Integração

Swing + JDBC + MVC

Aula 07 · Parte 2 · Uma aplicação completa em camadas, do banco à tela

Práticas de POO Ciência da Computação · Graduação

`view` · `controller` · `dao` · `model` · `DAO` · `transação` · `PostgreSQL`

VISÃO GERAL

O que vamos construir

Um **cadastro de alunos** completo: a View chama o Controller, que usa o DAO, que persiste no PostgreSQL.

`view / AlunoView`	JFrame: campos, combo de cursos, lista e botões.
`controller / AlunoController`	Recebe dados da View e orquestra o DAO.
`dao / AlunoDAO · CursoDAO`	SQL, JDBC e a transação de matrícula.
`model / Aluno · Curso`	Entidades de dados (POJOs).

MODEL

O banco: criando as tabelas

Execute este script no PostgreSQL. Note a **FK curso_id** e o contador **total_alunos** usado na transação.

script.sql

```sql
CREATE TABLE curso (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    total_alunos INT DEFAULT 0
);
CREATE TABLE aluno (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100),
    curso_id INT REFERENCES curso(id)
);
```

MODEL

Curso.java: a entidade

O `toString()` faz o combo exibir o nome do curso automaticamente.

model/Curso.java

```java
package model;
public class Curso {
    private Integer id;
    private String nome;
    private int totalAlunos;
    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }
    public String getNome() { return nome; }
    public void setNome(String n) { this.nome = n; }
    @Override
    public String toString() { return nome; }
}
```

MODEL

Aluno.java: a entidade

Aluno referencia um **Curso** (o "muitos" do relacionamento N-1).

model/Aluno.java

```java
package model;
public class Aluno {
    private Integer id;
    private String nome, email;
    private Curso curso;
    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }
    public String getNome() { return nome; }
    public void setNome(String n) { this.nome = n; }
    public void setEmail(String e) { this.email = e; }
    public Curso getCurso() { return curso; }
    public void setCurso(Curso c) { this.curso = c; }
}
```

DAO

ConnectionFactory: conexão centralizada

Único ponto de criação de **Connection**.
Todos os DAOs usam esta fábrica.

dao/ConnectionFactory.java

```java
package dao;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class ConnectionFactory {
    private static final String URL = "jdbc:postgresql://localhost:5432/escola";
    private static final String USER = "postgres";
    private static final String PASS = "123456";
    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASS);
    }
}
```

DAO

CursoDAO: listando cursos (1/2)

Usado para preencher o **JComboBox** de cursos na View.

dao/CursoDAO.java — listar()

```
01  package dao;
02  import java.sql.*;
03  import java.util.*;
04  import model.Curso;
05  public class CursoDAO {
06      public List<Curso> listar() {
07          List<Curso> l = new ArrayList<>();
08          try (Connection c = ConnectionFactory.getConnection();
09               PreparedStatement ps = c.prepareStatement("SELECT id, nome FROM curso")) {
10              ResultSet rs = ps.executeQuery();
11              while (rs.next()) l.add(montar(rs));
12          } catch (SQLException e) { throw new RuntimeException(e); }
13          return l;
14      }
```

DAO

CursoDAO: mapeando ResultSet (2/2)

O método privado `montar()` converte cada linha do ResultSet em um objeto Curso.

dao/CursoDAO.java — montar()

```
01      private Curso montar(ResultSet rs) throws SQLException {
02          Curso cu = new Curso();
03          cu.setId(rs.getInt("id"));
04          cu.setNome(rs.getString("nome"));
05          return cu;
06      }
07  }
```

DAO

AlunoDAO: matrícula com transação (1/6)

A matrícula faz **duas operações atômicas**: insere o aluno e incrementa o contador do curso.
Ou ambas confirmam, ou ambas desfazem.

dao/AlunoDAO.java — matricular()

```
01 public class AlunoDAO {
02     public void matricular(Aluno a) {
03         try (Connection c = ConnectionFactory.getConnection()) {
04             c.setAutoCommit(false);
05             try {
06                 inserir(c, a);
07                 atualizarTotal(c, a.getCurso().getId());
08                 c.commit();
09             } catch (SQLException e) {
10                 c.rollback(); throw new RuntimeException(e);
11             }
12         } catch (SQLException e) { throw new RuntimeException(e); }
13     }
```

DAO

AlunoDAO: inserir com a conexão compartilhada (2/6)

Recebe a **mesma conexão** da transação — por isso não abre/fecha a conexão aqui.

dao/AlunoDAO.java — inserir()

```
01      private void inserir(Connection c, Aluno a) throws SQLException {
02          String sql = "INSERT INTO aluno (nome, email, curso_id) VALUES (?,?,?)";
03          try (PreparedStatement ps = c.prepareStatement(sql)) {
04              ps.setString(1, a.getNome());
05              ps.setString(2, a.getEmail());
06              ps.setInt(3, a.getCurso().getId());
07              ps.executeUpdate();
08          }
09      }
```

DAO

AlunoDAO: incrementando o contador (3/6)

O `total_alunos` do curso é atualizado na mesma transação da inserção.

dao/AlunoDAO.java — atualizarTotal()

```
01      private void atualizarTotal(Connection c, int cursoId) throws SQLException {
02          String sql = "UPDATE curso SET total_alunos = total_alunos + 1 WHERE id = ?";
03          try (PreparedStatement ps = c.prepareStatement(sql)) {
04              ps.setInt(1, cursoId);
05              ps.executeUpdate();
06          }
07      }
```

DAO

AlunoDAO: listando com JOIN (4/6)

O **JOIN** traz cada aluno já com o nome do curso, evitando consultas N+1.

dao/AlunoDAO.java — listarComCurso()

```
01      public List<Aluno> listarComCurso() {
02          List<Aluno> l = new ArrayList<>();
03          String sql = "SELECT a.id, a.nome, c.nome AS curso FROM aluno a "
04                     + "JOIN curso c ON c.id = a.curso_id ORDER BY a.nome";
05          try (Connection c = ConnectionFactory.getConnection();
06               PreparedStatement ps = c.prepareStatement(sql)) {
07              ResultSet rs = ps.executeQuery();
08              while (rs.next()) l.add(montarAluno(rs));
09          } catch (SQLException e) { throw new RuntimeException(e); }
10          return l;
11      }
```

DAO

AlunoDAO: mapeando o ResultSet (5/6)

Converte a linha do `ResultSet` em um Aluno com seu Curso.

dao/AlunoDAO.java — montarAluno()

```
01      private Aluno montarAluno(ResultSet rs) throws SQLException {
02          Aluno a = new Aluno();
03          a.setId(rs.getInt("id"));
04          a.setNome(rs.getString("nome"));
05          Curso cu = new Curso();
06          cu.setNome(rs.getString("curso"));
07          a.setCurso(cu);
08          return a;
09      }
```

DAO

AlunoDAO: excluindo um aluno (6/6)

Exclusão simples pelo **id**.
Fecha a classe AlunoDAO.

dao/AlunoDAO.java — excluir()

```
01      public void excluir(Aluno a) {
02          String sql = "DELETE FROM aluno WHERE id = ?";
03          try (Connection c = ConnectionFactory.getConnection();
04               PreparedStatement ps = c.prepareStatement(sql)) {
05              ps.setInt(1, a.getId());
06              ps.executeUpdate();
07          } catch (SQLException e) { throw new RuntimeException(e); }
08      }
09  }
```

CONTROLLER

AlunoController: recebendo da View (1/2)

O Controller **monta a entidade** a partir dos dados da View e delega ao DAO.

controller/AlunoController.java — matricular()

```
01 package controller;
02 import dao.*;
03 import model.*;
04 import java.util.List;
05 public class AlunoController {
06     private final AlunoDAO alunoDAO = new AlunoDAO();
07     private final CursoDAO cursoDAO = new CursoDAO();
08     public void matricular(String nome, String email, Curso curso) {
09         Aluno a = new Aluno();
10         a.setNome(nome); a.setEmail(email); a.setCurso(curso);
11         alunoDAO.matricular(a);
12     }
```

CONTROLLER

AlunoController: consultas e exclusão (2/2)

Métodos que a View usa para **preencher o combo**, **listar** e **excluir**

.

controller/AlunoController.java — consultas

```
01      public List<Curso> listarCursos() { return cursoDAO.listar(); }
02      public List<Aluno> listarAlunos() { return alunoDAO.listarComCurso(); }
03      public void excluir(Aluno a) { alunoDAO.excluir(a); }
04  }
```

VIEW

AlunoView: a janela em Swing (1/6)

A View declara os **componentes Swing** e guarda a lista de alunos exibida.
Ela só fala com o Controller.

view/AlunoView.java — estrutura

```
01  package view;
02  import javax.swing.*;
03  import model.*;
04  import controller.AlunoController;
05  import java.util.List;
06  public class AlunoView extends JFrame {
07      private JTextField txtNome, txtEmail;
08      private JComboBox<Curso> cboCurso;
09      private JList<String> lista;
10      private JButton btnSalvar, btnAtualizar, btnExcluir;
11      private final AlunoController ctrl = new AlunoController();
12      private List<Aluno> alunosAtual;
```

VIEW

AlunoView: o construtor (2/6)

O construtor monta a tela, registra as **ações dos botões** e já carrega a lista inicial.

view/AlunoView.java — construtor

```
01      public AlunoView() {
02          super("Cadastro de Alunos");
03          initComponents();
04          acoes();
05          setDefaultCloseOperation(EXIT_ON_CLOSE);
06          setSize(560, 400);
07          setLocationRelativeTo(null);
08          listar();
09          setVisible(true);
10      }
```

VIEW

AlunoView: montando os componentes (3/6)

O combo de cursos vem **direto do Controller**.
A lista ocupa o centro da janela.

view/AlunoView.java — initComponents()

```
01      private void initComponents() {
02          setLayout(new BorderLayout());
03          txtNome = new JTextField(12); txtEmail = new JTextField(12);
04          cboCurso = new JComboBox<>(ctrl.listarCursos().toArray(new Curso[0]));
05          btnSalvar = new JButton("Salvar"); btnAtualizar = new JButton("Atualizar");
06          btnExcluir = new JButton("Excluir"); lista = new JList<>();
07          JPanel topo = new JPanel();
08          topo.add(new JLabel("Nome:")); topo.add(txtNome);
09          topo.add(new JLabel("Email:")); topo.add(txtEmail); topo.add(cboCurso);
10          topo.add(btnSalvar); topo.add(btnAtualizar); topo.add(btnExcluir);
11          add(topo, BorderLayout.NORTH);
12          add(new JScrollPane(lista), BorderLayout.CENTER);
13      }
```

VIEW

AlunoView: ações e salvar (4/6)

Ao salvar, a View lê os campos e **delega tudo ao Controller**.
Depois atualiza a lista.

view/AlunoView.java — acoes() e salvar()

```
01      private void acoes() {
02          btnSalvar.addActionListener(e -> salvar());
03          btnAtualizar.addActionListener(e -> listar());
04          btnExcluir.addActionListener(e -> excluir());
05      }
06      private void salvar() {
07          Curso curso = (Curso) cboCurso.getSelectedItem();
08          ctrl.matricular(txtNome.getText(), txtEmail.getText(), curso);
09          txtNome.setText(""); txtEmail.setText("");
10          listar();
11      }
```

VIEW

AlunoView: excluir e listar (5/6)

listar() mantém **alunosAtual** sincronizada com a JList para permitir a exclusão pelo índice.

view/AlunoView.java — excluir() e listar()

```
01      private void excluir() {
02          int i = lista.getSelectedIndex();
03          if (i >= 0) ctrl.excluir(alunosAtual.get(i));
04          listar();
05      }
06      private void listar() {
07          alunosAtual = ctrl.listarAlunos();
08          DefaultListModel<String> m = new DefaultListModel<>();
09          for (Aluno a : alunosAtual)
10              m.addElement(a.getNome() + " — " + a.getCurso().getNome());
11          lista.setModel(m);
12      }
```

VIEW

AlunoView: iniciando a aplicação (6/6)

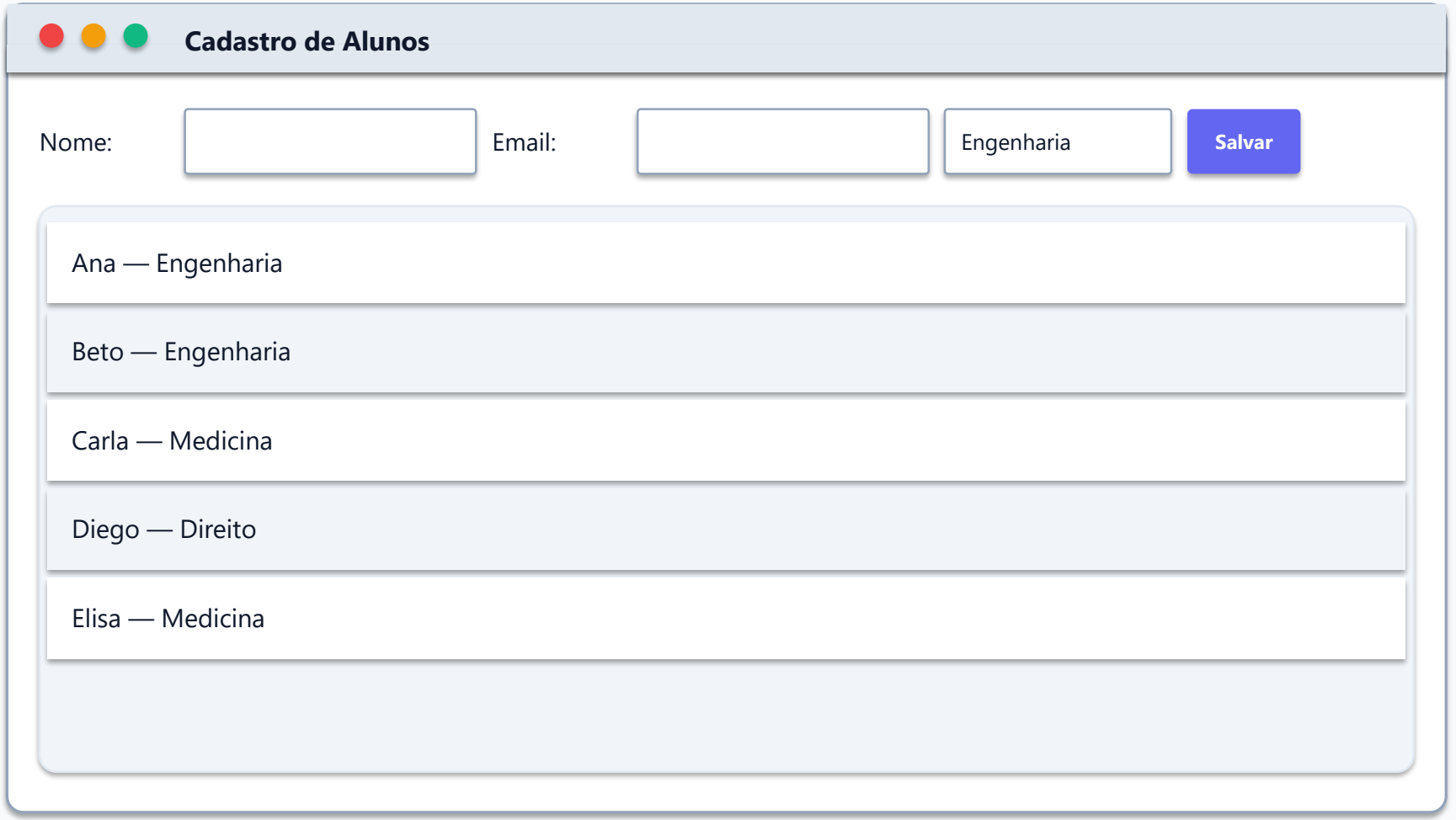
A aplicação inicia na **Event Dispatch Thread**, boa prática do Swing.

view/AlunoView.java — main()

```
01      public static void main(String[] args) {
02          SwingUtilities.invokeLater(() -> new AlunoView());
03      }
04  }
```

RESULTADO

A aplicação integrada rodando



FLUXO

O que acontece ao clicar em Salvar

1 **`View.salvar()`**

lê txtNome, txtEmail e o Curso do combo.

2 **`Controller.matricular()`**

monta o Aluno e chama o DAO.

3 **`AlunoDAO.matricular()`**

abre a transação (setAutoCommit false).

4 **`inserir() + atualizarTotal()`**

INSERT do aluno e UPDATE do contador.

5 **`commit() / rollback()`**

confirma ou desfaz as duas operações.

6 **`View.listar()`**

recarrega a JList com listarComCurso().

BOAS PRÁTICAS

Integração bem feita: o que seguir

View sem SQL

A tela só exibe e captura; o SQL fica no DAO.

Controller sem Swing

O Controller não importa javax.swing.

try-with-resources

Fecha Connection e PreparedStatement automaticamente.

Transação p/ várias operações

setAutoCommit(false) + commit/rollback na matrícula.

JOIN contra o N+1

listarComCurso() traz aluno+curso em uma consulta.

Conexão centralizada

Todos os DAOs usam a ConnectionFactory.

ATENÇÃO

Erros comuns na integração

SQL dentro da View

Acopla a tela ao banco e quebra o MVC.

Controller importando Swing

Impede reutilizar a lógica em outra interface.

Esquecer commit/rollback

A transação fica pendurada ou perde dados.

Não fechar a Connection

Esgota o pool de conexões do PostgreSQL.

Não atualizar a JList

Salva no banco mas a tela não reflete (falta listar()).

Ignorar o índice selecionado

Excluir pelo índice errado apaga outro aluno.

PRÁTICA

Exercícios: evolua a aplicação

EXERCÍCIO 1 — Fácil

Exibir o e-mail na lista

Altere listar() na View para mostrar "nome — email — curso" e ajuste o salvar() para limpar também o campo de e-mail após a matrícula.

EXERCÍCIO 2 — Médio

Filtrar alunos por curso

Adicione um botão "Filtrar" que chama alunosDoCurso() do DAO com o curso selecionado no combo, exibindo apenas esses alunos na JList.

EXERCÍCIO 3 — Desafio

Cadastro de curso com transação

Crie um campo e um botão para cadastrar um novo Curso. Use uma transação que insere o curso e já matricula o aluno digitado nele, com commit/rollback.

Resumo da aula

Swing + JDBC + MVC juntos

- Arquitetura em camadas: view → controller → dao → model
- ConnectionFactory centraliza a conexão
- Transação de matrícula (commit/rollback)
- JOIN para listar aluno com curso
- View Swing completa (combo, lista, botões)

PRÓXIMA AULA

Hibernate: ORM

Vamos trocar o SQL manual por mapeamento objeto-relacional com @Entity, @Id e HQL, mantendo a mesma arquitetura MVC.

```
@Entity + session.persist()
```

Obrigado!

Aula 07 · Parte 2 · Integração Swing + JDBC + MVC

Práticas de POO

Prof. Me. Célio Ricardo Castelano · Ciência da Computação