Java intensif
Programmation Web (3)

Serge Rosmorduc

2018–2019

Première partie I

Synchronized (et accès concurrents)

Accès concurrent et synchronized

Problèmes de l'accès à la session...

```
if (session.get("panier") == null) {
  session.setAttribute("panier", new Panier);
}
...
```

Accès aux beans sessions vus par glassfish

(extrait de code simplifié produit par une JSP)

```
appli.Panier panier = null;
synchronized (session) {
 panier = (appli.Panier) _jspx_page_context.getAttribute("panier", PageCo
 if (panier == null){
   panier = new appli.Panier();
   session.setAttribute("panier", panier, PageContext.SESSION_SCOPE);
 }
}
```

Deuxième partie II

Accès à une base de données : jdbc

JDBC

Java Database Connectivity

- API (définition d'interface) standard de bas niveau pour l'accès aux BD en java ;
- implémentée par des *drivers*

Un exemple : postgres

- L'interface jdbc est distribuée avec les sources, dans `postgresql-6.2/src/interfaces/jdbc` ;
- une fois compilée, la bibliothèque est utilisable sous tout ordinateur ;
- Les sources qui l'utilisent doivent contenir la ligne :

```
1        import java.sql.*;
```

- Pour charger le *driver* JDBC à utiliser, inclure dans le code :

```
1        Class.forName(postgresql.Driver);
```

(mais ça peut être fait par le serveur applicatif)

Se connecter et lire une base

- créer une connexion (`Connection`) à l'aide de la classe `DriverManager` ou d'une `DataSource` ;4
- créer un `Statement` (ou autre) pour communiquer ;
- quand on a fini, bien refermer le tout.
- la base est décrite par une URL — qui dépend du type de base.

Exemple

```
String url = "jdbc:h2:tcp://localhost//tmp/forum";
try (
 Connection db= DriverManager.getConnection(url, "root", "secret");
 Statement st = db.createStatement();
) {
  String sql= "insert into auteur values (2, 'lovelace', 'secret')";
  st.executeUpdate(sql);
} // try-with-resources !
```

Méthodes pour envoyer des requêtes

- execute : exécute une requête générale ;
- executeUpdate : modifie des lignes dans une table ; renvoie le nombre de lignes effectivement modifiées ;
- executeQuery : requête de type `select`.

Select

```
try (
  Statement st= db.createStatement();
  ResultSet res= st.executeQuery("select * from Etud");
) {
  while (res.next()) {
    System.out.println("col 1 = " + res.getString("Nom"));
  } // try-with-resources
}
```

- On peut accéder à un champ du résultat par son nom ou par son numéro — commencent à 1 !!!;
- on doit connaître son type (ici, « nom » est une chaîne de caractères ;
- on doit fermer le `ResultSet` après emploi (ici, automatique grâce au try-with-resources.

PreparedStatement

Commandes préparées

- À utiliser systématiquement pour intégrer des données venant de java dans une requête sql ;
- plus efficace dans certains cas (requêtes répétitives) ;
- surtout : protège contre l'injection SQL ;
- les données à injecter sont remplacées par des ' ?' dans la requête.

PreparedStatement

```
String insert= "insert into personne values (?,?,?)"
PreparedStatement pst = db.prepareStatement(insert);
pst.setInt(1, 10); // ID de 10
pst.setString(2, "Turing");
pst.setString(3, "Alan");
pst.executeUpdate();
pst.close(); // ou try-with-resources
```

Transactions

- par défaut, chaque requête forme une transaction ;
- pour changer ce comportement, on manipule l'objet `Connection` lié à la base de donnée :

```
maconnexion.setAutoCommit(false);
```

- ensuite :

`maconnexion.commit();` valide les requêtes déjà effectuées lors de cette transaction ;

`maconnexion.rollback();` annule les requêtes déjà effectuées ;

DataSource et DriverManager

- DriverManager : approche « manuelle », peu efficace ;
- DataSource : configurées dans le serveur applicatif ; gère un pool de connexions ; bonne séparation application/configuration.

Troisième partie III

Accès à une base de données : DAO

Architecture classique d'une application 3-tier

(faux ami : tier = étage en anglais ! !)

couche présentation (ui)
couche traitements (modèle, couche métier)
couche d'accès aux données (dao, persistance)

Accès au données : CRUD et DAO

DAO : Data Access Object
Pour chaque donnée métier, on aura la DAO correspondante, qui saura la sauvegarder dans une Base de Données
La DAO ne contient pas de logique métier. C'est un objet purement technique.
CRUD

- Create
- Retrieve
- Update
- Delete

La DAO

Pour chaque classe métier, par exemple la classe Personne, on aura la DAO correspondante
Elle fournit les méthodes :

- save(Personne p) : pour écrire les données d'une *nouvelle* entrée dans la base de données (Create) ;
- update(Personne p) : met à jour les données de la personne p ;
- delete(Personne p) ou delete(int id) : détruit les données de l'entrée correspondante ;
- findAll(), findById(id), etc... : une série de méthodes effectuant des recherches et renvoyant des objets (Retrieve).

Notez qu'il y a beaucoup de variations possibles sur les DAO ; nous présenterons un exemple — qui ne prétend pas être la seule solution possible.

Connection ?

- La DAO a besoin d'une connection ;
- en contexte multi-tâche, celle-ci ne doit pas être partagée simultanément par plusieurs threads : **ne pas conserver une connexion, ni l'utiliser en singleton !**
- une DataSource est un bâtisseur (factory) de connexion ; la datasource est, elle, partageable – et propose le pooling de connections ;
- on peut avoir des DAO à durée de vie courte, qui auront la Connection comme variables d'instances ;
- plusieurs DAO qui collaborent dans une même transaction doivent partager la même connexion ;
- dans le TP, on stockera les informations pour créer une connexion dans un singleton, qui servira à produire des connexions à la demande.

save

- typiquement fait un `insert` de l'objet ;
- s'occupe — éventuellement — de donner un identifiant aux objets ;
- retourne — éventuellement — l'objet créé ou son identifiant (pour traitement ou affichage ultérieur).

Argument de save

D'un point de vue logique, quand on appelle save, l'objet n'est pas ≪ complet ≫ : il n'a pas d'identifiant.
D'où trois possibilités :

1. on passe un objet ≪ normal ≫ comme argument, avec un identifiant indéfini (ou à null) ; c'est l'attitude la plus fréquente (voir l'exemple), mais elle est bancale d'un point de vue objet ;
2. on passe individuellement les données de l'objet comme argument (pour personne, le nom et le prénom) ; c'est un peu pénible ;
3. on crée une classe qui représente la requête de création et qui ne comporte pas l'id ; c'est propre, mais un peu redondant (duplication de code).

Bref, pas de solution parfaite...

save

```
public Personne save(Personne p) throws SQLException {
  String sql= "insert into personne values (null,?,?)";
  try(
    PreparedStatement pst =  db.prepareStatement(sql);
    Statement st= db.createStatement();
  ) {
    pst.setString(1, p.getNom());
    pst.setString(2, p.getPrenom());
    pst.executeUpdate();
    ResultSet r= st.executeQuery("select last_insert_id()");
    if (r.next()) {
      int id= r.getInt(1);
      p.setId(id);
      return p;
    } else {
      throw new RuntimeException("pas d'id trouvé??");
    }
  }
}
```

update

Mise à jour de *tous* les champs de l'objet.

```
public void update(Personne p) throws SQLException {
  String sql= "update personne set nom= ?, prenom= ? where id=?";
  try(PreparedStatement pst =  db.prepareStatement(sql);) {
    pst.setInt(1, p.getNom());
    pst.setInt(2, p.getPrenom());
    pst.setInt(3, p.getId());
    pst.executeUpdate();
  }
}
```

delete

```
public void delete(Personne p) throws SQLException {
  String sql= "delete from personne where id=?";
  try(PreparedStatement pst =  db.prepareStatement(sql);) {
    pst.setInt(1, p.getId());
    pst.executeUpdate();
  }
}
```

findAll

```
public List<Personne> findAll() throws SQLException {
  String sql= "select * from personne";
  try(
    Statement st= db.createStatement();
  ) {
    ResultSet r= st.executeQuery(sql);
    List<Personne> result= new ArrayList<>();
    while (r.next()) {
      int id= r.getInt("id");
      String nom= r.getString("nom");
      String prenom= r.getString("prenom");
      result.add(new Personne(id, nom, prenom));
    }
    return result;
  }
}
```

findById

```
public Personne findById(int id) throws SQLException {
  String sql= "select * from personne where id=?";
  try(
    PreparedStatement pst =  db.prepareStatement(sql);
  ) {
    pst.setInt(1, id);
    ResultSet r= pst.executeQuery();
    if (r.next()) {
      String nom= r.getString("nom");
      String prenom= r.getString("prenom");
      return new Personne(id, nom, prenom);
    } else {
      return null;
    }
  }
}
```

Les objets composites

- Les DAO sont faciles à écrire si les objets sont simples : pas de champs qui soient eux-même des objets composés ;
- mais dans un vrai modèle métier c'est peu plausible.
- donc, soit un modèle métier appauvri (chaque classe est l'image exacte d'une table ; au lieu d'objet liés les uns aux autres, on a des clefs étrangères) ;
- soit des DAO plus complexe à écrire : si un objet personne est lié à un objet ville, la méthode find de personne doit aussi charger la ville de la personne ;
- rapidement très compliqué à gérer : utilisation d'hibernate ou de JPA.