Abhishek Khatri
20012198

CS554 Web Programming 2 – Lab 03

1. Logging Server

I would store the logs in MongoDB since we want to allow the clients to be able to add custom fields into the logs. Some of the common fields for the logs would be 'timestamp' in UTC, 'type' which could be one of [error, info, warn] with default as info, 'host' which would be provided by the client, and the 'log' string itself. We could add the querying functionality using GET REST API and query parameters to easily query logs based on the keys in each log.
The clients could create or fetch their logs using APIs or some kind of library that would require authentication (using API keys or any other way). If it is being done used using a library, they could create a configuration instance after which they could use the provided functions to create/fetch their logs.
I would choose NodeJS for server as it fulfils the requirements to build such a server and integrates well with MongoDB using its official library.
Additionally, I believe we could also have Apache Hadoop integrated here, which would get the data from MongoDB, analyze it and generate some insights which could be stored back into MongoDB. These insights could be used by the client and viewed as a dashboard.

2. Expense Reports

I would store the expense records in some sort of relational database as the columns and type of data is fixed which will give us more efficiency with such data and queries. We could use AWS RDS for MySQL as it would be fully managed and MySQL is perfect for this, since our queries and datatypes are not going to be very complex for us to use something like PostgreSQL and increase complexity. For server, I would choose Express on NodeJS as it suffices the requirement for this along with a type-safe ORM such as Prisma which also comes with a database admin dashboard.
For templating we could use handlebars with Express, and 'wkhtmltopdf' for generating PDFs from these HTML and CSS templates. For sending out emails, I would choose SendGrid as it is a leading SMTP server with a very easy learning curve. It has very good integration and library for NodeJS with a generous free tier.

3. A Twitter Streaming Safety Service

I would use Twitter's filtered stream API to get real time stream of public tweets based on the filter keywords. When filtering based on keywords, we could use location keywords as well, say tweets with the keyword Hudson would direct tweets to the police station in the Hudson county, we could maintain a list of counties we want to expand to, or make use of some simple NLP APIs or libraries that will try to make sense of the tweet text and decide which county to alert for that particular tweet.
To make sure our system is stable and running, we could make use of some fault tolerance techniques such as replication, and deploy using Kubernetes so it can spin up a pod as and when required. We would also make use of some error reporting tool such as Sentry that would alert us when there is some error with our application on which we would have to take action.
Finally, to make all of this work seamlessly we could have a simple CI/CD pipeline to make sure

deployments happen safely with all tests with no downtime. For triggers, we could use Firebase real time database along with Firebase notifications as they are built for this very purpose, and make use of SendGrid SMTP for email or SMS notifications. As for history of tweets, I would make use of relational database since tweets have a fixed set of fields such as user, time, tweet, etc, twitter themselves store tweets in a relational database. Querying on a relational database would be much quicker. We could also maintain tables linking tweets to keywords and locations and query on them when needed. To store any media linked to the tweet, we could make use of any bucket such as S3. I would make use of NodeJS for this as it has official SDKs for twitter API, Firebase, SendGrid, and AWS. Along with this I would also make use of a type safe ORM such as Prisma which will provide me with all data functions for the database and even handle migrations.
The online website to update the keywords could be made using handlebars or a simple HTML, vanilla JS since it would be a simple website that could communicate with our server with the help of REST APIs.

4. A Mildly Interesting Mobile Application

For the purpose of this application, we can make use of MongoDB's GeoJSON data. We would store the image itself in a storage bucket for long term, cheap storage and its link in the mongoDB collection along with the coordiantes. MongoDB's GeoJSON makes it simple to query items within a radius based on coordinates. I would write my API using Express on NodeJS as it integrates very well with MongoDB and if we were to build a cross platform mobile application with something such as React Native, it would be full stack JS reducing maintenance cost as less developers would be able to work on the entire stack.
Additionally, we could make use of some Vision APIs provided by most cloud providers as well as some 3rd party APIs to tag images and store those tags with the image in MongoDB. This could then be used to show the user interesting things around them based on their interests.