Zipline Take-Home Challenge

The problem

Zipline both flies autonomous vehicles and operates a full-fledged logistics system. We run operations out of our distribution centers, which we call "nests." At a given nest, we have both an inventory of medical supplies and a collection of autonomous aircraft, which we call "zips.". When a customer places an order, our operations staff pack the ordered products into one or more packages. Each package can carry a maximum of 1.8 kg worth of supplies. That package is then loaded on to its own zip, which flies directly to the hospital and drops off the package before returning back to the nest. Although we're improving every day, our zips will never be 100% reliable. Sometimes events outside of our control, such as weather, might force a zip to end its flight early and return home without making a successful delivery. We call this a "mission failure."

Doctors don't necessarily care how their order might be split up and scheduled, but they depend on transparency of their order status, and the expediency of our service.

Your challenge:

To capture the behavior of our nest, we have provided a small simulator called `nest_sim.py`. This simulator runs as a web-service, and provides APIs (documented at the end of this problem description) that allow you to query the list of available hospitals and inventory, as well as to dispatch new flights and monitor their statuses. These are internal "zipline-facing" APIs, which should not be directly exposed to a customer.

It's your job to design and ***begin*** the implementation of a customer-facing web application that will serve 3 major functions:

1. Allow a doctor to place an order for medical products to be delivered to a hospital
2. Allow a doctor to track the status of a placed order
3. Allow a regulator to audit all of the fulfilled orders -- which products have been shipped to which hospitals

The first part of your work should focus on defining the APIs that will be required, and building your own backend web-service (preferably in python) to supply them. For the sake of expediency, you are free to neglect worrying about authentication or durable storage -- using an in-memory database as we do in nest_sim.py is totally acceptable.

Remember, doctors care about orders, not flights, so the backend of your system will need to internally invoke the nest-sim APIs to translate the order into one or more flights, dispatch them, and track their progress in aggregate. We leave it to your discretion how to handle a tracked flight which has a mission failure.

Once you have a functioning backend, please implement a web-based front-end for either the order-placing interface, or the order-tracking interface. You will likely want some minimal scripts or stubbed-out interface pages for verifying the functionality of the remaining pieces of the system.

We have provided a collection of requirements below to help you think about your implementation. Please pursue the ones you think are most important first. You do not have to complete them all.. You should still consider how they might fit into future work, however. Working is better than done; done is better than perfect.

You may use the framework of your choice for this challenge. Please include a README with instructions for how to run your project (you may safely assume we are comfortable with tools like docker, virtualenv, pip, and npm).

Requirements

- The order UI SHALL allow a doctor to choose a destination hospital
- The order UI SHALL allow a doctor to select integer quantities of different products
- The order UI SHALL confirm whether or not an order was placed successfully
- The order UI SHALL NOT allow a doctor to place an order for out-of-stock items
- The order UI SHALL NOT require the doctor to manually split their order
- The tracking UI SHALL provide the current status of their order. Statuses are:
    - CONFIRMED -- order has been received by nest and is being processed
    - SHIPPED - order is on a plane en route to the hospital
    - DELIVERED - order has reached the hospital
    - DELAYED - order has been delayed due to a mission failure
    - CANCELED - order needed to be canceled because it could no longer be fulfilled
- The tracking UI SHALL provide an estimated time of delivery for each package
- The tracking UI SHALL alert the user when a flight is 5 minutes away from delivery

Nest Simulator

The nest simulator depends on Flask and Sqlalchemy and should run under either python2 or python3. We've included a requirements.txt that should be sufficient to install these dependencies via pip. The simulator also assumes that you run it from the same directory as the products.csv and hospitals.csv support files. If your environment requires doing something different you can specify the paths to these files with --products and --hospitals respectively.

By default the simulator runs on localhost and port 12345. The bind address and port can be overridden with --addr and --port respectively.

After posting to create a new flight, it will go through the following states:

- PENDING: The flight will stay in this state until you use the confirmation API to confirm it. Unconfirmed flights will time out after 5 minutes.
- CONFIRMED: Flight has been confirmed by client and is waiting to launch.
- SHIPPED: Flight is en route to the hospital.
- DELIVERED: Flight has successfully delivered its products to the hospital.
- MISSION_FAILURE: A flight had to turn around early and will not deliver.
- COMPLETE: All processing related to this flight is done.

The nest simulator provides the following APIs:

`GET /time`

returns the current simulator time in seconds since start

Example:

```
$ curl localhost:12345/time
1122
```

POST /step_time <INT>

To help with testing, you can post a json payload to this API containing the number of seconds to step forward the simulator in seconds.

Example:

```
$ curl -H "Content-Type: application/json" -X POST \
       localhost:12345/step_time -d "1000"
{"success":true}
```

GET /inventory

Returns a JSON list containing the products available at the nest. Each entry has the following keys:

- **id**: the product id
- **product**: the human-readable name of the product
- **quantity**: the number of units remaining at the nest
- **mass_g**: the weight of this product in grams

Example:

```
$ curl localhost:12345/inventory
[{"id":1,"mass_g":700.0,"product":"RBC A+ Adult","quantity":30}, ...]
```

GET /hospitals

Returns a JSON list containing the destination hospitals. Each entry has the following keys:

- **id**: the hospital id
- **name**: the human-readable name of the hospital
- **flight_time_s**: the expected time in seconds for a delivery to this hospital

Example:

```
$ curl localhost:12345/hospitals
[{"flight_time_s":2134,"id":1,"name":"Bigogwe"}, ...]
```

POST /flight {hospital: <ID>, products: [<ID1>, ... <IDX>]}

Create a new new flight by posting a json payload containing the ID of the hospital you want to place a delivery for, and the IDs of the products you would like to deliver. The endpoint will only create a flight if the hospital and product ids are valid, and the requested quantities are available Once the flight has been created, nothing will happen until the client confirms the flight. If a flight is not confirmed in under 5 minutes, it will timeout. Note, multiple products can be included in a delivery by including the product id multiple times in the list.

Returns a JSON dictionary with the description of the flight. See the documentation for GET /flight/<id> for details.

Example:

```
$ curl -H "Content-Type: application/json" -X POST \
       localhost:12345/flight -d '{"hospital": 1, "products": [1,9,9]}'
{"delivered":false,"hospital":1,"id":12,"products":[1,9,9],"state":"PENDING"}
```

`POST /flight/<id>/confirm`

Confirm a flight by id. This is a necessary step before any flight will proceed past the PENDING state.

NOTE: In our simulator, the fate of a flight is secretly determined at the time that it is confirmed by the client. Under normal circumstances, there is a 10% randomized chance a flight will fail (this is higher than reality, but makes the issues more visible in this exercise). For testing purposes, if you want to force a flight to succeed or fail, you can add an optional query parameter: ?fail=0 or ?fail=1 as appropriate. e.g.
`localhost:12345/flight/12/confirm?fail=1`

Example:
```
       $ curl -X POST localhost:12345/flight/12/confirm
       {"success":true}
```

`POST /flight/<id>/cancel`

Cancel a flight by id.

Example:
```
       $ curl -X POST localhost:12345/flight/13/cancel
       {"success":true}
```

`GET /flight/<id>`

Returns the status of the flight by id

Returns a JSON dictionary with the description of the flight.

- **id**: the flight_id that can be used for the remaining flight APIs
- **hospital**: the id of the hospital this flight is delivering to
- **products**: the list of product ids that are being delivered
- **state**: the current state of the plane
- **delivered**: whether the flight has delivered the products to the hospital
- (optional) **delivery_eta_s**: remaining seconds until delivery
- (optional) **return_eta_s**: remaining seconds until the flight returns home
- (optional) **note**: a note describing the resolution of the flight if in the COMPLETE state

Example:
```
       $ curl localhost:12345/flight/17
       {"delivered":false,"delivery_eta_s":1742,"hospital":1,"id":17,"products":[3,3],
" state":"SHIPPED"}
```