applications orientées données (NSY135)

14 – Applications concurrentes

Auteurs: Raphaël Fournier-S'niehotta et Philippe Rigaux
(philippe.rigaux@cnam.fr,fournier@cnam.fr)

Département d'informatique
Conservatoire National des Arts & Métiers, Paris, France

Plan du cours

3 Transactions applicatives
- Les stratégies possibles
- Versionnement avec Hibernate
- Par défaut, le dernier commit l'emporte!
- Granularité des sessions
- Méthode des objets détachés
- Méthode des sessions longues

4 Résumé : savoir et retenir

Notion de transaction

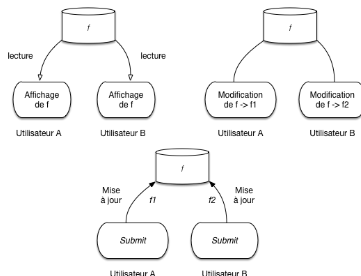
- Dans le cadre d'une application Web (ou d'une application interactive en général), la notion de "transaction" pour un utilisateur prend souvent la forme d'une séquence d'actions : affichage d'un formulaire, saisie, soumission, nouvelle saisie, confirmation, etc.
- On dépasse donc bien souvent le simple cadre d'un ensemble de mises à jour soumises à un moment donné par une action pour voir une "transaction" comme un séquence d'interactions (typiquement une séquence de requêtes HTTP dans le cadre d'une application Web).
- Un exemple typique est l'affichage des valeurs d'un objet, avec possibilité pour l'utilisateur d'effectuer des modifications en fonction de l'état courant de l'objet.
- Par exemple, on affiche un film et on laisse l'utilisateur éditer son résumé, ou les commentaires, etc.

Notion de transaction

- Les transactions vues jusqu'à présent (que nous appellerons **transactions en base**) s'exécutent dans le cadre d'une unique action, sans intervention de l'utilisateur.
- On va voir comment gérer des **transactions applicatives** couvrant plusieurs requêtes HTTP.
- Il est exclu de **verrouiller** des objets en attente d'une action utilisateur qui peut ne jamais venir.
- La stratégie à appliquer (pour laquelle Hibernate fournit un certain support) est une stratégie dite de **contrôle optimiste** : on vérifie, au moment des mises à jour, qu'il y n'a pas eu d'interaction concurrentielle potentiellement dangereuse.
- En BDD, l'algorithme de concurrence correspondant est dit "contrôle multiversions".
- Les transactions applicatives sont appelées également **transactions longues**, **transactions métier**, ou **transactions utilisateurs**.
- Elles supposent une mise en place au niveau du code de l'application, ce qui peut être assez lourd.

Stratégies possibles

- Avec l'exemple de notre édition du résumé d'un film, l'utilisateur consulte le résumé courant, le complète ou le modifie, et valide.
- Dans une situation concurrentielle, deux utilisateurs **A** et **B** éditent simultanément le film dans sa version f.
- Le premier valide une nouvelle version f_1 et le second une version f_2.
- Les trois phases de l'édition sont illustrées ci-dessous :



Stratégies possibles

- La question est **quelle mise à jour l'emporte ?**
- On peut envisager deux possibilités :
  - **Le dernier *commit* l'emporte**. Si **B** valide après **A**, la mise à jour de **A** (f_1) est perdue, et ce même si **A** a reçu un message de confirmation...
  - **Le premier *commit* l'emporte**. Si **B** valide après **A**, il reçoit une message indiquant que la version f a déjà été modifiée, et qu' il faut resoumettre sa modification sur la base de f_1.
- La seconde solution admet plusieurs variantes.
- On peut par exemple imaginer une tentative automatique de fusion des deux mises à jour.
- Toujours est-il qu'elle paraît préférable à la première solution qui efface sans prévenir une action validée.
- Dans la seconde stratégie, on reconnaît le principe de base du contrôle de concurrence multiversions.
- Pour l'implanter, on peut s'appuyer sur une gestion automatique de versions incrémentées fournie par Hibernate.

Versionnement avec Hibernate

- Hibernate (JPA) permet la maintenance automatique d'un numéro de version associé à chaque ligne.
- Il faut ajouter une colonne à la table à laquelle s'applique le contrôle multiversions, et la déclarer dans l'entité de la manière suivante :

```
1     @Version
2   public long version;
```

- Ici, la colonne s'appelle **version** (même nom que la propriété) mais on est libre de lui donner n'importe quel nom.
- On a choisi d'utiliser un compteur séquentiel, mais Hibernate propose également la gestion d'estampilles temporelles (marquage par le moment de la mise à jour).

Versionnement avec Hibernate

- L'interprétation de cette annotation est la suivante : Hibernate détecte toute mise à jour d'un objet le rendant **"dirty"**.
- Cela inclut la modification d'une propriété ou d'une collection.
- La requête **update** engendrée prend alors la forme suivante, illustrée ici sur notre classe **Film**, et en supposant que la version courante (celle de l'objet dans le cache) est 3.

```
1    update Film set [...], version=4 where id=:idFilm and version=3
```

Versionnement avec Hibernate

- Hibernate sait (par l'objet présent dans le cache) que la version courante est 3.
- La requête SQL se base sur l'hypothèse que le cache est bien synchrone avec la base de données, et ajoute donc une clause **version=3** pour avoir la garantie que la ligne modifiée est bien celle lue initialement pour instancier l'objet du cache.
- Si c'est le cas, cette ligne est trouvée, la mise à jour s'effectue et le numéro de version est incrémenté.
- **Si ce n'est pas le cas, c'est qu'une autre transaction a effectué une mise à jour concurrente et incrémenté le numéro de version de son côté**.
- L'objet dans le cache n'est pas synchrone avec la base, et la mise à jour doit être rejetée.
- Hibernate engendre alors une exception de type **StaleObjectStateException**.
- On se retrouve dans la stratégie 2 ci-dessus (le premier *commit* l'emporte), avec nécessité d'informer l'utilisateur qu'un mise à jour concurrente est passée avant la sienne.

Versionnement avec Hibernate

- Le mécanisme n'est sûr que si **toutes** les mises à jour de la table **Film** passent par le même code Hibernate...
- C'est la limite (forte) d'une gestion de la concurrence au niveau de l'application.
- Doté de ce mécanisme natif Hibernate, voyons comment l'utiliser dans le cadre de transactions applicatives.
- Auparavant, vérifions que, si nous faisons rien de plus, c'est la première solution, **le dernier** *commit* **l'emporte**, qui s'applique.

Mise à jour d'un film

- Par défaut, le dernier commit l'emporte !
- Implantons la fonction simplement la fonction de mise à jour d'un film.
- La première action affiche le formulaire :

```
1    update Film set [...], version=4 where id=:idFilm and version=3
```

- Il faut implanter la méthode **chercheFilmParTitre()** avec la requête HQL appropriée.

Page JSP (restreinte au body)

```
<body>

   <h2>Edition d'un film: le formulaire</h2>

<p>Vous éditez le film ${film.titre}, dont la version courante est ${film.version}</p>

<form action="${pageContext.request.contextPath}/transactions" method="get">
<input type="hidden" name="idFilm" value="${film.id}"/>
<input type="hidden" name="action" value="modifier"/>
<textarea name="resume" cols="80" rows="7">${film.resume }</textarea> $
<input type="submit">
</form>

</body>
```

- Nous sommes dans le contexte d'un contrôleur **Transactions**.
- Vous voyez que sur validation de ce formulaire, on appelle une autre action du même contrôleur, **modifier**.

Action modifier

```
if (action.equals("modifier")) {
  // Initialisation du modèle
  Transactions trans = new Transactions();

  // Recherche du film
  String idFilm = request.getParameter("idFilm");
  Film film = trans.chercheFilmParId(Integer.parseInt(idFilm));

  // On conserve la version avant modif, pour l'afficher
  request.setAttribute("versionAvantModif", film.getVersion());

  // Modification et validation
  film.setResume (request.getParameter("resume"));
  trans.updateFilm(film);

  // Affichage
  request.setAttribute("film", film);
  maVue = "/vues/transactions/modifier.jsp";
}
```

Code de la JSP

- Il faut implanter les méthodes **chercheFilmParId()** et **updateFilm()**, cette dernière selon le modèle de transaction présenté précédemment.

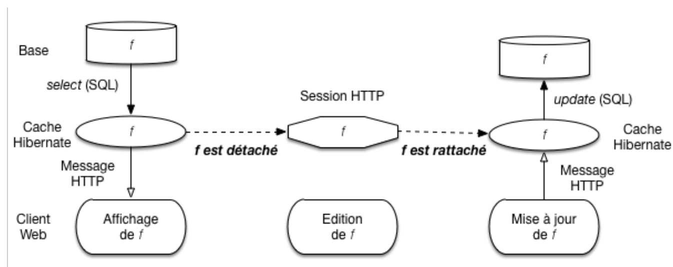
```
1    <h2>Mise à jour du film</h2>
2
3    <p>Vous avez demandé la mise à jour du film ${film.titre}, dont
4    la version courante est ${versionAvantModif }
5       </p>
6
7    <p>
8       Le résumé que vous avez saisi est: ${film.resume }
9    <p>
10   <p>
11   La version <i>après modification</i> est ${film.version}
12    </p>
13   <a href="${pageContext.request.contextPath}/transactions?action=editer">Retour au
14   formulaire d'édition.</a> $
```

- Il est très facile de vérifier, en ouvrant deux navigateurs sur la fonction d'édition du film, que le dernier utilisateur qui clique sur **Valider** l'emporte : la mise à jour du premier est perdue.

Granularité des sessions

- Dans l'implantation qui précède, le numéro de version ne nous sert à rien, car nous relisons à nouveau le film dans la méthode **modifier()**, et il n'y a donc aucune vérification que la version du film que nous **modifions** est celle que nous avons **éditée**.
- Jusqu'à présent, la portée d'une session coincide avec celle d'une requête HTTP.
- On ouvre la session quand une requête HTPP arrive, on la ferme avant de transmettre la vue en retour.
- C'est une méthode dite **session per request**, et la plus courante.
- On peut gérer une transaction applicative en gardant la même granularité (une session pour chaque requête) mais en exploitant la capacité d'un objet persistant à être **détaché** de la session courante.
- Dans ce cas on procède comme suit :
  - un objet persistant est placé dans le cache de la première session;
  - quand on ferme la première session, on **détache** l'objet et on le conserve bien au chaud (par exemple dans l'objet **HttpSession**) en attendant la requête HTTP suivante;
  - quand cette dernière arrive, on **ré-attache** l'objet persistant à la nouvelle session.

Méthode session-per-request-with-detached-objects



- Enfin il existe une méthode plus radicale, consistant à placer la session toute entière dans le cache applicatif survivant entre deux requêtes HTTP (typiquement l'objet **HttpSession**).
- Dans ce cas on ferme la connexion JDBC avec la méthode **disconnect()** et on en ouvre une nouvelle avec la méthode **reconnect()** quand la session est ré-activée.
- La méthode est dite **session-per-application-transaction** ou tout simplement session longue.

Méthode des objets détachés

- La méthode **session-per-request-with-detached-objects** est à placer dans la session HTTP l'objet à gérer par transaction applicative : code est très simple :

```
1       HttpSession httpSession = request.getSession();
2       httpSession.setAttribute("object", obj)
```

- La séquence est donc la suivante :
  1. on ouvre une session Hibernate, et on cherche les objets que l'on souhaite éditer,
  2. on place ces objets dans la session HTTP pour les préserver sur plusieurs requêtes HTTP,
  3. on ferme la session Hibernate et on affiche la vue.

Édition du film

```
// Version avec session applicative
Session hibernateSession = sessionFactory.openSession();

// Recherche du film
Film film = (Film) hibernateSession
        .createQuery("from Film where titre like :titre")
        .setString("titre", "Gravity").uniqueResult();
// On le place dans la session HTTP
HttpSession httpSession = request.getSession();
httpSession.setAttribute("filmModif", film);

// On ferme la session Hibernate
hibernateSession.close();

// Et on affiche le film dans le formulaire
request.setAttribute("film", film);
maVue = "/vues/transactions/editer2.jsp";
```

Édition du film

- À l'issue de cette action, l'objet **film** devient donc **détaché**.
- Dans l'action de mise à jour (quand l'utilisateur à soumis le formulaire), il faut **rattacher** à une nouvelle session Hibernate l'objet présent dans la session HTTP.
- Ce rattachement s'effectue avec la méthode **saveOrUpdate()** ou simplement **update()** : c'est notamment utile si on soupçonne que l'objet a déjà été modifié, et se trouve donc **dirty** avant même d'être réaffecté à la session.

Édition du film

```
Session hibernateSession = sessionFactory.openSession();
Transaction tx = null;
try {
  tx = hibernateSession.beginTransaction();

  // On prend le film dans la session HTTP
  HttpSession httpSession = request.getSession();
  Film film = (Film) httpSession.getAttribute("filmModif");

  // On le réinjecte dans la session Hibernate
  hibernateSession.saveOrUpdate(film);
  tx.commit();

  // Affichage
  request.setAttribute("film", film);
  maVue = "/vues/transactions/modifier2.jsp";
} catch (RuntimeException e) {
  if (tx != null)
    tx.rollback();
    throw e; // or display error message
  } finally {
    hibernateSession.close();
}
```

Édition du film

- La différence essentielle avec la version précédente est donc qu'on ne va pas chercher le film dans la base mais dans la session HTTP, et qu'on obtient donc l'objet avec le numéro de version **v** que l'on a vraiment édité.
- Si vous avez mis en place le versionnement Hibernate, la requête SQL comprendra une clause **where version=v**.

```
1     update Film set resume=?, version=? where id=? and version=?
```

- L'absence de cette version indiquerait qu'une mise à jour est intervenue entre temps.
- Hibernate lève une exception **StaleObjectStateException** avec le message suivant :

```
1     update Film set resume=?, version=? where id=? and version=?
```

- Faire l'expérience avec le code qui précède : éditer un même film avec deux navigateurs différents, et vérifier que le premier *commit* doit gagner, le second étant rejeté avec une exception **StaleObjectStateException**.

Méthodes des sessions longues

- Si on veut éviter de détacher/attacher des objets, on peut préserver l'ensemble de l'état d'une session entre deux requêtes HTTP.
- Attention : l'état d'une session comprend la connexion JDBC qu'il est impératif de relâcher avec **disconnect()** si on ne veut pas "enterrer" des connexions et se retrouver face au nombre maximal de connexions autorisé.
- Quand on reprend une action déclenchée par une nouvelle requête HTTP, il faut récupérer l'objet **session** Hibernate dans la session HTTP, et appeler **reconnect()**.
- On récupère alors l'ensemble des instances persistantes placées dans le cache.

♠ Il faut même penser à **fermer** la session quand la transaction applicative est terminée, ce qui suppose de savoir identifier une action "finale", ou au contraire de fermer/ouvrir la session pour toute action qui n'est pas intégrée à la séquence des actions d'une transaction applicative.

Plan du cours

3 Transactions applicatives
- Les stratégies possibles
- Versionnement avec Hibernate
- Par défaut, le dernier commit l'emporte!
- Granularité des sessions
- Méthode des objets détachés
- Méthode des sessions longues

4 Résumé : savoir et retenir

Résumé

- La question des transactions est délicate car elle implique des processus concurrents et une synchronisation temporelle qui sont difficiles à conceptualiser.
- Hibernate se comporte comme une application standard, la gestion de la concurrence étant réservée au SGBD.
- Le niveau d'isolation transactionnel par défaut des SGBD est permissif et autorise potentiellement (même si c'est rare) l'apparition d'anomalies dues à des transactions concurrentes. Il faut parfois savoir choisir le niveau d'isolation maximal pour s'en prémunir.
- Du point de vue de l'utilisateur, certaines transactions couvrent plusieurs sessions Hibernate, et la gestion de concurrence du SGBD devient alors ineffective. Hibernate fournit un support pour implanter un contrôle de concurrence multiversion dans ces cas là.