COMPLETE EVOLUTION PROCESS OF OUR PROJECT

Project Overview

We began by creating a **flowchart** to better understand the structure and workflow of our project. The system is divided into **two main interfaces**: one for **children** and one for **administrators (admins)**. The four primary components of the system are:

1. **Frontend**: Built with **React.js**.
2. **Middleware**: Consisting of **Express.js** and **Node.js**.
3. **Deep Learning Model**: Used for expression tracking and analysis.
4. **Database**: Managed with **MongoDB**.

System Workflow

- **Child Interface**:
    - The child interacts with the system by playing a game.
    - During gameplay, **images** are captured and stored on the server.
    - These images are then saved into the **MongoDB database** for future analysis.
- **Admin Interface**:
    - The admin can click on the **"Get Analysis"** button, which triggers an API call to retrieve the images stored in the database.
    - The images are sent to the **deep learning model** for analysis.
    - The analysis results are returned to the server, stored in the database, and displayed on the **admin dashboard**.
    - This structure ensures that the child's gameplay is smooth and uninterrupted while providing the admin with detailed, real-time analysis.

Game Development Process

- Initially, the game was created in **HTML**. However, as our project was built with **React**, we decided to **transition to React** early in the development process to avoid complications later on.
- **Game Ideas**:
    - We brainstormed various game concepts, such as **puzzles**, **scrambled letters**, **memory games**, and **word matching**. However, the primary focus of the project was **expression tracking**, so we decided to build a simple quiz game inspired from
    - Game Link.
- **Game Design for Dyslexic Children**:

    - The quiz game was designed to help improve **spelling** for dyslexic children.
    - To capture the child's emotional reactions, we introduced a **background color change** (from **red** to **green**) based on the correctness of their answers.
    - Additional interactive features, such as **confetti effects** and a **score display** at the end of the game, were added to keep the child engaged and to capture more expressive reactions for analysis.

Collaboration and Version Control

- To facilitate collaboration and version control, we set up **Git accounts** and created a **repository**. This allowed each team member to make changes and add new features efficiently.
    - We learned how to use Git commands such as **push**, **pull**, **commit**, **delete**, and **clone**, which streamlined the development process.

To handle image capturing for our project, I created a backend folder to save the captured images. Here's how the process evolved:

1. **Initial Attempt with Local Storage:**
   Before implementing a server-side solution, I experimented with storing the captured images in **local storage**. The idea was to check if the images were being captured correctly and to display them at the end of the game for verification. This was a temporary solution as I was not yet familiar with server-side storage or APIs.
2. **Current Implementation:**
   Now, we are capturing images every **30 seconds** during gameplay and saving them in the uploads folder located in the backend. The image-capturing logic is written directly in the **game component** of the React application.

This step-by-step process helped me transition from using local storage for basic testing to implementing a more robust server-side solution for image storage and retrieval.

Session ID Generation for easier tracking

To ensure proper tracking for each game session, we implemented the following process:

1. **Session ID Generation:**
   Every time the child clicks "Start Game," a unique **Session ID** is generated. This ID is crucial for associating each gameplay session with its respective data.
2. **Storing in MongoDB:**
   Using **MongoDB** and **Atlas**, we stored the Session ID along with the **image path** for the captured images. Instead of saving the images directly in the database (since image files are large and can affect database performance), we opted to store only the paths to the uploaded images saved in the backend uploads folder.

3. **Transition to MongoDB:**
   This was the first time we began using MongoDB and Atlas for managing our backend database, allowing us to structure and retrieve session-related data efficiently.

After creating the backend folder to save the webcam images captured during gameplay and completing the image capture functionality, we decided to move on to the facial expression detection model.

Adding Database Connection and Environmental Configuration

Database Connection Setup:

After establishing the connection with **MongoDB**, we created two essential files:

1. **dbconnection.js**: Responsible for setting up the connection to the MongoDB database.
2. **schema.js**: Defines the schema for the data we store, such as the **Session ID**, **image path**, and **model responses**.

Environmental Variables:

To securely manage sensitive data, such as the **MongoDB URL** and the **Hugging Face API token**, we introduced the use of environmental variables.

- **.env**: This file stores sensitive data and credentials like the MongoDB connection URL and the Hugging Face token.
- **.gitignore**: To prevent exposing sensitive files, we added **.env** and other unnecessary files to the .gitignore file.

Model Selection Process

Initially, we thought about building our own model. However, considering the time constraints and our limited experience with model development, we decided to use a **pre-existing model** for now. The plan was to build a custom model from scratch or adapt an existing one later, depending on the available time, while continuing to develop the web application.

Selecting a Pre-Existing Model:

Although many facial expression trackers were available, finding a model with comprehensive information and suitable performance was challenging. We explored options on **Hugging Face**, which provided access to a variety of models.

After some research, we shortlisted and tested the following models:

1. Facial Emotions Image Detection by dima806
2. Facial Expression Recognition using ViT by motheecreator
3. ViT Face Expression by trpakov

Although there were numerous models available, many lacked sufficient documentation for us to fully understand their functionality. In the end, we decided to use the third one of these models to proceed with our project.

This approach allowed us to focus on integrating the model with our application while leaving room for future improvements, such as building or customizing our own model.

Selecting the Kaggle Model

After exploring various models on **Hugging Face**, we decided to use a Kaggle model for its suitability to our needs and better performance. We integrated the model using the Hugging Face API token and began making API calls for emotion analysis.

Steps to Integrate the Model via API:

1. **Search for a Suitable Model:**
   We identified and tested models on Hugging Face to ensure they aligned with our requirements.

2. **Initial Testing:**
   Using a small javascript code , we tested the chosen model by providing sample images for analysis and verifying its accuracy.

3. **Creating a Hugging Face API Token:**

   - Created an account on Hugging Face.
   - Went to **Profile > Settings > Access Tokens** to generate a token.
   - Copied the token and used it in the code with the Bearer keyword for authentication.

4. **Model Usage for Emotion Detection:**

   - Initially, we analyzed images on-the-fly to ensure the model returned accurate results.
   - Once confirmed, we started saving the model's responses in **MongoDB** along with the Session ID and image path.

Adding Functionality for Admins

Early Admin Functionality:

At this stage, we hadn't fully separated the **child** and **admin** interfaces. The website had two options:

1. **Play Game**
2. **Get Analysis**

When "Get Analysis" was clicked, the system:

- Displayed all existing Session IDs as buttons.
- On selecting a specific Session ID, the emotion analysis was retrieved from MongoDB and displayed directly in the **console** (or command prompt) for verification and we then later on displayed the console json format on the screen as well.

This process ensured that the model's output and database integration worked seamlessly before we moved on to building a user-friendly interface for admins.

Improving Image Analysis Process

Initial Analysis Process:

Initially, images were directly sent to the model every time they were captured, and the analysis response was saved along with the image path in **MongoDB**. However, this process was tedious and prone to errors, such as **axios** and **cors** errors, which disrupted the child's gameplay (happening at the same time the child was playing the game)

Refining the Process:

To avoid disrupting the child's gameplay and smoothen the process, we changed the workflow:

1. **Image Capture**: Images were captured during gameplay and saved to MongoDB without triggering analysis in real-time.
2. **Admin Triggered Analysis**: The analysis was only triggered when the admin clicked on the **"Sessions"** button in the analysis page. This ensured no interference between the child interface and the admin interface.
3. **Issue with Overwritten Responses**: Initially, each image's model response was overwritten by the next analysis, leading to the loss of previous responses. To fix this, we saved the analysis results in **MongoDB**, and whenever the admin revisited the analysis, it would show the previously saved model response instead of overriding it.

Displaying Analysis Data

Simplifying the Display:

At this point, the model responses were working well, but the analysis was presented in a **JSON** format, which wasn't user-friendly. So, we aimed to improve the presentation:

1. **Overall Analysis**: We added a section to display the **overall analysis** for all images captured together.
2. **Detailed Image Analysis**: Below the overall analysis, we included a **"Detailed Analysis"** button. When clicked, each image would be shown along with its individual analysis.

Initially, we implemented this layout in **HTML** to get a rough idea of how we wanted to display the analysis. This would later be converted into a dynamic and more visually appealing format and into react.

Enhancing Analysis with Screenshots

Purpose of Screenshots:

We decided to include screenshots of the game taken at the same intervals as the WebCam images. The goal was to provide **game developers**, **therapists**, and **admins** with context—showing where the child took a bit of time or displayed a specific emotion. This made it easier to analyze, for example, which type of question might have made the child feel sad, happy, or angry, improving the detailed analysis.

Implementing Screenshots:

As we were converting the **HTML analysis page** into a React component, we simultaneously worked on the screenshot functionality. We needed to capture screenshots at the same intervals as the WebCam images without interrupting the child's gameplay.
To achieve this:

1. We used **HTML5's <canvas>** element, which allowed us to take screenshots without affecting the visual experience for the child.
2. This also gave us the flexibility to capture only the necessary part of the screen, ensuring that the child's gameplay remained intact.

Managing Session IDs and Displaying Additional Information

Improving Session ID Usability:

Initially, the **session IDs** were long and difficult to remember, making it hard to efficiently check the analysis. To solve this, we decided to ask the player's name at the beginning of the game and use it as part of the session ID. This made it easier to identify sessions and link them to specific players.

Additionally, we added the **date and time** of the session to the analysis page. This feature helps in cases where the same player plays multiple times, allowing us to track sessions based on time and date for more accurate comparisons.

Displaying Sessions

Session Page Layout:

The **session page** now displays the following information:

- **Player's Name**
- **Session ID Button**
- **Date and Time**

When you click on a session button, it takes you to the **overall analysis** for that session, which includes a button for **detailed analysis**. Clicking the detailed analysis button redirects to a separate page that shows in-depth analysis for each image captured during the session.

Improved MongoDB Schema for Easy Access

Schema Changes:

Initially, the **image paths**, **screenshot paths**, and **model responses** were stored separately in MongoDB. To simplify access for the **analysis page**, we redesigned the schema so that:

- **Screenshot paths** are stored in the form of an array, corresponding to the WebCam images.
- **Model responses** are also stored in the form of an array, corresponding to each screenshot.
- The **date and time** are recorded only once at the beginning of the session, making it easier to track sessions.

Media Capture Interval:

Previously, screenshots and webcame images were taken every **30 seconds**, but to improve analysis and enable faster verification, we now take them **every 10 seconds**.

UI Improvements and Features

Detail Analysis Page:

After refining the backend, we focused on improving the **UI**. We experimented with different ways to present the detailed analysis. After trying several formats (including a graph-based format), we decided to use the following layout for the analysis pages:

- **Overall Analysis Page**: Displays a **bar graph** and a **pie chart** to show emotion distribution.
- **Detailed Analysis Page**: Shows a **strip bar** for each captured image. The strip bar highlights the **maximum emotion** percentage at the top, with other emotions listed below. It also displays the **timestamp** at which each image and screenshot were captured.

Session Button and Streamlined Analysis:

Originally, the session ID was a button that led to the overall analysis page. We wanted to simplify the admin's experience, so we replaced the session ID button with an **Analyze Button**.

- When clicked, it fetches the model's analysis. If no analysis exists, it sends the image to the model, stores the response in MongoDB, and then displays two options:
    1. **Overall Analysis**
    2. **Detailed Analysis**

This allows the admin to choose the type of analysis they prefer.

UI Enhancements for User Experience

Date and Time Formatting:

We made the following updates to improve the user experience:

- **Date** is now displayed with the **month name** (e.g., "Nov" instead of "11").
- **Time** is now displayed in **AM/PM format** instead of military time.

User Interface Organization:

The previous interface had everything combined, but to better separate the admin and child experiences, we redesigned the login flow:

- **Login Page**: Now includes simple login and signup options.
- For the **admin**, the name displayed must start with **"admin"** (e.g., admin01), which redirects to the **admin page** (the analysis page).
- For the **child**, the name starts with **"child"** (e.g., child01), and redirects to the **game page**.

We also generated the **session ID** for the child as soon as they logged in, as there was only one game. This change streamlined the user experience for both admins and children.

New Login System and Enhancements

Login Page Update:

We moved from a signup and login page to a **single login page** where the **credentials were pre-saved in MongoDB**. This change helped to prevent a child from logging in as an admin by simply creating an account with the name "admin". To secure the login process and prevent unauthorized access, we implemented a **bcrypt** password hashing technique.

Model Building and Code Improvements

Model Building:

After the interfaces and image analysis were functional, we focused on improving our **model building** and fine-tuning processes. Several approaches were tested:

- **Creating a model from scratch**
- **Using pre-trained models**
- **Integrating DeepFace Python library with Flask**
- **Finetuning the Vision Transformer (ViT)**

In the end, we returned to the **ViT Transformer** model that was initially used, as it provided the best performance for our needs. However, through this process, we gained valuable experience in model development and learned new techniques for future projects.

Code Refactoring:

We refactored our code to make it more flexible for future use. Originally, image capture, screenshot capture, and session generation were part of the game code. To make this reusable for other games without affecting existing functionality, we converted these features into **React hooks**. This allowed us to easily apply them to new games.

New Game Creation and Enhancements

New Game:

We developed a new **quiz-like game** that involved **missing letters in words**. In addition to the basic quiz mechanics, we added interactive features such as:

- **Voice feedback** to indicate whether the answer was correct or incorrect.
- **Drag-and-drop functionality** to make the game more engaging for children.

We incorporated the existing hooks (for image capture, screenshot capture, and session generation) into this new game, ensuring that it could be analyzed in the same way as the first game.

Game Selection and Session Management

Game Selection Page:

Instead of directly redirecting the child to the game page upon login, we introduced a **game selection page**. Here, the child could choose between the two available games. Once a game was selected, the session ID was generated, and upon completion of the game and image capture, the session ID was "stopped." If the child chose to play the other game, a new session ID was generated.

Managing Multiple Games:

When the child played both games at the same time, we encountered difficulty in identifying which game was being analyzed on the admin page. To solve this, we added the **game name** to the MongoDB schema and displayed it in the **admin page**. The admin page now displayed the following information for each session:

- **Name**
- **Game**
- **Time**
- **Date**
- **Actions** (analyze button)

UI Enhancements for Better User Experience

Admin Page Improvements:

We added several enhancements to the **admin interface** to improve usability:

- Displaying the **hour** for each session, making it easier for admins to search for a specific child's session.
- Enabled sorting and filtering to check **for a specific child or game** and to analyze the **positive/negative feedback** for each game.
- Introduced a **profile dropdown** in the header for easy access to settings and logout options.

Final UI Tweaks:

Additional improvements were made to the UI to make navigation smoother and the interface more intuitive. For example, the **favicon** was updated to give the website a more professional look.

Backend and Frontend Refactoring for Better Structure

Backend Changes:

We made several changes to the backend to improve code readability and flexibility:

- **Separated Routes and Endpoints**: Routes for the **child** and **admin** were separated to make the code more understandable and easier to modify in the future. This way, each part of the system (child and admin) can evolve independently without affecting the other.

Frontend Changes:

In the frontend, we structured the code to improve organization and accessibility:

- **Separated Components, Styles, and Assets**: We organized the **components**, **styles**, and **assets** into separate folders. This allows for easier navigation and maintenance, ensuring that developers can quickly find and modify the required files without any confusion.

Question Management:

Previously, the questions and options (along with their images) were hardcoded directly into the game code. To improve flexibility, we moved the questions to **MongoDB** and designed the system to **fetch** the questions dynamically during gameplay. This change allows for easy modification of the questions without needing to touch the game's code.

Improved Login System with JWT Authentication

JWT Authentication:

We enhanced the login system by implementing **JWT (JSON Web Token)** for secure authentication. The login page now offers two options:

- **Login**: If the user has a pre-existing username and password, they can log in as either a **child** or **admin**.
- **Register**: Only **admins** can register new accounts. Admin registration requires providing details such as name, phone number, email ID, and role.

Admin Registration and Approval:

To ensure the security and confidentiality of the system (especially due to its focus on dyslexic children, which is sensitive data) and also to make sure that no child can enter as an admin, we introduced a **super admin** role that oversees the approval process for new admins. Here's how it works:

- A new **admin** registers with their details, but their account requires **approval** from the **super admin**.
- After registration, the super admin can check the admin's information and approve the request.
- Once an admin is approved, they receive an **email notification** with a **username** (their email ID) and a **uniquely generated password**. The password is long and difficult to remember, so we also added a **password change option** in the admin profile dropdown.

Admin and Super Admin Features

Admin Features:

- **Managing Child Profiles**: Each admin can manage their own set of **child profiles**. When they click on a child's name, they can create or update the child's **name**, **age**, and **password**.
- **Child List**: The admin's page also displays a list of the children they are managing, making it easier to keep track of multiple profiles.

Super Admin Features:

- **Full Access**: The **super admin** has full access to everything, including managing games, viewing analysis, and managing child profiles.
- **Managing Admins**: One key feature of the super admin is the ability to **approve or reject new admins**. This gives the super admin control over the entire administrative process, ensuring that only authorized individuals have access to sensitive data.

FINAL PROJECT OVERVIEW

The final project is a comprehensive system designed to support dyslexic children by providing a secure platform for gameplay and emotional analysis, all while ensuring strict role-based access control for both the children and administrators. The login and registration system is structured to differentiate between three types of users: children, admins, and super admins. Children can log in and access their designated game interface, where their gameplay is captured along with their emotional responses via images and screenshots. These images are stored on the backend in dedicated folders, and their paths are saved in the MongoDB database for easy access during analysis.

Admins, on the other hand, are responsible for managing the children's accounts and analyzing the data captured during gameplay. They can create new child accounts, update existing passwords, and access detailed or overall analysis of the games played by children. The system ensures that admins only have access to the data related to children, helping them track progress and make necessary interventions based on the analysis results. If the analysis for a particular session is not available, the system fetches it from the deep learning model, processes it, and saves the results back into the database for future use. The admin interface provides tools for efficiently reviewing this analysis, including the ability to filter results by game type, time, or emotional responses.

The super admin role has the highest level of control, with the ability to approve admin registrations and manage the overall structure of the system. Super admins can view both child and admin interfaces, allowing them to monitor the system's functioning across different user types. New admin users can register by providing their details, but their accounts require approval from the super admin before they can start managing child accounts. Once an admin is approved, they receive a confirmation email containing their username (linked to their Gmail account) and a uniquely generated password, which can later be changed to something simpler for ease of use.

The overall design of the system ensures that all game-related data and user actions are logged, with clear tracking of each child's progress. This allows for effective monitoring and reporting of the children's gameplay experiences, while also maintaining a high level of security and privacy for the sensitive data involved. The implementation of this project is aimed at not only improving dyslexic children's learning outcomes but also providing a robust platform for educators and administrators to track and support their development.

PROJECT SETUP

Setting Up Git and GitHub in VS Code

1. **Clone the Repository:**
   a. Open VS Code and go to the "Source Control" tab.
   b. Click on "Clone Repository" (or press Ctrl+Shift+P and type Git: Clone).
   c. A prompt will appear with a search bar. Paste the GitHub repository link (copy the link from the GitHub repo page, under the green "Code" button).
   d. After pasting the link, choose a folder on your laptop to clone the repo into. This will download the repo to that location.
2. **Syncing Code:**
   a. To pull any new changes made by others, open the terminal in VS Code and use the command: git pull
3. **Committing Changes:**
   a. Make the required changes to your code.
   b. Go to the "Source Control" tab in VS Code (or use Ctrl+Shift+G).
   c. Enter a commit message to describe the changes you've made.
   d. Click on the "Commit" button.
4. **Configure Git User:**
   a. If this is your first commit, VS Code will ask you to configure your Git user info (name and email).
   b. Open the terminal and run these commands to configure Git:
      i. *git config user.name "YourName"*

        ii. *git config user.email "YourEmail@example.com"*

    c. Once configured, you can proceed with committing changes.

5. **Push and Pull:**
    a. After committing, push your changes to GitHub using: *git push*
    b. To get the latest changes from the repo, use: *git pull*
    c. This keeps your local code in sync with the remote repo.
6. **Persistent Configuration:**
    a. The next time you open the repo, you won't need to reconfigure your Git user (unless you change computers or need to set it up for a different user). Just commit and push without configuring Git each time.
7. **Error Fix:**
    a. If you encounter any errors while committing (such as "fatal: unable to commit"), you may need to repeat the configuration steps or check for any issues with your repository setup.

How to Create a React App

1. **Install Node.js:**
    - Download and install Node.js. Node.js includes npm (Node Package Manager), which is required for installing React and other libraries.
    - After installation, check that Node.js and npm are installed by running the following commands in the terminal: *node -v* & *npm -v*
2. **Create a React App:**
    - Open the command prompt (or terminal in VS Code).
    - Run the following command to create a new React app:

      npx create-react-app appname
    - Replace appname with the name of your app.
3. **Troubleshooting Errors:**
    - If you encounter errors like "npm start" not working, you can try the following steps:
        - Run this command to install create-react-app globally:

          npm install -g create-react-app
        - Then, try creating the app again:

          npx create-react-app appname
4. **Check the Folder:**

- Make sure the folder has the necessary files like node_modules, public, src, and a couple of package.json files after running create-react-app.

5. **Run the App:**
   - After creating the app, navigate into the newly created folder: *cd appname*
   - Start the app with: *npm start*
   - This will open the app in your browser on localhost:3000.

How to Use a React App:

1. **Folder Structure:**
   - When you open a React app in VS Code, you'll see several pre-existing folders and files:
     - node_modules/ (installed dependencies)
     - public/ (static files like index.html, images, etc.)
     - src/ (dynamic app code)
     - package.json (project metadata and dependencies)
2. **Create a Components Folder:**
   - Inside the src/ folder, create a components/ folder to organize your reusable components.
   - For example:
     - header.js, header.css
     - footer.js, footer.css
     - button.js, button.css
     - You can also create files for other sections like game.js, home.js, layout.js, etc.
3. **Import Components into App.js:**
   - Open App.js and import the components you need. At the top of the file, the first line should always be: *import './App.css';*
   - Below that, import the other components you need:

     import Header from './components/Header';

     import Footer from './components/Footer';
4. **Edit App.js:**
   - Inside App.js, use the imported components to build the layout:

```
function App() {
  return (
    <div className="App">
      <Header />
      {/* Other components */}
      <Footer />
    </div>
  );
}
export default App;
```

5. **Styling with App.css:**
   - You can style your app using App.css. If you want to style individual components, create separate CSS files (e.g., header.css for the Header component) and import them into the component file: *import './header.css';*
6. **Running the App:**
   - To run the app, open a terminal in the root of the React project and run: *npm start*
   - This will open the app on localhost:3000 in your browser.

Troubleshooting Errors:

- If you encounter errors like missing modules or issues with dependencies (e.g., canvas-confetti or react-router-dom), you can install them using npm:

  ```
  npm install canvas-confetti
  npm install react-router-dom
  ```

This should set up your React app and get it running smoothly! Let me know if you need further assistance.

Setting Up Frontend and Backend for a Full-Stack Application

To set up the **frontend** and **backend** of a full-stack application, you'll need to create separate projects for each. Here's a basic guide to set them up:

1. Set Up Frontend (React)

Steps:

1. **Install Node.js and npm**:
   - Download and install Node.js from nodejs.org.
   - npm (Node Package Manager) is installed along with Node.js.

2. **Create a React App**:
   - Open your terminal (Command Prompt or PowerShell on Windows, Terminal on macOS/Linux).
   - Run the following command to create a new React project:

     ```
     npx create-react-app my-frontend
     ```

   - This will create a folder named my-frontend with the necessary files for a React app.
3. **Navigate to the Project Folder**:
   - Change into the new directory: *cd my-frontend*
4. **Start the React App**:
   - Run the following command to start the app: *npm start*
   - This will start the React development server and open the app in your browser.

Structure of the React App:

- The app will have a folder structure like this:

```
my-frontend/
  node_modules/
  public/
  src/
    App.js
    index.js
  package.json
```

2. Set Up Backend (Node.js with Express)

Steps:

1. **Install Node.js** (if not done previously):
   - Download and install Node.js from nodejs.org.
2. **Create a Backend Project**:
   - In a separate folder, initialize a new Node.js project:

     ```
     mkdir my-backend
     cd my-backend
     npm init -y
     ```

3. **Install Express and Other Dependencies**:
   - Install **Express** to handle routing and server creation: *npm install express*

        - Optionally, install **Mongoose** if using MongoDB: *npm install mongoose*

4. **Create a Server File**:
    - Inside the my-backend folder, create a file named server.js:

```
const express = require('express');
const app = express();
const PORT = process.env.PORT || 5000;

app.get('/', (req, res) => {
  res.send('Hello from the Backend!');
});

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

5. **Start the Backend Server**:
    - Run the following command to start the backend: *node server.js*

Structure of the Backend App:

- The backend will have a structure like this:

```
my-backend/
  node_modules/
  server.js
  package.json
```

3. Connect Frontend and Backend

To connect your React frontend to your Express backend, you'll make **API calls** from the frontend to the backend.

Steps to Connect:

1. **Enable CORS (Cross-Origin Resource Sharing)**:
    - In your backend (server.js), install and configure **CORS**: *npm install cors*
    - Add the following to your server.js to allow the frontend to access the backend:

```
const cors = require('cors');
app.use(cors());
```

2. **Make API Calls from the Frontend**:
   - In your React app, use fetch or **Axios** to make requests to your backend API.
   - Example of an API call from React:

```
import React, { useEffect } from 'react';

function App() {
  useEffect(() => {
    fetch('http://localhost:5000')  // Backend URL
      .then(response => response.text())
      .then(data => console.log(data));
  }, []);

  return (
    <div className="App">
      <h1>Frontend</h1>
    </div>
  );
}

export default App;
```

3. **Run Both Frontend and Backend Simultaneously**:
   - To run the frontend (React) and backend (Node.js) servers at the same time, you can use tools like **concurrently**: *npm install concurrently --save-dev*
   - Update your package.json in the frontend folder to run both servers:

```
"scripts": {
  "start": "concurrently \"npm run server\" \"npm run client\"",
  "client": "react-scripts start",
  "server": "node ../my-backend/server.js"
}
```

   - Now you can run both servers with one command: *npm start*

Current Setup (As Described)

1. **Two Separate Folders:**
   - One folder for the **React app** (which includes src, public, etc.).
   - Another folder for the **Git repository** (where you manage commits and pushes to GitHub).
2. **Managing Changes:**
   - When you make changes to the React app, you manually copy those changes to the Git repository folder.
   - You then commit those changes from the Git repository folder using Git.
   - To pull changes from GitHub, you use git pull, copy the updated files to the React app folder, and then run the app.
3. **Manual File Management:**
   - You're not able to drag and drop files between folders within VS Code directly, so you are making changes in the **file explorer** and then copying files manually between folders.

.gitignore and .env Files

.gitignore File:

- Purpose: The .gitignore file tells Git which files and folders to ignore, preventing them from being committed to the repository.
- Common things to ignore:
  - node_modules/ (dependencies)
  - .env (sensitive info like API keys)
  - Build folders (build/, dist/)
- How to Use:
  - Create a .gitignore file in the root of your project.
  - Add files/folders to exclude (e.g., node_modules/, .env).
  - Commit .gitignore to Git.

Note: No installation is needed for .gitignore. It's just a plain text file, and Git will automatically respect the rules inside the file.

.env File:

- Purpose: The .env file stores environment variables like API keys, database credentials, or other sensitive data securely, outside the codebase.

- How to Use:
  - Create a .env file and add variables in KEY=VALUE format.
  - Access the variables in your code using process.env.
  - Add .env to .gitignore to keep it out of version control.
- Installation Requirements:
  - For React apps (created with Create React App): No installation is needed as React automatically handles loading .env variables.
  - For Node.js apps: You need to install the dotenv package by running *npm install dotenv*. Then, include require('dotenv').config() at the top of your main server file to load the variables.

Making API Calls to the Model for Comparison

1. **Find a Model**:
   - Search for a suitable model on Hugging Face (e.g., for image comparison).
2. **Test the Model**:
   - Run a small Python script to verify the model works with your input (e.g., an image).
3. **Check Accuracy**:
   - Test the model with different inputs and ensure it provides accurate results.
4. **Create a Hugging Face Account**:
   - Sign up at Hugging Face and go to **Profile > Settings > Access Tokens**.
   - Create and copy a new token.
5. **Use the Token in Code**:
   - In your code, add the token to the API request headers like this:

   ```
   headers = {"Authorization": "Bearer YOUR_TOKEN_HERE"}
   response = requests.post(url, headers=headers, files={"file": open("your_image.jpg", "rb")})
   ```
6. **Reusing the Token**:
   - Reuse the token whenever needed for API calls, but store it securely.
7. **Handle Errors**:
   - Check the response code for errors (e.g., invalid token or model issues).

Setting Up MongoDB Atlas and Compass

1. Setting Up MongoDB Atlas

MongoDB Atlas is a cloud-based database service that makes it easy to create and manage databases.

Steps:

1. **Sign Up**: Go to the MongoDB Atlas website and create an account.
2. **Create a Cluster**:
   - Once logged in, click on **Create a Cluster**.
   - Select the **Free Tier (Shared Cluster)** for development.
   - Choose your preferred **cloud provider** and **region**.
3. **Create a Database User**:
   - Navigate to the **Database Access** tab.
   - Click on **Add New Database User**.
   - Set a **username** and **password** (e.g., admin).
   - Assign the **Atlas Admin** role for full access.
   - Save the password securely.
4. **Whitelist Your IP**:
   - Go to the **Network Access** tab.
   - You can either **allow access from anywhere** or whitelist your **current IP address**.
5. **Get the Connection String**:
   - In the **Clusters** tab, click **Connect**.
   - Select **Connect Your Application**.
   - Copy the provided connection string, replacing <username> and <password> with your credentials.

2. Setting Up MongoDB Compass

MongoDB Compass is a GUI tool for managing your MongoDB databases locally.

Steps:

1. **Download and Install**: Download MongoDB Compass from the official website and install it.
2. **Connect to Atlas**:
   - Open Compass and paste the **Atlas connection string** you copied earlier into the **Paste your connection string** field.
   - Replace <username> and <password> with your credentials.
   - Click **Connect** to link Compass to your Atlas cluster.

3. Create a Database and Collection

Both Atlas and Compass allow you to create and manage databases and collections.

In MongoDB Atlas:

1. Go to the **Clusters** tab and click **Browse Collections**.
2. Click **Add My Own Data**.
3. Provide a **database name** (e.g., gameDB) and a **collection name** (e.g., sessions), then click **Create**.

In MongoDB Compass:

1. After connecting, click **Create Database**.
2. Enter the **database name** and **collection name**, then click **Create Database**.

4. Manage Data : You can view and manage your data in both Atlas and Compass.

In Atlas:

1. Go to **Browse Collections** in your cluster.
2. Select the database and collection to view or manually insert documents.

In Compass:

1. After selecting your database, click **Insert Document** or use the **Query** tab to manage data interactively.

5. Backend Integration Using Mongoose

To integrate MongoDB into your backend, you can use the Mongoose library, which simplifies database interactions.

Steps:

1. Install Mongoose in your project: *npm install mongoose*
2. In your index.js or server.js file, connect to the MongoDB database:

```
const mongoose = require('mongoose');

mongoose.connect('mongodb+srv://<username>:<password>@clustername.mongodb.net/gameDB?retryWrites=true&w=majority', {

 useNewUrlParser: true,

 useUnifiedTopology: true,

})

.then(() => console.log('MongoDB connected!'))

.catch(err => console.error('Connection error:', err));
```

Note: Replace <username> and <password> with your MongoDB Atlas credentials.

Setting up bcrypt to hash passwords and store them in MongoDB

1. **Create a collection** in your MongoDB database called userauth to store credentials.
2. **No changes needed** in the .env file for bcrypt.
3. **Add a new schema** for user authentication in schema.js.
4. Run npm install bcrypt in your backend terminal to install bcrypt.
5. **Hardcode** the username and passwords in index.js to store them in the database (this step is commented out after the data is saved).
6. Passwords are stored as bcrypt hashes in the database for security.

This setup will ensure secure password storage using bcrypt hashing.

Setting Up JWT and Nodemailer with Steps

Install Necessary Packages

1. Run the following commands to install the required libraries:

   npm install jsonwebtoken

   npm install nodemailer

If Errors Occur

- Double-check the email and password credentials in the .env file.

Setting Up a Nodemailer-Test Email Account

1. **Open Google Account Settings**
   - Go to Google and log into your account.
   - Navigate to **Security** in your account settings.
2. **Enable 2-Step Verification**
   - Search for **2-Step Verification** and turn it on.
3. **Generate an App Password**
   - Search for **App Passwords** in the security settings.
   - In "Create App," type **nodemail-test** as the app name.
4. **Copy the App Password**
   - Google will generate a 12-character password. Copy it.
5. **Update the .env File**
   - Add your email ID and the app password in the .env file:

     EMAIL_ID=your-email@gmail.com

```
EMAIL_PASSWORD=your-app-password
```

This ensures you can securely send emails using **Nodemailer** and **JWT** for authentication-related processes like password recovery or account verification.

WHY USE THEM?

Git

We used Git to manage version control, allowing us to track changes in the codebase, work collaboratively, and easily revert to previous versions when needed. It ensured organized development and reduced conflicts in team workflows.

GitHub

GitHub was chosen to host the project's code online, making collaboration seamless. It allowed multiple developers to contribute, review, and maintain a centralized, synchronized repository.

React

React was used to create an interactive and dynamic user interface for both the child and admin interfaces. Its reusable components and state management capabilities made it easy to build a scalable frontend.

Frontend

The frontend connects the users (children and admins) to the backend. It was designed using React to ensure a user-friendly experience, allowing children to play games and admins to analyze data efficiently.

Backend

The backend powers the project's logic and handles tasks like storing and retrieving data from the database, processing images, and authenticating users. Node.js and Express.js were chosen for their speed, scalability, and robust support for RESTful APIs.

Hooks

React hooks were utilized to simplify state and logic management in the frontend. Custom hooks helped modularize functionality, such as session ID generation and image capturing, making the code reusable and easier to maintain.

.env

The .env file securely stores sensitive information like database credentials and API keys, preventing them from being exposed in the codebase. This approach enhances security and simplifies configuration changes.

.gitignore

The .gitignore file ensured that unnecessary or sensitive files (like node_modules/ and .env) were not added to the Git repository, keeping the repository clean and secure.

JWT

JSON Web Tokens were used for secure user authentication. They allowed us to verify the identity of users (admins or children) and restrict access to sensitive parts of the application.

bcrypt

bcrypt was chosen to hash passwords before storing them in the database. This ensured that user credentials were secure, reducing the risk of data breaches.

MongoDB

MongoDB was selected for its flexibility and ability to handle unstructured data. It stored user information, gameplay data, and images efficiently, supporting the dynamic needs of the project.

Hugging Face Model

The Hugging Face model was integrated to analyze children's facial expressions during gameplay. This allowed us to map emotions to specific gameplay moments, providing valuable insights for educators and developers.

Express

Express.js was used to build APIs that connected the frontend, backend, and database. Its lightweight structure and flexibility made it ideal for managing requests and processing data efficiently.

Node.js

Node.js allowed us to build a fast and scalable backend using JavaScript, making it easier to integrate with the React frontend and the project's overall architecture. It also facilitated seamless real-time operations and image processing tasks.

DIFFERENT LOGIN APPROACHES

1. Saving Credentials Directly in the Database (Without bcrypt)

- **What We Did**:
  Initially, credentials were saved as plain text in the database by the developer for testing purposes.
- **Why It Was Insecure**:
  - Plain-text passwords are vulnerable to database breaches.
  - Does not comply with security best practices.
- **Conclusion**:
  This method was only used for early testing and was abandoned in favor of more secure approaches.

2. Saving Credentials in the Database with bcrypt

- **What We Did**:
  - Used bcrypt to hash passwords before storing them in the database.
  - Compared hashed passwords during login to ensure secure authentication.

- **Advantages**:
    - Enhanced security through hashing and salting.
    - Protected passwords even if the database was compromised.
- **Limitation**:
    - While bcrypt made password storage secure, token-based authentication (JWT) offered better scalability for session management.
- **Conclusion**:
  Used as a foundation for securing passwords, but JWT was chosen for stateless authentication.

3. Using JWT for Authentication

- **What We Did**:
    - Implemented JWT (JSON Web Tokens) for secure and scalable user authentication.
    - After successful login, a JWT was generated and sent to the client for subsequent requests.
    - Verified tokens on the server to authenticate and authorize users.
- **Why We Chose JWT**:
    - Stateless: No need to store session data on the server.
    - Secure: Tokens are signed and verified, preventing tampering.
    - Scalable: Efficient for handling multiple users in a distributed system.

4. Approval and Sending Credentials via Email (With bcrypt)

- **What We Did**:
    - Used bcrypt to hash passwords and securely save them in the database.
    - Sent plaintext credentials to users via email using nodemailer (e.g., for account approval or onboarding).
- **Advantages**:
    - Ensured that users were notified of their credentials securely.
    - Used hashed passwords in the database for added protection.
- **Why We Transitioned to JWT**:
    - Sending plaintext credentials via email is less secure compared to token-based authentication.
    - JWT provided a more dynamic and reusable authentication mechanism.

Final Decision: JWT Authentication

- After evaluating all methods, JWT was chosen as the primary authentication mechanism for the following reasons:
    - Stateless and scalable authentication.
    - Ability to manage user sessions securely without storing sensitive data on the server.
    - Flexibility to integrate with different login systems and enhance security.

MODELS WE WORKED ON

From Scratch Model

https://colab.research.google.com/drive/1R0RoeVQVGVf3a4hCF3QPr66dyB88pzKh?usp=sharing

1. Downloading the Dataset

You used the kagglehub library to download a dataset from Kaggle (noamsegal/affectnet-training-data). This dataset contains images and their corresponding labels that represent facial emotions.

- The dataset's path is printed once it's downloaded.
- You then loaded the labels.csv file which contains the image paths and their labels.

2. Preparing the Dataset

- You created a DataFrame with the necessary columns: 'pth' (image path) and 'label' (corresponding emotion label).
- Using the train_test_split function, you split the dataset into training and testing sets. The stratify parameter ensures that the distribution of classes (labels) is maintained in both sets.
- Then, directories for train and test were created, with subdirectories for each class/label.
- You then iterated over the training and testing DataFrames and copied the images to their corresponding directories using shutil.copy.

3. Image Preprocessing

You created ImageDataGenerator instances for both training and testing data:

- **Training Data**: It is augmented using transformations like rotation, width/height shift, shear, zoom, and horizontal flipping. This is done to improve the model's generalization by simulating different image variations.
- **Test Data**: It is only rescaled (normalized by dividing by 255) to prepare it for inference.

The ImageDataGenerator objects are used to load and preprocess images in batches, which are passed into the model during training.

4. Model Architecture

- You built a **Convolutional Neural Network (CNN)** using the Sequential API of Keras.

The architecture consists of:

- **Conv2D Layers**: These are convolutional layers that apply filters (kernels) to the input image, helping the model learn spatial hierarchies of features. You used three Conv2D layers with increasing filter sizes (32, 64, and 128).
- **MaxPooling2D Layers**: These are used after each convolutional layer to reduce the spatial dimensions of the feature maps, which helps reduce computation and overfitting.
- **Flatten Layer**: This layer flattens the 2D output from the convolutional layers into a 1D vector to feed into the fully connected (dense) layers.
- **Dense Layer**: This layer (128 units) is a fully connected layer where each neuron is connected to every neuron in the previous layer.
- **Dropout Layer**: This is used to prevent overfitting by randomly dropping 50% of the neurons during training.
- **Output Layer**: The final layer uses a softmax activation function to produce probabilities for each class (emotion).

5. Model Compilation

- You compiled the model using:
  - **Adam Optimizer**: This is an adaptive optimizer that is well-suited for training deep learning models.
  - **Categorical Crossentropy Loss**: This loss function is used for multi-class classification problems.
  - **Accuracy Metric**: This is used to evaluate the performance of the model during training and testing.

6. Model Training

- The model is trained using the model.fit method. The training is done over 50 epochs, with a batch size of 32. You used the train_generator for training data and test_generator for validation data.
- The number of steps per epoch and validation steps are calculated based on the number of samples and batch size.

Observations:

1. **Slow Progress in Accuracy**:
   - The training accuracy started around 15% and gradually increased to ~40%.
   - Validation accuracy is fluctuating and seems to stabilize around 40–45% in later epochs.
   - The model might still be underfitting or requires further tuning.
2. **Dataset Issue**:
   - The warning indicates that the input generator might not have enough samples for steps_per_epoch. This needs to be verified in your dataset preparation or generator setup.

3. **Imbalance in Classes**:
    - If some labels dominate, the model might struggle to generalize. Check for label distribution.
4. **High Validation Loss Fluctuation**:
    - Fluctuations in val_loss and val_accuracy might indicate overfitting or improper augmentation.

Fine-Tuning a PreTrained Model

https://colab.research.google.com/drive/1b1PPGvac0jTPpCKTuI4jUsS4K-cMC3qF?usp=sharing

1. Dataset Overview

The dataset used for training is from Kaggle, specifically the AffectNet training data, which contains images labeled with different emotion categories. The dataset is organized into training and test sets, and the images are categorized based on emotion labels. We used labels.csv to extract the file paths and corresponding emotion labels for the images.

2. Data Preprocessing

- **Data Splitting**: The data was split into training and testing sets using an 80-20 split. Stratified sampling was applied to maintain the distribution of labels in both sets.
- **Directory Organization**: The images were organized into respective directories under dataset/train and dataset/test, with each emotion label having its own directory.
- **Data Augmentation**: For training, we used ImageDataGenerator with several augmentation techniques:
    - Rotation, width and height shifts, shear transformations, zoom, and horizontal flips.
    - A brightness range was also applied to ensure variability in the training data.

Test data was only rescaled to normalize pixel values to the range [0, 1].

3. Model Architecture

We implemented the model using **Transfer Learning** with **VGG16** as the base model:

- **Base Model (VGG16)**: Pretrained weights from the ImageNet dataset were used. The top layers were excluded, and the model was frozen (not trainable).
- **Custom Layers**:
    - A **GlobalAveragePooling2D** layer to reduce the dimensionality.
    - A **Dense** layer with 128 units and ReLU activation.
    - A **Dropout** layer with a rate of 0.5 to prevent overfitting.
    - A final **Dense** layer with the number of units equal to the number of emotion categories, and **softmax** activation for multi-class classification.

4. Model Compilation

- The model was compiled using the **Adam optimizer** with a learning rate of 0.0001, which is a lower rate to fine-tune the pretrained weights effectively.
- The loss function used was **categorical crossentropy**, suitable for multi-class classification tasks.
- **Accuracy** was used as the metric to evaluate model performance.

5. Model Training

The model was trained for **15 epochs** on the training data using a batch size of 32. The performance on the test set was evaluated after each epoch:

- **Training Data**: Used augmentation techniques to increase variability and improve model robustness.
- **Validation Data**: Only rescaled images were used for validation.
- **Epochs**: The training lasted for 15 epochs, and the performance on both training and validation data was monitored throughout.

Observations:

1. **Slow Initial Learning**:
    - In the first epoch, the model shows a low accuracy of **14.86%**, which indicates that the model is still learning and adjusting its weights.
    - The validation accuracy at this point is **24.68%**, suggesting that the model is starting to generalize to unseen data, but it still has significant room for improvement.
2. **Modest Improvement Over Epochs**:
    - Over the course of 15 epochs, the training accuracy gradually increased to **26.46%**, and the validation accuracy improved to **29.41%**.
    - The improvement is slow, suggesting that either the model architecture, data, or hyperparameters may need further tuning for better results.
3. **Loss Decrease**:
    - The training loss decreased from **2.2319** in epoch 1 to **1.8783** in epoch 15, indicating that the model is learning and the predictions are becoming more accurate, although not drastically.
    - The validation loss also shows a similar downward trend, from **1.9932** to **1.8783**, suggesting the model is improving at generalizing, but the drop is relatively small.
4. **Possible Underfitting**:
    - The training and validation accuracies are relatively low, indicating the model may be **underfitting**. This could be due to:
        - Not enough epochs to allow the model to learn the complex patterns in the data.

        - Insufficient fine-tuning of the VGG16 base model, as all layers were frozen.
        - The choice of architecture might not be optimal for this task.

5. **Potential Class Imbalance**:
    - The confusion matrix and classification report (though not fully shown in the output) are likely to reveal that certain emotion labels might be predicted more accurately than others.
    - Class imbalance could be a factor affecting the model's performance, as some classes might dominate the predictions, especially if there are fewer examples of certain emotions.

Note about From Scratch Model and Fine-Tuned Existing Model:

In both models, we've noticed a pattern with the training epochs: every alternate epoch seems to be working as expected, showing a gradual increase in accuracy. However, the epochs in between these are completing much faster (within a couple of seconds), which is causing fluctuations in accuracy, validation accuracy, loss, and validation loss.

If we focus only on the alternate epochs, the accuracy is increasing gradually, but the intermittent epochs seem to disrupt this progress. This behavior may suggest an issue with data loading, model stability, or other factors that need further investigation to ensure consistent and smooth training.

Models with Flask:

Given the challenges faced with training a custom model from scratch, we explored leveraging pre-built libraries like DeepFace and FER for facial emotion analysis. These libraries provide pre-trained models capable of analyzing emotions from images, making them suitable for integration into real-world applications. Below is a detailed account of the process:

DeepFace Integration

Implementation:
We integrated the **DeepFace** library into our project by creating a REST API using **Flask**. The API was designed to:

- Receive images via HTTP POST requests.
- Use DeepFace to analyze the emotions in the provided images.
- Return the results as a JSON response.
- Store the analyzed results in a database for further use.

Testing and Observations:

- We tested the API with general images sourced from the internet, including images labeled as "happy," "sad," and "angry." The library performed well with these controlled test cases.
- **Challenges**:
  While testing with real-time images, the results were often biased toward "neutral" as the dominant emotion, regardless of the actual expression. This inconsistency indicated

limitations in the library's ability to handle dynamic or subtle expressions captured in real-world scenarios.

FER Library Integration

Implementation:
After encountering limitations with DeepFace, we switched to the **FER** library. Following the same Flask API approach:

- Images were sent to the API endpoint.
- The FER library processed the images to predict emotions.
- The predicted results were returned as JSON and stored in the database.

Testing and Observations:

- FER demonstrated better accuracy than DeepFace when analyzing real-time images.
- The predictions were more consistent and aligned with the visible expressions, though occasional inaccuracies were noted for complex or overlapping emotions.

Hugging Face Model

For our project, we chose the **ViT-Face Expression model** available on Hugging Face:

https://huggingface.co/trpakov/vit-face-expression

It is a **finetuned version of the ViT (Vision Transformer)** architecture. This model is finetuned from https://huggingface.co/google/vit-base-patch16-224-in21k.

Pretrained Model Overview

- **Original Model:** The **google/vit-base-patch16-224-in21k** ViT model was pretrained on **ImageNet-21k**, a large-scale dataset consisting of over 14 million images and 21,000 classes.
- **Hardware:** The model was trained using **TPUv3 hardware** (8 cores) with a batch size of **4096** and a **learning rate warmup** for 10,000 steps.

Fine-tuning Details

- **Fine-tuned Dataset:** The model was finetuned on the **FER2013 dataset**, a well-known dataset containing facial images categorized into seven distinct emotions:
    1. **Angry**
    2. **Disgust**
    3. **Fear**
    4. **Happy**
    5. **Sad**
    6. **Surprise**

7. **Neutral**

This fine-tuned model performs facial expression recognition, mapping facial images to one of the seven emotions listed above.

Reasons for Choosing This Model

1. **Pre-trained Transformer-Based Model**
   The Vision Transformer (ViT) architecture is well-regarded for its robust performance in **image classification** tasks, especially for nuanced tasks like **facial expression recognition**. This allows us to take advantage of the model's pre-existing knowledge to improve accuracy and efficiency in identifying emotions in images.
2. **Wide Emotion Classification**
   The model outputs an array of emotions along with their corresponding **scores**, which matches our requirement for detailed emotional analysis. This is ideal for our project, where we need to capture and analyze various emotions expressed by the children.
3. **Strong Evaluation Metrics**
   - **Validation Set Accuracy:** 71.13%
   - **Test Set Accuracy:** 71.16%
     These strong metrics show that the model is effective at classifying facial expressions, which is crucial for our project's goal of tracking emotions during gameplay.
4. **Ease of Integration**
   Since the model is hosted on **Hugging Face**, it integrates easily into our project through the **Hugging Face transformers** library. This allows for rapid deployment and testing with minimal effort on our part.
5. **Community and Documentation**
   Hugging Face models are known for their extensive community support and well-maintained documentation. This makes it easy to find help when needed. Additionally, this particular model had **749,237 downloads** last month, showing its popularity and reliability within the community.
6. **Open Source**
   The model is open source, which provides **complete transparency**. It is also freely available for evaluation and adaptation, saving on licensing costs and offering flexibility in adapting it to our needs.
7. **Customizability**
   The Hugging Face ecosystem makes it easy to fine-tune the model further if needed. If we wish to customize the model for specific use cases, such as handling unique datasets or adjusting to demographic variations, it's straightforward to do so.

By using this pre-trained and fine-tuned model, we can leverage the power of deep learning and transformer-based models to analyze emotional expressions with high accuracy and efficiency.

PROBLEMS & SOLUTIONS

1. **API Authentication**
   Challenge: Securing API keys and sensitive data during development and deployment.
   Solution: Stored credentials in .env files and used the dotenv library to securely load them into the application.

2. **Asynchronous API Calls**
   Challenge: Managing multiple asynchronous requests for the HuggingFace model and handling potential failures or timeouts.
   Solution: Employed async/await for cleaner asynchronous calls and added robust error-handling with retry mechanisms.

3. **CORS Issues**
   Challenge: Encountering cross-origin resource sharing (CORS) errors while connecting the frontend and backend.
   Solution: Configured the backend with CORS middleware to handle preflight requests and set appropriate headers.

4. **Webcam Image Dimensions**
   Challenge: Maintaining consistent image dimensions for captured webcam images, especially when sending them to the model.
   Solution: Dynamically resized images using canvas elements on the frontend before encoding them for backend processing.

5. **Base64 Encoding**
   Challenge: Efficiently converting webcam images to Base64 format for model processing.
   Solution: Used browser-based encoding functions to convert images into Base64 format and ensured compatibility with the backend API.

6. **Model Processing Time**
   Challenge: The HuggingFace model sometimes took up to 20 seconds to analyze images, causing delays.
   Solution: Implemented retries with exponential backoff to handle delays gracefully and improve user experience.

7. **Handling Large Data Volumes**
   Challenge: Managing and storing a large volume of images and model responses in the database without affecting performance.
   Solution: Optimized database indexing, designed efficient schemas, and set up periodic cleanup of temporary data.

8. **Role-Based Access Control (RBAC)**
   Challenge: Restricting unauthorized access to certain pages and features based on user roles.
   Solution: Implemented middleware on the backend to verify user roles and ensured frontend navigation respected role-based restrictions.

9. **Integration of Webcam and Screenshot Functionality**
   Challenge: Capturing both webcam images and game screenshots simultaneously and linking them for analysis.
   Solution: Used JavaScript's MediaDevices API for webcam capture and appropriate game rendering frameworks for screenshot captures, ensuring both were linked for analysis.

10. **Managing Multiple Games and Session IDs**
    Problem: It was challenging to track which session ID corresponded to which game.
    Solution: Introduced a game selection page to let the child pick a game before generating the session ID. The session ID was deactivated after each game, and a new one was created for each new game.

11. **Identifying Games Played Simultaneously**
    Problem: Tracking which game data belonged to which gameplay became difficult when multiple games were played at once.
    Solution: Added a game name column in the MongoDB database and displayed the game name in the admin interface to easily identify which game was played.

12. **Difficulty in Changing Game Questions**
    Problem: Hard-coded questions made updating the game difficult.
    Solution: Moved game questions to MongoDB, allowing them to be fetched dynamically, making it easier to modify content without touching the codebase.

13. **Lack of Clear Separation Between Child and Admin Routes**
    Problem: Mixed routes for child and admin users made the code harder to maintain.
    Solution: Separated child and admin routes to make the code more organized and prevent accidental misuse.

14. **Handling Complex User Roles (Admin and Super Admin)**
    Problem: Managing multiple roles and permissions was challenging, especially with admin approval.
    Solution: Introduced a super admin role to approve admin registrations and manage all aspects of the platform, while admins could only manage child profiles within their scope.

15. **Improving the UI for Better Usability**
    Problem: The UI lacked user-friendly features like easy profile searching and performance tracking.
    Solution: Added search functionality, data visualization for game comparisons, and a profile dropdown for easier navigation and settings access.

16. **Securing the Platform and Sensitive Data**
    Problem: Sensitive information required strong security measures.
    Solution: Implemented JWT authentication, role-based access control, and an admin approval process to secure data and ensure only authorized personnel had access.

17. **Handling the Complexity of Emotion Recognition**
    Problem: Emotion recognition models were not consistently accurate for children.
    Solution: Opted for a pre-trained Vision Transformer (ViT) model, fine-tuned it, and integrated it with the frontend for real-time facial expression analysis.

18. **Optimizing the Game Experience for Children**
    Problem: The game lacked interactive features.
    Solution: Redesigned the game with drag-and-drop mechanics, added voice feedback, and made the game more engaging to improve the learning experience for children.

Future Scope

Suggestions for future enhancements

Personalized Analysis for Different Users:

Therapists:

- Display detailed game results, including:
    - The child's answer to each question.
    - The time taken to answer each question.
    - A breakdown of the child's emotional states during gameplay to assess engagement and learning outcomes.
- Provide insights on the child's progress over time and suggest potential areas of focus based on emotional trends.

Game Developers:

- Show metrics that help optimize game design, such as:
    - Average time taken by children to complete specific tasks or levels.
    - Emotional engagement levels at different points in the game.
    - Feedback on which game mechanics evoke the desired responses (e.g., joy, focus).
- Offer recommendations for enhancing user experience based on emotional and gameplay data.

Fine-Tune the Existing Model for Better Expression Tracking:

- Retrain the HuggingFace model using a more diverse dataset with facial expressions tailored for dyslexic children.
- Incorporate additional parameters like microexpressions, gaze detection, and context-based sentiment analysis for higher accuracy.
- Regularly update the model based on real-world data collected during gameplay to improve prediction and personalization.

TECHNICAL GLOSSARY

API (Application Programming Interface):
A set of protocols and tools that allow software applications to communicate with each other.

Asynchronous API Calls:
A method for handling multiple API requests without blocking the application's execution flow, ensuring smoother operations.

Base64 Encoding:
A process of converting binary data (like images) into a text format for safe transmission over networks.

CORS (Cross-Origin Resource Sharing):
A mechanism to control how resources on a web server can be accessed from a different domain, often required for frontend-backend communication.

Deep Learning:
A subset of machine learning that uses multi-layered neural networks to analyze and process complex data, such as images or emotions.

dotenv: A library used to manage environment variables securely in Node.js applications.

Environment Variables (.env):
Configuration files used to securely store sensitive information like API keys, database credentials, and secret tokens.

Express.js:
A lightweight web application framework for Node.js used to build backend services and APIs.

Facial Recognition Model:
A machine learning model designed to identify and categorize emotions based on facial expressions captured in images.

Frontend:
The client-facing interface of an application, often built with React.js in MERN stack projects.

Hugging Face:
An open-source platform providing pre-trained machine learning models for tasks like emotion detection and image recognition.

JSON Web Token (JWT):
A compact, URL-safe token used for secure authentication and data exchange between clients and servers.

MediaDevices API:
A JavaScript interface that enables access to multimedia input devices like webcams for image capture.

MERN Stack:
A JavaScript-based development framework consisting of MongoDB (database), Express.js (backend), React.js (frontend), and Node.js (runtime environment).

MongoDB:
A NoSQL database that stores data in flexible, JSON-like documents, used for managing user data and gameplay logs.

Pre-Trained Model:
A machine learning model that has already been trained on a large dataset and can be fine-tuned for specific tasks.

Role-Based Access Control (RBAC):
A security feature that restricts access to resources and operations based on user roles, such as admin or child.

Session ID:
A unique identifier assigned to a user session, allowing tracking and linkage of gameplay data to specific sessions.

Vision Transformer (ViT):
A state-of-the-art deep learning model architecture used for image classification and facial emotion recognition.

Webcam Integration:
The process of enabling applications to access and capture images from a webcam using browser APIs

---

REFERENCES & OTHER LINKS

Project Repositories

- **Old Repo**: Expression Tracker (Sentiment Analysis for Dyslexic Kids During Gameplay)
- **New Repo**: Expression Tracker 2.0

Milestone Presentations

- **Milestone 1**: MILESTONE_1.pptx
- **Milestone 2**: MILESTONE_2.pptx

Game and Login Demonstrations

- **Game Video**: Expression Tracker Game
- **Login Video**: Expression Tracker Login

Hugging Face Models Referenced

1. Facial Emotions Image Detection by dima806
2. ViT Facial Expression Recognition by motheecreator
3. ViT Face Expression by trpakov

Models Created by Us

- **From Scratch Model**: Colab Link
- **Pretrained Model**: Colab Link

Research Papers

1. Facial Expression Recognition in the Wild
2. Attention Is All You Need
3. Game-Based Learning as a Teaching and Learning Tool for Dyslexic Children
4. Front-End Development in React: An Overview
5. Prediction of Image Preferences from Spontaneous Facial Expressions

Learning Resources

- **MERN Stack Overview**: MongoDB Resources
- **GitHub Learning**: GitHub Learning Tutorial
- **JWT Learning Video**: JWT Tutorial
- **Model Building Video: Facial Recognition Model Video**