

JDBC

Olivier Cailloux

LAMSADE, Université Paris-Dauphine

Version du 19 décembre 2019

Présentation

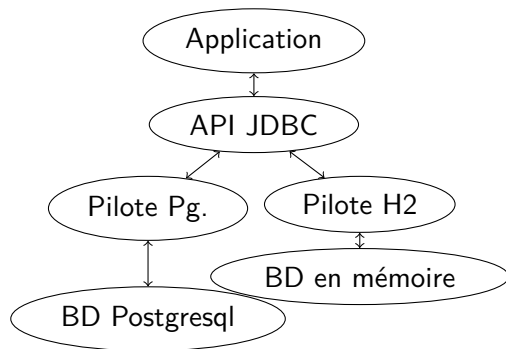
- JDBC ?

Présentation

- JDBC ? Java Database Connectivity

Présentation

- JDBC ? Java Database Connectivity
- Une API pour se connecter à des données relationnelles
- Programmation indépendante du fournisseur de BD
- App. programmée via API JDBC
- App. inclut pilotes du fournisseur
- Ces pilotes font la traduction



Instanciation

- Souhait : instancier pilote adéquat avec minimum de code spécifique à un fournisseur
- API JDBC nous fournit `DriverManager`
- Appeler `DriverManager#getConnection(url: String)`
- url au format `jdbc:subprotocol:subname`
- Exemple : `jdbc:postgresql:mydb` (cf. Doc JDBC PostgreSQL)

Mais comment ça marche ?

Fonctionnement de l'instanciation

- Le pilote fournisseur est inclus aux bibliothèques runtime de l'application
- Le JAR pilote inclut un fichier nommé (par convention) `META-INF/services/java.sql.Driver`
- Ce fichier nomme la classe que `DriverManager` doit charger
- `DriverManager` charge toutes ces classes (si plusieurs pilotes accessibles)
- Ayant l'URL, `DriverManager` cherche un pilote enregistré qui peut la lire
- Il instancie ce pilote et le renvoie à l'appelant ou l'utilise en arrière-plan

Remarques concernant l'instanciation

- Avec `DriverManager` on peut aussi obtenir le `Driver` (utile pour avoir n° de version par exemple)
- Beaucoup de tutoriels sur le net suggèrent d'enregistrer explicitement le pilote par exemple avec `Class.forName()`. Ce n'est plus nécessaire depuis longtemps (cf. explication précédente).

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 (Java EE)

- 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 USR set NAME = ? where ID = ?";
PreparedStatement stm = con.prepareStatement(s);
stm.setString(1, "NewName");
stm.setInt(2, 1234);
boolean isResultSet = stm.execute();
assert(!isResultSet);
assert(stm.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`

PostgreSQL

- Installer PostgreSQL (site ou `sudo apt-get install postgresql`)
- Possible d'utiliser l'interface graphique d'administration
pgAdmin voir logs si nécessaire
- Instructions ci-dessous pour ligne de commande linux, adapter pour autres OS
- En mode privilégié : `sudo -u postgres bash`
 - Se créer un utilisateur avec mot de passe : `createuser -P user`
 - Créer une base de données à laquelle cet utilisateur a accès : `createdb -O user db`
- Test connexion : `psql db` (ok sans mot de passe)
- Test connexion réseau : `psql -h localhost db` (exige mot de passe)
- Droits de connexion : voir `/etc/postgresql/9.4/main/pg_hba.conf` changer ligne IPv6 local connections puis reload

Serveur d'application

- Avec un serveur d'application, il faut renseigner le pilote JDBC utilisé dans JNDI
- Conseil : s'assurer d'abord que l'instanciation fonctionne avec un projet simple Java SE et Maven
- Via l'interface d'administration, indiquer la source à utiliser pour la connexion JNDI par défaut
- Inclure le pilote JDBC PostgreSQL dans les bibliothèques du serveur d'application

Deux patterns principaux

Deux patterns courants liés à persistance : *Active Record* et *DAO*

- Active Record : chaque objet responsable de sa propre persistance
- Chaque objet contient méthodes CRUD : Create, Read, Update, Delete
- DAO ?

Deux patterns principaux

Deux patterns courants liés à persistance : *Active Record* et *DAO*

- Active Record : chaque objet responsable de sa propre persistance
- Chaque objet contient méthodes CRUD : Create, Read, Update, Delete
- DAO ? Data Access Object

Deux patterns principaux

Deux patterns courants liés à persistance : *Active Record* et *DAO*

- Active Record : chaque objet responsable de sa propre persistance
- Chaque objet contient méthodes CRUD : Create, Read, Update, Delete
- DAO ? Data Access Object
- Classes dédiées à interaction avec BD
- Permet isolation de cet aspect
- Typiquement utilisé avec Data Transfer Object

Références

- Patterns of Enterprise Application Architecture
- Core J2EE Patterns - Data Access Object

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.