

Victor A. Lanni – RM364305
Rômulo Teixeira – RM361516
Arthur Roberto – RM365020
Luan Marques – RM362725

BACKEND E QUALIDADE
Web API de Gestão de Postagens com Node.js e TypeScript

FIAP
São Paulo
2025

Sumário

Web API de Gestão de Postagens.....	3
Informações do Projeto.....	3
Tecnologias Utilizadas.....	4
Node.js + Express.....	4
Prisma ORM + PostgreSQL/SQLite.....	4
JWT + Middleware de Autorização.....	4
Zod + Helmet + CORS + Rate-Limiting.....	4
Docker e Docker Compose.....	5
GitHub Actions.....	5
Jest + Supertest.....	5
Environment.....	5
Testes.....	6
Ferramentas utilizadas.....	6
Cobertura de testes.....	6
Cobertura de testes da aplicação.....	7
GitHub Actions – Integração Contínua (CI/CD).....	8
Docker e Containerização.....	9
Estrutura de containerização.....	9
Estrutura de Pastas.....	10
Pastas.....	10
Prisma.....	10
prisma-e2e.....	10
src.....	10
Endpoints da API.....	11
Autenticação.....	11
POST /auth/login.....	11
Recursos de Postagens.....	12
GET /posts/search.....	12
GET /posts.....	13
POST /posts.....	13
GET /posts/{id}.....	14
PUT /posts/{id}.....	14
DELETE /posts/{id}.....	15
PUT /posts/disable/{id}.....	15
PUT /posts/enable/{id}.....	16

Web API de Gestão de Postagens

A Web API de Gestão de Postagens é uma API RESTful desenvolvida em Node.js com TypeScript e Prisma ORM, que tem como principal objetivo oferecer uma solução escalável e segura para a gestão de postagens educacionais.

O desafio proposto consiste em solucionar um problema real enfrentado por professores(as) da rede pública: a ausência de uma plataforma centralizada, prática e tecnológica para publicar conteúdos didáticos.

A aplicação proporciona uma plataforma dinâmica de blog educacional, onde professores(as) podem criar, editar e gerenciar postagens, enquanto alunos(as) podem acessá-las e realizar buscas de forma intuitiva.

Para garantir um sistema eficiente, seguro e testável, a API foi construída com boas práticas de arquitetura, testes automatizados, autenticação JWT, controle de permissões por perfil (Aluno e Professor) e deploy containerizado via Docker, além de automação CI/CD com GitHub Actions.

Informações do Projeto

Na raiz da branch **main** no GitHub, dentro da pasta **vídeo**, estão disponíveis a apresentação(em vídeo) e esta documentação versionada.

Github: https://github.com/victor-a-l-001/FIAP_TechChallenger_Fase_2

Deploy: <https://fiap-techchallenger-fase-2.onrender.com/api-docs/#/>

Tecnologias Utilizadas

As ferramentas que garantem performance, segurança e agilidade no desenvolvimento do nosso sistema.

Node.js + Express

- **Node.js** fornece um runtime JavaScript de alta performance, ideal para I/O assíncrono.
- **Express** é um micro-framework minimalista que organiza rotas e middlewares de forma modular, facilitando a escalabilidade do back-end.

Prisma ORM + PostgreSQL/SQLite

- Prisma atua como camada de mapeamento objeto-relacional (ORM), permitindo definir o schema do banco com TypeScript e gerar queries seguras e tipadas.
- PostgreSQL oferece um banco relacional robusto, com suporte a transações complexas; SQLite é uma opção leve para ambientes de teste ou protótipos.

JWT + Middleware de Autorização

- **JWT (JSON Web Tokens)** transporta informações de sessão de forma compacta e auto-contida, sem necessidade de estado no servidor.
- **Middlewares** verificam o token em cada requisição, extraem o perfil do usuário e condicionam o acesso a endpoints conforme roles/permissions.

Zod + Helmet + CORS + Rate-Limiting

- Zod faz a validação e parsing de dados de entrada, garantindo que body/params/query estejam no formato esperado.
- Helmet adiciona headers de segurança HTTP (Content Security Policy, X-Frame Options etc.).
- CORS configura quais origens podem acessar sua API, evitando requisições indevidas.
- Rate-Limiting limita número de requisições por IP/perfil para mitigar ataques de força bruta e DoS.

Docker e Docker Compose

- **Docker** empacota a aplicação em containers isolados, replicando o ambiente de produção localmente.
- **Docker Compose** orquestra múltiplos containers (app, banco, cache etc.) através de um único docker-compose.yml, simplificando o setup.

GitHub Actions

- Define **workflows** que rodam automaticamente ao submeter PRs, commits ou tags.
- Pode executar lint, testes, build e deploy contínuo, garantindo qualidade e agilidade nas entregas.

Jest + Supertest

- **Jest** é um test runner poderoso para JavaScript/TypeScript, com mocks, snapshots e relatórios de cobertura.
- **Supertest** facilita testes de endpoints HTTP, simulando requisições à API.
- **Meta mínima** de cobertura de **80%** garante que ao menos as principais rotas e funções estejam cobertas.

Environment

Para executar o projeto localmente, crie na raiz um arquivo .env com as propriedades abaixo (já configuradas nos secrets do GitHub e do Reader, portanto não é necessário versionar este arquivo):

Propriedades

DATABASE_URL="postgres://postgres:postgres@db:5432/postgres"

**** Obs.: Para executar o projeto fora do Docker, é necessário mudar para o db para localhost****

NODE_ENV="local"

HOST="localhost"

PORT=3000

JWT_SECRET="sua-chave-secreta"

Testes

Para garantir a estabilidade, a qualidade e a confiabilidade da aplicação, implementamos uma estratégia de testes sólida, composta por:

- Testes unitários, que validam cada função isoladamente
- Testes de integração (E2E), que verificam o comportamento de fluxos completos

Ferramentas utilizadas

- **Jest:** Framework de testes para criação e execução de testes unitários e de integração, com suporte a mocks, snapshots e relatórios de cobertura.
- **Supertest:** Biblioteca responsável por simula requisições HTTP para validação de endpoints e comportamentos da API em ambiente controlado.
- **GitHub Actions:** Pipeline de CI/CD que dispara automaticamente a suíte de testes a cada push ou pull request no repositório.

Os casos de uso da aplicação dispõem de sua própria suíte de testes, garantindo:

1. Validação isolada de funções, rotas e regras de negócio
2. Verificação integrada de todos os componentes que participam de um mesmo fluxo

Cobertura de testes

Alcançamos **100% de cobertura** nos seguintes critérios:

- Linhas de código
- Funções
- Condicionais (branches)
- Arquivos de teste

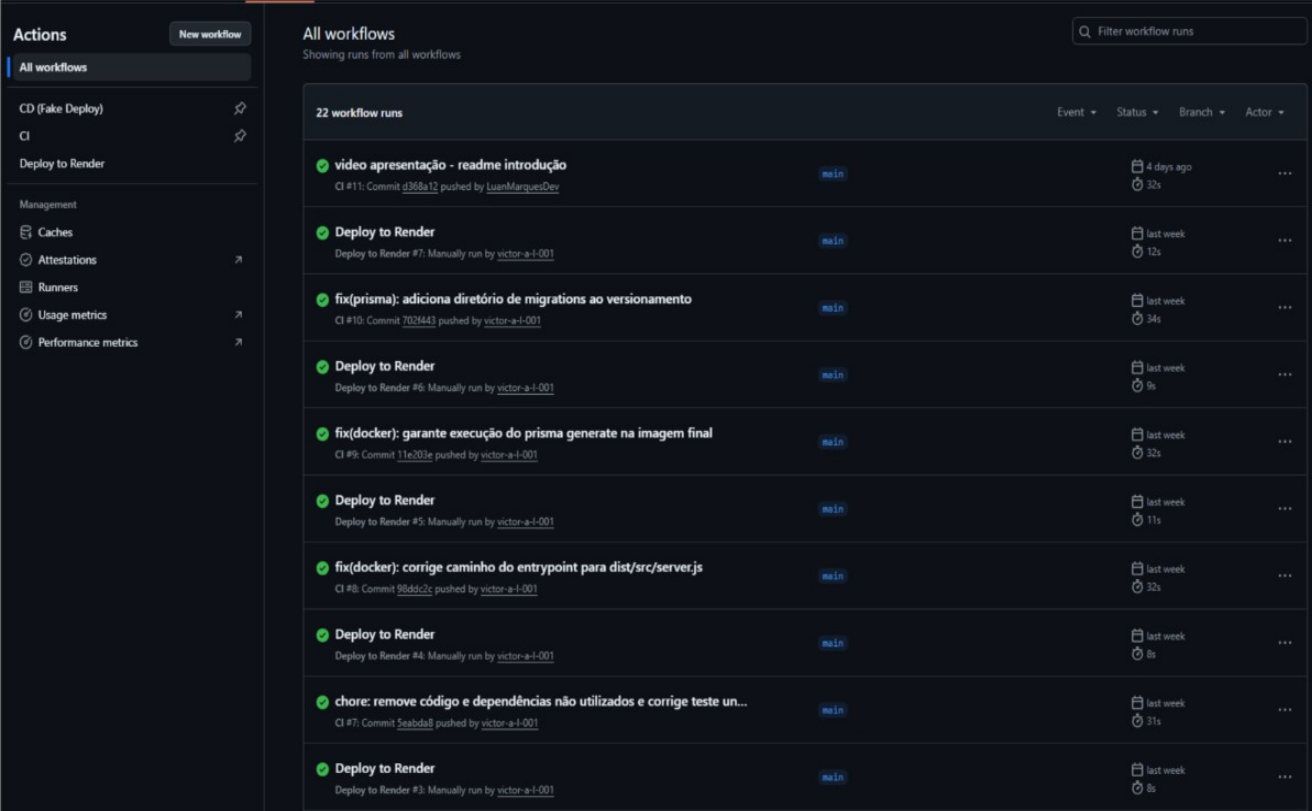
Cobertura de testes da aplicação

Para garantir o correto funcionamento dos principais componentes de regra de negócio e demais funcionalidades da aplicação, alcançamos 100% de cobertura de testes, permitindo identificar rapidamente falhas e assegurando maior escalabilidade e facilidade de manutenção a longo prazo.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	96.25	100	100	
src	100	80	100	100	4,29
src/controllers	100	95.83	100	100	
src/middlewares	100	100	100	100	
src/routes	100	100	100	100	
src/schemas	100	100	100	100	
src/types	100	100	100	100	
src/use-cases/post	100	100	100	100	
Test Suites: 18 passed, 18 total					
Tests: 104 passed, 104 total					

GitHub Actions – Integração Contínua (CI/CD)

A aplicação dispõe de pipelines de Integração Contínua (CI) implementadas via GitHub Actions, que automatizam o build, a execução de testes e a validação a cada push. Para o deploy, há integração com o Render, cujo trigger é acionado manualmente. Dessa forma, o código na branch principal está sempre verificado, testado e pronto para deployment em produção ou homologação, garantindo um fluxo de desenvolvimento mais ágil e confiável.



The screenshot displays the GitHub Actions 'All workflows' page. The left sidebar contains navigation links for 'Actions', 'All workflows', 'CD (Fake Deploy)', 'CI', 'Deploy to Render', and a 'Management' section with links to 'Caches', 'Attestations', 'Runners', 'Usage metrics', and 'Performance metrics'. The main area, titled 'All workflows', shows a list of 22 workflow runs. Each run entry includes a green status icon, a title, a commit hash and author, a branch name (main), a timestamp, a duration, and a menu icon. The runs are as follows:

Workflow Run	Status	Branch	Time	Duration
video apresentação - readme introdução	Success	main	4 days ago	32s
Deploy to Render	Success	main	last week	12s
fix(prisma): adiciona diretório de migrations ao versionamento	Success	main	last week	34s
Deploy to Render	Success	main	last week	9s
fix(docker): garante execução do prisma generate na imagem final	Success	main	last week	32s
Deploy to Render	Success	main	last week	11s
fix(docker): corrige caminho do endpoint para dist/src/server.js	Success	main	last week	32s
Deploy to Render	Success	main	last week	8s
chore: remove código e dependências não utilizados e corrige teste un...	Success	main	last week	33s
Deploy to Render	Success	main	last week	8s

Docker e Containerização

Para garantir consistência entre os ambientes de desenvolvimento, testes e produção, toda a aplicação foi empacotada em contêineres Docker. Isso elimina problemas de configuração e torna o projeto facilmente executável e portátil em qualquer máquina ou infraestrutura.

Estrutura de containerização

O projeto inclui dois arquivos principais:

- **Dockerfile**

Empacota a aplicação Node.js, seguindo estes passos:

- Instalação das dependências via `npm install` ou `yarn install`
- Geração do client Prisma (`prisma generate`)
- Compilação do código TypeScript (`tsc`)
- Definição do comando de inicialização da API (`node dist/server.js`)

- **Adaptação para deploy**

- Ajustes de build args e variáveis de ambiente para funcionar corretamente tanto no pipeline de deploy (Fake) quanto na plataforma Render.com.

- **docker-compose.yml**

Orquestra os serviços necessários, definindo dois containers:

- `api`: roda a aplicação Node.js, mapeando portas e variáveis de ambiente.
- `db`: executa o PostgreSQL, com volume dedicado para persistência dos dados.

Estrutura de Pastas

A organização do projeto foi estruturada com base em princípios de modularidade, coesão e separação de responsabilidades. Abaixo, segue a descrição das principais pastas e arquivos:

Pastas

- **Prisma**
 - Contém os arquivos de schema e migrações do banco de dados principal, utilizado em desenvolvimento e produção.
- **prisma-e2e**
 - Schema e configurações específicas para testes de integração (E2E), garantindo isolamento do banco de dados.

- **src**

Diretório principal do código-fonte da aplicação.

- **controllers**: Camada responsável por receber requisições HTTP e repassar para os casos de uso.
 - **domain**: Contém entidades do domínio.
 - **middlewares**: Funções intermediárias para autenticação, autorização e tratamento de erros.
 - **repositories**: Acesso e manipulação de dados no banco (CRUD).
 - **responses**: Formatos de resposta padronizados da API.
 - **routes**: Definição dos endpoints e associação com os controllers.
 - **schemas**: Validações de payloads (por exemplo, Zod).
 - **types**: Definições de tipos TypeScript compartilhados
 - **use-cases**: Lógica de negócio encapsulada em serviços (casos de uso).
- **tests**

Estrutura de testes automatizados.

- **e2e**: Testes ponta-a-ponta simulando chamadas reais à API.
 - **factories**: Fábricas de dados para uso nos testes E2E.
 - **unit**: Testes unitários organizados por módulo da aplicação (controllers, middlewares, routes, schemas, use-cases).

Endpoints da API

A seguir estão descritos os principais endpoints RESTful da aplicação, responsáveis pelas operações de criação, leitura, atualização, busca e exclusão de postagens. Todos os endpoints protegidos utilizam autenticação via JWT e diferenciam permissões de acesso por perfil (Aluno ou Professor).

Autenticação

POST /auth/login

Descrição: Realiza o login do usuário e retorna um token JWT para autenticação.

Perfil de Acesso: Professor e Aluno

Body

Campo	Tipo	Obrigatório	Descrição
email	string (email)	sim	E-mail do usuário
password	string	sim	Senha (mínimo 6 caracteres)

JSON – Request

```
{
  "email": "usuario@nulo.com",
  "password": "secret123"
}
```

Responses

Código	Descrição	Schema
200	Token retornado com sucesso	TokenResponse
400	Erro de validação (dados inválidos)	ApiError
401	Credenciais inválidas ou token malformado	ApiError

Exemplo 200 OK

```
{
  "token": "eyJhbGciOi..."
}
```

Recursos de Postagens

GET /posts/search

Descrição: Busca postagens cujo título ou conteúdo contenham o termo informado.

Perfil de Acesso: Professor e Aluno

Parâmetros de Path

Parâmetro	Tipo	Obrigatório	Descrição
q	string	sim	Termo de busca

Response

Código	Descrição	Schema
200	Resultado da busca	PostResponse[]
400	Erro de validação	—
401	Credenciais inválidas	—
403	Perfil de acesso não autorizado	—

Exemplo 200 OK

```
[
{
  "id": 1,
  "title": "Título Exemplo",
  "content": "Conteúdo Exemplo.",
  "authorId": 1,
  "createdAt": "2025-07-09T12:34:56Z",
  "updatedAt": "2025-07-10T08:00:00Z",
  "author": {
    "id": 1,
    "name": "Author",
    "email": "author@nulo.com"
  }
}
]
```

GET /posts

Descrição: Lista todas as postagens.

Perfil de Acesso: Professor e Aluno

Responses

Código	Descrição	Schema
200	Lista de postagens	PostResponse[]
401	Credenciais inválidas	—
403	Perfil de acesso não autorizado	—

POST /posts

Descrição: Cria uma nova postagem.

Perfil de Acesso: Professor

Request

Campo	Tipo	Obrigatório	Descrição
title	string	sim	Título da postagem
content	string	sim	Conteúdo da postagem
authorId	integer	sim	ID do autor existente

JSON – Request

```
{
  "title": "Titulo Exemplo",
  "content": "Conteúdo Exemplo.",
  "authorId": 1
}
```

Response

Código	Descrição	Schema
201	Postagem criada com sucesso	PostResponse
400	Erro de validação	—
401	Credenciais inválidas	—
403	Perfil de acesso não autorizado	—

GET /posts/{id}

Descrição: Obtém uma única postagem pelo seu ID.

Perfil de Acesso: Professor e Aluno

Parâmetros de Path

Parâmetro	Tipo	Obrigatório	Descrição
id	integer	sim	ID da postagem

Responses

Código	Descrição	Schema
200	Postagem encontrada	PostResponse
401	Credenciais inválidas	—
403	Perfil de acesso não autorizado	—
404	Postagem não encontrada	—

PUT /posts/{id}

Descrição: Atualiza os campos de uma postagem existente.

Perfil de Acesso: Professor

Parâmetros de Path

Parâmetro	Tipo	Obrigatório	Descrição
id	integer	sim	ID da postagem

Request

Campo	Tipo	Obrigatório	Descrição
title	string	não	Novo título
content	string	não	Novo conteúdo

JSON – Request

```
{
  "title": "Título Atualizado",
  "content": "Conteúdo atualizado."
}
```

Responses

Código	Descrição	Schema
200	Postagem atualizada com sucesso	PostResponse
400	Erro de validação	—
401	Credenciais inválidas	—
403	Perfil de acesso não autorizado	—
404	Postagem não encontrada	—

DELETE /posts/{id}

Descrição: Remove permanentemente uma postagem.

Perfil de Acesso: Professor

Parâmetros de Path

Parâmetro	Tipo	Obrigatório	Descrição
id	integer	sim	ID da postagem

Responses

Código	Descrição
204	Postagem removida com sucesso
401	Credenciais inválidas
403	Perfil de acesso não autorizado
404	Postagem não encontrada

PUT /posts/disable/{id}

Descrição: Desabilita (inativa) uma postagem.

Perfil de Acesso: Professor

Parâmetros de Path

Parâmetro	Tipo	Obrigatório	Descrição
id	integer	sim	ID da postagem

Responses

Código	Descrição
204	Postagem desabilitada com sucesso
401	Credenciais inválidas
403	Perfil de acesso não autorizado
404	Postagem não encontrada

PUT/posts/enable/{id}

Descrição: Habilita (ativa) uma postagem anteriormente desabilitada.

Perfil de Acesso: Professor

Parâmetros de Path

Parâmetro	Tipo	Obrigatório	Descrição
id	integer	sim	ID da postagem

Responses

Código	Descrição
204	Postagem habilitada com sucesso
401	Credenciais inválidas
403	Perfil de acesso não autorizado
404	Postagem não encontrada