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.

2.1 Templates

Os ficheiros de template serão armazenados no diretório `templates` dentro do pacote `app`. Os templates são ficheiros que contêm dados estáticos, bem como espaços reservados para dados dinâmicos. Um template é renderizado com dados específicos para produzir um documento final. O Flask utiliza a biblioteca de templates Jinja para renderizar os templates.

Na sua aplicação, vai utilizar templates para renderizar HTML, que será exibido no navegador do utilizador. No Flask, o Jinja é configurado para autoescape quaisquer dados que sejam renderizados nos templates HTML. Isto significa que é seguro renderizar input do utilizador; quaisqueres caracteres que eles tenham inserido que possam interferir com o HTML, como < and >, serão escaped com valores seguros que aparecerão da mesma forma no navegador, mas não causarão efeitos indesejados.

O Jinja tem uma aparência e comportamento bastante semelhantes ao Python. São utilizados delimitadores especiais para distinguir a sintaxe do Jinja dos dados estáticos no template. Qualquer coisa entre `{{` and `}}` é uma expressão que será incluída no documento final. `{%` and `%}` indica uma declaração de fluxo de controle, como `if` e `for`. Ao contrário do Python, os blocos são indicados por tags de início e fim, em vez de indentação, uma vez que o texto estático dentro de um bloco pode alterar a indentação.

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

2.1.1 O Layout Base

Para manter os outros organizados, os templates para um blueprint (i.e., sub-área da app) serão colocados num diretório com o mesmo nome do blueprint (e.g., account).

Cada página na aplicação terá o mesmo layout básico em torno de um corpo diferente.

O template base está no diretório `templates`.

Listing 1: app/templates/base.html

```
<!doctype html>
<title>{% block title %}{% endblock %} - Bank</title>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<nav>
  <h1><a href="/">Bank</a></h1>
  <ul>
    <li><a href="{{ url_for('account_index') }}">Accounts</a>
    </ul>
  </nav>
  <section class="content">
    <header>
      {% block header %}{% endblock %}
    </header>
    {% for message in get_flashed_messages() %}
      <div class="flash">{{ message }}</div>
    {% endfor %}
    {% block content %}{% endblock %}
  </section>
```

Depois do título da página e antes do conteúdo, o modelo faz um loop em cada mensagem retornada por *get_flashed_messages()*.

Pode utilizar *flash()* nas views para exibir mensagens de erro, e este é o código que irá mostrá-las.

Em vez de escrever toda a estrutura HTML em cada template, cada template irá estender um template base e substituir seções específicas.

Existem três blocos definidos que serão substituídos nos outros templates:

- `{% block title %}` vai alterar o título exibido na tab do browser e no título da janela.
- `{% block header %}` é semelhante a `title`, mas irá alterar o título exibido na página.
- `{% block content %}` é onde o conteúdo de cada página é colocado, como o formulário de login ou um post de blog.

2.1.2 Herança de Templates

Listing 2: app/templates/account/index.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Accounts{% endblock %}</h1>
{% endblock %}

{% block content %}
  {% for account in accounts %}
    <article class="post">
      <header>
        <div>
          <h1>{{ account['account_number'] }}</h1>
          <div class="about">in {{ account['branch_name'] }}</div>
        </div>
        <a class="action" href="{{ url_for('account_update',
           account_number=account['account_number']) }}">Edit</a>
      </header>
      <p class="body">  {{ account['balance'] }}</p>
    </article>
    {% if not loop.last %}
      <hr>
    {% endif %}
  {% endfor %}
{% endblock %}
```

`{% extends 'base.html'%}` informa o Jinja que este template deve substituir os blocos do template base. Todo o conteúdo renderizado deve aparecer dentro das tags `{% block %}` que substituem os blocos do template base.

Um padrão útil utilizado aqui é colocar `{% block title %}` dentro de `{% block header %}`. Isso define o bloco de título e depois exibe o valor dele no bloco de cabeçalho, para que tanto a janela quanto a página partilhem o mesmo título sem o escrever duas vezes.

As tags `input` usam o atributo `required` aqui. Isto instrui o navegador a não enviar o formulário até que esses campos sejam preenchidos. Se o utilizador estiver a usar um navegador mais antigo que não suporte esse atributo, ou se estiver a usar algo que não um navegador para fazer solicitações, queremos ainda assim validar os dados na view do Flask.

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

2.2 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`**:

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

Nota: `dict_row` é uma alternativa popular.

Este preambulo faz também setup de uma pool de ligações (`ConnectionPool`).

```
1 from psycopg_pool import ConnectionPool
2
3 pool = ConnectionPool(conninfo=DATABASE_URL)
4 # the pool starts connecting immediately.
```

Mantemos assim automaticamente uma pool de instâncias de `Connection` já preparadas para receber comandos SQL imediatamente sem precisar de re-estabelecer a ligação ao servidor de base de dados.

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

```
1  #!/usr/bin/python3
2  import psycopg
3  from flask import flash
4  from flask import Flask
5  from flask import jsonify
6  from flask import redirect
7  from flask import render_template
8  from flask import request
9  from flask import url_for
10 from psycopg.rows import namedtuple_row
11 from psycopg_pool import ConnectionPool
12
13
14 # postgres://{user}:{password}@{hostname}:{port}/{database-name}
15 DATABASE_URL = "postgres://db:db@postgres/db"
16
17 pool = ConnectionPool(conninfo=DATABASE_URL)
18 # the pool starts connecting immediately.
19
20 app = Flask(__name__)
21 log = app.logger
```

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

2.3.1 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 em termos da memória utilizada pela app? Se `account` tivesse milhares de registos?

Reimplemente o endpoint de maneira a que se utilize o mínimo de memória pela `app`.

Pista: Remova o *.fetchall()* e inspecione o valor de `cur.rowcount` nos logs da aplicação.

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

    with pool.connection() 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.")

    # API-like response is returned to clients that request JSON
       explicitly (e.g., fetch)
    if (
        request.accept_mimetypes["application/json"]
        and not request.accept_mimetypes["text/html"]
    ):
        return jsonify(accounts)

    return render_template("account/index.html", accounts=accounts)
```

2.3.2 RESTful APIs e Client-Side Scripting

Este `endpoint` está preparado para devolver `JSON Responses` a clientes que assim o requererem. Por exemplo, o comando seguinte deverá fazer output da lista de accounts para o seu terminal no formato JSON.

```
$ curl http://127.0.0.1:5001/accounts -H "Accept: application/json"
```

Nota: Pode utilizar o Postman como uma alternativa com GUI.

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

Listing 3: app/templates/account/update.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Account {{ account['account_number'] }} | Edit{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="account_number">Account Number</label>
    <input name="account_number" id="account_number" type="text" value="{{ request.form['account_number'] or account['account_number'] }}" disabled>
    <label for="balance">Balance</label>
    <input name="balance" id="balance" type="number" min="0" step="0.0001" placeholder="0.0000" value="{{ request.form['balance'] or account['balance'] }}" required>
    <input type="submit" value="Save">
  </form>
  <hr>
  <form action="{{ url_for('account_delete', account_number=account['account_number']) }}" method="post">
    <input class="danger" type="submit" value="Delete" onclick="return confirm('Are you sure?');">
  </form>
{% endblock %}
```

2.4.1 app.py - account_update

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

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

```
@app.route("/accounts/<account_number>/update", methods=("GET", "POST"))
def account_update(account_number):
    """Update the account balance."""

    with pool.connection() as conn:
        with conn.cursor(row_factory=namedtuple_row) as cur:
            account = cur.execute(
                """
                SELECT account_number, branch_name, balance
                FROM account
                WHERE account_number = %(account_number)s;
                """,
                {"account_number": account_number},
            ).fetchone()
            log.debug(f"Found {cur.rowcount} rows.")

    if request.method == "POST":
        balance = request.form["balance"]

        error = None

        if not balance:
            error = "Balance is required."
            if not balance.isnumeric():
                error = "Balance is required to be numeric."

        if error is not None:
            flash(error)
        else:
            with pool.connection() as conn:
                with conn.cursor(row_factory=namedtuple_row) as cur:
                    cur.execute(
                        """
                        UPDATE account
                        SET balance = %(balance)s
                        WHERE account_number = %(account_number)s;
                        """,
                        {"account_number": account_number, "balance":
                            balance},
                    )
                conn.commit()
            return redirect(url_for("account_index"))

    return render_template("account/update.html", account=account)
```

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

Não será então definida qualquer template ou página após um `Delete`. Iremos apenas redirecionar o utilizador para a página principal. Podemos fazer melhor?

2.5.1 app.py - account_delete

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

    with pool.connection() 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 redirect(url_for("account_index"))
```

3 Atualizações Concorrentes e Transações

3.1 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` da conta `A-102` para a conta `A-101`.

   Inicie uma transação com o comando:

   ```
   START TRANSACTION;
   ```

4. Coloque `500` 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 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.

    ```
    1 COMMIT;
    ```

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

3.2 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€ 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€ numa máquina multibanco.
4. Abra um terminal para a base de dados e inicie uma transação.
5. Abra uma janela do browser para a página http://web2.ist.utl.pt/istxxxxxx/app.cgi/accounts
6. No terminal retire 25€ da conta (abastecimento).
7. No browser retire 50€ 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. (Garantir o ROLLBACK)
9. Ao mesmo tempo que faz a alínea 8, repare no que acontece no browser. Como explica este fenómeno?
10. Consulte novamente os saldos das contas do cliente Cook.