INSTITUTO FEDERAL DO PIAUI - Campus Teresina Central
Curso de Técnico em Desenvolvimento de Sistemas
Programação Web Backend, Prof. Rogério Silva, **TDS 386 – 2025.1**
Atividade Prática Laboratório @ 26/Maio/2025

API Completa com FastAPI e SQLite

1. Visão Geral

Esta atividade prática tem como objetivo consolidar os conhecimentos em desenvolvimento de APIs RESTful utilizando FastAPI em Python. Implemente uma API completa, abordando desde a criação de endpoints básicos até a integração com banco de dados SQLite e a implementação de mecanismos robustos de autenticação e autorização.

2. Objetivos de Aprendizagem

Ao concluir essa atividade você está provando as seguintes habilidades e aprendizados:

- Criar e configurar projetos FastAPI.
- Definir endpoints para diferentes tipos de requisições (GET, POST, PUT, DELETE).
- Trabalhar com parâmetros de rota e query string.
- Enviar e processar corpos de requisição (JSON).
- Integrar a API com um banco de dados SQLite.
- Implementar um sistema de autenticação e autorização utilizando JWT (JSON Web Tokens).
- Gerenciar o cadastro e login de usuários.
- Implementar funcionalidades de recuperação de senha.
- Utilizar tokens de refresh para segurança aprimorada.
- Organizar o código em módulos (rotas, repositórios, utilidades, models).
- Testar a API utilizando ferramentas como Insomnia ou Postman.
- Realizar o deploy da aplicação utilizando Docker e a plataforma Render.

3. Cenário da Aplicação (Exemplo: API de Tarefas Simples)

API para gerenciar tarefas simples.

Modelos:

- **Usuário:**

    - id (inteiro, chave primária)
    - username (string, único)
    - email (string, único)
    - password_hash (string)
- **Tarefa:**
    - id (inteiro, chave primária)
    - titulo (string)
    - descricao (string, opcional)
    - concluida (booleano, padrão False)
    - usuario_id (inteiro, chave estrangeira para Usuário)

4. Requisitos Funcionais e Endpoints

4.1. Endpoints Básicos (GET, POST, PUT, DELETE)

- **GET /tasks**:
    - **Descrição:** Retorna uma lista de todas as tarefas.
    - **Autenticação:** Requer autenticação. Retorna apenas as tarefas do usuário autenticado.
- **GET /tasks/{task_id}**:
    - **Descrição:** Retorna os detalhes de uma tarefa específica pelo seu ID.
    - **Autenticação:** Requer autenticação. A tarefa deve pertencer ao usuário autenticado.
- **POST /tasks**:
    - **Descrição:** Cria uma nova tarefa.
    - **Autenticação:** Requer autenticação. A tarefa será associada ao usuário autenticado.
    - **Corpo da Requisição (JSON):**
      ```
      {
        "titulo": "Comprar mantimentos",
        "descricao": "Leite, pão, ovos"
      }
      ```

- **PUT /tasks/{task_id}**:
    - **Descrição:** Atualiza uma tarefa existente pelo seu ID.
    - **Autenticação:** Requer autenticação. A tarefa deve pertencer ao usuário autenticado.
    - **Corpo da Requisição (JSON):**
      ```
      {
        "titulo": "Comprar mantimentos",
        "concluida": true
      ```

```
}
```

- **DELETE /tasks/{task_id}**:
  - **Descrição:** Deleta uma tarefa pelo seu ID.
  - **Autenticação:** Requer autenticação. A tarefa deve pertencer ao usuário autenticado.

4.2. Parâmetros de Rota e Query String

- **Parâmetro de Rota:** O {task_id} nos endpoints GET /tasks/{task_id}, PUT /tasks/{task_id} e DELETE /tasks/{task_id} é um exemplo de parâmetro de rota.
- **Parâmetros de Query:**
  - **GET /tasks?concluida=true**: Retorna apenas as tarefas concluídas.
  - **GET /tasks?limit=10&offset=0**: Implementa paginação, retornando um número limit de tarefas a partir de um offset.

4.3. Envio de Corpo em Requisições POST/PUT

- A criação (POST /tasks) e atualização (PUT /tasks/{task_id}) de tarefas exigirão o envio de um corpo JSON com os dados da tarefa, conforme exemplificado acima.

4.4. Integração com Banco de Dados SQLite

- A aplicação deverá utilizar SQLite como banco de dados.
- As tabelas users e tasks deverão ser criadas automaticamente (se não existirem) na inicialização da aplicação.
- Todas as operações CRUD (Criar, Ler, Atualizar, Deletar) para usuários e tarefas deverão interagir com o banco de dados.

4.5. Autenticação e Autorização (JWT)

- **Cadastro (Signup):**
  - **POST /auth/signup**:
    - **Descrição:** Registra um novo usuário.
    - **Corpo da Requisição (JSON):**

```
{
  "username": "novo_usuario",
  "email": "email@example.com",
  "password": "senha_segura123"
}
```

    - **Regras:** A senha deve ser *hashed* (e.g., usando bcrypt via passlib) antes de ser armazenada no banco de dados.
- **Login (JWT):**

- **POST /auth/login**:
  - **Descrição:** Autentica um usuário existente.
  - **Corpo da Requisição (JSON):**

```
{
  "username": "novo_usuario",
  "password": "senha_segura123"
}
```

  - **Resposta (JSON):** Em caso de sucesso, retorna um access_token (JWT) e um refresh_token.

```
{
  "access_token": "eyJhbGciOiJIUzI1Ni...",
  "token_type": "bearer",
  "refresh_token": "eyJhbGciOiJIUzI1Ni..."
}
```

- **Rota "Me":**
  - **GET /auth/me**:
    - **Descrição:** Retorna as informações do usuário autenticado (e.g., username, email).
    - **Autenticação:** Requer o access_token no cabeçalho Authorization (formato Bearer <access_token>).

4.6. Desafios Adicionais

- **Recuperação de Senha:**
  - **POST /auth/forgot-password**:
    - **Descrição:** Inicia o processo de recuperação de senha.
    - **Corpo da Requisição (JSON):**

```
{
  "email": "email@example.com"
}
```

    - **Regras:** Simule o envio de um e-mail com um token de recuperação (por exemplo, um token temporário armazenado no DB com validade curta e associado ao usuário).
  - **POST /auth/reset-password**:
    - **Descrição:** Redefine a senha do usuário.
    - **Corpo da Requisição (JSON):**

```
{
```

```
"token_recuperacao": "token_gerado_anteriormente",
"nova_senha": "nova_senha_muito_segura"
}
```

- **Regras:** Valida o token de recuperação e atualiza a senha do usuário (com hashing).

- **Uso de Token de Refresh:**
  - **POST /auth/refresh**:
    - **Descrição:** Gera um novo access_token a partir de um refresh_token válido.
    - **Corpo da Requisição (JSON):**
      ```
      {
      "refresh_token": "eyJhbGciOiJIUzI1Ni..."
      }
      ```
    - **Resposta (JSON):**
      ```
      {
      "access_token": "eyJhbGciOiJIUzI1Ni...",
      "token_type": "bearer"
      }
      ```
    - **Regras:** O refresh_token deve ser validado e, se válido, um novo access_token é emitido.

5. Estrutura do Projeto

Sugestão de organização do código de forma para promover a modularidade e a manutenibilidade, ou seja, projeto funcional:

```
.
├── app/
│   ├── main.py               # Ponto de entrada da aplicação FastAPI e inicialização do DB
│   ├── routes/               # Módulo para os endpoints da API (FastAPI Routers)
│   │   ├── auth.py           # Rotas de auth (signup, login, me, forgot-password, reset-password, refresh)
│   │   ├── tasks.py          # Rotas para operações com tarefas (CRUD)
│   │   └── __init__.py       # Inicialização do pacote routes
│   ├── repositories/         # Módulo para interações com o banco de dados (camada de acesso a dados)
│   │   ├── user_repository.py # Funções CRUD para usuários
│   │   ├── task_repository.py # Funções CRUD para tarefas
│   │   └── __init__.py       # Inicialização do pacote repositories
│   ├── models/               # Módulo para os Pydantic Models (validação de entrada/saída) e modelos de DB
│   │   ├── user.py           # Models para User (e.g., UserCreate, UserResponse)
│   │   ├── task.py           # Models para Task (e.g., TaskCreate, TaskResponse)
│   │   ├── token.py          # Pydantic Models para tokens (e.g., Token, TokenData)
│   │   └── __init__.py       # Inicialização do pacote models
│   ├── utils/                # Módulo para funções utilitárias e helpers
│   │   ├── auth_utils.py     # Funções para hashing de senha, criação/validação de JWT
│   │   ├── db_utils.py       # Funções para inicialização do DB, obtenção de nome do banco, por exemplo.
│   │   └── __init__.py       # Inicialização do pacote utils
│   └── __init__.py           # Inicialização do pacote app
```

```
├── Dockerfile                # Arquivo para construir a imagem Docker da aplicação
├── requirements.txt          # Lista de dependências Python do projeto
├── README.md                 # Documentação do projeto, instruções para rodar localmente e deploy
```

6. Ferramentas e Tecnologias

- **Linguagem de Programação:** Python 3.9+
- **Framework Web:** FastAPI
- **Banco de Dados:** SQLite3
- **Autenticação:** PyJWT (para JSON Web Tokens), passlib (para hashing de senhas, especificamente bcrypt)
- **Validação de Dados:** Pydantic (já integrado ao FastAPI)
- **Testes de API:** Insomnia ou Postman (para enviar requisições e inspecionar respostas)
- **Containerização:** Docker
- **Plataforma de Deploy:** Render

7. Instruções Gerais - SUGESTÕES

1. **Configuração do Ambiente:**
    - Instalar Python (versão 3.9 ou superior) e o gerenciador de pacotes pip.
    - Criar um ambiente virtual para o projeto (python -m venv .venv).
    - Ativar o ambiente virtual (source .venv/bin/activate no Linux/macOS ou .\.venv\Scripts\activate no Windows).
2. **Criação do Projeto FastAPI:**
    - Iniciar o projeto seguindo a estrutura de pastas sugerida na seção 5.
    - Criar o arquivo main.py como ponto de entrada da aplicação.
3. **Implementação do Banco de Dados:**
    - Configurar a conexão com o SQLite no db_utils.py.
    - Definir os modelos de dados para User e Task em models/user.py e models/task.py utilizando SQLAlchemy (declarative base).
    - Implementar as funções CRUD (Create, Read, Update, Delete) nos arquivos user_repository.py e task_repository.py, interagindo com o SQLAlchemy.
    - Garantir que as tabelas sejam criadas automaticamente na inicialização da aplicação (e.g., em main.py ou db_utils.py).
4. **Desenvolvimento dos Endpoints:**
    - Implementar todos os endpoints listados na seção 4 dentro dos respectivos módulos em routes/.
    - Utilizar Path para parâmetros de rota e Query para parâmetros de query string.
    - Utilizar Request e Body (ou Pydantic Models) para processar os corpos das requisições POST/PUT.
    - Integrar as rotas com as funções dos repositórios para persistência de dados.

5. **Autenticação e Autorização:**
    - No auth_utils.py, implementar funções para:
        - Hashing e verificação de senhas (usando passlib.context.CryptContext).
        - Criação e validação de JWTs (access e refresh tokens).
    - Implementar os endpoints de autenticação (/auth/signup, /auth/login, /auth/me, /auth/forgot-password, /auth/reset-password, /auth/refresh) em routes/auth.py.
    - Proteger as rotas de tarefas (/tasks) utilizando dependências de segurança do FastAPI para JWT.
6. **Testes com Insomnia/Postman:**
    - Utilizar o Insomnia ou Postman para testar cada endpoint da API.
    - É esperado que você demonstre as requisições de:
        - Cadastro de usuário.
        - Login (e obtenção dos tokens).
        - Acesso à rota "Me" com o access_token.
        - Criação, leitura, atualização e exclusão de tarefas (com o access_token).
        - Fluxo de recuperação de senha.
        - Geração de novo access_token usando o refresh_token.
7. **Containerização com Docker:**
    - Criar um Dockerfile na raiz do projeto para empacotar a aplicação FastAPI.
    - O Dockerfile deve incluir as etapas para copiar o código, instalar dependências e rodar a aplicação.
    - Construir a imagem Docker localmente (docker build -t my-fastapi-app .).
    - Testar a aplicação localmente via Docker (docker run -p 8000:8000 my-fastapi-app).
8. **Deploy na Render:**
    - Criar uma conta gratuita na Render (se ainda não tiver).
    - Conectar o repositório Git do projeto (GitHub, GitLab, Bitbucket) à Render.
    - Configurar um novo "Web Service" na Render, apontando para o repositório e especificando o Dockerfile para o deploy.
    - Garantir que a aplicação esteja acessível publicamente após o deploy.

8. Critérios de Avaliação

- **Funcionalidade (40%):** Todos os endpoints implementados e funcionando corretamente conforme as especificações, incluindo tratamento de parâmetros, corpos de requisição e interações com o DB.
- **Organização do Código (20%):** Aderência à estrutura modular proposta (rotas, repositórios, models, utils), com código bem organizado e fácil de navegar.
- **Qualidade do Código (15%):** Código limpo, legível, com boa identação, comentários

relevantes e uso de boas práticas de programação Python e FastAPI.
- **Tratamento de Erros (10%):** Respostas de erro claras e informativas (e.g., códigos de status HTTP apropriados como 400 Bad Request, 401 Unauthorized, 404 Not Found, 403 Forbidden).
- **Segurança (10%):** Implementação correta e segura de hashing de senhas e JWT (geração, validação, refresh token).
- **Deploy e Docker (5%):** Aplicação containerizada corretamente com Docker e disponível e funcionando na plataforma Render.

9. Recursos e Dicas

- **Documentação Oficial do FastAPI:** Essencial para entender os conceitos e a sintaxe.
- **Tutoriais de JWT em Python:** Pesquisar por "Python JWT tutorial" ou "FastAPI JWT authentication" para exemplos práticos.
- **Exemplos de Dockerfile para FastAPI:** Muitos exemplos disponíveis online para referência.
- **Documentação da Render:** Para os passos específicos de deploy.
- **Insomnia/Postman:** Familiarize-se com a criação de requisições HTTP, cabeçalhos (especialmente Authorization), e corpos JSON.
- **Github da Disciplina**