Relatório de Análise Crítica

Arthur Lara Panzera, Diogo Caribe Brunoro, Felipe Augusto Pereira de Sousa, Guilherme de Almeida Rocha Vieira

[1]Pontifícia Universidade 30535-901 – Belo Horizonte – Minas Gerais – Brazil

{diogocaribebrunoro, felipeaps0918, guilhermearv3}@gmail.com,

arthurpanzera@outlook.com

Resumo. *Este relatório apresenta uma análise crítica e o processo de refatoração do projeto desenvolvido pelo Grupo 1, conforme solicitado para a disciplina. O trabalho abrange uma avaliação aprofundada da arquitetura e das tecnologias empregadas, a organização do repositório no GitHub, a facilidade de configuração do ambiente de desenvolvimento e sugestões construtivas para melhorias futuras. Adicionalmente, foram identificadas e refatoradas três seções específicas do código-fonte, visando otimizar sua legibilidade, manutenibilidade e desempenho. As refatorações foram documentadas com a apresentação do código antes e depois das modificações, justificando as abordagens aplicadas e demonstrando os benefícios alcançados.*

1. Intrudução

Este documento apresenta a análise crítica e o processo de refatoração de um projeto desenvolvido no contexto da disciplina de Laboratório de Desenvolvimento de Software. Nosso grupo foi responsável por analisar e refatorar o projeto do Grupo 1, formado por Gabriel Chagas, Arthur Santana, Joao Vitor Santana e Pedro Maia, e acessível em: `https://github.com/gabrielchagas13/lab-software-sistema-de-moeda`.

A análise crítica focou em quatro pilares principais:

- **Arquitetura e Tecnologias Utilizadas**;
- **Organização do GitHub**;
- **Dificuldade para Configuração do Ambiente**;
- **Sugestões de Melhorias**.

Adicionalmente, foram identificadas e refatoradas três seções do código-fonte para aprimorar sua compreensão e manutenção. As refatorações foram submetidas via Pull Requests e são detalhadas neste relatório, com o código antes e depois das modificações e suas respectivas justificativas.

2. Arquitetura e Tecnologias Utilizadas

O sistema foi desenvolvido utilizando uma arquitetura robusta e tecnologias modernas para garantir escalabilidade, segurança e uma experiência de usuário responsiva. A arquitetura do **backend segue o padrão MVC (Model-View-Controller)** para uma clara

separação de responsabilidades e, para otimizar a organização e comunicação entre as camadas, foram aplicados os padrões de projeto **DTO (Data Transfer Object)** e **Repository**. O frontend, embora modularizado, atua como uma ”View”independente que consome a API do backend.

2.1. Backend

A camada de backend foi implementada com as seguintes tecnologias, seguindo o padrão MVC:

- **Java 17**: Linguagem de programação robusta e amplamente utilizada, oferecendo performance e um ecossistema vasto.
- **Spring Boot 3.1.5**: Framework para desenvolvimento rápido de aplicações Java, simplificando a configuração e o deploy.
- **Spring Data JPA (Hibernate)**: Mapeamento Objeto-Relacional (ORM) para acesso eficiente e abstraído ao banco de dados.
- **Spring Security**: Framework de segurança para autenticação e autorização, configurado de forma básica para ambiente de desenvolvimento.
- **MySQL 8.0**: Sistema de gerenciamento de banco de dados relacional robusto e de código aberto.
- **Maven**: Ferramenta de automação de construção e gerenciamento de dependências para projetos Java.
- **Lombok**: Biblioteca que reduz o código boilerplate em classes Java, como getters, setters e construtores.
- **Bean Validation (JSR-303)**: API para validação de dados em JavaBeans, garantindo a integridade dos dados.

2.2. Frontend

A interface do usuário foi construída com foco em responsividade e usabilidade, empregando as seguintes tecnologias, e atuando como a ”View”que interage com o backend:

- **HTML5 + CSS3 + JavaScript (ES6+)**: Padrões web para estruturação, estilização e interatividade das páginas.
- **Estrutura MVC modularizada**: Organização do código em módulos (`pages/`, `styles/`, `services/`) para facilitar a manutenção e escalabilidade.
- **Layout responsivo com Flexbox e Grid**: Técnicas de CSS para garantir que a aplicação se adapte a diferentes tamanhos de tela.
- **Consumo de API REST via Fetch API**: Mecanismo nativo do navegador para fazer requisições HTTP assíncronas e interagir com o backend.

2.3. Infraestrutura

Para o ambiente de desenvolvimento e deploy, foram utilizadas as seguintes ferramentas e plataformas:

- **Docker e Docker Compose**: Para conteinerização da aplicação e orquestração de múltiplos serviços, garantindo ambientes consistentes.
- **Volume persistente (db_data)**: Configurado para o banco de dados, assegurando que os dados não sejam perdidos após a remoção ou atualização dos contêineres.
- **Rede interna Docker (sistema-moeda-network)**: Para comunicação segura e isolada entre os contêineres da aplicação.
- **Deploy em Vercel (frontend), Render (backend) e Neon (banco)**: Plataformas de cloud para hospedagem e deploy do frontend, backend e banco de dados, respectivamente, otimizando a disponibilidade e o gerenciamento.

3. Organização do Github

No geral, o repositório está bem-organizado, possuindo um arquivo README.md explicativo que detalha o projeto e as instruções de configuração, o que é um ponto forte. No entanto, algumas sugestões podem otimizar ainda mais a clareza e a coesão da estrutura do repositório:

- **Pastas docs e modelagens:** Atualmente, o repositório apresenta duas pastas distintas, docs e modelagens. A pasta docs contém tanto arquivos de documentação quanto alguns diagramas, enquanto a pasta modelagens também é dedicada a diagramas. Para uma organização mais clara e intuitiva, a sugestão seria consolidar todos os diagramas em uma única pasta, idealmente a modelagens, que se dedicaria exclusivamente aos modelos visuais e diagramas do projeto.
- **Agrupamento do código e Docker Compose:** As pastas backend e frontend estão no nível raiz do repositório, assim como o arquivo docker-compose.yml. Uma abordagem para agrupar logicamente esses elementos seria criar uma pasta raiz chamada codigo, e mover backend, frontend e o docker-compose.yml para dentro dela. Isso criaria uma separação clara entre o código-fonte/infraestrutura da aplicação e a documentação/recursos auxiliares.
- **Inclusão de diagramas no README:** Embora o README.md seja bem explicativo, a inclusão dos diagramas mais relevantes (como arquitetura ou fluxo de dados) diretamente no README principal, ou referências visíveis a eles, poderia enriquecer ainda mais a visão geral do projeto para quem o acessa pela primeira vez, eliminando a necessidade de navegar em outras pastas para obter essa informação crucial.

4. Dificuldade para Configuração do Ambiente

A configuração do ambiente de desenvolvimento mostrou-se, de modo geral, bastante acessível e fluida. O ponto positivo determinante foi a utilização da tecnologia de conteinerização **Docker**, que abstraiu a necessidade de instalar dependências locais complexas, como o Java JDK ou o servidor MySQL.

Graças à orquestração dos contêineres, foi possível inicializar todo o ecossistema da aplicação com a execução de apenas um comando no terminal. Isso garantiu agilidade e minimizou problemas de compatibilidade comuns em configurações manuais de ambiente.

5. Sugestões de Melhorias

Com base na análise da arquitetura, organização do código e processo de configuração, foram identificados pontos estratégicos para elevar a qualidade, a manutenibilidade e a segurança do projeto. A seguir, são apresentadas as principais sugestões:

- **Padronização das Variáveis de Ambiente**: Observou-se que o projeto atualmente não utiliza variáveis de ambiente, fazendo com que valores sensíveis ou parâmetros de configuração fiquem diretamente escritos no código-fonte. Recomenda-se fortemente a criação de um arquivo `.env.example` na raiz do repositório, listando todas as variáveis necessárias para o funcionamento do sistema. Assim, novos desenvolvedores terão um guia claro para configurar o ambiente local, além de promover maior segurança e evitar a exposição de informações sensíveis.

- **Separação da Estilização CSS**: Foi identificado o uso de regras de estilo diretamente dentro dos arquivos HTML. Para manter a organização, reduzir a duplicação de código e facilitar a manutenção, recomenda-se centralizar a estilização em arquivos CSS externos. Essa prática melhora a clareza do código, permite reaproveitamento de estilos e segue padrões amplamente adotados no desenvolvimento web.
- **Documentação da API com Swagger/OpenAPI**: Recomenda-se a integração da biblioteca *SpringDoc OpenAPI*. Isso permitiria a geração automática de uma interface interativa (Swagger UI), facilitando o entendimento dos *endpoints*, dos contratos de dados (DTOs) e códigos de resposta HTTP. Tal documentação é vital para reduzir o acoplamento intelectual entre as equipes de frontend e backend.
- **Ampliação da Cobertura de Testes**: A implementação de uma suíte robusta de testes unitários (com JUnit e Mockito) e testes de integração é fundamental. A presença de testes automatizados garante a integridade do sistema durante processos de refatoração e previne a regressão de funcionalidades já existentes.

6. Refatorações

As refatorações realizadas tiveram como objetivo aprimorar a legibilidade, manutenibilidade e concisão do código,

6.1. Refatoração 01 - Bloqueio de resgate duplicado de vantagens para alunos

Este Pull Request implementa uma regra de validação no frontend (perfil Aluno) para impedir que o usuário resgate a mesma vantagem múltiplas vezes caso já possua um cupom ativo para ela. A alteração impacta a visualização do catálogo de vantagens e a lógica de abertura do modal de confirmação.

Link do Pull Request: `https://github.com/gabrielchagas13/lab-software-sistema-de-moeda/pull/7`

6.1.1. Motivação e Contexto

Problema: Anteriormente, o sistema permitia que o botão ”Resgatar”ficasse ativo sempre que o aluno tivesse saldo suficiente, ignorando o fato de ele já ter adquirido aquela vantagem anteriormente. Isso gerava:

- **Má experiência de uso (UX):** O aluno não sabia quais itens já possuía.
- **Gastos acidentais:** Possibilidade de gastar moedas duplicadas no mesmo item sem necessidade.

Solução: Agora, o sistema cruza os dados das vantagens disponíveis com o histórico de cupons do aluno para bloquear visualmente e logicamente o resgate duplicado.

6.1.2. Detalhes da Implementação

- **Arquivo:** `js/vantagens_aluno.js`
- **Classe:** `VantagensAlunoManager`

1. Atualização no Método `renderVantagens`

- **Mapeamento Otimizado:** Criado um `Set` contendo os IDs das vantagens presentes em `this.meusCupons` para busca rápida ($O(1)$).
- **Tratamento de Tipos:** Adicionada conversão explícita para `String` (`String(id)`) na comparação para evitar falsos negativos causados por divergência de tipos (`Number` vs `String`) vindos da API.
- **Feedback Visual:**
  - Se o aluno já possui a vantagem, o botão muda para "Já Resgatado"(estilo `btn-secondary`).
  - O atributo `disabled` é aplicado ao botão.
  - A imagem do card recebe `opacity-50` para destacar visualmente os itens ainda disponíveis.

2. Segurança no Método `showConfirmModal`

- **Guard Clause:** Adicionada uma verificação redundante no momento do clique. Mesmo que um usuário force a habilitação do botão via Inspecionar Elemento, a função verifica novamente se o cupom já existe antes de abrir o modal.
- **Alerta:** Exibe um `Swal.fire` (alerta) informando que a vantagem já foi adquirida.

6.1.3. Como Testar

1. Faça login com um usuário do tipo ALUNO.
2. Certifique-se de ter saldo de moedas.
3. Resgate uma vantagem qualquer.
4. Após o resgate (e recarregamento da lista), verifique se o card da vantagem adquirida:
   - Exibe o botão cinza com texto "Já Resgatado".
   - O botão não é clicável.
   - A imagem está levemente transparente.
5. Tente resgatar uma vantagem diferente para garantir que o fluxo normal continua funcionando.

6.2. Refatoração 02 - Validação de Documentos (CPF/CNPJ) e Máscaras de Input

Este Pull Request implementa máscaras de entrada (*input masks*) e validação lógica matemática para documentos oficiais (CPF e CNPJ) na tela de registro de usuários. O objetivo é garantir a integridade dos dados antes do envio para a API e melhorar a usabilidade.

Link do Pull Request: `https://github.com/gabrielchagas13/lab-software-sistema-de-moeda/pull/9`

6.2.1. Motivação e Contexto

Problema: O formulário de cadastro permitia a entrada de qualquer sequência de caracteres nos campos de CPF e CNPJ. Além disso:

- Os dados eram enviados com formatação (pontos e traços), o que poderia quebrar o backend que espera apenas números.
- Não havia verificação se o CPF/CNPJ era matematicamente válido, permitindo cadastros ”lixo”.
- Problemas de cache no navegador impediam a atualização de scripts externos.

Solução:

- Integração da biblioteca `IMask` para formatar os campos visualmente enquanto o usuário digita.
- Implementação do algoritmo de Módulo 11 para validar CPF e CNPJ.
- Envio apenas dos números (`unmasked value`) para a API.

6.2.2. Detalhes da Implementação

- **Arquivos Alterados:**
  - `pages/registrar.html`
  - `js/registro.js`

1. Máscaras e UX Utilização da biblioteca `IMask` para aplicar os seguintes padrões:

- **CPF:** `000.000.000-00`
- **CNPJ:** `00.000.000/0000-00`
- **RG:** `00.000.000` (com suporte flexível para dígitos verificadores como ’X’).
- **Telefone:** `(00) 00000-0000`

2. Validação Lógica (Validators) Criação de um objeto utilitário `Validators` contendo:

- `isCPF(cpf)`: Verifica tamanho, dígitos repetidos e realiza o cálculo dos dois dígitos verificadores.
- `isCNPJ(cnpj)`: Realiza o cálculo matemático padrão de validação de CNPJ.
- `isRG(rg)`: Validação simples de comprimento mínimo.

3. Integração com Backend No momento do `submit`, o código agora captura `mask.unmaskedValue` (valor puro) em vez do `input.value` (valor formatado). O envio é interrompido (`throw new Error`) caso as validações falhem, exibindo um alerta via `SweetAlert2`.

4. Cache Busting Adicionado parâmetro de versionamento na chamada do script (`src="../js/registro.js?v=4"`) no HTML para forçar a renovação do cache do navegador e garantir que o usuário sempre carregue a versão mais recente da lógica de validação.

6.2.3. Como Testar

1. Acesse a tela de Criar Conta.
2. **Teste de Máscara:** Digite números nos campos CPF/CNPJ e verifique se a pontuação aparece automaticamente.
3. **Teste de Erro:**
    - Tente cadastrar um Aluno com CPF `111.111.111-11` (inválido).
    - O sistema deve exibir um alerta de erro e não enviar a requisição.
4. **Teste de Sucesso:**
    - Utilize um gerador de CPF/CNPJ válido online (ex: 4devs).
    - Preencha os dados e clique em finalizar.
    - O sistema deve exibir sucesso e redirecionar para o login.

6.3. Refatoração 03 - Correção de Fuso Horário nas Transações

Este Pull Request corrige um bug crítico onde a data e hora das transações (`dataTransacao`) estavam sendo gravadas com 3 horas de adiantamento em relação ao horário de Brasília (provável conflito entre UTC do servidor/banco e o horário local). A alteração remove a dependência da geração automática de data pelo Hibernate e passa a controlar o carimbo de tempo (*timestamp*) explicitamente via código Java, forçando o `ZoneId` correto.

Link do Pull Request: `https://github.com/gabrielchagas13/lab-software-sistema-de-moeda/pull/10`

6.3.1. Motivação e Contexto

Problema: Ao realizar um envio de moedas ou resgate de vantagem, o registro no banco de dados ficava com o horário UTC (ex: 15:00 em vez de 12:00). Mesmo configurando o `TimeZone` na classe `Main`, o Hibernate/Driver JDBC continuava convertendo o horário antes de persistir.

Solução: Removemos a anotação `@CreationTimestamp` da entidade e passamos a definir o `LocalDateTime.now()` usando o `ZoneId` `America/Sao_Paulo` diretamente na camada de serviço antes de salvar.

6.3.2. Detalhes da Implementação

- **Arquivos Alterados:**
    - `model/Transacao.java`
    - `service/TransacaoService.java`

1. Entidade `Transacao`

- **Removido:** A anotação `@CreationTimestamp` do campo `dataTransacao`.
- **Motivo:** Impedir que o Hibernate gerencie a data automaticamente usando o relógio do servidor/banco (que geralmente está em UTC).

2. Serviço `TransacaoService`

- **Adicionado:** `import java.time.ZoneId;`
- **Lógica:** Nos métodos de criação de transação (`enviarMoedas`, `resgatarVantagem`, `transferirVantagem`, etc.), a data é setada manualmente:
  `transacao.setDataTransacao(LocalDateTime.now(ZoneId.of("America/Sa`

6.3.3. Como Testar

1. Inicie a aplicação.
2. Realize uma operação de transação:
   - **Cenário A:** Professor envia moedas para um Aluno.
   - **Cenário B:** Aluno resgata uma vantagem.
   - **Cenário C:** Aluno transfere um cupom para outro.
3. Verifique o Horário:
   - **Via JSON (Postman/Insomnia):** No corpo da resposta, verifique se o campo `dataTransacao` bate exatamente com o horário do seu relógio (ex: se são 16:30, não pode aparecer 19:30).
   - **Via Banco de Dados:** Execute a query `SELECT * FROM transacao ORDER BY id DESC LIMIT 1;` e confira se a coluna `data_transacao` está correta.
4. **Resultado Esperado:** O horário registrado deve respeitar o fuso UTC-3 (Horário de Brasília), eliminando o adiantamento de 3 horas.