Configurando o Ambiente de Desenvolvimento

> https://fastapidozero.dunossauro.com/01/

Objetivos dessa aula:

- Introdução ao ambiente de desenvolvimento
  - ferramentas, testes, configuração, etc
- Instalação do FastAPI e suas dependências
- Configuração das ferramentas de desenvolvimento
- Execução do primeiro "Hello, World!" com FastAPI com testes!

O ambiente de desenvolvimento

1. Um editor de texto a sua escolha (Eu vou usar o GNU/Emacs)
2. Um terminal a sua escolha (Usarei o Terminator)
3. A versão 3.11 do Python instalada.
    - Caso não tenha essa versão você pode baixar do site oficial
    - Ou instalar via pyenv
4. O Poetry para gerenciar os pacotes e seu ambiente virtual (caso não conheça o poetry temos uma live de python sobre ele)
5. Git: Para gerenciar versões
6. Docker: Para criar um container da nossa aplicação

Coisas opcionais que podem ajudar

7. O pipx pode te ajudar bastante nesses momentos de instalações globais
8. O ignr para criar nosso gitignore
9. O gh para criar o repositório e fazer alterações sem precisar acessar a página do github

Python 3.11

```
pyenv update
pyenv install 3.11:latest
```

Poetry

Para instalar o poetry você pode fazer a instalação recomendada pelo site ou de forma mais simplificada via pipx

```
pip install pipx
pipx install poetry
```

Estrutura base do projeto

Vamos criar nossa estrutura com base na estrutura simples que o Poetry cria para nós

```
poetry new fast_zero
cd fast_zero
```

isso vai nos gerar essa estrutura:

```
.
├── fast_zero
│   └── __init__.py
├── poetry.lock
├── README.md
└── tests
    └── __init__.py
```

Contornando possíveis erros

Para que a versão que instalamos com pyenv seja usada em nosso projeto criado com poetry, devemos dizer ao pyenv qual versão do python será usada nesse diretório:

```
pyenv local 3.11.4  # Essa era a maior versão do 3.11 quando escrevi
```

Em conjunto com essa instrução, devemos dizer ao poetry que usaremos essa versão em nosso projeto. Para isso vamos alterar o arquivo de configuração do projeto o `pyproject.toml` na raiz do projeto:

```
[tool.poetry.dependencies]
python = "3.11.*"  # .* quer dizer qualquer versão da 3.11
```

Criando o ambiente virtual

```
poetry install
```

Eu sei, você quer FastAPI, veio por isso

Para instalar o fastapi

```
poetry add fastapi
```

Nosso olá mundo

```python
from fastapi import FastAPI

app = FastAPI()

@app.get('/')
def read_root():
    return {'message': 'Olá Mundo!'}
```

Executando esse código

Para que a execução ocorra, precisamos de um servidor

A recomendação do fastapi é pelo Uvicorn

```
poetry add uvicorn

# Para executar nossa estrutura

uvicorn fast_zero.app:app --reload
```

O "teste"

Se acessarmos http://localhost:8000 podemos ver nossa aplicação

O swagger

Se acessarmos http://localhost:8000/docs podemos ver os endpoinds da nossa aplicação e testar os requests

O ambiente de desenvolvimento

Para nosso ambiente vamos usar algumas ferramentas diferentes

- Taskipy: Para não termos que lembrar todos os comandos da aplicação
- Pytest: Para escrevermos os testes
- Blue: Para formatar e checar se estamos seguindo a PEP-8
- Isort: Para ordenar os imports
- Ruff: Um linter bem poderoso e rápido

Pytest

Se quisermos testar nossa aplicação, precisamos de um bom framework de testes e de algo que nos mostre a cobertura do código

```
poetry add --group dev pytest pytest-cov
```

A única configuração que precisamos fazer é dizer que para encontrar os testes e nossa aplicação é na raiz do repositório

```
[tool.pytest.ini_options]
pythonpath = "."
```

Com isso podemos ver o que está ou não testado

```
pytest --cov=fast_zero -vv
coverage html
```

Queremos ver a cobertura do código e os erros de forma verbosa

Taskipy

Bom, esses comandos são bem difíceis de lembrar:

```
uvicorn fast_zero.app:app --reload  # para rodar a aplicação
pytest --cov=fast_zero -vv          # teste
coverage html                       # cobertura
```

Com taskipy podemos fazer esses comando serem uma única palavra

```
task run   # para rodar o servidor
task test  # para executar os testes
```

Instalação e Configuração do taskpy

```
poetry add --group dev taskipy
```

```
[tool.taskipy.tasks]
run = 'uvicorn fast_zero.app:app --reload'
test = 'pytest -s -x --cov=fast_zero -vv'
post_test = 'coverage html'
```

Blue e Isort

Blue e Isort são formatadores de código. Que fazem com que todos sigam as mesmas regras na "diagramação" do código

```
poetry add --group dev blue isort
```

Se executarmos o blue

```
blue . --check --diff
```

Podemos ver que algumas coisas não estão "legais"

Configuração do blue e do isort

```toml
[tool.blue]
extend-exclude = '(migrations/)'

[tool.isort]
profile = "black"
line_length = 79  # Modelo da PEP-8
extend_skip = ['migrations']

[tool.taskipy.tasks]
format = 'blue .  && isort .'
```

Ruff

O ruff vai procurar por definições com nomes estranhos, imports esquecidos, variáveis não usadas e etc...

```
poetry add --group dev ruff
```

A configuração

```
[tool.ruff]
line-length = 79
exclude = ['.venv', 'migrations']
```

Fazendo uma cadeia de comandos com taskipy

```
[tool.taskipy.tasks]
lint = 'ruff . && blue --check . --diff'
format = 'blue .  && isort .'
run = 'uvicorn fast_zero.app:app --reload'
pre_test = 'task lint'
test = 'pytest -s -x --cov=fast_zero -vv'
post_test = 'coverage html'
```

Testando o nosso hello world

Dentro da pasta `test` vamos criar um arquivo chamado `test_app.py`

```
from fastapi.testclient import TestClient
from fast_zero.app import app

client = TestClient(app)
```

Testando de fato

```python
def test_root_deve_retornar_200_e_ola_mundo():
    response = client.get('/')
    assert response.status_code == 200
    assert response.json() == {'message': 'Olá Mundo!'}
```

Commit

```
ignr -p python > .gitignore
git init .
gh repo create
```