Lab: Web development in Python with Flask

In this lab, we are going to build a web application that displays information using HTML. We are going to use the `flask` library. You must install it with pip first: `pip install flask`.

Understand Flask

Flask is a web micro-framework that allows Python developers to very quickly create web applications and APIs with minimal code involved.

Basic structure of a Flask application

The minimal version of a Flask application is:

```
from flask import Flask

app = Flask(__name__)

if __name__ == "__main__":
    app.run(debug=True, port=8888)
```

This will run a Flask web application, in debug mode, listening on port 8888. Run it with: `python app.py`.

> ⚠️ By default, the application listens only on the local interfaces, so it is not available to anyone else but you. This also means that you must access it using `localhost` or `127.0.0.1`.

> 💡 Running the app in debug mode sets up live reload: the app will reload itself whenever you make changes to the code. Make sure you reload the web pages in your browser, though...

> ⚠️ Make sure you follow the steps in the lab, unless **you really know what you are doing**. For example, the application could be run with `flask run`. However, in order for it to work well and consistently, you would need a full understanding of Flask, your terminal settings, your Python installation, and your system configuration which you are very unlikely to have at this stage. Just don't do it.

Understanding routes and URLs

The `app` variable is a Flask instance and can be used to interact with Web methods and HTTP requests. You can use it to create "routes" in your application that return data to the web browser. A Flask route is just a Python function, that returns at least one value, typically a string (= the HTML code to be displayed in the browser).

```
@app.route("/")
def home():
    return "HEY THERE"
```

The function above will be called whenever an HTTP request is made on the route `/` (which is the default URL). If you browse to `http://127.0.0.1:8888/` you should see the text above displayed in the browser. Try to change it and refresh the browser page. Try to change the string to HTML code.

Using templates

It is very inconvenient to return raw HTML directly from the Python code, and it breaks best practices (separation of code and presentation). It is much better to use *templates*, aka HTML documents whose structure is fully created and complete. We can then just "fill" them with the information we want.

Flask looks for templates in the `templates` folder by default. You can render a template by calling the `render_template` function from Flask, and then return the value to the browser. Make sure you import it first!

```
from flask import Flask, render_template

@app.route("/")
def home():
    return render_template("home.html")
```

You can specify where "placeholders" should be in your template, and provide values for the placeholders in the `render_template` function.

```
<h3>My name is {{ name }}</h3>
```

With the HTML template above, `render_template` will replace `{{ name }}` with the value provided as the keyword argument `name=`. For example: `render_template("home.html", name="Tim")`.

`{{ ... }}` will update the template with the result of the Python expression provided. It is usually a variable.

You can also run logic by using `{% ... %}`. For example, the following will loop on the list `my_list`:

```
<ul>
    {% for element in my_list %}
        <li>List element: {{ element }}</p>
    {% endfor %}
</ul>
```

Write your first views

- Create a new route in your Flask application: `/customers`.

- When accessed, this route must:
  - read from the `customers.csv` file
  - display a list of all customers in HTML
- Create a new route in your Flask application: `/products`.
- When accessed, this route must:
  - read from the `products.csv` file
  - display a list of all products and their price using HTML

Use blocks and template scaffolding

It is very inconvenient to write a complete HTML file for every view / function we have in Flask. It is very likely that many if not all of our pages will have the same structure, and only specific "areas" (or "blocks") of the page will change.

This can be done by using *template scaffolding*, where you can build templates based on other templates.

Take a look at the template `base.html`. You can see that it contains a basic HTML document, with a specific instruction: `{% block content %}{% endblock %}`. This defines a new *block* that can be overriden by a child template.

Now, take a look at the template `customers.html`. You can see that the template is very short.

1. It *extends* the base template with `{% extends "base.html" %}`.
2. It overrides the `content` block with its own data.

Make changes to the `base.html` file, and the templates you already wrote to use template scaffolding.

> 💡 You may want to add styling to your HTML. Define CSS in the `<head>` of your base HTML template, and add classes / IDs as required to the HTML code.

Detailed instructions

- Create the `templates` and `data` folder in your Flask application folder.
- Move the files `base.html` and `customers.html` to the `templates` folder.
- Move the CSV files to the `data` folder.
- Look at the `customers.html` template.
  - It extends `base.html`.
  - It works with a variable called `customers`. The template iterates on that variable = it must be a list or an iterable.
  - Each element of the iterable is expected to have a `name` and `phone` attribute.
  - In a Flask template:
    - if `customer` is a dictionary, then `customer.name` **works as if** you used `customer["name"]`.
    - if `customer` is an object, then `customer.name` will return the attribute `name` on the object `customer`.
    - we don't have classes (yet) for the customers.

    - This means you must write Python code that returns a list of dictionaries, each dictionary will have the keys `name` and `phone`.
- Create a new route for the customers list: `@app.route("/customers")`.
- In the Python function, open the `customers.csv` file (remember: it is located in the `data` folder!).
- Read from the file - it is probably wise to use `csv.DictReader`.
- Process the data to obtain a list of dictionaries.
- Render the template: `return render_template("customers.html", customers=my_data)`.
- Repeat with the products page.

Create links using `url_for`

Make changes to your template in order to have a menu on each page, containing three links:

- home page
- customer list
- product list

> ⚠️ Do not write links manually! Use `url_for`.

It would be tempting to hardcode the links in the template, using the URL `/products`, `/customers`, etc. However, if the route changes in Flask (in the `@app` decorator), all the related links will break. Instead, you **SHOULD USE** `url_for` which returns the URL for a specific view. `url_for` takes the **name of the function** as first parameter. If the view uses arguments, make sure you provide them in the keyword format (`parameter=value`).

```
<a href="{{ url_for('home') }}">Homepage</a>
<a href="{{ url_for('customer_list') }}">Customers</a>
```

This creates links to 1. the view defined with `def home()` and 2. the view defined with `def customer_list()`. The parameters are NOT the URLs.

You can also use `url_for` to load "static" files (for example, CSS or JS files):

```
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
```

This will create a link to the file `style.css` located in the `static` folder of your app. Make sure you create it first!

Polish your project

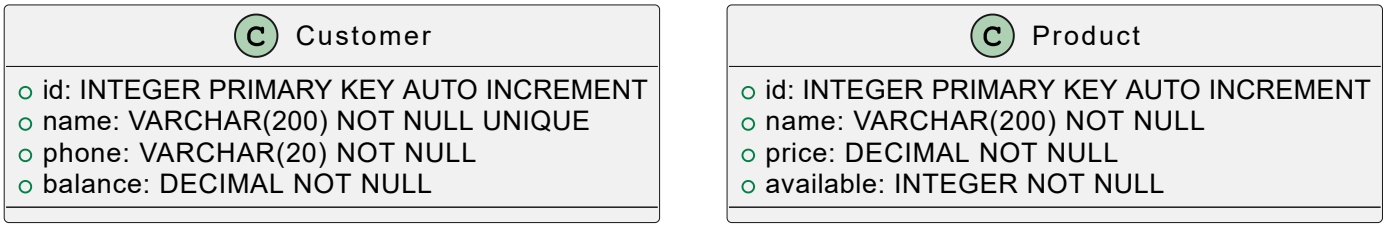
- Move the CSS code to a separate file in the `static` folder.
- Clean up the views to use templates throughout.
- Clean up the HTML code, CSS styles and Python files to make your website nice to look at.

Use a database with Flask

It is much easier and reliable to use a database rather than text files (JSON, CSV, etc). To keep things simple, we are going to use *SQLite*, which is a local database system that uses a single file. It is also natively supported in Python.

Instead of using SQL statements in our Python program, we are going to use a module that translates Python code into SQL statements, and vice versa. This is called an *ORM*, because it *maps* a *relational* database with Python *objects*.

For now, we are going to work with only two (unrelated) models: `Customer` and `Product`.



Any Python object from these classes can be saved to and retrieved from the database easily using an ORM.

Initial setup

- Install `Flask-SQLAlchemy` with `pip install flask-sqlalchemy`.
- Configure the database settings for Flask in `app.py`:

```
app = Flask(__name__)
# This will make Flask use a 'sqlite' database with the filename provided
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///i_copy_pasted_this.db"
# This will make Flask store the database file in the path provided
app.instance_path = Path("change_this").resolve()
# Adjust to your needs / liking. Most likely, you want to use "." for your instance
path. You may also use "data".
```

- Setup Flask-SQLAlchemy to use the ORM. Create a file `db.py` and use the following code:

```
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.orm import DeclarativeBase


class Base(DeclarativeBase):
    pass

db = SQLAlchemy(model_class=Base)
```

> 💡 This creates an object `db` which we can use for all tasks requiring the database: defining models, querying from and saving to the database. This object has the `Base` class registered as the base class for ORM-related features. We could make changes to this base class, but it is not required for this lab.

Make sure you connect your `db` object with your `app` object. In `app.py`:

```
db.init_app(app)
```

Create your models

We can now create the Python classes that will be stored in the database. Create a new file `models.py`:

```
from sqlalchemy import Boolean, Float, Numeric, ForeignKey, Integer, String
from sqlalchemy.orm import mapped_column, relationship

from db import db

class Customer(db.Model):
    id = mapped_column(Integer, primary_key=True)
    name = mapped_column(String(200), nullable=False, unique=True)
    phone = mapped_column(string(20), nullable=False)
    balance = mapped_column(Numeric, nullable=False, default=0)
```

This defines the entity `Customer`, with the relevant fields setup. `id` will be the primary key - SQLAlchemy will make it auto-increment by default.

⚠️ Make sure your class inherits from `db.Model`! Otherwise, it will not be "registered" by Flask-SQLAlchemy.

Following the same concept, create the `Product` class in the `models.py` file.

Create a script to seed the database

Before using the database, we must create it, and insert data into it.

- `db.create_all()` will create all tables (and the SQLite database if it does not exist)
- `db.drop_all()` will drop all tables

Once the database and tables are created, we can save objects to the database:

- we create instances of our classes
- we can then add them to the 'session'
- the session is automatically provided by the Flask-SQLAlchemy module, using the `db.session` variable.

- we commit changes to the database. You can add several objects to the session and commit once - it will commit all changes that have been added to the session.

For example:

```
for customer_name in ("Bob", "Tim"):
    # Create a new instance
    obj = Customer(name=customer_name)
    # Adds it to the session
    db.session.add(obj)
# Commit the session changes to the database (outside of the loop)
db.session.commit()
```

We are now going to create a dedicated Python script `manage.py` to help us manage our database operations. Flask-SQLAlchemy allows you to make database requests when the app is running through `python app.py`. However, `manage.py` is a different Python file and you will run it independently from the app. In order to make database requests, you will need to setup an "app context" first:

```
with app.app_context():
    obj = Customer(name="Tim")
    db.session.add(obj)
    db.session.commit()
```

Create the file `manage.py` and create separate functions for each of the following:

- drop all tables from the database
- create all tables in the database
- import data from the CSV files:
    - open the relevant CSV files and load their data
    - create `Customer` or `Product` instances
    - save them to the database

> 💡 You will need to import `db` from `db.py` and `app` from `app.py` !

Then, call the functions as required in the `if __name__ == "__main__"` block of your script. Run the script and check that your database now contains records.

Update your views to use the database objects

We now have records in our database. We can use them to display HTML content.

- You can create `SELECT` SQL statements with `db.select`.
- For example:
    - `statement = db.select(Customer).order_by(Customer.name)`

    - `statement = db.select(Customer).where(Customer.name == "Tim")`
- You can *execute* these statements in the current session with `records = db.session.execute(statement)`.
- `SELECT` statements return a list of "database records". Exploiting these raw records is a little bit complicated, so it is **strongly** recommended that you use `.scalars()` or `.scalar()` to obtain the actual results.
- `results = records.scalars()` (for a list of records, `.scalar()` for a single record)
- The results returned are instances of the `Customer` class.

Final result

This lab is complete when:

- you can run the app and navigate to the customers list and the products list using a browser.
- all data visible in HTML templates comes from the database.
- you can easily drop and recreate the database, as well as import data using the `manage.py` script.