Projeto A3

Gestão e Qualidade De Software

Bruno Barreto Dutra Santos - 823157408
Kauê Marinho Gorgone - 823146595
Miguel Melo de Almeida - 823155243
Victor Shigueru Agliardi Sato - 823118171

1.Introdução

O código-fonte do projeto refatorado está versionado em um repositório Git, com histórico de commits e possibilidade de publicação no GitHub.
github_url: “https://github.com/miguel-author/A3-Projetc"

O projeto consiste em uma aplicação full stack de lista de tarefas, com frontend em React (pasta `ldt-front`) e backend em NestJS (`task-api`). Na versão original, o backend estava acoplado a um banco relacional via TypeORM, e o frontend consumia a API com pouca separação de responsabilidades.

A refatoração envolveu:

- Migração do backend para MongoDB com Mongoose;
- Reorganização de serviços, controllers, DTOs e schemas;
- Melhoria de legibilidade e modularização no frontend;
- Criação e padronização de testes automatizados com Jest;
- Aplicação de princípios de boas práticas (SOLID, DRY, KISS, YAGNI).

2. Código Antigo

Frontend:

O frontend original em já utilizava React e React Router, porém com algumas limitações:

- Rotas sem proteção/autenticação;
- Lógica de navegação simples;
- Componentes com pouca documentação interna;
- Serviços de API pouco padronizados.

Deficiências:

Df1. "Rotas sem separação clara entre públicas e privadas"

Exemplo de trecho original:

App.js (versão antiga)

```
<BrowserRouter>
  <Routes>
    <Route path='/' element={<LoginPage/>}/>
    <Route path='/cadastro' element={<Cadastro/>}/>
    <Route path="/listadetarefas" element={<Ldt/>}/>
    <Route path="/novaTarefa" element={<NovaTarefa/>}/>
  </Routes>
</BrowserRouter>
```

Problema:

Todas as rotas ficavam expostas independentemente de autenticação, sem um componente de proteção (ex.: ProtectedRoute). Qualquer usuário poderia acessar as páginas de tarefas sem login, apenas digitando a URL.

Df2. "Formulário de nova tarefa acoplado e mais complexo que o necessário"

Exemplo de trecho original:

NovaTarefa.js (Antigo)

```
const[title, setTitle] = useState('');
const[description, setDescription] = useState('');
const[status, setStatus] = useState('');
const[expirationDate, setExpirationDate] = useState('');

     const createTask = async (e) => {
            e.preventDefault();
            try {
              await instance.post("http://localhost:3000/task", {
                title,
                description,
                status,
                expirationDate
              });
            } catch (error) {
              setMensagem('Erro ao criar tarefa');
            }
     }
```

Problema:

O componente fazia a chamada HTTP diretamente com axios/instance, acumulava muitos campos e estados para uma funcionalidade simples, e misturava responsabilidade de UI com acesso à API. Isso aumenta a complexidade e dificulta o reuso (violando KISS e DRY).

Df3. "Serviço de API pouco documentado e limitado"

Exemplo de trecho original:

Service.js (Antigo)

```
export const deleteTask = async (id) => {
            try {
              await instance.delete(`/task/${id}`);
            } catch (error) {
              console.error('Não foi possivel excluir a task',
error);
            }
          };

          export const updateTask = async (id, taskData) => {
            try {
              await instance.put(`/task/${id}`, taskData);
            } catch (error) {
              console.error('There was an error updating the
task!', error);
            }
          };
```

Problema:

O serviço tinha poucas operações centralizadas e nenhum contrato bem documentado (sem JSDoc ou descrição dos parâmetros). A API do frontend não era padronizada, dificultando a evolução.

Backend:

O backend original usava NestJS com TypeORM e entidades relacionais (Users/Tasks). A estrutura já separava controller, service e entity, mas com alguns pontos de acoplamento e ausência de algumas validações importantes.

Deficiências:

Df1. "Listagem de tarefas sem filtro por usuário"

Exemplo de trecho original:

task.service.ts (versão antiga)

```
  async findAll(): Promise<Tasks[]> {
            return await this.taskRepository.find();
          }
```

Problema:

O método `findAll` retornava todas as tarefas de todos os usuários, sem filtrar por usuário logado. Isso impacta segurança e organização de dados.

Df2. "Ausência de validação de ID e uso direto de tipos primitivos"

Exemplo de trecho original:

Task.service.ts (Antigo)

```
 async deleteTask(id: number): Promise<void>{
            await this.taskRepository.delete(id);
          }
```

Problema:

Não havia validação de formato/consistência de ID antes de tentar remover ou atualizar registros. Isso dificulta mensagens de erro mais claras ao usuário e abre margem para exceções técnicas.

Df3. "Acoplamento a TypeORM e banco relacional"

Exemplo de Trecho Original:

User.service.ts (Antigo)

```
constructor(
            @InjectRepository(User)
            private userRepository: Repository<User>
          ) {}
```

Problema:

Toda a camada de domínio de usuário e tarefas dependia diretamente de entidades e repositórios do TypeORM, dificultando a migração para outra tecnologia de banco (como MongoDB) e reduzindo flexibilidade.

Df4. "Testes automatizados limitados"

Problema:

Embora existissem alguns testes, eles se concentravam em partes específicas (principalmente auth e app.controller) e não cobriam os serviços de domínio principais (UsersService, TaskService). A estrutura de testes misturava arquivos `.spec.ts` dentro de `src`com poucos testes na pasta `test`.

3. Código Refatorado

Frontend

Melhorias:

Mh1. "Criação de rotas protegidas com ProtectedRoute"

Exemplo de trecho refatorado:

App.js (versão nova)

```
<BrowserRouter>
          <Routes>
            {/* Rotas públicas */}
            <Route path="/login" element={<Login />} />
            <Route path="/cadastro" element={<Cadastro />} />

            {/* Rotas privadas protegidas por token */}
            <Route
              path="/listadetarefas"
              element={
                <ProtectedRoute>
                  <LDT />
                </ProtectedRoute>
              }
            />
            <Route
              path="/novaTarefa"
              element={
                <ProtectedRoute>
                  <NovaTarefa />
                </ProtectedRoute>
              }
            />

            {/* Fallback */}
            <Route path="*" element={<Login />} />
          </Routes>
        </BrowserRouter>
```

Justificativa:

Foi criado o componente `ProtectedRoute` para aplicar o princípio de Single Responsibility (SOLID): ele cuida apenas de verificar autenticação via token no `localStorage` e redirecionar para `/login` quando necessário. Assim, as páginas de tarefa ficam protegidas e o código de verificação de login não se espalha pelos componentes (DRY/KISS).

Mh2. "Simplificação de Nova Tarefa e extração da lógica de API"

Exemplo de Trecho Refatorado:

novaTarefa.js (versão nova)

```
import { useState } from "react";
import { createTask } from "../service/service";

     export default function NovaTarefa() {
          const [title, setTitle] = useState("");

          async function handleSubmit(e) {
              e.preventDefault();
              await createTask({ title, description: "Criada via
front", status: "PENDENTE" });
              setTitle("");
            }
     }

          // service.js (versão nova)
     export async function createTask(taskData) {
          const response = await api.post("/tasks", taskData);
          return response.data;
     }
```

Justificativa:

A lógica de comunicação com o backend foi removida do componente e centralizada em `service.js`, aplicando DRY e separando responsabilidades. O formulário foi simplificado para focar no campo realmente usado pelo backend, reduzindo complexidade desnecessária (KISS/YAGNI) e deixando o fluxo mais claro para manutenção.

Mh3. "Serviço de API padronizado e documentado"

Exemplo de Trecho Refatorado:

service.js (versão nova)

```
const api = axios.create({
      baseURL: "http://localhost:3000",
      });

/**
* Busca todas as tarefas do usuário logado
*/
      export async function getTasks() {
      const response = await api.get("/tasks");
      return response.data;
      }
```

Justificativa:

Foi criada uma instância API única com baseURL centralizado, e cada função de serviço possui comentário explicando o propósito e parâmetros. Isso melhora a legibilidade e aplica DRY, além de facilitar futuras mudanças de URL ou headers.

Backend:

Melhorias:

Mh1. "Migração para MongoDB com Mongoose e isolamento por usuário"

Exemplo de Trecho Refatorado:

task.service.ts (versão nova)

```
async findAll(userId: string): Promise<Task[]> {
      return this.taskModel
      .find({ userId })// filtra somente tarefas do usuário
      .lean()
      .exec();
}

async remove(id: string, userId: string): Promise<void> {
      if (!Types.ObjectId.isValid(id)) {
      throw new BadRequestException('ID inválido.');
}
const removed = await this.taskModel.findOneAndDelete({ _id: id,
userId }).exec();
if (!removed) {
```

```
    throw new NotFoundException('Tarefa não encontrada ou não
pertence ao usuário.');
            }
}
```

Justificativa:

O serviço de tarefas agora trabalha diretamente com models do Mongoose e filtra todas as operações pelo `userId`, garantindo isolamento de dados por usuário. Inclui validação explícita de ObjectId antes das operações, melhorando robustez e mensagens de erro. A migração para schemas Mongoose reduz acoplamento ao TypeORM e aplica melhor o princípio de inversão de dependência (Nest injeta os models em vez de repositórios específicos).

Mh2. "Controle de IDs sequenciais com Counter e utilitário dedicado"

Exemplo de Trecho Refatorado:

users.service.ts (trecho)

```
const nextId = await getNextSequence(this.counterModel, 'userId');
const created = new this.userModel({
      ...dto,
      id_user: nextId,
      });
```

Justificativa:

O mecanismo de auto incremento foi abstraído para um utilitário (`getNextSequence`) e um schema de `Counter`, centralizando e reutilizando a lógica (DRY). O serviço de usuário apenas consome essa função, mantendo seu foco na regra de negócio (SRP - SOLID).

Mh3. "Reorganização e ampliação da suíte de testes"

Exemplo de Trecho Refatorado:

task.service.spec.ts (trecho)

```
 it('deve remover tarefa do usuário logado', async () => {
     const taskId = new Types.ObjectId().toString();
     const userId = new Types.ObjectId().toString();
```

```
jest.spyOn(mockTaskModel, 'findOneAndDelete').mockReturnValue({
exec: jest.fn().mockResolvedValue({ _id: taskId }),
} as any);

await expect(service.remove(taskId,
userId)).resolves.not.toThrow();

expect(mockTaskModel.findOneAndDelete).toHaveBeenCalledWith({ _id:
taskId, userId });
          });
```

Justificativa:

Os testes foram movidos para a pasta `test`, seguindo o padrão recomendado pelo NestJS, e foram adicionados testes específicos para TaskService, UserService e módulos de autenticação. Os mocks de Mongoose foram criados manualmente, permitindo validar a regra de negócio sem depender de um banco real. Isso garante que a refatoração continue funcionando e facilita futuras mudanças.

4. Critérios de Avaliação

Legibilidade:

Analise:

Houve melhoria significativa de legibilidade, com:

- Nomes de componentes e serviços mais claros (LDT, NovaTarefa, ProtectedRoute);
- Comentários explicativos apenas onde há regra de negócio importante;
- Uso de JSDoc/descrições curtas em serviços e controllers;
- Separação visual de seções de código (ex.: blocos de rotas públicas e privadas).

Estrutura:

Analise:

A estrutura foi refinada tanto no frontend quanto no backend:

- Frontend
    - Extração de ProtectedRoute e service.js para evitar repetição;

    - Componentes de página (LDT, NovaTarefa, Login) focados em UI.

- Backend:
    - Separação clara entre schemas, DTOs, services e controllers;
    - Isolamento da lógica de auto incremento (getNextSequence);
    - Filtro por usuário em todas as operações de tarefa.

  Isso reduz a complexidade e facilita a manutenção.

Comentários e Documentação:

Analise:

Comentários extensivos foram adicionados apenas em pontos estratégicos (por exemplo, em ProtectedRoute, serviços e métodos mais complexos). O restante do código ficou autoexplicativo através de nomes significativos e estruturas claras, evitando "comentário por comentar".

Boas Práticas:

Princípios Aplicados:

SOLID:

Exemplos:

- SRP: ProtectedRoute faz apenas a proteção de rota;
- Services (UsersService, TaskService) focados na regra de negócio;
- Inversão de dependência via @InjectModel (Nest injeta as dependências).

DRY:

Exemplos:

- Lógica de chamadas HTTP centralizada em service.js;
- getNextSequence utilizado para geração de IDs.

KISS:

Exemplos:

- Formulário de nova tarefa simplificado para os campos realmente usados;
- Fluxo de login + proteção de rotas mais direto e compreensível.

YAGNI:

Exemplos:

- Remoção de campos e funcionalidades não usadas no frontend;
- Remoção/reorganização de testes/arquivos que não agregam valor direto à regra de negócio atual.

5. Testes

OBJETIVO DOS TESTES

Os testes automatizados deste projeto foram desenvolvidos com o objetivo de garantir a confiabilidade e estabilidade do backend criado em NestJS.

Com eles, é possível assegurar que os principais fluxos da aplicação funcionem corretamente, evitando falhas, comportamentos inesperados e regressões após futuras alterações no código.

TECNOLOGIAS UTILIZADAS NOS TESTES

- Jest: Framework principal de testes do NestJS
- Supertest: Utilizado para testes envolvendo controllers HTTP (mockados)
- Mocks e Spies do Jest: Para simular dependências e comportamentos do banco
- MongoDB Mockado (sem necessidade de banco externo)
- MongoDB Memory Server (opcional para testes unitários sem dependência externa)

TIPOS DE TESTES REALIZADOS

- Testes unitários de serviço: garantindo que regras de negócio funcionem independente do controller ou banco.
- Testes de controllers: verificando chamadas corretas ao serviço, parâmetros e retornos.
- Testes de autenticação: validando fluxo de login, validação de senha, geração de token e proteção via JWT.
- Tratamento de erros: simulação de condições como:
    - ID inválido
    - entidade inexistente
    - e-mails duplicados
    - acesso não autorizado

Analise:

- Aumento significativo da qualidade e confiabilidade do código
- Redução do risco de bugs em produção
- Facilidade para atualização futura (refatoração segura)
- Código auto-documentado pelo comportamento testado
- Mocks de Mongoose garantem independência do banco real;

6. Conclusão

A refatoração do projeto A3 trouxe melhorias claras em legibilidade, estrutura de código, organização de responsabilidades e cobertura de testes. A migração do backend para MongoDB com Mongoose, somada à reorganização do frontend com rotas protegidas e serviços de API centralizados, tornou o sistema mais escalável, seguro e fácil de manter.