Using Flask to create API routes

Rather than creating web forms to interact with our platform, we are going to use API endpoints.

An API endpoint is like a web page, but designed to be used by scripts and applications rather than humans. Data received and sent by the API uses a serialization format. It is very common to use JSON for that case.

Create a route that outputs JSON

Instead of returning HTML through a template and the `render_template` function, Flask can also return a full JSON response. You just need to return a *JSON-serializable* value in your function (typically: a list or dictionary). To make it explicit that the view returns JSON, you can also return a call to the `jsonify` function from Flask.

Create a new route `/api/customers` that returns a JSON list of all the customers with all their attributes. You could use something like:

```
@app.route("/api/customers")
def customers_json():
    statement = db.select(Customer).order_by(Customer.name)
    results = db.session.execute(statement)
    customers = [] # output variable
    for customer in results.scalars():
        json_record = {
            "id": customer.id,
            "name": customer.name,
            "phone": customer.phone,
            "balance": customer.balance,
        }
        customers.append(json_record)

    return jsonify(customers)
```

Create a route that takes in a URL parameter

When working with APIs, it is common to implement a `CRUD` platform. It allows:

- `C`reate
- `R`ead
- `U`pdate
- `D`elete

When updating or deleting entities, we need to refer to the specific instance that we want to work with. It is common to pass that reference in the URL of the API call. For example, `/api/customers/16` represents the customer with ID 16 in the database. In Flask, you can set a route parameter like this:

```
@app.route("/api/customers/<int:customer_id>")
def customer_detail_json(customer_id):
    statement = db.select(Customer).where(Customer.id == customer_id)
    result = db.session.execute(statement)
    # ...
```

Complete the code above to create a route that displays the JSON version of the customer instance with the given ID.

The following URL converters are available and commonly used:

- `<string:variable>`: strings without any `/` (default)
- `<int:variable>` and `<float:variable>` for numbers
- `<path:variable>` for strings including `/`

Create a method on your models for JSON conversion

You will often need to convert an object instance to JSON. Python cannot natively convert an instance from a custom class to JSON. On your class models, create a new method that returns a dictionary with all the attributes retrieved from the database. For example, for the customer class:

```
class Customer(db.Model):
    # [...]

    def to_json(self):
        return {
            "id": self.id,
            "name": self.name,
            "phone": self.phone,
            "balance": self.balance,
        }
```

Update your code to use this method in your Flask views.

Create a route with a different HTTP method

Each action in a `CRUD` API is typically associated with a HTTP method:

Method	Action
`GET`	read
`PUT` / `POST`	create and update
`DELETE`	delete

In Flask, all routes default to the `GET` method. You can specify which method(s) the route supports in the `@app.route` decorator. For example, to delete a customer with the `DELETE` HTTP method:

```
@app.route("/api/customers/<int:customer_id>", methods=["DELETE"])
def customer_delete(customer_id):
    customer = db.session.execute(db.select(Customer).where(Customer.id ==
customer_id))
    db.session.delete(customer)
    db.session.commit()
    return "deleted"
```

💡 Note that we used `db.session.delete` to delete the customer instance from the database!

Create more views for `Customer` and `Product`

So far, we have only managed "empty" HTTP requests - in the sense that we only used information available in the URL or the HTTP method. However, for CRUD operations, the client needs to send a lot of information to the server (for instance, product name, product price, etc). It would be inconvenient (and/or insecure) to encode the data in the URL. Instead, we are going to use the *body* of the HTTP request. It can contain any content (text / string), but it is usually **encoded in JSON**.

In Flask, you can access the *decoded JSON data from the request body* by using `request.json` (make sure you import `request` from Flask first). For example:

```
from flask import Flask, render_template, request

@app.route("/example", methods=["POST"])
def example():
    print(request.json)
```

Create the following views:

URL	Method	Action	Status code
/api/customers	POST	receives JSON with `name` and `phone` attribute, creates customer	201 if successful, 400 if request invalid
/api/customers/1234	PUT	receives JSON with `balance`, updates customer	204 if successful, 400 if request invalid, 404 if customer ID not found
/api/customers/1234	DELETE	deletes customer	204 if successful, 404 if customer ID not found

URL	Method	Action	Status code
/api/products	POST	receives JSON with `name` and `price`, creates product	201 if successful, 400 if request invalid
/api/products/1234	PUT	receives JSON with `name`, and/or `price` and/or `quantity` - updates product	204 if successful, 400 if request invalid, 404 if customer ID not found
/api/products/1234	DELETE	deletes product	204 if successful, 404 if customer ID not found

Make sure your views check parameters sent in the JSON (and the URL). You may want to use `db.get_or_404` to make the code easier to read.

Remember:

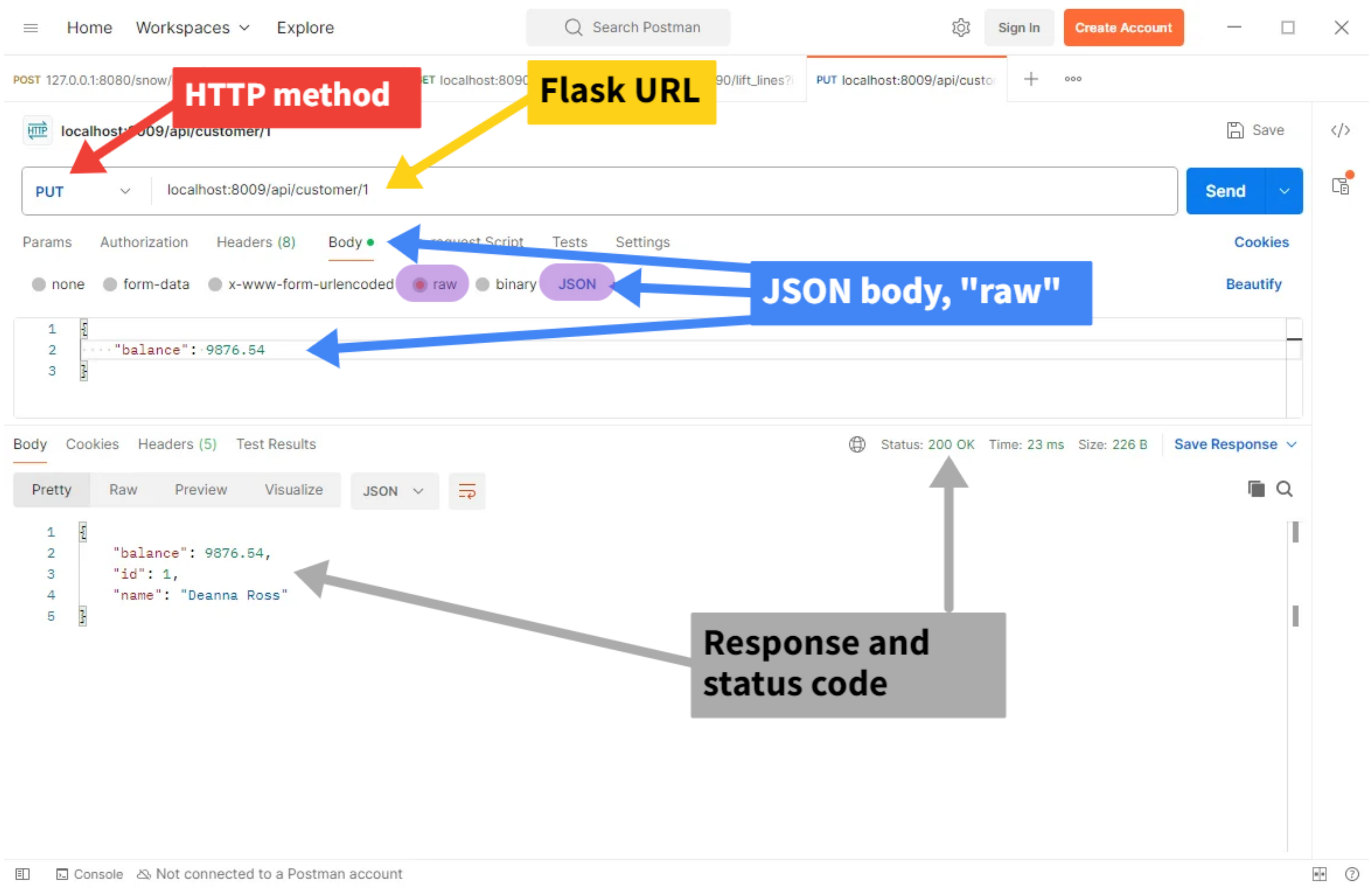
- name cannot be empty (product or customer)
- price must be a positive float
- quantity must be a positive integer

Invalid requests should return HTTP code 400 (`Invalid Request`). You can return a specific HTTP error code as the second value of the return tuple. See an example below:

```
@app.route("/example/<int:customer_id>", methods=["POST"])
def update_customer(customer_id):
    data = request.json
    customer = db.get_or_404(Customer, customer_id)

    if "balance" not in data:
        return "Invalid request", 400

    balance = data["balance"]
    if not isinstance(value, [int, float]):
        return "Invalid request: balance", 400

    customer.balance = value

    db.session.commit()
    return "", 204
```

Test your views with Postman

Run the Flask application, and make requests to the API endpoints using Postman. Make sure the website interface changes accordingly (you will need to refresh the pages).



```
{
    "balance": 9876.54
}
```

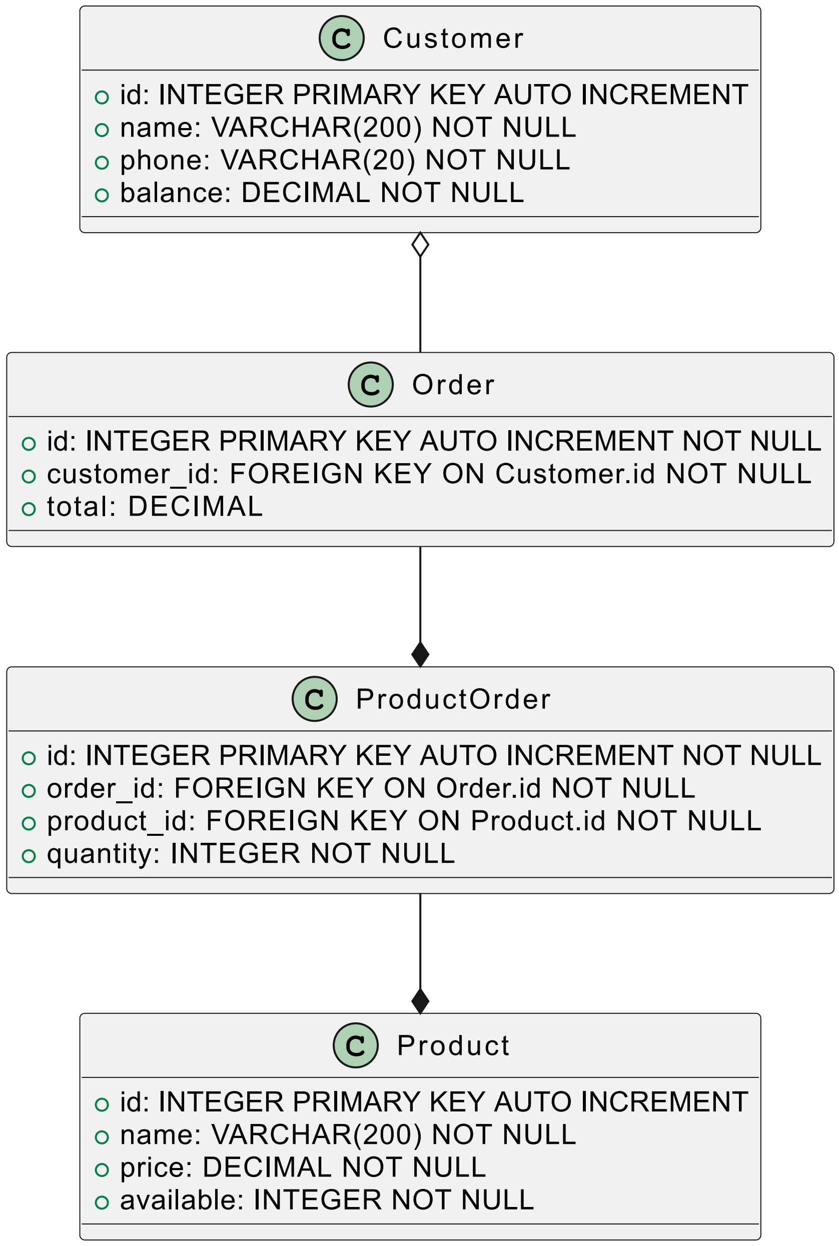
```
{
    "balance": 9876.54,
    "id": 1,
    "name": "Deanna Ross"
}
```

Add new models to manage orders

We are now going to add more models to our application to support orders. The relationships are:

- a customer has many orders (one-to-many)
- an order has many products, and a product can belong to many orders (many-to-many)

There is no easy way to build a many-to-many relationship in a relational database. Instead, we build an association table that has two one-to-many relationships.


C Customer
o id: INTEGER PRIMARY KEY AUTO INCREMENT
o name: VARCHAR(200) NOT NULL
o phone: VARCHAR(20) NOT NULL
o balance: DECIMAL NOT NULL
C Order
o id: INTEGER PRIMARY KEY AUTO INCREMENT NOT NULL
o customer_id: FOREIGN KEY ON Customer.id NOT NULL
o total: DECIMAL
C ProductOrder
o id: INTEGER PRIMARY KEY AUTO INCREMENT NOT NULL
o order_id: FOREIGN KEY ON Order.id NOT NULL
o product_id: FOREIGN KEY ON Product.id NOT NULL
o quantity: INTEGER NOT NULL
C Product
o id: INTEGER PRIMARY KEY AUTO INCREMENT
o name: VARCHAR(200) NOT NULL
o price: DECIMAL NOT NULL
o available: INTEGER NOT NULL

With SQLAlchemy, you setup relationships by:

- defining a field as a foreign key (the actual field in the database)
- defining another field as a `relationship` (the "ORM" field that we use in Python)

See for example:

```
class Customer(db.Model):
    # ...
    orders = relationship("Order")


class Order(db.Model):
    # ...
    customer_id = mapped_column(Integer, ForeignKey(Customer.id), nullable=False)
    customer = relationship("Customer", back_populates="orders")
```

`Customer` and `Order` are now linked with a relationship:

- if `my_order` is an instance of the `Order` class, then `my_order.customer` returns the customer instance of the relationship
- if `tim` is an instance of `Customer`, `tim.orders` returns a list of `Order` instances owned by the customer (`back_populates`)

On your `ProductOrder` class, you will need:

- `order_id` and `product_id`
- `product`: relationship to `Product` class
- `order`: relationship to `Order` class
- `quantity`: how much of that product has been ordered

You will also need a relationship on `Order` (the list of association instances for that order). For example, `items = relationship("ProductOrder")`. Make sure this field is back-populated from the `order` relationship in `ProductOrder` !...

> If you have done everything right, you should be able to follow all the relationships in Python:
>
> - `customer.orders` = list of orders for `customer`
> - `order.customer` = customer instance for `order`
> - `order.items` = list of **ProductOrder** instances for `order`
> - `order.items[0].product` = first product instance in the `order`
> - `order.items[0].product.name` = name of the first product instance in the `order`
> - `order.items[0].quantity` = quantity of the first product in the `order`

Recreate tables to fit the new models and seed the database

Create / update the models as required. Make sure you use default values for some of the fields. The `total` of an order is not known (`NULL` in SQL, `None` in Python) until the order is actually processed by the store. The field must be nullable!

Your models have changed - so your database tables must be updated too. Use the scripts you created earlier: drop the tables, create them again, and import data in order to have products and customers available.

Create random orders for debugging / testing

In your `manage.py` script, create a new function to randomize orders. You can use the following logic:

- randomly select one customer
- randomly select one product
- randomly select the quantity of the product in the order

- create a `ProductOrder` instance
- repeat for X orders

💡 To randomly select one record from the database, **do not select all records and use Python's `random`**! This is very inefficient - it is much better to use the `RANDOM` database function, and limit the results to 1 record (`LIMIT 0, 1`). SQLAlchemy abstracts SQL functions to Python functions in the `sqlalchemy.sql.functions` module (see the documentation). For example:

```
from sqlalchemy.sql import functions as func

# ...
def something():
    statement = db.select(Customer).order_by(func.random()).limit(1)
```

Creating orders with products

`Product` and `Order` are linked through the `ProductOrder` association model. To manage products in an order, you will now manage these entities instead. If your association field is called `items`, then you can create an order with products like this:

```
# Find a random customer
cust_stmt = db.select(Customer).order_by(func.random()).limit(1)
customer = db.session.execute(cust_stmt).scalar()

# Make an order
order = Order(customer=customer)
db.session.add(order)

# Find a random product
prod_stmt = db.select(Product).order_by(func.random()).limit(1)
product = db.session.execute(prod_stmt).scalar()
rand_qty = random.randint(10, 20)

# Add that product to the order
association_1 = ProductOrder(order=order, product=product, quantity=rand_qty)
db.session.add(association_1)

# Do it again
prod_stmt = db.select(Product).order_by(func.random()).limit(1)
product = db.session.execute(prod_stmt).scalar()
rand_qty = random.randint(10, 20)
association_2 = ProductOrder(order=order, product=product, quantity=rand_qty)
db.session.add(association_2)

# Commit to the database
db.session.commit()
```

Create / update views for orders and customers

Create and update your Python code and HTML templates in order to have the following views available on the website (HTML):

View	Route	Details
Homepage	`/`	With links to products list, customers list, orders list (navigation)
List of customers	`/customers`	Showing name, phone and balance. With links to customer detail pages.
Customer detail	`/customer/CUSTOMER_ID`	With links to all orders associated with the customer.
List of products	`/products`	Showing name, price, quantity available.
List of orders	`/orders`	With links to all individual order pages.
Order detail	`/order/ORDER_ID`	With a link to the customer page. Lists all products in the order, with their price, quantity ordered, and quantity available. Your view must also calculate the *estimated total* for the order, and display it.

Make sure these pages are easy to access (update the menu as required).

Estimated total

The Flask templating language (Jinja) is not made for logic and calculations. Only perform basic computation in the templates - the logic should belong in the models or in your backend code. Rather than making complex operations in the template, you can create a *method* on the `Order` class that computes the estimated total for the order. Your Flask template can then simply call this method from the `Order` *instance* in the template. Don't forget to `round` the result!

Create an API view to make an order

Create the route `/api/orders` with HTTP method `POST`. This route expects JSON data in the request body, containing the customer ID and all the items that are in the order.

For example, the following JSON is an order for customer with ID `123`, with 2 onions, 1 lemon and 1 chicken breast.

```
{
    "customer_id": 123,
    "items": [
        {"name": "onions", "quantity": 2},
        {"name": "lemon", "quantity": 1},
        {"name": "chicken breast", "quantity": 1},
    ]
}
```

Make sure your code validates the JSON data!

- the customer ID must be present and be a valid customer, otherwise return HTTP status 404
- the `items` must be present and a list with at least 1 item
- each item must have:
    - `name`: a product name - if the product does not exist in the store, ignore it
    - `quantity`: the quantity ordered
    - Note: a customer may order more of a product than is available in stock!

Test your view with Postman and your web interface.

Create views to be used with HTML forms

> Please note this lab is mostly about web API development, and not HTML / front-end development.

For quicker debugging, it could also be useful to have buttons to perform actions directly on the website. We are going to create a button to delete an order from the web interface. We could create an endpoint that uses the `DELETE` HTTP method - unfortunately, pure HTML can only perform `GET` and `POST` requests in forms. We will work around this limitation by creating a `POST` route for this action.

Create the `/orders/ORDER_ID/delete` route, with HTTP method `POST`. Obtain the associated order instance from the database (`order = db.get_or_404(Order, order_id)` would work well here). You can then delete the object through the session: `db.session.delete(order)`. Don't forget to commit!

Once the instance is deleted, there is nothing else to do - we should redirect the user to the list of orders. You can do this by returning `redirect` with the URL required. You can even combine it with `url_for`!

```
from flask import [......], redirect, url_for

def order_delete(order_id):
    # [...]
    db.session.delete(order)
    return redirect(url_for("orders"))
```

Update your HTML template to call the view

In the template that displays the details of an order, add a form with a submit button. Make sure the form uses the method `POST` and the correct URL (see above).

```
<form method="POST" action="{{ url_for('order_delete', order_id=order.id) }}">
    <button type="submit">Delete</button>
</form>
```

Clicking the form will now trigger the `action` URL (= the view you created before). The order will be deleted, and the browser will be redirected to the list of orders.

Final result

This lab is complete when:

- you can run the app and navigate to the list of required web pages
- you can use the API to perform C(R)UD operations on customers and products
- you can use the API to create a new order
- API endpoints perform parameter and URL validation, and return HTTP status 400 (invalid request) or 404 (object not found)
- you can use the `manage.py` script to completely reset your database, and end up with generated products, customers, and orders
- the web interface has a (functional) button that allows to delete an existing order