O guião de laboratório de Desenvolvimento Web e Transações é uma continuação dos guiões de Introdução às Bases de Dados e de SQL. Já deverá, portanto, ter uma base de dados do exemplo do Banco criada. Se ainda não criou esta base de dados deverá seguir as instruções no Lab 1.

1 Create, Read, Update and Delete (CRUD)

1.1 Bases de Dados

O acrónimo CRUD refere-se às principais operações implementadas por bases de dados. Cada letra do acrónimo corresponde a um statement de Structured Query Language (SQL).

CRUD	SQL
Create	INSERT
Read	SELECT
Update	UPDATE
Delete	DELETE

Podem ser implementadas com bases de dados relacionais, bases de dados de documentos, bases de dados de objetos, bases de dados XML, ficheiros de texto ou ficheiros binários.

1.2 RESTful APIs

O acrónimo CRUD é também usado em APIs RESTful. Cada letra do acrónimo pode corresponder a um método de Hypertext Transfer Protocol (HTTP):

CRUD	HTTP
Create	POST, PUT se tivermos `id` ou `uuid`
Read	GET
Update	PUT para substituir, PATCH para modificar
Delete	DELETE

Em HTTP, os métodos GET (leitura), PUT (criação e atualização), POST (criação - caso não tenhamos `id` ou `uuid`) e DELETE (eliminação) são operações CRUD.

2 Flask (Web Framework)

Flask é uma microframework web escrita em Python. É classificado como uma microframework porque não requer ferramentas ou bibliotecas específicas. O Flask não possui uma camada de abstração de base de dados, validação de formulários ou outros componentes onde bibliotecas de terceiros pré-existentes fornecem funções comuns.

CRUD	Flask
Create	POST
Read	GET
Update	POST
Delete	POST

Recomenda-se a leitura da documentação oficial em Flask Quickstart.

3 Implementação RESTful API em Flask

Existe uma implementação inicial da funcionalidade REST em https://github.com/bdist/bankapi

1. Abra o Terminal.
2. Altere o directório atual para a pasta anterior ao `bdist-workspace/`.
3. Execute este comando para criar o seu clone local do projeto `bankapi` na pasta `app`

```
1 git clone https://github.com/bdist/bankapi.git app
```

4. Altere o directório atual para a pasta `bdist-workspace/`

```
1 cd bdist-workspace/
```

5. Nesse directório, inicie o `bdist-workspace` desta vez utilizando o ficheiro de configuração `docker-compose..app.yaml` que permite lançar a `app` juntamente com o restante ambiente de desenvolvimento

```
1 docker compose -f docker-compose..app.yml up
```

6. Instale o Bruno (https://www.usebruno.com)
7. Crie uma nova Colection com o nome `bankapi` (selecione a pasta `app`)

8. Crie e execute o seu primeiro pedido REST (selecione New Request)

   Request Method

   ```
   GET
   ```

   Request URL

   ```
   http://127.0.0.1:8080/accounts/A-101/update
   ```

   Response Code

   ```
   200
   ```

   Response Body

   ```
   ["A-101", "Downtown", "500.0000"]
   ```

3.1 app.py - Preambulo

Neste preambulo estão os **`import`** das bibliotecas que queremos utilizar `flask` e `psycopg`.

Os métodos de fetch do `Cursor`, por omissão, devolvem os registos recebidos a partir da base de dados como tuplos. Isto pode ser alterado para adequar-se melhor às necessidade do programador através da utilização de *row factories* alternativas.

Note a escolha de `namedtuple_row` feita no seguinte **`import`**:

```
from psycopg.rows import namedtuple_row
```

Nota: Uma outra opção popular para *row factory* é `dict_row`.

```
import os
from logging.config import dictConfig

import psycopg
from flask import Flask, jsonify, request
from psycopg.rows import namedtuple_row


# DATABASE_URL environment variable if it exists, otherwise use this.
# Format postgres://username:password@hostname/database_name.
DATABASE_URL = os.environ.get("DATABASE_URL", "postgres://bank:
    bank@postgres/bank")
```

Nota: Deve alterar os componentes de `DATABASE_URL` de acordo.

3.2 Read - GET and SELECT - Consultar registos base de dados

3.2.1 app.py - account_index

As aplicações web modernas utilizam URLs com significado para ajudar os utlizadores. Os utlizadores têm maior probabilidade de gostar de uma página e retornar a ela se a página utilizar um URL com significado que eles se possam lembrar e usar para visitar diretamente a página.

Utilize o decorador *route()* para vincular uma função a um URL. É possível associar várias regras a uma função. Repare na associação da raiz do site `/` bem como `/account` ao método *account_index*.

Repare na utilização do `.fetchall()` para ir buscar à base de dados, de uma só vez, todos registos de `account`. Quais são as implicações na quantidade de memória utilizada? Qual a solução a adotar se `account` tiver milhares de registos?

```
@app.route("/", methods=("GET",))
@app.route("/accounts", methods=("GET",))
def account_index():
    """Show all the accounts, most recent first."""

    with psycopg.connect(conninfo=DATABASE_URL) as conn:
        with conn.cursor(row_factory=namedtuple_row) as cur:
            accounts = cur.execute(
                """
                SELECT account_number, branch_name, balance
                FROM account
                ORDER BY account_number DESC;
                """,
                {},
            ).fetchall()
            log.debug(f"Found {cur.rowcount} rows.")

    return jsonify(accounts)
```

3.3 Update - POST and UPDATE - Atualizar registos da base de dados

É possível tornar partes do URL dinâmicas. Repare no parâmetro na route seguinte.

Nota: É importante validar sempre os dados do cliente no servidor, mesmo que o cliente faça também alguma validação.

Listing 1: app.py - account_update GET

```
@app.route("/accounts/<account_number>/update", methods=("GET",))
def account_update_view(account_number):
    """Show the page to update the account balance."""
```

```
4
5       with psycopg.connect(conninfo=DATABASE_URL) as conn:
6           with conn.cursor(row_factory=namedtuple_row) as cur:
7               account = cur.execute(
8                   """
9                   SELECT account_number, branch_name, balance
10                  FROM account
11                  WHERE account_number = %(account_number)s;
12                  """,
13                  {"account_number": account_number},
14              ).fetchone()
15              log.debug(f"Found {cur.rowcount} rows.")
16
17      return jsonify(account)
```

Listing 2: app.py - account_update POST

```
1  @app.route("/accounts/<account_number>/update", methods=("POST",))
2  def account_update_save(account_number):
3      """Update the account balance."""
4
5      balance = request.args.get("balance")
6
7      error = None
8
9      if not balance:
10         error = "Balance is required."
11     if not is_decimal(balance):
12         error = "Balance is required to be decimal."
13
14     if error is not None:
15         return error, 400
16     else:
17         with psycopg.connect(conninfo=DATABASE_URL) as conn:
18             with conn.cursor(row_factory=namedtuple_row) as cur:
19                 cur.execute(
20                     """
21                     UPDATE account
22                     SET balance = %(balance)s
23                     WHERE account_number = %(account_number)s;
24                     """,
25                     {"account_number": account_number, "balance":
                           balance},
26                 )
27             conn.commit()
28         return "", 204
```

3.4 Delete - POST and DELETE - Apagar registos da base de dados

Listing 3: app.py - account_delete

```
@app.route("/accounts/<account_number>/delete", methods=("POST",))
def account_delete(account_number):
    """Delete the account."""

    with psycopg.connect(conninfo=DATABASE_URL) as conn:
        with conn.cursor(row_factory=namedtuple_row) as cur:
            cur.execute(
                """
                DELETE FROM account
                WHERE account_number = %(account_number)s;
                """,
                {"account_number": account_number},
            )
        conn.commit()
    return "", 204
```

4 Atualizações Concorrentes e Transações

4.1 Prova de conceito

1. Abra um primeiro terminal `psql` e aceda à base de dados `bank.sql`.
2. Escreva uma consulta para obter os dados das contas do cliente `Cook`.
3. O cliente `Cook` pretende transferir *500 EUR* da conta `A-102` para a conta `A-101`.

   Inicie uma transação com o comando:

   ```
   START TRANSACTION;
   ```

4. Coloque *500 EUR* na conta `A-101`.
5. Consulte os saldos das contas do cliente `Cook`. Neste momento, qual é o `balance` do cliente `Cook`?
6. Abra um segundo terminal `psql` e ligue-se à mesma base de dados enquanto esta transação permanece em curso no primeiro terminal.
7. No segundo terminal, consulte os saldos das contas do cliente `Cook`.
8. No primeiro terminal, onde está a transação em curso, retire *500 EUR* da conta `A-102`.
9. No segundo terminal, consulte os saldos das contas do cliente `Cook`.
10. No primeiro terminal, confirme a transação que permanece em curso.

    Confirme a transação em curso com o comando.

```
COMMIT;
```

11. No segundo terminal, consulte os saldos das contas do cliente Cook. Confirme os resultados da transação.

4.2 Implementação de atualizações Concorrentes e Transações com Flask/Psycopg

1. Implemente o mecanismo de transações no Flask/Psycopg.
2. O cliente Cook acabou de abastecer *25 EUR* de gasolina numa estação de serviço e vai pagar com multibanco (conta A-101).
3. Ao mesmo tempo, a sua esposa, que tem um cartão multibanco para a mesma conta, vai levantar *50 EUR* numa máquina multibanco.
4. Abra um terminal para a base de dados e inicie uma transação.
5. Abra o Bruno e prepare um pedido POST para o endpoint update
6. No terminal retire *25 EUR* da conta (abastecimento).
7. No Bruno retire *50 EUR* da conta (levantamento). O que se passa quando tenta fazer isto? Porquê?
8. A linha telefónica da estação de serviço está intermitente e a ligação cai. O pagamento que estava a ser feito tem de ser cancelado. Cancele essa transação. (ROLLBACK)
9. Ao mesmo tempo que faz a alínea 8, repare no que acontece no Bruno. Como explica este fenómeno?
10. Consulte novamente os saldos das contas do cliente Cook.