Project Report - the FaceMergeQuiz App

Gur Keinan 213635899 Idan Pipano 213495260 Itamar Reinman 3269935285

Introduction

Have you ever wondered what a merged image of your Soft4ML TA's face and your Soft4ML lecturer's face would look like? Or what would your child look like had you married Britney Spears? Even more wild—what would the face of a person you choose, combined with someone your friend picks, look like? If this sounds interesting, FaceMergeQuiz is the app for you!

Details on the Utility of the Server

Game Workflow: On the homepage, statistics about the games you have played so far are displayed colorfully, and there is an inviting "Join a game" button.

Joining a game: A player can choose between three options: 1) create a new game and receive a game ID to send to a friend; 2) join an existing game using a game ID; or 3) join a random game – this means either waiting for another player to choose this option if there is no one currently waiting, or being matched with a player who is currently waiting.

How a game works: Once a game starts, each player uploads 1) an image, 2) the right answer, the name of the persona in the image, and 3) two distractions. After both players finish uploading, the server merges the two faces. Each player is then displayed with the merged image and the three answers the other player wrote. Now, each player has to guess the true answer (the name of the person the other player uploaded). If you guess correctly – you win! (regardless of whether the other player guesses correctly). Otherwise, you lose. After you submit your guess, you are told whether you won or lost and then redirected to the home page.

How players can upload images: We provided the user three options for choosing an image for the game. First, you can upload an image from your device. However, you might not currently have exciting photos in your device, so for the second option, we added a search bar that lets the user search the web via our very own search engine and choose a photo from there. Because even this can be too much effort for extremely lazy users, we added a third option: users can select one of six predefined images (can you guess who is in those images?) presented to them on the screen. In this option, the user can click on their favorite image for it to be chosen.

Robustness for Varied Behaviors: We designed FaceMergeQuiz to be resilient to various disruptions during the game. Such disruptions include a player leaving mid-game or canceling a match. Handling these disruptions prevents users from getting stuck or experiencing game delays.

Security Features: To prevent code use for password cracking and as a protection against DoS attacks, we have implemented access restrictions on the registration and login screens. We let a specific API address send a request to one of these routes no more than once per second, with the

system ignoring any requests that exceed this limit. Additionally, we have created conditions designed to prevent access to various routes unless it follows the system's predefined flow.

Technical Implementation Details

We have used the following tools in our app:

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. 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 keep the user logged in 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 Gradio API Integration:* The core of our app is the image merging process. 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 merging itself is done by an asynchronous call to a gradio API. 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.

- *Search Engine logic:* we created a custom search engine and focused the search on portrait images, which we found to work best for the face-merging API. Additionally, we filtered the images based on their size to achieve stable results.

Front-End Interaction: Complex UI Behaviors

- *Dynamic Image Selection and Preview:* The user interface allows for three ways to submit images, as detailed earlier. This is handled with JS functions like searchPhotos() for fetching images from a server-side search query. The system ensures that only one image source is selected, and the JS function previewFile() displays only one image at any time (keeping only the last image 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, keeping the interface responsive.

- *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 is not left active.

Database Integration (MongoDB for Real-Time Updates)

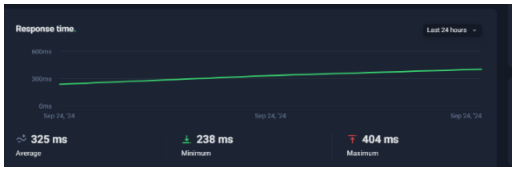
MongoDB serves as the backend database, supporting the real-time nature of the game by managing user information, game states, and image data in a non-relational, document-oriented structure. This architecture provides several advantages:

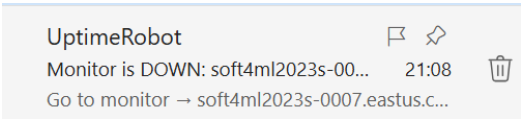
- *Concurrent Game Management:* Each player's game session is stored as a document in MongoDB, which tracks essential information such as player IDs, game status, and the images they upload. The database supports multiple concurrent games, with each game being dynamically updated as players interact 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 smooth matchmaking experience.

Aside: we used an uptime-robot to monitor that our app is up.

Interestingly, this only worked in the VM and did not work on our personal laptops. We suspect that this is due to our laptops' firewall blocking the requests uptime-robot sent.





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

Written below are the main difficulties we experienced during the creation of our app:

- Writing tests that simulate two players playing our game (face_merge_quiz/tests/test_game_flow.py) was quite challenging.
- Dealing with containers, docker networks, mountings and so on was challenging at first.
- Debugging js code. This was challenging because we are not familiar with js. It took us time to even understand where we can see the outputs of console.log(). We also performed imports in js to avoid code duplication, which was tricky to debug at first:

```
<script src="/static/js/show_predefined_images.js"></script>
```

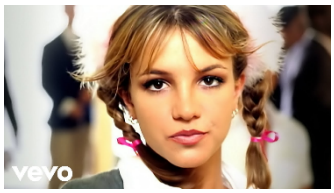
Appendix 2 – Our Favorite FaceMerges

The great thing about this project is that we got to see A LOT of face merges in the development and debugging process. Here are our top 3:

- Ori Farkas & Gur Keinan:



- Britney Spears & Sarit Hadad:

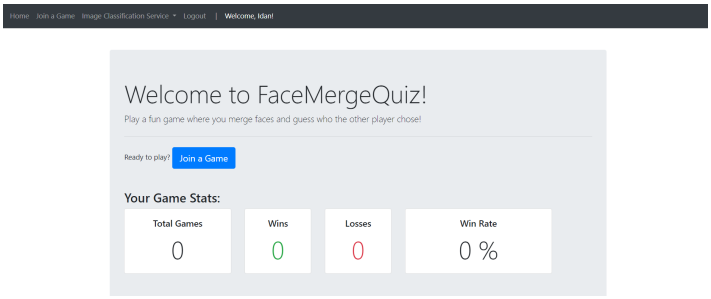



- Idan Pipano & Itamar Reinman:

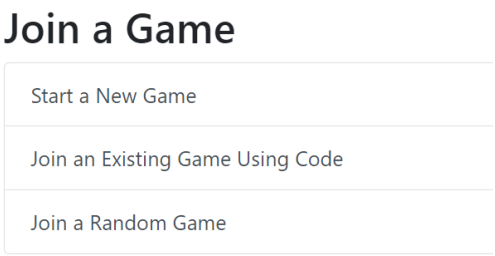


Appendix 3 – Images from the FaceMergeQuiz App

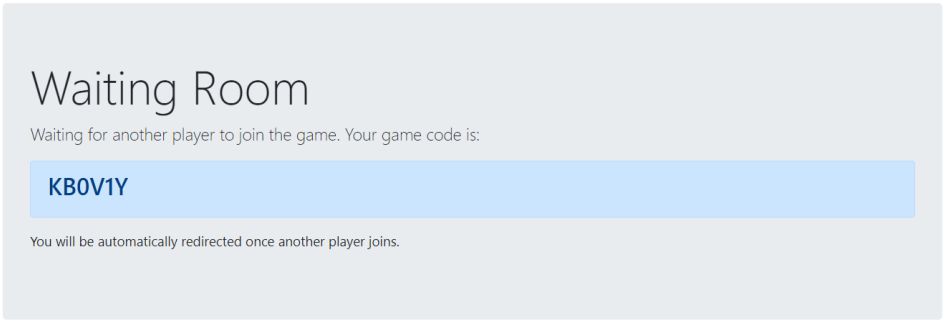
Home page



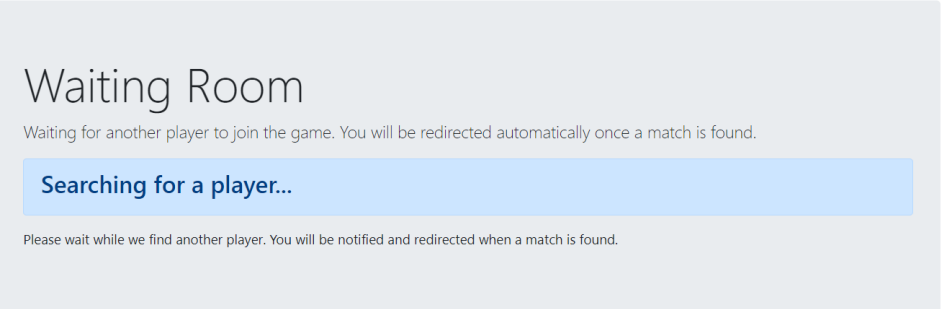
Joining a game



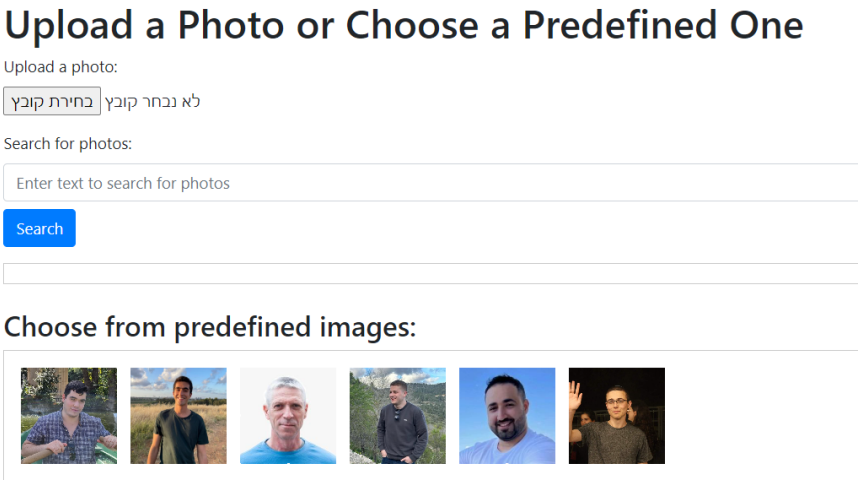
Waiting room – created game



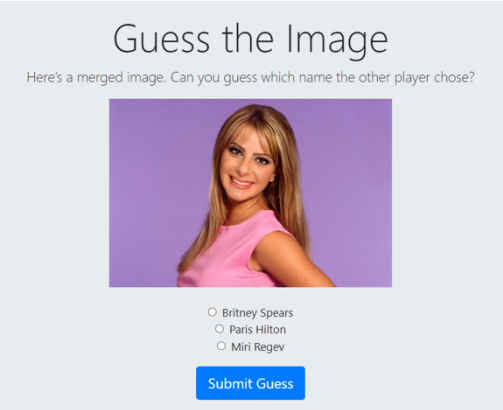
Waiting room - random game



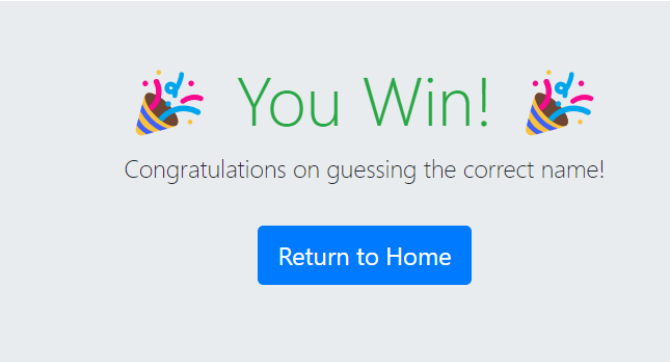
Choosing an image



Guessing what the other player uploaded!



Being told whether you guessed correctly:



Appendix 4 – Endpoints Description

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

Endpoints for the image classification API:

1. `/api/status`
   - Method(s): GET
   - Authentication Required: Yes
   - Description: performs a GET request to the /status endpoint in the image classification API container.
   - Role: Show the status of the image classification API.
2. `/api/upload_image`
   - Method(s): GET, POST
   - Authentication Required: Yes
   - Description: the GET is responsible for displaying a nice page to the user where it can upload an image she would like to classify. The POST makes a POST request to the /upload_image endpoint of the image classification API and shows the user the results in a nice format.
   - Role: Responsible for letting the user interact with the image classification API's POST /upload_image.
3. `/api/async_upload`
   - Method(s): GET, POST

- Authentication Required: Yes
- Description: the GET is responsible for displaying a nice page to the user where it can upload an image she would like to classify. The POST makes a POST request to the /async_upload endpoint of the image classification API and shows the user the request id it returned.
- Role: Responsible for letting the user interact with the image classification API's POST /async_upload.

4. `/api/result`
   - Method(s): GET, POST
   - Authentication Required: Yes
   - Description: the GET is responsible for displaying a nice page to the user where she can enter a request_id. Then when the user submits the request_id, the POST calls the image classification API's GET /result/<request_id> and shows the response to the user.
   - Role: Responsible for letting the user interact with the image classification API's GET /result/<request_id>.

<u>Endpoints for the FaceMergeQuiz app:</u>

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, POST

- 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. The POST is relevant only for testing, where a user id and game code are sent directly.

- 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, POST

- Authentication Required: Yes

- Description: Periodically checks if another player has joined the game created by the user. The POST is relevant only for testing, where a user id is sent directly.

- 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 do not 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, POST

- 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. The POST is relevant only for testing, where a user id is sent directly.

- 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. The POST is relevant only for testing, where a user id is sent directly.

- 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, POST

- 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. The POST is relevant only for testing, where a user id is sent directly.

- 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.

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. `/search_photos` - Search Photos

- Method(s): POST

- Authentication Required: No

- Description: Handles the search engine interaction – it receives a query from the user and returns relevant images.

- Role: Responsible for handling the interaction with the search engine.

18. `/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.

19. `/waiting-for-other` - Wait for Other to Upload Image

- Method(s): POST

- Authentication Required: Yes

- Description: Waiting page until the second player uploads their image.

- Role: Waiting page until the second player uploads their image.

20. `/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.

21. `/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.

22. `/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.

23. `/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.

24. `/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 do not want the active player to wait forever for an image that is not 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.

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

25. `/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.

26. `/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.