Lab: Flask and APIs

In this assignment, we are going to expand on the previous lab and add an API to our website to make it easier to interact with other platforms and programs. We are also going to use a *database*: we will use SQLite, but we could use any database backend that is supported by the module we are going to use (SQLAlchemy).

Download the starter template

- Download the starter template from D2L, and unzip it.
- The code is set up in 3 different files:
  - `app.py`: the main Flask application file. This is where (almost) everything happens.
  - `database.py`: a Python module to connect with the database. We use a separate file to prevent circular import issues.
  - `models.py`: this Python module contains classes that will be mapped to tables in our database.

You will need to install the following modules in your virtual environment: `pip install flask-sqlalchemy`.

Understanding ORM (Object Relational Mapping)

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*.

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()
```

The `db` object can now be used to interact with our database.

The app configures the location of the database file (when using SQLite):

```
from flask import Flask
from pathlib import Path

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///store.db"
app.instance_path = Path(".").resolve()
db.init_app(app)
```

Defining models

You can define new tables in the database by creating classes. These classes must inherit from the `db.Model` class (see the `db` object created above). For example:

```python
class Product(db.Model):
    name = db.Column(db.String, unique=True, primary_key=True, nullable=False)
    price = db.Column(db.Float)
```

This will define a table `product` with two columns (or fields):

- `name` is a "string" (will be transformed into `VARCHAR` automatically by SQLAlchemy). It is the primary key, and cannot be null.
- `price` is a "float".

Creating the tables

Flask does **not** create the tables automatically. You need to create them by calling the `db.create_all()` method. You can also run the script `create_tables.py`. It will create the tables for all models declared in your app.

⚠ if you make changes to your models, `create_all()` will **not** update your database schema. You will need to drop the database and recreate it. When using SQLite, it is as simple as deleting the database file.

💡 You only need to create the tables **ONCE** after you made changes to your models.

Using the ORM: make queries

You can now easily make queries by using the model classes. For instance, to get all products in the database: `Product.query.all()`. This returns a list of `Product` instances. Each instance has the attributes as defined in the model. With the model provided:

```python
products = Product.query.all()
first = products[0]
print(first.name, first.price)
```

```python
products = Product.query.filter(Product.price < 5)
for product in products:
    print(product.name, product.price)
```

You can also retrieve a specific object from the database using its primary key (on the product model, `name` is the primary key and it is a string).

```
nutella = Product.query.get("Nutella")
print(nutella.price)
```

Using the ORM: saving objects to the database

You can also create instances of your models, and send them to the `db.session.add` method. Do not forget to use `db.session.commit` to commit the changes to the database. For example:

```
orange_juice = Product(name="Orange juice", price=4.99)
db.session.add(orange_juice)
db.session.commit()
```

Debug / interactive Python shell

⚠ If you want to debug your program / database manually, you can do so in the interactive shell. The database connection requires an *app context* to be setup. You can do so with `app.app_context().push()`.

From example, in an interactive Python shell (run with `python`):

```
>>> from app import app, db
>>> from models import Product
>>> app.app_context().push()
>>> Product.query.all()
[...]
```

💡 You can also use the `debug.py` script provided.

Make changes to the models, and insert data

Make changes to the `Product` model, to add a field `quantity`. This is the number of items of a given product available in the store. Regenerate the database / tables. You should now be able to insert a bunch of products using the `create_products.py` script.

You should also see the store inventory on the homepage of your Flask application. Inspect the `app.py` source code and make sure you understand what happens in the `home()` view.

Create the `Product` API endpoints

We are now going to create API endpoints to retrieve, add, modify, and delete products from the database. API endpoints will use the JSON format. It is easy to return JSON with Flask - you can either use `jsonify`, or return a dictionary in your function view. Remember that only Python standard types can be transformed into JSON. If you want to return a Product instance or a list of products, you will need to "transform" these objects back into Python dictionaries and lists. This is called *serialization*.

Study the `api_get_product` function to make it work

This view returns the JSON representation of a given product in the database.

- we query the database to obtain a Product instance
- we call the `.to_dict()` method on this instance to transform it to a dictionary that can be converted to JSON
- we then return the JSON from the view

Add the `.to_dict()` method on the Product class. It should simply return all attributes in a dictionary. Once this is done, you can check the view at the URL: http://localhost:5000/api/product/<product_name>.

Study the `api_create_product` function

This function is called by the URL `/api/product` using the `POST` method. It expects to receive a JSON payload as part of the request. The JSON payload should be a dictionary, containing the keys `name`, `price` and `quantity`.

Test your endpoint with Postman

- create a new request in Postman for `http://localhost:5000/api/product`
- select the `POST` method
- switch to the `Body` tab, and select `raw` format
- in the dropdown to the right, select `JSON`
- type the JSON in the field, and submit.

For example: `{"name": "test", "price": 2.99, quantity: 1000}`.

Test your endpoint with HTTPie

It is very easy to make HTTP requests on the command line using `httpie`. First, install it with pip: `pip install httpie`. In the terminal (not a Python shell!), the following command will make a POST request similar to the one above: `http POST http://localhost:5000/api/product name="test" price=2.99 quantity=1000`

Add the other endpoints

- Create the `/api/product/<string:name>` endpoint, with the HTTP method `DELETE`.
  - This endpoint deletes the product with name `name`.
- Create the `/api/product/<string:name>` endpoint, with the HTTP method `PUT`.
  - This endpoint receives a JSON payload (a dictionary with price and quantity).
  - It updates the store inventory based on the values received by the endpoint.
  - Make sure the price is a valid value (float) as well as quantity (integer).
  - If they are not valid, you should return an error message with HTTP status code 400.

💡 The two endpoints respond to the same URL, but with different *HTTP methods*. It is better to have separate functions for each HTTP method.

Create the `Order` model

In the `models.py` file, create a new model `Order`. This model has the following fields:

- `id`: a unique number identifying the order. SQL Alchemy will automatically transform the first `db.Integer` field into an auto incrementing index.
- `name`: the name of the customer making the order (string)
- `address`: the address of the customer making the order (string)
- `completed`: a boolean field - default value should be `False`. If `True`, it means the order has been processed.

Deal with relationships

- Each order also contains a *list* of products, with a quantity (for example: `10 apples`).
- We cannot represent this using a flat table.
- We need to add an "association table" (which is going to be an association object in Python).

This is done by creating an intermediary object `ProductsOrder`. For example:

```
class Order(db.Model):
    # ... other fields
    # add a many-through-many relationship using an intermediary model
    products = db.relationship('ProductsOrder', back_populates='order')

class ProductsOrder(db.Model):
    # Product foreign key is name
    product_name = db.Column(db.ForeignKey("product.name"), primary_key=True)
    # Order foreign key is ID
    order_id = db.Column(db.ForeignKey("order.id"), primary_key=True)
    # This is how many items we want
    quantity = db.Column(db.Integer, nullable=False)

    # Relationships and backreferences for SQL Alchemy
    product = db.relationship('Product')
    order = db.relationship('Order', back_populates='products')
```

This allows to have, for an order with `10 apple` and `1 cheese`:

- Order with ID 1
- ProductsOrder with product `apple`, order `1`, and quantity `10`
- ProductsOrder with product `cheese`, order `1`, and quantity `1`

The easiest way to add products to an order is to deal with the `ProductsOrder` objects directly. Inspect the code in `create_order.py` to see how it can be done.

Create the endpoint `api_get_order` at `/api/order/<int:order_id>`

- This function receives the `order_id` as argument.
- Load the order from the database, and return a JSON dictionary with the following data:
  - `customer_name`: customer name
  - `customer_address`: customer address
  - `completed`
  - `products`: a list of dictionaries, with each dictionary having
    - `name`: product name
    - `quantity`: amount of the product `name` in the order
  - `price`: the total expected price for the order

For example:

```
{
  "customer_name": "Tim",
  "customer_address": "555 Seymour Street, Vancouver",
  "products": [
    {
      "name": "apple",
      "quantity": 10,
    },
    {
      "name": "cheese",
      "quantity": 1,
    },
  ],
  "price": 23.89
}
```

> The easiest way is to create a `to_dict()` method on the `Order` class that computes / builds the dictionary above! You can make queries to the database from anywhere in your Flask code.

Create the endpoint `api_create_order` at `/api/order` (`POST`)

- This function only works with the HTTP POST method.
- It does not receive arguments, but has a JSON payload.
- The JSON payload has the same format as the output of `api_get_order`.
- The view must create the order instance according to the data provided in the JSON.
- The view returns the JSON version of the order (see above).
- If the order contains a product that *does not exist in the inventory*, return an error message with HTTP status code 400. **The app should accept orders with products that exist in the inventory, even if the**

quantity available is 0.

Create the endpoint `api_process_order` at `/api/product/<int:order_id>`

- This endpoint receives the argument `order_id`.
- It must only work for the HTTP method `PUT`.
- It receives a JSON payload: a dictionary.
- If the JSON payload is missing or invalid, return an error message with HTTP status code 400.
- If the dictionary contains the key `process` and its value is `True`, process the order:
  - if the order was already processed (`completed` is `True`), nothing needs to be done
  - otherwise, see below
  - the view returns the order in JSON format (see `api_get_order`)

Processing the order

To process the order:

- make sure there are enough products in the store to fulfill the order.
- if the order contains more products than available, adjust the order accordingly.
- Example:
  - a customer wants to buy `10 milk` in their order, but there is only 5 milk available
  - the order should be adjusted to `5 milk`

Adjust the store inventory as required to reflect the products that were bought from the order. Then, set the `completed` attribute of the order to `True`.

> It would be a good idea to create a `process()` method on your class that runs the required operations.

Submission and grading

- You must demo your app to get a grade (no demo = 0).
- The demo must have the following steps:
  - delete the database file
  - create the tables with `create_tables.py`
  - create products with `create_products.py`
- And (each step is worth 1 mark, 12 marks total):
  - **create** a new product with the JSON endpoint `/api/product`
  - **update** an existing product with the JSON endpoint `/api/product/<product_of_your_choice>`
    - it cannot be the same product as the one you created before
  - **view** the updated product with the JSON endpoint `/api/product/<updated_product>`
  - **delete** an existing product with the JSON endpoint `/api/product/<product_of_your_choice>`
    - it must be a different product from the ones you created / updated before
  - **view** the deleted product with the JSON endpoint `/api/product/<deleted_product>`
    - it must return a page with status code 404

- load the homepage of the app in your browser and confirm that the store inventory has been updated
- **create** a new order with the JSON endpoint `/api/order`
- **view** the created order with the JSON endpoint `/api/order/<created_order_id>`
- **create** a new order with the JSON endpoint `/api/order`
    - this order must contain at least one product that does not exist in the store inventory
    - the view must return a page with status code 400
    - the order must not exist in the database
- **create** a new order with the JSON endpoint `/api/order`
    - this order must contain products with an invalid quantity, such as: floating point numbers, strings, negative integers, etc.
    - the view must return a page with status code 400
    - the order must not exist in the database
- **create** a new order with the JSON endpoint `/api/order`
    - this order must contain at least one product in a quantity that exceeds the store inventory
    - for instance, make an order with `1000 apple` and `1000 cheese`
    - the view must return the order in JSON format
    - the order must exist in the database, with the quantities provided in the request
- **process** the first order created using the JSON endpoint `/api/order/<first_created_order_id>`
    - the view must return the order in JSON format
    - load the store inventory page in the browser, and confirm it was updated accordingly
- **process** the second order created using the JSON endpoint `/api/order/<second_created_order_id>`
    - the view must return the order in JSON format
    - the order must have been updated in the database (according to the store inventory)
    - load the store inventory page in the browser, and confirm it was updated as required

Resources and hints

- To debug and demo your app, use the Python interactive shell.
- I created the `debug.py` script that provides you with an environment where you can access your models and run database requests: `python -i debug.py`.
- You may also want to use a program to "view" your SQLite database. DB Browser for SQLite is free and works well. Make sure you **close** the database if you try to move / delete the database file.
- For the demo, you must have your JSON requests already prepared. You can:
    - use httpie (`pip install httpie`) commands in the terminal, and save them in a text file to run them again later
    - use Postman and save your requests to run them again later
    - use Python `requests` (`pip install requests`) to make a Python script that runs all the required requests.
        - `r = requests.post(url, json={"some": "data"})` will make a `POST` request to `url`
        - `r.json()` gives you the Python object decoded from the JSON response

    - **do not mix up** `request.json` (to access JSON data from the request in Flask) with `r.json()` (to decode JSON data from a request)!
- You will need to have a terminal active and running the Flask app all the time to debug it.
- There are lots of different ways to do the same thing with SQL Alchemy (for instance, using `db.session` instead of `Product.query`). Make sure you use the syntax that you are the most comfortable with, and remain consistent throughout the code.

💡 Get bonus marks on the midterm exam

You can do extra work on this assignment to get bonus marks on your midterm exam grade. *Please discuss it with me first*. You must use Python and do the work in Flask, not in the "frontend" using Javascript.

Additional tasks include, but are not limited to:

- add filters on the HTML interface to sort the inventory by product name, price, or quantity
- add HTML pages and routes to view:
    - a list of pending orders
    - a list of completed orders
    - any order, pending or completed
- add JSON endpoints to extract orders for a customer with a given name (for example, all orders for customer name "Tim")
- add HTML pages (with forms) to search through orders by name
- add HTML pages (with forms) to update the store inventory
- add date/time attributes on the order (for instance: when the order was received, when the order was processed)
- add filters on the HTML interface and JSON endpoints to sort / filter orders by name or dates
- create multiple stores with a distinct inventory