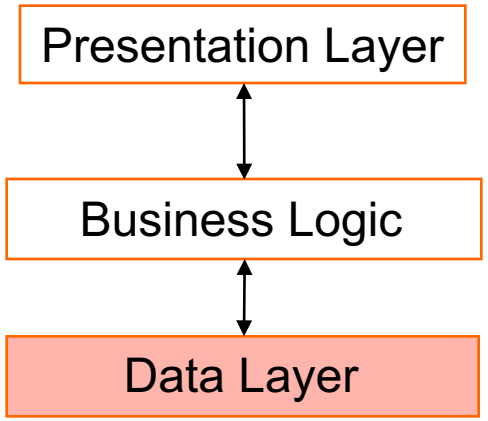
Desenvolvimento de Sistemas Software

Aula Teórica 25: Comunicação com bases de dados - JDBC

Desenvolvimento de Sistemas Software
José Creissac Campos / António Nestor Ribeiro / Rui Couto

Contexto

- Até agora tem sido usados ficheiros de texto e *file objects* para persistência.
- São formatos não escaláveis.
- Bases de dados fornecem persistência mais adequada.
- As ligações à base de dados são efectuadas na camada de dados, *Data Layer*.
- Esta camada permite isolar o acesso aos dados, por forma a que o resto da aplicação não esteja dependente da origem ou estrutura sob a qual os dados estão armazenados.



JDBC

- JDBC: **J**ava **D**ata**B**ase **C**onnectivity
- É uma API para Java, que define um conjunto de métodos para interagir com bases de dados.
- Diversas bibliotecas implementam esta API (e.g. MySQL, Postgres, SQLITE, etc.).
- A interação é feita através de classes específicas, que são as mesmas independentemente do motor de bases de dados.
- As API JDBC é parte do Java SE.
- As bibliotecas têm de ser carregadas de acordo com o motor de base de dados.

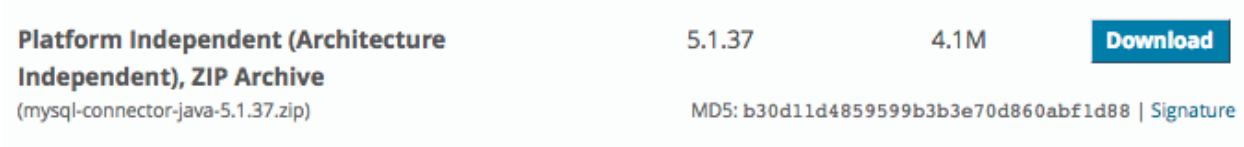
JDBC

- Passos usuais para integrar o JDBC:
  1. Importar a biblioteca para o projecto;
  2. Inicializar o *driver;*
  3. Estabelecer uma ligação;
  4. Executar as operações;
  5. Fechar a ligação;

JDBC – 1 Importar a biblioteca

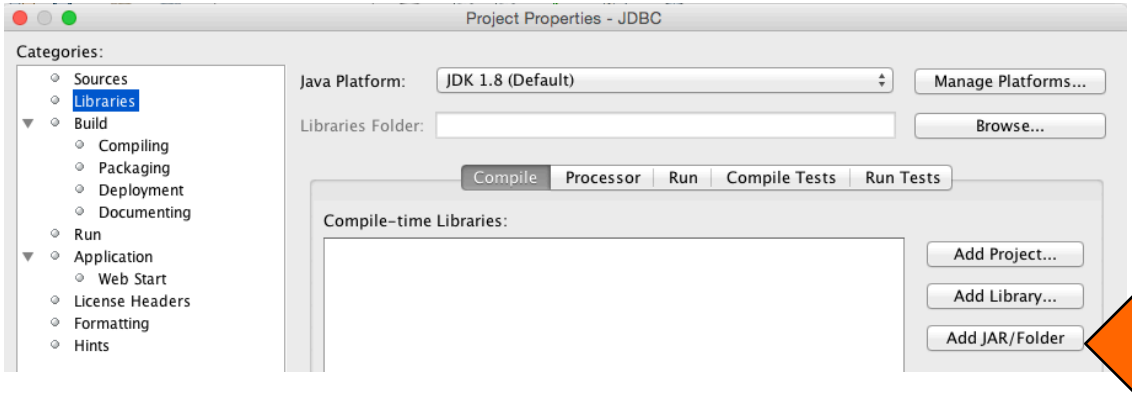
- Fazer download de biblioteca (jar) correspondente:
  - Exemplo para MySQL (connector/j):

    http://dev.mysql.com/downloads/connector/j/



- Importar a biblioteca para o projeto:
  - Exemplo NetBeans:

    Project > Properties > Libraries



mysql-connector-java-5.1.24-bin.jar

JDBC – 2 Inicializar o driver

- Adicionar *packages* respectivos:

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

- Carregar a classe do *driver* (exemplo MySQL):

```
Class.forName("com.mysql.jdbc.Driver");
```

excepção: `ClassNotFoundException`

```
try {
    Class.forName("com.mysql.jdbc.Driver");
}
catch (ClassNotFoundException e) {
    // Driver não disponível
}
```

JDBC – 2 Inicializar o driver

- Outras drivers:
  - Oracle:
    - `Class.forName("oracle.jdbc.driver.OracleDriver");`
  - PostgreSQL:
    - `Class.forName("org.postgresql.Driver");`
  - SQLite
    - `Class.forName("org.sqlite.JDBC");`

Drivers JDBC

- Existem várias implementações de drivers JDBC para os diversos motores de BD.
  - **Type 1**: drivers que implementam a API JDBC como um mapeamento para uma outra API. A portabilidade está dependente da existência do driver destino. A ligação a ODBC é um exemplo destes drivers;
  - **Type 2**: drivers que são escritos parcialmente em Java e numa outra linguagem. É utilizado um cliente específico para o acesso à base de dados pretendida;
  - **Type 3**: drivers escritos em Java e que comunicam com um servidor de middleware que faz o interface com as fontes de dados;
  - **Type 4**: drivers em que o cliente liga directamente à base de dados utilizando exclusivamente Java.

JDBC – 3 Estabelecer ligação

- Classe **`DriverManager`** disponibiliza os seguintes métodos de classe:

```
Connection getConnection(String url);
Connection getConnection(String url, String login, String pass);
Connection getConnection(String url, java.util.Properties.info);
```

excepção: `SQLException`

- **`Connection`** é uma interface que define um conjunto de métodos para operar com a base de dados.

JDBC – 3 Estabelecer ligação

- Ligação por URL:

  protocolo:subprotocolo:identificador

  •**protocolo**: é constante e representado pela *string* "jdbc"

  •**subprotocolo**: é função do motor da base de dados e da forma de acesso (direta ou indireta, por exemplo através de ODBC). No caso de MySQL o subprotocolo é "mysql".

  •**identificador**: Indica dados da base de dados a utilizar. No caso do MySQL o identificador é um URI da forma

  ```
  //ip/databaseName?user=dbusername&password=dbpassword
  ```

JDBC – 3 Estabelecer ligação

- Exemplo MySQL para ligar a base de dados local, com nome **alunos**, em que username é **uname** e password é **pwd**:

```
Connection con;

try {
    con =
DriverManager.getConnection("jdbc:mysql://localhost/alunos?user=uname&password=pwd
");
}
catch (SQLException e) {
    // Erro ao estabelecer a ligação
}
```

JDBC – 3 Estabelecer ligação

- Outras configurações:
  - Oracle:

```
DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:database", "username",
"password");
```

  - PostgreSQL:

```
DriverManager
.getConnection("jdbc:postgresql://localhost:5432/database",
”username", ”password");
```

  - SQLite:

```
DriverManager.getConnection("jdbc:sqlite:test.db");
```

JDBC – 4 Executar operações SQL

- Tipos de comandos SQL:
  - DDL: Data Definition Language

```
CREATE TABLE idFunc (
     cod VARCHAR(10) NOT NULL,
     nome VARCHAR(50),
     primary key(cod))
```

  - Selecção

```
SELECT * FROM cotações
```

  - Actualização

```
UPDATE clientes
          SET numerário = 100000
          WHERE nome = "João"
```

JDBC - 4 Executar operações SQL

- Process comum:
  - Criar um statement:

Interface **Connection**

```
Statement  createStatement();
```

excepção: `SQLException`

Interface **Statement**

```
ResultSet  executeQuery(String  sql);
int executeUpdate(String  sql);
```

excepção: `SQLException`

JDBC – 4 Executar operações SQL: SELECT

- Exemplo para comandos de selecção

```
Connection con;
... // iniciar a ligação

Statement st;
ResultSet res;
String sql;

sql = "SELECT nome FROM clientes WHERE numerário > 100000";

try {
    st = con.createStatement();
    res = st.executeQuery(sql);
} catch (SQLException e) {
    // lidar com as excepções
} finally {
    //fechar ligação
}
```

JDBC – 4 Executar operações SQL: Actualização/DD

- Do interface **Connection**:

```
int executeUpdate(String sql);
```

- Se for um comando de atualização (UPDATE, INSERT ou DELETE)

  Devolve o número de registos afectados

- Comandos DDL (ex: CREATE TABLE)

  devolve 0

```
Statement st = con.createStatement();

try {
    st = con.createStatement();
    int count = st.executeUpdate("UPDATE...");
} catch (SQLException e) {
    // lidar com as excepções
} finally {
    //fechar ligação
}
```

JDBC – 4 Executar operações SQL: Result Set

- Comandos de selecção resultam num conjunto de dados.
- Acesso aos resultados é feito através do **ResultSet**.

- Interface **ResultSet**
  - Funciona como iterador sobre os registos devolvidos

```
boolean next();
```

  - Dentro de um registo fornece um conjunto de métodos para aceder aos campos, por exemplo:

```
getString(int índiceDoCampo);
getString(String nomeDoCampo);
```

  - Um **ResultSet** está disponível até ser fechado, ou, o **Statement** ser reutilizado ou fechado.

JDBC – 4 Executar operações SQL: Result Set

- Exemplo de iteração de um **ResultSet**

```
Statement  st = con.createStatement();
ResultSet  rs;

try {
    rs = st.executeQuery("SELECT  saldo  FROM contas");
    int total = 0;
    while (rs.next()) {
        total += rs.getInt  ("saldo");
    }
    // nunca fazer isto em casa!!!
    System.out.println("Soma  :" + total");
} catch (SQLException  e) {
    // lidar com as excepções
} finally {
    //fechar ligação
}
```

JDBC – 4 Executar operações SQL

- JDBC não lida com questões de segurança (e.g. *SQL injection*)
- Input do utilizador é sempre (potencialmente) malicioso
- 'SELECT * FROM utilizador WHERE ID = N AND DATE > 'X'

```
' OR '1' = '1'; DELETE * FROM utilizador;
```

- **PreparedStatement** permite evitar esses problemas e efectuar queries de forma mais eficaz.
- Parâmetros das queries são definidos com ?, e o valor é atribuido com a seguinte interface:

```
setString(índiceDoParametro, string);
setInt(índiceDoParametro, inteiro);
setFloat(índiceDoParametro, float);
…
```

JDBC – 4 Executar operações SQL

JDBC – 4 Executar operações SQL

- Exemplo de *select*

```
PreparedStatement  st;
ResultSet  rs;

try {
    st = connect.prepareStatement("SELECT  nome  FROM contas WHERE saldo  > ? AND
name  = ?");
    st.setInt(1,  10000);
    st..setString(2,  ”João");
    rs = st.executeQuery();
    while  (rs.next())  {
        int  saldo  = rs.getInt(2);  //índice
        //nunca  fazer  isto  em  casa!!!
        System.out.println(“Saldo:  “ + saldo);
    }
} catch  (SQLException  e)  {
    // lidar  com  as  excepções
} finally  {
    //fechar  ligação
}
```

JDBC – 4 Executar operações SQL: *Transaction*

- A classe connection suporta também transacções.
- Permite executar um conjunto de operações, garantindo unicidade das operações.
- É conseguido à custa da configuração da instância de **Connection**.
- API:

```
con.setAutoCommit(false); //inicia transacção
con.commit(); //efectua transação
con.rollback(); //anula operações da transacção
```

  - Excepção:

```
SQLException //no caso de transacção ser abordata
```

JDBC – 4 Executar operações SQL: *Transaction*

- Exemplo de *transaction*:

```
try {
    con.setAutoCommit(false); //inicia transacion

    st = con.prepareStatement("INSERT INTO ...");
    st.executeUpdate();

    st = con.prepareStatement("UPDATE ...");
    st.executeUpdate();

    con.commit(); //efectua transaction
} catch (SQLException e) {
    con.rollback(); //anula transaction
} finally {
    //fechar ligação
}
```

JDBC – 4 Executar operações SQL: Timeout

- Ligações não permanecem activas indefinidamente.
- Se deixarmos uma ligação aberta, eventualmente ela vai expirar.
- Solução:
  - Abrir a conexão antes de efetuar as operações;
  - Fechar **antes** de retornar.

JDBC – 4 Executar operações SQL: Timeout

- Exemplo.

```
Public int saldo() {
    PreparedStatement st;
    ResultSet rs;
    int total = 0;

    try {
        st = connect.prepareStatement("SELECT nome FROM contas WHERE saldo > ? AND
name = ?");
        preparedStatement.setInt(1, 10000);
        preparedStatement.setString(2, ”João");
        rs = st.executeQuery();
        if (rs.next()) {
            total = rs.getInt(2); //índice
        }
    } catch (SQLException e) {
        // lidar com as excepções
    } finally {
        //fechar ligação
    }
    return total;
}
```

JDBC – 5 Fechar ligação

- Interface **Connection** fornece os métodos:

```
void close();

boolean isClosed();
```

excepção: `SQLException`

```
try {
    con.close();
}
catch (SQLException e) {
    // Erro ao terminar a ligação
}
```

Sumário

- A comunicação de aplicações Java com bases de dados necessita de:
  - Importar biblioteca para o projecto (através do IDE).
  - Importar no código packages respectivos.
  - Carregar a driver no código.
  - Criar statements e executar.
  - Fechar ligação.
- Transacções devem ser feitas de forma explicita.
- Driver não lida com todos os problemas:
  - É necessário lidar com *timeouts*, *sql injection*, etc.

Referências

- API de Java Connection:

  http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html