This guide about Web Development and Transactions is part of the guides for the Information Systems and Databases course. At this point, you should already have the Bank's sample database created. If you need to create the Bank database, you can do so following the instructions in Lab 1.

1 Create, Read, Update and Delete (CRUD)

1.1 Databases

The acronym CRUD refers to the major operations which are implemented by databases. Each letter in the acronym can be mapped to a standard Structured Query Language (SQL) statement.

CRUD	SQL
Create	INSERT
Read	SELECT
Update	UPDATE
Delete	DELETE

CRUD functionality can for example be implemented with document databases, object databases, XML databases, text files, or binary files.

1.2 RESTful APIs

The acronym CRUD also appears in the discussion of RESTful APIs. Each letter in the acronym may be mapped to a Hypertext Transfer Protocol (HTTP) method:

CRUD	HTTP
Create	POST, PUT if we have `id` or `uuid`
Read	GET
Update	PUT to replace, PATCH to modify
Delete	DELETE

In HTTP, the GET (read), PUT (create and update), POST (create - if we don't have `id` or `uuid`), and DELETE (delete) methods are CRUD operations.

2 Flask (Web Framework)

Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions.

CRUD	Flask
Create	POST
Read	GET
Update	POST
Delete	POST

We recommend reading the official documention at Flask Quickstart.

2.1 Templates

The template files will be stored in the `templates` directory inside the `app` package. Templates are files that contain static data as well as placeholders for dynamic data. A template is rendered with specific data to produce a final document. Flask uses the Jinja template library to render templates.

In your application, you will use templates to render HTML which will display in the user's browser.

In Flask, Jinja is configured to autoescape any data that is rendered in HTML templates. This means that it's safe to render user input; any characters they've entered that could mess with the HTML, such as < and > will be escaped with safe values that look the same in the browser but don't cause unwanted effects.

Jinja looks and behaves mostly like Python. Special delimiters are used to distinguish Jinja syntax from the static data in the template. Anything between `{{` and `}}` is an expression that will be output to the final document. `{%` and `%}` denotes a control flow statement like `if` and `for`. Unlike Python, blocks are denoted by start and end tags rather than indentation since static text within a block could change indentation.

We recommend reading the official documention at Flask Templates.

2.1.1 Base Layout

To keep the others organized, the templates for a blueprint will be placed in a directory with the same name as the blueprint.

Each page in the application will have the same basic layout around a different body.

The base template is directly in the `templates` directory.

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>
```

After the page title, and before the content, the template loops over each message returned by *get_flashed_messages()*.

You used *flash()* in the views to show error messages, and this is the code that will display them.

Instead of writing the entire HTML structure in each template, each template will extend a base template and override specific sections.

There are three blocks defined here that will be overridden in the other templates:

- `{% block title %}` will change the title displayed in the browser's tab and window title.
- `{% block header %}` is similar to `title` but will change the title displayed on the page.
- `{% block content %}` is where the content of each page goes, such as the login form or a blog post.

2.1.2 Template Inheritance

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">EUR {{ account['balance'] }}</p>
    </article>
    {% if not loop.last %}
      <hr>
    {% endif %}
  {% endfor %}
{% endblock %}
```

`{% extends 'base.html'%}` tells Jinja that this template should replace the blocks from the base template. All the rendered content must appear inside `{% block %}` tags that override blocks from the base template.

A useful pattern used here is to place `{% block title %}` inside `{% block header %}`. This will set the title block and then output the value of it into the header block, so that both the window and page share the same title without writing it twice.

The `input` tags are using the `required` attribute here. This tells the browser not to submit the form until those fields are filled in. If the user is using an older browser that doesn't support that attribute, or if they are using something besides a browser to make requests, you still want to validate the data in the Flask view.

It's important to always fully validate the data on the server, even if the client does some validation as well.

2.2 app.py - Preambulo

In the preamble we find the **`import`** statements for the libraries that we are going to use `flask` e `psycopg`.

The fetch methods for the `Cursor` object, encapsulate the records returned from the database as Python tuples. This behaviour can be adjusted by the programmer as needed by choosing different *row factories*.

Note the choice of `namedtuple_row` at the following **`import`**:

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

Note: Another popular option for *row factory* is `dict_row`. In this course our choice is to use `namedtuple_row`.

```
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
12
13 # postgres://{user}:{password}@{hostname}:{port}/{database-name}
14 DATABASE_URL = os.environ.get("DATABASE_URL", "postgres://db:
       db@postgres/db")
```

Note: You should adjust the `DATABASE_URL` components if needed.

2.3 Read - GET and SELECT - Read records from the database

2.3.1 app.py - account_index

Modern web applications use meaningful URLs to help users. Users are more likely to like a page and come back if the page uses a meaningful URL they can remember and use to directly visit a page.

Use the *route()* decorator to bind a function to a URL. You can attach multiple rules to a function. Take notice of the attachment of the root of the website `/` as well as `/account` to the *account_index* method.

Notice the use of `.fetchall()` to fetch from the database, at once, all the records in the `account` table. What are the implications in terms of app memory usage? What if the `account` table had thousands of records?

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

2.4 Update - POST and UPDATE - Update records in the database

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>
```

```
16    <form action="{{ url_for('account_delete', account_number=account['
          account_number']) }}" method="post">
17      <input class="danger" type="submit" value="Delete" onclick="return
            confirm('Are you sure?');">
18    </form>
19  {% endblock %}
```

You can make parts of the URL dynamic. Take notice of the <account_number> parameter in the following route.

Note: It's important to always fully validate the data on the server, even if the client does some validation as well.

Listing 4: app.py - account_update GET

```
1  @app.route("/accounts/<account_number>/update", methods=("GET",))
2  def account_update_view(account_number):
3      """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 render_template("account/update.html", account=account)
```

Listing 5: 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.form["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         flash(error)
```

```
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 redirect(url_for("account_index"))
```

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

We have not defined a template or page to present after a `Delete`. We simply redirect the user to the front page without any confirmation. Can we do better?

Listing 6: app.py - account_delete

```
1  @app.route("/accounts/<account_number>/delete", methods=("POST",))
2  def account_delete(account_number):
3      """Delete the account."""
4
5      with psycopg.connect(conninfo=DATABASE_URL) as conn:
6          with conn.cursor(row_factory=namedtuple_row) as cur:
7              cur.execute(
8                  """
9                  DELETE FROM account
10                 WHERE account_number = %(account_number)s;
11                 """,
12                 {"account_number": account_number},
13             )
14         conn.commit()
15     return redirect(url_for("account_index"))
```

3 Concurrent Updates and Transactions

3.1 Proof of Concept

1. Open the first terminal and use `psql` to connect to the `bank.sql` database.
2. Write a query to display the data for the accounts of customer `Cook`.

3. The customer Cook wants to transfer *500 EUR* from account A-102 to account A-101.

   Start a transaction with command:

   ```
   START TRANSACTION;
   ```

4. Add *500 EUR* in the account A-101.
5. Check the account balances for customer Cook. At this point, what is the total balance for customer Cook?
6. Open a second terminal psql and connect to the same database while keeping the transaction in progress in the first terminal.
7. In the second terminal, check the account balances for customer Cook.
8. In the first terminal, where the transaction is still underway, subtract *500 EUR* from account A-102.
9. In the second terminal, check the account balances for customer Cook.
10. In the first terminal, confirm the transaction underway.

    Confirm the transaction underway with the command.

    ```
    COMMIT;
    ```

11. In the second terminal, check the account balances for customer Cook. Confirm that the results of the transaction are now visible.

3.2 Implementing Transactions with Flask/Psycopg

1. The customer Cook has just refilled *25 EUR* of fuel at a service station and will pay by ATM (account A-101).
2. At the same time, his wife, who has an ATM card for the same account, will withdraw *50 EUR* from an ATM machine on the other side of town.
3. Open a terminal and use psql to connect to the bank.sql database.

   Start a transaction with command:

   ```
   START TRANSACTION;
   ```

4. Open a browser window to page http://127.0.0.1:5001/accounts
5. In the terminal withdraw *25 EUR* from account A-101 (fuel).

6. In the browser withdraw *50 EUR* from account `A-101` (withdrawal). What happens when you try to do this? Why?
7. Suppose that, at the gas station, the power just went down, and the connection is dropped. The payment that was being made will be rolled back by the database due to a connection timeout.

   To simulate this behaviour, cancel this transaction explicitly. Cancel the transaction with command:

   ```
   1  ROLLBACK;
   ```

8. At the same time as step 7, note what happens in the browser. How do you explain this phenomenon?
9. Once again, check the account balances for customer Cook.