Visão Geral

O sistema tem como objetivo fornecer uma API RESTful para gerenciamento de tarefas colaborativas. Usuários podem ser cadastrados com nome, login e senha, e cada um pode ser responsável por uma ou mais tarefas.

As tarefas possuem nome, conteúdo, um usuário responsável, e campos para indicar se estão ativas ou finalizadas. A API permitirá criar, editar, atribuir, listar e concluir tarefas, além de ativar ou desativar usuários e tarefas.

A autenticação será feita via login e senha, com uso de tokens JWT para proteger os endpoints. Apenas usuários ativos poderão interagir com o sistema. Essa API servirá como base para aplicações web ou mobile que exigem organização e controle de atividades em equipe.

Decisões arquiteturais

Foi adotada a Clean Architecture como base para a organização do projeto, com o objetivo de garantir uma separação clara de responsabilidades entre as camadas da aplicação. Essa abordagem permite maior facilidade de manutenção, testabilidade e escalabilidade do sistema.

A aplicação foi dividida em camadas principais:

- Controller: responsável por receber as requisições HTTP e direcioná-las para a camada de caso de uso.
- Usecase: contém as regras de negócio e coordena o fluxo entre controladores e repositórios.
- Repository: implementa o acesso ao banco de dados, isolando a lógica de persistência.
- Model/Entity: define as estruturas de dados principais, como usuário e tarefa.

Além disso, foi adotado o padrão Repository para abstrair a comunicação com o banco de dados e o padrão DTO para separar os dados usados na comunicação externa (respostas da API).

A injeção de dependência é feita manualmente no ponto de entrada da aplicação (main.go), facilitando testes e a troca de implementações.

Essa arquitetura foi escolhida por sua flexibilidade e independência de frameworks, o que facilita futuras mudanças, como trocar o banco de dados ou a interface da aplicação sem impactar a lógica central do sistema.

Modelagem de Dados

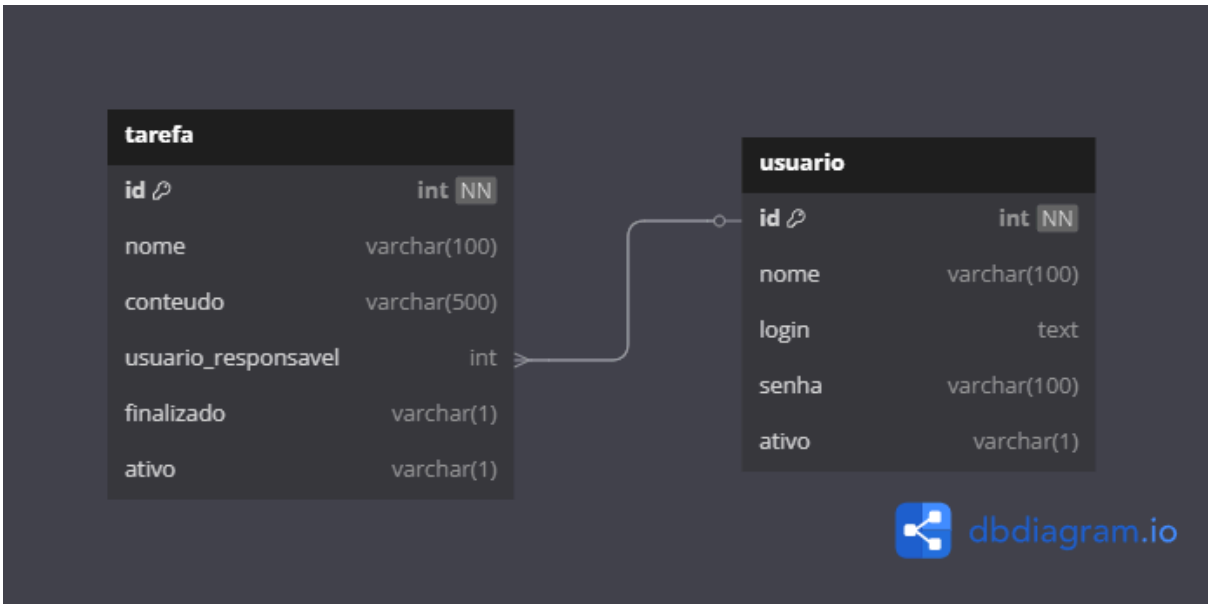

Tabela usuário

Campo	Tipo	Descrição
id	Int(Pk)	Identificador único do usuário.
nome	varchar(100)	Nome do usuário.
login	text	Nome de login usado para autenticação.
senha	varchar(100)	Senha do usuário (preferencialmente armazenada com hash).
ativo	varchar(1)	Indica se o usuário está ativo (A) ou inativo (I).

Tabela tarefa

Campo	Tipo	Descrição
id	int (PK)	Identificador único da tarefa.
nome	varchar(100)	Nome ou título da tarefa.
conteudo	varchar(500)	Descrição ou detalhes da tarefa.
usuario_responsavel	int (FK)	ID do usuário responsável (relacionado à tabela usuario).
finalizado	varchar(1)	Indica se a tarefa foi concluída (S) ou não (N).
ativo	varchar(1)	Indica se a tarefa está ativa (A) ou inativa (I).

Relacionamento:

- Uma tarefa pode ter um usuário responsável.
- A chave estrangeira usuario_responsavel referencia usuario.id.

Fluxo de Requisições

<link do swagger>

Configuração e Deploy

Tecnologias Utilizadas

- Linguagem: Go (Golang)
- Framework Web: Gin
- Banco de Dados: MySQL 8.0
- Gerenciamento de dependências: Go Modules (go.mod)
- Gerenciamento de containers: Docker + Docker Compose

Pré-requisitos

- Docker e Docker Compose instalados
- Go instalado (versão recomendada: 1.20 ou superior)

Instalação e Execução Local

1. Clone o repositório

    <git clone <seu-repositorio>
    <cd <pasta-do-projeto>

2.Suba o banco de dados com Docker

    <docker-compose up -d>

3.(Opcional) Configure variáveis de ambiente

    Atualmente, as credenciais estão fixas no arquivo db/connection.go, mas o ideal seria movê-las para variáveis de ambiente.

4.Instale as dependências Go

    <go mod tidy>

5.Execute o servidor

    <go run main.go>

O servidor será iniciado em:
http://localhost:8000

Serviço	Porta Local
API(gin)	8000
MySQL (docker)	3306

Testes Automatizados

Tipo de Teste: Testes Unitários com Mock

A estratégia adotada no projeto é baseada em **testes unitários**. Cada teste é focado em **unidades isoladas do sistema**, principalmente os controladores (handlers da API), verificando seu comportamento **independentemente do banco de dados real**.

Para isso, foi utilizada a biblioteca sqlmock, que permite simular respostas do banco de dados sem a necessidade de conexão com um ambiente externo. Isso torna os testes mais rápidos, confiáveis e reproduzíveis.

Por que testes unitários?

- **Isolamento:** testam apenas a lógica da aplicação, sem depender de serviços externos.
- **Rapidez:** executam em milissegundos.
- **Detecção precoce de erros:** ajudam a validar regras de negócio antes da integração.
- **Facilidade de manutenção:** tornam o código mais modular e testável.

Estratégia Utilizada

Os testes unitários foram aplicados diretamente sobre os **endpoints da API**, simulando requisições HTTP com httptest, validando os comportamentos esperados em cada cenário.

Cada rota foi testada com:

- Dados válidos simulados
- Respostas esperadas do banco via mock
- Verificação de código de status HTTP
- Assertivas sobre comportamento correto dos controladores

Bibliotecas utilizadas:

- testing: framework de testes nativo do Go

- sqlmock: simulação de banco de dados
- httptest: simulação de requisições HTTP
- testify/assert: assertivas mais legíveis