SAIC Code Assessment

Rohan More

Date: 6/14/2021

**Demo**

A demo of this assessment can be viewed at <https://saic.rohanmore.xyz>. Please note that if you receive status codes in the range of 400 to 500, hard reload the page and try again. This is an issue with CORS.

**How to use**

In order to use this application, you must deploy the jar binary into a lambda function. Change the credentials and bucket within the java source code and recompile with dependencies. Next, create an API from AWS API Gateway and create a resource. The resource will need at least a post method. This method will use Lambda integration to run the jar and get a response. Change the API, and API key variable in the “App.js” file of the web app. Once done, build and deploy.

**How it was made**

When I was given the code assessment, on the first day I created a new AWS account and played around with services I knew I needed. These services were S3, Rekonigition, Lambda, and API Gateway. I just wanted to get a feel of how things worked and did research on how to develop with these services. After playing around with these services I decided to take it one step at a time. The total time it took to complete this assessment was around on average 4-5 hours of coding, debugging, and testing through the span of 3-4 days (~20 hours). Research for this assessment took around 2 hours.

I first wanted to develop a backend to compare the faces within a bucket. I read documentation supplied by Amazon and figured it was very straight forward. At first, I wanted to use the serverless took kit, but I had trouble integrating different packages, so I switched to Maven. With maven I could easily configure dependencies and switch between different compilers. Because this program only needed to do one task, as soon as it ran, it will compare faces and respond with a json file. My logic for this was to take a starting point, the first image and loop through the rest and compare it to the starting image. Another approach I tried was to use a collection and indexes the faces, but I could not figure out how to get unmatched faces. With each comparison, I will use the source and target image names and the confidence/similarity of the comparison. I did this because each comparison provides a lot of data, but to make it simple I decided to stick with these 4 values. I wanted to speed this up, so I used an executor service with 5 threads which called a function to compare faces. However, this version does not work with lambda as it only throws status code 503 when you try and run it. I will still provide the source code for this version as it is much faster.

After completing the backend, I wanted to create the frontend of the assessment. I created a react-app and added a button and a table. To invoke any API, I needed to use fetch or something similar. I ended up using axios as I wanted to try something new and axios is very popular. With the table I created pagination to minimize the amount of data displayed at a time. Next, I moved onto the API. I ended up using AWS API Gateway to create a Restful API. This API consisted of a resource called “trigger-compare”. Within this resource, it contained two methods options and post. The options method was used as a preflight check that is needed for CORS. The post method invoked a Lambda function which contained our backend java program. This method also utilized proxy integration to display only the body of the backend response. Once configured, I tested the API with postman to confirm it worked. After completing the API, I implemented axios to make a post request to my cors-anywhere proxy and my API I just created. Once the API responds with data, react will automatically put it in a table and display it. To achieve this, I used the useState hook. If the API responds with an error, it will display it under the button instead of a table.

**Documentation and Testing**

API – API Gateway

In order to make appropriate REST calls, an API is needed. As this assessment does not require multiple methods, a simple API was developed. The API was created used API Gateway which can be used to trigger Lambda functions.

The API consists of one resource called “/trigger-compare” which contains two methods “Options and Post”. Options is needed because of preflight request (CORS) and post is needed to give the data from the function to the client. Testing for this was done with postman application and a mock response from API gateway.

An API key was generated to prevent unauthorized usage of the web application.

Backend Java – Face Comparison

The program was made with Java 8 and utilized maven to manage dependencies. The program is run on an AWS Lambda instance and responds with status code 200 and a json styled body.

Documentation was done within java files in the form of Javadoc’s. Functions have comments above describing what they do and what they return.

To test the backend part of this assessment, I used the sample data provided and ran it through AWS Rekonigition. I then matched this data to the response of my java program. As far as efficiency of the program, it takes a long time seeing it needs to call comparisons individually. I have provided two source folders, one contains the version of the program that works 100% of the time, and one that utilizes threads to speed up comparisons. However, the multithreaded version throws status code 503 when used in lambda, thus the previous version is being used in the live demo.

Front End ReactJS – Trigger REST API

The front end of the web app was created in ReactJS. The web app relies on axios and bootstrap.

The front end of this assessment consists of a button to trigger an event, a spinner to display that there is something happening, and a table styled by bootstrap. When the button was clicked, it made a post request to a cors-anywhere proxy I had setup because of preflight check issues. While waiting for the response, we display a bootstrap spinner. Once the API responds with something, we hide the spinner and display our json response into a table.

The application was tested with hard coded json arrays to prevent unnecessary requests to API Gateway and Lambda functions, mostly because I went over my quota.

**Resources Used**

AWS S3 - <https://docs.aws.amazon.com/s3/index.html>

This was used to understand how S3 worked and how to access items in a bucket programmatically.

AWS Rekonigition - <https://docs.aws.amazon.com/rekognition/index.html>

This was used to understand how to make comparisons.

AWS API Gateway - <https://docs.aws.amazon.com/apigateway/index.html>

This was used to understand how to make an API and integrate with Lambda.

Stack Overflow - <https://stackoverflow.com/>

I used Stack Overflow to troubleshoot CORS errors which really didn’t help.

CORS-Anywhere - <https://github.com/Rob--W/cors-anywhere>

This was used to bypass a lot of CORS errors when attempting to contact my API. Ended up deploying a cors-anywhere server on Heroku so I didn’t have to use third parties.