**Project Report- the FaceMergeQuiz App**

Gur Keinan 213635899 Idan Pipano XXXXXXXXX Itamar Reinman XXXXXXXXX

**Introduction**

The FaceMergeQuiz app is an innovative, exciting multiplayer game that merges creativity with friendly competition. Designed for users who enjoy puzzles, visual challenges, and social gaming, FaceMergeQuiz combines two distinct player-uploaded images into a single merged photo. The players must then guess critical elements of the resulting image in a fun and interactive quiz format.

The app introduces a dynamic gameplay experience, where users can challenge friends with private games or test their luck by being paired with random opponents. By incorporating real-time interactions, image manipulation, and a seamless user experience, FaceMergeQuiz turns simple images into an engaging competition.

Leveraging powerful backend technology to ensure a smooth user experience, FaceMergeQuiz promises a fast-paced, visually rich environment that keeps players entertained and connected. Whether solving a puzzle with a friend or competing with players across the globe, FaceMergeQuiz transforms image-based gaming into an unforgettable experience.

**Details on the Utility of the Server**

The FaceMergeQuiz app offers an intuitive and engaging workflow that keeps players immersed in the game from start to finish while ensuring a seamless and responsive experience at every stage.

Game Workflow: From Creation to Guessing

The journey begins when a player either creates a game (private or public) or joins a random game through a matchmaking system. The server immediately handles game creation, generating a unique game code for private matches or placing the user in a queue for random pairings.

Once both players are connected, the game advances to the image submission phase. Each player uploads or selects an image from a URL, which the server processes and stores securely. The server ensures that both photos are received and processed in real-time before merging them.

The merging process is where the magic happens! The server calls an external image-processing API to combine the two-player images into a mysterious composite image. Once the merged image is ready, both players are presented with it, along with a quiz where they must guess the correct answer related to the image.

The guessing phase is the heart of the gameplay. Players are given multiple-choice options, cleverly shuffled, and must select the correct answer. The app guarantees a fun and challenging experience, whether they win or lose.

Search Engine paragraph?

Robustness for Varied Behaviors

We’ve built FaceMergeQuiz with resilience at its core, ensuring the game can gracefully handle disruptions. Whether a player leaves mid-game, experiences connectivity issues, or cancels a match, the server manages these scenarios effortlessly. If a player quits while in the waiting room or during the game, the system cleans up their session, removes them from matchmaking queues, and cancels ongoing games. This prevents users from becoming stuck or experiencing game delays, allowing other players to continue with minimal disruption.

Additionally, games can be easily cancelled and restarted, with all related data securely deleted or rolled back. This flexibility keeps gameplay flowing smoothly and ensures a frustration-free experience for all players, regardless of interruptions.

API Integration for Image Merging

One of the app's standout features is its seamless image merging, achieved through integration with the Gradio API. Once both players have uploaded their images, the server sends them to Gradio, which performs advanced image manipulation to combine them into a single composite. The resulting merged image is then retrieved and stored securely in the database, ready to be displayed in the quiz.

This real-time API interaction ensures that the merging process is swift and seamless, offering players an impressive and enjoyable visual experience. Using an external service allows the app to maintain performance and responsiveness without compromising on the complexity of the image-processing tasks.

Using MongoDB for Real-Time Data Management

At the core of the FaceMergeQuiz app is a highly scalable and efficient database powered by MongoDB. MongoDB allows the server to dynamically manage user data, game sessions, and player matchmaking queues in parallel, providing real-time updates as multiple users interact with the app simultaneously.

*- User Data:* MongoDB securely stores user credentials, hashed passwords, and personal statistics (e.g., wins, losses), ensuring a persistent and secure user experience.

*- Game Sessions:* Every game is tracked in MongoDB, storing data such as game status, player information, images, and quiz answers. This real-time tracking allows the server to manage multiple active games at once, effortlessly supporting many concurrent users.

*- Dynamic Updates:* MongoDB is continually updated as games progress—tracking everything from image uploads to quiz results. This allows for real-time adjustments, such as handling users who quit mid-game, pairing new users for random matches, or retrieving quiz data.

The app’s ability to handle multiple players concurrently, updating their game status, sessions, and queues, is a testament to the robustness of MongoDB and the way it is used in this architecture. Whether it’s managing a surge of new players joining at once or dynamically matching and managing random game queues, MongoDB ensures the system remains responsive and scalable.

The FaceMergeQuiz server is the backbone of a fast, fluid, and highly interactive gaming experience. Its ability to handle real-time image merging, dynamic matchmaking, and robust user management ensures players enjoy a smooth, uninterrupted experience. The combination of Flask, MongoDB, and API integrations provides a scalable, reliable infrastructure capable of handling the most demanding scenarios with ease and excitement!

**Technical Implementation Details**

The FaceMergeQuiz app is a technically sophisticated system that integrates a range of web development best practices, real-time interaction, and complex user behaviors. The backend, primarily built using Flask, works seamlessly with a dynamic and highly interactive front-end powered by HTML, JavaScript, and asynchronous API calls. Below, we break down the key aspects of the technical implementation, emphasizing non-trivial elements and the seamless interaction between front-end and back-end components.

Backend Implementation (Flask Routing and Logic)

The routes.py file in the backend of FaceMergeQuiz contains the core logic that drives the app. Each route defines a specific functionality, handling everything from user authentication to game state management, and from image processing to database interactions. Here are some of the notable backend processes:

*- Session Management and User Authentication:* Using Flask-Login, user sessions are maintained with secure authentication, and MongoDB handles the storage of user credentials, wins, and losses. These sessions allow seamless transitions between pages and games without the need for constant re-authentication.

*- Dynamic Game Handling:* The app supports private and random game creation. For instance, the /start-game route allows a user to initiate a game, while routes like /check-created-game and /check-random-game handle real-time updates to check if a second player has joined. These routes enable non-blocking gameplay where users can wait for an opponent to connect or start immediately when both players are ready.

*- Image Handling and API Integration:* One of the standout features is the image merging functionality, powered by external APIs (e.g., Gradio). Routes like /upload\_image handle the image submission by converting user-uploaded files or selected images into base64 format and storing them in MongoDB. The server then sends these images to the API for merging, retrieves the merged result, and presents it to both players in a quiz format. The use of asynchronous calls here ensures that the game remains responsive even while the external API processes the images.

*- Polling and Game Cancellation:* Routes such as /check\_game\_status allow real-time polling of game status to check if an opponent has uploaded an image or if the game has been canceled. This functionality helps maintain the integrity of the game flow, ensuring that players remain informed and the app handles unexpected events like user disconnection or mid-game cancellations.

Front-End Interaction: Complex UI Behaviors

The front-end implementation exemplifies several advanced web development techniques that are tightly integrated with the back-end Flask routes. The HTML and JavaScript code not only facilitates a smooth user experience but also handles non-trivial behaviors such as asynchronous requests, real-time updates, and form validation. The following features highlight these aspects:

*- Dynamic Image Selection and Preview:* The user interface allows for multiple ways to submit images—either by uploading a local file or selecting a pre-defined image from a search query. This dual input method is handled seamlessly with JavaScript functions like previewFile() for local uploads and searchPhotos() for fetching images from a server-side search query. The system ensures that only one image source is selected at a time, clearing the other when a new image is chosen.

*- Asynchronous Image Fetching and Display:* When the user performs a photo search via the front-end search bar, a POST request is sent to the /search\_photos route, and the results are dynamically displayed in the photo grid without requiring a page reload. This non-blocking interaction enhances the user experience, keeping the interface responsive and intuitive.

*- Real-Time Polling for Game Status:* The front-end implements polling mechanisms using the checkGameStatus() function to continuously check if a game has been canceled by the opponent or if both players have uploaded their images. This process involves repeated AJAX requests to the /check\_game\_status route, maintaining constant communication between the front-end and back-end while keeping the user informed of any changes without manual refreshes.

*- Handling Well-Behaved and Unexpected Navigation:* A crucial feature is the differentiation between well-behaved navigation (e.g., submitting forms or proceeding to the next step in the game) and unexpected exits (e.g., closing the browser or leaving the page mid-game). This distinction is managed through JavaScript flags like isWellBehavedNavigation. If the player leaves the game unexpectedly, the cancelGame() function is triggered, which sends a request to the /cancel\_game route to cleanly exit the game and update the database, ensuring the game isn't left in a limbo state.

*- Form Validation and Submission:* The front-end uses the FormData API to handle form submissions efficiently. Before uploading, the form ensures that all necessary inputs, such as image and answer fields, are filled. Upon submission, the data is packaged and sent asynchronously to the /upload\_image route, which handles image processing and answer storage on the server-side. The result of this submission is immediately communicated back to the user via dynamic status messages, making the user aware of the game’s current state (waiting for other players, image ready, etc.).

Database Integration (MongoDB for Real-Time Updates)

MongoDB serves as the backend database, supporting the real-time nature of the game by managing user information, game states, and image data in a non-relational, document-oriented structure. This architecture provides several advantages:

*- Concurrent Game Management:* Each player’s game session is stored as a document in MongoDB, which tracks essential information such as player IDs, game status, and the images they upload. The database supports multiple concurrent games, with each game being dynamically updated as players interact with it (e.g., uploading images or submitting guesses).

*- Queue Management for Random Games:* MongoDB handles real-time queues for users waiting to join random games. As players enter or leave these queues, their status is updated immediately in the waiting\_users\_collection, ensuring a fluid matchmaking experience.

*- Real-Time Data Handling:* MongoDB’s ability to handle high-velocity data transactions allows the system to support real-time updates. For example, once both players have uploaded images, the merged image is generated and stored in the game document. This allows the game to progress smoothly without delays, as the merged image is retrieved instantly when required by the players.

In summary, the FaceMergeQuiz app exemplifies an elegant integration of front-end interactivity, real-time back-end processing, and robust data management through MongoDB. The app’s ability to handle dynamic image uploads, merging via external APIs, and seamless game play for multiple concurrent users underscores its technical sophistication. The use of Flask for routing, combined with MongoDB’s scalability and JavaScript-driven dynamic UI, ensures that the app provides a responsive, engaging, and reliable experience for users.

**Appendix 1 - Reflection on the Most Challenging Aspects of the Project**

**Appendix 2 – Images from the FaceMergeQuiz App**

**Appendix 3 – Endpoints Description**

This section will review all the app's endpoints and describe their functionality and role in the overall app workflow.

Add the classification API endpoint.

1. `/` - Home

- Method(s): GET

- Authentication Required: Yes

- Description: This endpoint renders the home page for logged-in users. It displays general information about the user and the current game status.

- Role: This is the main landing page after the user logs in.

2. `/sign-up` - Sign Up

- Method(s): GET, POST

- Authentication Required: No

- Description: Users can sign up for an account by providing a username and password. On form submission (POST), the app checks if the username already exists and, if not, creates a new user with encrypted credentials and logs it into the database.

- Role: Facilitates user registration and redirects the user to the home page upon successful sign-up.

3. `/login` - Login

- Method(s): GET, POST

- Authentication Required: No

- Description: The login page allows users to enter their username and password to authenticate. On a successful login, the user is redirected to the home page.

- Role: Handles user login, verifies credentials, and initiates a user session.

4. `/logout` - Logout

- Method(s): GET

- Authentication Required: Yes

- Description: Logs the user out of the app and redirects them to the home page.

- Role: Ends the user’s session and ensures proper logout behavior.

5. `/join-game` - Join Game

- Method(s): GET

- Authentication Required: Yes

- Description: Allows the logged-in user to create a new game or join an ongoing game. At this age, the user chooses one of 3 options – create a new game, join a game using code (a game someone else has already created), and join a random game – wait until someone else wants to join a random match and be paired up.

- Role: Handles game joining for users, directing them to the game lobby.

6. `/start-game` - Start Game

- Method(s): GET

- Authentication Required: Yes

Description: This feature enables the user to create a new game with a unique game code. It checks for any existing games the user may have initiated and deletes them before creating a new one to avoid duplication, as it does not hurt any other user.

- Role: Responsible for game creation, storing the game details in the database, and redirecting the user to the waiting room for the new game.

7. `/waiting-room-created-game` - Waiting Room for Created Game

- Method(s): GET

- Authentication Required: Yes

- Description: Renders the waiting room for a user who has created a game and is waiting for another player to join.

- Role: Provides an interface for the user to wait for a second player to join the created game.

8. `/check-created-game` - Check Created Game Status

- Method(s): GET

- Authentication Required: Yes

- Description: Periodically checks if another player has joined the game created by the user.

- Role: Used to poll the server to check the status of the created game.

9. `/leave-created-game-waiting-room` - Leave Waiting Room for Created Game

- Method(s): POST

- Authentication Required: Yes

Description: This allows the user to leave the waiting room and cancel the game if no other player has joined. It is used in cases where a player leaves the game, and we don’t want another player to join an already created game accidentally. If we recognize an inactive player (one that left the waiting room), we delete the created game from the database.

- Role: Manages game cancellation and session cleanup when the user decides to leave the created game's waiting room.

10. `/join-random-game` - Join Random Game

- Method(s): GET

- Authentication Required: Yes

- Description: Puts the user in a queue for a random game. If a match is found, it pairs the user with another player and starts a game. A game begins if two or more players are waiting in the queue. After a pairing has been found, a new game object is created to contain all the information that is important for the workflow of the game, and both players are removed from the waiting queue. One player is also transferred from the waiting room to the page at the beginning of the game.

- Role: Handles joining random games, placing the user in a waiting pool, and pairing them with other users.

11. `/leave-random-waiting-room` - Leave Random Waiting Room

- Method(s): POST

- Authentication Required: Yes

- Description: Activated if a user in the waiting room for the random game is recognized to be inactive. We remove such players from the waiting queue, ensuring that no active player will later be paired up with an inactive player.

- Role: Cancels random game match attempts and cleans up session data.

12. `/check-random-game` - Check Random Game Status

- Method(s): GET

- Authentication Required: Yes

- Description: Periodically checks if the user has been paired with another player for a random game. A player in the waiting room occasionally pools the server and checks if it has been paired up.

- Role: Provides the user with real-time status updates about their random game matchmaking.

13. `/waiting-room-random-game` - Waiting Room for Random Game

- Method(s): GET

- Authentication Required: Yes

Description: This is a waiting room for users who have joined a random game and are waiting for another player to be matched.

- Role: Manages the user interface for waiting in a random game matchmaking process.

14. `/enter-code` - Enter Game Code

- Method(s): GET, POST

- Authentication Required: Yes

Description: This feature allows users to enter a specific game code to join a private game. It will check if the code entered matches any existing games.

- Role: Facilitates joining private games using a game code.

15. `/game-ready` - Game Ready

- Method(s): GET

- Authentication Required: Yes

- Description: Displays the page when both players are ready and the game has been initiated.

- Role: Finalizes the game setup and ensures that both players are ready to proceed with the game.

16. `/load\_image` - Load Image

- Method(s): GET

- Authentication Required: Yes

Description: This allows the user to load an image for the game. The users have two choices: either uploading an image locally from their device or searching for an image in our built-in search engine, which contains many images of celebrities.

- Role: Responsible for handling the game’s image upload or selection process.

17. `/upload\_image` - Upload Image

- Method(s): POST

- Authentication Required: Yes

- Description: Handles the image upload process from the user and stores it in the database.

- Role: Manages user image uploads and updates the game data with the provided images.

18. `/check\_merge\_ready` - Check If Images Are Merged

- Method(s): GET

- Authentication Required: Yes

- Description: Periodically checks if both users have uploaded their images and if they have been successfully merged.

- Role: Used to poll the server for the status of the image merging process.

19. `/show\_merged\_image` - Show Merged Image

- Method(s): GET

- Authentication Required: Yes

- Description: Displays the merged image from both users and provides a multiple-choice quiz for guessing the correct answer.

- Role: Presents the merged image to the user and allows them to make a guess based on the provided options.

20. `/submit\_guess` - Submit Guess

- Method(s): POST

- Authentication Required: Yes

- Description: Handles the guess submission from the user and determines if the guess is correct or incorrect.

- Role: Processes user guesses and determines game outcomes.

21. `/game\_result/<result>` - Game Result

- Method(s): GET

- Authentication Required: Yes

- Description: Displays the game result (win/lose) after the guess has been submitted.

- Role: Provide feedback on the game's result, congratulating the user or encouraging them to try again.

22. `/cancel\_game` - Cancel Game

- Method(s): POST

- Authentication Required: Yes

- Description: Cancels the game initiated by the user and updates the game status to `canceled`. This function is used if one player is recognized as inactive during the game (after a pairing was found) and before the inactive player has uploaded their image. In this case, we don’t want the active player to wait forever for an image that isn’t going to be sent, so we inform them that the game has been canceled and transfer them to the home page. Then, we clean up the remains of the canceled game to avoid any accidents later on.

- Role: Allows users to cancel their active game if they wish to exit early.

23. `/check\_game\_status` - Check Game Status

- Method(s): GET

- Authentication Required: Yes

- Description: Checks if the game has been canceled and cleans up any associated game data.

- Role: Polls the server for the current game status to determine whether it is active or canceled.

24. `/game\_cancelled` - Game Cancelled

- Method(s): GET

- Authentication Required: Yes

- Description: Displays a message to the user if the game has been canceled.

- Role: Informs the user that their game was canceled either by them or their opponent.