Introdução à Engenharia de Software, Prof. Dr. Rodrigo Silva.

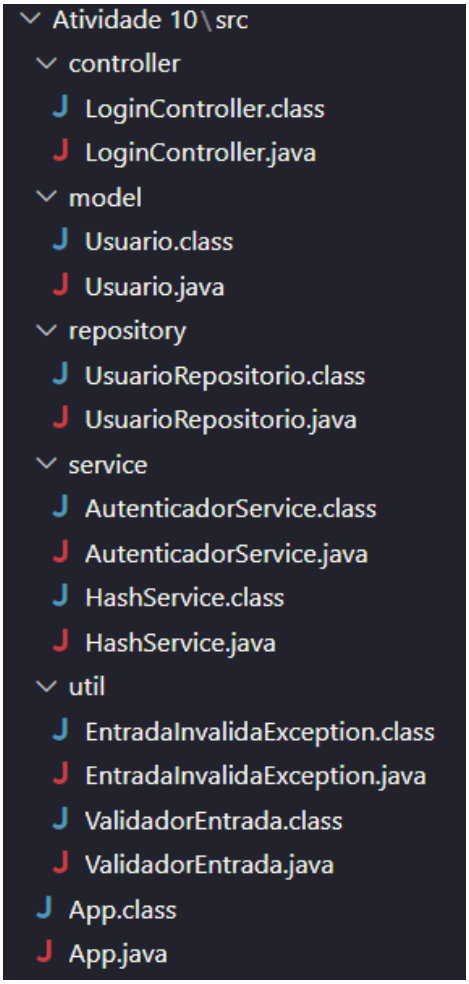
Atividade 10 - Implementação com foco em boas práticas.

Gustavo Micher Santana, 10737606.

Aplicação de Clean Code e SRP no projeto

Neste projeto, procurei deixar o código limpo, legível e organizado, seguindo algumas práticas de Clean Code que facilitam a leitura e a manutenção. Por exemplo:

Nomes claros e descritivos: Todas as classes, métodos e variáveis têm nomes que dizem exatamente o que fazem, como LoginController, AutenticadorService e UsuarioRepositorio. Isso faz com que quem ler o código consiga entender a função de cada parte sem precisar decifrar nada.



Classes pequenas e objetivas: Cada classe tem uma única responsabilidade, sem misturar coisas diferentes. Por exemplo, o HashService cuida apenas de gerar o hash da senha, o ValidadorEntrada só valida o usuário e a senha, e o LoginController só coordena o fluxo de login.

Fluxo de código organizado: O App funciona apenas como ponto de entrada, delegando tudo para as camadas de Controller e Service. Isso deixa o código mais modular e fácil de testar.

Quanto ao SRP (Single Responsibility Principle), ele foi aplicado em praticamente todas as partes do projeto:

O Controller é responsável só por receber os dados do usuário, chamar os serviços e mostrar a resposta.

O Service cuida apenas da lógica de negócio, como autenticar o usuário e comparar hashes.

O Repository é responsável unicamente por armazenar e recuperar os usuários.

A model (Usuario) representa o objeto de usuário, sem se preocupar com lógica ou exibição.

As classes utilitárias (ValidadorEntrada e EntradaInvalidaException) cuidam só de validação e tratamento de erros relacionados à entrada.

Essa separação deixa o código muito mais fácil de entender, manter e evoluir, além de permitir adicionar novas funcionalidades no futuro sem precisar mexer em várias partes diferentes do projeto.

Adicionar validação de entrada e tratamento de exceção

Para deixar o projeto mais robusto, adicionei validação de entrada e tratamento de exceções.

Validação de entrada: Antes de tentar autenticar, o sistema verifica se o usuário e a senha foram realmente informados e se não estão vazios. Isso evita que a aplicação quebre ou tente processar dados inválidos. Essa lógica ficou concentrada na classe ValidadorEntrada, deixando o resto do código mais limpo.

```java
package util;

public class ValidadorEntrada {

    public void validar(String usuario, String senha) throws EntradaInvalidaException {
        if (usuario == null || usuario.isBlank()) {
            throw new EntradaInvalidaException("O nome de usuário não pode estar vazio.");
        }
        if (senha == null || senha.isBlank()) {
            throw new EntradaInvalidaException("A senha não pode estar vazia.");
        }
    }
}
```

Tratamento de exceção: Criei uma exceção específica (EntradaInvalidaException) para lidar com problemas de entrada, e também capturamos erros inesperados no main. Com isso, a aplicação consegue informar ao usuário o que aconteceu de forma clara, sem travar, e o fluxo de execução continua seguro.

```java
package util;

public class EntradaInvalidaException extends Exception {
    public EntradaInvalidaException(String mensagem) {
        super(mensagem);
    }
}
```

Essa etapa deixa o código mais confiável, previne erros simples e facilita a manutenção, porque qualquer problema de entrada é tratado de forma centralizada.

Simular o uso de hash de senha (MessageDigest SHA-256)

Para melhorar a segurança, implementei a simulação de hash de senha, usando SHA-256.

Em vez de armazenar senhas em texto puro, o sistema gera um hash da senha informada pelo usuário e compara com o hash armazenado.

Essa lógica ficou em uma classe separada (HashService), garantindo que o cálculo do hash seja feito sempre de forma consistente e isolada do resto da aplicação.

Mesmo que alguém consiga acessar os dados do “repositório”, não será possível ver a senha real, apenas o hash.

Essa etapa adiciona uma camada de segurança realista e segue a ideia de boas práticas, sem complicar o fluxo do login.

Criar estrutura de camadas (Controller, Service, Repository)

Para deixar o projeto mais organizado e fácil de manter, estruturei o código em camadas separadas, seguindo uma ideia bem comum em aplicações Java.

Controller: É a camada que recebe os dados do usuário e coordena o fluxo da aplicação. No projeto, o LoginController pega o nome de usuário e a senha, chama os serviços responsáveis pela autenticação e retorna a resposta ao usuário. Ele não faz nenhuma lógica complexa por conta própria, apenas orquestra o que precisa ser feito.

```java
package controller;

import service.AutenticadorService;
import util.ValidadorEntrada;
import util.EntradaInvalidaException;

import java.util.logging.Logger;

public class LoginController {

    private static final Logger logger = Logger.getLogger(LoginController.class.getName());

    private final ValidadorEntrada validador = new ValidadorEntrada();
    private final AutenticadorService autenticador = new AutenticadorService();

    public void processarLogin(String usuario, String senha) throws EntradaInvalidaException {
        logger.info("Processando login para o usuário: " + usuario);

        validador.validar(usuario, senha);
        boolean autenticado = autenticador.autenticar(usuario, senha);

        if (autenticado) {
            System.out.println("✅ Bem-vindo, " + usuario + "!");
            logger.info("Login bem-sucedido para o usuário: " + usuario);
        } else {
            System.out.println("❌ Usuário ou senha incorretos.");
            logger.warning("Tentativa de login falhou para o usuário: " + usuario);
        }
    }
}
```

Service: Aqui é onde fica a lógica de negócio da aplicação. O AutenticadorService cuida de verificar se o usuário existe e se a senha fornecida bate com o hash armazenado. Também usei o HashService para calcular o hash da senha. Separar essa lógica do Controller mantém o código mais limpo e facilita testes futuros.

```java
public String gerarHashSHA256(String texto) {
    try {
        MessageDigest digest = MessageDigest.getInstance(algorithm:"SHA-256");
        byte[] hashBytes = digest.digest(texto.getBytes());
        StringBuilder hexString = new StringBuilder();

        for (byte b : hashBytes) {
            String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1) hexString.append(c:'0');
            hexString.append(hex);
        }

        logger.fine(msg:"Hash gerado com sucesso para entrada.");
        return hexString.toString();

    } catch (NoSuchAlgorithmException e) {
        logger.severe("Erro ao gerar hash SHA-256: " + e.getMessage());
        throw new RuntimeException(message:"Algoritmo SHA-256 não disponível.", e);
    }
}
```

Repository: Essa camada é responsável por armazenar e recuperar dados. No meu caso, o UsuarioRepositorio simula um banco de dados de usuários, fornecendo os hashes de senha correspondentes. O Service consulta o Repository, mas não precisa se preocupar com como os dados estão armazenados.

```java
package repository;

import model.Usuario;
import service.HashService;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Logger;

public class UsuarioRepositorio {

    private static final Logger logger = Logger.getLogger(UsuarioRepositorio.class.getName());

    private static final Map<String, Usuario> usuarios = new HashMap<>();
    private final HashService hashService = new HashService();

    public UsuarioRepositorio() {
        // Simula criação de usuários dinâmicos (sem senhas fixas)
        adicionarUsuario("admin", "123");
        adicionarUsuario("gustavo", "abc123");
        adicionarUsuario("maria", "senhaSegura");
    }

    private void adicionarUsuario(String nome, String senhaPura) {
        String hash = hashService.gerarHashSHA256(senhaPura);
        usuarios.put(nome, new Usuario(nome, hash));
        logger.info("Usuário adicionado ao repositório: " + nome);
    }

    public Usuario buscarPorNome(String nome) {
        return usuarios.get(nome);
    }
}
```

Essa separação em camadas traz vários benefícios: o código fica modular, fácil de entender e de manter, é possível trocar ou atualizar partes específicas sem afetar o restante do sistema e ainda facilita a implementação de testes automatizados.