Order management and data processing

Now that we have customers, products, and orders, we can process orders and start keeping track of our sales.

Update your models

Update the `Order` class and add two fields:

- `created`: must be a `DateTime` field. This is when the order was "received" by the store.
  - you can use the SQL functions from SQLAlchemy (`func.now()`) or use Python `datetime`.
- `processed`: a `DateTime` field. This is when the order was "processed" by the store.
  - by default, an order is not processed (= the field will be `NULL`). Make sure it is nullable in your model!

Using the `manage.py` script, recreate your tables and generate random orders.

Add a process method on your `Order` class

Our store will have the following logic when managing orders:

- do not process an order that was already processed
- the current customer balance must be > 0
- if a customer ordered more of a product than is available in the store, there are three strategies:
  - `adjust` (default): the order is adjusted to match the quantity available in the store
  - `reject`: the order is not processed
  - `ignore`: the product is ignored (its quantity is set to 0)
- for each product in the order:
  - substract the quantity ordered from the quantity available in the store
  - compute the price (quantity x product price)
- calculate the total price for the order
- substract the order price from the customer balance
- set the `processed` field to the current date / time

Implement this logic in the `process` method:

- your method should return `True` when processing was successful
- it may return other information if processing was unsuccessful (for example: the reason why)
- it should take a `strategy` parameter that can be `adjust` (the default), `reject` or `ignore`

Implement the API route to process an order

Create the `/api/orders/ORDER_ID` route. It accepts the `PUT` HTTP method, with a JSON payload. The JSON payload is a dictionary, and must contain the `process` key. The order is processed if `process` is `true`. The JSON payload may also contain the `strategy` key. It can be one of `adjust`, `reject` or `ignore`. If not provided, the default is `adjust`. If an invalid value is provided, return an HTTP error code 400.

Examples:

Process an order:

```
{
    "process": true
}
```

Process an order, ignore products over the store quantity:

```
{
    "process": true,
    "strategy": "ignore"
}
```

Do not process an order (!?):

```
{
    "process": "whatever"
}
```

Improve the web interface

Update the templates

Make sure your views display the `created` and `processed` fields.

You may want to do conditional formatting depending on the `processed` field. You can do the following in your template:

```
{% for order in orders %}
    {% set class_name = "pending" %}
    {% if order.processed %}
        {% set class_name = "processed" %}
    {% endif %}
    <p class="{{ class_name }}">{{ order.id }} {{ order.customer.name }}</p>
{% endfor %}
```

Update the delete method for an order

Make sure that if an order was already processed, it cannot be deleted from the web interface (button).

⚠️ Do not only hide the button from the interface! There are malicious people that could still trigger a request to your Flask route without using the form (for example using Postman).

Add a "Process" button in the web interface

On each order page, add a "process" button that allows to process an order from the web interface.

Test your API and your web interface

- Process an order with the button.
- Make sure the customer balance is adjusted.
- Make sure the store quantity is adjusted.
- Process another order with an API request using Postman. Experiment with different strategies.

Improve the code with blueprints

By now, your `app.py` is very crowded - there are lots of functions, and they are all mixed up (API, HTML, customers, products, orders, ...). Because we need the `app` object to declare a route, we cannot just split the code into separate files (this would lead to circular imports).

We are going to reorganize the code by using "blueprints". A Flask blueprint is like a mini Flask object that we can we add routes to. We can then add complete blueprints to an existing app. This solves the circular dependency problem, and also allows to organize your code in a clean way.

Create the "API customers" blueprint

- Create a new folder `routes`. This is going to be our Python package that contains all the blueprints for our application.
- Create a new file `api_customers.py` in that folder.
- Create a blueprint, and move your routes from `app.py` to this file.

For example:

```
from flask import Blueprint, jsonify, request

from db import db
from models import Customer

# Creates a Blueprint object (similar to Flask). Make sure you give it a name!
api_customers_bp = Blueprint("api_customers", __name__)

@api_customers_bp.route("/", methods=["GET"])
def api_customer_list():
    stmt = db.select(Customer).order_by(Customer.name)
    results = db.session.execute(stmt).scalars()
    return jsonify([cust.to_json() for cust in results])
```

In your `app.py`, you can now import this blueprint and add it to your existing app:

```
from routes.api_customers import api_customers_bp
app.register_blueprint(api_customers_bp, url_prefix="/api/customers")
```

⚠️ Note the `url_prefix` parameter! This will add a prefix to all the blueprint's URLs. So `/` will match `/api/customers/`! You can also leave the `url_prefix` empty (the default is `/`), but it is not recommended.

Repeat the operation for all API endpoints for products and orders.

Improve your blueprints

In your `routes` folder, create the `__init__.py` file. This will allow you to import blueprints in an easier way. For example:

```
from .api_customers_bp import api_customers_bp
from .api_products_bp import bp as api_products_bp
```

Then in your `app.py`: `from routes import api_products_bp`.

Move all your HTML endpoints to another blueprint. You will now have to update all your `url_for` references, because the functions are now defined in the blueprint, and not in `app.py`.

```
example = Blueprint("html", __name__)

@example.route("/")
def homepage():
    pass
```

You can access the URL for this function with `url_for` by using: `url_for("html.homepage")` (`<BLUEPRINT NAME>.<FUNCTION NAME>`).

Prepare the project and the live demo

To receive your marks for the project, you must:

- submit all your code to the Learning Hub, AND
- demo your project in class (last week before the final exams)

During the demo, you must show that the application works as expected. You have **5 minutes** for your demo. If one of the required steps (see below) fails, the demo stops. You cannot fix your problems (if any) live during the demo. You can demo as many times as you want, and even before the deadline, but the demo must happen during class. Demos during office hours on the last week should be **THE EXCEPTION**, and will only be accepted for reasonable circumstances.

You must use (and make changes to) the `demo.py` script during the demonstration.

- This script expects the Flask application to be up and running, and *makes HTTP requests* to your API/app.
- Make sure you adjust the `FLASK_URL` variable as required.

Demo steps

1. Stop the application. Delete the database.
2. Create the database, create the tables, run the application.
3. Seed the database with data. It must contain:
    1. products
    2. customers
    3. several non-empty orders
4. Demonstrate your HTML interface:
    1. list products
    2. list customers
    3. list orders
    4. access a specific order
    5. delete an order using the button
5. Run the `demo.py` script. The scrips must make HTTP requests and:
    1. create at least 3 new products (choose products and prices, see below)
    2. make at least 3 orders, including:
        1. at least 2 orders that cannot be fulfilled (= more ordered than what the store has in stock): *NOK orders*
        2. at least 1 order that can be processed: *OK order*
    3. demonstrate the products and orders exist (HTML interface)

> For all the following steps, you can either use Postman (or the HTTP client of your choice), or keep working with `demo.py`.

6. Process the *OK order*:

    1. using the HTML interface, take note of the customer balance and the quantity of products in the store
    2. process the order (no strategy required)
    3. demonstrate the customer balance was properly updated, as well as the products' quantities
7. Process the first *NOK order*:
    1. process the order with strategy `reject`
    2. confirm the order was not processed, and quantities / balance not updated
    3. process the order with strategy `ignore`
    4. confirm the order was processed, and the relevant product(s) ignored
8. Process the second *NOK order*:
    1. process the order with the default strategy
    2. confirm the order was processed, and the relevant product(s) adjusted
9. Demonstrate your API does parameter validation:
    1. create an order with a non-existing product
    2. create an order with an invalid value
    3. create a product with an invalid price
10. Demonstrate anything you are proud of / have been working hard on.