API TrabEngSoftwareApplication

Introdução

A aplicação é um sistema de gerenciamento de tarefas colaborativas, com foco em atribuição de responsabilidades, controle de prazos e comentários colaborativos. Ela oferece funcionalidades voltadas para três domínios principais: usuários, tarefas e comentários.

Funcionalidades

Gestão de Usuários

- Permite criação, busca por nome ou ID, atualização e remoção de usuários.
- Cada usuário pode ser responsável por uma ou mais tarefas.
- Suporte para identificação de usuários por nome ou ID, facilitando buscas dinâmicas.

Gestão de Tarefas

- Permite criar tarefas com título, descrição, prioridade, status, data de entrega e responsável.
- Suporta edição de tarefas existentes e remoção quando necessário.
- Permite consultar tarefas por ID, ou listar tarefas atribuídas a um usuário específico.
- Suporte para filtros avançados de busca, como status, prioridade e prazo de entrega (due date), permitindo refinar os resultados conforme critérios de gerenciamento.

Comentários em Tarefas

- Permite que usuários adicionem comentários em tarefas específicas.
- Suporte para listar todos os comentários relacionados a uma tarefa.
- Funcionalidade de remoção de comentários, mantendo o controle sobre a comunicação interna da tarefa.

Tecnologias Utilizadas

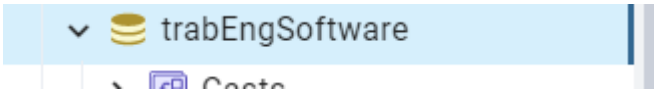
Foi decidido utilizar o **Java Spring Boot** para montar a API com CRUD, em razão de experiências prévias e, especialmente, a facilidade em trabalhar com queries do banco através do JPARepository, que apenas com o nome do método, já permite fazer queries bastante complexas no banco.

Para o banco de dados utilizou-se **PostgreSQL** , sendo a conexão feita através da depêndencia do pom.xml postgresql. Através da file application.yml, pode-se configurar os dados relacionados ao banco, como a URL de acesso e o usuário e senha utilizados no pgadmin

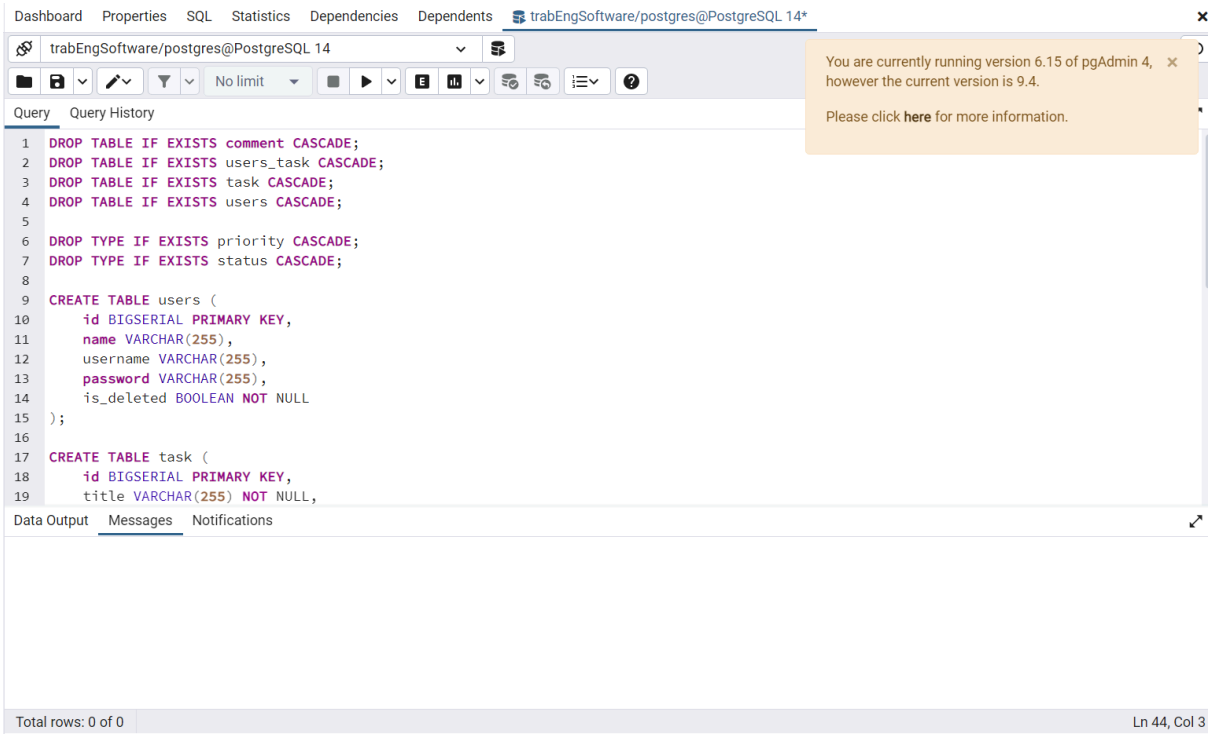
.

```
datasource:
  url: jdbc:postgresql://localhost:5432/trabEngSoftware
  username: username
  password: password
```

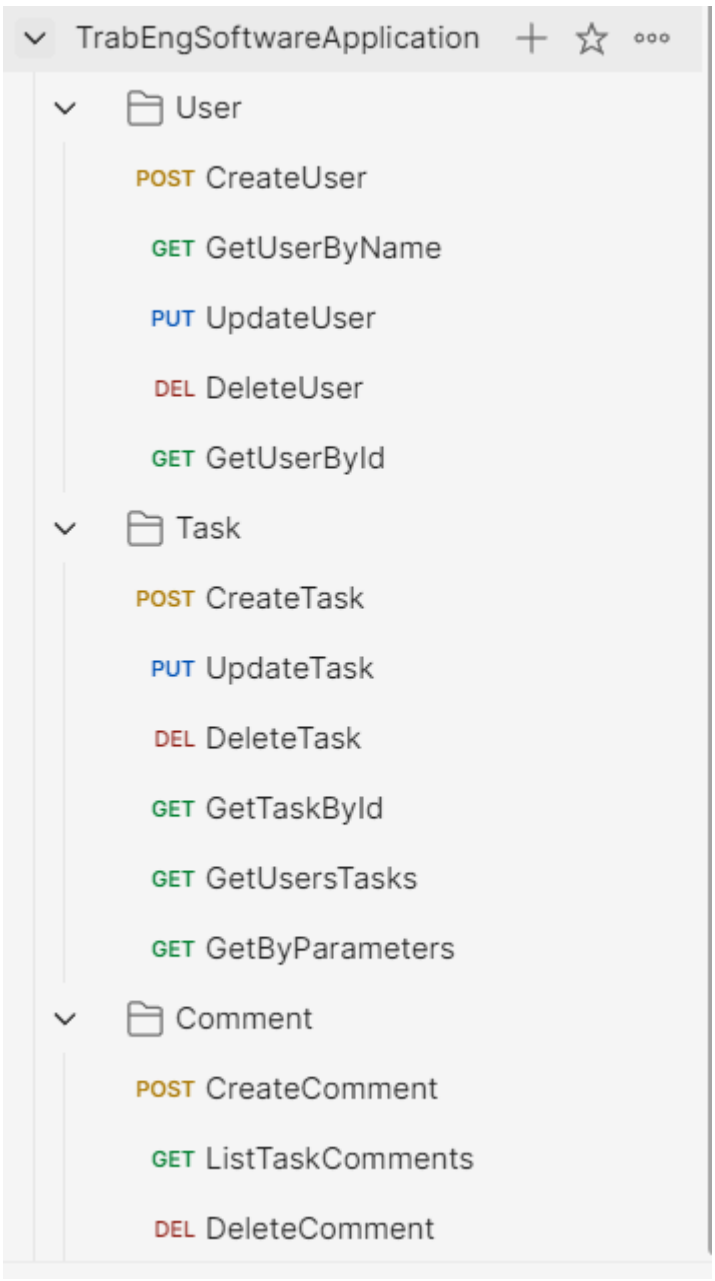
Variáveis de usuário e senha devem ser modificadas de acordo com as utilizadas pelo usuário em seu próprio computador. O banco foi criado localmente, utilizando o pgadmin. Deve-se criar uma base de dados com o mesmo nome colocado na URL.



Lá, deve-se utilizar a query de criação do banco, que conforme feito foi colocada em schema.sql na pasta data.

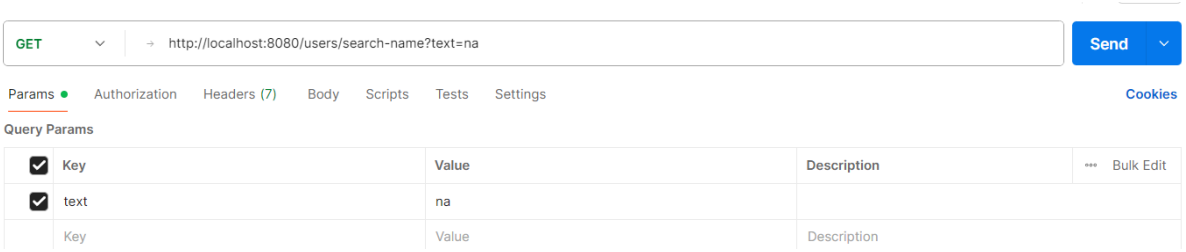


Para consumo da API, foi utilizado o aplicativo **Postman**, que permite o consumo dos Endpoints e uma interface para organizá-los. O arquivo JSON com a lógica utilizada no Postman pode ser encontrado no folder data file postman.json

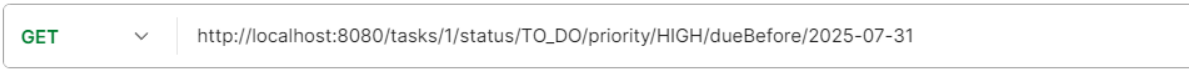


Nas que têm body pode-se utilizar a aba “Body” utilizando a opção raw em JSON.

Nas que possuem a mudança através do comando “?” da URL pode-se usar o PARAMS



Por fim, nas que possuem mudanças diretas na URL, deve-se modificar o nome.



Dependências Spring no Projeto

spring-boot-starter-web

- **Finalidade:** Criação de aplicações web com suporte a REST.
- **Inclui:** Spring MVC, Tomcat embutido, Jackson (para JSON).
- **Uso:** Controladores (@RestController), mapeamentos de rota (@RequestMapping, @GetMapping etc.).

spring-boot-starter-data-jpa

- **Finalidade:** Integração com bancos relacionais usando Spring Data e JPA (Hibernate por padrão).
- **Uso:** Repositórios como UserRepository, TaskRepository, CommentRepository com métodos como save(), findById().

spring-boot-starter-validation

- **Finalidade:** Suporte a validações com Bean Validation (Hibernate Validator).
- **Uso:** Anotações como @NotNull, @NotBlank, @Valid em DTOs e requisições.

spring-boot-starter-actuator

- **Finalidade:** Expõe endpoints para monitoramento e métricas da aplicação (como /actuator/health).

- **Uso:** Ajudar na observabilidade da aplicação em produção.

postgresql

- **Finalidade:** Driver JDBC para conexão com banco de dados PostgreSQL.

- **Escopo:** runtime – necessário apenas em tempo de execução.

lombok

- **Finalidade:** Geração automática de código como getters, setters, toString(), @Builder, etc.

- **Uso:** Reduz boilerplate nas classes de modelo e DTOs.

- **Requer IDE plugin** para funcionar corretamente durante o desenvolvimento.

spring-boot-starter-test

- **Finalidade:** Suporte completo para testes com JUnit, Mockito, Hamcrest, etc.

- **Uso:** Criação de testes unitários e de integração (@SpringBootTest, @WebMvcTest, @MockBean).

Testes Unitários

Ferramentas Utilizadas

- JUnit 5: Framework de testes utilizado para estruturar os testes (@Test).

- Mockito: Biblioteca para simular (mockar) dependências e controlar o comportamento de objetos.

- Mockito JUnit Extension: Ativada via @ExtendWith(MockitoExtension.class) para habilitar injeção de mocks.

- ArgumentCaptor: Utilizado para capturar e inspecionar os argumentos passados a métodos como save().

Estrutura dos Testes

- Cada teste é feito para um cenário específico, validando:
    - Operações bem-sucedidas.
    - Manipulação correta das entidades.
    - Interações entre serviços e repositórios.
- Os repositórios (UserRepository, TaskRepository, etc.) são mockados com @Mock, evitando dependência do banco real.
- A classe de serviço que está sendo testada (ex: CommentService) é anotada com @InjectMocks para que os mocks sejam injetados automaticamente.

Padrão dos Testes

1. Setup dos dados (ex: instanciar Comment, Task, Users).
2. Simulação de comportamento com when(...).thenReturn(...).
3. Chamada do método real da service (tested.metodo()).
4. Verificação:
    - Asserções com assertEquals, assertNotNull, assertTrue.
    - Verificação de chamadas com verify(...).
    - Captura e inspeção do objeto persistido com ArgumentCaptor.

Além de conhecimento prévio, foram utilizadas essas abordagens por se tratar de uma API com bastante contato com tabelas de banco de dados, exigindo mocks.

Arquitetura do Sistema

A arquitetura que mais se aproxima da feita é a MVC. O padrão MVC (Model-View-Controller) é uma arquitetura de software que organiza a estrutura de uma aplicação em três camadas principais: Model, View e Controller. Seu principal objetivo é separar as responsabilidades de manipulação de dados, lógica de

apresentação e controle de fluxo, tornando o código mais modular, reutilizável e fácil de manter.

Model

- Representa as entidades de domínio da aplicação, como User, Task e Comment.
- Utiliza JPA (Hibernate) para persistência no banco PostgreSQL.
- Contém atributos do banco de dados e relacionamentos (@ManyToOne, @OneToMany).

Controller

- Expõe os endpoints RESTful (como /users, /tasks, /comments).
- Recebe requisições HTTP, valida os dados de entrada e delega para a camada de serviço.
- Utiliza anotações como @RestController, @RequestMapping, @GetMapping, @PostMapping.

Service

- Centraliza a lógica de negócio da aplicação.
- Lida com regras como: verificação de existência de entidades, tratamento de erros (ResponseStatusException), controle de responsáveis e comentários.
- Interage com os repositórios para acessar o banco.
- Utiliza mappers para converter DTOs em entidades e vice-versa.

Repository

- Interfaces como UserRepository, TaskRepository, CommentRepository.
- Usam Spring Data JPA para fornecer métodos prontos (findById, save, etc.).

- Permitem a criação de consultas personalizadas por nome de método (findByTaskId, findByUsernameContaining, etc.).

DTO (Data Transfer Objects)

- Objetos de transporte de dados para entrada (CreateTaskRequest, CreateCommentRequest) e saída (TaskResponse, CommentResponse, UserResponse).

- Validados com @Valid, @NotBlank, @NotNull, garantindo integridade dos dados da API.

Mapper

- Responsáveis pela conversão entre entidades (Model) e DTOs (Request/Response).

Endpoints da API TrabEngSoftwareApplication

UserController

1. Criar Usuário

- **POST** `/users`

- **Body (JSON):**

```
{
  "name": "name",
  "username": "username",
  "password": "password"
}
```

- **Descrição:** Cria um novo usuário no sistema.

---

2. Buscar Usuário por Nome

- **GET** `/users/search-name?text=User`
- **Query Params:**
    - `text` — texto para busca por nome.
- **Descrição:** Retorna uma lista de usuários cujo nome contenha o texto especificado.

---

3. Buscar Usuário por ID

- **GET** `/users/{id}`
- **Descrição:** Retorna os dados de um usuário específico.

---

4. Atualizar Usuário

- **PUT** `/users/{id}`
- **Body (JSON):**

```
{
  "name": "Novo Nome",
  "username": "novouser",
  "password": "novasenha"
}
```

- **Descrição:** Atualiza os dados de um usuário específico.

---

5. Remover Usuário

- **DELETE** `/users/{id}`
- **Descrição:** Remove um usuário do sistema.

TaskController

1. Criar Tarefa

- **POST** `/tasks`
- **Body (JSON):**

```
{
  "title": "Finalizar relatório",
  "description": "Preparar e revisar o relatório anual da empresa",
  "priority": "MEDIUM",
  "status": "TO_DO",
  "responsibleId": 1,
  "dueDate": "2025-07-10"
}
```

- **Descrição:** Cria uma nova tarefa e associa um responsável.

---

2. Atualizar Tarefa

- **PUT** `/tasks/{id}`
- **Body (JSON):**

```
{
  "title": "Finalizar relatório Urgente",
  "description": "Preparar e revisar o relatório anual da empresa",
  "priority": "HIGH",
  "status": "TO_DO",
  "dueDate": "2025-07-11"
}
```

- **Descrição:** Atualiza os dados de uma tarefa existente.

---

3. Deletar Tarefa

- **DELETE** `/tasks/{id}`
- **Descrição:** Remove uma tarefa do sistema.

---

4. Buscar Tarefa por ID

- **GET** `/tasks/{id}`
- **Descrição:** Retorna os dados de uma tarefa específica.

---

5. Listar Tarefas por Responsável

- **GET** `/tasks/assigned-to/{userId}`
- **Descrição:** Retorna as tarefas atribuídas a um determinado usuário.

---

6. Buscar por Parâmetros (Filtros)

- **GET** `/tasks/{userId}/status/{status}/priority/{priority}/dueBefore/{date}`
- **Exemplo:** `/tasks/1/status/OPEN/priority/HIGH/dueBefore/2025-07-31`
- **Descrição:** Retorna tarefas atribuídas a um usuário, filtrando por status, prioridade e data limite.

CommentController

1. Criar Comentário

- **POST** `/tasks/{taskId}/comments`
- **Body (JSON):**

```
{
  "userId": 1,
  "content": "Esse é o meu comentário sobre a tarefa."
}
```

- **Descrição:** Adiciona um comentário à tarefa especificada.

---

2. Listar Comentários de uma Tarefa

- **GET** `/tasks/{taskId}/comments`
- **Descrição:** Retorna todos os comentários de uma tarefa.

---

3. Deletar Comentário

- **DELETE** `/tasks/{taskId}/comments/{commentId}`
- **Descrição:** Remove um comentário específico de uma tarefa.