Kotlin et les bases de données
L'API JDBC via Java

Kotlin est un excellent candidat pour le développement d'applications de bases de données

- **Robuste et sécurisé**
- **Facile à comprendre**
- **Automatiquement téléchargeable par le réseau**

Avant JDBC, il était difficile d'accéder à des bases de données SQL depuis

Java.

Objectifs :

Permettre aux programmeurs de java d'écrire un code indépendant de la base de données et du moyen de connectivité utilisé

Qu'est ce que JDBC ?

JDBC(Java DataBase Connectivity)

- API Java adaptée à la connexion avec les bases de données relationnelles (SGBDR)
- Fournit un ensemble de classes et d'interfaces permettant l'utilisation

  sur le réseau d'un ou plusieurs SGBDR à partir d'un programmeJava

JDBC permet de communiquer avec plusieurs bases de données variées.

- Très grosses bases de données : Oracle, Informix, Sybase,…
- Petites bases de données : FoxPro, MS Access, mSQL.
- Il permet également de manipuler des fichiers textes ou les feuilles de calculs Excel.
- Liberté totale vis a vis du constructeurs

API JDBC

JDBC fournit un ensemble complet d'interfaces qui permet un accès mobile à une base de données. Java peut être utilisé pour écrire différents types de fichiers exécutables, tels que :

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs)

API JDBC

Est fournit par le package java.sql

permet de formuler et de gérer les requêtes aux bases de données relationnelles.

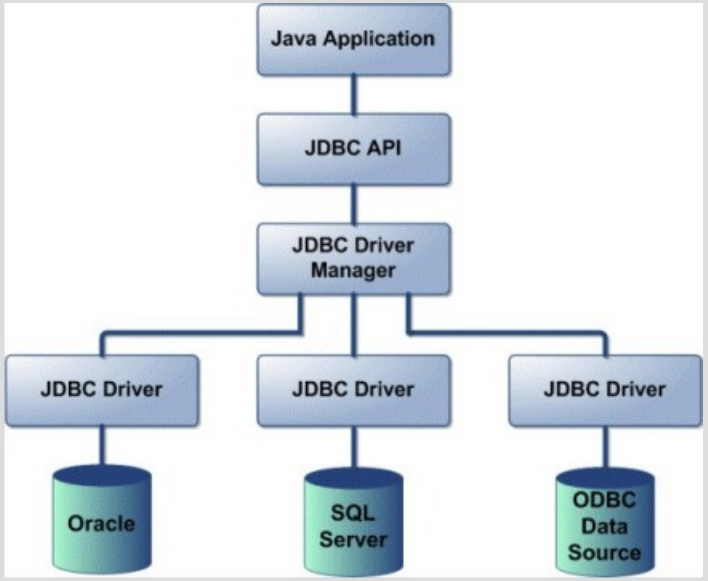
- supporte le standard SQL-2 Entry Level
- **8 interfaces :**
  - **Connexion à une base de données éloignée**
  - **Création et exécution de requêtes SQL**
  - **Statement**
  - **CallableStatement, PreparedStatement**
  - **DatabaseMetaData, ResultSetMetaData**
  - **ResultSet**
  - **Connection**
  - **Driver**

Principe de fonctionnement

Chaque base de données utilise un pilote (driver) qui est propre et qui permet de convertir les requêtes JDBC dans un langage natif du SGBDR.

Ces drivers dits JDBC existent pour tous les principaux constructeurs

(Oracle, Sybase, Informix, DB2,…)



Architecture JDBC

Un modèle à 2 couches

- **La couche externe : API JDBC**

C'est la couche visible pour développer des applications java accédant à des

SGBD.

- **Les couches inférieurs :**

- Destinées à faciliter l'implémentation de drivers pour des bases de données.

- Représentent une interface entre les accès de bas niveau au moteur du SGBDR et la partie applicative

Mettre en oeuvre JDBC

- Importer le package java.sql
- Enregistrer le driver JDBC
- Etablir la connexion à la base de données
- Créer une zone de description de requête
- Exécution des commandes SQL
- Inspection des résultats (si disponible)
- Fermer les différents espaces

Enregistrer le driver JDBC -
URL de Connexion à la base de données

Sur SQLDeveloper à la connexion :

- Nom de connexion : ...
- Login : **s3....**
- Passwd : **s3...**
- Nom du serveur : **172.26.82.31**
- Port : **1521**
- **xe**

en ***Kotlin*** :

```
Class.forName("oracle.jdbc.driver.OracleDriver").newInstance()

conn = DriverManager.getConnection  ("jdbc:oracle:thin:@172.26.82.31:1521:xe",
      username, password);
```

Avec ***var conn: Connection? = null*** **// où conn est de type Connection**

Exemple

```
class SessionOracle(username : String,password : String) {
var conn: Connection? = null
    Init { this.username = username
      this.password = password
      Try { Class.forName("oracle.jdbc.driver.OracleDriver").newInstance()
    conn = DriverManager.getConnection("jdbc:oracle:thin:@172.26.82.31:1521:xe",username, password)
         println("connexion réussie")
        } catch (ex: SQLException) {
           println(e.errorCode)//numéro d'erreur
            println(e.message)// message d'erreur qui provient d'oracle,
        }
}
    fun getConnectionOracle(): Connection? = conn
}
```

Statement

Une fois le connexion établie, pour pouvoir envoyer des requêtes SQL, il faut obtenir une instance de Statement sur laquelle on invoque des méthodes.

3 types de Statement :

Statement : requêtes statiques simples

PreparedStatement : requêtes dynamiques pré compilées

(avec paramètres E/S)

CallableStatement : procédures stockées.

A partir de l'instance de l'objet Connection, on récupère le statement associé :

Statement

val stmt: Statement = conn!!.**createStatement**()

val prepareStmt : **PreparedStatement**= conn!!.**prepareStatement**(str);

Val callstmt : **CallableStatement**=conn!!.**prepareCall**(str)

Exécution d ’une requête

3 types d’exécution :

executeQuery() : pour les requêtes (SELECT) qui retournent un RésultSet (tuples résultants).

executeUpdate() : pour les requêtes (*INSERT,UPDATE, DELETE, CREATE TABLE, DROP TABLE*) qui retournent un entier (nombre de tuples traités).

execute() : pour les procédures stockées.

Les résultats sont retournés dans une instance de la classe ***ResultSet***. Ce résultat est constitué d'une suite de lignes, le passage à la ligne suivante se fait par la méthode ***next*** de la classe *ResultSet*.

Exemple

```
fun read(){
        ….
        val requete: String="SELECT * FROM employe"
        try {
            val stmt: Statement = conn!!.createStatement()
            val result: ResultSet= stmt.executeQuery(requete)
            /* Parcourir le résultat du select avec la fonction next();*/
            while (result!!.next()) {
                val id = result.getInt("nuempl")
                val nom=result.getString("nomempl")
                println("$id $nom")
            } catch (e : SQLException) {…..}
        }
```

Exécution d ’une requête

- Le code SQL n’est pas interprété par java ou kotlin.C’est le pilote associé à la connexion (et au final par le moteur de la base de données) qui interprète la requête SQL. Si une requête ne peut s’exécuter ou qu’une erreur de syntaxe SQL à été détectée, l’exception **SQLException** est levée.

- Le driver JDBC effectue d’abord un accès à la base pour découvrir les types des colonnes impliquées dans la requête puis un 2ème pour l’exécuter.

Traitement des types retournés

- On peut parcourir le *ResultSet* **séquentiellement** avec la méthode : ***next***()
- Les colonnes sont référencés par **leur numéro** (commencent à 1) ou par leur **nom**.
- L'accès aux valeurs des colonnes se fait par les méthodes de la forme **getXXX**().

Lecture du type de données XXX dans chaque colonne du tuple courant

Types de données JDBC

- Le driver JDBC traduit le type JDBC retourné par le SGBD en un type **java ou kotlin** correspondant.
- Le XXX de getXXX() est le nom du type Java correspondant au type JDBC attendu.

Chaque driver a des correspondances entre les types SQL du SGBD.

Le programmeur est responsable du choix de ces méthodes.

SQLException est générée si mauvais choix

Cas des valeurs nulles

Pour repérer les valeurs NULL de la base :

Utiliser la méthode wasNull() de ResultSet

Revoie **true** si l'on vient de lire NULL, false sinon.

Les méthodes **getXXX**() convertissent une valeur NULL SQL en une valeur acceptable par le type d'objet demandé :

- Les méthodes (**getString**,…) peuvent retourner un null java.
- Les méthode numeriques (**getInt**(), …) retournent 0.
- **getBoolean**() retourne false.

Fermer les différents espaces

Pour terminer proprement un traitement, il faut fermer les différents espaces ouverts sinon le garbage collector s'en

occupera mais moins efficacement.

Chaque objet possède une méthode **close**() ;

```
resultset.close() ;

statement.close() ;

connection.close();
```

Requêtes pré-compilées

L'objet **PreparedStatement** envoi une requête **sans valeurs** à la base de données pour pré-compilation et **spécifiera le moment voulu la valeur des paramètres .**

plus rapide qu'un **Statement** classique.

le SGBD n'analyse qu'une seule fois la requête, pour de nombreuses exécutions d'une même requête SQL avec des paramètres variables.

val **requete**: String="insert into employe values(?,?,?,?,?)"

val ps : PreparedStatement ps=conn !!.prepareStatement(**requete**);

Les **arguments dynamiques** sont specifies par un "?"

Ils sont ensuite positionnés par les méthodes setInt(), setString(), …

ps.setInt(1,valeur1)…..ps.setInt(5, valeur5)

ps.executeUpdate()

Requêtes pré-compilées

SetNull() positionne le paramètre à NULL de SQL

Ces méthodes nécessitent 2 arguments

- Le premier (int) indique le numéro relatif de l'argument dans la requête.

- Le second indique la valeur à positionner.

val requete : String="UPDATE empl set sal= **?** where name = **?**"

val ps : PreparedStatement ps = conn!!.prepareStatement (requete);

```
for (...) {
                ps.setFloat(1, salary[i]);
                ps.setString(2, name[i]);
                ps.executeUpdate();
}
```

Accès aux procédures stockées

Accès aux données :

```
val requete: String="call lecture.liste_employes(?)"
val callstmt: CallableStatement = conn!!.prepareCall(requete)
        callstmt.registerOutParameter(1,OracleTypes.CURSOR)
        callstmt.execute()
        val result: ResultSet= callstmt.getObject(1) as ResultSet
```

Boucle pour récupérer le résultat

Accès aux procédures stockées

```
val requete: String="call MAJ.changer_de_chef(?,?)"
callstmt = conn!!.prepareCall(requete);
callstmt.setInt(1,numServ);
callstmt.setInt(2,numEmp);
callstmt.execute();
```

Pattern DAO (Data Access Object)

Permet de faire un lien entre la couche métier et la

couche de persistance

- Faire le **mapping** entre le système de stockage et les objets java
- Les DAOs sont placés dans la couche dite « d'accès aux données »

Utilité des DAOs

- Plus facile de modifier le code qui gère la persistance (changement de SGBD ou même de modèle de données)
- Factorise le code d'accès à la base de données plus facile pour le spécialiste des BD d'optimiser les accès (ils n'ont pas à parcourir toute l'application pour examiner les ordres SQL)

CRUD

Les opérations de base de la persistance :

- **Create**
- **Retrieve**
- **Update**
- **Delete**

DAOEmploye

```
package DAO

….

class DAOEmploye(val ss: SessionOracle) {

    var session: SessionOracle? = null

init { this.session=ss

    }

fun read(){

}

fun create(val ee:employe)…

fun delete(val ee:employe)…

fun update(val ee:employe)...

}
```