CS-554 Lab 5

Eric Altenburg

April 15, 2021

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

Scenario 1: Logging

Prompt

In this scenario, you are tasked with creating a logging server for any number of other arbitrary pieces of technologies.

Your logs should have some common fields, but support any number of customizable fields for an individual log entry. You should be able to effectively query them based on any of these fields.

How would you store your log entries? How would you allow users to submit log entries? How would you allow them to query log entries? How would you allow them to see their log entries? What would be your web server?

Response

When tasked with creating a logging server to hold logs for various technologies, there are obviously going to be differences in the fields for the individual log entries. Because of this, I would opt to use MongoDB to create a database where each technology will have its own collection of log files. Breaking the files up into collections for each technology will allow for the users to easily access any specific log while not having to worry about overlap between different technologies. For a user to submit a log entry for a given technology, they would have to interact with either an interface on a web server where each field of a form would directly correspond to the file's field, or they can choose to directly submit a request to the web server's URL through an application such as Postman. With the latter option, because the fields are not immediately specified they would have to have prior experience with this service this way they know which fields are needed for a technology's log format. The former form would be constructed using React as it is likely that there might be several hundred logs that will need to be displayed. To help fix this issue, pagination is extremely easy to implement with various built in npm modules or through the use of simply passing around props.

To query a log entry, the user would first have to be properly authenticated with their login to the service through the use of cookies. If they are not authorized, they must login or sign up for the service. Once they are authorized, they will then be shown a list containing all the logs they have uploaded to the server through a simple `getAll()` data function. Optionally, they could apply a filter to their list of logs by either looking for a certain log ID, field, or to simply search through the list for a keyword. Finally, to build this service's web server, I would use Node.js to create the web app and create the data functions.

Scenario 2: Expense Reports

Prompt

In this scenario, you are tasked with making an expense reporting web application.

Users should be able to submit expenses, which are always of the same data structure: `id`, `user`, `isReimbursed`, `reimbursedBy`, `submittedOn`, `paidOn`, and `amount`.

When an expense is reimbursed you will generate a PDF and email it to the user who submitted the expense.

How would you store your expenses? What web server would you choose, and why? How would you handle the emails? How would you handle the PDF generation? How are you going to handle all the templating for the web application?

Response

The web application I would make for these expenses would all be kept in a MongoDB database with one single collection called `expenses`. Within this collection, it will have an `id` as an `ObjectId`, `user` as `String`, `isReimbursed` as `Boolean`, `reimbursedBy` as `String`, `submittedOn` as `String`, `paidOn` as `String`, and `amount` as `Number`. This data would then work directly with a

Node.js web server since the two technologies work well together and operate efficiently. When a specific expense is reimbursed, this will generate a PDF through the use of LaTeX, and it will then email it to the user. The PDF generation will be handled through the npm module node-latex where it will received a `.tex` template created by javascript to then be compiled by the module. As for the email handling, I would utilize the npm module Nodemailer this way once the PDF is generated, it will then grab it and immediately send it to the specified user. Finally, to accommodate the various templates that will be required for this web application, handlebars will be used.

Scenario 3: A Twitter Streaming Safety Service

Prompt

In this scenario, you are tasked with creating a service for your local Police Department that keeps track of Tweets within your area and scans for keywords to trigger an investigation.

This application comes with several parts:

- *An online website to CRUD combinations of keywords to add to your trigger. For example, it would alert when a tweet contains the words (`fight` or `drugs`) AND (`SmallTown USA HS` or `SMUHS`).*
- *An email alerting system to alert different officers depending on the contents of the Tweet, who tweeted it, etc.*
- *A text alert system to inform officers for critical triggers (triggers that meet a combination that is marked as extremely important to note).*
- *A historical database to view possible incidents (tweets that triggered an alert) and to mark its investigation status.*
- *A historical log of all tweets to retroactively search through.*
- *A streaming, online incident report. This would allow you to see tweets as they are parsed and see their threat level. This updates in real time.*
- *A long term storage of all the media used by any tweets in your area (pictures, snapshots of the URL, etc).*

Which Twitter API do you use? How would you build this so its expandable to beyond your local precinct? What would you do to make sure that *this system is constantly stable? What would be your web server technology? What databases would you use for triggers? For the historical log of tweets? How would you handle the real time, streaming incident report? How would you handle storing all the media that you have to store as well? What web server technology would you use?*

Response

To allow my service to communicate with Twitter, I would opt to use the npm module Twitter API Client because of its compatibility and use of promises which will allow for cleaner code instead of working with callback functions. Additionally, this module comes with a built in cache which would allow me to forgo the use of Redis in my tech stack. If the application were to be expanded, then there would have to be multiple different collections for each precinct to hold the data, and the use of Apache Hadoop would be useful this way a precinct can perform their own crime analytics across all their machines. As for stability, I would use Amazon Web Services this way the service is not being run locally which can leave it susceptible to attacks or natural disasters; outsourcing would be the most cost effective and efficient option. Since we are using various npm packages, it is only natural to use Node.js for the web server technology.

As it was previously mentioned, since additional MongoDB collections would be used as the service expands beyond the precinct, MongoDB would be the main database used to house various triggers for investigations. Per precinct, it will store tweets, images, and other forms of media from said tweets to ensure that a history of all information is saved and capable of being searched for at a later date. However, since the media will likely be comparatively large files to the tweets themselves, the use of Amazon S3 to host these forms of media will allow us to offset the storage issues by simply referencing the link to these items in the MongoDB database. When it comes to ensuring real-time streaming of the incident reports, the use of socket.io will allow for the website to easily communicate back-and-forth with the server.

In short, all the technology used for this service would be Node.js, MongoDB, the Twitter API Client, Amazon Web Services, Amazon S3, and socket.io.

Scenario 4: A Mildly Interesting Mobile Application

Prompt

In this scenario, you are tasked with creating the web server side for a mobile application where people take pictures of mildly interesting things and upload them. The mobile application allows users to see mildly interesting pictures in their geographical location.

Users must have an account to use this service. Your backend will effectively amount to an API and a storage solution for CRUD users, CRUD 'interesting events', as well as an administrative dashboard for managing content.

How would you handle the geospatial nature of your data? How would you store images, both for long term, cheap storage and for short term, fast retrieval? What would you write your API in? What would be your database?

Response

Since MongoDB is capable of handling goespatial data such as GeoJSON, it will be used to allow us to easily query a location. However, in order to achieve the GeoJSON data type, it is likely that ogr2ogr would have to be used as many geospatial data is often encoded as shapefiles; this allows us to easily convert between data types. In order to store images for the long-term and for cheap storage, using Amazon S3 once again will allow us to offload the hosting of these images to an outside source while we can simply keep the link to them in our database making it lightweight. As for short-term retrieval, the use of Redis will allow us to cache images that are frequently used or have been used recently. Using this allows us to bypass the need to pull our images from Amazon S3 every time we need to serve it to the user. As for the API endpoints, they will be created with Python and Flask since I have prior experience with these technologies.