

Relatório de Análise Crítica do Sistema de Moeda Estudantil

Arlindo Sergio, Arthur Astolfi , Vitor Costa

1. Análise do Projeto

(i) Arquitetura e Tecnologias Utilizadas

O projeto utiliza uma arquitetura padrão MVC (Model-View-Controller) aplicada a uma API REST com Spring Boot (v3.5.5) e Java 17.

- **Backend:** A estrutura está bem definida em camadas (`controller`, `service`, `repository`, `entity`, `dto`), o que é uma boa prática para separação de responsabilidades. O uso de Spring Data JPA simplifica a persistência de dados.
- **Banco de Dados:** Utiliza MySQL (`mysql-connector-j`).
- **Ponto de Atenção:** A exposição direta de Entidades JPA nos Controllers (ex: `public List<Aluno> getAllAlunos()`) é ruim porque Isso cria acoplamento forte entre o modelo de banco de dados e a API, podendo gerar problemas de referência circular (JSON infinito) e falhas de segurança (exposição de dados sensíveis como senhas ou IDs internos). O ideal é utilizar sempre DTOs (Data Transfer Objects) para resposta.

(ii) Organização do GitHub

A organização dos diretórios é boa, separando claramente a implementação (`implementacao/back` e `implementacao/front`) da documentação de projeto (`projeto/LabXX...`).

- Pontos Positivos: Histórico de laboratórios (Lab03, Lab04, Lab05).
- Pontos Negativos: O `README.md` na raiz é extremamente pobre ("USAR JAVA 17" e um link).

(iii) Dificuldade para Configuração do Ambiente

A configuração do ambiente apresenta alta dificuldade e riscos para terceiros:

1. Credenciais Hardcoded: O arquivo `application.properties` contém senhas de banco de dados (`root`) e, pior, a senha de e-mail pessoal exposta (`spring.mail.password=bunda`). Isso impede que outros desenvolvedores rodem o projeto sem alterar o código e representa um grave risco de segurança.
2. Falta de Instruções: Não há instruções para rodar o projeto.
3. Utilização de Gradle sem informações de como utilizar

(iv) Sugestões de Melhorias

1. Segurança: Remover imediatamente credenciais do código fonte, utilizando Variáveis de Ambiente.
2. Documentação API: Adicionar o SpringDoc OpenAPI (Swagger) para documentar os endpoints automaticamente, facilitando o teste e integração com o frontend.
3. Uso de DTOs: Refatorar os métodos **GET** para retornarem DTOs ao invés de Entidades, prevenindo vazamento de dados e problemas de serialização e utilização de Paginação ao invés de List..
4. Dockerização: Criar um **Dockerfile** e **docker-compose.yml** para rodar a aplicação e o banco MySQL, garantindo que o ambiente funcione igual em qualquer máquina.

2. Refatorações de Código

Abaixo estão três refatorações identificadas para melhorar a segurança, a coesão e a manutenibilidade do código.

Refatoração 1: Remoção de Lógica de Negócio do Controller

Arquivo: **VantagemController.java** Problema: O método **create** no Controller está realizando lógica de negócio (buscar o Aluno e montar o objeto Vantagem). O Controller deve apenas receber a requisição e passar para o Service.

Antes:

```
// VantagemController.java
@PostMapping
public Vantagem create(@RequestBody VantagemCreateDTO dto) {
    // Controller acessando repositório de outra entidade e montando objeto
    Aluno aluno = alunoRepository.findById(dto.getAlunoid())
        .orElseThrow(() -> new RuntimeException("Aluno não encontrado!"));

    Vantagem vantagem = new Vantagem();
    vantagem.setFoto(dto.getFoto());
    vantagem.setValor(dto.getValor());
    // ... sets manuais ...
    vantagem.setAluno(aluno);

    return vantagemService.createVantagem(vantagem);
}
```

Depois:

```
// VantagemController.java
@PostMapping
public Vantagem create(@RequestBody VantagemCreateDTO dto) {
    // Controller delega tudo para o Service
    return vantagemService.createVantagem(dto);
}

// VantagemService.java
```

```

@Autowired
private AlunoRepository alunoRepository;

public Vantagem createVantagem(VantagemCreateDTO dto) {
    Aluno aluno = alunoRepository.findById(dto.getAlunoId())
        .orElseThrow(() -> new RuntimeException("Aluno não encontrado!"));

    Vantagem vantagem = new Vantagem();
    vantagem.setFoto(dto.getFoto());
    vantagem.setValor(dto.getValor());
    vantagem.setQuant(dto.getQuant());
    vantagem.setDescricao(dto.getDescricao());
    vantagem.setEmpresald(String.valueOf(dto.getEmpresald()));
    vantagem.setAluno(aluno);

    return vantagemRepository.save(vantagem);
}

```

Justificativa: Aumenta a coesão. O Service passa a ser o único responsável pelas regras de criação de uma vantagem, facilitando testes unitários e reaproveitamento de código.

Refatoração 2: Proteção de Credenciais Sensíveis

Arquivo: `application.properties` Problema: Senhas de banco de dados e e-mail estão expostas em texto plano (hardcoded).

Antes:

```

spring.datasource.password=root
spring.mail.username=pedrobarros813@gmail.com
spring.mail.password=bunda

```

Depois:

```

spring.datasource.password=${DB_PASSWORD}
spring.mail.username=${MAIL_USERNAME}
spring.mail.password=${MAIL_PASSWORD}

```

Refatoração 3: Melhoria no Tratamento de Exceções

Arquivo: `AlunoService.java` Problema: O uso de `RuntimeException` genérica gera respostas HTTP 500 (Internal Server Error) quando um ID não é encontrado, o que é semanticamente incorreto para uma API REST (deveria ser 404 Not Found).

Antes:

```

public Aluno updateAluno(Long id, Aluno updateAluno) {

```

```

return alunoRepository.findById(id)
.map(aluno -> {
    // ... updates ...
    return alunoRepository.save(aluno);
}).orElseThrow(() -> new RuntimeException("Aluno not found with id " + id));
}

```

Depois:

```

import org.springframework.web.server.ResponseStatusException;
import org.springframework.http.HttpStatus;

public Aluno updateAluno(Long id, Aluno updateAluno) {
    return alunoRepository.findById(id)
        .map(aluno -> {
            aluno.setCpf(updateAluno.getCpf());
            aluno.setNome(updateAluno.getNome());
            // ... outros sets ...
            return alunoRepository.save(aluno);
        }).orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND,
            "Aluno não encontrado com id " + id));
}

```

Refatoração 4: Uso de DTO (Data Transfer Object) para Respostas

Arquivo: [AlunoController.java](#) e [AlunoService.java](#).

Problema: O método `getAllAlunos` retorna diretamente a entidade de banco de dados `Aluno`. Isso acopla a API à estrutura do banco. Se você adicionar um campo sensível no banco (ex: `senha` ou `audit_log`), ele será exposto na API automaticamente. Além disso, pode causar erros de "LazyInitialization" ou loops infinitos de JSON se houver relacionamentos bidirecionais.

Antes:

```

@GetMapping
public List<Aluno> getAllAlunos() {
    return alunoService.getAllAlunos();
}

```

Depois:

```

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

```

@Service

public class AlunoService {

@Autowired

private AlunoRepository alunoRepository;

// ... outros métodos ...

// Alterado de List para Page e recebe Pageable

public Page<Aluno> getAllAlunos(Pageable pageable) {

return alunoRepository.findAll(pageable);

}

}

// Controller

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.web.PageableDefault;

import org.springframework.http.ResponseEntity;

// ... outros imports

@RestController

@RequestMapping("/alunos")

public class AlunoController {

@Autowired // Ou via construtor, como sugerido anteriormente

private AlunoService alunoService;

```

@GetMapping

public ResponseEntity<Page<AlunoDTO>> getAllAlunos(

    @PageableDefault(page = 0, size = 10, sort = "nome") Pageable
    pageable

) {

    // Busca a página de entidades (Aluno)

    Page<Aluno> pageAlunos = alunoService.getAllAlunos(pageable);

    // Converte cada Aluno da página para AlunoDTO

    Page<AlunoDTO> pageDtos = pageAlunos.map(aluno -> new AlunoDTO(

        aluno.getNome(),

        aluno.getEmail(),

        aluno.getCurso()

    ));

    return ResponseEntity.ok(pageDtos);

}

// ... outros métodos

}

```