

# CSCI 5410

## Serverless Data Processing

### Activity - 1

**Name:** Ashish Nagpal

**Banner ID:** B00957622

**Deployed URL:** <https://main--image-hashtags.netlify.app>

**Github URL:** <https://github.com/ashishnagpal2498/image-captioning>

## Table of Contents

<b>Aim of the task</b>	<b>3</b>
<b>Application Overview</b>	<b>3</b>
User Authentication	3
Image Upload	3
Image Captions Generation	4
Viewing the Gallery	4
Key Functionalities	4
<b>Thought Process:</b>	<b>4</b>
Development Steps	5
Technologies	5
Challenges Faced	6
<b>Screenshots:</b>	<b>6</b>
<b>References:</b>	<b>6</b>

## Aim of the task

The aim of the task is to create a full-stack web application that allows users to upload images, generate hashtags for those images using AWS Rekognition [1], and view the images with their hashtags. The application includes authentication to ensure that only registered users can upload images and view their personal galleries. The hashtags are basically labels generated which I have also referred as captions in the project.

## Application Overview

### User Authentication

The application consists of user authentication where users can log in and log out. Each user has a unique token that is used to authorize their actions within the application. User authentication ensures that only registered users can upload images and view their galleries. If a user is a new user, they can sign up to the application and the details are stored in the dynamoDB table. Otherwise, the user can login to access the application.

#### Workflow:

1. **Sign Up:** User has visited the application for the first time, thereby they need to register to the application by filling up the form. After the successful form submission a lambda function is triggered and the user details are stored in the dynamoDB table [2]. The user is redirected to the login page for credentials.
2. **Sign In:** When a user logs in, the application sends the login credentials to an authentication Lambda function.
3. **Token Generation:** The Lambda function verifies the credentials against stored user data in DynamoDB. If the credentials are correct, the Lambda function generates a JWT token[3].
4. **Token Usage:** The JWT token is returned to the client and stored. This token is then sent with subsequent requests to authorize the user.
5. **Token Validation:** For every API call that requires authentication, the server-side Lambda functions validate the token using the secret key to ensure the user is authenticated.

## Image Upload

Users can upload images through a drag-and-drop interface using a pre-signed URL of S3 bucket. Images uploaded by users are stored in S3 for further processing.

#### Workflow:

1. **Pre-signed URL:** When a user selects an image to upload, the application requests a pre-signed URL [4] from a Lambda function. The pre-signed URL allows secure, temporary access to upload a specific file to S3.
2. **Uploading Image:** The application uses this pre-signed URL to upload the image directly to the S3 bucket. The upload is done using a PUT request, which includes the user's email as an additional metadata.

- **S3 Bucket:** The S3 bucket stores the uploaded images. Each image is tagged with metadata, including the user's email extracted from the token.

## Image Captions Generation

Once the image is uploaded to the S3 bucket, an AWS Lambda function is triggered which generates the hashtags for the image using AWS Rekognition. The hashtags, along with the image URL, user email, and a unique image ID, is then stored in a DynamoDB table.

### Workflow:

1. **S3 Trigger:** When an image is uploaded to the S3 bucket, an S3 event triggers a Lambda function.
2. **Lambda Function:** The lambda function extracts the user email from object metadata and calls AWS Rekognition to generate labels.
3. **AWS Rekognition:** The image is analyzed and labels are generated for it which describes the image's content.
4. **Storing Hashtags:** The Lambda function then stores the image URL, caption, user email, and a unique image ID in a DynamoDB table. This table acts as the application's database for storing image metadata and captions.

## Viewing the Gallery

Users can view their gallery of uploaded images and their generated hashtags. The gallery fetches images and hashtags from the DynamoDB table and displays it for the logged in user.

1. **Lambda function ( Fetching and retrieval data ):** When a user navigates to the gallery, the application makes a request to a Lambda function to fetch images and hashtags from DynamoDB. The Lambda function queries the DynamoDB table using the user's email to retrieve all images and captions associated with that user.
2. **Displaying Data:** The retrieved data (image URLs and captions) is sent back to the client, which then displays the images and captions in the gallery view.

## Key Functionalities

1. **User Authentication:** Ensures that only authenticated users can upload images and view their galleries.
2. **Image Upload:** Users can upload images using a drag-and-drop interface.
3. **AWS Integration:** The application uses AWS S3 for storing images, AWS Lambda for executing code on image upload, and AWS Rekognition for generating image captions.
4. **Data Storage:** Image URLs, captions, and user information are stored in a DynamoDB table.
5. **Image Gallery:** Users can view their uploaded images along with the generated captions.

## Thought Process:

While creating the application I narrowed down the requirements and approached gradually towards the solution. First, I created the signup page using ReactJs and created the Lambda function to handle the signUp request. Following this, I created the login, token setup and then image uploads to S3.

## Development Steps

1. Set Up User Authentication:
  - I created Lambda functions for user registration and login and setup function URL with CORS of lambda functions.
  - The JWT tokens were generated upon successful login and validated these tokens for subsequent requests.
  - The jwt package is not available in the Lambda function, thereby I created a lambda layer for it.
2. Implement Image Upload:
  - Developed a Lambda function to generate pre-signed URLs for secure image uploads to S3.
  - Created a React component for image upload using react-dropzone and Axios to upload images using the pre-signed URL.
  - Enable CORS on S3 bucket to accept data from pre-signed S3 URL
3. Display Image Gallery:
  - Developed a Lambda function to fetch images and captions from DynamoDB.
  - Created a React component to display the fetched images and captions in a gallery format.
4. Generate Image Captions:
  - Set up an S3 trigger to invoke a Lambda function whenever a new image is uploaded.
  - Integrated AWS Rekognition in the Lambda function to analyze images and generate captions.
  - Stored image metadata, captions, and user information in DynamoDB.

## Technologies

1. **Authentication:** I choose JSON Web Tokens (JWT) for token-based authentication due to its simplicity and security. AWS Lambda was used to handle authentication logic.
2. **Image Storage:** AWS S3 was selected for storing images because of its scalability, security, and integration with other AWS services.
3. **Image Recognition:** AWS Rekognition was chosen for its powerful image analysis capabilities which works on APIs without any required knowledge of machine learning.
4. **Data Storage:** AWS DynamoDB was selected for storing image metadata and hashtags due to its flexibility and scalability.

## Challenges Faced

1. **CORS error** - While developing the lambda functions, I faced Cross-Origin Resource Sharing (CORS) issue. CORS is basically a security feature implemented by web browsers to restrict how resources on a web page can be requested from another domain. After following the documentation[], I enabled CORS, however the issue persisted. The error which I got looked like

```
=====
https://wp3b7oru7quatr53oo4vj6h4sy0vwmkq.lambda-url.us-east-1.on.aws/
from origin 'http://localhost:3000' has been blocked by CORS policy:
Response to preflight request doesn't pass access control check: No
'Access-Control-Allow-Origin' header is present on the requested resource.
=====
```

Following this, I searched and found out that when adding the token in the headers for authentication, the header for that should also be added. Upon looking at the screenshot, in the medium article[6] I understood the error and corrected it.

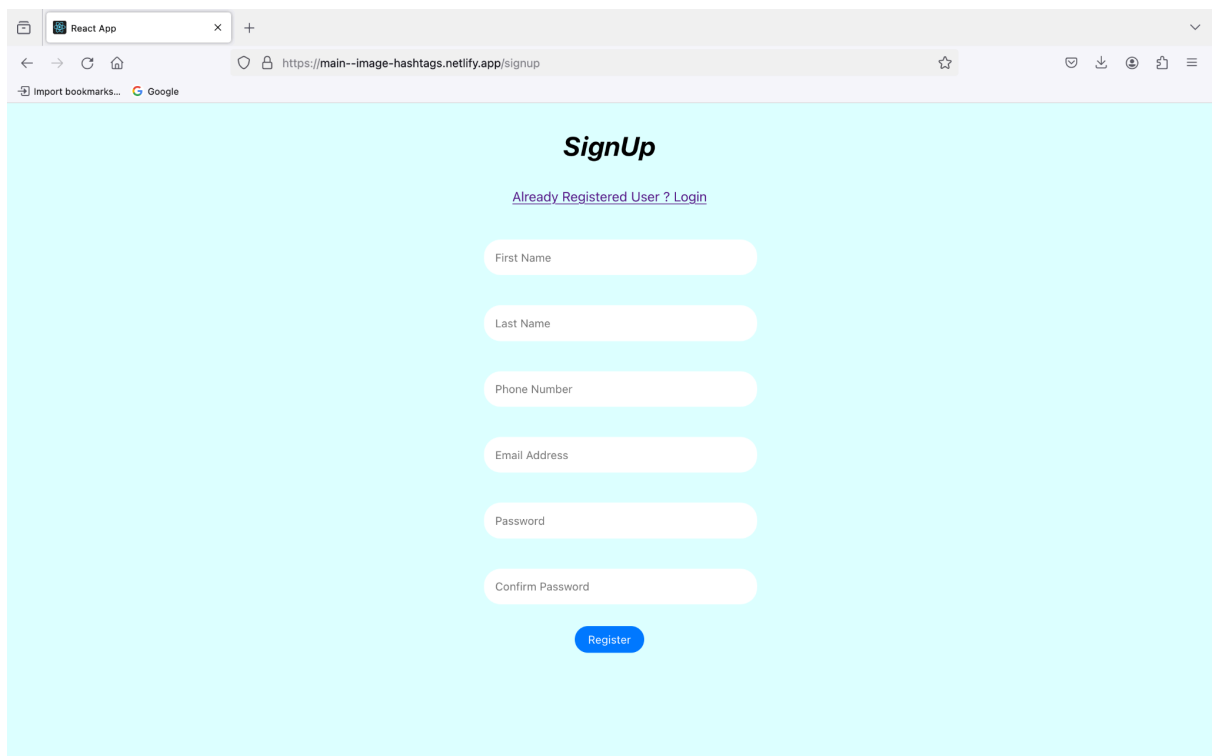
2. **Pre-signed URL upload** - After generating the pre-signed URL, when I tried to upload an image to the bucket using PUT method, I was getting the 403 forbidden error. Next, to test the same, I tried to use the Postman and got the following error -

```
=====
<Error> <Code>SignatureDoesNotMatch</Code> <Message>The request
signature we calculated does not match the signature you provided. Check
your key and signing method.</Message>
<AWSAccessKeyId>fdvfddfdf</AWSAccessKeyId> <StringToSign>PUT
=====
```

Upon following the approach mentioned in the medium article[7], I found out that the ContentType parameter was missed while creating the presigned URL due to which the error was caused.

## Screenshots:

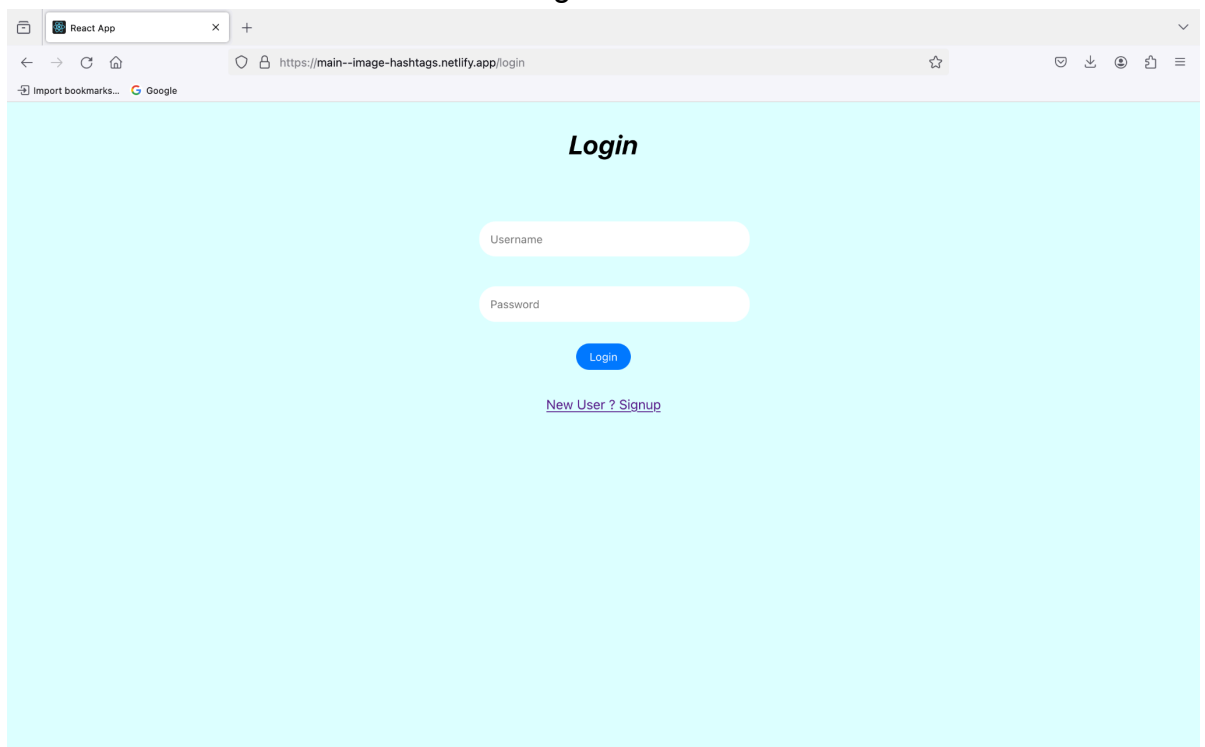
### 1. SignUp



A screenshot of a web browser displaying the 'SignUp' page. The browser's address bar shows the URL 'https://main--image-hashtags.netlify.app/signup'. The page has a light blue background. At the top, the title 'SignUp' is centered in a bold, black font. Below the title, there is a link '[Already Registered User ? Login](#)'. The form consists of six white input fields with rounded corners, stacked vertically: 'First Name', 'Last Name', 'Phone Number', 'Email Address', 'Password', and 'Confirm Password'. At the bottom of the form is a blue button with the text 'Register' in white.

Figure 1: Sign Up

### 2. Login



A screenshot of a web browser displaying the 'Login' page. The browser's address bar shows the URL 'https://main--image-hashtags.netlify.app/login'. The page has a light blue background. At the top, the title 'Login' is centered in a bold, black font. Below the title, there are two white input fields with rounded corners: 'Username' and 'Password'. At the bottom of the form is a blue button with the text 'Login' in white. Below the button is a link '[New User ? Signup](#)'.

Figure 2: Login

### 3. Dashboard

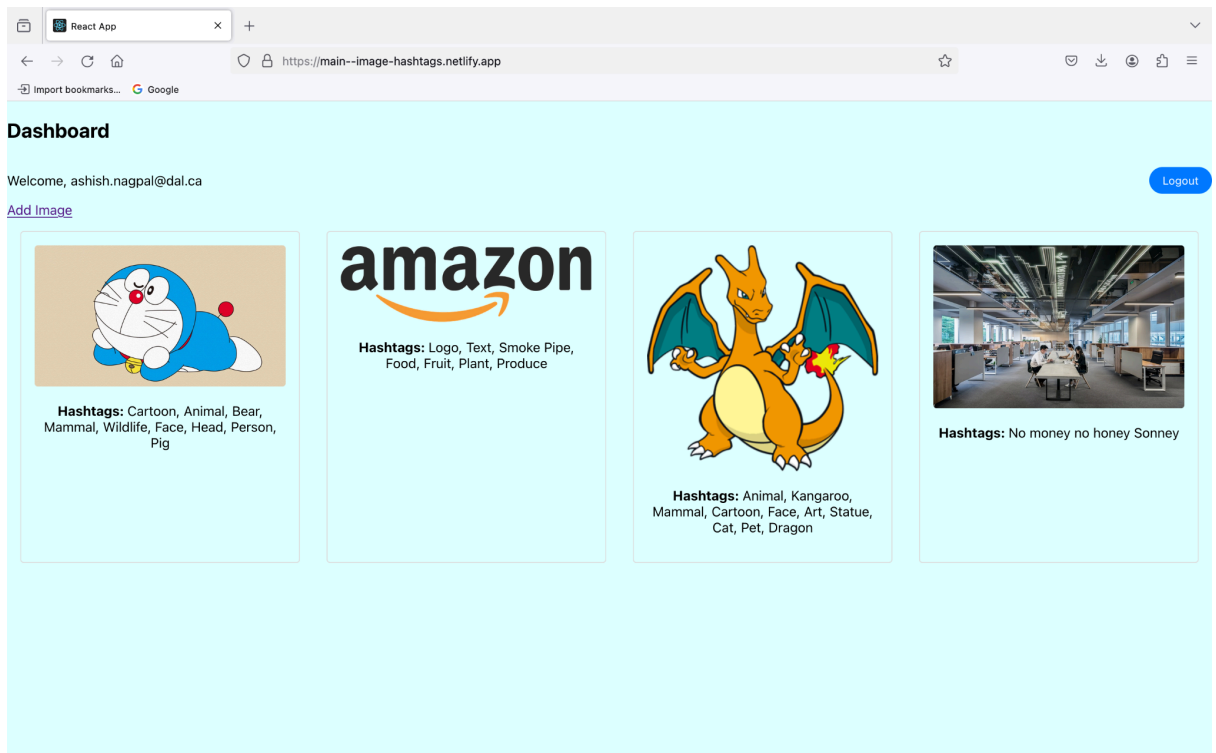


Figure 3: Dashboard

### 4. Upload Image

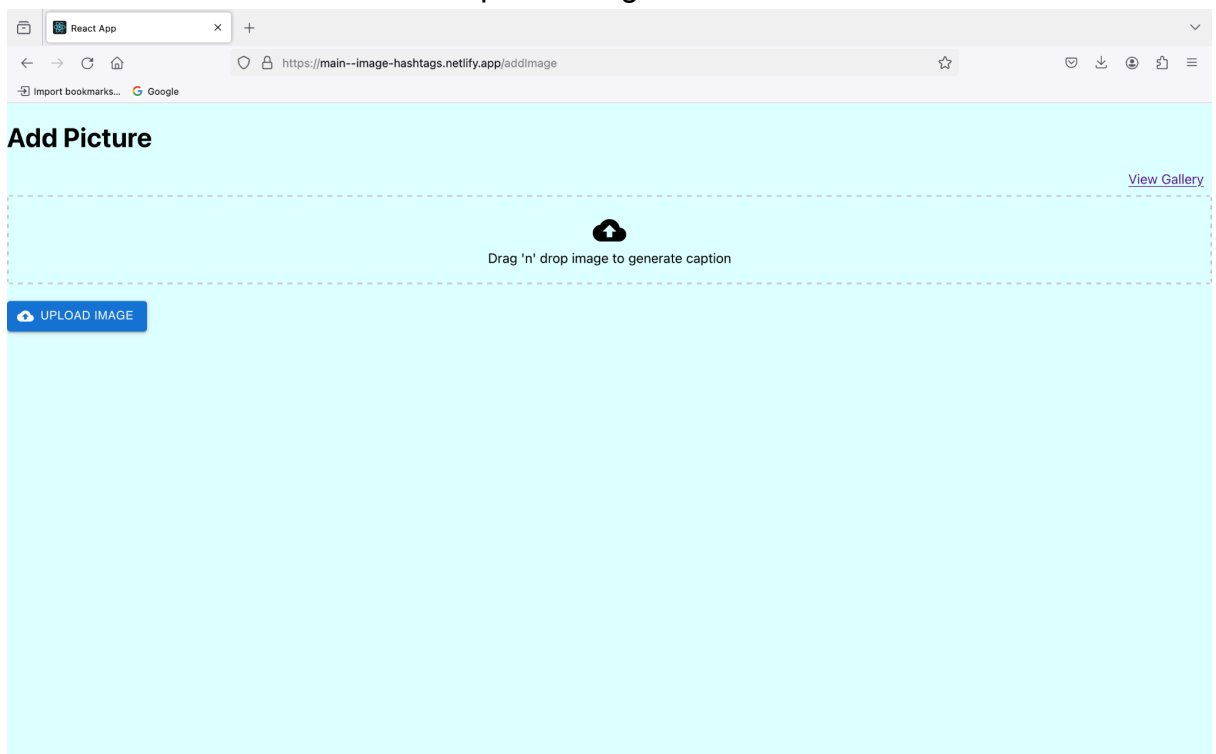


Figure 4: Upload Image



## References:

- [1] "Detecting labels in an image," *Amazon*. [Online]. Available: <https://docs.aws.amazon.com/rekognition/latest/dg/labels-detect-labels-image.html> [Accessed: June 03, 2024].
- [2] "Tutorial: Build a CRUD API with Lambda and DynamoDB," *Amazon*. [Online]. Available: <https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-dynamodb.html> [Accessed: June 03, 2024].
- [3] F. P. Code, "How to generate jwt token using Python," *Medium*, September 09, 2023. [Online]. Available: <https://medium.com/@amr2018/how-to-generate-jwt-token-using-python-36c2305c5a14> [Accessed: June 03, 2024].
- [4] "Working with presigned URLs," *Amazon*. [Online]. Available: <https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-presigned-url.html> [Accessed: June 03, 2024].
- [5] "Working with presigned URLs," *Amazon*. [Online]. Available: <https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-presigned-url.html> [Accessed: June 03, 2024].
- [6] D. Dabare, "Fixing Cors issue on Lambda functions," *Medium*, May 31, 2023. [Online]. Available: <https://devakadabare.medium.com/fixing-cors-issue-on-lambda-functions-b228c34d5966> [Accessed: June 03, 2024].
- [7] B. Poudyal, "Upload files directly to S3 with pre-signed URL," *Medium*. Oct 12, 2021. [Online]. Available: <https://medium.com/weekly-webtips/upload-files-directly-to-s3-with-pre-signed-url-31beff41157e> [Accessed: June 03, 2024].