PROJECT DOCUMENTATION - GROUP 9

Group Members (id - 9) :-

Siddhesh More (su4822)
Atharva Jadhav (hz6667)
Vedashree Bhat (dl4176)

A. Documentation - Intro Page :

Application Name : FaceInsight

URL of Deployed Application: https://frontend-dot-pinsight.wm.r.appspot.com

Application Purpose :

FaceInsight is an innovative web application that processes user photos from **Pinterest** using the Google Vision API. It provides detailed emotional insights, identifies patterns in uploaded images, and organizes them into well-structured albums. The application offers users a seamless way to explore and relive their memories through meaningful emotional and contextual summaries.

Key Functionalities Include:

1. **Emotional Analysis**: Analyze emotions like happiness, sadness, anger, and surprise by detecting faces across multiple photos.
2. **Pattern Recognition and Organization**: Recognize patterns in uploaded images and segregate them into clearly categorized albums for better organization.
3. **Object and Scene Categorization**: Identify objects (e.g., pets, buildings, nature) and scenes in photos, grouping them contextually.
4. **Photo Sentiment Report**: Generate a comprehensive summary of users' photos, capturing emotional trends across events or timeframes.

FaceInsight focuses on providing a straightforward yet insightful way to organize and analyze Pinterest photo collections, offering users an enriched experience of their photo memories.

B. Documentation - Description Page :

Frontend :

1. App.js - https://github.com/siddheshmore12/Project2/blob/main/pinsight/src/App.js

- **Purpose**: Acts as the entry point for the React application, setting up the routing structure.
- **Functionality**:
  - Uses React Router to define navigation paths for various components such as ***Login, Dashboard,*** and ***SentimentAnalysis***.
  - Handles Pinterest OAuth callback redirection and routes for additional features like viewing albums and performing sentiment analysis on a specific Pinterest pin.
  - Ensures smooth user experience by dynamically rendering components based on the URL path.

2. Login.js - https://github.com/siddheshmore12/Project2/blob/main/pinsight/src/components/Login.js

- **Purpose**: Provides the login interface for the application.
- **Functionality**:
  - Displays a login page styled with Bootstrap, allowing users to log in using their Pinterest account.
  - Integrates the Pinterest icon for a user-friendly interface.
  - Redirects users to the backend's Pinterest OAuth endpoint for authentication.
  - Enhances aesthetics by using a background image and responsive design.

3. LoginCallback.js - https://github.com/siddheshmore12/Project2/blob/main/pinsight/src/components/LoginCallback.js

- **Purpose**: Handles the Pinterest OAuth callback after user authentication.
- **Functionality**:
  - Extracts the ***accessToken*** from the URL query parameters.
  - Securely stores the token in ***localStorage*** for use in subsequent API calls.
  - Redirects authenticated users to the ***SentimentAnalysis*** page or alerts them in case of errors during login.
  - Simplifies user flow by managing the OAuth token seamlessly in the background.

4. Dashboard.js - https://github.com/siddheshmore12/Project2/blob/main/pinsight/src/components/Dashboard.js

- **Purpose**: Serves as the main page for displaying the user's Pinterest pins.
- **Functionality**:
    - Fetches pins from the backend using the accessToken for authentication.
    - Displays pins in a grid layout, including their title, description, link, and image.
    - Allows users to navigate to specific features such as Sentiment Analysis or viewing albums.
    - Handles loading states, error messages, and token validation to ensure a robust user experience.
    - Includes a button for navigating to the album view.

5. SentimentAnalysis.js - https://github.com/siddheshmore12/Project2/blob/main/pinsight/src/components/SentimentAnalysis.js

- **Purpose**: Performs and displays sentiment analysis results for a specific Pinterest pin.
- **Functionality**:
    - Sends the ***pinId*** and ***accessToken*** to the backend to fetch sentiment data.
    - Displays results in a visually appealing format using Bootstrap ***ProgressBar*** components for emotions like joy, anger, sorrow, and surprise.
    - Generates a downloadable PDF report of the sentiment analysis using the ***jsPDF*** library.
    - Handles loading states, errors, and missing data gracefully, ensuring usability.

6. Album.js - https://github.com/siddheshmore12/Project2/blob/main/pinsight/src/components/Album.js

- **Purpose**: Provides an interface for users to view albums created from their Pinterest pins.
- **Functionality**:
    - Fetches album data from the backend, using ***accessToken*** for secure access.
    - Displays a collection of user-created albums and provides navigation options for exploring album contents.
    - Serves as an extension of the ***Dashboard***, allowing better organization of pins.

Backend :

1. firebase-setup.js - https://github.com/siddheshmore12/Project2/blob/main/backend/firebase-setup.js

This file handles the setup of Firebase in your application. It imports the Firebase Admin SDK and initializes it with credentials from a service account JSON file. The Firebase Admin SDK allows the application to interact with Firebase services, particularly Firestore (a NoSQL

database). The Firestore instance is exported from this file, enabling other parts of the application to read from and write to the Firestore database.

Main Function: The main purpose of this file is to initialize and configure Firebase services, particularly Firestore, using the provided service account credentials. It makes the Firestore database accessible to other modules in the application by exporting the ***db*** instance.

2. server.js - https://github.com/siddheshmore12/Project2/blob/main/backend/server.js

This file sets up the backend server using Express.js. Key functionalities include:

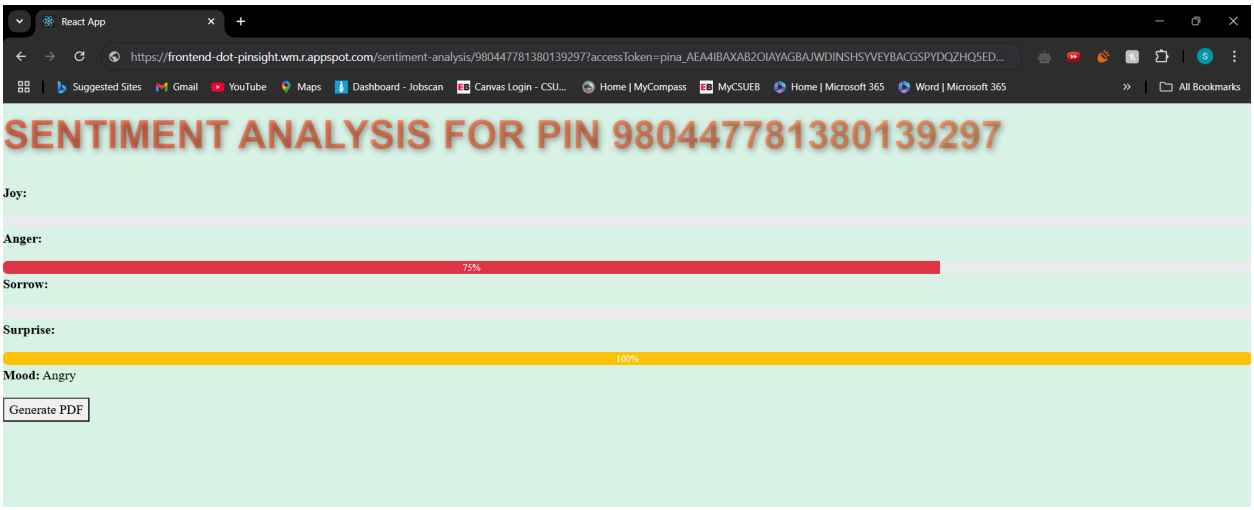
- **OAuth Integration with Pinterest**: It redirects users to Pinterest's OAuth for authentication, exchanges the authorization code for an access token, and fetches user profile and pins.
- **Sentiment Analysis**: The ***/sentiment-analysis*** endpoint fetches pin details and uses the Google Vision API to analyze images and detect emotions like joy, anger, sorrow, etc.
- **Pin Categorization**: The ***/fetch-albums*** endpoint fetches pins and categorizes them based on labels detected by the Google Vision API.
- **Pin Data Management**: The server stores Pinterest user profiles and pin data into Firebase Firestore, manages token-based authentication, and handles various endpoints for pin retrieval and analysis.

Main Function: The main purpose of this file is to implement the backend functionality of the application. It handles user authentication with Pinterest, performs sentiment analysis on images, and allows users to fetch and categorize Pinterest pins based on visual content using Google Cloud Vision APIs. The file also integrates Firebase Firestore for persistent data storage (such as user profiles and pins).

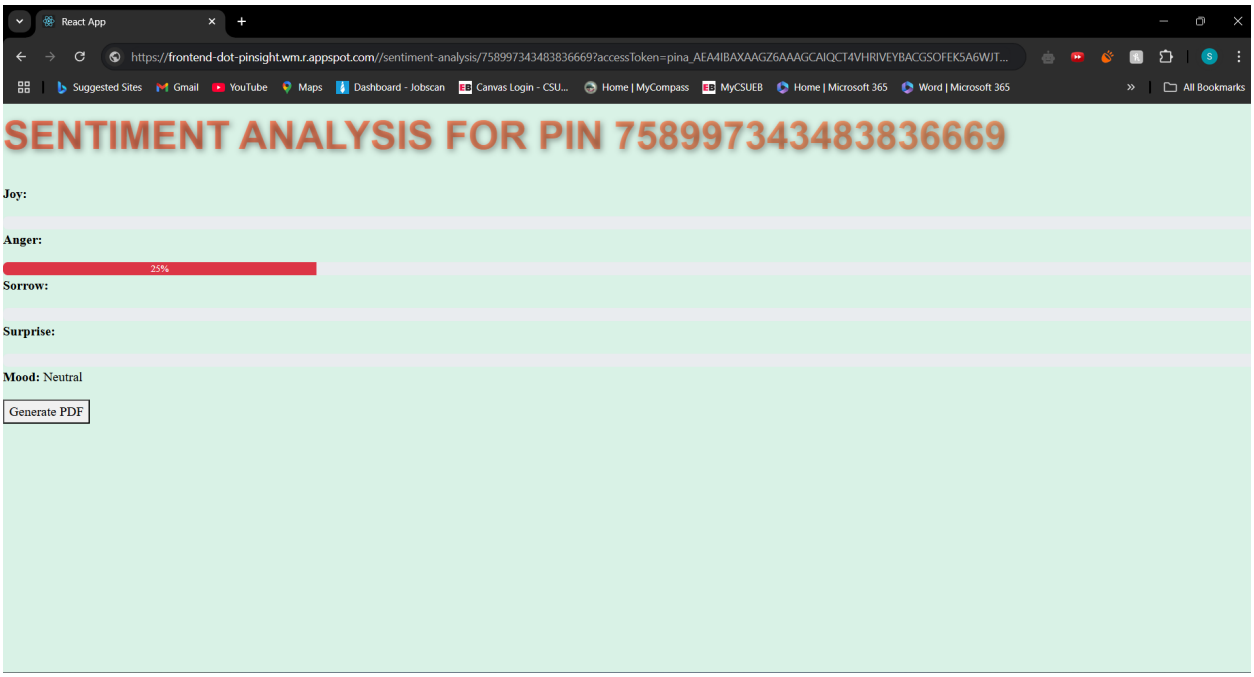
C. Documentation - Demonstration of Application working Page :

1. Screenshots showing results for users :

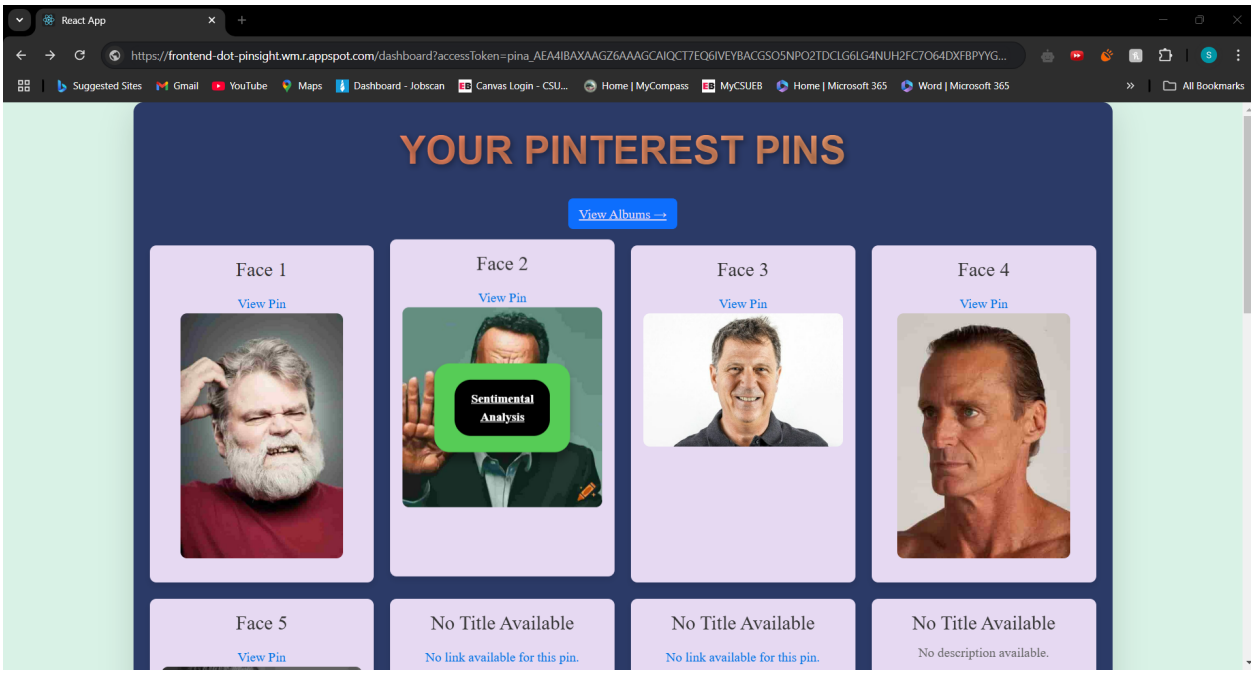
User 1 with ID - 980447781380139397 -:



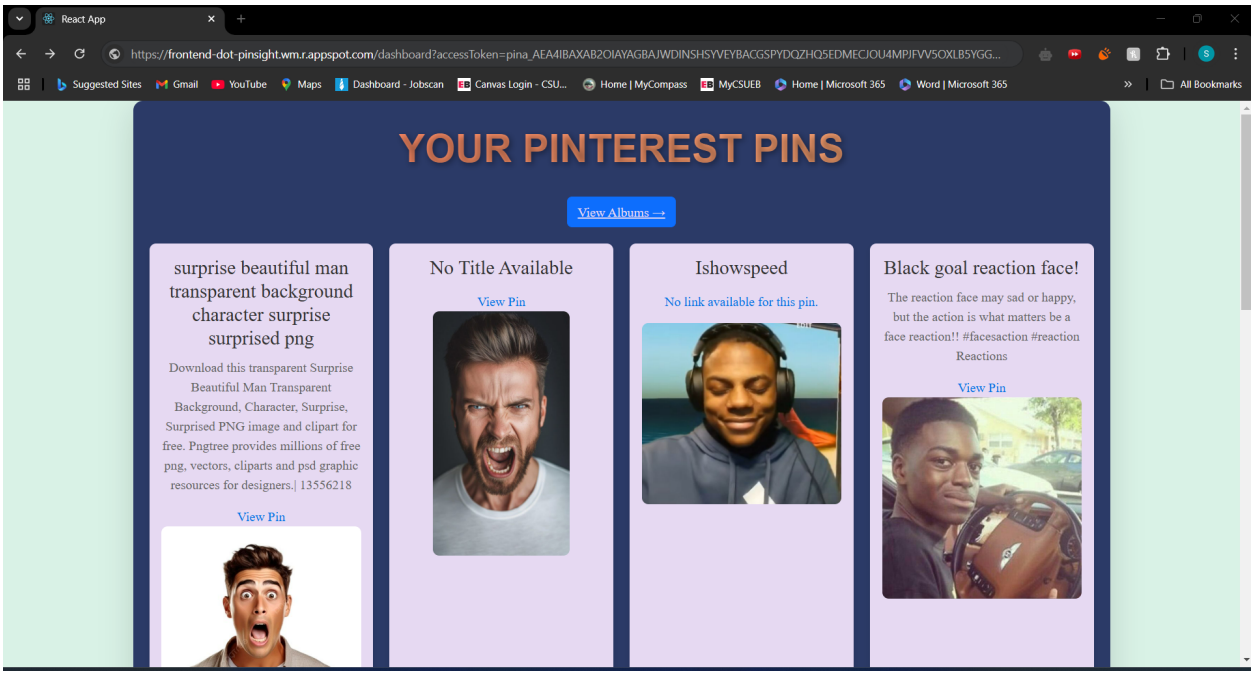
User 2 with ID - 758997343483836669 -:



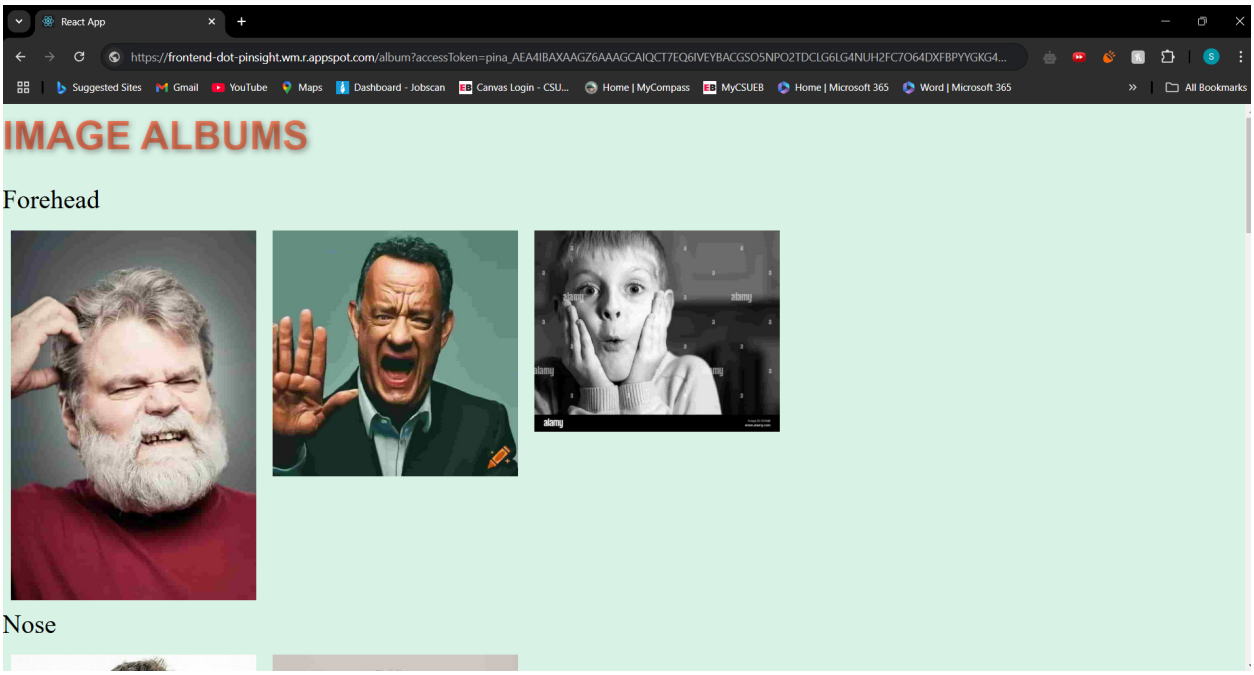
Dashboard User 1 :-



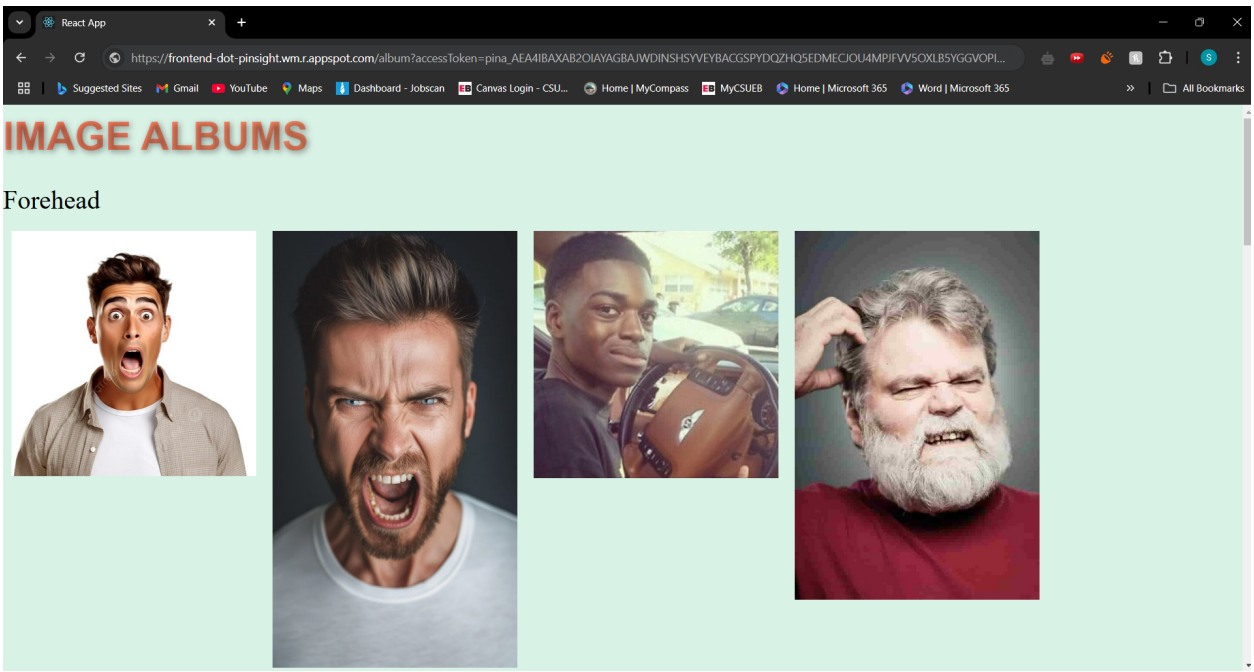
Dashboard User 2 :-



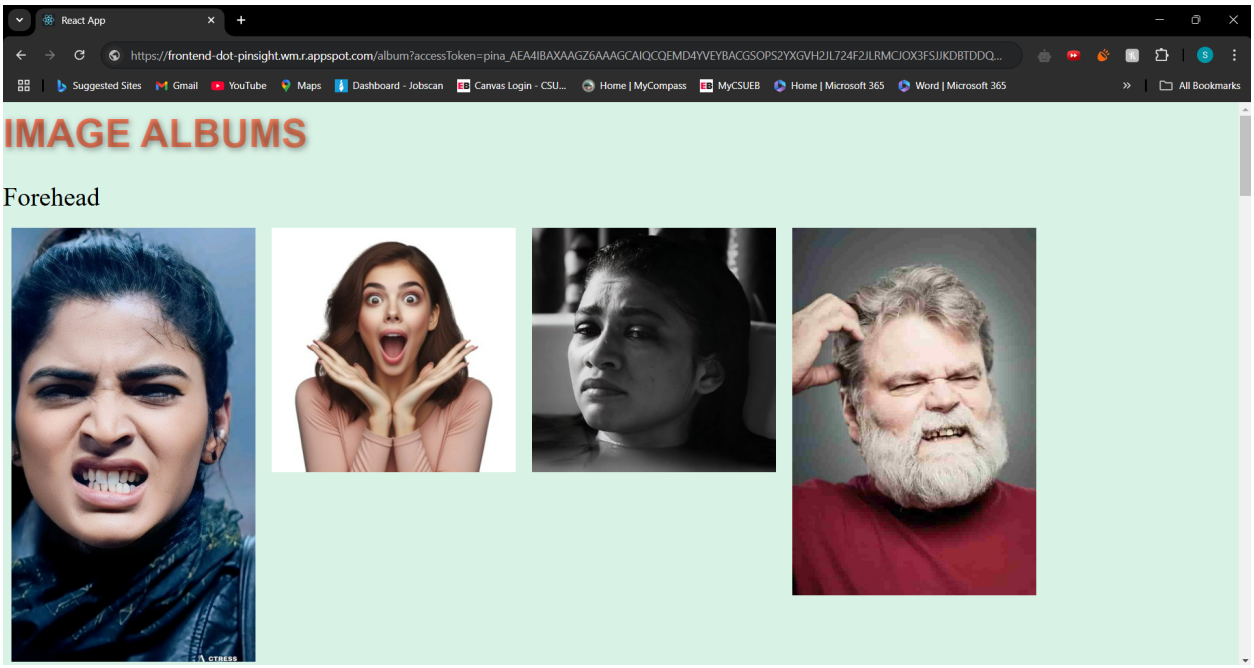
Album User 1 :-



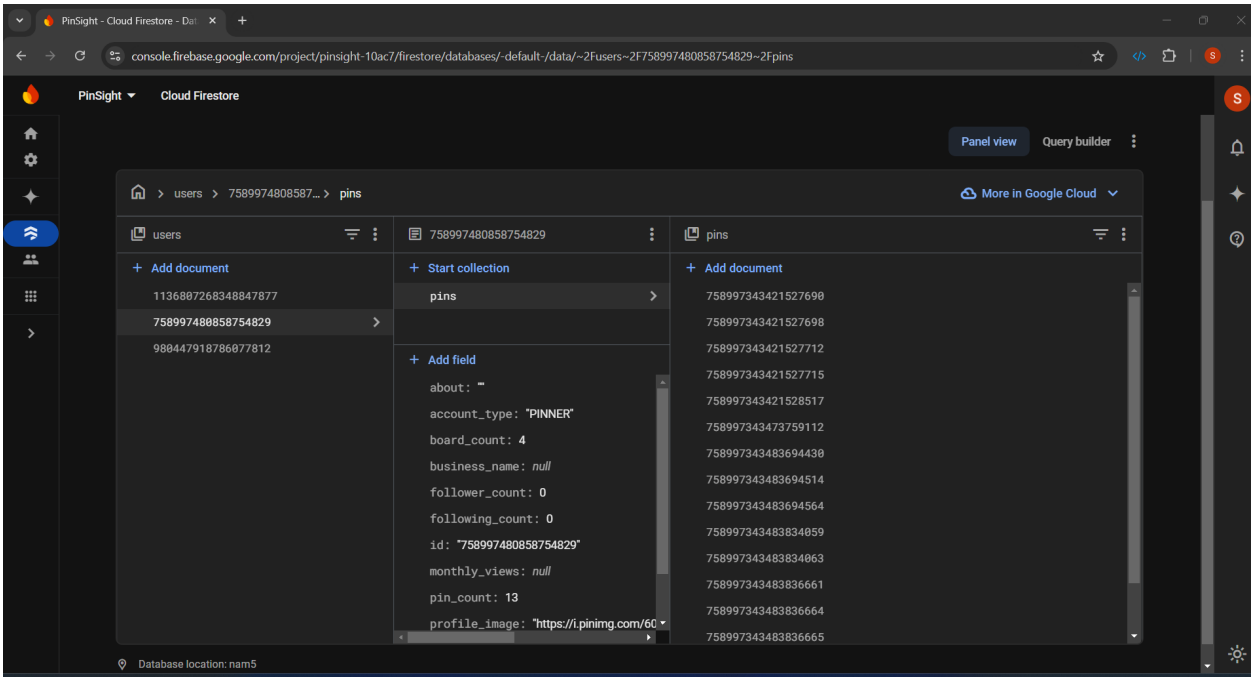
Album User 2 :-



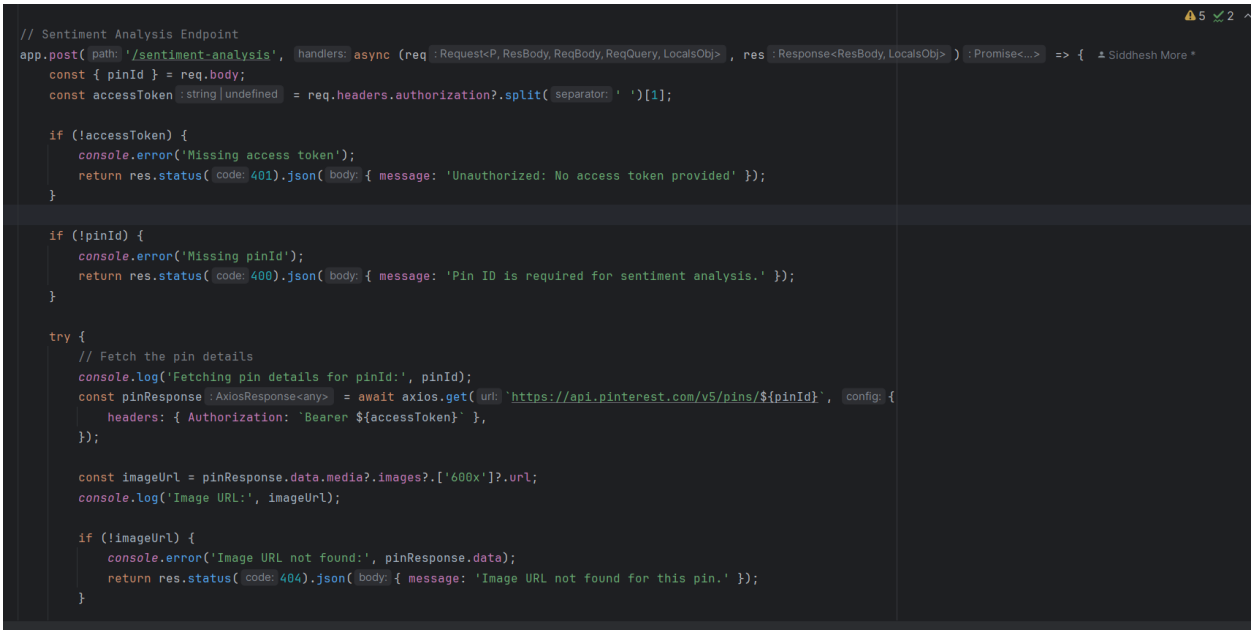
2. Screenshot after adding a new image to the pinterest account and the resulting change in output when running the Application :



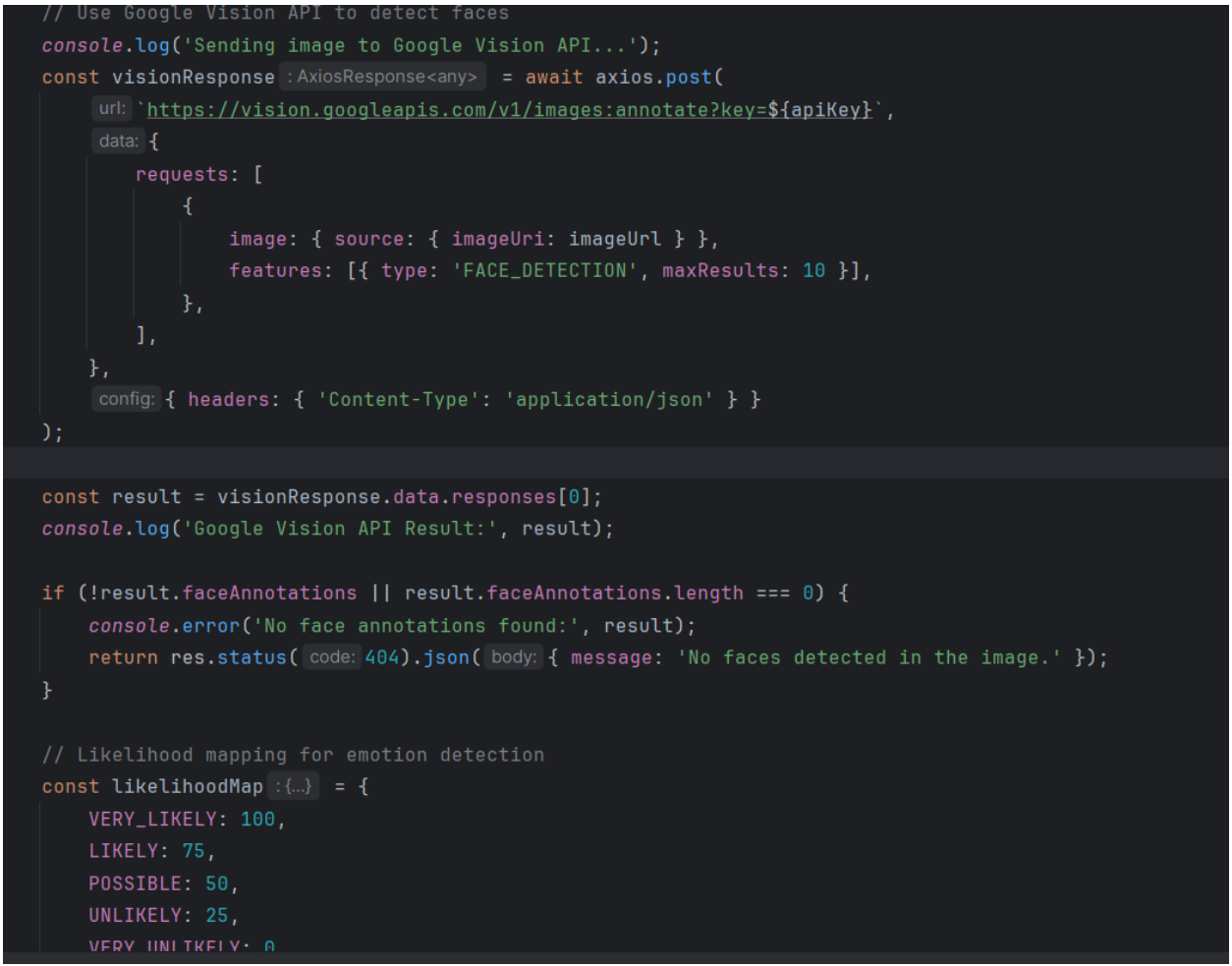
3. Screenshots of the data stored in "Google Database" :



4. Screenshots showing how the application uses the Google Vision API and what calls are made and example of returned results :



```
// Sentiment Analysis Endpoint
app.post( path: '/sentiment-analysis', handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : Promise<...> => {  Siddhesh More *
    const { pinId } = req.body;
    const accessToken : string | undefined  = req.headers.authorization?.split( separator: ' ')[1];

    if (!accessToken) {
        console.error('Missing access token');
        return res.status( code: 401).json( body: { message: 'Unauthorized: No access token provided' });
    }

    if (!pinId) {
        console.error('Missing pinId');
        return res.status( code: 400).json( body: { message: 'Pin ID is required for sentiment analysis.' });
    }

    try {
        // Fetch the pin details
        console.log('Fetching pin details for pinId:', pinId);
        const pinResponse : AxiosResponse<any>  = await axios.get( url: `https://api.pinterest.com/v5/pins/${pinId}`, config: {
            headers: { Authorization: `Bearer ${accessToken}` },
        });

        const imageUrl = pinResponse.data.media?.images?.['600x']?.url;
        console.log('Image URL:', imageUrl);

        if (!imageUrl) {
            console.error('Image URL not found:', pinResponse.data);
            return res.status( code: 404).json( body: { message: 'Image URL not found for this pin.' });
        }
```



```
// Use Google Vision API to detect faces
console.log('Sending image to Google Vision API...');
const visionResponse : AxiosResponse<any>  = await axios.post(
    url: `https://vision.googleapis.com/v1/images:annotate?key=${apiKey}`,
    data: {
        requests: [
            {
                image: { source: { imageUri: imageUrl } },
                features: [{ type: 'FACE_DETECTION', maxResults: 10 }],
            },
        ],
    },
    config: { headers: { 'Content-Type': 'application/json' } }
);

const result = visionResponse.data.responses[0];
console.log('Google Vision API Result:', result);

if (!result.faceAnnotations || result.faceAnnotations.length === 0) {
    console.error('No face annotations found:', result);
    return res.status( code: 404).json( body: { message: 'No faces detected in the image.' });
}

// Likelihood mapping for emotion detection
const likelihoodMap : {...}  = {
    VERY_LIKELY: 100,
    LIKELY: 75,
    POSSIBLE: 50,
    UNLIKELY: 25,
    VERY_UNLIKELY: 0
```

```
app.post( path: '/sentiment-analysis', handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> )
        const likelihoodMap : {...} = {
            VERY_LIKELY: 100,
            LIKELY: 75,
            POSSIBLE: 50,
            UNLIKELY: 25,
            VERY_UNLIKELY: 0,
        };

        // Parse the face data
        const analysis : ({...})[] = result.faceAnnotations.map((face) : {...} => ({
            joy: likelihoodMap[face.joyLikelihood] || 0,
            anger: likelihoodMap[face.angerLikelihood] || 0,
            sorrow: likelihoodMap[face.sorrowLikelihood] || 0,
            surprise: likelihoodMap[face.surpriseLikelihood] || 0,
            blurred: likelihoodMap[face.blurredLikelihood] || 0,
            underExposed: likelihoodMap[face.underExposedLikelihood] || 0,
            mood: getMood(
                likelihoodMap[face.joyLikelihood],
                likelihoodMap[face.angerLikelihood],
                likelihoodMap[face.sorrowLikelihood],
                likelihoodMap[face.surpriseLikelihood]
            ),
        }));

        console.log('Sentiment Analysis Result:', analysis);
        return res.status( code: 200).json(analysis);
    } catch (error) {
        console.error('Error performing sentiment analysis:', error.message);
        res.status( code: 500).json( body: { message: 'Failed to perform sentiment analysis.', error: error.message });
```

```
// Step 2: Categorize Pins using Google Vision API
const categorizedPins : {} = {};

for (const pin of pins) {
    if (pin.media?.images?.['600x']?.url) {
        try {
            // Analyze the pin's image
            const [result : IAnnotateImageResponse ] = await client.labelDetection(pin.media.images['600x'].url);
            const labels : string[] = result.labelAnnotations.map(label : IEntityAnnotation => label.description);

            // Use the first label as the category
            const primaryLabel : string = labels[0] || 'Uncategorized';

            // Group pins by category
            if (!categorizedPins[primaryLabel]) {
                categorizedPins[primaryLabel] = [];
            }
            categorizedPins[primaryLabel].push(pin);
        } catch (visionError) {
            console.error('Error analyzing pin image:', visionError.message);
        }
    }
}
```

D. Documentation - Google Database Page :

Part 1: Setup of Firestore and Structure of Stored Data

Google Firestore Setup:

In this project, we use **Google Firestore**, a flexible, scalable NoSQL database designed for storing, syncing, and querying data in real-time. Firestore is part of Firebase, which simplifies back-end development for mobile and web apps.

Firestore Initialization:

- The project initializes Firebase Admin SDK by using the service account credentials from the ***pinsight-10ac7-firebase-adminsdk-jt8ed-f6e2a3a011.json*** file. This file contains the necessary credentials to authenticate and connect to Firebase.
- The ***admin.initializeApp()*** method is used to set up the Firebase Admin SDK, which allows for server-side interactions with Firestore, such as reading and writing data.

```
const admin = require('firebase-admin');
const serviceAccount = require('./pinsight-10ac7-firebase-adminsdk-jt8ed-f6e2a3a011.json');

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
});

const db = admin.firestore();
```

Firestore Data Structure: The data is organized into **collections** and **documents**, which are the core units of storage in Firestore.

- The root collection is users, where each document within the collection represents a user, identified by a unique userId.
- Inside each user document, we store data such as:
  - ***account_type:*** The type of the user's account (e.g., "BUSINESS").
  - ***business_name:*** The name of the business associated with the account.
  - ***profile_image:*** URL of the user's profile image.
  - ***pin_count:*** The number of pins the user has posted.
  - ***follower_count and following_count:*** Metrics for the user's engagement on the platform.

An example document for a user is shown below:

```
users (collection)
  └── 1136807268348847877 (document)
      ├── account_type: "BUSINESS"
      ├── business_name: "PinSight"
      ├── follower_count: 0
      ├── pin_count: 0
      ├── profile_image: "https://s.pinimg.com/images/user/default_600.png"
      └── username: "smore50088"
```

In the ***users*** collection, the document ID is the ***userId***, which uniquely identifies a user. Each user document can have subcollections like ***photos***, where additional data (such as images or metadata) related to the user can be stored.

Part 2: Adding, Updating, and Removing Data in the App

Adding Data: The app allows the addition of data to Firestore through API calls. The primary functionality is shown in the **/save-data** endpoint, which saves user-specific data such as images.

```
app.post('/save-data', async (req, res) => {
  const { userId, imageData } = req.body;

  if (!userId || !imageData) {
    return res.status(400).json({ message: 'Missing required fields: userId or imageData' });
  }

  try {
    const userRef = db.collection('users').doc(userId);
    await userRef.collection('photos').add(imageData);

    res.status(200).json({ message: 'Data saved successfully' });
  } catch (error) {
```

```
    console.error('Error saving data to Firestore:', error.message);
    res.status(500).json({ message: 'Internal Server Error', error: error.message });
  }
});
```

Explanation:

- The ***/save-data*** endpoint receives a ***userId*** and ***imageData*** in the request body.
- The app checks if the required fields (***userId*** and ***imageData***) are present. If not, a 400 error is returned with an appropriate message.
- If the data is valid, the code accesses the Firestore database, retrieves the document corresponding to the user ***(userId)***, and then adds the ***imageData*** to the ***photos*** subcollection of that user.
- If the operation is successful, a success message is returned. In case of an error, an error message is returned.

Updating Data: While the code provided does not include explicit update operations, Firestore allows updating documents using the ***update()*** method. For example, to update the ***profile_image*** field for a user, the code would look like this:

```
const userRef = db.collection('users').doc(userId);
await userRef.update({ profile_image: 'new-image-url' });
```

This updates the ***profile_image*** field for the specific user identified by ***userId***.

Removing Data: To remove data, Firestore provides the ***delete()*** method. For instance, to delete a specific photo from a user's ***photos*** subcollection, the code could be:

```
const photoRef = db.collection('users').doc(userId).collection('photos').doc(photoId); await photoRef.delete();
```

This would delete the document identified by ***photoId*** within the ***photos*** subcollection of the specific user.

Queries: Firestore allows querying for documents based on various criteria. For example:

- To fetch all photos of a specific user, the app could query the ***photos*** subcollection like this:

```
const photosSnapshot = await db.collection(‘users').doc(userId).collection('photos').get();
```

To filter pins by a certain condition, such as pins with a specific label or emotion, Firestore supports powerful query filters like:

const query = db.collection('pins').where('emotion', '==', 'joy'); const snapshot = await query.get();
This would retrieve all pins where the detected emotion is ‘joy’.

E. Documentation - Code Styling and Comments :

In the development of this project, we have adhered to consistent ***CamelCase*** styling across all variable names, function names, and other identifiers. This approach ensures readability and aligns with common JavaScript conventions, making the code easy to follow and maintain. Additionally, we have made sure to apply this styling to all parts of the codebase, from the initialization of variables to the naming of methods and object properties. Furthermore, we understand the importance of clarity in graduate-level work, which is why we have ensured that the entire code is ***fully commented.*** Every function, loop, and significant code block has been thoroughly explained to provide clear context for each operation. This documentation aims to ensure that anyone reviewing the code can quickly understand its purpose and flow.

F. Documentation -Code Page (main GitHub link):

— https://github.com/siddheshmore12/Project2/

G. Documentation - YouTube Video Page

— https://youtu.be/0B6HROwuWeg