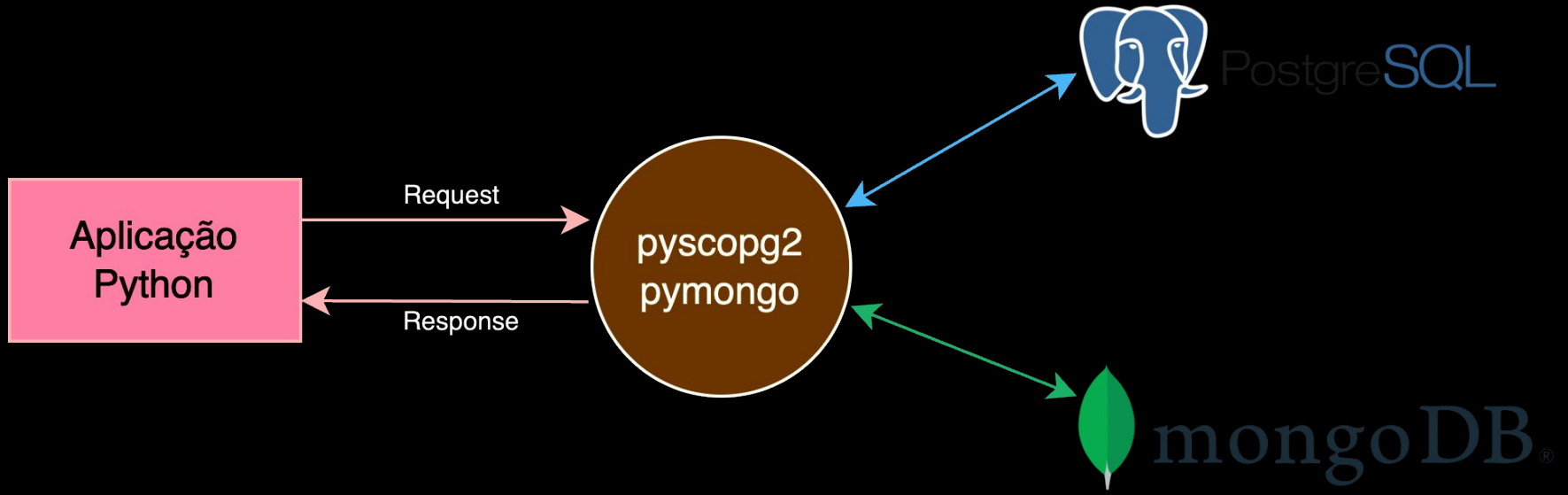
Aula 04

- Integração do Python com PostgreSQL parte 1

Integração do Python com PostgreSQL parte 1

Python Conectores



psycopg2

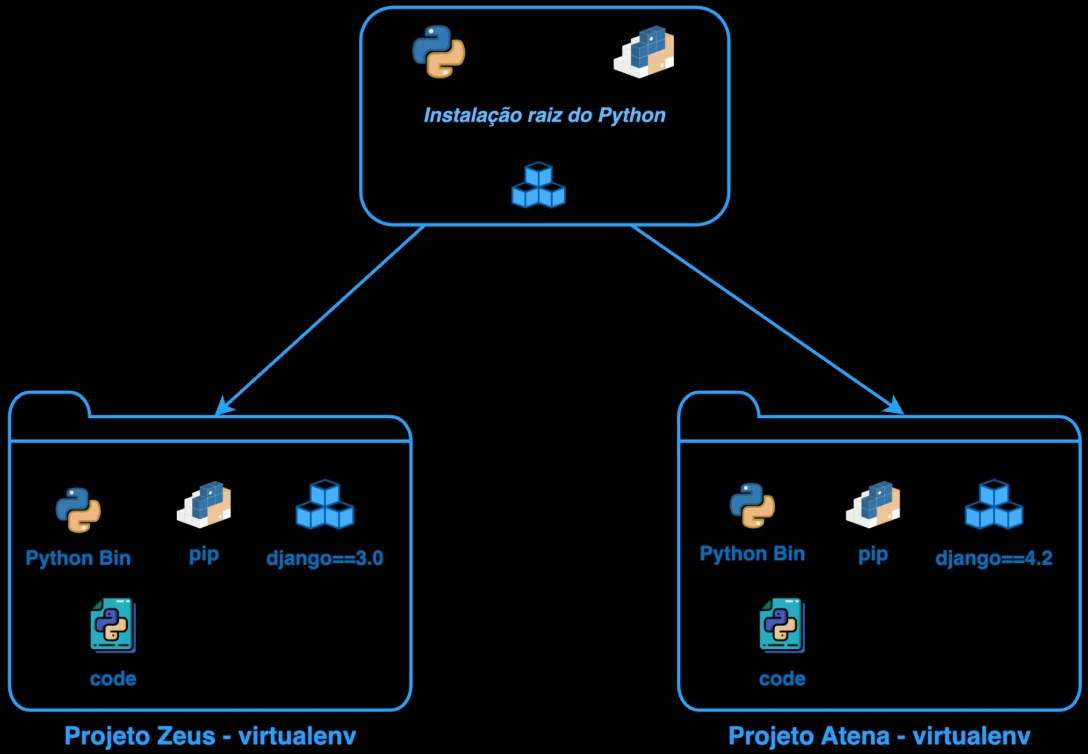
- O psycopg é um adaptador de banco de dados PostgreSQL mais popular no Python;
- Essa versão 2 é uma implementação thread safe, ou seja foi implementada para aplicativo multi threads com uma grande quantidade de "INSERT" ou "UPDATE";
- Você pode encontrar mais detalhes na página da lib no Python Package Index.
- Porém o psycopg2 requer alguns pré-requisitos (um compilador C, alguns pacotes de desenvolvimento), o que resulta em erro de instalação. Para resolver isso, a lib psycopg2-binary foi criada, para que reúna um pacote pré-compilado com todas as bibliotecas necessárias do psycopg2 para evitar conflitos/erros.

psycopg2-binary

- Faça a instalação do psycopg2-binary usando o pip:

```
pip install psycopg2-binary
```

Ambientes virtuais



Ambientes virtuais

- O módulo **venv,** disponibiliza suporte à criação de **ambientes virtuais** leves, cada um com seu próprio conjunto independente de pacotes Python instalados;
- Um ambiente virtual é criado sobre uma instalação existente do Python, conhecido como o Python “base” do ambiente virtual, e pode, opcionalmente, ser isolado dos pacotes no ambiente base, de modo que apenas aqueles explicitamente instalados no ambiente virtual estejam disponíveis.
- A criação de ambientes virtuais é feita executando o comando venv:

Conectando-se ao Postgres com Python

- Para conectar no PostgreSQL com Python precisamos de uma conexão.
- Com o banco de dados já criado, podemos criar uma conexão;
- Crie um novo arquivo conexao_db.py com o seguinte código:

```python
import psycopg2


if __name__ == '__main__':
    conexao = psycopg2.connect(host='localhost', database='livraria', user='postgres', password='postgres')
    print('Conectado ao banco de dados')
    conexao.close()
```

Isolando conexões com o padrão Factory

- Em todos os lugares que vamos nos comunicar com o banco de dados, vamos precisar de uma conexão, e para não espalharmos o mesmo código em diversos lugares, podemos utilizar o padrão Factory e isolar nosso código de conexão para ser utilizados em todos os pontos do sistema que precisem se comunicar com o banco de dados.
- Para isso, vamos alterar nosso módulo conexao_db.py para conexao_factory.py e alterarmos o código:

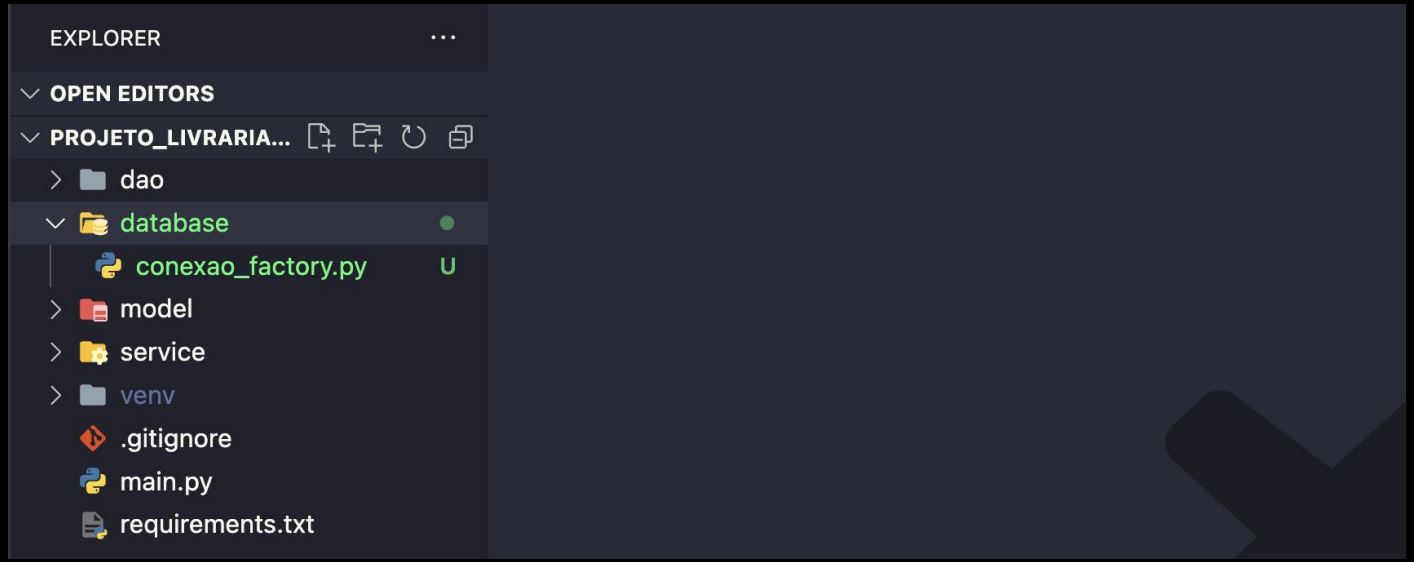
```python
import psycopg2


class ConexaoFactory:

    def get_conexao(self):
        return psycopg2.connect(host='localhost', database='livraria', user='postgres', password='postgres')
```

Camada database

- Para não deixarmos nossa classe ConexaoFactory solta no código, podemos criar uma nova camada no nosso projeto com o nome database, assim mantemos nosso projeto organizado à medida que evoluímos o mesmo:



Alterando model Categoria

- Como deixamos o postgresql gerar o Id automaticamente, precisamos remover o id do construtor do nosso model Categoria e deixar o valor inicial como zero (0) como você pode ver ao lado:

```python
class Categoria:

    def __init__(self, nome: str):
        self.__id: int = 0
        self.__nome: str = nome

    def __str__(self):
        return f'{self.__id} | {self.__nome}'

    @property
    def id(self) -> int:
        return self.__id

    @id.setter
    def id(self, id: int):
        self.__id = id

    @property
    def nome(self) -> str:
        return self.__nome

    @nome.setter
    def nome(self, nome: str):
        self.__nome = nome
```

CategoriaDAO.adicionar

- Agora podemos usar o cursor para salvar uma nova categoria no CategoriaDAO e precisamos fazer o commit.

```python
from model.categoria import Categoria
from database.conexao_factory import ConexaoFactory


class CategoriaDAO:

    def __init__(self):
        self.__conexao_factory = ConexaoFactory()

    # ...

    def adicionar(self, categoria: Categoria) -> None:
        conexao = self.__conexao_factory.get_conexao()
        cursor = conexao.cursor()
        cursor.execute(f"INSERT INTO categorias (nome) VALUES ('{categoria.nome}')")
        conexao.commit()
```

Nos protegendo de SQL Injection

- Da forma que fizemos podemos sofrer com SQL Injection, então precisamos alterar nosso código para utilizar argumentos, que podem ser **posicionais** como na imagem abaixo ou nomeados (próximo slide):

```python
class CategoriaDAO:

    def __init__(self):
        self.__conexao_factory = ConexaoFactory()

    # ...

    def adicionar(self, categoria: Categoria) -> None:
        conexao = self.__conexao_factory.get_conexao()
        cursor = conexao.cursor()
        cursor.execute(f"INSERT INTO categorias (nome) VALUES (%s)", (categoria.nome))
        conexao.commit()
```

Usado argumentos nomeados

- Também podemos utilizar argumentos nomeados como na imagem ao lado, e é a forma que devemos utilizar para facilitar o entendimento do código. Ela é a forma mais utilizada no mercado.

Veja mais em:
https://www.psycopg.org/psycopg3/docs/basic/params.html

```python
class CategoriaDAO:

    def __init__(self):
        self.__conexao_factory = ConexaoFactory()

    # ...

    def adicionar(self, categoria: Categoria) -> None:
        conexao = self.__conexao_factory.get_conexao()
        cursor = conexao.cursor()
        cursor.execute("""
            INSERT INTO categorias (nome) VALUES (%(nome)s)
            """,
            ({'nome': categoria.nome, }))
        conexao.commit()
```

Fechando recursos

- Tudo certo, conseguimos salvar nossas categorias no banco de dados, porém o nosso código ainda não ficou completo, precisamos sempre fechar os recursos, no nosso caso o cursor e a conexão, porque do contrário isso pode ser altamente prejudicial quando nosso projeto estiver em produção.

```python
from model.categoria import Categoria
from database.conexao_factory import ConexaoFactory


class CategoriaDAO:

    def __init__(self):
        self.__conexao_factory = ConexaoFactory()

    # ...

    def adicionar(self, categoria: Categoria) -> None:
        conexao = self.__conexao_factory.get_conexao()
        cursor = conexao.cursor()
        cursor.execute("""
            INSERT INTO categorias (nome) VALUES (%(nome)s)
            """,
            ({'nome': categoria.nome, }))
        conexao.commit()
        cursor.close() # novo
        conexao.close() # novo
```

Alterando o CategoriaService.adicionar

- Precisamos alterar o método adicionar do CategoriaService para deixamos de lidar com id e assim passar essa responsabilidade para o banco de dados.

```python
class CategoriaService:

    # ...

    def adicionar(self):
        print('\nAdicionando categoria ... ')

        try:
            nome = input('Digite o nome da categoria: ')
            nova_categoria = Categoria(nome)
            self.__categoria_dao.adicionar(nova_categoria)
            print('Categoria adicionada com sucesso!')
        except Exception as e:
            print(f'Erro ao inserir categoria! - {e}')
            return

        input('Pressione uma tecla para continuar ... ')
```

Alterando o listar da CategoriaDAO

- Agora precisamos utilizar a nossa conexão, junto com o cursor, para executar o comando SELECT no banco de dados e em seguida geramos assim nossa lista de categorias salvas no banco de dados. Veja que não alteramos o tipo de retorno do método, já que ainda precisamos retornar um list[Categoria] para o service.

- O 'fetchall' retorna os resultados como uma lista de tuplas, ou uma lista vazia se não existir nenhum registro.

```python
from model.categoria import Categoria
from database.conexao_factory import ConexaoFactory


class CategoriaDAO:

    def __init__(self):
        self.__conexao_factory = ConexaoFactory()

    def listar(self) -> list[Categoria]:
        categorias = list()

        conexao = self.__conexao_factory.get_conexao()
        cursor = conexao.cursor()
        cursor.execute("SELECT id, nome FROM categorias")
        resultados = cursor.fetchall()
        for resultado in resultados:
            cat = Categoria(resultado[1])
            cat.id = resultado[0]
            categorias.append(cat)
        cursor.close()
        conexao.close()

        return categorias
```

CategoriaDAO.remover

- Agora podemos implementar o método remover do CategoriaDAO. Veja que utilizamos o rowcount para saber se teve alguma linha excluída da tabela ou não.

```python
class CategoriaDAO:

    # ...

    def remover(self, categoria_id: int) -> bool:
        conexao = self.__conexao_factory.get_conexao()
        cursor = conexao.cursor()
        cursor.execute("DELETE FROM categorias WHERE id = %s", (categoria_id,))
        categorias_removidas = cursor.rowcount
        conexao.commit()
        cursor.close()
        conexao.close()

        if (categorias_removidas == 0):
            return False
        return True
```

CategoriaDAO.buscar_por_id

- Agora para concluir todas as funcionalidade do CategoriaDAO, precisamos implementar a busca por id.

```python
class CategoriaDAO:

    # ...

    def buscar_por_id(self, categoria_id) -> Categoria:
        cat = None
        conexao = self.__conexao_factory.get_conexao()
        cursor = conexao.cursor()
        cursor.execute("SELECT id, nome FROM categorias WHERE id = %s", (categoria_id,))
        resultado = cursor.fetchone()
        if (resultado):
            cat = Categoria(resultado[1])
            cat.id = resultado[0]
        cursor.close()
        conexao.close()
        return cat
```