APS 1: Criando uma interface REST

Megadados – Projeto 1

Refatorando o projeto

Agora que a fase 1 do projeto está completa, verifique o repositório https://github.com/Insper/Megadados2020-2-Projeto1-alunos que o professor desenvolveu para servir de referência para vocês. A tag 1.0 representa aquilo que era esperado do projeto 1. Porém este projeto está bastante incompleto: vamos refatorar esse código para facilitar os desenvolvimentos futuros. Lembrando que ainda teremos que desenvolver as *features* de usuário e de listas (por enquanto temos apenas uma lista global).

Criando testes unitários

Antes de qualquer refatoração, temos que criar testes unitários. Assim saberemos que nossos movimentos de refatoração não quebraram o código.

Como pensar em testes unitários? Mais importante do que *cobertura de código* (ou seja, quantas linhas de código são "tocadas" pelos nossos testes), o importante é *cobertura de cenários*, temos que garantir (dentro do razoável) que todos os usos legítimos e ilegítimos estão explorados. Portanto, a referência para a criação de testes é realmente a lista de casos de uso do sistema. Na hora de construir os testes, pense assim: o que você faria para testar o sistema "na mão"? Automatize isso!

Um teste geralmente está organizado em 3 partes:

- *Set-up*: Construir o ambiente no qual o teste vai rodar
    - Carregar dados de teste
    - Instanciar os objetos necessários para rodar os testes, incluindo os *mock objects* – versões *fake* de objetos do sistema para isolar o componente de teste
- Execução do teste em si
- Verificação de resultados

Vamos construir o teste mais simples possivel do nosso sistema: vamos testar que ao acessar a API sem especificar rota nenhuma ('/'), vai dar ruim – para garantir que ninguem vai pendurar algo na raiz no futuro.

Se acessarmos diretamente a raiz do serviço, temos como resposta um código 404 – not found – e um corpo da resposta que é um JSON {"detail":"Not Found"}. Então é só automatizar isso!

O código deste teste é basicamente o código-exemplo de teste do tutorial do FastAPI:

```
def test_read_main_returns_not_found():
    response = client.get('/')
    assert response.status_code == 404
    assert response.json() == {'detail': 'Not Found'}
```

Toda e qualquer mudança no código requer um novo branch de trabalho. Crie um branch 'tests' no git. Neste branch crie um arquivo 'test_main.py' que vai conter os testes unitários.

Instale “pytest”, e no arquivo test_main.py insira o seguinte código:

```
from fastapi.testclient import TestClient

from main import app

client = TestClient(app)


def test_read_main_returns_not_found():
    response = client.get('/')
    assert response.status_code == 404
    assert response.json() == {'detail': 'Not Found'}
```

Rode o teste simplesmente digitando “pytest”, deverá funcionar.

Entregável

Escreva os demais testes. Coloque a tag “1.1”.

Git tem dois tipos de tag: *annotated* e *lightweight*. Verifique a documentação do Git para conhecer a diferença entre elas. Crie a tag como *annotated*, e faça o push para o *remote* com a *flag* `--tags` para indicar ao Git que você quer que a informação de tags também seja transferida ao servidor remoto. Verifique agora o github: você verá que a aba “tags” agora contém seu tagging.

Dependency Injection – preparando o terreno

Ao construir os testes uma coisa ficou clara rapidamente: cada teste que roda modifica nossa base de dados, e se não executamos as operações de limpeza da base, teremos problemas. Isso acontece porque a base de dados é parte integral do nosso “main.py” – é um dicionário global!

O correto é usar a estratégia de injeção de dependência (*dependency injection*) onde, ao invés de “cimentar” a base de dados dentro do microsserviço, devemos passar um objeto responsável pelo manuseio de dados. Assim, quando vamos executar os testes, podemos criar um objeto *mock*, e não estaremos alterando a base de dados original!

Vamos primeiramente criar um branch “dbsession” para construir o nosso objeto de gerenciamento de dados (sem realmente usar um banco de dados ainda).

No lugar da linha:

```
tasks = {}
```

vamos colocar o código abaixo:

```
class DBSession:
    tasks = {}

    def __init__(self):
        self.tasks = DBSession.tasks
```

```
def get_db():
    return DBSession()
```

Insira a dependencia usando o padrão do FastAPI: crie um argumento de função para passar o "get_db" para a função, e lá dentro troque os usos de "tasks" por "db.tasks" – é só a etapa inicial da refatoração, não o estágio final ainda, ok?

Por exemplo:

```
async def read_tasks(completed: bool = None, db: DBSession = Depends(get_db)):
    if completed is None:
        return db.tasks
    return {
        uuid_: item
        for uuid_, item in db.tasks.items() if item.completed == completed
    }
```

Faça o mesmo para todos os usos de "tasks" e verifique que nenhum teste quebrou.

Agora você pode trabalhar na remoção de todos os "db_tasks", substituindo estes por métodos na classe DBSession, sempre checando se os testes não quebram.

Ainda não fizemos uso do *dependency injection*, tenha paciencia. No próximo handout vamos começar a usar bancos de dados, e vai ficar claro porque desejamos deixar tudo pronto para o *dependency injection.*

Quando tudo estiver terminado e *committed*, faça o merge com o *branch* principal e delete o branch "dbsession". Excelente, vai tomar um café!

Entregável

Termine a implementação do DBSession. Coloque a tag "1.2"

Modularização

Por enquanto o código está todo em dois arquivos: um com o sistema todo, outro com os testes. Vamos modularizar melhor esse sistema, seguindo as instruções de https://fastapi.tiangolo.com/tutorial/bigger-applications/. Crie um branch "refactor" para esta tarefa.

Crie um diretório "api", vamos mover o código para lá (use "git mv"). Crie o arquivo vazio "__init__.py" para indicar que "api" é um pacote. Com isso os testes unitários devem ter quebrado, pois agora temos que indicar que estamos importando "app" de um pacote de módulos, e não diretamente de um módulo. Conserte esse erro colocando um '.' antes de "main" no import:

```
from .main import app
```

Pode rodar o pytest da raiz do projeto mesmo, ele é espertinho e localiza todos os arquivos test_*.py.

Separe o código de "main.py" em mais dois arquivos:

- database.py: inclui DBSession e get_db
- models.py: inclui o modelo Pydantic

Dica: a cada pequeno movimento de código, rode os testes unitários.

Agora crie um diretório “routers”, um arquivo vazio “__init__.py” dentro dele, e o arquivo “task.py” lá dentro. Mova todo o código de roteamento para lá. Conserte todos os problemas resultantes de mover código.

Por fim, remova todos os “/task” de todas as rotas, e siga as instruções em https://fastapi.tiangolo.com/tutorial/bigger-applications/#include-an-apirouter-with-a-prefix-tags-responses-and-dependencies para colocar o prefixo “/task” de modo mais esperto! Remova tambem o argumento “tags=[‘task’]” de todas as rotas, passe para o main.py no roteador.

Commit, *merge*, e remova o *branch*

Entregável

Complete a modularização. Marque como tag “1.3”

Rubrica

Conceito	Descrição
I	• Não fez, ou entregou groselha
D	• Fez pelo menos os testes unitários, faltam algumas chamadas
C	• Atingiu conceito D • Fez os tres itens, mas os testes unitários não estão completos
B	• Atingiu conceito C • Código apresenta boa qualidade e estética (sim, é importante!) • Testes quase completos
A	• Atingiu conceito B • Testes completos