SISTEMA DE GERENCIAMENTO DE TAREFAS: To-Do List API

Adrian Gobara Falci,
Lorena Gobara Falci
Maria Vitória Garcia Pimenta

1 INTRODUÇÃO

Este documento apresenta a documentação técnica do Sistema de Gerenciamento de Tarefas desenvolvido como projeto final da disciplina de Desenvolvimento *Backend*. Trata-se de uma Interface de programação de aplicações (*Application Programming Interface* - API) REST que permite aos usuários criar contas, realizar login e gerenciar suas tarefas pessoais de forma organizada e segura.

O sistema foi desenvolvido utilizando Node.js e Express.js com TypeScript e implementa autenticação baseada em *tokens* JWT, garantindo que cada usuário tenha acesso exclusivo às suas próprias tarefas.

2 TECNOLOGIAS UTILIZADAS

2.1. BANCO DE DADOS

Para persistência de dados, foi utilizado o PostgreSQL como sistema gerenciador de banco de dados relacional. A comunicação com o banco é realizada através do TypeORM, um Mapeamento Objeto-Relacional (*Object-Relational Mapping* - ORM) que permite trabalhar com entidades JavaScript/TypeScript ao invés de *queries* SQL diretas, facilitando a manutenção e escalabilidade do código.

2.2 SEGURANÇA

Para a proteção da aplicação foram utilizadas duas bibliotecas principais, tanto para autenticação quanto para segurança dos dados:

- bcryptjs: Responsável por realizar o *hash* das senhas dos usuários antes de armazená-las no banco de dados.

- jsonwebtoken: Implementa o padrão *JSON Web Token* (JWT) para autenticação, gerando *tokens* com validade de 60 dias que identificam o usuário em cada requisição.

2.3 VALIDAÇÃO

O Zod foi implementado como biblioteca de validação de dados, criando *schemas* que definem exatamente quais campos são esperados em cada requisição e seus tipos. Isso garante que dados inválidos sejam rejeitados antes mesmo de chegarem à camada de negócio.

2.4 OUTRAS BIBLIOTECAS

- dotenv: Gerencia variáveis de ambiente.
- cors: Habilita requisições de diferentes origens (*Cross-Origin Resource Sharing*).
- socket.io: Implementa comunicação em tempo real via *WebSockets* para atualizações de projetos.

3 ARQUITETURA DO PROJETO

O projeto segue uma arquitetura em camadas bem definida, separando responsabilidades em diferentes diretórios:

```
BackendProject/
├── src/
│   ├── config/          # Configurações (data-source)
│   ├── controllers/     # Controladores das rotas
│   ├── entities/        # Entidades do TypeORM
│   ├── errors/          # Classes de erro customizadas
│   ├── middlewares/     # Middlewares (validação, autenticação, erros)
│   ├── migrations/      # Migrações do banco de dados
│   ├── routes/          # Definição de rotas
│   ├── schemas/         # Schemas de validação (Zod)
│   ├── services/        # Lógica de negócio
│   ├── sessions/        # Gerenciamento de sessões
│   ├── app.ts           # Configuração do Express
│   └── server.ts        # Inicialização do servidor
├── .env                 # Variáveis de ambiente
└── tsconfig.json
```

Quando uma requisição chega ao sistema, ela passa por diversas camadas, seguindo o seguinte fluxo:

- Express recebe a requisição e a direciona para a rota apropriada.
- *Middlewares* validam os dados e/ou o *token* de autenticação.
- *Controller* extrai informações da requisição.
- *Service* executa a lógica de negócio e se comunica com o banco.
- *Controller* retorna a resposta formatada ao cliente.

4 FUNCIONALIDADES IMPLEMENTADAS

4.1 SISTEMA DE AUTENTICAÇÃO

O sistema implementa um fluxo completo de autenticação e autorização. Novos usuários podem criar contas fornecendo um nome de usuário e senha. A senha é imediatamente transformada em *hash* usando bcrypt antes de ser armazenada, garantindo que mesmo com acesso ao banco de dados as senhas não possam ser descobertas.

Usuários registrados podem fazer login informando suas credenciais. O sistema valida o nome de usuário, compara a senha fornecida com o *hash* armazenado e, em caso de sucesso, gera um *token* JWT contendo o ID do usuário.

Todas as operações com tarefas requerem autenticação. O *middleware* validateToken intercepta as requisições, extrai o *token* do *header* Authorization, valida sua assinatura e expiração, e injeta a sessão do usuário na requisição.

4.2 GERENCIAMENTO DE TAREFAS

O *Create, Read, Update and Delete* (CRUD) completo de tarefas foi implementado com as seguintes funcionalidades:

- Criação: Usuários autenticados podem criar tarefas informando título, descrição e status de conclusão. O sistema valida que não existam tarefas duplicadas (mesmo título) para aquele usuário e vincula automaticamente a tarefa ao usuário logado.
- Listagem: Retorna todas as tarefas pertencentes ao usuário autenticado, exibindo ID, título, descrição e status de conclusão.
- Busca Individual: Permite recuperar uma tarefa específica por ID, validando que ela pertence ao usuário requisitante.

- Atualização: Usuários podem modificar título, descrição ou status de suas tarefas, com validação de propriedade.
- Exclusão: Remove permanentemente uma tarefa do sistema após validar que o usuário é o proprietário.

4.3 SEGURANÇA E VALIDAÇÃO

Todos os *endpoints* validam os dados recebidos usando *schemas* Zod, rejeitando automaticamente requisições mal-formatadas ou com campos inválidos. O sistema garante que usuários só possam acessar e modificar suas próprias tarefas, retornando erro 403 (*Forbidden*) em tentativas de acesso não autorizado.

Um *middleware* centralizado captura todos os erros da aplicação, classificando-os e retornando respostas apropriadas com códigos HTTP corretos e mensagens descritivas.

5 API *ENDPOINTS*

5.1 AUTENTICAÇÃO

POST /auth - Realiza login no sistema. Recebe *username* e *password* no corpo da requisição e retorna um *token* JWT em caso de sucesso. Retorna erro 401 se as credenciais forem inválidas.

5.2 USUÁRIOS

POST /users - Cria um novo usuário no sistema. Recebe” username” e “password”, valida se o nome de usuário já não existe, cria o hash da senha e persiste no banco. Retorna os dados do usuário criado (sem a senha).

5.3 TAREFAS (Autenticação Obrigatória)

Todos os *endpoints* de tarefas requerem o *header* Authorization: Bearer {token}.

POST /tasks - Cria uma nova tarefa vinculada ao usuário autenticado e valida duplicidade de título.

GET /tasks - Lista todas as tarefas do usuário autenticado.

GET /tasks/:id - Retorna uma tarefa específica por ID, validando propriedade.

PUT /tasks/:id - Atualiza uma tarefa existente (título, descrição ou status).

DELETE /tasks/:id - Remove uma tarefa do sistema.

6 MODELO DE DADOS

6.1 ESTRUTURA DO BANCO DE DADOS

O banco de dados relacional em PostgreSQL criado a partir das *migrations* do TypeORM possui duas tabelas principais:

- Tabela *user*: Armazena informações dos usuários com os campos *id* (UUID gerado automaticamente), *username* (*string* única) e *password* (*string* com *hash* bcrypt).
- Tabela *task*: Armazena as tarefas com os campos *id* (UUID), *title* (*string*), *description* (*string*), *done* (booleano com valor padrão *false*) e *userId* (chave estrangeira referenciando a tabela *user*).

6.2 RELACIONAMENTOS

Existe um relacionamento um-para-muitos entre *user* e *task*, onde um usuário pode ter várias tarefas. A configuração de *cascade delete* garante que ao deletar um usuário, todas as suas tarefas sejam removidas automaticamente.

7 CONFIGURAÇÃO E EXECUÇÃO

7.1 VARIÁVEIS DE AMBIENTE

O sistema requer um arquivo .env na raiz do projeto contendo credenciais do PostgreSQL, (*host*, porta, usuário, senha, *database*), porta da aplicação (padrão 3000) e chave secreta para assinatura dos *tokens* JWT.

7.2 INSTALAÇÃO E EXECUÇÃO

A instalação é realizada através do comando *npm install* que baixa todas as dependências. As *migrations* devem ser executadas com *npm run typeorm migration:run* para criar as tabelas no banco. O servidor é iniciado em modo desenvolvimento com *npm run dev*, que ativa *hot-reload* para facilitar o desenvolvimento.

8 RELATÓRIO DE PARTICIPAÇÃO

8.1 LORENA GOBARA FALCI (Github: @loregbrw)

Responsável pela configuração inicial e infraestrutura do projeto. Criou a estrutura de pastas, configurou o TypeScript e instalou as dependências necessárias. Implementou a configuração do *DataSource* para conexão com PostgreSQL, incluindo validações de variáveis de ambiente e *pool* de conexões.

Na camada de segurança, desenvolveu o sistema de autenticação, implementando o *endpoint* de criação de usuários com *hash* de senha usando bcrypt. Criou os três middlewares do sistema: validateBody para validação de dados com Zod, handleError para tratamento centralizado de erros, e validateToken para validação de *tokens* JWT.

Implementou também a classe UserSession que gerencia a sessão do usuário durante as requisições e o *endpoint* de login que valida credenciais e gera *tokens* JWT.

8.2 MARIA VITÓRIA GARCIA PIMENTA (Github: @mavigpimenta)

Responsável pelas funcionalidades de criação e exclusão de tarefas. Implementou o *endpoint* de criação de *task* com toda a lógica de validação de duplicidade, verificação de existência do usuário e vinculação automática da tarefa ao usuário autenticado.

Desenvolveu também o *endpoint* de deleção, implementando as regras de autorização que garantem que apenas o proprietário da tarefa possa excluí-la. Criou validações como verificação de existência da tarefa e controle de acesso baseado no ID do usuário.

8.3 ADRIAN GOBARA FALCI (Github: @adriansito124)

Responsável por completar o CRUD de tarefas implementando as operações de leitura e atualização. Desenvolveu o *endpoint* que retorna a lista de todas as tarefas do usuário autenticado, formatando adequadamente a resposta para omitir informações desnecessárias.

Implementou também o *endpoint* para busca de tarefas específicas pelo *id* com validação de propriedade, e o *endpoint PUT* que permite atualização de título, descrição e status de conclusão. Trabalhou na integração entre *controllers* e *services*, garantindo que as respostas da API seguissem um padrão consistente.

9 CONSIDERAÇÕES FINAIS

O desenvolvimento do projeto foi finalizado sem dificuldades significativas. A arquitetura em camadas bem definida e o uso do Git para controle de versão facilitaram o trabalho colaborativo. Cada desenvolvedor trabalhou em *branches* separadas, focando em suas responsabilidades específicas sem gerar conflitos.

A integração do código foi realizada através de *pull request*s, permitindo revisão e *merge* organizado das funcionalidades. A estrutura modular do projeto possibilitou que os três membros desenvolvessem simultaneamente sem dependências bloqueantes.

A aplicação implementa autenticação robusta, validação de dados, controle de acesso e tratamento de erros, seguindo boas práticas de desenvolvimento *backend*. A divisão de responsabilidades entre os membros da equipe foi bem equilibrada, com cada integrante contribuindo em áreas específicas que se complementaram para criar um sistema completo.

O uso de TypeScript, TypeORM e padrões bem definidos resultou em código organizado e de fácil manutenção.

Projeto desenvolvido para a disciplina de Desenvolvimento Backend.
Repositório: https://github.com/DebugoDev/BackendProject