Lab 3

Peter Rauscher

“I pledge my honor that I have abided by the Stevens Honor System”

Scenario 1: Logging

In this scenario, I would use a stack consisting of an Express server, Handlebars rendering engine, a SaaS log-management software called Loggly, and an npm package called *node-loggly*, and Node.js library that offers easy interaction with the SaaS product. I chose Loggly because it is accessible over a standard REST API, making it intercompatible with any other languages or technologies existing within our scenario (mobile applications, etc). Additionally, it supports an arbitrary length and complexity of data in each log entry, where data can be submitted as singular values, arrays, or any type of JSON object, allowing for new fields or attributes to be added on the fly. Searches and queries are also done over REST methods, but the node-loggly package supplies a JS function to perform these as well. Searches can be for substrings or filtered by multiple field values, much like in MongoDB, by passing a multi-attribute JSON object in. The user-authentication system is already handled by Loggly, so as each user signs up and logs in on our Express application we would simply pass those credentials on to the Loggly service. Submitting, searching, and displaying these logs would all be done through front-end web pages on our Express application, and rendered using Handlebars templates. I would use this stack because it covers any arbitrary complexity of logs as the application requirements change, and it is implemented in a stack I am already comfortable with and have experience in.

Scenario 2: Expense Reports

The backbone of this technical stack would again be an Express server, with a Mongo database, accompanied by npm libraries and packages. The main web application would consist of a user registration page, login page, an expense report submission page for users, and some management portal which allows creditors of these expenses to view all reports and specify the email from which receipts will be sent. When signing up, users would create their username and password, and specify the email to which they would like to receive their reimbursement receipts. When management logs in and pays the expenses, they would indicate so and the *paidOn* date would be automatically populated in Mongo, and a PDF would be generated using an npm package called *pdfjs* and filed within the server’s filesystem, under the *id* of the report in Mongo. The pdf would be populated with some company letterhead using the *doc.header()* and *doc.footer()* methods from *pdfjs,* and populated with content from the MongoDB entry on the report using *doc.text()* coupled with formatting and alignment methods also provided. This pdf would be emailed to the requesting user’s *email* from Mongo, being attached with the *client.send({attachment: { path: ‘path/to/receipt.pdf’ }})* method of *emailjs*.

Scenario 3: A Twitter Streaming Safety Service

The website portion of this application would again consist of an Express server, using Handlebars to render web pages, and Mongo to handle database operations. At minimum, the following pages would be necessary: A trigger creation page where keywords could be added or deleted, a live feed incident report page, a historical timeline of all tweets within the area, and a historical timeline of all tweets that triggered an alert. Twitter's own PowerTrack API allows for every public tweet made to the platform to be filtered by a specialized language, so this API would fit the bill for this project. The geographic radius could be fixed based on the locality of the police department, and keywords would be fed into the API directly from the CRUD operations on the Express application side. Triggered alerts that include task-force specific keywords would also be filtered at this level, directing drug-related incidents to the drug enforcement team and EMS when necessary, or gun-violence incidents to the SWAT team. MongoDB would handle long-term storage of the tweets and blob storage of the media attached to those tweets, and depending on the hardware limitations of the server the application would run on, I would utilize a Redis cache of all the tweets matching these criteria dating back at least 24 hours, so immediate and relevant incidents could be assessed and dealt with as fast as possible. For email alerts, I would utilize the *emailjs* npm package referenced in the previous scenario. For text alerts, I would use the MessageBird API despite not having an npm library, as it is accessible via REST methods and could be interfaced with using *axios*, a package which would already be included with the stack for use with PowerTrack. To ensure the system were constantly stable and online (as crime never sleeps) I would create redundant virtual private servers with AWS, which has data centers worldwide and is protected by CloudFlare from DDOS attacks or DNS poisoning. This way, even if one server was compromised from cyberattacks, power outages, or natural disasters, an identical system from a different geography would take its place.

Scenario 4: A Mildly Interesting Mobile Application

For this application, I would use Amazon S3 storage buckets for long-term storage of pictures because they are cheap, have multiple backups, and always available. The frontend would be a React Native application, which would allow it to run completely cross-platform, including on smartphones as intended. The administrative dashboard and backend API would consist of an Express server serving images first from a Redis cache of the most recently uploaded pictures, then from a MongoDB server storing its data on an S3 bucket. As for the geospatial component, when a user loads the picture feed, the server would first request ALL of the most recently uploaded images from Redis, and filter by using two npm packages, called *read-exif* and *distance-between-geocoordinates*. Using *read-exif*, we would read the coordinates from which each photo was taken from its EXIF data, and then use *distance-between-geocoordinates* to compare that to the user's current coordinates. If it is within some reasonable radius, we would allow that photo through the filter, and display it on the feed. Once we have gone through the Redis cache, if there are not enough photos to fill the user's feed, we would perform the same operation on the photos residing on the Mongo server.