Projeto SQL

Megadados

Introdução

Neste projeto vocês irão desenvolver um microserviço responsável por processar as solicitações do usuário, interagir com o banco de dados e fornecer os resultados apropriados de volta ao usuário.

O que precisa ser feito

Vamos construir uma API REST com ORM utilizando o framework FastAPI. O tema da API será livre.

Construa uma aplicação usando FastAPI que serve uma API REST para o projeto. Incorpore o uso do banco de dados MySQL no projeto usando o SQLModel.

Requisitos

Será necessário que sua API possibilite uma funcionalidade (ex: gestão de compromissos) sendo obrigatório ter no mínimo:

- Rota(s) de CRUD para duas entidades diferentes (você pode escolher quais). Ex: Despesa, Categorias, Usuário.
  - Cada entidade deve ter pelo menos três atributos.
  - Utilize atributos de **pelo menos** três tipos diferentes de dados (inteiro, data e hora, string, etc.). Você pode escolher quais!

Grupos

Projeto **individual**

Crie o repositório em https://classroom.github.com/a/vBZ5WQtN.

Entrega

- **Entrega de código até 14/10/2025 23:59**
- **Check de projeto até 22/10 (horários das aulas ou atendimentos)**

Entrega

- Datas para check de projeto:
  - Todos os atendimentos a partir de 24/09
    - Garantia mínima de 04 slots por atendimento

- Todas as aulas a partir de 24/09
  - Garantia mínima de quatro slots por aula

Rubrica Para a Entrega de Código

A partir do D, para avançar para um conceito maior, deve cumprir os anteriores.

Conceito	Pontos	Descrição
I	0	• Não atingiu D OU Não entregue no prazo
D	4	• A API consegue ser iniciada com sucesso **E** • Cobre requisitos do projeto de forma básica (ORM implementado, ainda que de forma parcial e requisitos cumpridos, ainda que de forma parcial)
C	6	• API cobre todos os requisitos **E** • ORM implementado sem falhas com SQLModel + MySQL
B	8	• Credenciais bem-gerenciadas (e.g. sem secrets committed no github) **E** • Usou tipos corretamente **E** • Tem um bom README **E** • Adicionou diagrama do modelo relacional no README **E** • Fez um vídeo (deixar link no README) descrevendo e demonstrando as funcionalidades da API
A	10	• API RESTful (Sem defeitos) **E** • Tem requirements.txt **E** • Tem .env.example **E** • Projeto bem organizado (boa organização em arquivos e pastas) **E** • Usou facilidades do FastAPI para que o site de documentação gerado automaticamente seja bem informativo: o Titulos das chamadas o Descrição dos argumentos o Exemplos de argumentos o entre outros

Check do projeto

Além da entrega do repositório, haverá uma etapa síncrona e presencial para check do projeto. Nas datas disponibilizadas pelos professores, havendo slot disponível, o aluno pode livremente escolher realizar sua checagem de projeto.

1. **Roteiro pré-checagem:** este roteiro deve ser realizado antes de adicionar seu nome na lista de checagem.

- Regular brilho do monitor para o nível adequado
- Abrir pasta do projeto no VSCode

- Manter aberto os principais arquivos do projeto
- Manter visível a estrutura de árvore de diretórios do projeto
- Manter um tamanho de fonte adequado para leitura pelo professor
- Desative todas ferramentas de auxílio desativadas: IA, autocomplete, autoformat. Caso não obedeça, o conceito a ser avaliado será considerado como não atingido.
- Manter a API em execução
- Manter o site da documentação de sua API aberto (rota /docs)

2. **Adicione nome na lista:** adicione seu nome na lista de checagem.
    - Faça isto apenas caso já tenha feito a entrega do projeto (commits)
    - Faça isto apenas caso ainda existam slots disponíveis no dia
    - Faça isto apenas caso já tenha feito o roteiro **pré-checagem**
3. **Checkagem:**
    - O professor irá chamar seu nome
    - O professor irá iniciar um cronômetro de 10 minutos (tempo máximo)
    - O professor irá até sua mesa e irá te entrevistar, coletando evidências de que os objetivos foram atingidos. Sua avaliação será conforme a rubrica para check do projeto.

Rubrica Para Check do projeto

A partir do D, para avançar para um conceito de nota maior, é necessário cumprir o anterior.

Conceito	Pontos	Descrição
I	0	Não atingiu D OU Não realizou check no prazo
D	4	O aluno entende o próprio código
C	6	O aluno tem domínio básico sobre os conceitos teóricos que sustentam o projeto (ex: REST, ORM)
B	8	Entende o suficiente para propor modificações e sabe como implementá-las E tem domínio avançado sobre os conceitos técnicos e teóricos que sustentam o projeto
A	10	Demonstra que realizou pesquisa autônoma, adquirindo conhecimento sobre assuntos correlatos (excede os materiais fornecidos)

Nota no projeto

Seja:

E: Nota de entrega de código

C: Nota de check de projeto

Para check realizado até 09/10 (somente para os primeiros 15% projetos da turma), a Nota Final no projeto 1 (NP1) será:

NP1 = MIN(10, (E + 1.25*C) / 2)

Para check realizado até 16/10:

NP1 = (E + C) / 2

Para check realizado até o prazo final 22/10:

NP1 = (E + 0.8*C) / 2

Para check não realizado:

NP1 = min(E, C)

Informaçoes adicionais úteis:

Tarefas Recomendadas Antes de Começar a Programar!

Antes de começar de fato a programar a entrega, recomendamos que:

- Estude o material sobre REST do site tutorial mencionado acima.
- Instale FastAPI (https://fastapi.tiangolo.com/learn/) e faça o tutorial.
    - Faça todas as etapas do tutorial básico até “Dependencies” incluindo a parte sobre ORM!
    - Vocês vão ter que relembrar também como funciona o protocolo HTTP: verbos, estrutura das mensagens, URI, etc.
    - A partir do tutorial você vai tomar contato com várias *features* de Python moderno: *type hints*, *context managers*, co-rotinas e Python assíncrono, especificação ASGI. Vai aprender também sobre *frameworks* interessantes para aplicações Web em Python: uvicorn (servidor ASGI), Starlette (framework Web), Pydantic (manipulação de dados e tipos em Python) entre outros.
        - Dica das trincheiras: **APROVEITE!** Vocês vão ver o quanto vocês vão querer ter projetos na sua vida profissional onde vocês poderão aprender novas tecnologias!

CRUD

A sigla *CRUD* vem do inglês *Create/Read/Update/Delete*, que indica as funções principais de um sistema de armazenamento de informações:

- *Create*: criar novos itens de dados;
- *Read*: consultar o sistema para resgatar itens de dados armazenados;
- *Update*: alterar itens de dados já existentes no sistema;
- *Delete*: remover itens de dados do sistema.

Esta sigla também é uma brincadeira em língua inglesa, já que a palavra *crud* significa sujeira – uma possível alusão ao estado deplorável de muitos sistemas de informação por aí afora!

Nosso microsserviço é, portanto, um CRUD (mas não deverá ser *crud*, ok?).

REST

A sigla *REST* significa *REpresentational State Transfer*. Trata-se de um padrão de projeto em arquitetura de sistemas no qual um serviço de informações tem as seguintes características:

- Arquitetura cliente-servidor: facilita a separação de responsabilidades entre a exibição da informação (responsabilidade do cliente) e o gerenciamento desta (no servidor)
- Ausência de estado (statelessness): toda a informação necessária para satisfazer uma requisição do cliente deve estar contida na própria requisição – em um serviço RESTful não existe a noção de *sessão*.
- *Cacheability*: Posto que os serviços que obedecem a estratégia REST não tem estado, podemos imaginar que duas consultas à mesma informação devem resultar na mesma resposta! (Diferente do caso em que haja estado armazenado no serviço, como um contador por exemplo). Claro que, entre duas consultas, a informação do banco de dados ao qual o serviço REST está conectado pode mudar. Um serviço REST deve, portanto, informar na sua resposta:
    - se a informação provida é *cacheable* ou não
    - se for *cacheable*, por quanto tempo
- Sistema em camadas: um sistema REST não deverá saber se está diretamente conectado ao sistema cliente ou não. Com isso, é possível inserir camadas de sistema entre o cliente e o serviço RESTful, tais como *caches*, *load balancers*, *proxies,* etc.
- Interface uniforme: Esta é uma das características principais de um sistema REST.
    - Baseado em recursos: O sistema está organizado em torno da idéia de recursos e suas representações. Quando o sistema cliente quer manipular algum recurso, tal recurso estará identificado diretamente na URI. Por exemplo, em um serviço web RESTful para um restaurante, vamos supor que o cliente deseja saber quais os pratos do menu. A URI não deve ser algo do tipo {dominio}/sistema?action=consulta&section=pratos", mas sim “{dominio}/pratos”
    - Mas então como especificar a “ação” a ser realizada? Usando os verbos do protocolo HTTP!
        - Create: POST
        - Read: GET
        - Update: PUT (update/replace) ou PATCH (update/modify)
        - Delete: DELETE
    - As respostas incluem toda a informação necessária para a manipulação do item de informação sendo enviado. Por exemplo: os metadados da resposta devem incluir informação sobre a *cacheability* do item, o tipo de dados (imagem, audio, texto, json,

xml), etc. O estado da resposta também é relevante: uma resposta bem sucedida deve retornar o código 200, já uma condição de erro deve ser indicada com o código HTTP adequado (404, 420, etc).
    - Hypermedia As The Engine Of Application State (HATEOAS): a informação de uma requisição web e de uma resposta incluem, além do texto principal, uma série de metadados (código de resposta, headers) que são chamados de hipermídia. Alem disso, quando necessário, a resposta de um sistema web RESTful pode incluir URIs para que o sistema cliente recupere outras partes relevantes da informação, se autorizado a fazê-lo.
- Code-on-demand: trata-se de uma característica opcional dos sistemas REST na qual o sistema pode retornar código executável (e.g. Javascript) ao cliente.

Para entender melhor como construir um sistema REST, leia o excelente material tutorial presente no site https://www.restapitutorial.com/ . Para ler em mais detalhes, o próprio site tem uma versão PDF do seu material, que é mais completa – veja a seção “Resources”

Object-Relational Mapping

Vocês devem ter observado que existem alguns paralelos entre um modelo relacional e uma arquitetura de classes composta de PODs (“Piece-Of-Data”) apenas:

- Uma tabela é similar a uma classe
- Uma coluna de tabela é similar a um campo de uma classe
- Um objeto é similar a uma linha de uma tabela
- Uma chave primária é similar a uma referência (e.g. um ponteiro) para um objeto
- Uma chave estrangeira é similar a um campo em um objeto que referencia outro objeto

Tendo essa observação em mente surgiram várias bibliotecas que permitem criar objetos em nossas aplicações, e que são diretamente mapeados para linhas em tabelas de um banco de dados relacional. Essas bibliotecas são chamadas de bibliotecas *Object-Relational Mapping (ORM)*.

Em Python, uma das bibliotecas ORM mais conhecidas é o SQLAlchemy (https://www.sqlalchemy.org/).

Os ORMs se prestam a uma outra função: abstrair o detalhe de qual banco de dados está sendo usado na aplicação. Vocês devem ter observado também que existem vários SGBDs relacionais e que estes são apenas parcialmente compatíveis entre si, em termos de características e de especificação da sua variante de SQL. Porém, apesar das incompatibilidades em termos de detalhes, quase todos fazem a mesma coisa – são compatíveis em conceito. Uma biblioteca ORM também serve para abstrair esses detalhes.

Como essa abstração de detalhes é construída? Uma biblioteca ORM define uma maneira de descrever os dados que é independente de SGBD, é feita em nível de aplicação. Internamente, a biblioteca traduz as construções feitas pelo usuário da biblioteca em comandos específicos para o SGBD escolhido pelo usuário. A

Figura 1 mostra a arquitetura do SQLAlchemy. Podemos ver que a camada ORM não se comunica diretamente com o SGBD, mas sim com uma abstração do SGBD que é formada pelos componentes “Schema/Types”, “SQL Expression Language” e “Engine”. Esta última componente representa diretamente o SGBD, e é responsável pela comunicação com o SGBD usando bibliotecas específicas para cada um destes (componente “DBAPI”).

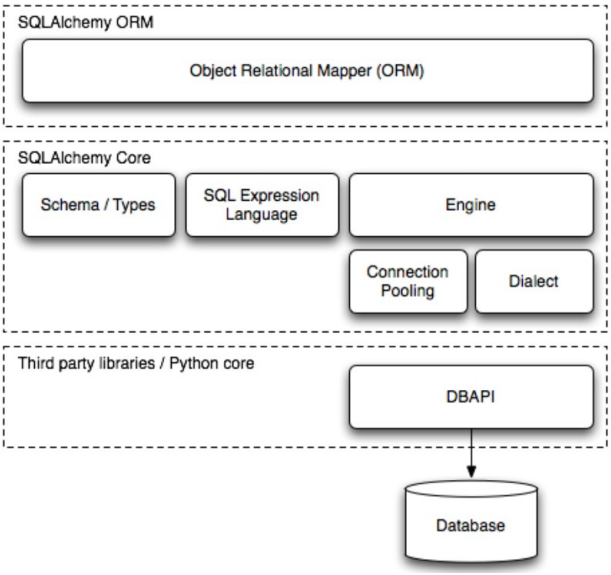

Figura 1: Arquitetura do SQLAlchemy. Fonte: Michael Bayer. SQLAlchemy. In Amy Brown and Greg Wilson, editors, The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks 2012 http://aosabook.org