CS-554 Lab 3
Choosing Technical Stacks

Naveen Mathews Renji, 20016323

Scenario 1: Logging

To Create a Logging Server for any number of other arbitrary pieces of technologies.

1. Database to store the Logs – MongoDB
2. Submission of Logs – ReactJS, NodeJS
3. Querying of Logs – ReactJS, Express.JS, MongoDB, NodeJS
4. Viewing Logs – ReactJS, Express.JS, MongoDB, NodeJS
5. Web Server – Express.JS, NodeJS

NodeJS will be the JavaScript language we use for server side and client-side JavaScript. MongoDB will support common fields as well as any number of customizable fields for an individual log entry. Thereby making every log customizable. MongoDB also enables querying based on any field and also easy integration with NodeJS and Express.JS. Users can use a front-end to submit their logs to the backend using a simple ReactJS frontend that accepts user logs through a simple form that also has a button that allows users to add additional fields to the form which will enable them to log customizable entries.

A user can send a query using the ReactJS form for querying which can have optional input fields as the search parameters. This request can be sent to the backend using the Express server request which hits the GET route for the querying API from the MongoDB which retrieves the right Log from the MongoDB server. Once the logs are retrieved from the backend using Express requests to the MongoDB, we can send them to our front-end React page that renders the information provided either as a JSON or in a tabular form with a dynamic table that allows custom fields. I would use Express.JS as my web server as it is usable by all the Techs that are being used here and it is also a very simple to use server that allows API endpoints for submitting and querying the logs.

Scenario 2: Expense Reports

Making an Expense Report web application

1. Database to store structured expenses data – PostgreSQL
2. Submitting the expenses – NodeJS, Express, ReactJS
3. Reimbursement of Expense – NodeJS, ExpressJS, Nodemailer, PDFKit,
4. Web Server – Express.js, NodeJS
5. Front-End - ReactJS

The Expense Report web app will run on an express server as it is easy to use and supports all the other tech I have chosen to use. NodeJS server-side runtime environment will allow the use of various Node.js modules that I am using such as PDFKit and Nodemailer. NodeJS will also support my API endpoints for storing and querying expenses from my DB. I will be using PostgreSQL as my DB as it is a relational database and supports complex queries which will be useful to generate calculated values from the database which will be useful in an expense report. The id field can be the unique identifier in our db. Using the API endpoints created using express routes, nodejs and postgresql, we can retrieve the data to generate an expense report. The data retrieved will be passed to the PDFKit node module using a template generated using the EJS module which allows template generation for dynamic HTML pages. Once the endpoint is hit, the Template will be generated and then we will use the Nodemailer node module to send the email to the users. As Nodemailer supports attachments. Therefore, if the isReimbursed is true, we can generate the PDF using EJS template and PDFKit and send the email with the attachment using Nodemailer to the user.

Scenario 3: A Twitter Streaming Safety Service

Creating a Twitter Streaming Safety Service for a local Police Department

1. Twitter API – Twitter Filtered stream and User lookup API
2. Database for storing trigger keywords – MongoDB
3. DB for historical log of tweets – MongoDB
4. Web server technology – Node.js and Express.js
5. Real-time streaming incident report – Socket.IO
6. Storing media files – Amazon S3
7. SMS API – Twilio Message API
8. Email to officers – Nodemailer node module with EJS and PDFKit to structure and create pdf which can be attached

I would use the Twitter API Filtered Stream. This API provides much of the functionality required. I can create rules, either on my own or with the assistance of the police, that will contain a combination of trigger words. These rules will flag tweets that match those rules. A streaming connection can then be established to receive tweets in JSON format through a persistent HTTP Streaming connection. To create these rules, I will use a Frontend React app that has an input field to take in keywords tagged as trigger or critical trigger words. These words will be used as rules in the Twitter Filtered Stream API to match tweets. React and Nodejs can be used to update or delete trigger words. Words can be stored in MongoDB for a specific police or precinct profile.

When triggering tweets are detected, Nodemailer and NodeJS can send an email to precinct officers. The Twitter user's lookup API can be used to fetch the user details, which will be combined with the tweet and other metadata such as location and tweet edits. EJS templates will be used to format the tweet data, and a PDF can be generated and sent as an attachment to the precinct officers using PDFKit and Nodemailer. To notify officers through SMS, I will use Twilio messaging API when a tweet meets certain critical trigger word combinations. For storing the historical database, I will use MongoDB and Amazon S3 to store media used in tweets for the long term. These two databases will manage the entire application data. MongoDB will store a log of all tweets in a collection, allowing for queries based on matching criteria.

To create a real-time streaming incident report, Socket.IO can be utilized. As new tweets are parsed, Socket.IO can push them to the browser in real-time. This allows officers to see the tweets as they are being processed and monitor their threat level as determined by the keywords combinations. A combination of Node.js and the React front-end framework can be used. When a new tweet is parsed by the server, Socket.IO can emit a message containing the relevant tweet data such as tweet text, user, and threat level. The client-side code can then use this data to update the incident report view in real-time.

Scenario 4 : A Mildly Interesting Mobile Application

1. CDN for short-term storage – Cloudflare
2. DB for user information and admin info – MongoDB
3. DB for long-term storage of media- amazon S3
4. Geospatial capability: MongoDB geospatial indexing
5. Web server – Express.JS and NodeJS
6. Front-end – React and Chart.js for management visualization

To handle the geospatial nature of the data, I would use a combination of GeoJSON and the MongoDB Geospatial Index feature. We can use GeoJSON objects to store the location data of the interesting events. These objects can then be indexed by MongoDB, allowing us to efficiently query for events within a certain radius of a given location. For storing the images, we would use Amazon S3 for long-term and cost-effective storage. However, for quick image loading, we would utilize a CDN service such as CloudFlare, which can cache images geospatially. This means that images that are within a certain radius of the user can be retrieved quickly from the CDN's cache. For the API, I would use Node.js and Express.js to create the necessary routes for CRUD operations on users and interesting events.

User authentication and login can be handled using middleware provided by these frameworks. As for the database, we would use MongoDB to store users and S3 for event data. We would create separate collections for users and admins. MongoDB is a good choice for this scenario because it has built-in support for GeoJSON objects and Geospatial Indexing, which would allow us to easily handle the geospatial data. To create the administrative dashboard, we would use React. The dashboard can retrieve user and event data from the MongoDB server and S3 and display it in an organized and easy-to-use interface. We can also use the Chart.js library to create visualizations of metrics such as the number of interesting events or the number of interesting events around a particular location.