Tornando o projeto assíncrono

> https://fastapidozero.dunossauro.com/4.0/08/

Objetivos da Aula

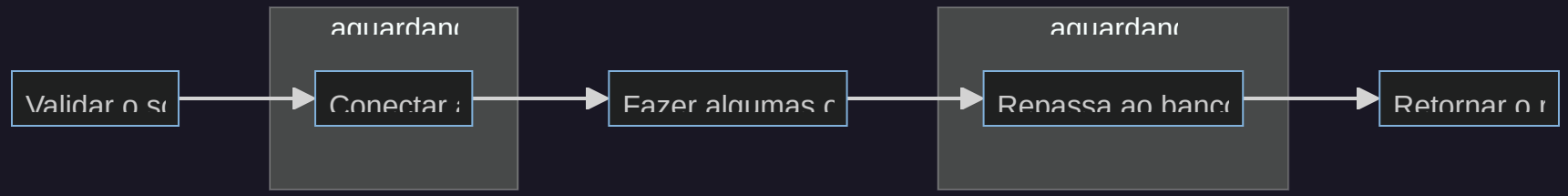
- Introduzir os conceitos de programação assíncrona
- Refatorar nossa aplicação para suportar AsyncIO
  - Tanto a aplicação (SQLAlchemy, FastAPI)
  - Quanto os testes (Pytests, Fixtures)

O bloqueio da aplicação

> Parte 1

Bloqueio de I/O

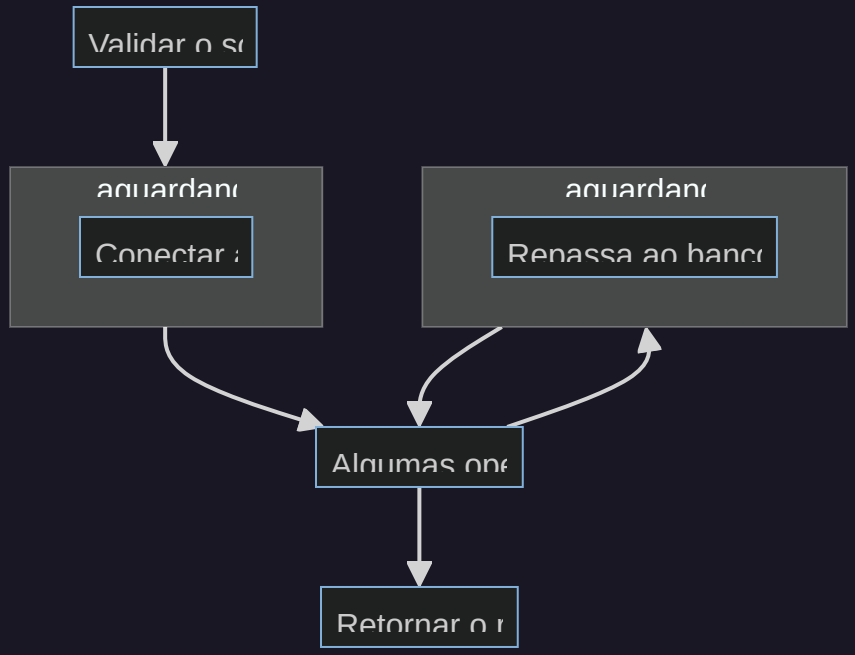
Um dos problemas mais comuns ao lidarmos com uma aplicação web é a necessidade de estarmos sempre disponíveis para responder a requisições enviadas pelos clientes. Quando recebemos uma requisição, normalmente precisamos processá-la:



Esse fluxo de execução é chamado de **bloqueante**, pois a aplicação fica "parada", aguardando a resposta de sistemas externos, como o banco de dados.

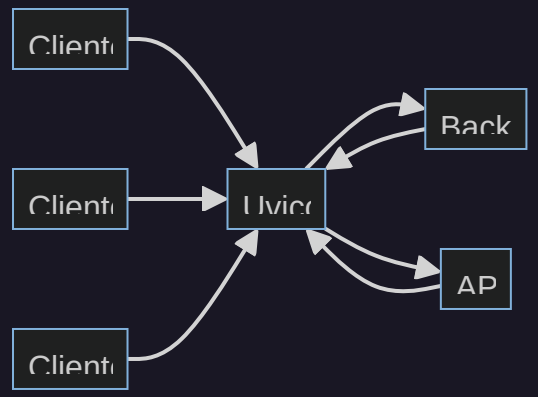
Bloqueio de I/O

```python
@router.put('/{user_id}', response_model=UserPublic)
def update_user(
    user_id: int,
    user: UserSchema,
    session: Session,
    current_user: CurrentUser,
):
    if current_user.id != user_id:
        raise HTTPException(
            status_code=HTTPStatus.FORBIDDEN,
                        detail='Not enough permissions'
        )
    try:
        current_user.username = user.username
        current_user.password = get_password_hash(user.password)
        current_user.email = user.email
        session.commit()
        session.refresh(current_user)

        return current_user

    except IntegrityError:
        raise HTTPException(
            status_code=HTTPStatus.CONFLICT,
            detail='Username or Email already exists',
        )
```



O bloqueio é da aplicação

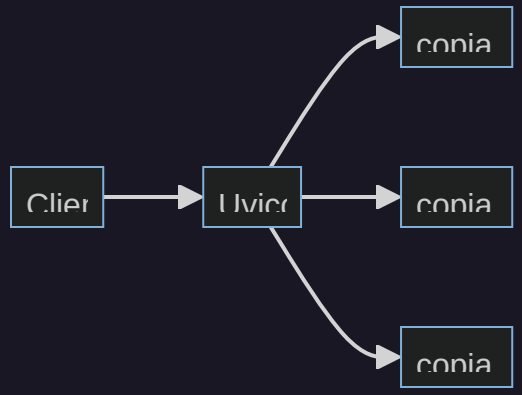
Uma coisa que deve ficar **extremamente clara** é que embora tenhamos um bloqueio na aplicação, **não** temos um bloqueio no **servidor** (uvicorn).



Por padrão, são `2048` requisições que podem aguardar no backlog

O servidor pode "copiar" a aplicação

Com isso distribuir as requisições de forma paralela




```
uvicorn fast_zero.app:app --workers 3
```

A aplicação ainda bloqueia, mas como tem mais, faz coisas ao mesmo tempo.

Aplicação não bloqueante

> Parte 2

Corrotinas

Embora esse assunto possa se estender de forma sem controle, uma corrotina *assíncrona* basicamente é uma função em python que pode ser **escalonada** durante o bloqueio de I/O.

São criadas pela palavra reservada `async` e o escalonamento é feito pela palavra `await`.

```
# código ilustrativo ao nosso contexto
async def get_users() -> list[User]:
    result = await session.scalars(select(User))
    return result
```

Corrotinas

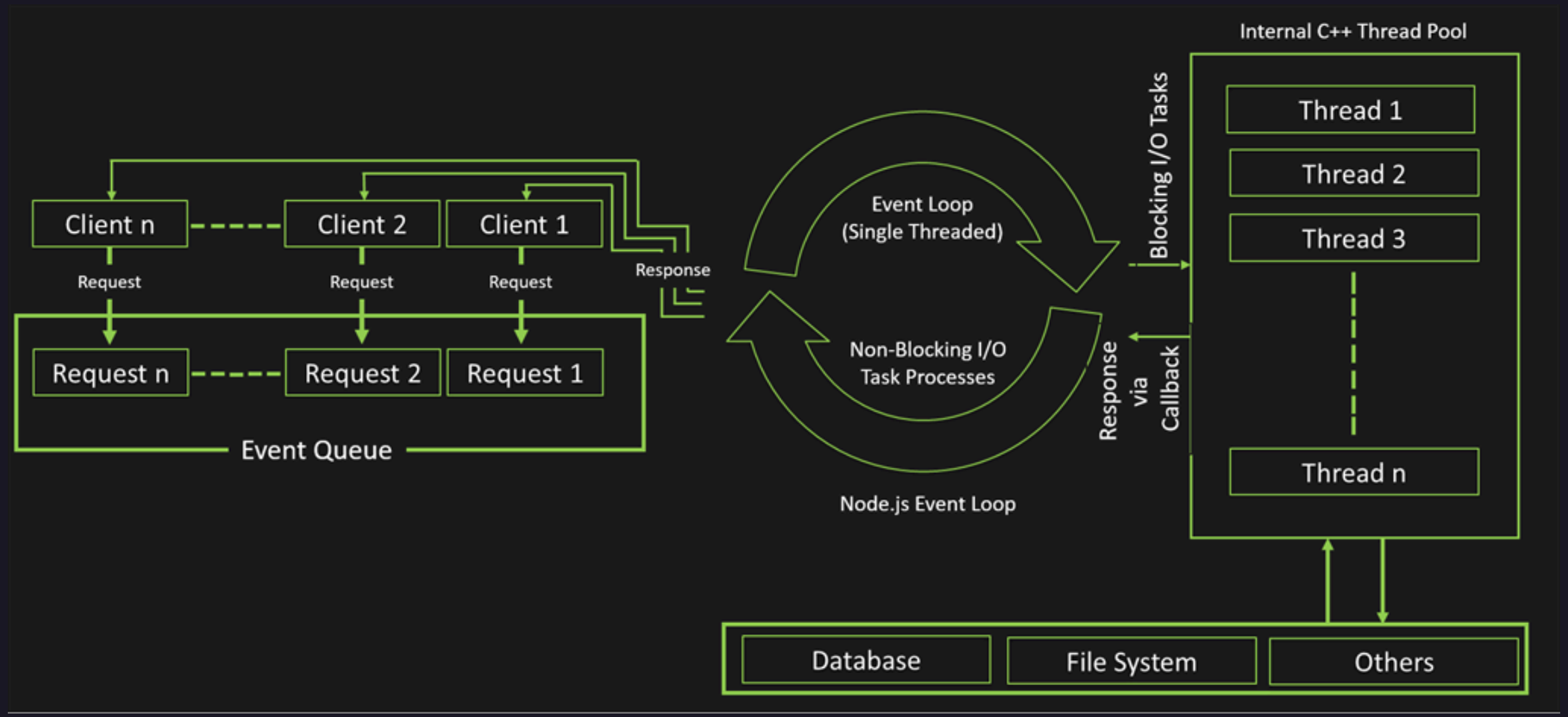
Uma das características de uma corrotina, é o fato dela não ser executada quando chamada diretamente:

```
>>> async def foo():
...     ...
...
>>> foo()
<coroutine object foo at 0x7f2e0ec79850>
```

Ela precisa ser executada por um agente externo. Um **loop de eventos**.

Loop de eventos

É responsável por executar e coordenar todas as corrotinas



https://viblo.asia/p/ben-trong-nodejs-2-thu-vien-libuv-RQqKLRMOl7z#_libuv-5

Cooperatividade e Escalonamento

- Cooperatividade é a habilidade de "passar a vez" para que outra corrotina seja executada.
- Escalonamento é o que o loop faz ao trocar entre corrotinas durante a cooperação.

Bando de dados e bloqueios

> Parte 3

Instalando o suporte a asyncio no sqla

Embora o suporte a `asyncio` seja nativo no `sqlalchemy` 2.0. Algumas plataformas (como Silicon) não tem os pacotes pré-compilados (wheels) do `greenlet`. Para isso, vamos fazer uma instalação explícita:

```
poetry add "sqlalchemy[asyncio]"
```

> obs: A partir da versão 2.1, o suporte a asyncio deverá **sempre** ser instalado manualmente!

SQLite + AsyncIO

Como estamos utilizando o banco de dados SQLite, que não possui suporte nativo a asyncio no Python, precisamos instalar uma extensão chamada aiosqlite. Ela permite a execução assíncrona com bancos SQLite:

```
poetry add aiosqlite
```

Alterando o `.env`

Para que nossa conexão esteja ciente que o aiosqlite está sendo usado, devemos alterar a variável de ambiente para contemplar essa alteração:

```
DATABASE_URL="sqlite+aiosqlite:///database.db"
```

Sessão com suporte a AsyncIO

```python
#fast_zero/database.py
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine

from fast_zero.settings import Settings

engine = create_async_engine(Settings().DATABASE_URL)


async def get_session():
    async with AsyncSession(engine, expire_on_commit=False) as session:
        yield session
```

`expire_on_commit` deve ser `False` pois não sabemos se todas as corrotinas "liberaram" a sessão. Para prevenir erros, essa opção de ser ativada.

Pytest + AsyncIO

Embora o SQLAlchemy e o FastAPI lidem de forma nativa com programação assíncrona, o pytest ainda não. Para isso, precisamos instalar uma extensão que adicione esse suporte. A pytest-asyncio fornece um mecanismo de marcação para testes e também um para criação de fixtures assíncronas:

```
poetry add --group dev pytest-asyncio
```

Pytest + AsyncIO

Uma exigência formal do pytest-asyncio é que seja configurado o escopo padrão das fixtures:

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

> Para evitar cair em mais um assunto, o tópico de escopos de fixtures serão abordados na aula 11

Fixture async para `session`

```python
import pytest_asyncio
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine

@pytest_asyncio.fixture
async def session():
    engine = create_async_engine(
        'sqlite+aiosqlite:///:memory:',
        connect_args={'check_same_thread': False},
        poolclass=StaticPool,
    )
    # ...
```

Fixture async para `session`

```python
@pytest_asyncio.fixture
async def session():
    # ...
    async with engine.begin() as conn:
        await conn.run_sync(table_registry.metadata.create_all)

    async with AsyncSession(engine, expire_on_commit=False) as session:
        yield session

    async with engine.begin() as conn:
        await conn.run_sync(table_registry.metadata.drop_all)
```

Testando o banco de dados

```python
#tests/test_db.py
import pytest

@pytest.mark.asyncio
async def test_create_user(session, mock_db_time):
    with mock_db_time(model=User) as time:
        new_user = User(
            username='alice', password='secret', email='teste@test'
        )
        session.add(new_user)
        await session.commit()

    user = await session.scalar(select(User).where(User.username == 'alice'))
    # ...
```

Rodando um único arquivo

```
task test tests/test_db.py
```

Refatorando com teste!

> Parte 4

Técnica de refatoração com testes

Uma das grandes vantagens de termos uma boa cobertura de testes é que podemos fazer mudanças estruturais no projeto e garantir que tudo funcione da forma como já estava antes.

Os testes nos trazem uma **segurança** para que tudo possa mudar internamente sem alterar os resultados da API. Para isso, a estratégia que vamos usar aqui é a de caminhar executando um teste por vez.

Uma das funcionalidades legais do pytest é poder executar somente um único teste, ou um grupo deles, usando o nome do teste como base. Para isso, podemos chamar task test passando a flag `-k` seguida do nome do teste. Algo como:

```
task test -k test_create_user

# ...
tests/test_users.py::test_create_user FAILED
```

Para cada teste que falhar, vamos nos organizando para fazer a conversão do código para assíncrono.

Para listar todos os testes presentes no nosso projeto, podemos usar a flag `--collect-only` do pytest:

```
task test --collect-only
<Dir fast_zero>
  <Package tests>
    <Module test_app.py>
      <Function test_root_deve_retornar_ok_e_ola_mundo>
    <Module test_auth.py>
      <Function test_get_token>
    <Module test_db.py>
      <Coroutine test_create_user>
...
```

Router `auth`

Acredito que começar pelo router `auth` pode ser menos assustador, já que até o momento ele tem somente um endpoint (`login_for_access_token`) e um teste (`test_get_token`).

```
task test -k test_get_token
# ...
FAILED tests/test_auth.py::test_get_token - AttributeError: 'coroutine' object has no attribute 'password'
```

Esse erro é interessante, pois o que ele notifica é que um objeto corrotina não tem o atributo password. Precisamos analisar o código para entender em qual o objeto está buscando password:

```
#fast_zero/routers/auth.py
@router.post('/token', response_model=Token)
def login_for_access_token(form_data: OAuth2Form, session: Session):
    user = session.scalar(select(User).where(User.email == form_data.username))

    # ...

    if not verify_password(form_data.password, user.password):
    # ...
```

Agora precisamos de `await`

```python
@router.post('/token', response_model=Token)
async def login_for_access_token(form_data: OAuth2Form, session: Session):
    user = await session.scalar(
        select(User).where(User.email == form_data.username)
    )
    # ...
```

Tentando de novo

Problemas na fixture de `user`!

```
task test -k test_get_token

# ...

tests/test_auth.py::test_get_token /home/dunossauro/07/tests/conftest.py:75: RuntimeWarning: coroutine 'AsyncSession.commit' was never awaited
  session.commit()
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
/home/dunossauro/07/tests/conftest.py:76: RuntimeWarning: coroutine 'AsyncSession.refresh' was never awaited
  session.refresh(user)
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
PASSED
```

Transformando a fixture e async

Como temos duas interações de I/O com o banco nesse fixture `.commit` e `.refresh`, devemos aguardar as duas:

```
@pytest_asyncio.fixture
async def user(session):
    # ...
    session.add(user)
    await session.commit()
    await session.refresh(user)
    # ...
```

> De novo agora... `task test -k test_get_token`

Corrigindo os tipos

Embora o comportamento do código esteja correto e sem nenhum problema aparente. Precisamos corrigir o tipo usado para injeção de depenências que não é mais Session, mas AsyncSession:

```python
# fast_zero/routers/auth.py
from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession
# ...
OAuth2Form = Annotated[OAuth2PasswordRequestForm, Depends()]
Session = Annotated[AsyncSession, Depends(get_session)]
```

Podemos partir para o router de `users` agora

O cabeçalho

```
# ...
from sqlalchemy.ext.asyncio import AsyncSession
# ...
Session = Annotated[AsyncSession, Depends(get_session)]
```

Endpoint de POST

```
task test -k test_create_user
```

Precisamos adicionar os awaits...

```
async def create_user(user: UserSchema, session: Session):
    db_user = await session.scalar(
        select(User).where(
            (User.username == user.username) | (User.email == user.email)
        )
    )
    # ...
    session.add(db_user)
    await session.commit()
    await session.refresh(db_user)

    return db_user
```

Endpoint de GET

```
task test -k test_read

FAILED tests/test_users.py::test_read_users - AttributeError: 'coroutine' object has no attribute 'all'
```

```python
async def read_users(
    session: Session, filter_users: Annotated[FilterPage, Query()]
):
    query = await session.scalars(
        select(User).offset(filter_users.offset).limit(filter_users.limit)
    )
    users = query.all()

    return {'users': users}
```

Endpoint de PUT

```
task test -k test_update
# ...
FAILED tests/test_users.py::test_update_user - AttributeError: 'coroutine' object has no attribute 'id'
```

A corrotina nesse caso é no `current_user`:

```python
from sqlalchemy.ext.asyncio import AsyncSession
# ...
async def get_current_user(
    session: AsyncSession = Depends(get_session),
    token: str = Depends(oauth2_scheme),
):
    # ...
    user = await session.scalar(
        select(User).where(User.email == subject_email)
    )
```

Agora o PUT de verdade

```
async def update_user(...):
    # ...
    try:
        current_user.username = user.username
        current_user.password = get_password_hash(user.password)
        current_user.email = user.email
        await session.commit()
        await session.refresh(current_user)

        return current_user
```

O DELETE

```python
@router.delete('/{user_id}', response_model=Message)
async def delete_user(...):
    # ...
    await session.delete(current_user)
    await session.commit()
```

```
task test -k test_delete
# ...
tests/test_users.py::test_delete_user PASSED
```

Só pra garantir...

```
task test
```

Cobertura de testes assíncrona

> Parte 5

Vamos olhar o arquivo de cobertura

... Estranho, não?

```
[tool.coverage.run]
concurrency = ["thread", "greenlet"]
```

Vamos tentar de novo

```
task test
```

Agora olhar a cobertura de novo!

Migrações async

> Parte 6

Vamos tentar aplicar a migração

```
alembic upgrade head
# ...
sqlalchemy.exc.MissingGreenlet: greenlet_spawn has not been called;
can't call await_only() here. Was IO attempted in an unexpected place?
(Background on this error at: https://sqlalche.me/e/20/xd2s)
```

O problema do `.env`

Como nosso `.env` aponta para uma conexão async, temos que fazer com que a migração seja async:

```
import asyncio

from logging.config import fileConfig

from sqlalchemy.ext.asyncio import async_engine_from_config
from sqlalchemy import pool

## Vamos reescrever essa função!
def run_migrations_online():
    asyncio.run(run_async_migrations())
```

Executando a conexão async

```
async def run_async_migrations():
    connectable = async_engine_from_config(
        config.get_section(config.config_ini_section),
        prefix="sqlalchemy.",
        poolclass=pool.NullPool,
    )

    async with connectable.connect() as connection:
        await connection.run_sync(do_run_migrations)

    await connectable.dispose()
```

Fazendo a migração async

```
def do_run_migrations(connection):
    context.configure(connection=connection, target_metadata=target_metadata)

    with context.begin_transaction():
        context.run_migrations()
```

Mais uma tentativa

```
alembic upgrade head
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
```

Suplementar / Para próxima aula

Na próxima aula, vamos adicionar randomização em testes para facilitar a criação dos dados de teste. Caso não conheça o Faker ou Factory-boy, pode ser uma boa para entender melhor a próxima aula:

- Randomização de dados em testes unitários com Faker e Factory-boy | Live de Python #281

Exercícios e Quiz

1. Reveja os endpoints criados por você em exercícios anteriores e adicione async e await para que eles se tornem não bloqueantes também.
2. Altere o endpoint read_root para suportar asyncio.

> Não esqueça de responder o quiz

Commit

```
git add .
git commit -m "Refatorando estrutura do projeto: Suporte a asyncio, tornando o projeto não bloqueante"
```