

# Banco de Dados com JDBC

## 1 – Motivação

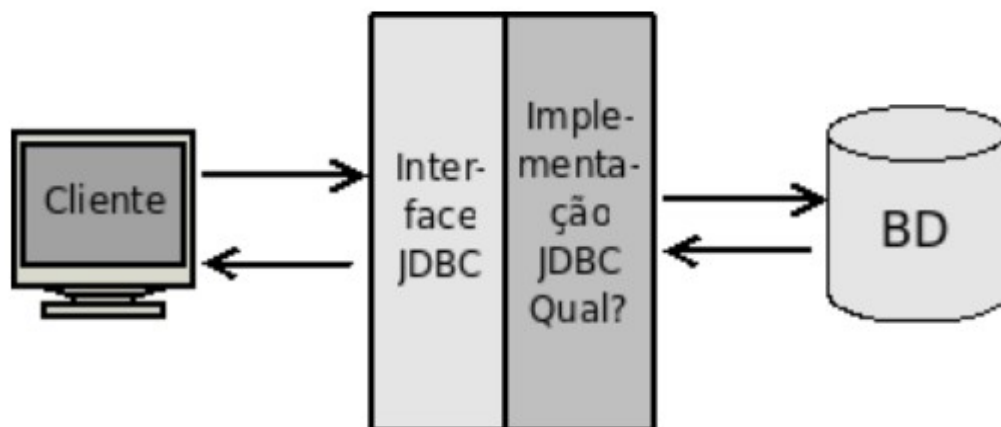
Muitos sistemas precisam manter as informações com as quais eles trabalham para permitir consultas futuras, geração de relatórios ou possíveis alterações nas informações. Para que esses dados sejam mantidos para sempre, esses sistemas geralmente guardam essas informações em um banco de dados, que as mantém de forma organizada e prontas para consultas.

A maioria dos bancos de dados comerciais são os chamados relacionais, que é uma forma de trabalhar e pensar diferente ao paradigma orientado a objetos.

O processo de armazenamento de dados é também chamado de persistência. A biblioteca de persistência em banco de dados relacionais do Java é chamada JDBC (Java Database Connectivity), e também existem diversas ferramentas do tipo ORM (Object Relational Mapping) que facilitem bastante o uso do JDBC.

## 2 – Conexão em Java

A conexão a um banco de dados é feita de maneira elegante com Java. Para evitar que cada banco tenha a sua própria API e um conjunto de classes e métodos, têm um único conjunto de interfaces muito bem definidas que devem ser implementadas. Esse conjunto de interfaces fica dentro de um pacote `java.sql` e nos referiremos a ele como JDBC.



Entre as diversas interfaces deste pacote, existe a interface Connection, que define métodos para executar uma query (como um insert e select), comitar a transação, fechar a conexão, entre outros. Se precisarmos trabalhar com MySQL ou Postgres, precisamos de classes concretas que implementem essas interfaces do pacote java.sql.

Esse conjunto de classes concretas, fará a ponte entre o código cliente que usa a API JDBC e o banco de dados. São estas classes que sabem se comunicar através do protocolo proprietário do banco de dados. Esse conjunto de classes recebe o nome de driver. Todos os principais banco de dados do mercado possuem drivers JDBC para que você possa utilizá-los com Java. O nome do driver é análogo ao que usamos para impressoras: como é impossível que um sistema operacional saiba conversar com todo tipo de impressora existente, precisamos de um driver que faça o papel de “tradutor” dessa conversa;

Para demonstrar a utilização de JDBC, iremos criar o projeto Alunos utilizando MAVEN, este projeto utilizará o banco de dados POSTGRES.

### 3 – Banco de Dados

Acessar o pgAdmin do Postgres e criar o database BethaCodeAlunos.

Neste banco de dados iremos criar a tabela de alunos com a seguinte estrutura:

```
CREATE TABLE aluno (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(80) NOT NULL,
    idade INTEGER NOT NULL,
    cidade CHARACTER(30) NOT NULL,
    estado CHARACTER(2) NOT NULL
);
```

### 4 – Projeto Java

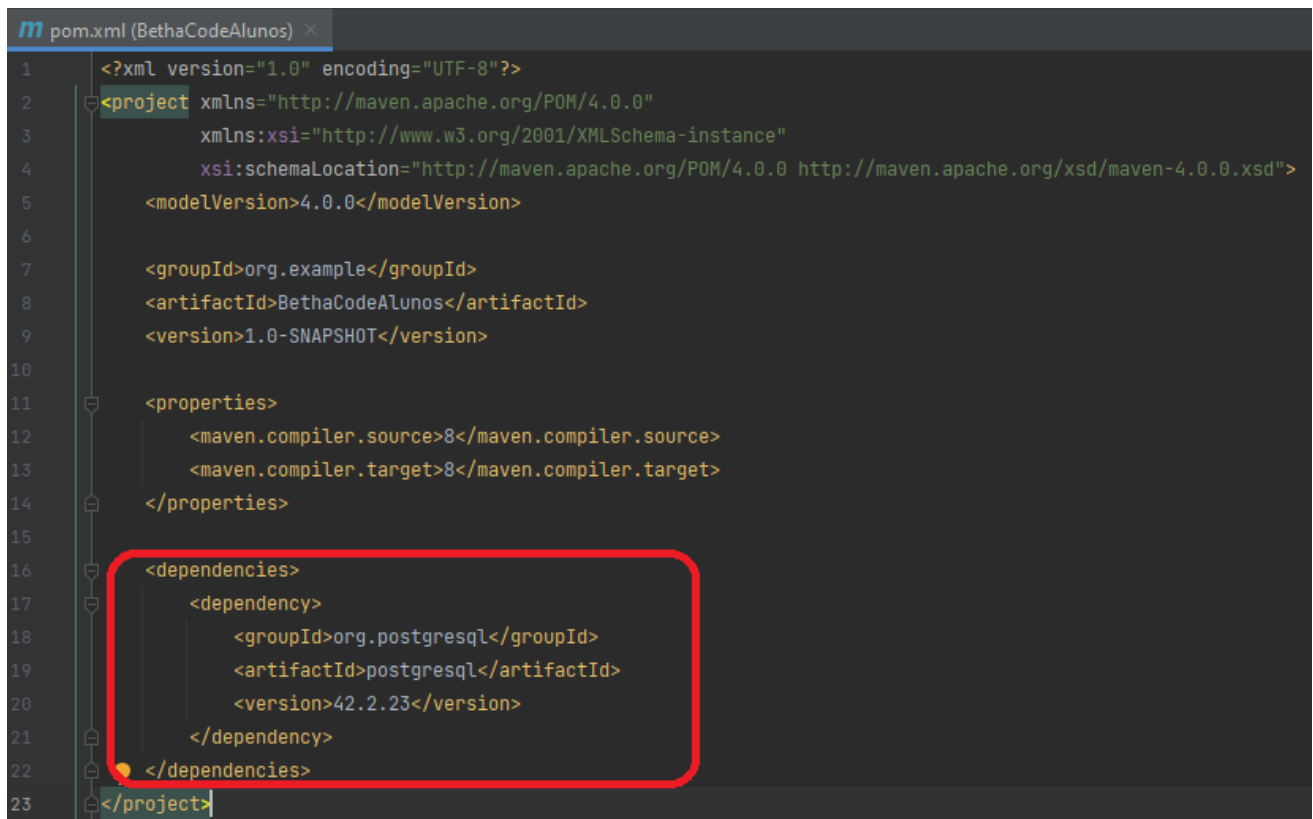
Iremos criar o projetos AlunosJDBC utilizando o MAVEN.

#### 4.1 – Dependências

Para a execução deste projeto, precisaremos utilizar o jar do PostGres para realizar a conexão e iteração com o nosso banco de dados.

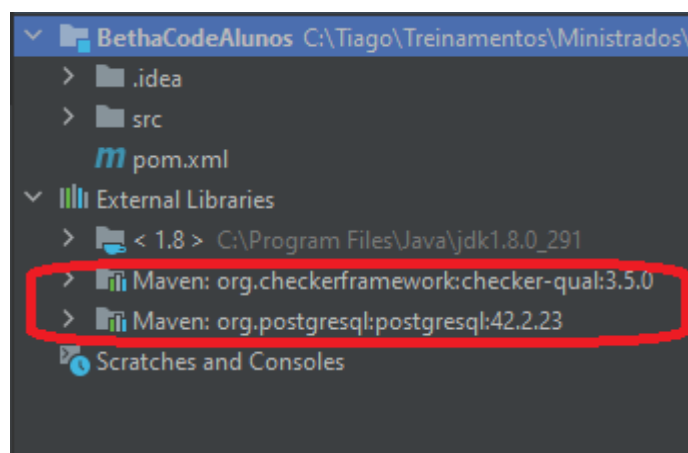
Em projetos realizados pelo MAVEN, temos o arquivo POM.xml. Neste arquivo temos as devidas configurações do projeto e as dependências utilizadas.

Precisaremos alterar o arquivo POM.xml, adicionando como dependência o jar do Postgres.



```
m pom.xml (BethaCodeAlunos) x
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>org.example</groupId>
8      <artifactId>BethaCodeAlunos</artifactId>
9      <version>1.0-SNAPSHOT</version>
10
11     <properties>
12         <maven.compiler.source>8</maven.compiler.source>
13         <maven.compiler.target>8</maven.compiler.target>
14     </properties>
15
16     <dependencies>
17         <dependency>
18             <groupId>org.postgresql</groupId>
19             <artifactId>postgresql</artifactId>
20             <version>42.2.23</version>
21         </dependency>
22     </dependencies>
23 </project>
```

Desta forma será baixado automaticamente o jar referente ao PostGRES.

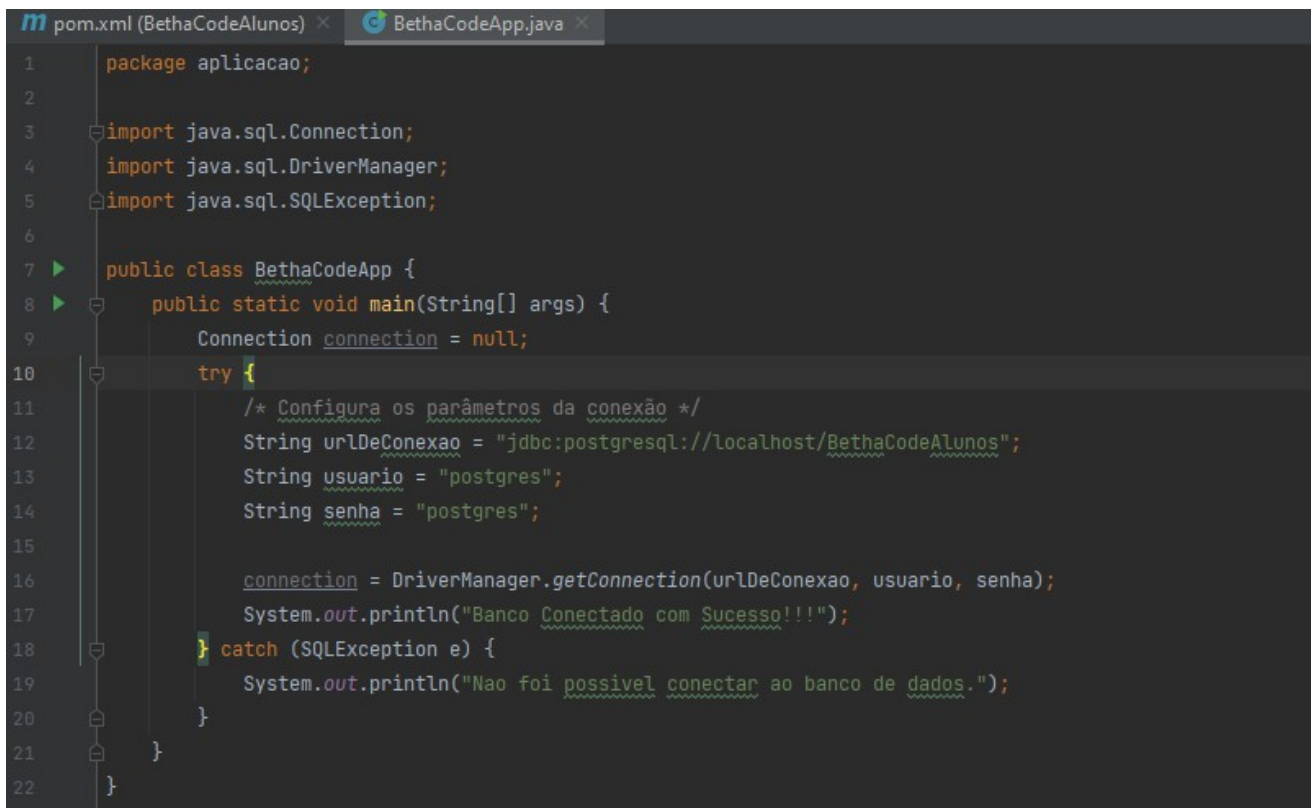


## 4.2 – Conexão com Banco

Nesta etapa iremos realizar a conexão com o banco de dados.

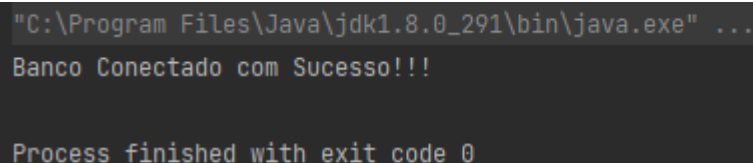
Para isso precisaremos informar o banco de dados, usuário e senha. No exemplo estamos fazendo a conexão diretamente no método main, são adicionadas as informações de url de conexão, usuário e senha.

Sendo utilizado o DriverManager para gerar uma conexão com o Banco de Dados. Pode ser verificado que para utilização do método getConnection é necessária a implementação do SQLException.



```
1 package aplicacao;
2
3 import java.sql.Connection;
4 import java.sql.DriverManager;
5 import java.sql.SQLException;
6
7 public class BetaCodeApp {
8     public static void main(String[] args) {
9         Connection connection = null;
10        try {
11            /* Configura os parâmetros da conexão */
12            String urlDeConexao = "jdbc:postgresql://localhost/BethaCodeAlunos";
13            String usuario = "postgres";
14            String senha = "postgres";
15
16            connection = DriverManager.getConnection(urlDeConexao, usuario, senha);
17            System.out.println("Banco Conectado com Sucesso!!!");
18        } catch (SQLException e) {
19            System.out.println("Nao foi possivel conectar ao banco de dados.");
20        }
21    }
22 }
```

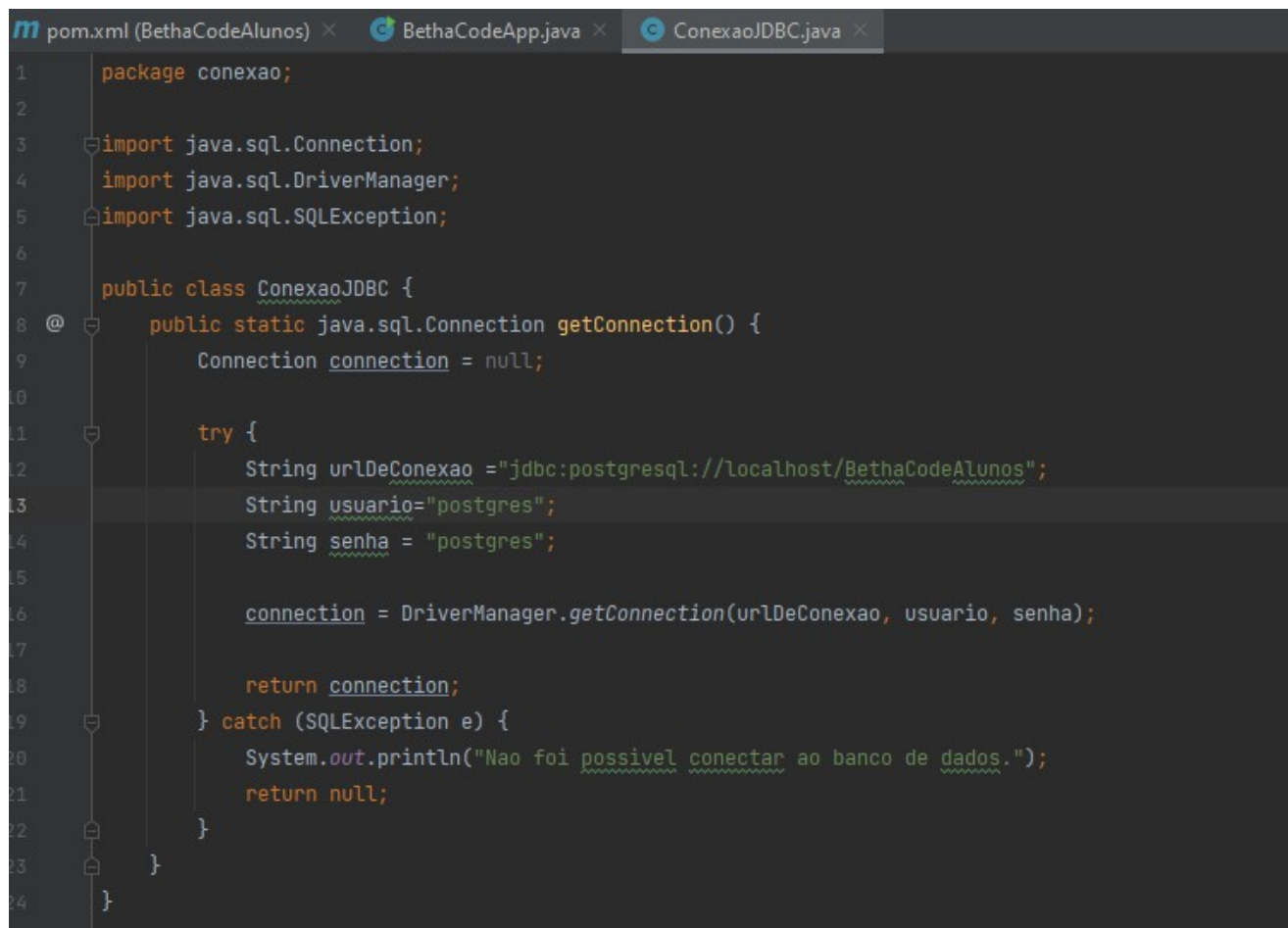
Se informado os dados corretamente na URL de conexão e as informações de senha e usuário teremos como retorno:



```
"C:\Program Files\Java\jdk1.8.0_291\bin\java.exe" ...
Banco Conectado com Sucesso!!!

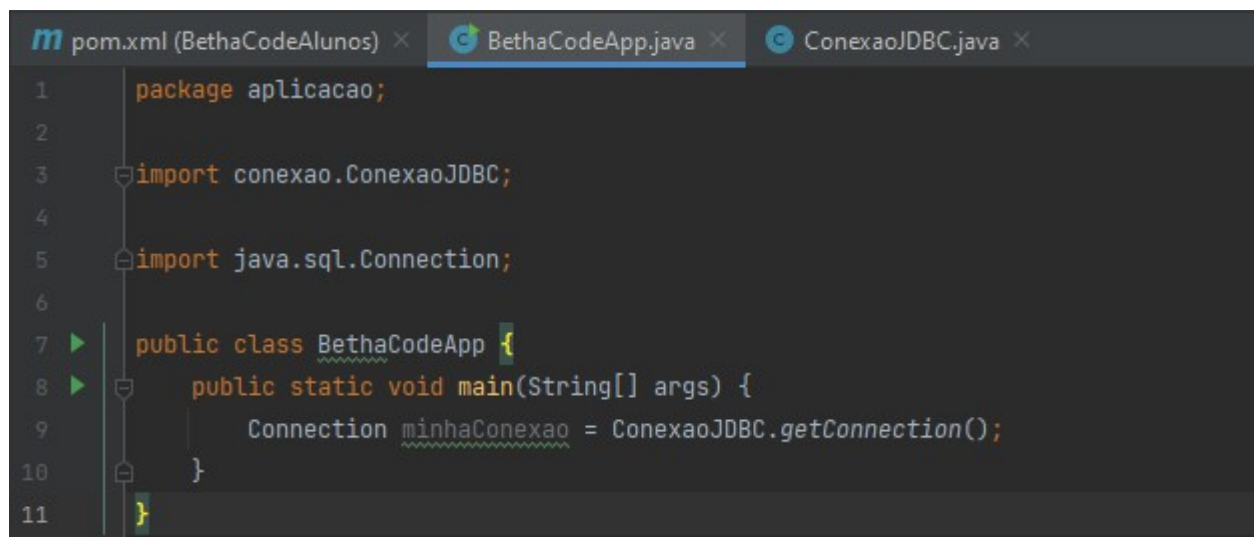
Process finished with exit code 0
```

Porém para realizar a conexão, iremos criar a classe `ConexaoJDBC` que vai retornar uma `connection` para nossa aplicação.



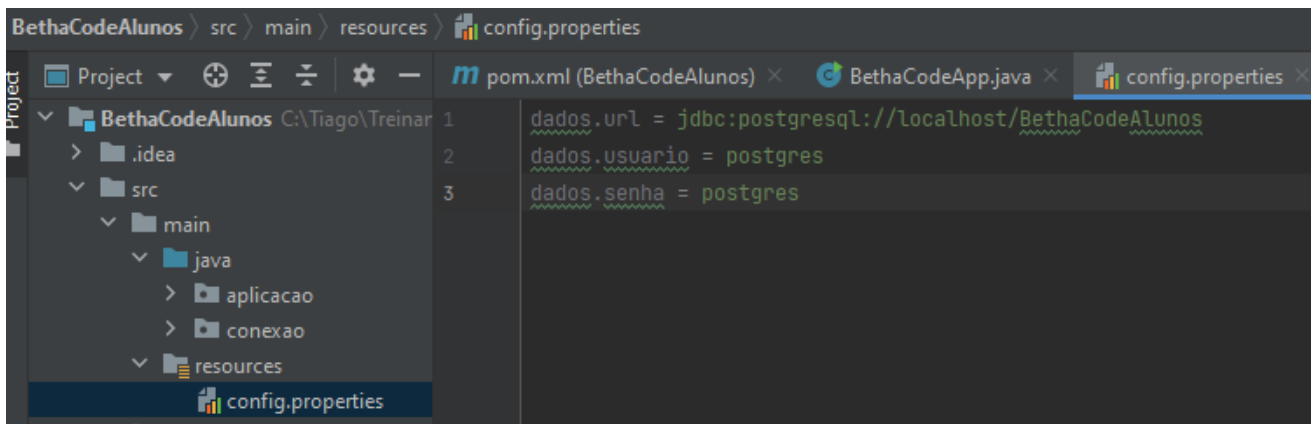
```
1 package conexao;
2
3 import java.sql.Connection;
4 import java.sql.DriverManager;
5 import java.sql.SQLException;
6
7 public class ConexaoJDBC {
8     @ public static java.sql.Connection getConnection() {
9         Connection connection = null;
10
11         try {
12             String urlDeConexao = "jdbc:postgresql://localhost/BethaCodeAlunos";
13             String usuario = "postgres";
14             String senha = "postgres";
15
16             connection = DriverManager.getConnection(urlDeConexao, usuario, senha);
17
18             return connection;
19         } catch (SQLException e) {
20             System.out.println("Nao foi possivel conectar ao banco de dados.");
21             return null;
22         }
23     }
24 }
```

Desta forma no main de nossa aplicação, precisaremos criar uma instância de conexão com o banco e utilizarmos o nosso método.



```
1 package aplicacao;
2
3 import conexao.ConexaoJDBC;
4
5 import java.sql.Connection;
6
7 public class BetaCodeApp {
8     public static void main(String[] args) {
9         Connection minhaConexao = ConexaoJDBC.getConnection();
10     }
11 }
```

Para finalizar nossa conexão, iremos adicionar um arquivo de configuração com as informações da conexão, desta forma os dados de usuário e senha não estarão expostos diretamente no código de conexão. Em resources, vamos criar o arquivo config.properties com as informações de conexão.



Na classe que gera a Conexão, iremos precisar criar um método para carregar a informação do arquivo config.properties criado.

```
public class ConexaoJDBC {

    public static Properties getProp() throws IOException {
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        Properties props = new Properties();
        InputStream file = loader.getResourceAsStream("config.properties");
        props.load(file);
        return props;
    }
}
```

No método com a conexão, iremos alterar para utilizar nossa nova função.

```
public static java.sql.Connection getConnection() {
    Connection connection = null;

    try {
        Properties prop = getProp();
        String urlDeConexao = prop.getProperty("dados.url");
        String usuario = prop.getProperty("dados.usuario");
        String senha = prop.getProperty("dados.senha");

        connection = DriverManager.getConnection(urlDeConexao, usuario, senha);

        return connection;
    } catch (SQLException | IOException e) {
        System.out.println("Nao foi possivel conectar ao banco de dados.");
        return null;
    }
}
```

### 4.3 – Modelo

Com nossa conexão pronta, precisamos criar o modelo, será uma classe que representa um objeto de Aluno.

Neste caso nossa classe de modelo tem apenas os atributos de acordo com a definição da tabela, além do construtor adicionado a classe (vazio e com os atributos, menos ID) e o sobreposição do toString com as informações do Aluno, também será necessário adicionar os Getters e Setters dos atributos.

```
public class Aluno {  
  
    private Integer id;  
    private String nome;  
    private Integer idade;  
    private String cidade;  
    private String estado;  
  
    public Aluno(String nome, Integer idade, String cidade, String estado {  
        this.nome = nome;  
        this.idade = idade;  
        this.cidade = cidade;  
        this.estado = estado;  
    }  
  
    public Integer getId() {  
        return id;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
}
```

Agora com o modelo definido, precisamos realizar as operações de CRUD (Insert / Update / Delete / Select), para isso será utilizado o conceito de DAO. Teremos a classe AlunosDAO e neste classe teremos as operações para criar / atualizar / selecionar ou excluir um aluno em nosso banco de dados.

#### 4.4 – DAO

Iremos criar a classe AlunoDAO, neste teremos o atributo conexão e quando ser instanciada nossa classe no construtor, iremos utilizar o nosso método para buscar a conexão com o banco de dados.

```
1 package dao;
2
3 import conexao.ConexaoJDBC;
4 import java.sql.Connection;
5
6 public class AlunoDAO {
7     private Connection minhaConexao;
8
9     public AlunoDAO(){
10         this.minhaConexao = ConexaoJDBC.getConnection();
11     }
12 }
13
```

Tendo a nossa conexão em AlunoDAO, iremos criar o método para salvar um novo Aluno no Banco de Dados. Iremos criar o método criarAluno. Neste método utilizamos o PreparedStatement adicionando o sql. Sendo utilizado o executeUpdate para aplicar o comando no banco de dados.

```
public void criarAluno(Aluno novoAluno){
    try{
        String meuSql = "insert into aluno (nome, idade, cidade, estado) values (?, ?, ?, ?)";

        PreparedStatement preparedStatement = minhaConexao.prepareStatement(meuSql);
        preparedStatement.setString(1, novoAluno.getNome());
        preparedStatement.setInt(2, novoAluno.getIdade());
        preparedStatement.setString(3, novoAluno.getCidade());
        preparedStatement.setString(4, novoAluno.getEstado());

        int linhaInserida = preparedStatement.executeUpdate();
        System.out.println("Adicionado novo aluno! Inserida " + linhaInserida + " linha(s).");
    }catch (SQLException e){
        System.out.println("Gerado erro na inserção de dados..");
        System.out.println(e.getMessage());
    }
}
```



Então no método main, precisamos criar um Aluno, adicionar as suas devidas informações e executar o método criarAluno para gravar o mesmo no banco de dados.

```
package aplicacao;

import dao.AlunoDAO;
import model.Aluno;

public class BetaCodeApp {
    public static void main(String[] args) {
        AlunoDAO alunoDAO = new AlunoDAO();

        //Criando um novo Aluno
        Aluno novo = new Aluno( nome: "Tiago Valério", idade: 40, cidade: "Içara", estado: "SC");
        alunoDAO.criarAluno(novo);
    }
}
```

Desta forma ao executar teremos como resultado, que foi criado um novo aluno no banco de dados.

```
"C:\Program Files\Java\jdk1.8.0_291\bin\java.exe" ...
Adicionado novo aluno! Inserida 1 linha(s).

Process finished with exit code 0
```

1

select \* from aluno

Data Output

Explain

Messages

Notifications

	id [PK] integer	nome character varying (80)	idade integer	cidade character (30)	estado character (2)
1	1	Tiago Valério	40	Içara	SC

Então, como conseguimos realizar a inserção de novos alunos em nosso banco de dados, iremos criar o método para buscar todos estes alunos que estão cadastrados em nosso banco de dados.

Iremos criar o método buscarAlunos, este irá selecionar todos os alunos cadastrados e vai retornar uma Lista de Alunos.

Para selecionar informações também utilizamos a PreparedStatement, a grande diferença é a utilização do método executeQuery e não o executeUpdate. Como podemos ter vários registros no retorno de ResultSet, precisamos utilizar o while para consumir o retorno, enquanto existir informações de alunos cadastrados.

```
public List<Aluno> buscarAlunos(){
    List<Aluno> alunosCadastrados = new ArrayList<>();
    try{
        String meuSql = "select * from aluno";
        PreparedStatement preparedStatement = minhaConexao.prepareStatement(meuSql);
        ResultSet resultadoSql = preparedStatement.executeQuery();
        while (resultadoSql.next()){
            Aluno cadastrado = new Aluno();
            cadastrado.setId(resultadoSql.getInt( columnLabel: "id"));
            cadastrado.setNome(resultadoSql.getString( columnLabel: "nome"));
            cadastrado.setIdade(resultadoSql.getInt( columnLabel: "idade"));
            cadastrado.setCidade(resultadoSql.getString( columnLabel: "cidade"));
            cadastrado.setEstado(resultadoSql.getString( columnLabel: "estado"));
            alunosCadastrados.add(cadastrado);
        }
    }catch (SQLException e){
        System.out.println("Gerado erro na busca de dados..");
        System.out.println(e.getMessage());
    }
    return alunosCadastrados;
}
```

Na aplicação precisamos criar uma Lista de Alunos e executar o comando buscarAlunos, populando a mesma.

```
public class BetaCodeApp {
    public static void main(String[] args) {
        AlunoDAO alunoDAO = new AlunoDAO();

        //Criando um novo Aluno
        /*Aluno novo = new Aluno("Benício Fernandes", 7, "Içara", "SC");
        alunoDAO.criarAluno(novo);*/

        //Buscar alunos cadastrados
        List<Aluno> alunosCadastrados = alunoDAO.buscarAlunos();
        alunosCadastrados.forEach(aluno -> System.out.println(aluno));
    }
}
```

Funcionando o processo teremos como o retorno, os alunos cadastrados.

```
"C:\Program Files\Java\jdk1.8.0_291\bin\java.exe" ...
Aluno{id=1, nome='Tiago Valério', idade=40, cidade='Içara', estado='SC'}
Aluno{id=2, nome='Benício Fernandes', idade=7, cidade='Içara', estado='SC'}
```

Nosso próximo método, vai buscar o aluno conforme o código dele, e teremos como retorno a informação do Aluno.

```
public Aluno buscarAlunoPorId(Integer id){
    Aluno alunoCadastrado = new Aluno();
    try{
        String meuSql = "select * from aluno where id = ?";
        PreparedStatement preparedStatement = minhaConexao.prepareStatement(meuSql);
        preparedStatement.setInt( parameterIndex: 1, id);
        ResultSet resultado = preparedStatement.executeQuery();
        if(resultado.next()){
            alunoCadastrado.setId(resultado.getInt( columnLabel: "id"));
            alunoCadastrado.setNome(resultado.getString( columnLabel: "nome"));
            alunoCadastrado.setIdade(resultado.getInt( columnLabel: "idade"));
            alunoCadastrado.setCidade(resultado.getString( columnLabel: "cidade"));
            alunoCadastrado.setEstado(resultado.getString( columnLabel: "estado"));
        }
    }catch (SQLException e){
        System.out.println("Gerado erro na busca de dados..");
        System.out.println(e.getMessage());
    }
    return alunoCadastrado;
}
```

Na aplicação precisaremos criar um objeto de aluno e utilizar o buscarAlunoPorId.

```
public class BetaCodeApp {
    public static void main(String[] args) {
        AlunoDAO alunoDAO = new AlunoDAO();

        //Buscar aluno por Id
        Aluno alunoCadastrado = alunoDAO.buscarAlunoPorId(1);
        System.out.println(alunoCadastrado);
    }
}
```

Funcionando nossa busca por id, teremos como retorno.

```
Aluno{id=1, nome='Tiago Valério', idade=40, cidade='Içara', estado='SC'}

Process finished with exit code 0
```

Então iremos implementar nosso método para atualizarAluno, neste método iremos passar um aluno para o método e iremos realizar a atualização no banco de dados.

```
public void atualizarAluno(Aluno alunoAtualizar){
    try{
        String meuSql = "update aluno set nome = ?, idade = ?, cidade = ?, estado = ? where id = ? ";
        PreparedStatement preparedStatement = minhaConexao.prepareStatement(meuSql);
        preparedStatement.setString( parameterIndex: 1, alunoAtualizar.getNome());
        preparedStatement.setInt( parameterIndex: 2, alunoAtualizar.getIdade());
        preparedStatement.setString( parameterIndex: 3, alunoAtualizar.getCidade());
        preparedStatement.setString( parameterIndex: 4, alunoAtualizar.getEstado());
        preparedStatement.setInt( parameterIndex: 5, alunoAtualizar.getId());

        int atualizados = preparedStatement.executeUpdate();

        System.out.println("Aluno: " + alunoAtualizar.getId() + " atualizado com sucesso!!!");
    }catch(SQLException e){
        System.out.println("Gerado erro na atualização de dados..");
        System.out.println(e.getMessage());
    }
}
```

Com este método criado, precisamos buscar um aluno cadastrado em nosso banco de dados, atualizar as informações do aluno e passar para o método atualizarAluno.

```
public class BetaCodeApp {
    public static void main(String[] args) {
        AlunoDAO alunoDAO = new AlunoDAO();

        //Busca aluno por ID, altera a cidade para Criciúma e atualiza no banco
        Aluno alunoCadastrado = alunoDAO.buscarAlunoPorId(2);
        alunoCadastrado.setCidade("Criciúma");
        alunoDAO.atualizarAluno(alunoCadastrado);
    }
}
```

Funcionando o processo teremos como retorno:

```
"C:\Program Files\Java\jdk1.8.0_291\bin\java.exe" ...
Aluno: 2 atualizado com sucesso!!!

Process finished with exit code 0
```

Para finalizar nosso CRUD, iremos criar o método `excluirAluno`. Este vai receber um aluno e realizar a sua exclusão.

```
public void excluirAluno(Aluno alunoAtualizar){
    try{
        String meuSql = "delete from aluno where id = ? ";
        PreparedStatement preparedStatement = minhaConexao.prepareStatement(meuSql);
        preparedStatement.setInt( parameterIndex: 1, alunoAtualizar.getId());

        int excluidos = preparedStatement.executeUpdate();
        System.out.println("Aluno: " + alunoAtualizar.getId() + " excluido com sucesso!!!");
    }catch(SQLException e){
        System.out.println("Gerado erro na atualização de dados..");
        System.out.println(e.getMessage());
    }
}
```

Da mesma forma que na atualização, precisamos buscar o aluno e depois realizar a sua exclusão na aplicação.

```
public class BetaCodeApp {
    public static void main(String[] args) {
        AlunoDAO alunoDAO = new AlunoDAO();

        //Busca aluno por ID, e exclui no banco
        Aluno alunoCadastrado = alunoDAO.buscarAlunoPorId(2);
        alunoDAO.excluirAluno(alunoCadastrado);
    }
}
```

Funcionando o processo teremos como retorno.

```
"C:\Program Files\Java\jdk1.8.0_291\bin\java.exe" ...
Aluno: 2 excluido com sucesso!!!

Process finished with exit code 0
```

## 5 – Material Complementar

<https://www.youtube.com/watch?v=P-psR8L9zUI>

<https://www.devmedia.com.br/aprendendo-java-com-jdbc/29116>

<http://www.universidadejava.com.br/java/java-jdbc/>

## 6 – Exercício

Desenvolva o mesmo processo de CRUD, para os cadastros de disciplinas e cursos.

A tabela de curso deve ter a estrutura.

### CURSO

id	serial
descricao	char(60),
numero_creditos	inteiro,
coordenador	char(60)

### Disciplina

id	serial,
descricao	char(60),
numero_creditos	integer,
id_curso	integer