

JDBC

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Divisão de Computação
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande

Agenda

- 1 JDBC
- 2 Inserindo Dados No Banco
- 3 Fechando a conexão propriamente
- 4 DAO - Data Access Object
- 5 Fazendo Pesquisas no Banco de Dados

JDBC

Conectar-se a um banco de dados com Java é feito de maneira elegante.

Para evitar que cada banco tenha a sua própria API e conjunto de classes e métodos, **temos um único conjunto de interfaces muito bem definidas que devem ser implementadas.**

- Esse conjunto de interfaces fica dentro do pacote *java.sql* e nos referiremos a ela como JDBC.
 - Ex: interface **Connection**: define métodos para executar uma query, fechar a conexão e etc.

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

public class Main {
    public static void main(String[] args) throws ClassNotFoundException, SQLException {
        String database = "docente";
        String host = "localhost";
        String port = "5432";
        String user = "postgres";
        String password = "postgres";
        String url = "jdbc:postgresql://" + host + ":" + port + "/" + database;
        Class.forName("org.postgresql.Driver");
        Connection con;
        con = DriverManager.getConnection(url, user, password);
        System.out.println("Conexão realizada com sucesso.");
        con.close();
    }
}
```

Para que o código acima venha a funcionar, você deverá colocar o *driver* do PostgreSQL nas bibliotecas do projeto que você criou.

JDBC

E o Class.forName?

Até a versão 3 do JDBC, antes de chamar o `DriverManager.getConnection()` era necessário registrar o driver JDBC que iria ser utilizado através do método `Class.forName("XXXXXX")`, que carregava essa classe, e essa se comunicava com o `DriverManager`.

- A partir do JDBC 4, que está presente no Java 6, esse passo não é mais necessário.

JDBC

Mas lembre-se (1):

Caso você utilize JDBC em um projeto com Java 5 ou anterior, será preciso fazer o registro do Driver JDBC, carregando a sua classe, que vai se registrar no DriverManager.

Mas lembre-se (2):

Isso também pode ser necessário em alguns servidores de aplicação e web, como no Tomcat 7 ou posterior, por proteção para possíveis vazamentos de memória:

JDBC

E quem sabe encapsular a conexão?

```
public class ConnectionFactory {  
    public Connection getConnection() {  
        try {  
            return DriverManager.getConnection(  
                "jdbc:mysql://localhost/fj21", "root", "");  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

JDBC

E quem sabe encapsular a conexão?

```
public class ConnectionFactory {  
    public Connection getConnection() {  
        try {  
            return DriverManager.getConnection(  
                "jdbc:mysql://localhost/fj21", "root", "");  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

O método **getConnection()** é uma fábrica de conexões, isto é, ele cria novas conexões. Basta invocar o método e recebemos uma conexão pronta para uso, não importando de onde elas vieram e eventuais detalhes de criação.

Inserindo Dados No Banco

Para inserir dados em uma tabela de um banco de dados entidade-relacional basta usar a cláusula INSERT. Precisamos especificar quais os campos que desejamos atualizar e os valores.

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

Inserindo Dados No Banco

Para inserir dados em uma tabela de um banco de dados entidade-relacional basta usar a cláusula INSERT. Precisamos especificar quais os campos que desejamos atualizar e os valores.

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

Observação

O exemplo possui 3 pontos negativos importantíssimos

Inserindo Dados No Banco

- 1 O programador que não escreveu o código original não consegue bater o olho e entender o que está escrito. **O que o código acima faz? Mais difícil ainda é saber se faltou uma vírgula, um fecha parênteses talvez.**

Inserindo Dados No Banco

- 1 O programador que não escreveu o código original não consegue bater o olho e entender o que está escrito. **O que o código acima faz? Mais difícil ainda é saber se faltou uma vírgula, um fecha parênteses talvez.**
- 2 Outro problema é o SQL Injection. O que acontece quando o contato a ser adicionado possui no nome uma aspas simples? O código SQL se quebra todo e para de funcionar ou, pior ainda, o usuário final é capaz de alterar seu código SQL para executar aquilo que ele deseja (SQL injection).

Inserindo Dados No Banco

- ❶ O programador que não escreveu o código original não consegue bater o olho e entender o que está escrito. **O que o código acima faz? Mais difícil ainda é saber se faltou uma vírgula, um fecha parênteses talvez.**
- ❷ Outro problema é o SQL Injection. O que acontece quando o contato a ser adicionado possui no nome uma aspas simples? O código SQL se quebra todo e para de funcionar ou, pior ainda, o usuário final é capaz de alterar seu código SQL para executar aquilo que ele deseja (SQL injection).
- ❸ **Mais um problema que enxergamos aí é na data.** Ela precisa ser passada no formato que o banco de dados entenda e como uma String, portanto, se você possui um objeto `java.util.Calendar` (que é o nosso caso), você precisará fazer a conversão desse objeto para a String.

Inserindo Dados No Banco

Por esses três motivos não usaremos código SQL como mostrado anteriormente. Vamos imaginar algo mais genérico e um pouco mais interessante:

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

Não colocamos os pontos de interrogação de brincadeira, mas sim porque realmente não sabemos o que desejamos inserir.

Inserindo Dados No Banco

As cláusulas são executadas em um banco de dados através da interface `PreparedStatement`.

Para receber um **`PreparedStatement`** relativo à conexão, basta chamar o método **`prepareStatement`**, passando como argumento o comando SQL com os valores vindos de variáveis preenchidos com uma interrogação.

```
String sql = "insert into contatos " +  
             "(nome,email,endereco,dataNascimento) " +  
             "values (?, ?, ?, ?)";  
PreparedStatement stmt = connection.prepareStatement(sql);
```

Inserindo Dados No Banco

Logo em seguida, chamamos o método **setString** do **PreparedStatement** para preencher os valores que são do tipo **String**, passando a posição (começando em 1) da interrogação no SQL e o valor que deve ser colocado:

```
// preenche os valores  
stmt.setString(1, "Caelum");  
stmt.setString(2, "contato@caelum.com.br");  
stmt.setString(3, "R. Vergueiro 3185 cj57");
```


Inserindo Dados No Banco

Precisamos definir também a data de nascimento do nosso contato, para isso, precisaremos de um objeto do tipo `java.sql.Date` para passarmos para o nosso **PreparedStatement**.

```
java.sql.Date dataParaGravar = new java.sql.Date(  
    Calendar.getInstance().getTimeInMillis());  
stmt.setDate(4, dataParaGravar);
```

Nesse exemplo, estamos passando a data atual.

Por fim, uma chamada a **execute()** executa o comando SQL:

```
stmt.execute();
```

Fechando a conexão propriamente

Veja o exemplo:

```
public class JDBCInserere {  
    public static void main(String[] args) throws SQLException {  
        Connection con = null;  
        try {  
            con = new ConnectionFactory().getConnection();  
            // faz um monte de operações.  
        } catch (SQLException e) {  
            System.out.println(e);  
        } finally {  
            con.close();  
        }  
    }  
}
```

Dessa forma, mesmo que o código dentro do **try** lance **exception**, o **con.close()** será executado. Garantimos que não deixaremos uma conexão pendurada sem uso.

Fechando a conexão propriamente

No Java 7 há a estrutura **try-with-resources**. Ela permite declarar e inicializar, dentro do `try`, objetos que implementam **AutoCloseable**.

- Ao término do `try`, o compilador insere instruções para invocar o **close** desses recursos, além de se precaver em relação a exceções que podem surgir por causa dessa invocação:

```
try(Connection con = new ConnectionFactory().getConnection()) {  
    // faz um monte de operações.  
    // que podem lançar exceptions runtime e SQLException  
} catch(SQLException e) {  
    System.out.println(e);  
}
```

Nosso código ficaria mais reduzido e organizado, além do escopo de `con` só valer dentro do `try`

DAO - Data Access Object

Já foi possível sentir que colocar código SQL dentro de suas classes de lógica é algo nem um pouco elegante e muito menos viável quando você precisa manter o seu código.

A ideia a seguir é remover o código de acesso ao banco de dados de suas classes de lógica e colocá-lo em uma classe responsável pelo acesso aos dados. Assim o código de acesso ao banco de dados fica em um lugar só, tornando mais fácil a manutenção.

DAO - Data Access Object

Que tal se pudéssemos chamar um método **adiciona** que adiciona um **Contato** ao banco?

Em outras palavras quero que o código a seguir funcione:

```
public static void main(String[] args) {  
    // pronto para gravar  
    Contato contato = new Contato();  
    contato.setNome("Caelum");  
    contato.setEmail("contato@caelum.com.br");  
    contato.setEndereco("R. Vergueiro 3185 cj87");  
    contato.setDataNascimento(Calendar.getInstance());  
  
    // grave nessa conexão!!!  
    Misterio bd = new Misterio();  
  
    // método elegante  
    bd.adiciona(contato);  
  
    System.out.println("Gravado!");  
}
```

DAO - Data Access Object

O código anterior já mostra o poder que alcançaremos: através de uma única classe seremos capazes de acessar o banco de dados e, mais ainda, somente através dessa classe será possível acessar os dados.

Esta ideia, inocente à primeira vista, é capaz de isolar todo o acesso a banco em classes bem simples, cuja instância é um objeto responsável por acessar os dados.

Da responsabilidade deste objeto surgiu o nome de *Data Access Object* ou simplesmente DAO, um dos mais famosos padrões de projeto (*design pattern*).

DAO - Data Access Object

O que falta para o código acima funcionar é uma classe chamada ContatoDao com um método chamado adiciona . Vamos criar uma que se conecta ao banco ao construirmos uma instância dela:

```
public class ContatoDao {  
  
    // a conexão com o banco de dados  
    private Connection connection;  
  
    public ContatoDao() {  
        this.connection = new ConnectionFactory().getConnection();  
    }  
}
```

DAO - Data Access Object

Agora que todo **ContatoDao** possui uma conexão com o banco, podemos focar no método adiciona , que recebe um Contato como argumento e é responsável por adicioná- lo através de código SQL:

```
public void adiciona(Contato contato) {  
    String sql = "insert into contatos " +  
        "(nome,email,endereco,dataNascimento)" +  
        " values (?, ?, ?, ?)";  
  
    try {  
        // prepared statement para inserção  
        PreparedStatement stmt = con.prepareStatement(sql);  
  
        // seta os valores  
  
        stmt.setString(1, contato.getNome());  
        stmt.setString(2, contato.getEmail());  
        stmt.setString(3, contato.getEndereco());  
        stmt.setDate(4, new Date(  
            contato.getDataNascimento().getTimeInMillis()));  
  
        // executa  
        stmt.execute();  
        stmt.close();  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
}
```


Fazendo Pesquisas no Banco de Dados

Para pesquisar também utilizamos a interface `PreparedStatement` para montar nosso comando SQL.

- Mas como uma pesquisa possui um retorno (diferente de uma simples inserção), usaremos o método `executeQuery` que retorna todos os registros de uma determinada *query*.

Método `executeQuery`

O objeto retornado pelo método é do tipo **ResultSet** do JDBC, o que nos permite navegar por seus registros através do método **next**.

- Esse método retornará **false** quando chegar ao fim da pesquisa, portanto ele é normalmente utilizado para **fazer um laço nos registros**.

Fazendo Pesquisas no Banco de Dados

```
// pega a conexão e o Statement
Connection con = new ConnectionFactory().getConnection();
PreparedStatement stmt = con.prepareStatement("select * from
contatos");

// executa um select
ResultSet rs = stmt.executeQuery();

// itera no ResultSet
while (rs.next()) {
    String nome = rs.getString("nome");
    String email = rs.getString("email")

    System.out.println(nome + " :: " + email);
}

stmt.close();
con.close();
```

Para retornar o valor de uma coluna no banco de dados, basta chamar um dos métodos *get* do **ResultSet**, dentre os quais, o mais comum: *getString*.

Fazendo Pesquisas no Banco de Dados

Podemos novamente aplicar as ideias de **DAO** e criar um método *getLista()* no nosso **ContatoDao**.

Mas o que esse método retornaria?

Um **ResultSet**?

E teríamos o código de manipulação de **ResultSet** espalhado por todo o código?

Vamos fazer nosso **getLista()** devolver algo mais interessante, uma **lista de Contato**:

Fazendo Pesquisas no Banco de Dados

```
PreparedStatement stmt = this.connection
    .prepareStatement("select * from contatos");
ResultSet rs = stmt.executeQuery();
List<Contato> contatos = new ArrayList<Contato>();
while (rs.next()) {
    // criando o objeto Contato
    Contato contato = new Contato();
    contato.setNome(rs.getString("nome"));
    contato.setEmail(rs.getString("email"));
    contato.setEndereco(rs.getString("endereco"));
    // montando a data através do Calendar
    Calendar data = Calendar.getInstance();
    data.setTime(rs.getDate("dataNascimento"));
    contato.setDataNascimento(data);
    // adicionando o objeto à lista
    contatos.add(contato);
}
rs.close();
stmt.close();
return contatos;
```

JDBC

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Divisão de Computação
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande