# COMP3207 Report: Group O

Kissa: find a purrfect match

German Nikolishin
[gn2g21@soton.ac.uk] [32566964]

Danny Martinez
[dmh1g19@soton.ac.uk] [31138039]

Daniel Braghis
[ddb1u20@soton.ac.uk] [32161204]

Irfan Qureshi
[ibma1u21@soton.ac.uk] [33355479]

Augustas Mirinas
[am26g21@soton.ac.uk] [33141673]

January 2024

## 1 Overview

"Kissa: find a purrfect match" is a dating app emphasising mindful connections. The app allows one to meet other people based on their pictures of their cats. In the world of dating, we believe that cats reflect the personality of their owners, which is backed up by our survey. Kissa allows users to register and fill in their personal information about themselves and their cats. Interestingly, the other A user cannot see your data or pictures until there is a match between you two. Another notable feature is the limited number of connections. Users are encouraged to meet with their matches to match with others. This way, we encourage users to be mindful of their matches. The app is deployed as a mobile-first website with primitive matching algorithms and simplified messaging.

## 2 Tools Used

### 2.1 Frontend

- React
- Hooks
- Stateless
- Javascript

## 2.2 Backend

- FastAPI
- Pydantic
- MongoDB, GraphFS, Base64
- Python 3.10

## 2.3 Cloud

- Terraform
- Docker Containers & Compose
- MongoDB hosted on CosmosDB
- Azure Container Manager & Application Azure, Key Vault,

## 3 Project Statistics

- Lines of code
  - Python:
  - Javascript:
  - Total:
- Resources used
  - Medium Article - haversine distance calculation
  - Fast API docs
  - Stack Overflow
  - Pydantic docs
  - Terraform on Azure docs
  - Azure Container Apps docs
  - cloc implementation
  - React Reference
  - Geolocation API
  - Design Inspirations from Tinder

## 4 Design and Implementation

Our frontend and backend have been implemented as separate components. This ensures the correct separation of concerns, allowing us to develop both independently. This also facilitates the potential implementation of the mobile app client independent of the frontend application.

## 4.1 Frontend

Our frontend is a mobile-first web application with responsive elements to accompany desktop clients built with React. This technology was selected to allow elements be more responding and animated to the API state. The app has multiple pages, some nested into parent pages:

- */login* - First page to be visited. Displays options to either log in or register a new account by redirection to */register*
- */register* - Page where a new account can be created. Consists of three states: 1 - fill in user information. 2 - upload user profile picture. 3 - upload images of the cat.
- */app* - Parent page containing three main application pages - to edit user and cat profiles, match with other users and messaging. Parent page itself displays the menu, which is used to navigate between three main pages.
  - */* - Main page displaying one potential match at a time with options to either match with the suggested user or skip them.
  - */profile* - Page for profile editing. Uses React states to automatically detect changes in user preferences or cat profile and call the API to save these changes.
  - */messages* - Page for messaging. User has a selection of matches and ability to select one at a time. When selected, a chat with that match is displayed. To send a message HTTP request is used and the chat is being refreshed every 5 seconds to attempt to get latest messages.

All these pages are using cookies - either creating them (when logging in) or reading access token from cookies to use the API. We are also using JavaScript Geolocation API to match between relevant users based on their location. Location range can be edited as well.

## 4.2 Backend

The architecture of the backend server strictly follows a stateless and flexible structure using FastAPI through separation of concerns and route handling.

The application uses MongoDB with CosmosDB to store and retrieve user data, including base64 images using GraphFS.
We have opted for NoSQL database for the flexibility in the schema management and easy-to-use setup that was vital for the rapid development of the prototype. We have also separated the endpoints into three major categories

- *profiles* - personal and cat-related profile operations and generation of potential matches
- *pictures* - CRUD operations of pictures
- *matches* - CRUD operations of matches

This separation facilitates the clear navigation of our REST API which can also be observed from auto-generated OpenAPI documentation.

Our application has an authentication system with session management based on *bearer* tokens that follow the modern react specification. Users can sign up and provide their data, which is represented in different formats and gets stored in MongoDB database in the cloud. All incoming schemas get validated against defined models to ensure the model safety of the prototype. The API endpoints also ensure meaningful error codes and messages are returned to the client.

The matching algorithm is implemented trivially. When a user updates their location via *PATCH* request, the backend first finds all users within a given radius calculated by harverstine distance formula and then filters in entries matching user preferences: age range and gender.
We have also attempted to restrict the CORS setting to reduce the security exposure of our prototype. The user also provides images stored in a separate table and references in other schemas. Users can interact through a simplified messaging service using HTTP requests.

### 4.3 Cloud

We decided to use Azure as our cloud platform. The infrastructure is mostly managed using Terraform with the Azure CLI. The advantages of this approach are the ease of use compared to the Web interface with variables available to setup services instead of keeping track of assigned names and values used for the setup. Finally, the approach also offers a "recipe" for setting up the infrastructure on other Azure accounts.

We opted for Azure Container Apps as the environment for our application because this enables the use of Docker containers which ease the development for a multi-platform team like ours (MacOS, Windows and Linux). The API and Web App are separate containers with the API having a main and dev container app to separate stable releases from the development branch to not interfere with frontend development.

We use GitHub and a CI/CD pipeline for version management through GitHub actions that trigger deployments on our Container Apps. The builds are not triggered automatically because the repository is private. Because setting up CI/CD is not supported through Terraform, the container apps are the only resources allocated and configured manually in our infrastructure.

Azure Key Vault stores sensitive data like hash keys to generate tokens or database connection strings. The secrets are injected from the key vault as container environment variables at deployment time.

## 5 Evaluation

We believe we have implemented the core functionality of the originally designed prototype. In particular, we have envisioned to provide a proof-of-concept design of the application that allows to match people based on their cats, and this goal has been achieved functionally. The backend provides the necessary functionality of authentication, authorisation, schema validation, and meaningful error reporting and serves the CRUD operations required by the frontend. In the frontend, we have strived for simplicity of use resembling the UI of modern dating applications. We have successfully managed to implement the UI closely resembling the mock-ups.

Given limited resources, time, and unforeseen circumstances, we were unable to integrate the AI-based image segregation service, allowing filtering out non-cat images uploaded on the platform. However, we believe this feature can easily be integrated with the current design. Additionally, user messaging is handled primitively via HTTP requests instead of web sockets. This introduces a significant application flaw and should be addressed in the next development cycle. Finally, our code lacks unit tests, which were compensated by the end-to-end tests using Postman and shell scripts to ensure the application behaves correctly. We acknowledge that this unit testing is vital for the future development of our service.

Overall, we are satisfied with our final result and believe that we delivered the proof-of-concept of service with a unique dating workflow. All group members have gained a learning experience in group collaboration, utilisation of modern web tooling, and cloud deployment while working on the prototype.

## 6 Contributions

### 6.1 German Nikolishin

- Ideation, Mockup designs, project management, and code reviews
- CRUD operations of *profile*, *match* endpoints
- Matching algorithm and testing

### 6.2 Danny Martinez

- Picture upload
- Authentication
- Back end and front end form handling (login register)

### 6.3 Daniel Braghis

- Azure Infrastructure
- Containerization
- Version control and CI/CD

### 6.4 Irfan Qureshi

- Implemented front end designs for the login, register and not found pages

### 6.5 Augustas Mirinas

- Frontend route structure and management
- Main page design and implementation - matching, profile editing and messaging
- Frontend testing and pull request validation

*The sixth member was unable to contribute to the project and applied for special considerations.*