Johnny Wong September 22, 2022
CS554-WS Lab 3
I pledge my honor that I have abided by the Stevens Honor System.

Scenario 1: Logging

To create a logging server that can support any number of other arbitrary pieces of technology, an Express server (our web server running on NodeJS) alongside some NoSQL Database like MongoDB would be able to meet all of the requirements. Log entries would be stored as documents in the Mongo database. These documents would contain the common fields on the outermost level. The customizable fields can be contained in another object within the *details* field. See example below for more clarity:

```
{
    "commonField1": "something1",
    "commonField2": "something2",
    "details": {
        "customField1": "hi",
        "customField2": "bye",
        "customField3": "and so on"
    },
    "user": "patrickHill554"
}
```

Users would be able to submit log entries by sending them to the server through a POST request. Users must be logged in first in order to attach a username to each log entry. In the request's body, the common fields should be filled out fully and the customizable fields should be contained in the *details* object, as seen above. This POST request will be routed to some kind of submit log route for the Express server to process and make sure the request was made with correct inputs. To query log entries, users would be able to send GET requests to some route on the Express server that queries the database using the fields **from *req.query***. An example would be sending a GET request to the route `/query?commonField1=something1`. Users would be able to see their own log entries by querying on the user field for their username.

Scenario 2: Expense Reports

To create an expense reporting web application, I would leverage an Express web server (running on NodeJS) and a NoSQL database like MongoDB. Although expense data points have a uniform data structure, I would still opt to use Mongo because of how I can add more potential fields in the future to each expense document without making changes to an entire collection. An expense document could look something like this:

```
{
    "id": "some-unique-id",
    "user": "patrickHill123",
    "isReimbursed": true,
    "submittedOn": "06/21/2022",
    "paidOn": "06/21/2022",
    "amount": 419.99
}
```

I would choose to use an Express web server because of how easy it is to utilize other Node packages such as **Puppeteer** and **Nodemailer** to be able to create a PDF of an expense and email it to the user that submitted the expense. Templating can be solved using the package **Express-Handlebars**.

Users can submit an expense by filling out an HTML form on a submit page with fields such as “Submitted On”, “Paid On”, and “Amount”. Users must be logged on to submit the form and the web server will perform substantial input checking before saving the expense in the Mongo database. When the user is reimbursed, a server admin can POST to some reimbursement route with the expense’s id in the request body (i.e. `/reimburse`). This route will set the *isReimbursed* field to true. Puppeteer will generate a PDF of an expense document HTML page that was created using an Express Handlebars template with the relevant expense data found in the database. It will then be emailed to the expense’s corresponding user leveraging the Nodemailer package.

Scenario 3: A Twitter Streaming Safety Service

I would run an Express server to create routes and listeners for this safety service. I would use the Twitter API called **PowerTrack** in order to be able to filter for Tweets that this service is supposed to trigger on such as the combination of (“fight” OR “drugs”) AND (“SmallTown USA HS” OR “SMUHS”). PowerTrack will be able to provide my service all of the data I want and need through a simple GET request after supplying my ruleset for triggers. For the email and text alert systems, I would use the Node packages **Nodemailer** and **Twilio** respectively. This service can be built to be expandable beyond my local precinct by hosting this service on a cloud server such as AWS or Azure. That way, any precinct would be able to change rule sets to include their specific coordinates they want to filter for as well as their own trigger keywords; the service would be containerized for each precinct and their respective rulesets. I would use Redis to store possible incidents and triggers along with their investigation status because of its retrieval speed and ease of changing an investigation status when storing these possible incidents as flat objects. I would use MongoDB to store all Tweets to be able to retroactively search through them since PowerTrack is giving my service back JSON data to begin with. I would also store all of the media attached to these Tweets in MongoDB as well - if this media was greater than 16 MB I would have to leverage GridFS to divide each file into chunks and store them each separately. The real time streaming incident report can be a page created in React that maps out an array of incident data to a list of user friendly components to inform users of these incidents. This page can use the hook useEffect to fetch this data from our service and periodically refresh it in order to update it in real time.

Scenario 4: A Mildly Interesting Mobile Application

The API would be written in Node operating on an Express server. Due to the geospatial nature of the data, we can utilize MongoDB's method of storing geospatial data as GeoJSON objects. GeoJSON objects each contain a GeoJSON Point that contains the coordinates of where an image was taken. To store images for the long term cheaply, we can store them as chunks using GridFS in MongoDB. For short term, fast retrieval, we can utilize Redis to store hashes of binary data of images and their corresponding coordinates. Using Redis to cache users' recently taken images would provide them the quickest retrieval times, but will become very expensive with larger memory requirements. We can opt to store and fetch them from MongoDB when memory approaches capacity, albeit being a slower solution. The API written on the Express server will consist of routes for saving (creating) GeoJSON objects of images, reading and querying for specific images, updating images with captions or crops/filters/etc., and deleting images users do not want anymore.