

# 01 - Configurando o Ambiente de Desenvolvimento

| <https://fastapidozero.dunossauro.com/4.0/01/>

# 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)
  - Kate, vim, emacs, vsodium, ...
2. Um terminal a sua escolha (Usarei o Terminator)
  - Ghostty, tilix, iTerm2, windows terminal (WSL/Powershell)
3. Uma versão suportada do python
  - 3.9+
4. O pipx pode te ajudar bastante nesses momentos de instalações globais
  - Que vamos instalar juntos

# O ambiente de desenvolvimento

5. O Poetry (2+) para:

- Gerenciar os pacotes
- Instalações de versões do python; e
- Ambiente virtual

6. Git: Para gerenciar versões

7. Docker: Para criar um container da nossa aplicação

- Aula 11 pra frente

| Presentes no **apêndice A** também :)

# Caso seja preciso

Materiais de qualidades e de pessoas incrível que fazem materiais abertos como eu:

1. Curso de git do teomewhy
2. Curso de Docker da LinuxTips
3. Ajuda para configurar o ambiente - Apêndice A

# Configuração do ambiente (pipx)

Como nem todo mundo conhece as ferramentas, vamos iniciar com a instalação do pipx. `pipx` é um instalador de pacotes globais do python, geralmente usado para bibliotecas de terminal, que são executáveis.

```
pip install --user pipx      # windows
python -m pipx ensurepath   # windows

sudo apt/dnf install pipx   # linux
pipx ensurepath              # linux
```

# Configuração do ambiente (poetry)

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

```
pipx install poetry  
pipx inject poetry poetry-plugin-shell
```

# Instalação das ferramentas externas

Isso pode te ajudar a ter menos dificuldade, caso trave em algum lugar

| [https://fastapidozero.dunossauro.com/4.0/apendices/a\\_instalacoes/](https://fastapidozero.dunossauro.com/4.0/apendices/a_instalacoes/)



# Estrutura base do projeto

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

```
poetry new --flat fast_zero  
cd fast_zero
```

isso vai nos gerar essa estrutura:

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

## A versão do python

Uma boa prática é sempre usar a maior versão **possível** do python no projeto. Nem sempre todas as bibliotecas suportam a versão mais recente, mas no nosso caso, tudo funciona na `3.13`:

```
poetry python install 3.13 # Mais atual hoje!  
poetry env use 3.13        # Pra usar essa versão no projeto
```

## A versão do python

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:

```
[project]
# ...
requires-python = ">=3.13,<4.0"
```

## Criando o ambiente virtual

```
poetry install
```

# Eu sei, você quer FastAPI, veio por isso

Para instalar o fastapi

```
poetry add fastapi[standard]
```

# Nosso olá mundo [0]

Um código python simples!

```
# fastzero/app.py
def read_root():
    return {'message': 'Olá Mundo!'}
```

No terminal:

```
python -i fastzero/app.py
```

# Nosso olá mundo [1]

```
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

Isso inicia o servidor de desenvolvimento do FastAPI:

```
fastapi dev fast_zero/app.py
```

## O "teste"

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

## O swagger

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

## O redoc

Se acessarmos <http://localhost:8000/redoc> podemos ver os endpoints e suas respostas de forma mais detalhada.

**O ambiente de desenvolvimento**

## Para nosso ambiente vamos usar algumas ferramentas diferentes

Ferramentas de desenvolvimento são bastante pessoais. Selecionei 3 que representam bem o que esperamos de um ambiente de desenvolvimento:

- `Ruff`: Um linter e formatador bem poderoso e rápido
- `Pytest`: Para escrevermos os testes
- `Taskipy`: Para não termos que lembrar todos os comandos da aplicação

# Ruff

O Ruff é uma ferramenta moderna em python, compatível com os projetos de análise estática escritos e mantidos originalmente pela comunidade no projeto PYCQA e tem duas funções principais:

1. Analisar o código de forma estática (Linter): Efetuar a verificação se estamos programando de acordo com boas práticas do python.
2. Formatar o código (Formatter): Efetuar a verificação do código para padronizar um estilo de código pré-definido.

Para instalar:

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

# Configurando o ruff

Para configurar o ruff montamos a configuração em 3 tabelas distintas no arquivo `pyproject.toml`. Uma para as configurações globais, uma para o linter e uma para o formatador.

A global:

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

# O linter do ruff

Durante a análise estática do código, queremos buscar por coisas específicas. No Ruff, precisamos dizer exatamente o que ele deve analisar.

- **I** (Isort): ordenação de imports em ordem alfabética
- **F** (Pyflakes): procura por alguns erros em relação a boas práticas de código
- **E** (pycodestyle): erros de estilo de código
- **W** (pycodestyle): avisos sobre estilo de código
- **PL** (Pylint): "erros" em relação a boas práticas de código
- **PT** (flake8-pytest): boas práticas do Pytest



# Configuração no pyproject.toml

```
[tool.ruff.lint]
preview = true
select = ['I', 'F', 'E', 'W', 'PL', 'PT']
```

Para mais informações sobre a configuração e sobre os códigos do ruff e dos projetos do PyCQA, você pode checar a [documentação do ruff](#) ou as documentações originais dos projetos [PyQCA](#).

# Formatador do ruff

A formatação do Ruff praticamente não precisa ser alterada. Pois ele vai seguir as boas práticas e usar a configuração global de 79 caracteres por linha. A única alteração que farei é o uso de aspas simples `'` no lugar de aspas duplas `"`:

```
[tool.ruff.format]
preview = true
quote-style = 'single'
```

| Novamente uma escolha bastante opionada :)

# Usando o ruff

O ruff é feito para ser usado no terminal, alguns comandos são bem interessantes. Como:

- `ruff check .`: Faz a checagem dos termos que definimos antes
- `ruff format .`: Faz a formatação do nosso código sendo as boas práticas

# Pytest

O `Pytest` é uma framework de testes, que usaremos para escrever e executar nossos testes. O configuraremos para reconhecer o caminho base para execução dos testes na raiz do projeto `.`:

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

Também vamos usar o `pytest-cov` para ver o que está ou não coberto pelos testes.

# Configuração do pytest

O configuraremos para reconhecer o caminho base para execução dos testes na raiz do projeto `.`:

```
[tool.pytest.ini_options]
pythonpath = "."
addopts = '-p no:warnings'
```

Na segunda linha dizemos para o pytest adicionar a opção `no:warnings`. Para ter uma visualização mais limpa dos testes, caso alguma biblioteca exiba uma mensagem de warning, isso será suprimido pelo pytest.

## 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 e mais chatos ainda de digitar.

```
ruff check .           # para checar
ruff format .          # para formatar
fastapi dev fast_zero/app.py  # para rodar a aplicação
pytest --cov=fast_zero -vv    # teste
coverage html           # cobertura
```

| Por esse motivo você não gosta de usar o shell, eu sei...

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 taskipy

## Instalação:

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

## Configuração:

```
[tool.taskipy.tasks]  
lint = 'ruff check'  
format = 'ruff format'  
run = 'fastapi dev fast_zero/app.py'  
test = 'pytest -s -x --cov=fast_zero -vv'
```



## Juntando comandos com taskipy

Alguns comandos fazem mais sentido quando compostos. Queremos fazer mais, com menos comandos:

```
[tool.taskipy.tasks]  
pre_format = 'ruff check --fix'  
pre_test = 'task lint'  
post_test = 'coverage html'
```

## Cadeia de comandos com taskipy

Em outros momentos, queremos fazer uma coisa, só se a primeira der certo, para isso podemos fazer:

```
pre_test = 'task lint'  
test = 'pytest -s -x --cov=fast_zero -vv'  
post_test = 'coverage html'
```



primeiro a task de lint, se der certo, test, se der certo, coverage :)

# Voltando ao código

| Aos testes, na verdade!

# 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

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

def test_root_deve_retornar_ok_e_ola_mundo():
    client = TestClient(app) # Arrange

    response = client.get('/') # Act

    assert response.status_code == HTTPStatus.OK # Assert
    assert response.json() == {'message': 'Olá Mundo!'} # Asset
```

# A estrutura de um teste

A estrutura de um teste, costuma contar com 3 ou 4 fases importantes.

- Organizar (Arrange)
- Agir (Act)
- *Afirmar* (Assert)
- *teardown*

# Commit

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

# Exercício

Crie um repositório para acompanhar o curso e suba em alguma plataforma, como [Github](#), [gitlab](#), [codeberg](#), etc. E compartilhe o link no [repositório do curso](#) para podermos aprender juntos.

! Não se esqueça de responder o [quiz](#) dessa aula