**Project 2: Backend REST Services**

**CS4981 ML Production Systems**

**Overview**

You are going to build a prototype for an end-to-end spam classification system. In this first project, you are going to handle the ingestion of emails and events from users interacting with a mailbox service. You will implement the email endpoint and store the received emails in PostgreSQL, an open-source SQL database. PostgreSQL has some advanced features such as supporting JSON documents as a column type that you will be taking advantage of. The mailbox service will log the events to files. You provided with a data simulator that will make requests to each of the services.

**Instructions**

**Part I: Implement Email Ingestion REST Service**

You are going to implement two REST services. The first service mimics an endpoint that receives emails. (Think of this as equivalent to a mail server the receives emails via SMTP.) This endpoint will be used for ingesting email into your system.

You should implement the REST services in Python. (This will be more important when you implement a prediction endpoint later on – it needs to be able to load and serve a model.) I recommend using the Flask framework. Here are two tutorials on creating REST services with Flask (one with basic Flask and the other using a library called [flask-restx](https://flask-restx.readthedocs.io/en/latest/)):

* [Flask REST API Tutorial](https://pythonbasics.org/flask-rest-api/)
* [Tutorial for Flask with the flask-restx library](https://flask-restx.readthedocs.io/en/latest/quickstart.html)

The service will have a single endpoint:

POST /email

The post request will receive a JSON object that looks like so:

{

"to": "someone@somewhere.com",

"from": "someone\_else@somewhere\_else.com",

"subject: "some important summary",

"body": "Lots of text.\\n\\nMore text."

}

Your REST service should return a new, unique, autogenerated integer id:

{

"email\_id" : 4138

}

At this point, your REST service isn't very useful. You need to store the emails somewhere. For this service, you are going to store the emails in a [PostgreSQL](https://www.postgresql.org/) or [MongoDB](https://www.mongodb.com/) database. Postgres is an open-source relational database that had advanced features like the ability to store JSON objects. I provide instructions for Postgres.

After you install Postgres, you can create a database and user like so:

Connect to postgres:

$ psql -U postgres

Create the database and switch to it:

postgres=# CREATE DATABASE email\_ingestion;

postgres=# \c email\_ingestion

email\_ingestion=#

Create the appropriate table:

email\_ingestion=# CREATE TABLE emails (

email\_id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,

received\_timestamp timestamp NOT NULL,

email\_object jsonb NOT NULL);

Create a user for the ingestion service and grant privileges:

email\_ingestion=# CREATE USER ingestion\_service WITH PASSWORD 'puppet-soil-SWEETEN';

email\_ingestion=# GRANT SELECT, INSERT ON emails TO ingestion\_service;

You can follow [this tutorial](https://www.digitalocean.com/community/tutorials/how-to-use-a-postgresql-database-in-a-flask-application) from DigitalOcean that explains how to use the psycopg2 library to interact with a Postgres database.

Your Flask application will need to construct the URI for accessing the database. The URI takes the following form:

postgresql://username:password@host:port/database

The default port for Postgres is 5432.

You don't want to store the login information in your application. The credentials could be checked into Git or would be stored in plain text in a Docker image or static binary. That would be incredibly insecure.

Modern applications, especially those designed to be deployed to cloud environments, follow a series of conventions defined by the [12 Factor App](https://12factor.net/) manifesto. The third factor is the strict separation of code and configuration. The configuration information is passed to the app through environmental variables. (Passing files to apps run in Docker containers is complicated.) Your app should expect the following environmental variables to be defined:

POSTGRES\_DATABASE

POSTGRES\_USERNAME

POSTGRES\_PASSWORD

POSTGRES\_HOST

POSTGRES\_PORT

The POSTGRES\_PORT variable is optional and defaults to 5432 if not specified. The rest are required. If any of the required variables are not provided, your app should report an error to standard error and quit with a non-zero exit code. You can access the environmental variables from with Python using the environ field of the os module.

You can run your application like so:

$ POSTGRES\_DATABASE=email\_ingestion POSTGRES\_USERNAME=ingestion\_service POSTGRES\_PASSWORD='puppet-soil-SWEETEN' POSTGRES\_HOST=127.0.0.1 python email\_service.py

You can test it with:

$ curl -X POST http://localhost:8888/email -H 'Content-Type: application/json' -d '{"to": "someone@somewhere.com", "from": "someone\_else@somewhere\_else.com", "subject: "some important summary", "body": "Lots of text. More text."}'

**Part II: Implement Mock Mailbox REST Service**

The second service you will implement is a mailbox management service. This service would serve as the backend for an email client (e.g., mimicking a protocol like IMAP or the backend for a web mail client). Your goal is not to the implement the logic to track the state of the mailboxes. Rather you want to implement a REST API that receive requests and log those requests. You will use the log of events to figure which emails are spam.

You can do everything in a single Python script named email\_service.py . The server should listen on port 8888 and implement a single REST endpoint:

GET /mailbox/email/<email\_id:int>

Returns a JSON object with the key "email" and an associated value of a String containing the entire email text

GET /mailbox/email/<email\_id:int>/folder

Get the folder containing the given email. Examples of folders include "Inbox", "Archive", "Trash", and "Sent".

GET /mailbox/email/<email\_id:int>/labels

Returns a JSON object with the fields "email\_id" and "labels". The value for labels is a list of strings. Valid labels include "spam", "read", and "important". No label may be repeated.

GET /mailbox/folder/<folder:str>

Lists the emails in a given folder. Returns a list of email\_ids.

GET /mailbox/label/<label:str>

List emails with the given label. Returns a list of email\_ids.

PUT /mailbox/email/<email\_id:int>/folder/<folder:str>

Moves email to the given folder. Folders include "Inbox", "Archive", "Trash", and "Sent".

PUT /mailbox/email/<email\_id:int>/label/<label:str>

Mark the given email with the given label. Valid labels include "spam", "read", and "important".

DELETE /mailbox/email/<email\_id:int>/label/<label:str>

Remove the given label from the given email. Valid labels include "spam", "read", and "important".

You can test your implementation using curl:

$ curl http://localhost:8888/mailbox/email/1 -X GET

{

"email" : {

"to": "someone@somewhere.com",

"from": "someone\_else@somewhere\_else.com",

"subject: "some important summary",

"body": "Lots of text.\\n\\nMore text."

}

}

$ curl http://localhost:8888/mailbox/email/1/folder -X GET

{

"folder": "Inbox"

}

$ curl http://localhost:8888/mailbox/email/1/labels -X GET

{

"labels" : [

"read",

"spam",

"important"

]

}

$ curl http://localhost:8888/mailbox/email/1/label/read -X PUT

{

"label": "read"

}

$ curl http://localhost:8888/mailbox/email/1/folder/label -X DELETE

{

"label" : "read"

}

Most tutorials show how to use the built-in Python logger library for logging. This library writes the output in an unstructured text format that is difficult to parse – it's expected that humans will read the log. Instead, we want to use an approach called structured logging in which the log file format is designed to be easily parsed. There is a popular Python library for structured logging called [structlog](https://www.structlog.org/en/stable/).

You can configure logging like follows:

with open("log\_file.json", "wt", encoding="utf-8") as log\_fl:

structlog.configure(

processors=[structlog.processors.TimeStamp(fmt="iso"),

structlog.processors.JSONRenderer()],

logger\_factory=structlog.WriteLoggerFactory(file=log\_fl))

The logging needs to be configured before you call app.run(). In the event handler functions, you can access and use the logger as follows:

logger = structlog.get\_logger()

logger.info(event="email::id::folder::put",

email\_id=email\_id,

folder=folder)

The output file will contain one JSON object per line. It will look like so:

{"email\_id": 1, "label": "unread", "event": "email::id::label::delete", "timestamp": "2022-10-22T00:35:36.977079Z"}

{"email\_id": 1, "label": "unread", "event": "email::id::label::put", "timestamp": "2022-10-22T00:35:39.189347Z"}

{"email\_id": 1, "event": "email::id::labels::get", "timestamp": "2022-10-22T00:35:42.013160Z"}

{"email\_id": 1, "event": "email::id::folder::get", "timestamp": "2022-10-22T00:35:44.793943Z"}

**Part III: Test with Simulator**

1. Download the email\_json\_dataset1.zip and spam\_simulator.zip files from the shard Box folder and unzip it:

<https://msoe.box.com/s/l9xl4udcwvqzrzwtxwrqw4bejehomk2b>

2. Create a second virtual environment named simulator. Install the updated versions of pip and wheel. Install the simulator's dependencies in your Python environment:

(simulator) $ python install -r requirements

3. Make sure that you have Postgres and the two REST services running.

4. Now, run the simulator:

(simulator) $ python spam\_simulator.py simulate-user --email-dir path/to/email\_data --email-url http://127.0.0.1:8888/ --mailbox-url http://127.0.0.1:8889/ --number-emails 1000 --average-events-per-email 20

5. Check that Postgres table and the local log file have entries.

**Submission Instructions**

Store your code in a private GitHub repository and give read access to the user "rnowling". Submit the URL to Canvas.