Max Shi

Scenario 1:

For this logging application scenario, I would have to create a full-stack application to make this server usable to the client. I would store logs in a NoSQL database, such as MongoDB, where I can create a collection for all users and a collection for all logs. This logs collection would be indexed by each user's ID, such that a user would be able to quickly access their own logs, while not accessing other users' logs. These log entries would be submitted with a simple frontend, which could be made with React, to a backend made with Express, in order to authenticate users and provide form validation to users in order to submit the common and customizable fields in their log entries. Furthermore, this frontend could make requests to the server in order to get all log entries, and parse data to effectively query the data on the client side. Alternatively, parameters could be passed to the serve to query for data directly from the database. Either way, these entries would be displayed on the frontend application in a table. We could also expose and API to the backend server that makes use of authentication keys in order to allow clients to bypass the frontend that we provide as standard to the client.

Another consideration is the amount of data being uploaded to this logging application. If we are serving thousands of clients with this application, the indexing in something like MongoDB might be too slow. Thus, alternative solutions such as Elasticsearch would improve speed of queries while being able to store large amounts of data.

Scenario 2:

For simplicity's sake, an express server using handlebars for web templating can accomplish this task. Handlebars is perfect for creating a form with the desired fields (without user customization) and doing form validation before sending it to an express endpoint. This express endpoint could store the data in a NoSQL database, but with these consistent fields, a SQL based database may make sense for fast querying. There seems to be some fields in the data with relational connections, such as reimbursedBy and user, which could refer to other tables in a SQL database. Emails can be handled by the Nodemailer package, which supports attachments, for the PDF to be sent. To generate the PDF, handlebars can generate an HTML template to load the data from the fields, and that HTML can be fed into wkhtmltopdf to generate the PDF. Then, the PDF can be attached to the email from Nodemailer and sent to the user.

Scenario 3:

In this scenario, I would utilize the MERN stack. On the backend, I would have to use a worker connected to the streaming Twitter API to get tweets in real time and process the information. This worker could search through the streaming data to search for the necessary keywords to process a trigger to know when to alert users and authorities. Emails could be sent using Nodemailer, and a text alert system could be set up through services like Twilio. The triggers could be stored in MongoDB to be accessed and updated by the worker, but long-term storage of all the tweets should be handled by something like

Elasticsearch, which can process the large amounts of data that would require. The historical database, while not necessarily a large amount of data, could benefit from Elasticsearch based on the fast querying capabilities if the dataset were to grow large. Storing media in long term storage could be handled similarly to scenario 1, where we use AWS to store large files to be accessed occasionally. Furthermore, this backend would expose a REST endpoint to allow for CRUD updates to the combinations of keywords to trigger alerts.

The React frontend would be able to handle both adding keywords to the triggers as well as the streaming incident report. Adding keywords to the triggers would require some sort of user login system to ensure correct permissions before adding the triggers, so something like Firebase authentication could be used to authenticate users. Then, a simple form could be used to send CRUD updates to the backend. The streaming incident report could be implemented with socket.io on the frontend and backend to stream events from the backend worker to be displayed on the frontend.

To expand to different precincts, the backend could store different sets of data under different IDs to separate the precincts, and then use that as a parameter in the routes to create different endpoints for each precinct to use. Therefore, new precincts can be dynamically added to the application.

Scenario 4:

The backend for this application would be an express endpoint joined with a NoSQL database to support the CRUD operations on the users and the 'interesting events'. This express server can also serve an administrative database through a simple templating framework such as handlebars, as something only exposed to the administrators does not need something as heavyweight as React or Vue for the frontend. The API being in express makes for easy interfacing as a REST API, which also supports file uploads through middleware. In order to store both the data for each image as well as the image itself, we would have to combine NoSQL with a technology such as AWS's S3 buckets, which can store files such as images for long term storage. For short term storage, Redis allows the storage of strings as binary data, up to 512MB, therefore this can be used to store images for short term, fast retrieval.

For the geospatial data, a universal format such as latitude and longitude could be used to index each of the images in their collections. To speed up searching for nearby interesting images, a 2-D array can be used to index and store entries by the degree number in their locations, e.g. a location of 40.7440° N, 74.0324° W could be stored at the pair [40, 74]. This would allow us to group images together, so we do not have to search the whole database to find nearby images to show the user or render on a map.