Backend Engineering Take-Home Assignment

Task: Build API that handles fragrance formula submissions

Definitions:
1. **Idempotency**: Applying an operation once or multiple times will give the same intended result every time
2. **Atomicity**: a series of operations treated as a single indivisible unit. Either all succeed or none succeed. Eg all or nothing. In this case, both storing the formula and publishing to queue.
3. **Message queue integration**: A system where messages are *delivered*, *acknowledged*, and *re-delivered* until processed successfully. Enables async communication by storing messages until they can be processed. Assignment states “in-memory message queue”
4. **Formula uniqueness**: defined by its material make-up. Formulas with the same name but different formulas are permitted.

API Endpoint Data Definitions:

A Fragrance Formula is
- (String) name
- (tuple<Material>) materials

A Material is
- (String) name
- (Decimal) concentration

```
{
  "name": "Summer Breeze",
  "materials": [
    {"name": "Bergamot Oil", "concentration": 15.5},
    {"name": "Lavender Absolute", "concentration": 10.0},
    {"name": "Sandalwood", "concentration": 5.2}
  ]
}
```

Unit Test Checklist - a rough list of questions to map to unit tests/to ask before submitting:
1. Is the system idempotent?
    1. Does the system *detect* duplicate formula submissions?
    2. Does the system *handle* duplicate formula submissions?
    3. When the same formula is submitted multiple times, is it *persisted only once*? (should only be persisted once)
    4. When the same formula is submitted multiple times, is a single message published, or many? (should be a single message)
2. Is the system atomic?
    1. Check all error message cases

TODO

1. ~~Create Repo, README outline, set up local space.~~
    1. ~~Python version: 3.11.7 - decided not to upgrade~~
2. ~~Plan~~
    1. ~~Pick framework —> Flask~~
3. ~~Define common Data Definitions~~
    1. ~~Material~~
    2. ~~Fragrance Formula (FF)~~
4. Define Init Actions
    1. ~~Accept FF~~
    2. ~~Validate FF~~
    3. ~~Deserialize FF~~

~~pause - write unit tests for the above 3~~

~~5. db/persistence: Check if Dupe - define and make sure unique~~
~~6. process - eg implement a message queue and place FF in message queue~~

1. ~~Enqueue~~:
    1. ~~assign a unique ID to each message~~
    2. ~~acquire lock -> append message -> release lock~~
    3. ~~return success message with ID~~
2. ~~Dequeue:~~
    1. ~~acquire lock -> pop message from left -> release lock~~

~~a message queue needs:~~

- ~~in-process/visibility tracking - so that a consumer can inform that they have started to process an item~~
- ~~acknowledgement (ACK) system - each message can be ack'd after it is "processed"~~
- ~~retries/visibility timeout - messages not ack'd after a while become high priority~~
- ~~thread safety - use a lock~~
- ~~idempotency - relies on the database for idempotency~~

~~5. Implement Rollback Strategy as part of atomicity~~

~~5. Return to README/Notes to articulate the duplicate detection strategy~~

7. Review Checklist

Resources/Reference
400 Bad Request - bad JSON, missing required parameter, wrong type, malformed data
401 Unauthorized - missing/invalid/expired token
403 Forbidden - user authenticated but not allowed
404 Not Found - wrong url path
405 Method Not Allowed - eg a POST to /users/123 when only GET is supported there
429 Too Many Requests - rate limiting/ throttling
409 Conflict - Duplicate Resource, queue full
500 Internal Server Error - not the client's fault eg database crash, unhandled Python exception

200 - OK
201 - Created (POST resource creation)
202 - Accepted - POSTed and queued
204 - No Content - successful delete/update with no body

https://docs.python.org/3.11/library/index.html

While considering which framework:

Searching project examples for the 3 frameworks in question: https://github.com/search?q=FastAPI&type=repositories

https://github.com/search?q=Flask%20API%20Example&type=repositories

https://www.reddit.com/r/Python/comments/xs7s6a/if_you_could_choose_any_python_web_framework_to/

https://flask.palletsprojects.com/en/stable/

Flask Notes:
https://flask.palletsprojects.com/en/stable/installation/

To activate the virtual env:
`. .venv/bin/activate`
to deactivate it:
`deactivate`

Flask CLI:
https://flask.palletsprojects.com/en/stable/cli/

Note: DEFAULT IS GET - anything else defined like

```
@app.route("/submit", methods=["POST"])
def submit():
    data = request.form["data"]
    return f"You submitted: {data}"
```

Running app:

```
export FLASK_APP=app.py
export FLASK_ENV=development  # enables debug mode
```

`flask run`
Then visit `http://127.0.0.1:5000/`

Pytest Fixtures: https://docs.pytest.org/en/stable/how-to/fixtures.html

URL naming convention: https://stackoverflow.com/questions/64029213/what-is-the-convention-for-separating-multiple-words-on-a-flask-route

exception testing in pytest: https://pytest-with-eric.com/introduction/pytest-assert-exception/

Redis: Redis (REmote DIctionary Server) is an open source, in-memory, NoSQL key/value store that is used primarily as an application cache or quick-response database.

https://redis.io/docs/latest/operate/oss_and_stack/install/archive/install-redis/

https://www.ibm.com/think/topics/redis#:~:text=Redis%20(REmote%20DIctionary%20Server)%20is,cache%20or%20quick%2Dresponse%20database.

Reading about idempotency:

https://osamadev.medium.com/idempotency-in-apis-making-sure-api-requests-are-safe-and-reliable-7d5cb51520fe

https://www.cockroachlabs.com/blog/idempotency-in-finance/

ChatGpt Queries:

I'm about to start my first ever Flask project with Python. Give me a 1-page review/introduction to Flask with everything I need to know to begin.

What's the recommended project structure when working in Python with Flask and Pytest. I need a "models" directory which contains data definitions.

When did python introduce data classes and should they be used over a regular class? what would be the use case for a data class vs implementing your own class.

How do you extract a module item from a Pytest fixture? I need to extract the fixture as an object and call its own method on it.

This way of initializing a timestamp is causing there to be precision errors on the number of decimal points in a time stamp. How can I modify it to be precise? it is defined outside of my class: @dataclass class FormulaCreatedEvent: name: str id: int created_timestamp: float = field(default_factory=time.time)