JDBC – Java Database Connectivity

Jorge Viana Doria Junior, M.Sc.
jjunior@unicarioca.edu.br

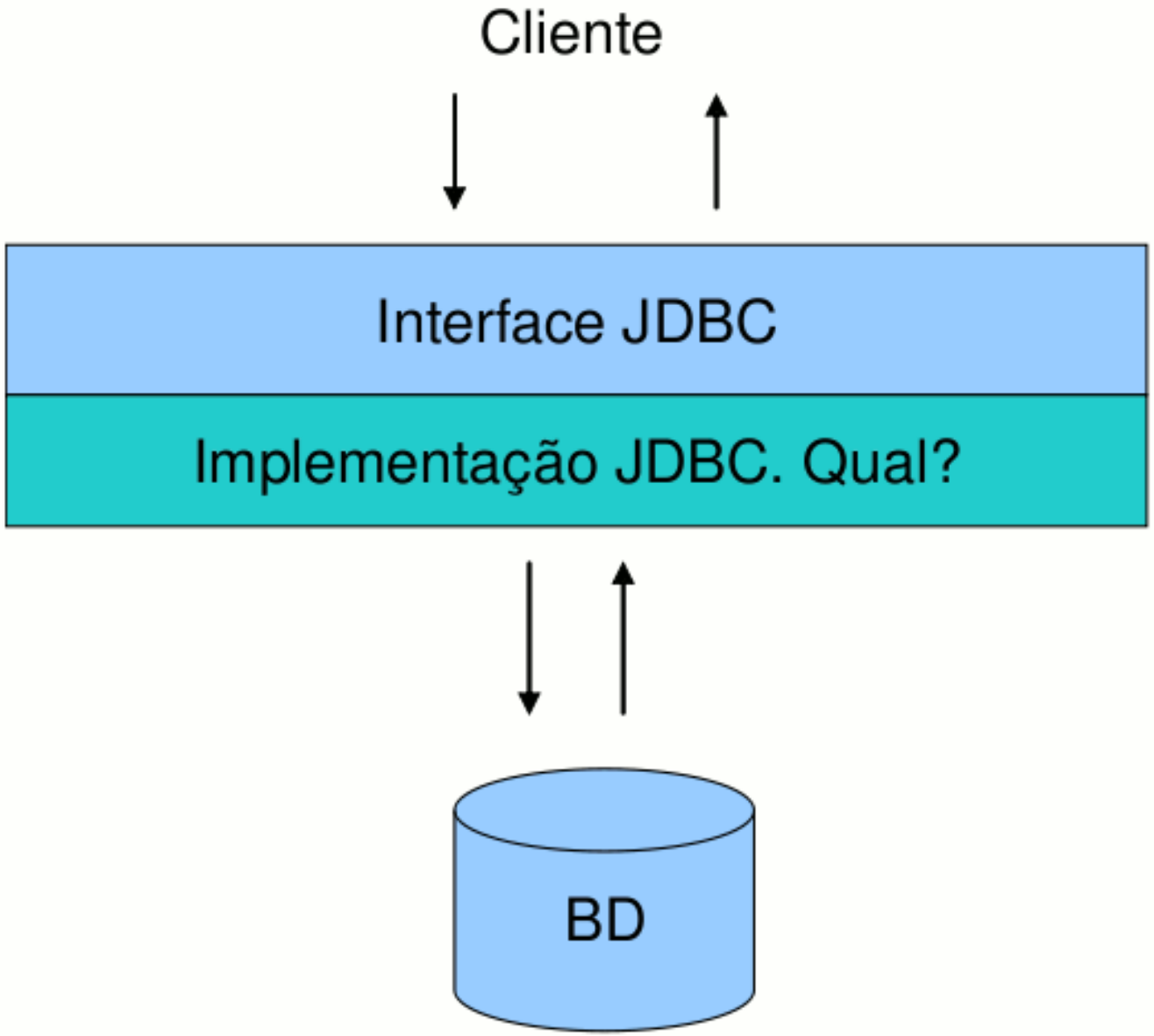
Problema

- Comunicação com o banco via sockets
  - Todo banco de dados possui um protocolo proprietário que pode ser acessado via sockets.
  - Poderíamos então abrir sockets diretamente com o banco de dados para realizar os comandos que desejássemos.
- Desvantagens:
  - Lidar com a complexidade dos protolocos proprietários de banco de dados.
  - Aprender um novo protocolo para cada banco de dados que quiséssemos acessar.

Solução

- JDBC é uma interface para acesso a bancos de dados através de SQL.
  - Permite o acesso a bancos de dados de forma padronizada
  - Código SQL é usado explicitamente dentro do código Java
  - API única, independente do Banco de Dados
  - Pacote: **java.sql**
- Para usar JDBC é preciso ter um driver JDBC
  - O banco deve ter pelo menos um driver ODBC, se não tiver driver JDBC
  - No JDK há um driver ODBC que permite o acesso a bancos

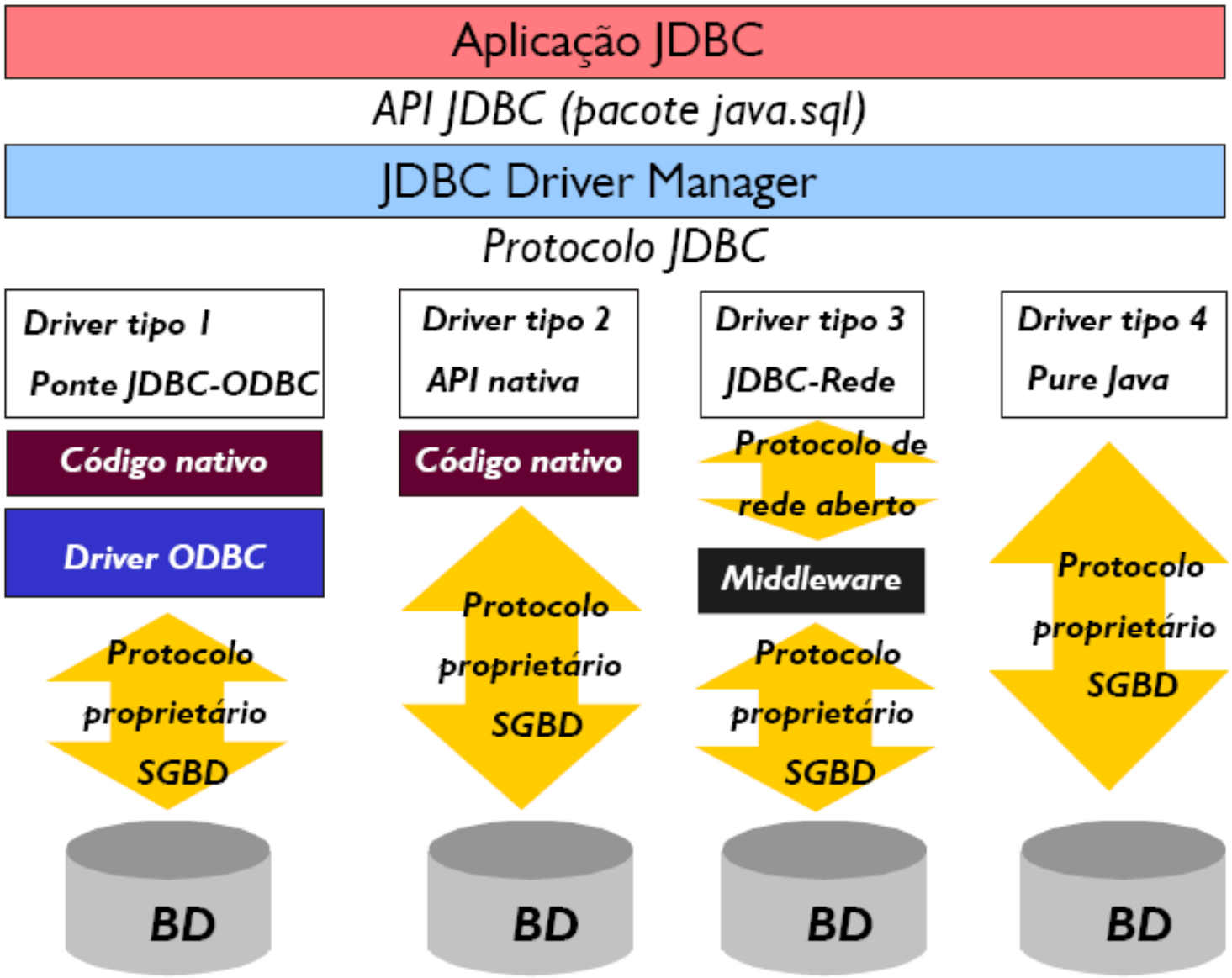
JDBC



Categorias de Drivers

- Tipo 1: ponte JDBC-ODBC
  - Usam uma ponte para ter acesso a um banco de dados. Este tipo de solução requer a instalação de software do lado do cliente.
- Tipo 2: solução com código nativo
  - Usam uma API nativa. Esses drivers contém métodos Java implementados em C ou C++. Requer software no cliente.
- Tipo 3: solução 100% Java no cliente
  - Oferecem uma API de rede via middleware que traduz requisições para API do driver desejado. Não requer software no cliente.
- Tipo 4: solução 100% Java
  - Drivers que se comunicam diretamente com o banco de dados usando soquetes de rede. É uma solução puro Java. Não requer código adicional do lado do cliente.

Categorias de Drivers



Passos para uso do JDBC

1. Adicionar o driver JDBC ao classpath
2. Carregar a classe do driver JDBC
3. Estabelecer a conexão a partir de uma URL
4. Criar um Statement
5. Executar uma query ou update
6. Processar os resultados
7. Fechar a conexão

Conectando com o banco

- Adicionar o driver JDBC ao classpath
  - MySQL:
    - mysql-connector-java-5.1.6-bin.jar

Conectando com o banco

- Carregar a classe do driver
  - Basta carregar o driver através do método **`Class.forName:`**

```
Class.forName("com.mysql.jdbc.Driver");
Class.forName("org.h2.Driver");
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

  - No java 6 e com driver que implementa a API JDBC 4 não é mais necessário carregar a classe para que o driver seja registrado no DriverManager.
    - Basta inserir o jar do driver no classpath e confiar no recurso de auto-discovery ou auto-loading de Driver JDBC.
  - Como o DriverManager sabe se um jar possui um driver?
    - O jar de um Driver possui o arquivo: META-INF/services/java.sql.Driver. Este arquivo armazena a classe do driver. No caso do mySQL será: com.mysql.jdbc.Driver.

Conectando com o banco

- Uso do driver
  - O serviço que permite o uso do driver é delegado para um gerente de drivers: o **`DriverManager`**.
- Obtendo a conexão
  - Através do **`DriverManager`**, usamos o método **`getConnection`** com uma url que indica que banco queremos acessar.
  - O padrão da url para acessar o mysql é:
    - **`jdbc:mysql://host/banco`**
    - **Exemplo: `jdbc:mysql://localhost/teste`**
  - Exemplos de URL para acessar o H2 é:
    - **`jdbc:h2:teste`**
    - **`jdbc:h2:db/poo;MODE=MYSQL;`**
    - **`jdbc:h2:tcp://localhost/teste`**

URL

- A URL JDBC tem o seguinte formato:
  - jdbc:subprotocolo:dsn
    - Subprotocolo – identifica qual driver será instanciado.
    - dsn - nome que o subprotocolo usa para identificar um servidor e/ou base de dados.
- Exemplos de URL:
  - `jdbc:odbc:nomedabasededados`
  - `jdbc:oracle:thin@localhost:nomedabasededados`
  - `jdbc:postgresql://localhost/nomedabasededados`
  - `jdbc:mysql://localhost/nomedabasededados`
  - `jdbc:h2:teste`
  - `jdbc:h2:tcp://localhost/teste`
  - `jdbc:derby:teste; create=true`
  - `jdbc:derby://localhost/teste; create=true`

Conectando com o banco - MySQL

```
import java.sql.*;

public class ExemplolJDBC {
  public static void main(String[] args) {
    try {
      Class.forName("com.mysql.jdbc.Driver");
      Connection con =
      DriverManager.getConnection("jdbc:mysql://localhost/posbd4",
                                  "root","root");
      System.out.println("Conectado");
      con.close();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Conectando e consultando o banco - MySQL

```
import java.sql.*;

public class Principal {

  public static void main(String[] args) {
    try {
      Connection c = DriverManager.getConnection(
                        "jdbc:mysql://localhost/teste",
                        "root", "root");
      PreparedStatement ps = c.prepareStatement(
                        "select nome from atleta order by nome");
      ResultSet rs = ps.executeQuery();
      while (rs.next()) {
        String nome = rs.getString("nome");
        System.out.println(nome);
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Conectando com o banco - H2

```
import java.sql.*;

public class ExemploH2 {

  public static void main(String[] args) {
    try {
      Class.forName("org.h2.Driver");
      Connection c =
             DriverManager.getConnection("jdbc:h2:db/test");
      System.out.println("Conectado");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Conectando e consultando o banco - H2

```
import java.sql.*;

public class ExemploH2 {

  public static void main(String[] args) {
    try {
      Class.forName("org.h2.Driver");
      Connection c = DriverManager.getConnection("jdbc:h2:db/test");
      PreparedStatement ps =
              c.prepareStatement("select name from test");
      ResultSet rs = ps.executeQuery();
      while (rs.next()) {
        String nome = rs.getString("name");
        System.out.println(nome);
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Fábrica de conexões - MySQL

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {

  public static Connection getConnection() throws SQLException {
    try {
      Class.forName("com.mysql.jdbc.Driver");
      return DriverManager.getConnection("jdbc:mysql://localhost/teste",
                                         "root","root");
    } catch (ClassNotFoundException e) {
      throw new SQLException(e.getMessage());
    }
  }

}
```

Fábrica de conexões – H2

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {

  public static Connection getConnection() throws SQLException {
    try {
      Class.forName("org.h2.Driver");
      return DriverManager.getConnection("jdbc:h2:db/teste");
    } catch (ClassNotFoundException e) {
      throw new SQLException(e.getMessage());
    }
  }

}
```

Criação da base no MySQL

```
create table contatos (id int primary key auto_increment,
nome varchar(50), email varchar(100), endereco
varchar(100));
```

Comandos SQL

- Opção 1

```
String sql = "insert into contatos
  (nome,email,endereco) values ('" + nome +
  "', '" + email + "', '" + endereco + "')";
```

- Opção 2

```
String sql = "insert into contatos
  (nome,email,endereco) values (?,?,?)";
```

- Vantagens da opção 2:
  - Melhor legibilidade
  - Evita SQL Injection (injeção de SQL)
  - Pode ser pre-compilada pelo SGBD

Comandos SQL

- A execução de comandos SQL pode ser feita via:
  - Statement
  - PreparedStatement
    - Interface que estende a interface Statement.
    - Possibilita o uso de comandos pre-compilados pelo SGBD.
    - Torna-se mais rápido quando usado repetidamente
    - Os parâmetros são representados pelo símbolo “?”.

Executando um comando SQL

```
// ...
PreparedStatement stmt = con.prepareStatement(
    "insert into contatos (nome,email,endereco) values (?,?,?)");
stmt.setString(1, "Caelum");
stmt.setString(2, "contato@caelum.com.br");
stmt.setString(3, "R. Vergueiro 3185 cj57");
stmt.executeUpdate();
stmt.close();
// ...
```

- O índice dos parâmetros inicia por 1.
- Para executar o comando podemos usar os métodos:
  - **`boolean execute()`** - qualquer comando SQL
  - **`int executeUpdate()`** - DML (insert, update, delete) ou DDL.
  - **`ResultSet executeQuery()`** - Somente para consultas

ResultSet

- Permite navegar por seus registros através do método next().

```
// ...
PreparedStatement stmt =
  this.con.prepareStatement("select * from contatos");
ResultSet rs = stmt.executeQuery();
List<Contato> contatos = new ArrayList<Contato>();
while (rs.next()) {
  Contato contato = new Contato();
  contato.setNome(rs.getString("nome"));
  contato.setEmail(rs.getString("email"));
  contato.setEndereco(rs.getString("endereco"));
  contatos.add(contato);
}
rs.close();
stmt.close();
return contatos;
```

Result Set

- O método executeQuery(), da interface Statement, retorna um objeto ResultSet.
  - Cursor para as linhas de uma tabela
  - Pode-se navegar pelas linhas da tabela recuperar as informações armazenadas nas colunas
- Os métodos de navegação são: next(), previous(), absolute(), first() e last()
- Métodos para obtenção de dados: getInt(), getString(), getDate()...

Recuperando dados de ResultSets

- Métodos getXXX
  - Usam-se métodos dos ResultSets começados por get
  - Recuperam um dado de acordo com o tipo
  - Como parâmetros podem ser usados a posição do campo (começando de 1) ou o nome do campo na tabela

Ex:
rs. getInt("codigo") ou rs.getInt(1)
rs. getString("descricao") ou rs.getString(2)

Recuperando dados de ResultSets

Método de ResultSet	*Tipo de dados SQL92*
`getInt()`	`INTEGER`
`getLong()`	`BIG INT`
`getFloat()`	`REAL`
`getDouble()`	`FLOAT`
`getBigDecimal()`	**`DECIMAL`**
`getBoolean()`	`BIT`
`getString()`	`CHAR, VARCHAR`
`getDate()`	`DATE`
`getTime()`	`TIME`
`getTimestamp()`	`TIME STAMP`
`getObject()`	*Qualquer tipo (Blob)*

Passos para uso do JDBC

- **1. Carregar o driver**
  2. Estabelecer uma conexão com o BD
  3. Criar uma declaração (PreparedStatement) e setar os parâmetros
  4. Executar o comando ou consulta SQL
  5. Processar o resultado
  6. Fechar a conexão

  Em Java:
  1. Class.forName("com.mysql.jdbc.Driver");

  2. Connection con = DriverManager.getConnection ("jdbc:mysql://localhost/teste", "root","root");

  3. PreparedStatement ps = con.prepareStatement();

  4.1. ps.executeUpdate("delete from Produtos"); // ou

  4.2. ResultSet rs = ps.executeQuery("select * from Produtos");

  5. while(rs.next()) {
  System.out.println(“Cód.: " + rs.getString(1) + " Desc: " + rs.getString(2));
  }

  6. con.close();

Fechando recursos

- Statements, PreparedStatements, ResultSets e Connections possuem um método close().
- Sempre que esses recursos não forem mais utilizados, é importante fechá-los.
- Ao fechar a conexão, PreparedStatements e ResultSets que estão associados à conexão serão automaticamente fechados.
- Exemplo de tratamento de exceções:

```
try {
  // ...
} catch (SQLException e) {
  // ...
} finally {
  try {
    con.close();
  } catch (SQLException e) {
    // ...
  }
}
```

Transações

- Para permitir a criação de transações com mais de um comando, o modo de auto-commit deve ser desabilitado
- O método **`setAutoCommit`** deve ser invocado no objeto do tipo Connection passando-se o valor false para ele

```
con.setAutoCommit(false);
```

- Para realizar, respectivamente, o **commit** ou o **rollback** em uma transação utilize os seguintes métodos do objeto Connection:

```
con.commit()

con.rollback()
```

- Nota: após terminar o uso de uma conexão, feche-a invocando o método **`close()`** da mesma

Fechando recursos com uso de transação

```
// ...

Connection conn = null;
try {
  conn = ConnectionFactory.getConnection();
  conn.setAutoCommit(false);
  // ...
  conn.commit();
} catch (SQLException ex) {
    try {
      conn.rollback();
    } catch (SQLException e) {
      // ...
    }
} finally {
    try {
       conn.close();
    } catch(SQLException ex){
      // ...
    }
}
```

DataSource

- Representa uma fábrica de conexões para o banco de dados físico
  - Alternativa mais adequada ao uso DriverManager
  - Sua implementação é de responsabilidade do fornecedor do banco de dados
    - Implementação Básica – cria conexões físicas da mesma forma do DriverManager
    - Implementação de Pool de Conexões – cria conexões que participarão de um pool (esta abordagem é usada em aplicações multi-camadas)
    - Implementação de Transações Distribuídas – semelhante à anterior exceto pelo fato de trabalhar com transações distribuídas

DataSource

```
...
DataSource ds = new
OracleConnectionPoolDataSource(url);
ds.getConnection(user, password);
...
```

```
...
Context initContext = new InitialContext();
Context envContext = (Context)initContext.lookup("java:/comp/env");
DataSource ds = (DataSource)envContext.lookup("jdbc/cefet");
conn = ds.getConnection());
...
```

Referências

- Apostila da Caelum FJ-21 – Java para desenvolvimento Web – Capítulo 2 - JDBC
- JDBC na Wikipedia:
  - http://pt.wikipedia.org/wiki/JDBC
- Encontrando Drivers JDBC
  - http://developers.sun.com/product/jdbc/drivers
- Página sobre o uso de Java com MySQL
  - http://dev.mysql.com/usingmysql/java/

Referências

- Elsmari, R., Navathe, Shamkant B. “Sistemas de Banco de Dados”. 6ª Edição, Pearson Brasil, 2011. → Capítulo 13
- Silberschatz, A., Korth, H., Sudarshan, S. “Sistema de Banco de Dados”. 5ª Edição, Editora Campus, 2006.
- Slides Prof. José Maria (UFC).

Dúvidas?