

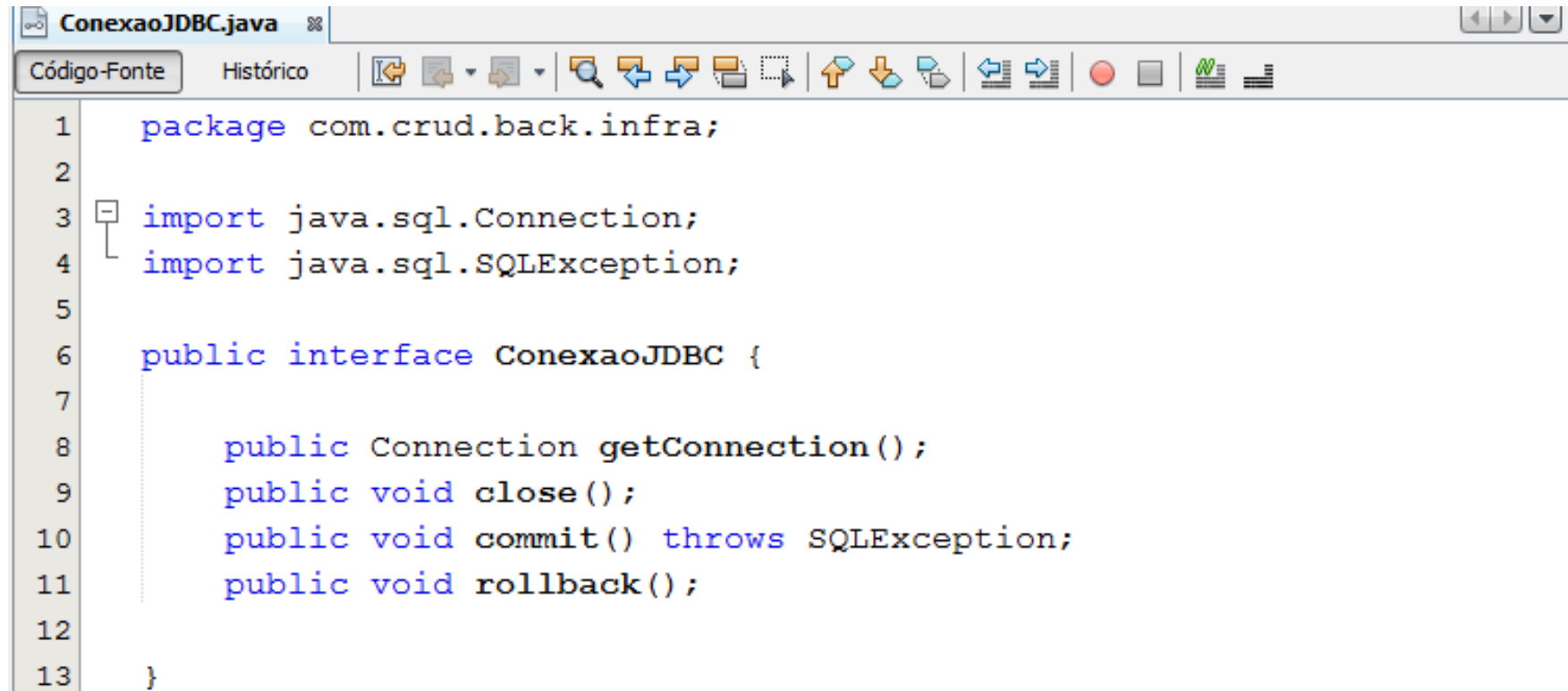
# Introdução ao AngularJS

Professor Anderson Henrique



# JDBC (Infra-Estrutura)

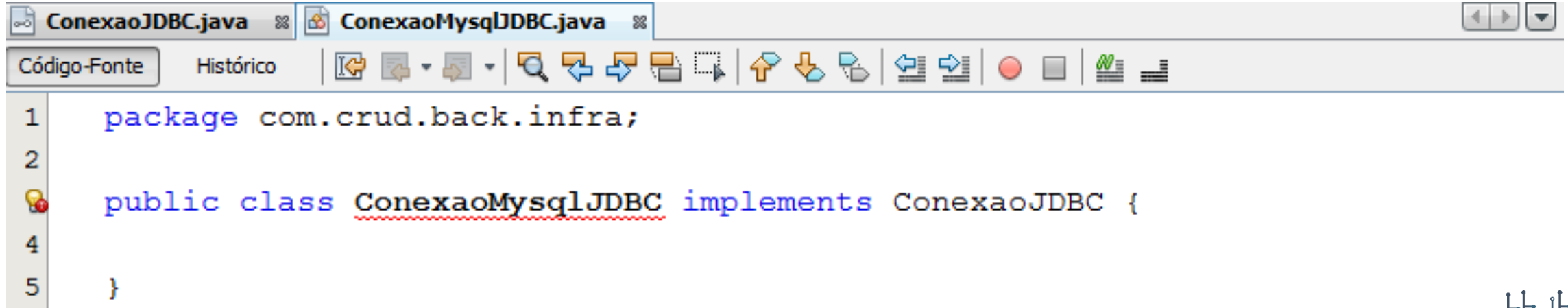
Crie um novo pacote dentro da pasta Pacotes de Códigos-fonte chamado **com.crud.back.infra** e dentro do pacote crie uma interface chamada **ConexaoJDBC**



```
1 package com.crud.back.infra;
2
3 import java.sql.Connection;
4 import java.sql.SQLException;
5
6 public interface ConexaoJDBC {
7
8     public Connection getConnection();
9     public void close();
10    public void commit() throws SQLException;
11    public void rollback();
12
13 }
```

Temos o método para pegar a conexão, um para fechar a conexão, um para confirmar as transações (**commit**) e outro para desfazer as transações (**rollback**)

Neste mesmo pacote vamos criar uma classe java que vai implementar essa interface chamada **ConexaoMySQLJDBC**



```
ConexaoJDBC.java  ConexaoMySQLJDBC.java
Código-Fonte  Histórico
1  package com.crud.back.infra;
2
3  public class ConexaoMySQLJDBC implements ConexaoJDBC {
4
5  }
```

Todos os métodos da interface `ConexaoJDBC` devem ser implementados obrigatoriamente

Vamos criar um construtor da classe que vai utilizar o driver de conexão do MySQL, para isso devemos adicionar essa dependência no **Maven** no arquivo **pom.xml**

```
35 <dependency>
36     <groupId>mysql</groupId>
37     <artifactId>mysql-connector-java</artifactId>
38     <version>8.0.21</version>
39 </dependency>
```

Vamos declarar um atributo privado do tipo Connection chamado **connection** que receberá o valor **null**

```
12 public ConexaoMysqlJDBC() throws ClassNotFoundException, SQLException{
13     Class.forName("com.mysql.cj.jdbc.Driver");
14
15     Properties properties = new Properties();
16     properties.put("user", "root");
17     properties.put("password", "");
18
19     this.connection =
20         DriverManager.getConnection("jdbc:mysql://localhost:3306/api_java", properties);
21     this.connection.setAutoCommit(false);
22 }
```

Assim que instanciarmos a classe o **construtor** se responsabilizará por estabelecer uma conexão com o Banco de Dados

O método **getConnection** simplesmente retorna o atributo que armazenará a conexão

```
24      @Override
25      public Connection getConnection() {
26          return this.connection;
27      }
```

No método **close**, vamos verificar se o valor do atributo é diferente de **null**, se **SIM**, vamos tentar fechar a conexão, **CASO** ocorra exceção, vamos lançar um **log** no servidor Tomcat

```

31  @Override
32  public void close() {
33      if(this.connection != null){
34          try{
35              this.connection.close();
36          }catch(SQLException ex){
37              Logger.getLogger(ConexaoMysqlJDBC.class.getName())
38                  .log(Level.SEVERE, null, ex);
39          }
40      }
41  }

```

O log que será lançado será do nível SEVERO pelo SQLException

No método **commit** vamos simplesmente pegar a conexão e dar um commit, dentro do método vamos chamar o método **close()** que tem a verificação se existe conexão, o tratamento da exceção será diferente

```
43      @Override
44      public void commit() throws SQLException {
45          this.connection.commit();
46          this.close();
47      }
```

Para finalizar, no método **rollback** vamos verificar se existe conexão, se **SIM** vamos dar um rollback, **CASO** ocorra exceção vamos lançar um log no nível SEVERO no servidor Tomcat e independente se lançar ou não finalmente vamos fechar a conexão com o método **close()**



```
49      @Override
50      public void rollback() {
51          if(this.connection != null){
52              try{
53                  this.connection.rollback();
54              }catch(SQLException ex){
55                  Logger.getLogger(ConexaoMysqlJDBC.class.getName())
56                      .log(Level.SEVERE, null, ex);
57              }finally{
58                  this.close();
59              }
60          }
61      }
```

Pronto! Finalizamos a implementação de todos os métodos da classe

Precisamos criar o banco de dados e a tabela que será usada na nossa aplicação, para isto vamos iniciar o serviço do **WAMPServer**, no **phpMyAdmin**, na guia **SQL** inserir e executar esse script

```
1  /* criando o banco de dados */
2  CREATE DATABASE api_java;
3
4  /* selecionando o banco */
5  USE api_java;
6
7  /* criando a tabela no banco */
8  CREATE TABLE chamado(
9      id INT PRIMARY KEY AUTO_INCREMENT,
10     assunto VARCHAR(256) NOT NULL,
11     mensagem VARCHAR(2048) NOT NULL,
12     status VARCHAR(32) NOT NULL
13 );
```

Onde vamos utilizar a nossa classe `ConexaoMySQLJDBC`? Dentro do pacote **`com.crud.back.bean`** vamos criar uma classe chamada **ChamadoDAO**

Nesta classe vamos utilizar a nossa interface **`ConexaoJDBC`** e não a classe `ConexaoMySQLJDBC`

Vamos criar um **atributo privado final** do tipo `ConexaoJDBC`

Este atributo deve ser inicializado com o **construtor**, assim criaremos no construtor uma **nova conexão** do tipo `ConexaoMySQLJDBC`

```
11 public class ChamadoDAO {
12
13     private final ConexaoJDBC conexao;
14
15     public ChamadoDAO() throws SQLException, ClassNotFoundException {
16         this.conexao = new ConexaoMysqlJDBC();
17     }
```

Agora, na classe ChamadoDAO teremos os métodos: **inserir**, **alterar**, **excluir**, **listar** e **listarTudo** e precisamos implementá-los

O método **inserir** não retorna valor, a string da **query** recebe os values **curingas**, depois, tentamos preparar o **statement** na conexão pegando a conexão, **setamos** os valores do objeto e inserimos na sintaxe da query, executamos o statement e damos um **commit**, CASO ocorra uma exceção desfaz a operação na conexão com o método **rollback** , lance a exceção

```
19 public void inserir(Chamado chamado) throws SQLException{
20     String sqlQuery = "INSERT INTO chamado (assunto, mensagem, status) "
21         + "VALUES (?, ?, ?)";
22     try {
23         PreparedStatement stmt = this.conexao.getConnection()
24             .prepareStatement(sqlQuery);
25         stmt.setString(1, chamado.getAssunto());
26         stmt.setString(2, chamado.getMensagem());
27         stmt.setString(3, chamado.getStatus().toString());
28         stmt.executeUpdate();
29
30         this.conexao.commit();
31     } catch (SQLException e) {
32         this.conexao.rollback();
33         throw e;
34     }
35 }
```

No método **alterar** vamos retorna um valor **int**, declaramos uma variável **linhasAfetadas** que recebe o valor **0**, declaramos a string da **query** passando os valores como curingas, tentamos preparar o **statement** na conexão, pegando a conexão, setamos os valores sendo o 4º parâmetro o valor do **id** e pedimos para **executar** o **update** neste statement o qual retornará a **quantidade** de **linhas afetadas** e armazenará na **variável** e pedimos para dar um **commit**, CASO ocorra exceção, vamos desfazer a operação na conexão com o método **rollback** e lança a exceção, no final o método retornará o valor de **linhas afetadas**

```
39 public int alterar(Chamado chamado) throws SQLException{
40     int linhasAfetadas = 0;
41     String sqlQuery = "UPDATE chamado SET assunto=?, mensagem=?"
42         + "status=? WHERE id=?";
43     try {
44         PreparedStatement stmt = this.conexao.getConnection()
45             .prepareStatement(sqlQuery);
46         stmt.setString(1, chamado.getAssunto());
47         stmt.setString(2, chamado.getMensagem());
48         stmt.setString(3, chamado.getStatus().toString());
49         stmt.setLong(4, chamado.getId());
50         linhasAfetadas = stmt.executeUpdate();
51         this.conexao.commit();
52     } catch (SQLException e) {
53         this.conexao.rollback();
54         throw e;
55     }
56     return linhasAfetadas;
57 }
```



No método **excluir** vamos retorna um valor do tipo int, como parâmetro passamos o valor do **id**, declaramos a string query, tentamos preparar o statement na conexão pegando a conexão, setamos o valor do id para o curinga do statement e pedimos para dar o commit, CASO ocorra exceção, vamos desfazer a operação na conexão com o método **rollback** e lança a exceção, no final o método retornará o valor de **linhas afetadas**

```
59 public int excluir(long id) throws SQLException{  
60     int linhasAfetadas = 0;  
61     String sqlQuery = "DELETE FROM chamado WHERE id=?";  
62     try {  
63         PreparedStatement stmt = this.conexao.getConnection()  
64             .prepareStatement(sqlQuery);  
65         stmt.setLong(1, id);  
66         linhasAfetadas = stmt.executeUpdate();  
67         this.conexao.commit();  
68     } catch (SQLException e) {  
69         this.conexao.rollback();  
70         throw e;  
71     }  
72     return linhasAfetadas;  
73 }
```

No método **listar** retornamos um objeto do tipo **Chamado**, passamos como parâmetro o valor do **id**, declaramos a string **query**, tentamos preparar o **statement** nesta conexão pegando a conexão, setamos o valor do **id**, **executamos** o **statement** e o resultado guardamos no objeto **resultSet rs**, SE encontrar algum **registro** retorna os valores desse registro através do método **parser**, CASO ocorra exceção lança a exceção, no final o método retornará o valor **null**

O método **parser** retornará os valores das colunas dentro de um objeto chamado **c**

```
76 public Chamado listar(long id) throws SQLException{
77     String sqlQuery = "SELECT * FROM chamado WHERE id=?";
78     try {
79         PreparedStatement stmt = this.conexao.getConnection()
80             .prepareStatement(sqlQuery);
81         stmt.setLong(1, id);
82         ResultSet rs = stmt.executeQuery();
83         if(rs.next()){
84             return parser(rs);
85         }
86     } catch (SQLException e) {
87         throw e;
88     }
89     return null;
90 }
```

```
92 private Chamado parser(ResultSet resultSet) throws SQLException{
93     Chamado c = new Chamado();
94     c.setId(resultSet.getLong("id"));
95     c.setAssunto(resultSet.getString("assunto"));
96     c.setMensagem(resultSet.getString("mensagem"));
97     c.setStatus(Status.valueOf(resultSet.getString("status")));
98     return c;
}
```

Método **privado** só poderá ser utilizado dentro da própria classe em que ele foi declarado e implementado

No método **listarTudo** retornamos uma lista de objetos do tipo **Chamado**, declaramos a string **query**, tentamos preparar o **statement** nesta conexão pegando a conexão, **executamos** o **statement** e o resultado guardamos no objeto **resultSet rs**, ENQUANTO encontrar um próximo **registro** retorna os valores desse registro através do método **parser** adiciona na lista e retorna a lista de chamados, CASO ocorra exceção lança a exceção

O método **parser** retornará os valores das colunas dentro de um objeto chamado **c**

```
103 public List<Chamado> listarTudo() throws SQLException{
104     String sqlQuery = "SELECT * FROM chamado ORDER BY id DESC";
105     try {
106         PreparedStatement stmt = this.conexao.getConnection()
107             .prepareStatement(sqlQuery);
108         ResultSet rs = stmt.executeQuery();
109         List<Chamado> chamados = new ArrayList<>();
110         while(rs.next()){
111             chamados.add(parser(rs));
112         }
113         return chamados;
114     } catch (SQLException e) {
115         throw e;
116     }
117 }
```

**Dúvidas?**

**Professor Anderson Henrique**





## Para a próxima aula

01 – Otimização da nossa Aplicação

02 – Finalizando Aplicação Completa com JavaEE 6 – (FULLSTACK)

Professor Anderson Henrique

