Banco de Dados
com Java · JDBC

Aula 05 · Connection, Statement, PreparedStatement e ResultSet

Práticas de POO

Ciência da Computação · Graduação

`java.sql · DriverManager · PostgreSQL · CRUD · SQL Injection`

AULA 05

O que você vai aprender hoje

01
O que é JDBC
Entender a API Java para conexão com bancos de dados SQL.

02
Arquitetura e drivers
Conhecer o DriverManager e os 4 tipos de drivers JDBC.

03
Conexão com PostgreSQL
Configurar o driver e estabelecer a conexão com o banco.

04
As 4 classes essenciais
Dominar Connection, Statement, PreparedStatement e ResultSet.

05
CRUD completo
Executar SELECT, INSERT, UPDATE e DELETE via Java.

06
Segurança e boas práticas
Prevenir SQL Injection e gerenciar recursos corretamente.

FUNDAMENTOS

O que é JDBC?

- **JDBC** (Java Database Connectivity) é a API do Java que permite a comunicação com bancos de dados relacionais padrão SQL, por meio do pacote `java.sql`.

O QUE PERMITE

- Conectar a bancos de dados
- Executar comandos SQL
- Consultar e manipular dados
- Transações e procedimentos

BANCOS SUPORTADOS

- PostgreSQL
- MySQL / MariaDB
- Oracle e SQL Server
- Firebird, SQLite e outros

FUNDAMENTOS

Arquitetura do JDBC

COMO FUNCIONA

A aplicação chama a API JDBC, que usa o DriverManager para localizar o driver adequado.

O driver é o responsável por traduzir as chamadas Java para o protocolo específico do banco de dados.

FUNDAMENTOS

Os 4 tipos de drivers JDBC

Tipo 1 — JDBC-ODBC Bridge

Usa uma ponte ODBC. Lento e dependente do SO. Em desuso.

Tipo 2 — Native API

Usa bibliotecas nativas do banco. Requer instalação local.

Tipo 3 — Network Protocol

Comunica via servidor intermediário. Flexível, menos comum.

Tipo 4 — Pure Java (Thin)

100% Java, fala direto com o banco. O mais usado hoje.

O driver do PostgreSQL é do Tipo 4 (Pure Java) — basta adicionar o arquivo JAR ao projeto.

FUNDAMENTOS

Os 6 passos de uma conexão JDBC

1 Carregar o driver

```
Class.forName()
```

2 Abrir conexão

```
DriverManager.getConnection()
```

3 Criar statement

```
createStatement()
```

4 Executar SQL

```
executeQuery() /
executeUpdate()
```

5 Processar resultado

```
ResultSet.next()
```

6 Fechar recursos

```
close()
```

CONFIGURAÇÃO

Configurando o driver do PostgreSQL

- Para conectar ao PostgreSQL, é preciso adicionar o **driver JDBC** (arquivo JAR) ao projeto.
- Ele já vem instalado com o JDK? Não — precisa ser baixado separadamente.

DEPENDÊNCIA MAVEN

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.7.3</version>
</dependency>
```

SEM MAVEN (JAR MANUAL)

- Baixe em jdbc.postgresql.org
- Adicione o .jar ao build path
- Classe: org.postgresql.Driver

```
String DRIVER =
 "org.postgresql.Driver";
```

CONEXÃO

DriverManager e a URL de conexão

- O **`DriverManager`** é a classe que gerencia os drivers disponíveis e estabelece a conexão física com o banco, usando o método **`getConnection()`**.

Estabelecendo a conexão

```
01  // getConnection(url, usuario, senha)
02  Connection conexao = DriverManager.getConnection(
03          url, usuario, senha);
```

A URL de conexão

identifica o banco, o servidor e a porta.

Cada SGBD tem seu próprio formato de URL.

CONEXÃO

A URL de conexão do PostgreSQL

Formato geral da URL

```
jdbc:postgresql://servidor:porta/nome_do_banco
```

`jdbc:postgresql://`	Prefixo que identifica o protocolo e o SGBD
`servidor`	Endereço do banco (localhost, IP ou domínio)
`porta`	Porta de acesso — padrão do PostgreSQL é 5432
`nome_do_banco`	Nome do database a ser acessado

```
Exemplo:  jdbc:postgresql://localhost:5432/escola
```

FUNDAMENTOS

Tratamento de exceções no JDBC

- Operações de banco de dados podem falhar.
- O JDBC sinaliza erros por meio de **exceções verificadas**, que devem ser tratadas obrigatoriamente.

PRINCIPAIS EXCEÇÕES

- SQLException — erros de SQL/conexão
- ClassNotFoundException — driver não encontrado

ESTRUTURA TRY/CATCH

```
try {
    // conexão e SQL
} catch (SQLException e) {
    e.printStackTrace();
}
```

AS 4 CLASSES

Connection: a conexão com o banco

- A classe **`Connection`** representa uma conexão física com o banco de dados.
- É por meio dela que você cria statements, gerencia transações e controla o commit/rollback.

RESPONSABILIDADES

- Representa a sessão com o banco
- Cria objetos Statement
- Controla transações (commit/rollback)
- Define o auto-commit

CRIAÇÃO

```
Connection conn = DriverManager
    .getConnection(url, user, pass);
```

Sempre feche a conexão quando terminar — use try-with-resources ou close() no finally.

AS 4 CLASSES

Connection: métodos essenciais

`createStatement()`

Retorna um Statement para SQL simples

`prepareStatement(sql)`

Retorna um PreparedStatement parametrizado

`setAutoCommit(false)`

Desativa auto-commit para controle de transação

`commit()`

Confirma as operações da transação

`rollback()`

Desfaz as operações da transação

`close()`

Fecha a conexão e libera recursos

Boa prática: use try-with-resources para garantir o fechamento automático da conexão.

EXEMPLO

Conectando ao PostgreSQL com try-with-resources

ConexaoPostgres.java

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

public class ConexaoPostgres {

    public static void main(String[] args) {
        String url = "jdbc:postgresql://localhost:5432/escola";
        String usuario = "postgres";
        String senha = "minhaSenha123";

```

EXEMPLO

Conectando ao PostgreSQL com try-with-resources

ConexaoPostgres.java

```
11
12          // try-with-resources garante o fechamento automático
13          try (Connection conn = DriverManager.getConnection(url, usuario, senha)) {
14              System.out.println("Conectado com sucesso!");
15              System.out.println("Banco: " + conn.getCatalog());
16          } catch (SQLException e) {
17              System.err.println("Erro na conexão: " + e.getMessage());
18              e.printStackTrace();
19          }
20      }
21  }
```

AS 4 CLASSES

Statement: executando SQL simples

- A classe **`Statement`** é usada para executar comandos SQL estáticos.
- Ela é criada a partir de uma Connection e permite executar SELECT, INSERT, UPDATE e DELETE.

QUANDO USAR

- SQL sem parâmetros dinâmicos
- Consultas fixas e conhecidas
- DDL (CREATE, DROP, ALTER)

CRIAÇÃO

```
Statement stmt = conn.createStatement();
```

MÉTODOS

- executeQuery() — SELECT
- executeUpdate() — INSERT/UPDATE/DELETE

executeQuery: executando SELECT

- O método **`executeQuery()`** é usado exclusivamente para comandos SELECT.
- Ele retorna um **`ResultSet`** com os dados consultados.

Consultando dados com Statement

```java
String sql = "SELECT id, nome, email FROM alunos WHERE ativo = true";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);

// itera sobre os resultados
while (rs.next()) {
    int id = rs.getInt("id");
    String nome = rs.getString("nome");
    String email = rs.getString("email");
    System.out.println(id + " - " + nome + " - " + email);
}
```

AS 4 CLASSES

executeUpdate: INSERT, UPDATE e DELETE

- O método **executeUpdate()** é usado para comandos que **modificam** dados.
- Ele retorna um **int** indicando quantas linhas foram afetadas.

Modificando dados com Statement

```java
// INSERT
String sqlInsert = "INSERT INTO alunos (nome, email) VALUES ('João', 'joao@email.com')";
int linhasInseridas = stmt.executeUpdate(sqlInsert);
System.out.println("Linhas inseridas: " + linhasInseridas);

// UPDATE
String sqlUpdate = "UPDATE alunos SET ativo = false WHERE id = 5";
int linhasAtualizadas = stmt.executeUpdate(sqlUpdate);

// DELETE
String sqlDelete = "DELETE FROM alunos WHERE id = 10";
int linhasDeletadas = stmt.executeUpdate(sqlDelete);
```

AS 4 CLASSES

ResultSet: navegando nos resultados

- O `ResultSet` é uma tabela virtual que armazena os resultados de uma consulta SELECT.
- Ele mantém um cursor que aponta para a linha atual.

NAVEGAÇÃO

- next() — avança para a próxima linha
- previous() — volta uma linha
- first() / last() — vai para o início/fim
- isFirst() / isLast() — verifica posição

LEITURA DE DADOS

- getInt(coluna)
- getString(coluna)
- getDouble(coluna)
- getBoolean(coluna)

AS 4 CLASSES

ResultSet: métodos de leitura

Método	Retorna	Tipo SQL
`getInt(col)`	`int`	INTEGER, INT
`getString(col)`	`String`	VARCHAR, TEXT
`getDouble(col)`	`double`	DOUBLE, FLOAT
`getBoolean(col)`	`boolean`	BOOLEAN
`getDate(col)`	`Date`	DATE
`getTimestamp(col)`	`Timestamp`	TIMESTAMP

Você pode passar o nome da coluna ou o índice (começando em 1).

AS 4 CLASSES

Mapeamento de tipos: SQL → Java

Tipo SQL (PostgreSQL)	Tipo Java	Método get
SERIAL / INTEGER	int	getInt()
BIGINT	long	getLong()
VARCHAR / TEXT	String	getString()
BOOLEAN	boolean	getBoolean()
DECIMAL / NUMERIC	BigDecimal	getBigDecimal()
DATE	Date	getDate()
TIMESTAMP	Timestamp	getTimestamp()

EXEMPLO

Consulta completa com iteração

listarAlunosAtivos() — consulta completa

```
01  public void listarAlunosAtivos() {
02      String sql = "SELECT id, nome, email, data_cadastro FROM alunos WHERE ativo = true ORDER
BY nome";
03
04      try (Connection conn = DriverManager.getConnection(url, user, pass);
05           Statement stmt = conn.createStatement();
06           ResultSet rs = stmt.executeQuery(sql)) {
07
08          System.out.println("=== Alunos Ativos ===");
```

EXEMPLO

Consulta completa com iteração

listarAlunosAtivos() — consulta completa

```
09          while (rs.next()) {
10              int id = rs.getInt("id");
11              String nome = rs.getString("nome");
12              String email = rs.getString("email");
13              Date dataCad = rs.getDate("data_cadastro");
14
15              System.out.printf("ID: %d | Nome: %s | Email: %s | Cadastro: %s%n",
16                                id, nome, email, dataCad);
17          }
18      } catch (SQLException e) {
19          e.printStackTrace();
20      }
21  }
```

PREPAREDSTATEMENT

PreparedStatement: a evolução do Statement

- O `PreparedStatement` é uma versão aprimorada do Statement.
- Ele pré-compila o SQL e aceita parâmetros dinâmicos, sendo mais seguro e eficiente.

`Statement`

- SQL concatenado em string
- Compilado a cada execução
- Vulnerável a SQL Injection
- Simples, porém limitado

`PreparedStatement`

- SQL com parâmetros (?)
- Pré-compilado pelo banco
- Imune a SQL Injection
- Reutilizável e mais rápido

Regra de ouro: sempre que houver dados do usuário no SQL, use PreparedStatement.

PREPAREDSTATEMENT

Parâmetros dinâmicos: o coringa ?

- No PreparedStatement, os valores são substituídos pelo coringa **?**.
- Os parâmetros são preenchidos na ordem em que aparecem, começando em **1**.

Preenchendo parâmetros com setXXX()

```java
// SQL com 3 parâmetros dinâmicos
String sql = "INSERT INTO alunos (nome, email, idade) VALUES (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(sql);

// preenchendo os parâmetros na ordem (índice começa em 1)
pstmt.setString(1, "Maria Silva");    // 1º ? → nome
pstmt.setString(2, "maria@email.com"); // 2º ? → email
pstmt.setInt(3, 21);                   // 3º ? → idade

int linhasAfetadas = pstmt.executeUpdate();
```

SEGURANÇA

SQL Injection: o perigo da concatenação

Métodos setXXX: preenchendo parâmetros

Exemplo de ataque — campo de login

```
// Usuário digita:  ' OR '1'='1
String sql = "SELECT * FROM usuarios WHERE nome = '" + entrada + "'";
// SQL resultante: SELECT * FROM usuarios WHERE nome = '' OR '1'='1'  ← retorna TUDO!
```

CONSEQUÊNCIAS

- Acesso não autorizado a dados sensíveis
- Exclusão ou alteração de tabelas inteiras
- Roubo de senhas e informações pessoais
- Comprometimento total do banco de dados

SEGURANÇA

PreparedStatement previne SQL Injection

- Com o PreparedStatement, os parâmetros são **enviados separadamente** do comando SQL.
- O banco trata o valor apenas como dado, nunca como código executável.

O mesmo ataque, agora neutralizado

```java
// Com PreparedStatement, o ataque é inofensivo
String sql = "SELECT * FROM usuarios WHERE nome = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, "' OR '1'='1");  // tratado como texto literal!

// O banco procura um usuário com o nome literal "' OR '1'='1" → não encontra nada
```

POR QUE FUNCIONA

- SQL é pré-compilado antes dos dados
- Parâmetros nunca viram código SQL

Além da segurança

o PreparedStatement também é mais rápido em execuções repetidas, pois o banco reutiliza o plano de execução já compilado.

PREPAREDSTATEMENT

Métodos setXXX: preenchendo parâmetros

Método	Tipo Java	Tipo SQL (PostgreSQL)
`setInt(idx, valor)`	`int`	`INTEGER, SERIAL`
`setLong(idx, valor)`	`long`	`BIGINT`
`setString(idx, valor)`	`String`	`VARCHAR, TEXT`
`setDouble(idx, valor)`	`double`	`DOUBLE PRECISION`
`setBoolean(idx, valor)`	`boolean`	`BOOLEAN`
`setDate(idx, valor)`	`Date`	`DATE`
`setTimestamp(idx, valor)`	`Timestamp`	`TIMESTAMP`

O índice sempre começa em 1 (não em 0) e segue a ordem dos ? no SQL.

CRUD

INSERT: inserindo registros

inserirAluno() — INSERT com PreparedStatement

```
01  public boolean inserirAluno(String nome, String email, int idade) {
02      String sql = "INSERT INTO alunos (nome, email, idade) VALUES (?, ?, ?)";
03
04      try (Connection conn = DriverManager.getConnection(url, user, pass);
05           PreparedStatement pstmt = conn.prepareStatement(sql)) {
06
07          // preenchendo os parâmetros na ordem dos ?
08          pstmt.setString(1, nome);
09          pstmt.setString(2, email);
10          pstmt.setInt(3, idade);
11
12          int linhas = pstmt.executeUpdate();
13
```

CRUD

INSERT: inserindo registros

inserirAluno() — INSERT com PreparedStatement

```
13
14          if (linhas > 0) {
15              System.out.println("Aluno inserido com sucesso!");
16              return true;
17          }
18          return false;
19
20      } catch (SQLException e) {
21          e.printStackTrace();
22          return false;
23      }
24  }
```

CRUD

UPDATE: alterando registros

atualizarEmail() — UPDATE com PreparedStatement

```
01  public boolean atualizarEmail(int id, String novoEmail) {
02      String sql = "UPDATE alunos SET email = ? WHERE id = ?";
03
04      try (Connection conn = DriverManager.getConnection(url, user, pass);
05           PreparedStatement pstmt = conn.prepareStatement(sql)) {
06
07          // ordem dos parâmetros segue a ordem dos ? no SQL
08          pstmt.setString(1, novoEmail);   // 1º ? → novo email
09          pstmt.setInt(2, id);             // 2º ? → id do aluno
10
11          int linhas = pstmt.executeUpdate();
12
```

CRUD

UPDATE: alterando registros

atualizarEmail() — UPDATE com PreparedStatement

```
12
13          if (linhas > 0) {
14              System.out.println("Email atualizado com sucesso!");
15              return true;
16          } else {
17              System.out.println("Nenhum aluno encontrado com esse ID.");
18              return false;
19          }
20
21      } catch (SQLException e) {
22          e.printStackTrace();
23          return false;
24      }
25  }
```

CRUD

DELETE: excluindo registros

excluirAluno() — DELETE com PreparedStatement

```
01  public boolean excluirAluno(int id) {
02      String sql = "DELETE FROM alunos WHERE id = ?";
03
04      try (Connection conn = DriverManager.getConnection(url, user, pass);
05           PreparedStatement pstmt = conn.prepareStatement(sql)) {
06
07          pstmt.setInt(1, id);   // único parâmetro: o ID a excluir
08
09          int linhas = pstmt.executeUpdate();
10
```

CRUD

DELETE: excluindo registros

excluirAluno() — DELETE com PreparedStatement

```
10
11          if (linhas > 0) {
12              System.out.println("Aluno excluído com sucesso!");
13              return true;
14          } else {
15              System.out.println("Nenhum aluno encontrado com esse ID.");
16              return false;
17          }
18
19      } catch (SQLException e) {
20          e.printStackTrace();
21          return false;
22      }
23  }
```

MÉTODOS

executeQuery vs executeUpdate

executeQuery()

Usado exclusivamente para SELECT. Retorna um ResultSet com os dados.

```
ResultSet rs = pstmt.executeQuery();
while (rs.next()) {
    // processa cada linha
}
```

executeUpdate()

Usado para INSERT, UPDATE e DELETE. Retorna um int com o nº de linhas afetadas.

```
int linhas = pstmt.executeUpdate();
if (linhas > 0) {
    // operação bem-sucedida
}
```

Usar o método errado gera SQLException em tempo de execução.

AVANÇADO

Transações: commit e rollback

- Uma **transação** agrupa várias operações SQL em uma unidade atômica: ou todas são confirmadas (commit), ou todas são desfeitas (rollback) em caso de erro.

Controlando transações manualmente

```java
try (Connection conn = DriverManager.getConnection(url, user, pass)) {
    conn.setAutoCommit(false);   // desativa o commit automático

    try {
        inserirAluno(conn, "Maria", "maria@email.com", 21);
        inserirMatricula(conn, 1, "2025-1");

        conn.commit();   // confirma todas as operações
        System.out.println("Transação confirmada!");
    } catch (SQLException e) {
        conn.rollback();   // desfaz tudo em caso de erro
        System.err.println("Transação revertida: " + e.getMessage());
    }
}
```

BOAS PRÁTICAS

ConnectionFactory: centralizando a conexão

- Em projetos reais, a conexão é centralizada em uma classe **`ConnectionFactory`**.
- Isso evita repetir as credenciais em vários lugares e facilita a manutenção.

ConnectionFactory.java

```java
public class ConnectionFactory {

    private static final String URL = "jdbc:postgresql://localhost:5432/escola";
    private static final String USUARIO = "postgres";
    private static final String SENHA = "minhaSenha123";

    // retorna uma nova conexão a cada chamada
    public static Connection getConnection() {
        try {
            return DriverManager.getConnection(URL, USUARIO, SENHA);
        } catch (SQLException e) {
            throw new RuntimeException("Erro ao conectar ao banco", e);
        }
    }
}
```

BOAS PRÁTICAS

Escrevendo código JDBC com qualidade

Use try-with-resources

Garante o fechamento automático de Connection, Statement e ResultSet.

```
try (Connection c = ...) { }
```

Prefira PreparedStatement

Sempre que houver dados do usuário, previne SQL Injection.

```
conn.prepareStatement(sql)
```

Centralize a conexão

Use uma ConnectionFactory para evitar credenciais repetidas.

```
ConnectionFactory.getConnection()
```

Feche os recursos

Conexões abertas consomem memória e travam o banco.

```
rs.close(); stmt.close();
```

ATENÇÃO

Erros comuns ao trabalhar com JDBC

Esquecer o driver no classpath

Sem o JAR do PostgreSQL, ocorre ClassNotFoundException.

Não fechar a conexão

Conexões abertas esgotam os recursos do banco.

Concatenar dados do usuário no SQL

Abre brecha para SQL Injection. Use PreparedStatement.

Usar executeQuery para INSERT

executeQuery é só para SELECT. Use executeUpdate.

Índice de parâmetro errado

Os índices do PreparedStatement começam em 1, não em 0.

Ignorar o retorno de executeUpdate

O int retornado indica se a operação afetou alguma linha.

MAPA

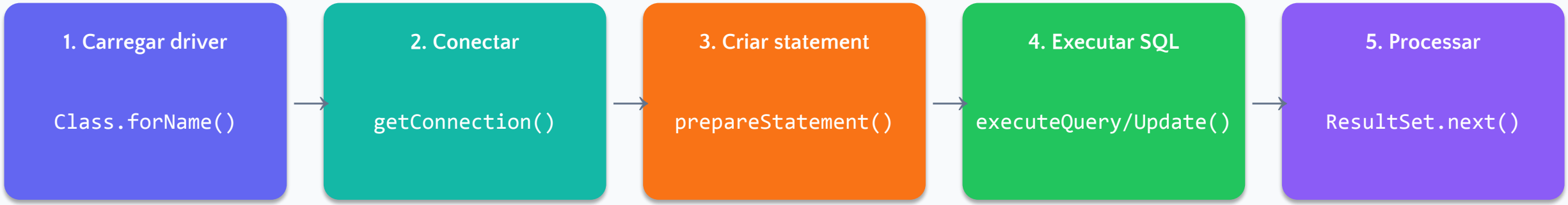
As 4 classes essenciais do JDBC

Classe	Função	Método principal
Connection	Estabelece a conexão com o banco	`createStatement()` / `prepareStatement()`
Statement	Executa SQL estático	`executeQuery()` / `executeUpdate()`
PreparedStatement	Executa SQL com parâmetros (?)	`setXXX()` + `executeQuery/Update()`
ResultSet	Armazena o resultado do SELECT	`next()` + `getXXX()`

ResultSet só é necessário para SELECT. Para INSERT/UPDATE/DELETE, basta Connection + Statement/PreparedStatement.

FLUXO

O ciclo completo de uma operação JDBC



6. Fechar recursos → sempre feche ResultSet, Statement e Connection (ou use try-with-resources)

ORDEM DE FECHAMENTO

Feche na ordem inversa da criação: primeiro o ResultSet, depois o Statement e por último a Connection.

PRÁTICA

Exercícios: aplique o que aprendeu

EXERCÍCIO 1 Fácil

Listagem de alunos

Crie um programa que conecte ao PostgreSQL (banco escola), execute um SELECT na tabela alunos usando Statement e imprima todos os registros (id, nome, email) percorrendo o ResultSet com next(). Use try-with-resources.

DICAS PARA COMEÇAR

- Crie a tabela: CREATE TABLE alunos (id SERIAL PRIMARY KEY, nome VARCHAR(100), email VARCHAR(100));
- Use a URL: jdbc:postgresql://localhost:5432/escola
- Percorra com while (rs.next()) e leia com getInt() e getString()

PRÁTICA

Exercício 2: CRUD completo

EXERCÍCIO 2 Médio

CRUD de cidades com PreparedStatement

Crie a tabela cidade (id SERIAL, nome VARCHAR, uf CHAR(2)). Implemente quatro métodos usando PreparedStatement: inserirCidade(), listarCidades(), atualizarCidade() e excluirCidade(). Cada método deve abrir e fechar a conexão com try-with-resources e verificar o retorno de executeUpdate().

REQUISITOS

- INSERT com 2 parâmetros (nome, uf)
- UPDATE com 3 parâmetros (nome, uf, id)
- DELETE com 1 parâmetro (id)
- SELECT percorrido com ResultSet

PRÁTICA

Exercício 3: desafio completo

EXERCÍCIO 3 Desafio

Mini sistema de cadastro com menu

Construa um sistema de cadastro de alunos via console (Scanner) com um menu de opções: 1-Cadastrar, 2-Listar, 3-Atualizar, 4-Excluir, 5-Sair. Use uma classe ConnectionFactory para centralizar a conexão, PreparedStatement em todas as operações e uma classe Aluno (modelo) para transportar os dados. Trate todas as SQLExceptions.

DESAFIOS EXTRAS

- Validar se o e-mail já existe antes de inserir
- Usar transação (commit/rollback) ao cadastrar aluno + matrícula
- Criar método buscarPorId(int id) que retorna um objeto Aluno

Resumo da aula

Sua aplicação agora conversa com o banco

- JDBC e a arquitetura de drivers
- Conexão com PostgreSQL
- Connection, Statement e ResultSet
- PreparedStatement e parâmetros dinâmicos
- Prevenção de SQL Injection
- CRUD completo e transações

PRÓXIMA AULA

DAO e persistência orientada a objetos

Vamos aplicar o padrão DAO (Data Access Object) para separar a lógica de acesso ao banco e integrar JDBC com a programação orientada a objetos.

```
class AlunoDAO { ... }
```

Obrigado!

Aula 05 · Conexão Java com Banco de Dados (JDBC)

Práticas de POO

Prof. Me. Célio Ricardo Castelano · Ciência da Computação