AIT MATCH

by

Tatiya Seehatrakul

A Thesis Submitted in

AT70.25 Full Stack Application Development

Instructure: Dr. Chantri Polprasert

Teacher Assistant: Sunil Prajapati

Asian Institute of Technology

School of Engineering and Technology

Thailand

November 2024

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to my esteemed instructor, Dr. Chantri Polprasert, for his invaluable guidance, constructive feedback, and unwavering support throughout the course of this thesis. His expertise has been instrumental in shaping the direction and successful completion of this project.

I am also sincerely thankful to Mr. Sunil Prajapati, Teaching Assistant for the course, for his technical assistance and timely advice, which significantly contributed to overcoming various challenges during the development of this work.

I would like to extend my appreciation to the Asian Institute of Technology, particularly the School of Engineering and Technology, for providing the necessary resources and an intellectually stimulating environment that facilitated the completion of this thesis.

ABSTRACT

AIT Match is a dating platform created exclusively for students at the Asian Institute of Technology (AIT), offering a balanced approach between social and academic life within the university. The platform is designed to foster meaningful connections by leveraging both personal and academic information to create a tailored dating experience. Users can only register with a valid AIT email, ensuring that the platform remains a secure and trusted space for students.

Key features of AIT Match include personalized profile creation, where profiles are displayed based on personal characteristics and academic background, and robust preference filters, allowing users to search for matches by personality traits such as age, gender, MBTI type, interests, and relationship preferences, as well as academic information like schools, programs, and degrees. Additionally, real-time messaging is enabled exclusively between matched users, promoting communication only after mutual interest has been established.

AIT Match also includes an admin and reporting system, where users can report any inappropriate behavior to our admin team. This feature ensures that any issues are addressed promptly, maintaining a safe and respectful environment for all users.

By integrating academic and personal interests, AIT Match provides a platform that aligns with the values and community spirit of AIT. This application serves as a unique and focused alternative to traditional dating apps by prioritizing academic context and personal connections within the university environment.

CONTENTS

CHAPTER 1
INTRODUCTION

1.1 Background of the Study

In the digital age, online platforms have become a widely accepted medium for forming relationships, particularly among university students who are adept with technology. However, existing dating applications cater to a broad demographic, making them less tailored to the unique social and academic dynamics of university life. These generalized platforms often lack the contextual depth necessary to foster meaningful connections based on shared academic experiences, social interests, or professional aspirations.

At the Asian Institute of Technology (AIT), students come from diverse cultural and academic backgrounds, creating an environment rich in opportunities for meaningful connections. However, the demands of academic life, coupled with the challenges of navigating a multicultural environment, often make it difficult for students to engage in social and romantic relationships. Many students express a desire to connect with peers who share similar academic goals, career aspirations, and social interests, yet there is no existing platform specifically designed to facilitate this within the university context.

AIT Match is designed to address this gap by offering a university-exclusive dating platform where AIT students can connect based on shared academic and social interests. The platform focuses on providing a secure and trusted environment where users can form meaningful connections reflective of their academic and personal lives. Unlike generic dating apps, AIT Match is uniquely positioned to leverage the academic and social context of university life, fostering deeper relationships while ensuring authenticity through a verified university email registration process. This approach ensures that the platform remains exclusive to the AIT community, enhancing trust and connection within the student body.

1.2 Statement of the Problem

In today's digital landscape, many platforms are designed to facilitate social and romantic connections. However, these platforms target a broad audience and do not meet the unique needs of university students, particularly those seeking meaningful relationships grounded in their academic environment. For students at the Asian Institute of Technology (AIT), the absence of a platform that integrates both academic and social aspects presents several challenges:

1. **Limited Focus on Academic Connections:** Existing platforms often emphasize quick, surface-level interactions, lacking features to connect users based on their academic programs, courses, or career aspirations, making it harder to find peers with aligned academic interests.
2. **Inadequate Filtering Options:** Without filters for academic backgrounds, students have fewer opportunities to connect with peers in their field of study. The ability to search by academic criteria such as degree program, faculty, or school is crucial but often restricted or unavailable.
3. **Lack of University-Specific Features:** Current platforms do not offer features tailored to the university experience, leaving students without a secure environment to connect based on both academic and personal interests.
4. **Time Constraints of Master's and Doctoral Students:** Postgraduate students, particularly those pursuing master's and doctoral degrees, often struggle to engage in social or romantic activities due to their demanding schedules. A platform that integrates academic and personal preferences can simplify this process, allowing them to find like-minded peers more efficiently.

The lack of a platform that balances academic and social interests within a university-specific environment creates a significant gap for AIT students. This project aims to fill that gap by developing a platform that integrates academic information with personal preferences, offering a balanced approach to dating and social interaction within the AIT community.

1.3 Objectives

The primary objective of this project is to develop a dating platform exclusively for students at the Asian Institute of Technology (AIT), with a focus on enhancing both social and academic engagement. The specific objectives of the project are as follows:

1. **Create a University-Exclusive Dating Platform:** Provide a dating platform specifically for AIT students, fostering connections within the university community.
2. **Ensure User Safety and Trust**: Implement a university email verification process to ensure that all users are part of the AIT community, fostering a safer and more trusted environment for interaction.
3. **Build a Secure and User-Friendly Platform**: Develop an intuitive and secure platform that enhances user experience, ensuring ease of use while maintaining a high level of data security and privacy.
4. **Develop a Customized Matching System**: Build a matching system that allows users to filter potential matches by academic and personal preferences, such as degree, program, interests, and personality traits.
5. **Enhance Social Engagement**: Encourage students to build relationships through shared social interests, facilitating meaningful connections beyond academic settings.
6. **Enhance Academic Engagement**: Promote interactions based on academic backgrounds, helping students connect through shared academic interests, programs, and schools.

1.4 Target User

The primary target users of AIT Match are students from the Asian Institute of Technology (AIT), encompassing all academic disciplines. AIT students come from a diverse array of cultural, professional, and educational backgrounds, each pursuing rigorous academic programs that require significant time and dedication. Given these academic demands, many students find it challenging to engage in social activities or form meaningful connections within their community. AIT Match aims to address this gap by providing a platform that allows students to connect with peers who share not only social interests but also academic and professional goals. This approach ensures that users can find others who understand the unique pressures and opportunities associated with university life.

By catering to the entire AIT student body, AIT Match fosters a sense of inclusivity and belonging within the community. The platform allows students to filter potential matches based on shared academic fields, extracurricular activities, and personal interests, thereby facilitating connections that are both relevant and meaningful. Whether students are pursuing degrees in engineering, management, science, or any other discipline, AIT Match creates a space where academic and social engagement can intersect. The platform's focus on shared academic experiences and extracurricular involvement ensures that it serves as a tool for building lasting relationships within the AIT community, enriching the student experience across all programs.

1.5 Scope of the Project

The scope of this thesis encompasses the development of a web-based platform designed to facilitate user interaction through a set of well-defined functionalities. The platform is structured around user and administrative roles, with the following key features:

- **Development of a Web-based Platform:** The project involves the creation of a web-based platform that enables users to access its features via a web browser on multiple devices.
- **Data Management through SQL Database Architecture:** The platform utilizes an SQL-based database architecture to efficiently manage and store user data, ensuring scalability, reliability, and data integrity.
- **User Functionalities:** Users can act both as general users and as administrators. The functionalities provided to general users include the following:
  - **Landing Page:** The landing page presents a navigation bar that provides access to the homepage, information about the platform, contact details, user login, and sign-up options.
  - **User Authentication:** The platform supports secure user authentication through registration with a university email address and password. Users can reset their password via email if needed.
  - **Main Page:** Upon successful login, users are directed to the main page. The navigation bar on this page includes links to key features such as profile creation, preference management, matches, and messages. The homepage provides a searchable list of profiles, with sorting options such as alphabetical (A-Z or Z-A) and random profile display.
  - **Profile Management:** Upon initial registration, users are required to create a personal profile. This profile contains both personal and academic information, including first name, last name, username, email address, date of birth, gender, MBTI type, interests, relationship preferences, and a profile picture URL. Additionally, users can manage academic details such as degrees, schools, programs, and educational background. Users can view their own profiles as well as those of other users, with the option to edit their information at any time.
  - **Preference Management:** Users have the ability to set preferences to filter displayed profiles based on a range of criteria, including age range, preferred

gender, interests, relationship preferences, MBTI type, degrees, schools, programs, and username. Filtered profiles are prominently displayed on the homepage, tailored to each user's preferences.

  - **Matching System:** The platform includes a robust matching system that allows users to express interest in connecting with others. When a user sends a match request, the recipient is notified in real time through a notification icon displayed on the navigation bar. If the recipient accepts the request, both profiles appear in the "Matches" section, enabling the users to communicate with each other. Pending match requests are displayed on a dedicated "Match Requests" page, allowing users to manage incoming connection requests easily.
  - **Messaging Functionality:** The messaging feature is available exclusively to users who have successfully matched. Messages between users are displayed in the "Messages" section, along with a timestamp indicating when each message was sent.
  - **Report System:** Users can report inappropriate behavior on profiles to the AIT Match admin. The report status is updated in real time, and once the admin reviews the report, the status is changed accordingly. This system ensures prompt handling of issues to maintain a safe and respectful environment on the platform.

- **Admin Functionalities:** The system includes a single admin, designated by the developer, with distinct capabilities separate from regular users. The admin has access to the following functionalities:
  - **Profile Deletion:** The admin has the authority to view and delete user profiles. If a user engages in inappropriate or harmful behavior, their account may be banned from the platform.
  - **Report Management:** Only the admin has access to view reports submitted by users, which are displayed on the admin dashboard. The admin can review these reports and take appropriate action, such as removing the reported user from the platform to ensure the safety of all users. The status of each report is visible to the reporting user after it has been reviewed. Once reviewed, the admin can choose to delete or retain the report, while users do not have permission to delete it.

CHAPTER 2
RELATED WORK

AIT Match draws inspiration from various online dating and social networking platforms. Mainstream dating apps like Tinder, Bumble, and Hinge cater to a wide audience and utilize appearance-based matching systems. While effective for casual connections, they lack features suited to the academic and social needs of university communities, often missing out on fostering connections based on shared academic or professional interests.

Student-focused platforms like Friendsy and Datamatch attempt to bridge this gap by offering student-only dating experiences requiring university email registration. However, these platforms generally lack customization for individual institutions, limiting their effectiveness in facilitating connections in a specific academic setting like AIT.

Professional networking platforms such as LinkedIn and Facebook provide tools for academic and campus connections, but they are not tailored for romantic or deep social engagement. Similarly, broad social networks like Facebook serve diverse users and interests, making it difficult to foster close, university-centered interactions within an academic environment.

Moderation-focused apps like Coffee-Meets-Bagel emphasize authenticity and safe interactions but do not offer university-verified spaces or prioritize academic connections. These limitations underscore the need for AIT Match, which combines the safety features of mainstream apps with the social and academic engagement crucial in a university community. Fig. 2.1 illustrates the related applications.

Figure 2.1 *Related dating applications including Tinder, Bumble, Hinge, Friendsy, Datamatch, LinkedIn, Facebook, and Coffee Meets Bagels.*

CHAPTER 3
METHODOLOGY

3.1 Planning and Research

The requirements for the system were gathered through online research conducted by the author in collaboration with a group of individuals experienced in using dating applications. During this process, the author examined existing dating apps to identify potential areas for improvement, with a particular focus on incorporating academic-based filters for profile searching. The collected data was then analyzed to plan the system's implementation, define its scope and functionalities, and present the design through diagrams, including an ER Diagram to visually represent the structure of all database tables.

3.2 System Design

3.2.1 Back-End Architecture Design

The architecture was designed for a web-based platform optimized for laptops and computers, with the code written once but capable of running across all operating systems in web browsers. A Multi-Page Architecture (MPA) was implemented to separate each page and function, allowing for easier development and maintenance of different components independently. This architecture is ideal for navigating between different pages, enhancing the user experience by providing a seamless and organized interface. The back-end architecture design is illustrated in Fig. 3.1.

3.2.2 Back-End Stack Design

The platform was developed using the Ruby on Rails stack, following the MVC (Model View Controller) design pattern. The tools and programming languages used include Rails, Ruby, PostgreSQL, HTML, CSS, Javascript, and Docker. The back-end stack design is illustrated in Fig. 3.2.

1. **Rails (Rails Team, 2004):** A web development framework that follows the MVC architecture.
2. **Ruby (Ruby Team, 1995):** The programming language used within the Ruby on Rails framework for web development.
3. **PostgreSQL (PostgreSQL Global Development Group, 1997):** An open-source, reliable database system capable of handling large-scale data and supporting a wide range of data types, such as JSON and XML.

4. **HTML (W3C, 1993):** To create the structure of web pages and organize content (headings, paragraphs, images, links).
5. **CSS (W3C, 1996):** To style the HTML elements on the web pages, including colors, fonts, spacing, and edges, to enhance the visual appearance of the website.
6. **JavaScript (ECMA International, 1993):** To create dynamic content such as button clicks, animations, and dynamically updating content without requiring a page refresh.
7. **Docker (Docker, Inc., 2013):** A containerization platform that allows for consistent development, testing, and deployment environments across different systems.

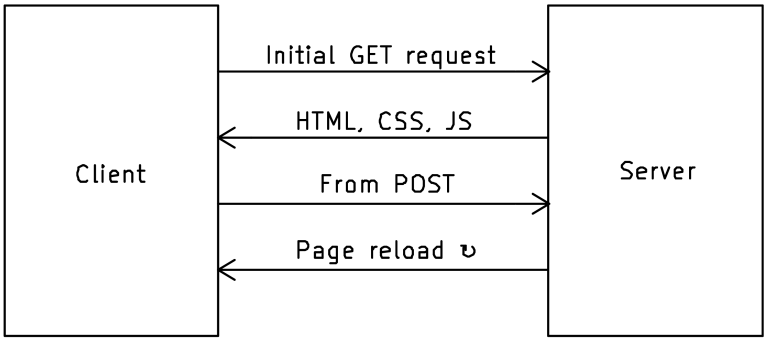

Figure 3.1 *Back-End Architecture Design of AIT Match Platform.*

Figure 3.2 *Back-End Stack Design of AIT Match Platform, including, Rails, Ruby, PostgreSQL, HTML, CSS, Javascript, and Docker.*

3.2.3 Back-End Database Design

The database efficiently manages all user and admin data, facilitating secure and meaningful connections. The primary entities in the system include:

1. **Users:** The *Users* table stores essential information for user authentication and management, such as email, encrypted passwords, reset password details, and user roles. This table has a one-to-one relationship with the *Profiles* table, allowing each user to have a single associated profile. Additionally, it has a many-to-many relationship with the *Matches* table, enabling users to connect with multiple other users.
2. **Profiles:** The *Profiles* table contains detailed user-specific information, such as first name, last name, username, MBTI type, gender, degree, school, program, educational background, and birthday. It holds a one-to-one relationship with the *Users* table and a many-to-many relationship with the *Interests* table through the *Profile_Interests* join table. This structure allows users to showcase their personal and academic details within the system.
3. **Active Storage:** The application uses Active Storage to handle file uploads, specifically for user profile pictures. This implementation involves three tables:
   (a) **active_storage_attachments:** This table tracks the relationship between records (such as profiles) and the files attached to them. Each entry corresponds to an attachment.
   (b) **active_storage_blobs:** This table stores the actual files and their metadata, including filename and content type.
   (c) **active_storage_variant_records:** This table manages variants of uploaded files, which can be used to create different versions (like thumbnails) of the images.
4. **Genders, Degrees, Schools, Programs, MBTI:** These preset tables store standardized values for user attributes such as gender, degree, school, program, and MBTI type. They are associated with the *Profiles* table, ensuring that users can select these attributes consistently during profile creation and updates, thereby maintaining structured and uniform data across the platform.
5. **Profile_Interests:** This join table manages the many-to-many relationship between *Profiles* and *Interests*. It links user profiles to their selected interests, which are stored in the *Interests* table. By storing user preferences in this way, the system

can enhance matchmaking by aligning users based on shared hobbies or interests.

6. **Profile_Relationships:** The *Profile_Relationships* table facilitates a many-to-many relationship between user profiles and predefined relationship types from the *Relationships* table. This structure allows users to specify their preferred types of relationships, enriching the platform's ability to match users based on compatible preferences.
7. **Interests:** The *Interests* table contains a predefined list of interests from which users can choose. It establishes a many-to-many relationship with *Profiles* through the *Profile_Interests* table. This connection helps the system to accurately match users who share similar interests, improving the overall relevance of suggested profiles.
8. **Relationships:** The *Relationships* table stores predefined relationship preferences. It is linked to the *Profiles* table via the *Profile_Relationships* table, enabling users to indicate the types of relationships they are seeking and improving the system's ability to create meaningful matches.
9. **Preferences:** The *Preferences* table has a one-to-one relationship with the *Users* table and stores users' filtering preferences, such as preferred age range and relationship types. It is linked to various tables, including *PreferredGenders*, *PreferredDegrees*, *PreferredSchools*, *PreferredPrograms*, *PreferredMBTIs*, and *PreferredRelationships*, allowing users to define specific preferences for potential matches.
10. **PreferredGenders, PreferredDegrees, PreferredSchools, PreferredPrograms, PreferredMBTIs, PreferredInterests, PreferredRelationships:** These tables store user preferences in categories such as gender, degree, school, program, MBTI type, interests, and relationship types. They establish many-to-many relationships with the *Users* table, contributing to the system's filtering and matchmaking capabilities by aligning users with others based on their preferred attributes.
11. **Matches:** The *Matches* table facilitates a many-to-many relationship between users, represented by the requestor and receiver profiles. When two users mutually accept each other, a match is created, allowing them to communicate through the system. This table also tracks the status of match requests, indicating whether a match is pending or confirmed.

12. **Conversations:** The *Conversations* table stores communication between matched users. Each conversation is associated with a match and involves two profiles, corresponding to the sender and receiver. This structure ensures that all messages between matched users are appropriately organized and tracked.
13. **Messages:** The *Messages* table is linked to *Conversations* and stores individual messages exchanged between users. It contains fields such as the message content, the sender profile, and timestamps, ensuring a detailed history of all communications between matched users.
14. **Reports:** The *Reports* table is associated with the *Profiles* table and stores reports submitted by users regarding other profiles. It includes fields such as reporter_profile_id, identifying the user who submitted the report, reported_profile_id, identifying the user being reported, reason, detailing the justification provided by the reporter, and status, indicating the current state of the report ("pending" or "reviewed"). This structure enables system administrators to efficiently track, review, and manage user reports, ensuring a safe and respectful platform environment.

The ER diagram depicting the back-end database design is presented in Fig. 3.3 (see this link for the diagram). A more detailed version can be found in Fig. 3.4 (see this link for the detailed diagram).

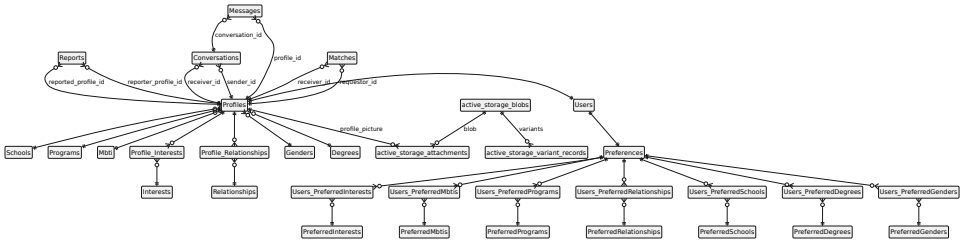

Figure 3.3 *Back-End Database Design of AIT Match Platform.*

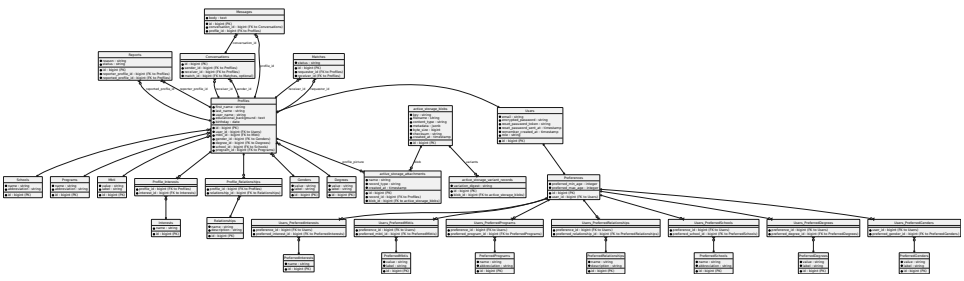

Figure 3.4 *Detailed Back-End Database Design of AIT Match Platform.*

3.2.4 Front-End User Interface Design

1. **User Interface Design and Color Palette:** The user interface (UI) was designed with a focus on real-world usability and user experience (UX). The goal was to make the application intuitive and easy to use across all platforms. Ruby on Rails uses '.html.erb' files to generate dynamic views, which incorporate both HTML for structuring the web pages and CSS for styling. This combination ensures that the front-end is both functional and visually appealing, allowing for consistent and responsive user interactions.

   The color scheme of the website is inspired by tones that create a balanced, safe, and welcoming environment. The green color, derived from the university's branding, represents trustworthiness and security, making users feel comfortable using the platform. Soft pink tones are used to encourage romantic interactions subtly and add warmth to the user experience. The white and grey tones provide a neutral contrast that keeps the interface clean, readable, and visually balanced. The AIT Match color palette is illustrated in Fig. 3.5.

Figure 3.5 *AIT Match color palette.*

2. **Navigation Bar:**
   - The logo is prominently displayed.
   - **Landing Page Navigation Bar:** Includes buttons for the about page, contact page, login page, and sign-up page.
   - **User Navigation Bar:** Includes buttons for home, profile, preferences, matches, messages, and logout.
   - **Admin Navigation Bar:** Includes buttons for dashboard, reports, and logout.

3. **Login/Sign-up Page:** Allows users to log in or sign up using their university email and password.
4. **Landing Page:**
   - **About Page:** Displays information and an overview of the platform.
   - **Contact Page:** Provides the contact information for the developers.
5. **Users' Main Page:**
   - **Home Page:** Displays all profiles and filtered profiles. Users can match with other users and view profiles.
   - **Profile Page:** Allows users to view and edit their profiles.
   - **Preference Page:** Allows users to set their preferences to filter other profiles on the home page based on basic information, personality traits, and academic information.
   - **Matches Page:** Displays profiles of users who have already matched with the current user.
   - **Messages Page:** Shows the latest messages exchanged with matched profiles.
   - **Report Page:** Users can report inappropriate profiles to the admin with a detailed reason for each report. The report status will be updated by the admin after review. Users can view their report history but do not have the right to delete any reports.
   - **Logout Button:** Logs the user out of the system and redirects them to the login/sign-up page.
6. **Admin's Main Page:**
   - **Dashboard Page:** Displays all user profiles in the system. The admin has the ability to delete profiles as necessary.
   - **Report Page:** Shows all reports submitted by users. The admin reviews each report, updates the status accordingly, and has the authority to delete reports, which users cannot do.
   - **Logout Button:** Logs the user out of the system and redirects them to the login/sign-up page.

3.3 System Development

3.3.1 Environment Setup

The AIT Match project uses Ruby on Rails with Docker to streamline development and deployment. Docker ensures consistency across different environments (development, production, etc.). This architecture ensures isolated, reproducible development environments, minimizing environment-specific issues and improving the deployment process. The key components of the environment setup are as follows:

1. **Database Configuration:** The `database.yml` file configures the PostgreSQL database connection, specifying the adapter, host, username, and password. It ensures that Rails can communicate with the database in both development and production environments, integrating seamlessly with the Docker PostgreSQL container.
2. **Docker Compose for Database:** Docker Compose orchestrates multiple containers, including the PostgreSQL database (`match_db`) and PgAdmin (`match_pgadmin`). The services are connected via an internal network (`match_network`) for communication. Volumes are used to persist data across container restarts, ensuring database state retention.
3. **Docker Compose for Rails Application:** Docker Compose also manages the Rails application container (`match_web`) and its dependencies like Redis. The Rails service is set up using a Dockerfile and commands to manage the database and precompile assets. Environment variables configure external services, such as Gmail for email notifications and Redis for caching. Redis runs in a separate container and links to the Rails container via the internal network, ensuring persistent data and consistent environments for development and production.

3.3.2 User Authentication

After the environment setup is completed, the next step is to implement user authentication. This is critical to manage access control, allowing only authorized users to log in, register, or reset their passwords securely.

1. **Add Devise and Necessary Gems for Authentication**

   The first task is to add the Devise gem to the Rails application, which provides a complete solution for handling user authentication.

   — Install the required gems for authentication:

   (a) **Devise:** A flexible authentication solution for Rails applications.

   (b) **Bcrypt:** Used for secure password encryption.

   (c) **ORM_Adapter, Railties, Warden:** Additional libraries for interacting with the ORM and ensuring authentication logic and security.

   — After adding these gems to the `Gemfile`, run `bundle install` to install them inside the container.

   — Generate the Devise configuration and user model using the following commands:

   ```
   ./bin/rails generate devise:install
   ./bin/rails generate devise User
   ```

2. **Validations for Secure User Authentication**

   To ensure the integrity of the registration and login process, validation rules are applied to the Devise model. The following steps ensure that users provide valid information when registering and logging in:

   (a) **Email Validation:** Ensure that users can only register using university-provided email addresses. The following validation enforces this format:

   ```
   validates :email, format: { with: /\Ast\d{6}@ (ait.
      ac.th|ait.asia)\z/, message: "must follow AIT
      format" }
   ```

   (b) **Password Validation:** Passwords must be at least six characters long and must not be blank:

```
config.password\_length = 6..128
```

(c) **Unique Email Validation:** Devise automatically ensures that no two users can register with the same email, preventing duplicate accounts.

(d) **Default Devise Modules:** The following Devise modules are included for user authentication and password recovery:

```
devise  :database_authenticatable, :registerable,
        :recoverable, :rememberable, :validatable
```

3. **User Registration and Database Schema**

The registration information for users is stored in the `users` table. The table is generated using a migration file provided by Devise, which is responsible for creating the necessary fields for authentication, password recovery, and other features.

(a) **email**: A string field that stores the user's email address. This is the unique identifier for each user in the system, and it cannot be null or left blank.

(b) **encrypted_password**: A string field that stores the encrypted version of the user's password. This field is mandatory and cannot be blank.

(c) **reset_password_token**: A string field used to store the token for password reset functionality. This token is used to verify the identity of users when they request to reset their password.

(d) **reset_password_sent_at**: A timestamp indicating when the password reset token was sent to the user. This helps ensure that password reset requests expire after a certain period of time.

(e) **remember_created_at**: A timestamp used for the "remember me" functionality, which allows users to stay logged in across sessions.

(f) **timestamps**: These fields include `created_at` and `updated_at`, which store the time of creation and the last update of the user's record, respectively.

Additionally, indexes are added to the `email` and `reset_password_token` fields to ensure uniqueness and improve query performance. As shown in Table 3.1, the user schema includes fields for storing user information securely.

Field	Data Type
id	[PK] bigint
email	character varying
encrypted_password	character varying
reset_password_token	character varying
reset_password_sent_at	timestamp without time zone (6)
remember_created_at	timestamp without time zone (6)
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)
role	character varying

Table 3.1 *Users Table Schema*

4. **Redirection After Login and Password Reset**
   (a) After users log in successfully, they are redirected to the `root_path`, where they can access their main page.
   (b) Users can reset their password by email. The system uses (SMTP) or Simple Mail Transfer Protocol to send password reset instructions. To enable this, the following configuration is added to `development.rb` and `production.rb`:

```
config.action_mailer.smtp_settings =
            {...SMTP configuration...}
```

5. **SMTP Configuration with Docker Compose** To ensure email notifications for password reset and other features, configure SMTP settings in the Docker Compose file. Add Gmail credentials as environment variables:

```
environment:
    - GMAIL_USERNAME=your-email@gmail.com
    - GMAIL_APP_PASSWORD=your-google-app-password
```

This configuration allows the Rails application to send password reset emails through Gmail's SMTP service.

6. **Front-End Views** A simple and user-friendly front-end is essential for the user authentication process. The following components are designed using HTML and CSS:
   (a) **Login and Sign-Up Forms:** These forms, generated by Devise, allow users to log in or sign up using their university email and password. Users who forget their password can click a link to reset it. The sign-up form is located at devise/registrations/new.html.erb, and the login form is located at `devise/sessions/new.html.erb`.
   (b) **Password Reset Form:** This form allows users to request password reset instructions by entering their registered email address. The password reset form is located at devise/passwords/new.html.erb, and the password edit form (for entering a new password after receiving reset instructions) is located at `devise/passwords/edit.html.erb`.
   (c) **Validation Errors:** Displays clear error messages when users attempt to sign up with invalid email formats or weak passwords. All preset error messages generated by Devise are configured in `app/config/locales/devise.en.yml`.

3.3.3 Landing Page

The landing page serves as the introductory section of the AIT Match platform, designed to provide essential information to both new and returning users. To implement the landing page, three controllers and views are generated to handle the different sections: Welcome Page, About Page, and Contact Page.

1. **Back-End Controllers:** To manage each section, the following controllers are generated:
   - **WelcomePageController**: Handles the logic and view for the welcome page.
   - **AboutPageController**: Manages the view displaying an overview of the AIT Match website.
   - **ContactPageController**: Displays the contact details of the developer for support or inquiries.
2. **Front-End Views:** These pages are implemented using `.html.erb` files in Rails, incorporating basic HTML and CSS to display information, with no complex back-end logic involved. The goal is to create simple yet informative views that effectively communicate the platform's purpose and make it easy for users to navigate and interact with the website.
   - **Welcome Page:** The welcome page greets users and provides a brief explanation of the AIT Match platform. It aims to engage visitors by showing the total number of users who have joined the website, encouraging new users to register and participate. The view is located `welcome_page/index.html.erb`.
   - **About Page:** The about page provides an overview of the AIT Match platform, its goals, and how it facilitates meaningful connections between students. This page is intended for users to understand the core purpose of the platform and its unique features. The view is located `about_page/index.html.erb`.
   - **Contact Page:** The contact page contains the contact information for the developer or support team. It provides users with the necessary details to reach out for help, report issues, or offer suggestions for improving the platform. The view is located in `contact_page/index.html.erb`.

3.3.4 Navigation Bar

To provide a seamless navigation experience for users, three distinct navbars are created for the AIT Match platform, depending on the user's role and login status: one for general users before logging in, one for regular users after logging in, and one specifically for admins after logging in.

— **Navigation Bar for Welcome Pages:** This navigation bar is designed for users who have not yet logged in or registered. It includes links to essential pages including "Home", "About", and "Contacts", allowing visitors to explore basic information about the platform. There are also clear "Log In" and "Sign Up" buttons to encourage new users to join the platform or log in to their existing accounts. The logo is displayed on the left to maintain brand visibility. This navigation bar is located in `layouts/_navbar_welcome_page.html.erb`.

— **Navigation Bar for Users:** Once a user is logged in, the main navigation bar is shown. This provides access to more personalized sections of the platform, such as "Home", "Profile", "Preferences", "Matches", "Messages", and "Reports". Depending on the user's profile and preferences, the links dynamically adjust—for instance, showing a "Create Profile" button if the user hasn't completed their profile yet. There's also a "Match Requests" icon to alert users of new match requests. The "Log Out" button is also prominently displayed for easy access. This navigation bar is located in `layouts/_navbar.html.erb`.

— **Navigation Bar for Admins:** Upon logging in, admins are presented with a dedicated navigation bar designed for managing users and handling reports. This navigation bar includes three main buttons: "Dashboard" for user management, "Reports" for overseeing user-reported issues, and a prominently displayed "Log Out" button for convenient access. The admin navigation bar is located in `layouts/_navbar_admin.html.erb`.

— **Footer:** At the bottom of all navbars, a simple footer is included across both the welcome pages and the logged-in user pages. The footer displays the text "AIT Match © 2024", ensuring consistency and providing a clean, professional touch to the website's branding throughout the platform.

3.3.5 Profiles

1. **Preset Data for Profile Creation:** To ensure consistency and ease of use, dropdown selections are provided for various fields in the profile creation form. These presets include options from tables such as Degree, Gender, Interest, MBTI, Program, Relationship, and School. A new `presets` table is created to store these options for each of them, meaning that every table has its own preset table for storing preset data from which users can select to create their profiles. Additionally, the Docker Compose file is configured to automatically seed the data every time the Docker environment is restarted.

   The dataset used for seeding this data is specified within the `seeds.rb` file, accessible at the following link: Seed Dataset Raw File.

2. **Profile Access Control:** To ensure users are authenticated before accessing any pages, a `'before_action :authenticate_user!'` filter is implemented in the application controller. This ensures that only logged-in users can navigate the website. Additionally, users are required to complete their profiles before accessing other pages on the platform. This is achieved through a custom filter `'before_action :check_profile_completion'`, which checks if the user's profile is completed. If a profile does not exist, the system redirects the user to the profile creation page with an alert message.

```
before_action :authenticate_user!
before_action :check_profile_completion

private
def check_profile_completion
  if current_user.profile.nil?
    flash[:alert] = "Please complete your profile
       before accessing this page."
    redirect_to new_profile_path
  end
end
```

3. **Profile Create:** After logging into the system for the first time, users are required to create a profile before accessing any other pages. The profile creation form includes fields for first name, last name, username, birthday, and other academic and personal details, such as MBTI, gender, degree, school, and program. There are also sections for interests and preferred relationship types, where users can select multiple options. Each field in the profile creation form is validated to ensure data correctness, such as unique usernames and mandatory fields like first name, last name, and email.

```
def create
    @profile = Profile.new(profile_params)
    @profile.user = current_user

    if @profile.save
      redirect_to @profile, notice: 'Profile was
          successfully created.'
    else
      # Render the new template with errors
      render :new, status: :unprocessable_entity
    end
end
```

4. **Profile New:** After logging in, users who have not yet created a profile are prompted to do so. If a user attempts to create a profile again after already having one, the system prevents this action, as each user is allowed only a single profile.

```
def new
    if current_user.profile.present?
      flash[:alert] = 'You already have a profile.
          Please edit your existing profile.'
      redirect_to profile_path(current_user.profile)
    else
      @profile = Profile.new
    end
end
```

5. **Profile Edit:** Once the profile is successfully created, users can access other sections of the platform. The profile can be viewed and edited at any time. Users can update their details, add new interests, or change preferences such as relationship types. The system ensures that each update is validated for correctness.

```
def edit
    @profile = current_user.profile
end
```

6. **Profile Update:** After editing the profile, the system updates the information in the database. A success message is shown, and the user is redirected back to their profile page.

```
def update
    @profile = current_user.profile

    if @profile.update(profile_params)
      redirect_to @profile, notice: 'Profile was
          successfully updated.'
    else
      # Render the edit template with errors
      render :edit, status: :unprocessable_entity
    end
end
```

7. **Profile Show:** Users can view their complete profile information in a dedicated "Profile" section, which includes personal details, academic background, interests, and preferences. Current users can view both their own profile and the profiles of other users.

```
def show
    @profile = Profile.find_by(id: params[:id])
    if @profile.nil?
      redirect_to profiles_path, alert: "Profile not
         found."
    else
      @interests = @profile.interests
      @user = @profile.user
    end
end
```

8. **Profile Delete:** Users can delete their profile, but to prevent accidental deletions, the system requires users to confirm their username before proceeding with deletion. The user must enter their username correctly, and if the entered username does not match, the system will display an error message and prevent the profile from being deleted.

```
def destroy
    if @profile.destroy
      redirect_to root_path, notice: 'Profile was
         successfully deleted.'
    else
      redirect_to profile_path(@profile), alert: '
         Something went wrong. Unable to delete profile
         .'
    end
end
```

```
<script>
document.addEventListener("DOMContentLoaded", function()
    {
    const form = document.getElementById("
       profileDeletionForm");

    form.addEventListener("submit", function(event) {
        const usernameInput = event.target.querySelector
           ("input[name='username']");
        const enteredUsername = usernameInput.value;
        const currentUsername = "<%= current_user.
           profile.user_name %>";

        if (enteredUsername !== currentUsername) {
            alert("Username does not match. Please type
               your correct username.");
            event.preventDefault(); // Prevent form
               submission
        }
    });
});
</script>
```

9. **Profile Index:** The system provides several key functionalities for profile management, including the ability to search profiles, filter by preferences, and view specific profiles. Users can edit their profile through an intuitive interface, and sorting options (A-Z, Z-A, or random) allow for flexibility in how profiles are displayed. Profiles can be viewed by others, and notifications are shown when matches or messages are received.

```
def index
    # Fetch all profiles, excluding the current user's
       own profile
    @profiles = Profile.where.not(user: current_user)

    # Initialize @filtered_profiles as all profiles
    @filtered_profiles = @profiles

    # Check if filtering should be disabled (via
       disable_filters parameter)
    if params[:disable_filters].present?
      # Skip applying filters
      @filtering_disabled = true
    else
      # Apply filters only if the user has set
         preferences and no sorting is applied
      if current_user.preference.present?
        apply_filters(current_user.preference)
        @filtering = true
    end
end
```

10. **Profile Database Schema:** The profile information for users is stored in the `profiles` table. The table is generated to hold important user profile details such as academic background, MBTI type, and profile picture. Below is an explanation of the fields in this table:
    (a) **id**: A primary key (bigint) that uniquely identifies each profile record.
    (b) **user_id**: A foreign key (bigint) linking each profile to the corresponding user.
    (c) **first_name**: A string field storing the user's first name.
    (d) **last_name**: A string field storing the user's last name.
    (e) **user_name**: A unique string field for the user's username, used for identifying them within the system.
    (f) **mbti_id**: A foreign key (bigint) linking to the MBTI type in the MBTI table.
    (g) **gender_id**: A foreign key (bigint) linking to the gender type in the Genders table.
    (h) **degree_id**: A foreign key (bigint) linking to the degree type in the Degrees table.
    (i) **school_id**: A foreign key (bigint) linking to the school information in the Schools table.
    (j) **program_id**: A foreign key (bigint) linking to the program information in the Programs table.
    (k) **educational_background**: A text field storing additional information about the user's educational background.
    (l) **created_at**: A timestamp storing when the profile was first created.
    (m) **updated_at**: A timestamp storing the most recent update time of the profile.
    (n) **birthday**: A date field storing the user's birthday.

    Additionally, indexes are added to some fields like `user_id` to optimize queries and enforce uniqueness.

Field	Data Type
id	[PK] bigint
user_id	bigint
first_name	character varying
last_name	character varying
user_name	character varying
mbti_id	bigint
gender_id	bigint
degree_id	bigint
school_id	bigint
program_id	bigint
educational_background	text
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)
birthday	date

Table 3.2 Profiles Table Schema

3.3.6 Active Storage for Profile Pictures

Active Storage is utilized in the AIT Match application to handle user-uploaded files, specifically for storing profile pictures. This feature allows users to easily upload and manage their profile images, enhancing the personalization aspect of the platform. According to Rails Team (2024), the implementation of Active Storage involves three key tables:

(a) **active_storage_attachments:** This table keeps track of the relationships between records (in this case, profiles) and the files attached to them. Each entry corresponds to an attachment, storing fields such as:

- **id**: A primary key that uniquely identifies each attachment.
- **name**: The name of the attachment.
- **record_type**: The type of the record that the file is associated with, in this case, `Profile`.
- **record_id**: The ID of the record that the attachment belongs to.
- **blob_id**: A foreign key linking to the blob containing the actual file data.
- **created_at**: A timestamp indicating when the attachment was created.

Field	Data Type
id	[PK] bigint
name	character varying
record_type	character varying
record_id	bigint
blob_id	bigint
created_at	timestamp without time zone (6)

Table 3.2 Active Storage Attachments Table

(b) **active_storage_blobs:** This table stores the actual files and their metadata. Each entry corresponds to a file uploaded by users and contains fields such as:

- **id**: A primary key for the blob.
- **key**: A unique identifier for the file.
- **filename**: The original name of the uploaded file.
- **content_type**: The MIME type of the file (e.g., image/png).

— **metadata**: Additional information about the file, such as analysis results.
— **byte_size**: The size of the file in bytes.
— **created_at**: A timestamp for when the file was created.

Field	Data Type
id	[PK] bigint
key	character varying
filename	character varying
content_type	character varying
metadata	text
byte_size	bigint
created_at	timestamp without time zone (6)

Table 3.3 Active Storage Blobs Table

(c) **active_storage_variant_records:** This table stores records for variants of the uploaded files, which can be used to create different versions (like thumbnails). It contains:
— **id**: A primary key for the variant record.
— **blob_id**: A foreign key linking to the corresponding blob.
— **variation_digest**: A character varying field that holds the identifier for the variant's configuration.

Field	Data Type
id	[PK] bigint
blob_id	bigint
variation_digest	character varying

Table 3.4 Active Storage Variant Records Table

The integration of Active Storage ensures that user profile pictures are managed efficiently while maintaining a high level of security and accessibility.

11. **Front-End Views:**
    - **Profile New:** After logging into the system, users are required to complete their profile on this page before accessing other parts of the platform. They must fill out all mandatory fields, including First Name, Last Name, unique Username, Birthday, MBTI Type, Gender, Degree, School, Program, Educational Background, and Profile Picture. Additionally, users have the option to select Interests (up to five) and Preferred Relationship types. This comprehensive profile setup ensures that users have sufficient information available on the platform, fostering meaningful connections based on shared interests and preferences. The view is located at `profiles/new.html.erb`.
    - **Profile Index:** This page displays all profiles in the system, allowing users to view, search, and sort profiles by username, or apply filters based on their preferences. The filtered profiles are shown on the index page, with selected preference details displayed on the sidebar, providing context for the filtered results. Each profile card presents user information and offers options to view the profile in detail or initiate a match request by clicking the buttons. The search bar, sorting dropdown, and filter preferences enable users to customize their browsing experience, enhancing navigation and interaction within platform. The view is located at `profiles/index.html.erb`.
    - **Profile Show:** This profile page is accessed when a user views their own profile or another user's profile. The system checks the profile ID to determine which profile to display. The page includes fields for profile picture, role, name, username, email, birthday, age, gender, MBTI type, degree, school, program, educational background, relationship preferences, and interests. Users can navigate back, report, or edit the profile based on permissions. This organized layout provides quick access to essential profile details. The view is located at `profiles/show.html.erb`.
    - **Profile Edit:** After initially setting up their profile, this page enables users to make edits to previously configured details. Additionally, users have the option to permanently delete their profile by confirming their username, adding a layer of security to the deletion process. This design offers a user-friendly interface for managing and updating profile information effectively. The view is located at `profiles/edit.html.erb`.

3.3.7 Preferences

1. **Preferences Preset Data:** Similar to the profile creation process, preset data is provided to facilitate preference selection. These presets allow users to select from predefined categories such as Degree, Gender, Interest, MBTI, Program, Relationship, and School, ensuring a standardized format for filtering profiles. Each preference category is associated with a dedicated table, allowing for the efficient storage and retrieval of user preferences.

   The dataset used for seeding this data is specified within the `seeds.rb` file, accessible at the following link: Seed Dataset Raw File.

2. **Preference Access Control:** To ensure users are authenticated before accessing any pages, a `'before_action :authenticate_user!'` filter is implemented in the application controller. This ensures that only logged-in users can navigate the website. Additionally, users are required to complete their profiles before accessing other pages on the platform. This is achieved through a custom filter `'before_action :check_profile_completion'`, which checks if the user's profile is completed. If a profile does not exist, the system redirects the user to the profile creation page with an alert message.

```
before_action :authenticate_user!
before_action :check_profile_completion

private
def check_profile_completion
  if current_user.profile.nil?
    flash[:alert] = "Please complete your profile
       before accessing this page."
    redirect_to new_profile_path
  end
end
```

3. **Preference Storage Structure:** User preferences are stored in a dedicated `preference` table, along with several many-to-many associated tables to allow multiple selections for each preference type. The associated tables include:
   - `users_preferred_degree`
   - `users_preferred_gender`
   - `users_preferred_interest`
   - `users_preferred_mbti`
   - `users_preferred_program`
   - `users_preferred_relationship`
   - `users_preferred_school`

   This structure allows users to select multiple options for each preference, such as indicating interest in more than one degree or relationship type. For example, the `UsersPreferredGender` model relates user preferences for gender types to the `preference` table and the `PreferredGender` table, following a similar structure across all related tables to accommodate multiple selections:

```
class UsersPreferredGender < ApplicationRecord
  belongs_to :preference
  belongs_to :preferred_gender
end
```

4. **Preference New:** After creating a profile, users can proceed to set their preferences. This process involves selecting values for various criteria, such as age range, gender, interests, and relationship types. Each preference selection is validated to ensure consistency and correctness. For example, users can only select options available within the preset data, ensuring the integrity of filterable criteria.

```
def new
    @preference = Preference.new
    load_filter_options
end
```

5. **Preference Create:** Once a user creates and saves their preferences, the preference data is stored in the corresponding database, accompanied by a notification message confirming the save action.

```
def create
    @preference = current_user.build_preference(
       preference_params)
    if @preference.save
      redirect_to profiles_path, notice: '
         Preferences were successfully set.'
    else
      load_filter_options
      render :new
    end
end
```

6. **Preference Edit:** Users can edit their preferences at any time, continuing from their previously set choices.

```
def edit
    load_filter_options
end
```

7. **Preference Update:** After users edit their preferences and save, the updated data is stored in the database, accompanied by a notification message.

```
def update
    if @preference.update(preference_params)
      redirect_to profiles_path, notice: '
         Preferences were successfully updated.'
    else
      load_filter_options
      render :edit
    end
end
```

8. **Filtering System:** The primary method responsible for filtering profiles according to user preferences is the `apply_filters` function within the `ProfilesController`. This function takes the `preference` object containing the user's selections and applies multiple filters based on the specified criteria.

   The filtering process in `apply_filters` includes the following steps:

   (a) Initial Setup with `distinct`: Ensures unique profiles by selecting `profiles.*` and calling `.distinct`.

   (b) Filtering Criteria:
   - Age Range: Filters profiles based on the selected age range.
   - Gender: Filters by the user's preferred gender selections.
   - Interests: Filters by matching interests between profiles.
   - Relationship Types: Filters by preferred relationship types.
   - MBTI: Filters based on personality compatibility.
   - School and Program: Filters profiles by preferred schools and programs.

   Each `if` block within `apply_filters` checks if a preference is present before applying the corresponding filter, ensuring that only active preferences are used. This method is called within the `index` action of the `ProfilesController` to display profiles filtered according to the user's preferences, offering a tailored profile viewing experience. Below is the code for the `apply_filters` method that performs profile filtering based on user preferences:

```
private
def apply_filters(preference)
  @filtered_profiles = @filtered_profiles.select('
      profiles.*').distinct

  # Filter by age range
  if preference.preferred_min_age.present? && preference
      .preferred_max_age.present?
    today = Date.today
    min_birthdate = today.years_ago(preference.
        preferred_max_age)
    max_birthdate = today.years_ago(preference.
        preferred_min_age)
    @filtered_profiles = @filtered_profiles.where("
        birthday >= ? AND birthday <= ?", min_birthdate,
        max_birthdate)
  end

  # Filter by gender preferences
  if preference.preferred_genders.any?
    @filtered_profiles = @filtered_profiles.joins(:
        gender).where(genders: { id: preference.
        preferred_gender_ids })
  end

  # Filter by interest preferences
  if preference.preferred_interests.any?
    @filtered_profiles = @filtered_profiles.joins(:
        interests).where(interests: { id: preference.
        preferred_interest_ids })
  end

  # Filter by relationship preferences
  if preference.preferred_relationships.any?
    @filtered_profiles = @filtered_profiles.joins(:
        relationships).where(relationships: { id:
        preference.preferred_relationship_ids })
  end

  # Filter by MBTI preferences
  if preference.preferred_mbti.any?
    @filtered_profiles = @filtered_profiles.joins(:mbti)
        .where(mbtis: { id: preference.preferred_mbti_ids
         })
  end

  ...
```

9. **Preferences Database Schema:** The preferences information for users is stored in the `preferences` table. This table holds user-defined filtering criteria, such as age range and other settings, which are used to customize the profile results displayed to each user. Below is an explanation of the fields in this table:
   (a) **id**: A primary key (bigint) that uniquely identifies each preferences record.
   (b) **user_id**: A foreign key (bigint) linking each preference set to the corresponding user.
   (c) **preferred_min_age**: An integer field specifying the minimum preferred age for matching profiles.
   (d) **preferred_max_age**: An integer field specifying the maximum preferred age for matching profiles.

   Additionally, indexes are added to fields like `user_id` to optimize query performance and ensure data integrity.

Field	Data Type
id	[PK] bigint
user_id	bigint
preferred_min_age	integer
preferred_max_age	integer
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.2 *Preferences Table Schema*

Field	Data Type
id	[PK] bigint
preference_id	bigint
preferred_degree_id	bigint
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.3 *Users Preferred Degrees Table Schema*

Field	Data Type
id	[PK] bigint
preference_id	bigint
preferred_gender_id	bigint
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.4 *Users Preferred Genders Table Schema*

Field	Data Type
id	[PK] bigint
preference_id	bigint
preferred_interest_id	bigint
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.5 *Users Preferred Interests Table Schema*

Field	Data Type
id	[PK] bigint
preference_id	bigint
preferred_mbti_id	bigint
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.6 *Users Preferred MBTIs Table Schema*

Field	Data Type
id	[PK] bigint
preference_id	bigint
preferred_program_id	bigint
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.7 *Users Preferred Programs Table Schema*

Field	Data Type
id	[PK] bigint
preference_id	bigint
preferred_relationship_id	bigint
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.8 *Users Preferred Relationships Table Schema*

Field	Data Type
id	[PK] bigint
preference_id	bigint
preferred_school_id	bigint
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.9 *Users Preferred Schools Table Schema*

The tables presented above store user-defined preferences that enable individuals on the AIT Match platform to filter potential matches based on specific criteria. Each table corresponds to a distinct aspect of user preferences, including degree, gender, interest, MBTI, program, relationship type, and school, and is linked to the main preferences table through foreign keys. This structure allows for flexible and customizable matching, ensuring that users can tailor their connections to align with both personal and academic interests.

10. **Front-End Views:**
    - **Preference New:** Upon logging into the system, users are prompted to complete their profile on this page before they can access other sections of the platform. After completing their profile setup, users are directed to create their initial preferences by selecting relevant fields, including minimum and maximum age range, preferred genders, MBTI types, degrees, schools, programs, and relationship types. Once the preferences are saved, the data is stored across all related tables in the database. This view is located at `preferences/new.html.erb`.
    - **Preference Edit:** After setting up their preferences, users can edit them at any time by modifying the same fields available in the initial preference setup. Once the preferences are updated, the new data is saved in the database, replacing the previous entries. This view is located at `preferences/edit.html.erb`.

3.3.8 Matches

Two profiles can initiate a match and engage in messaging if both agree to connect. The matching process involves three statuses: pending, accepted, and declined. A user (referred to as the requestor) can send a match request to another user (the receiver) and await their response. If the receiver accepts, the profiles are matched and can start chatting; otherwise, the request is marked as declined.

1. **Matches Access Control:** To ensure users are authenticated before accessing any pages, a `'before_action :authenticate_user!'` filter is implemented in the application controller. This ensures that only logged-in users can navigate the website. Additionally, users are required to complete their profiles before accessing other pages on the platform. This is achieved through a custom filter `'before_action :check_profile_completion'`, which checks if the user's profile is completed. If a profile does not exist, the system redirects the user to the profile creation page with an alert message.

```
before_action :authenticate_user!
before_action :check_profile_completion

private
def check_profile_completion
  if current_user.profile.nil?
    flash[:alert] = "Please complete your profile
       before accessing this page."
    redirect_to new_profile_path
  end
end
```

2. **Match Create:** As the requestor, a user can send a match request to another user (the receiver) only if the status of a previous request between them is neither "accepted" nor "pending." When a match request is created, the system displays a notice to the requestor indicating whether the request was successfully sent or if it failed. The create method first identifies the receiver's profile using the profile_id parameter. It then checks for any existing match requests with a "declined" status between the sender and receiver; if such a request exists, it updates the status to "pending," allowing the request to be resent. If no declined match exists, the method creates a new match request with a "pending" status. Finally, it attempts to save the request, redirecting to the receiver's profile with a success message upon success or showing an error alert if the request fails.

```
def create
    # Find the receiver profile using the ID passed from
        the button
    @receiver = Profile.find(params[:profile_id])

    # Check if a match already exists between the
       current user and the receiver in either direction
    existing_match = Match.find_by(
      "(requestor_id = :requestor_id AND receiver_id = :
         receiver_id) OR
       (requestor_id = :receiver_id AND receiver_id = :
          requestor_id)",
      requestor_id: current_user.profile.id, receiver_id
         : @receiver.id
    )

    if existing_match
      if existing_match.status == 'declined'
        # Update the existing declined match back to
           pending
        existing_match.update(status: 'pending')
        redirect_to profile_path(@receiver), notice: '
           Match request sent again.'
      elsif existing_match.status == 'accepted'
        # Prevent creating a duplicate match if already
           matched
        redirect_to profile_path(@receiver), alert: 'You
            are already matched with this user.'
      else
        # If a pending match already exists, do not
           create a new one
        redirect_to profile_path(@receiver), alert: 'You
            already have a pending match request with
           this user.'
      end
    else
      # Create a new match request if no match exists in
          either direction
      @match = current_user.profile.sent_matches.build(
         receiver: @receiver, status: 'pending')

      if @match.save
        redirect_to profile_path(@receiver), notice: '
           Match request sent.'
      else
        redirect_to profile_path(@receiver), alert: '
           Unable to send match request.'
      end
    end
end
```

3. **Match Update:** The update method allows the receiver of a pending match request to accept or decline it. After verifying that the current user is indeed the intended receiver, the method checks the desired status. If the request is accepted or declined, it updates the match status accordingly and redirects the user to the match requests page with a corresponding confirmation message. If the user is not authorized to modify the match, an alert is displayed.

```
def update
@match = Match.find(params[:id])

if @match.receiver == current_user.profile && @match.
    status == 'pending'
  if params[:status] == 'accepted'
    @match.update(status: 'accepted')
    redirect_to matches_requests_path, notice: 'Match
        accepted.'
  elsif params[:status] == 'declined'
    @match.update(status: 'declined')
    redirect_to matches_requests_path, notice: 'Match
        declined.'
  end
else
  redirect_to matches_requests_path, alert: 'You are not
      authorized to modify this match.'
end
end
```

4. **Match Accept:** The accept method enables the receiver of a pending match request to accept it and initiates a conversation between the requestor and receiver upon acceptance. First, the method verifies that the current user is the designated receiver and that the match status is pending. If these conditions are met, the match status is updated to "accepted," and a new conversation is created to facilitate communication between the matched profiles. The user is then redirected to the match requests page with a success message. If authorization fails, an alert is displayed instead.

```
def accept
    @match = Match.find(params[:id])
    if @match.receiver == current_user.profile && @match
       .status == 'pending'
      @match.update(status: 'accepted')
      # Create a conversation after the match is
         accepted
      Conversation.between(@match.requestor.id, @match.
         receiver.id, @match)
      redirect_to matches_requests_path, notice: 'Match
         accepted. Conversation created!'
    else
      redirect_to matches_requests_path, alert: 'You are
          not authorized to modify this match.'
    end
end
```

5. **Match Decline:** The decline method allows the receiver of a pending match request to reject it. The method first confirms that the current user is the intended receiver and that the match status is pending. If these conditions are satisfied, the match status is updated to "declined," and the user is redirected to the match requests page with a notification indicating the decline. If authorization checks fail, an alert message is displayed to inform the user of insufficient permission to modify the match.

```
def decline
    @match = Match.find(params[:id])

    if @match.receiver == current_user.profile && @match
        .status == 'pending'
      @match.update(status: 'declined')
      redirect_to matches_requests_path, notice: 'Match
          declined.'
    else
      redirect_to matches_requests_path, alert: 'You are
          not authorized to modify this match.'
    end
end
```

6. **Match Index:** The index method retrieves and organizes match data for the current user. It gathers all pending match requests received by the user's profile, as well as profiles that have already been successfully matched. This data is then available to display pending requests and established matches on the match requests page, providing users with a clear overview of their match interactions.

```
def index
    @matches = current_user.profile.received_matches.
        where(status: 'pending')
    @matched_profiles = current_user.profile.
        matched_profiles
end
```

7. **Match Requests:** The requests method identifies all pending match requests for the current user, specifically where the user is the receiver of the requests. By accessing the user's profile, it retrieves matches with a "pending" status, storing them in @matches. Additionally, it calculates the total count of pending requests, storing this number in @match_requests_count, allowing for a quick overview of the user's current match requests awaiting response.

```
def requests
    # Get the current user's profile
    user_profile = current_user.profile

    # Find all pending match requests where the user is
       the receiver
    @matches = Match.where(receiver_id: user_profile.id,
        status: 'pending')
    @match_requests_count = @matches.count # Count
       pending requests
end
```

8. **Matched Profiles:** The matched_profiles method retrieves all accepted matches for the current user by first identifying the user's profile. It searches for matches where the user is either the requestor or receiver with a status of "accepted." For each accepted match, it gathers the profile of the matched user (either the requestor or receiver, excluding the current user), storing these profiles in @matched_profiles. This approach provides the user with a list of profiles they have successfully matched with on the platform.

```
def matched_profiles
    # Get the current user's profile
    user_profile = current_user.profile

    # Find all matches where the user is either the
       requestor or the receiver and the status is '
       accepted'
    @matches = Match.where("(requestor_id = ? OR
       receiver_id = ?) AND status = ?", user_profile.id
       , user_profile.id, 'accepted')

    # Retrieve the matched profiles (either requestor or
        receiver, excluding the current user's profile)
    @matched_profiles = @matches.map do |match|
      match.requestor_id == user_profile.id ? match.
          receiver : match.requestor
    end
end
```

9. **Match Delete:** The destroy method allows a user to delete an existing match if they are either the requestor or receiver in that match. The method first retrieves the match based on the provided ID and verifies that the current user is part of the match. If authorized, it deletes the match and redirects the user to the matched profiles page with a success message, allowing them the option to send a new match request in the future. If unauthorized, the user is redirected with an alert indicating they do not have permission to delete the match.

```
def destroy
    @match = Match.find(params[:id])

    # Ensure the current user is part of the match
    if @match.requestor == current_user.profile ||
       @match.receiver == current_user.profile
      @match.destroy
      redirect_to matches_matched_profiles_path, notice:
          "Match deleted successfully. You can send a
         match request again."
    else
      redirect_to matches_matched_profiles_path, alert:
         "You are not authorized to delete this match."
    end
end
```

10. **Matches Database Schema:** The matches table records information about user-initiated match requests. Each record represents an individual match request between two users, identified by the fields requestor_id (the user who sends the match request) and receiver_id (the user receiving the request). The status field tracks the state of the match, which could be "pending", "accepted", or "declined". This schema helps manage and track user interactions on the platform by indicating if a match request is active, accepted, or dismissed.
    (a) **id**: A primary key (bigint) that uniquely identifies each match record.
    (b) **requestor_id**: A foreign key (bigint) linking to the profile of the user who sent the match request.
    (c) **receiver_id**: A foreign key (bigint) linking to the profile of the user who received the match request.
    (d) **status**: A character varying field that indicates the current status of the match (e.g., "pending", "accepted", or "declined").

    Additionally, fields like created_at and updated_at provide timestamps for when each match request was created and last updated, aiding in tracking the progression of each match interaction.

Field	Data Type
id	[PK] bigint
requestor_id	bigint
receiver_id	bigint
status	character varying
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.10 *Matches Table Schema*

11. **Front-End Views:**
    - **Matched Profiles:** This page dynamically generates a list of matched profiles for the logged-in user. It first checks if there are any matched profiles available. For each matched profile, it retrieves the corresponding match record with an accepted status. Each profile is displayed in a "profile card" format, showing the username and email. Users can choose to delete the match, view the profile, or start a conversation with the matched profile. If no matches are found, a message indicating "No matches yet" is displayed. This view is located at `matches/matched_profiles.html.erb`.
    - **Match Requests:** This page displays a list of pending match requests for the current user, if any are available. Each match request is shown in a "request card" format, displaying the requestor's username and email. For each pending request, the user is provided with "Accept" and "Decline" buttons, allowing them to either approve or reject the match. The number of pending match requests is also shown in the navigation bar as a number to notify the user. If there are no pending match requests, a message indicating "No pending match requests" is displayed. This view is located at `matches/matched_requests.html.erb`.

3.3.9 Chat

The chat system in this application is designed to facilitate communication between users who have matched. Once two profiles are matched, they gain access to a private chat where they can exchange messages. Only matched profiles are allowed to engage in conversations, ensuring privacy and relevance in interactions.

1. **Chat System:** The chat system comprises two main controllers: The ConversationsController and the MessagesController. The ConversationsController manages the creation and display of conversations between matched users. A conversation is only established between profiles that have mutually matched, and each conversation is restricted to the two involved participants. The MessagesController, on the other hand, handles the messages exchanged within an established conversation, ensuring messages are delivered between the two profiles only.
2. **Conversations and Messages Access Control:** To ensure users are authenticated before accessing any pages, a `'before_action :authenticate_user!'` filter is implemented in the application controller. This ensures that only logged-in users can navigate the website. Additionally, users are required to complete their profiles before accessing other pages on the platform. This is achieved through a custom filter `'before_action :check_profile_completion'`, which checks if the user's profile is completed. If a profile does not exist, the system redirects the user to the profile creation page with an alert message.

```
before_action :authenticate_user!
before_action :check_profile_completion

private
def check_profile_completion
  if current_user.profile.nil?
    flash[:alert] = "Please complete your profile
       before accessing this page."
    redirect_to new_profile_path
  end
end
```

3. **Conversation Index:** This action retrieves all conversations where the current user's profile is either the sender or receiver, enabling the user to view a list of their ongoing conversations.

```
def index
    # Fetch all conversations where the current user's
       profile is either the sender or receiver
    @conversations = Conversation.where("sender_id = ?
       OR receiver_id = ?", current_user.profile.id,
       current_user.profile.id)
end
```

4. **Conversation Create:** This action creates a new conversation between the current user and the specified receiver profile if no existing conversation is found. If a conversation already exists, the action redirects the user to the existing conversation's page.

```
def create
    receiver = Profile.find(params[:receiver_id])

    # Find the conversation between the current user and
       the receiver or create one
    @conversation = Conversation.between(current_user.
       profile.id, receiver.id)

    if @conversation.nil?  # If no existing conversation
       found
      @conversation = Conversation.new(sender_id:
         current_user.profile.id, receiver_id: receiver.
         id)

      if @conversation.save
        # Successfully created new conversation
      else
        redirect_to profiles_path, alert: "Unable to
           create conversation."
        return
      end
    end

    # Redirect to the show page of the conversation
    redirect_to conversation_path(@conversation) #
       Ensure @conversation has an id
end
```

5. **Conversation Show:** This action displays a conversation, allowing the participants to view the messages exchanged. Only the sender or receiver of the conversation can access it. If either participant's profile has been deleted, the conversation becomes unavailable.

```
def show
    @conversation = Conversation.find_by(id: params[:id
       ])

    # Check if the conversation exists and the
       participants still exist
    if @conversation.nil? || @conversation.sender.nil?
       || @conversation.receiver.nil?
      redirect_to conversations_path, alert: "This
         conversation is no longer available."
      return
    end

    # Ensure that only participants in the conversation
       can view it
    if @conversation.sender == current_user.profile ||
       @conversation.receiver == current_user.profile
      @messages = @conversation.messages.order(
         created_at: :asc)
      @message = Message.new
    else
      redirect_to conversations_path, alert: "You are
         not authorized to view this conversation."
    end
end
```

6. **Conversation Delete:** This action enables users to delete conversations, removing both the conversation and its messages from view.

```
def destroy
    @conversation = Conversation.find(params[:id])
    if @conversation.destroy
      redirect_to conversations_path, notice: '
          Conversation deleted successfully.'
    else
      redirect_to conversations_path, alert: 'Unable to
          delete the conversation.'
    end
  end
```

7. **Message Index:** This action retrieves all messages within a specific conversation and renders them in JSON format. This data includes each message and the sender's username, making it suitable for use in real-time chat displays.

```
def index
    @conversation = Conversation.find(params[:
        conversation_id])
    @messages = @conversation.messages.order(created_at:
        :asc)

    render json: @messages.to_json(include: { profile: {
        only: :user_name } })
end
```

8. **Message Create:** This action enables users to send a new message within a conversation. Before saving, the controller verifies that the sender or receiver is the current user, ensuring that only participants in the conversation can send messages. Once a message is saved, it is broadcast via ActionCable for real-time updates, displaying the sender's username, message content, and timestamp in Bangkok time format.

```
def create
    @conversation = Conversation.find(params[:
        conversation_id])

    if @conversation.sender == current_user.profile ||
        @conversation.receiver == current_user.profile
      @message = @conversation.messages.build(
          message_params)
      @message.profile = current_user.profile

      if @message.save
        # In MessagesController
        ActionCable.server.broadcast "conversation_#{
            @conversation.id}", {
          id: @message.id,  # Add this line to send the
              message ID
          profile: @message.profile.user_name,
          message: @message.body,
          created_at: @message.created_at.in_time_zone("
              Bangkok").strftime("%H:%M, %d %b %Y")
        }
        redirect_to conversation_path(@conversation)
      else
        redirect_to conversation_path(@conversation),
            alert: "Message could not be sent."
      end
    else
      redirect_to conversations_path, alert: "You are
          not authorized to send messages in this
          conversation."
    end
end
```

9. **Real-Time Chat Implementation Using Rails ActionCable and WebSockets:** The real-time chat feature within the system is powered by Rails ActionCable (Rails Team, 2015), utilizing WebSocket connections (IETF, 2011) and Redis (Salvatore Sanfilippo and Redis Labs, 2009) to enable live, instant communication between matched users. This implementation includes a JavaScript function that initializes the chat functionality upon page load, specifically when the Turbo frame loads, ensuring that the chat system is correctly set up each time the user navigates to the chat page. The script identifies the active conversation by locating the conversation's unique ID on the page, which is then used to subscribe to a specific ConversationChannel for that session.

   With this subscription, the system maintains a persistent WebSocket connection, enabling two-way communication between the client and server. Redis acts as a pub-sub server, efficiently managing message broadcasts by distributing each message across all subscribed WebSocket channels. Through callback methods in the JavaScript, key WebSocket events are managed, including connection establishment, disconnection, and message receipt. When a new message is broadcast, the chat display is dynamically updated by appending the new message with the sender's username and timestamp, while avoiding duplicates. If no conversation ID is found, the system gracefully handles this by logging an error message, maintaining stable functionality. This integration of JavaScript, ActionCable, Redis, and WebSocket provides a seamless, interactive chat experience, allowing matched users to communicate instantly within the application.

10. **Conversations Database Schema:** The conversations table records the initiation of conversations between users who are matched. Each record represents a unique conversation between two profiles, identified by sender_id (the user who initiates the conversation) and receiver_id (the user who receives the conversation request). The table also includes a match_id field to associate the conversation with a specific match request, facilitating the connection between matched users.
    (a) **id**: A primary key (bigint) that uniquely identifies each conversation record.
    (b) **sender_id**: A foreign key (bigint) linking to the profile of the user who initiated the conversation.
    (c) **receiver_id**: A foreign key (bigint) linking to the profile of the user receiving the conversation.
    (d) **match_id**: A foreign key (bigint) linking to the match that initiated this conversation.

    Additionally, fields like created_at and updated_at provide timestamps for when each conversation was initiated and last updated, aiding in tracking the timing of user interactions.

Field	**Data Type**
id	[PK] bigint
sender_id	bigint
receiver_id	bigint
match_id	bigint
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.11 *Conversations Table Schema*

11. **Messages Database Schema:** The messages table stores individual messages exchanged between users within a conversation. Each record represents a single message, with fields for conversation_id (linking to the relevant conversation), profile_id (identifying the sender of the message), and body (containing the text of the message). This schema enables real-time communication by recording each interaction within the context of a specific conversation.
    (a) **id**: A primary key (bigint) that uniquely identifies each message record.
    (b) **conversation_id**: A foreign key (bigint) linking to the conversation to which this message belongs.
    (c) **profile_id**: A foreign key (bigint) linking to the profile of the user who sent the message.
    (d) **body**: A text field containing the content of the message.

    Additionally, created_at and updated_at fields provide timestamps for when each message was sent and last modified, supporting chronological tracking of conversation flows.

Field	Data Type
id	[PK] bigint
conversation_id	bigint
profile_id	bigint
body	text
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.12 *Messages Table Schema*

12. **Front-End Views:**
    - **Conversation Index:** This page provides the structure and functionality for displaying a list of active conversations for the user within the AIT Match application. Each conversation is shown as an item in a list, where the other participant's username and the most recent message in the conversation are displayed. If a conversation has no messages, a placeholder text "No messages shown" is displayed instead. For each conversation, two buttons are available: "Delete Chat" and "Chat." The "Delete Chat" button allows the user to delete the entire conversation, while the "Chat" button navigates to the conversation page, where the user can continue messaging. If there are no active conversations, a message indicating "No conversations yet" is displayed. This view enables users to manage their active conversations efficiently by providing quick access to chat and delete functionalities. The view is located at `conversations/index.html.erb`.
    - **Conversation Show:** This page provides a real-time messaging interface for the AIT Match application, enabling users to exchange messages in an ongoing conversation. The header displays the conversation partner's username, and all messages appear in a scrollable container with the sender's name, message content, and timestamp in Bangkok time. The input area allows users to compose new messages. A JavaScript script establishes a WebSocket connection via ActionCable, subscribing to a ConversationChannel unique to the active conversation. When a new message is broadcast, it updates the chat display in real-time, with duplicate messages avoided and the window auto-scrolling to the latest entry. This setup, located at `conversations/show.html.erb`, enhances responsiveness and ensures seamless user communication.
    - **Message Partial View:** The `_message.html.erb` partial view renders each individual message in the AIT Match chat interface, displaying the sender's username, message content, and timestamp. This partial enables consistent and reusable message formatting across the conversation interface. Structuring messages as a partial enhances code modularity and maintainability, ensuring that updates to message styling are reflected across all instances, contributing to a cohesive user experience in the real-time chat feature.

3.3.10 Report

On the user side, the `ReportsController` enables users to report other profiles for inappropriate behavior. A user can create a report by navigating to the report form, where they specify the reason for reporting. The `new` action initializes a report instance for the reported profile, and the `create` action saves the report to the database with a status of "pending." The report includes details such as the reporter's profile, the reported profile, and the reason for the report. Users can also view a list of reports they have submitted (`index` action) and access individual reports (`show` action) for reference.

1. **Report Access Control:** To ensure users are authenticated before accessing any pages, a `'before_action :authenticate_user!'` filter is implemented in the application controller. This ensures that only logged-in users can navigate the website. Additionally, users are required to complete their profiles before accessing other pages on the platform. This is achieved through a custom filter `'before_action :check_profile_completion'`, which checks if the user's profile is completed. If a profile does not exist, the system redirects the user to the profile creation page with an alert message.

```
before_action :authenticate_user!
before_action :check_profile_completion

private
def check_profile_completion
  if current_user.profile.nil?
    flash[:alert] = "Please complete your profile
       before accessing this page."
    redirect_to new_profile_path
  end
end
```

2. **Report Index:** The index action retrieves all reports that the current user has submitted. This action filters the reports based on the current user's profile, showing only those reports where the user is listed as the reporter. The list of reports is stored in the @reports instance variable, making it available for display in the view. This allows users to track and review the reports they have submitted regarding other profiles' behavior.

```
def index
    # Retrieve reports where the current user is the
        reporter
    @reports = current_user.profile.
        reports_as_reporter
end
```

3. **Report Show:** The show action displays a specific report submitted by the user. It finds the report by matching the report ID passed in the request parameters with the reports associated with the current user's profile. This ensures that users can view detailed information about each report they have submitted, including the status and any specific details that might be shown in the view, while keeping access restricted to only their own reports.

```
def show
    @report = current_user.profile.
        reports_as_reporter.find(params[:id])
end
```

4. **Report New:** The new action initializes a new report instance and identifies the profile being reported. It creates a blank Report object assigned to the @report variable and locates the @reported_profile based on the reported_profile_id provided in the request parameters. This setup enables the new view to render a report form where users can specify the reason for reporting a particular profile.

```
def new
    @report = Report.new
    @reported_profile = Profile.find(params[:
        reported_profile_id])
end
```

5. **Report Create:** The create action is responsible for saving a new report to the database. It first locates the profile being reported and then builds a new report linked to the current user's profile (as the reporter). The report parameters, including the reason, are merged with the reported profile's details and given a default status of "pending." If the report saves successfully, the user is redirected to the reported profile's page with a success message. Otherwise, the new form is re-rendered, allowing the user to correct any issues with their submission.

```
def create
    @reported_profile = Profile.find(params[:
       reported_profile_id])
    @report = current_user.profile.
       reports_as_reporter.build(report_params.merge
       (reported_profile: @reported_profile, status:
        'pending'))

    if @report.save
      redirect_to profile_path(@reported_profile),
         notice: 'Report submitted successfully.'
    else
      render :new
    end
end
```

6. **Reports Database Schema:** The reports table stores data on user-generated reports, where each record documents a specific complaint submitted by one user against another. This schema captures essential fields like the identities of the reporting user and the reported user, along with a reason for the report. The status field records the current state of the report, facilitating efficient management and tracking of reported issues.
   (a) **id**: A primary key (bigint) that uniquely identifies each report.
   (b) **reporter_profile_id**: A foreign key (bigint) linking to the profile of the user who submitted the report.
   (c) **reported_profile_id**: A foreign key (bigint) linking to the profile of the user being reported.
   (d) **reason**: A character varying field that specifies the reason provided by the reporting user for submitting the report.
   (e) **status**: A character varying field indicating the current state of the report.

   Additionally, fields like created_at and updated_at provide timestamps for when each report was created and last updated, supporting an audit trail of the reporting process.

Field	**Data Type**
id	[PK] bigint
reporter_profile_id	bigint
reported_profile_id	bigint
reason	character varying
status	character varying
created_at	timestamp without time zone (6)
updated_at	timestamp without time zone (6)

Table 3.13 *Reports Table Schema*

7. **Front-End Views:**
   - **Report New:** This view is for reporting allows users to submit a report by specifying the reason for their complaint against another user. The form includes a text area for the user to describe the issue, helping to provide context for the report. Upon submission, the report is recorded with the relevant details and marked as “pending” for administrative review. This reporting functionality enhances the platform’s safety measures by enabling users to flag inappropriate behavior for further investigation. The view is located at `reports/new.html.erb`.
   - **Conversation Index:** This page page provides users with an organized interface to review the reports they have submitted against other profiles. Users can filter and sort their reports using a search bar, which allows searches by username, and sorting buttons to arrange reports by date. Each report entry displays the reported user’s username (linked to their profile), the current status of the report, and the submission date, which is formatted to Bangkok time. A "View Details" button enables users to examine the specifics of each report; however, users do not have permissions to delete or modify the reports. They must wait for the admin to review and update the status, ensuring a structured and secure process for managing reports within the platform. The view is located at `reports/index.html.erb`.
   - **Report Show:** This page allows users to view detailed information about a specific report they have submitted. This page displays essential details, including the username of the reported user (with a link to their profile), the reason for the report, the current status of the report, and the date and time it was submitted, formatted in Bangkok time. Users can return to the list of their submitted reports by clicking the “Back to My Reports” button, conveniently placed at the bottom of the page. This interface provides a straightforward, user-friendly layout for reviewing individual report submissions, ensuring users can monitor the status and details of their reports without any ability to alter or delete them. This separation of permissions emphasizes the platform’s commitment to a structured and secure reporting system, where reports are managed by administrators. The view is located at `reports/show.html.erb`.

3.3.11 Admin

1. **Admin System Overview:** The administrative system in this application is vital for platform security and creating a safe environment. It comprises two main components: the report management system and the user management system, managed by the `Admin::ReportsController` and `Admin::UsersController`. These components enable administrators to monitor user interactions and manage accounts, thus reinforcing platform security.
2. **report management system:** The report management system empowers administrators to review and respond to user-submitted reports of inappropriate behavior or community guideline violations. Through `Admin::ReportsController`, administrators can view all reports, check individual details, update report statuses, and delete reports post-review. This capability allows the administration to promptly address user concerns, helping to cultivate a safe environment by responding to reported behaviors within the community.
   (a) **Admin Report Index:** Retrieves all reports from the database and displays them in a list view. This provides the admin with an overview of all reports that users have submitted for review.

```
def index
  @reports = Report.all
end
```

   (b) **Admin Report Show:** Displays the details of a specific report. By using Report.find(params[:id]), the system retrieves the report based on its ID, allowing the admin to review the reason and context provided by the user.

```
def show
  @report = Report.find(params[:id])
end
```

(c) **Admin Report Update:** Updates the status of a specific report. The admin can modify the report's status to reflect its current state. If the update is successful, a success message is displayed; otherwise, an error message appears.

```
def update
  @report = Report.find(params[:id])
  if @report.update(report_params)
    redirect_to admin_reports_path, notice: "
       Report status updated."
  else
    render :show, alert: "Failed to update
       report status."
  end
end
```

(d) **Admin Report Delete:** Deletes a specific report from the system. After finding the report by its ID, the destroy action removes it from the database and redirects to the report list with a confirmation message.

```
def destroy
  @report = Report.find(params[:id])
  @report.destroy
  redirect_to admin_reports_path, notice: '
     Report was successfully deleted.'
end
```

3. **User management system:** The user management system, facilitated by `Admin::UsersController`, provides administrators with control over user accounts, including viewing, editing, and, when necessary, deleting profiles. For instance, users flagged multiple times or violating platform policies may be removed to maintain community safety. Additionally, to avoid oversight issues, administrators cannot delete their own accounts, ensuring that the platform's administrative functions remain intact.

   (a) **Admin User Index:** Lists all user profiles, excluding the currently logged-in admin's own profile, to prevent the admin from accidentally modifying their own account. This allows the admin to manage the user base effectively.

```
def index
  # Display all users except the currently
     logged-in admin
  @users = User.where.not(id: current_user.id)
end
```

(b) **Admin User Show:** In the administrative view, the interface resembles the profile display accessible to regular users, enabling the admin to view all user profiles in the system. However, administrative privileges are limited; the admin cannot edit profiles or submit reports. The only permitted action is deleting user profiles, ensuring the admin's role focuses on maintaining platform integrity without interfering in user interactions or content.

```
def show
    @profile = Profile.find_by(id: params[:id])
    if @profile.nil?
      redirect_to profiles_path, alert: "Profile not
          found."
    else
      @interests = @profile.interests
      @user = @profile.user
    end
end
```

(c) **Admin User Delete:** Deletes a user account from the system. This action prevents the admin from deleting their own account to avoid accidental loss of administrative access. If the user deletion is successful, the admin is redirected to the user list with a confirmation message.

```
    def destroy
      @user = User.find(params[:id])

      # Prevent admin from deleting their own
          account
      if @user == current_user
        redirect_to admin_users_path, alert: "You
            cannot delete your own account."
      else
        @user.destroy
        redirect_to admin_users_path, notice: "User
            profile deleted successfully."
      end
    end
```

4. **Admin Database Schema:** The Admin Database Schema in this application is designed to integrate seamlessly with both the User Management and Report Management systems, utilizing the same Users and Reports tables accessible by the user side. By sharing these tables, the platform ensures data consistency and simplifies database design, allowing both users and administrators to access relevant data through a unified structure. Administrators, however, are granted additional privileges within this schema, enabling them to manage user accounts and report entries directly. For instance, while users can view or submit reports, administrators can take corrective actions such as deleting records in these tables. This approach supports streamlined data management while providing the necessary administrative oversight to maintain platform integrity and user safety.

5. **Front-End Views:**
   - **Admin Report Index:** This page provides an interface for administrators to manage user reports within the platform. At the top, administrators can search and filter reports using a custom search bar that allows filtering by both the reporter's and the reported user's usernames. Additionally, two buttons are provided to sort reports by the date they were submitted, either from newest to oldest or vice versa. The main table below displays all reports with relevant information, including the reporter's username, the reported user's username (with a link to view details), the status of each report, the date it was reported, and options for actions. Administrators have the ability to view the detailed report page through the "View" button, and if necessary, can delete any report by using the "Delete" button, which is accompanied by a confirmation prompt to prevent accidental deletions. This page thus serves as a comprehensive tool for administrators to efficiently monitor, review, and manage user reports, thereby helping to maintain a safe and respectful platform environment. The view is located at `admin/reports/index.html.erb`.
   - **Admin Report Show:** This page serves as the "Report Details" view for the admin, allowing them to review and manage individual reports submitted by users. It displays essential information such as the reporter's username, the reported user's username, the reason for the report, the current status, and the date and time of submission. Additionally, an editable status field is provided, enabling the admin to update the report's status to either "Pending" or "Reviewed" based on their review process. At the bottom of the page, two buttons offer the admin options to either navigate back to the reports list or submit the updated status, ensuring efficient navigation and streamlined management of user reports within the platform. The view is located at `admin/reports/show.html.erb`.

— **Admin User Index:** This page serves as the User Management interface for administrators, providing them with essential tools to search, sort, and manage user accounts within the platform. The search functionality allows admins to quickly locate users by their username, while the sort buttons enable the admin to organize the user list alphabetically, either in ascending or descending order. The user table displays core user information such as Username, Email, Role, and options for viewing profiles or deleting accounts. Admins can view any user's profile through a direct link if a profile is associated with the user. For account removal, a delete button is provided next to each user (excluding the admin's own account), and deletion is confirmed by prompting the admin to type the user's username, adding an additional layer of security. This interface streamlines user management by offering easy access to user information and controls, ensuring efficient oversight of the platform's user base. The view is located at `admin/users/index.html.erb`.

— **Admin User Show:** In the administrative interface, the "Profile" section allows the admin to view the complete profile information of each user within the system. This section includes detailed personal information, academic background, interests, and preferences, providing the admin with a comprehensive overview of each user's profile. While admins can view these profiles to ensure compliance with platform standards, they are restricted from editing any information or interacting with profiles beyond the view access. This structured access for administrators is essential for monitoring and maintaining the integrity of user profiles on the platform. The view is located at `profiles/show.html.erb`.

In summary, the report and user management systems collectively provide a robust framework supporting platform security and effective operation. The report management system enables the administration to mitigate disruptive behaviors, while the user management system ensures adherence to platform standards. Together, these systems protect the user community, enhance security, and maintain application integrity.

CHAPTER 4
CONCLUSION

The AIT Match platform represents a significant advancement in creating a secure, personalized social network designed specifically for university students. By integrating a robust back-end stack that includes Rails, Ruby, PostgreSQL, HTML, CSS, JavaScript, and Docker, this platform enables efficient and scalable real-time interactions. Core features such as personalized profiles, academic filters, interest-based connections, and real-time messaging cater directly to the unique needs of students at the Asian Institute of Technology (AIT), facilitating meaningful connections in a trusted and secure environment.

AIT Match transcends conventional dating platforms by emphasizing both academic and social engagement, offering users a space to connect with peers who share similar academic programs, interests, or career aspirations. The platform's matching algorithm, privacy-focused infrastructure, and verified user profiles ensure that interactions are safe and authentic, fostering a strong sense of community within AIT. Users are required to log in with their university email, adding an extra layer of security by verifying users' affiliation with AIT.

The platform also includes a report system within the admin framework, allowing users to report inappropriate behavior by submitting a report form that includes the reason for reporting. Administrators have the authority to review these reports and, if necessary, delete the reported profiles from the platform, enhancing security and ensuring a respectful environment by removing problematic users.

The platform's deployment on AIT's CSIM (Computer Science & Information Management) server leverages institutional support for stable and secure access, while Docker and DockerHub facilitate scalability and maintainability for future development. GitHub serves as a centralized repository, supporting version control and collaborative development management. Overall, AIT Match not only demonstrates technical excellence but also showcases the potential of targeted social networks to enhance student life through friendship, collaboration, and academic connections, setting a precedent for similar platforms in educational environments.

CHAPTER 5
PRELIMINARY RESULT

5.1 Code Repository and Deployment

5.1.1 GitHub Repository

The GitHub repository provides all essential code, configurations, and documentation for the AIT Match project, offering a comprehensive overview of its technical setup. It includes database schemas, controller logic, feature implementations, Docker configurations, and dataset seeding files. This repository serves as a valuable resource for understanding the platform's infrastructure, design, and development processes.

Figure 5.1 *GitHub Repository: AIT Match*

5.1.2 DockerHub Repository

The DockerHub repository hosts a pre-configured Docker image of the AIT Match platform, based on the source files from GitHub. This image allows quick deployment with all required dependencies and configurations, facilitating a seamless setup across various environments. The DockerHub repository streamlines deployment and ensures consistency in both development and production. Additionally, the platform is deployed on the CSIM (Computer Science & Information Management) server at the Asian Institute of Technology (AIT), providing an institutionally supported and secure environment for the platform's users within the university.

Figure 5.2 *Dockerhub Repository: AIT Match*

5.2 User Interface Results

5.2.1 Landing Page: Welcome

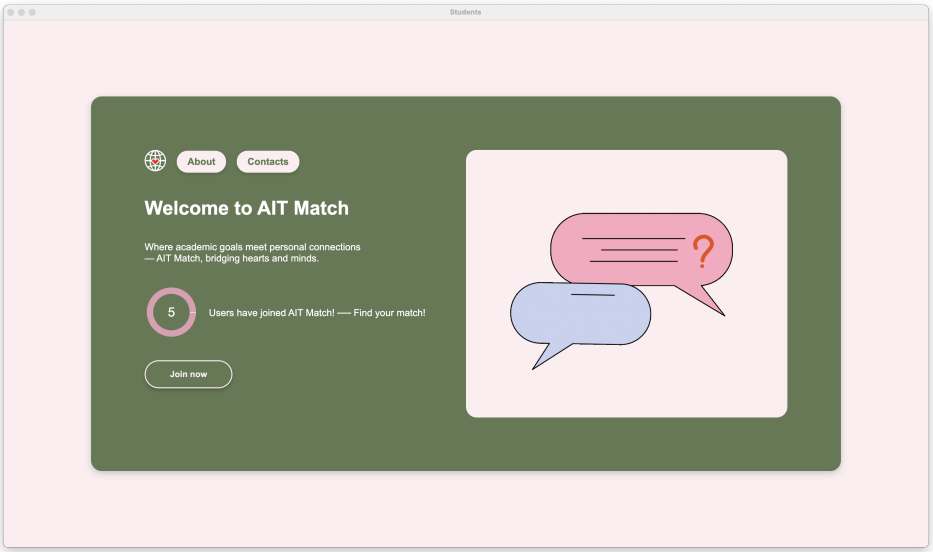

Figure 5.3 *Welcome Page.*

5.2.2 Landing Page: About

Figure 5.4 *About Page.*

5.2.3 Landing Page: Contact

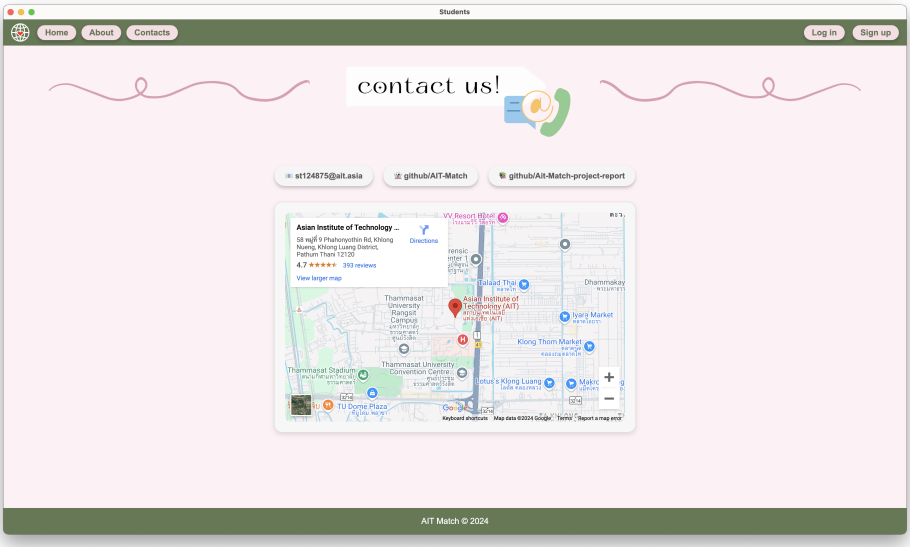

Figure 5.5 *Contact Page.*

5.2.4 User Authentication: Sign Up Page

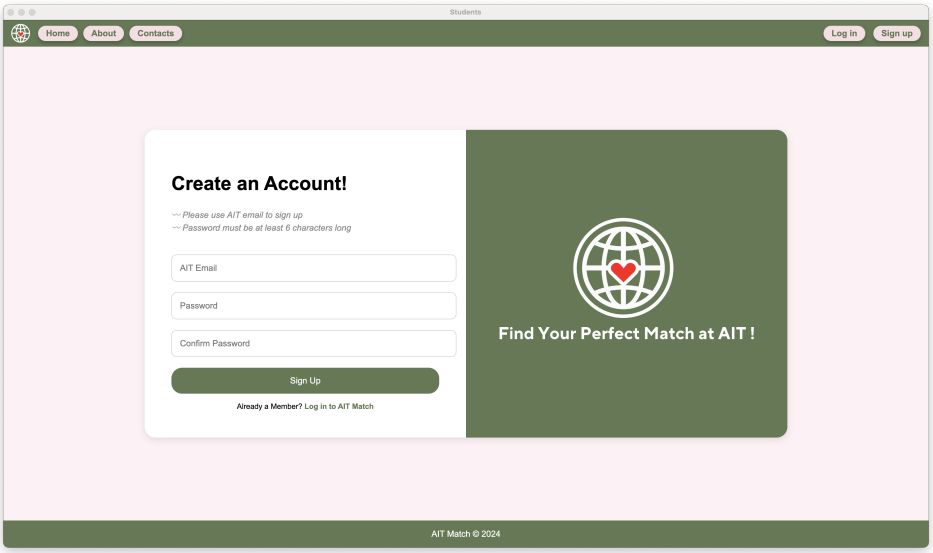

Figure 5.6 *Sign Up Page.*

5.2.5 User Authentication Page: Login Page

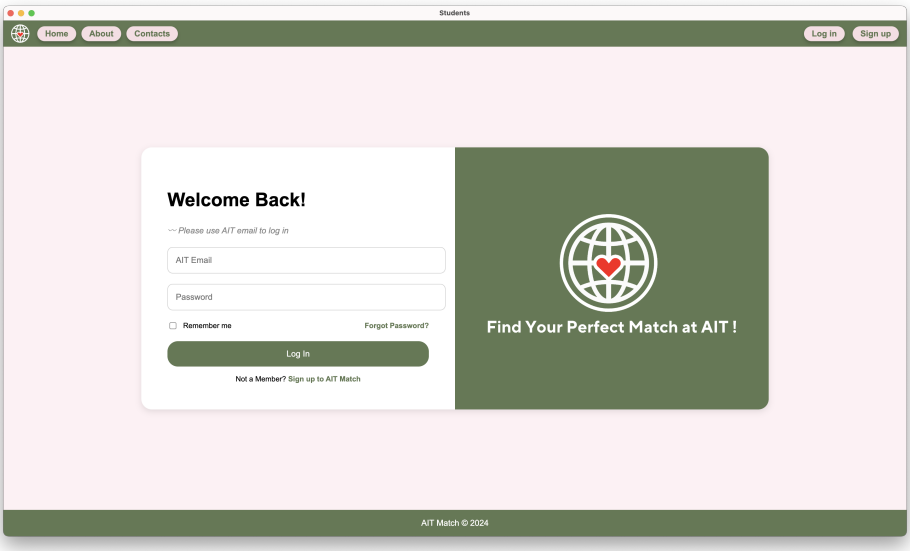

Figure 5.7 *Login Page.*

5.2.6 Profile: Profile Creation Page

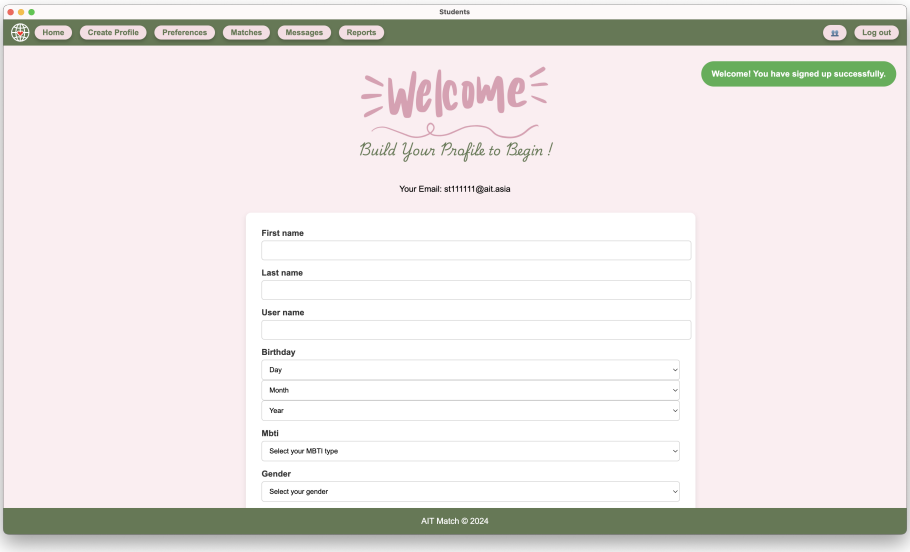

Figure 5.8 *Profile CreationPage.*

5.2.7 Profile: Profile Creation Page (Continued)

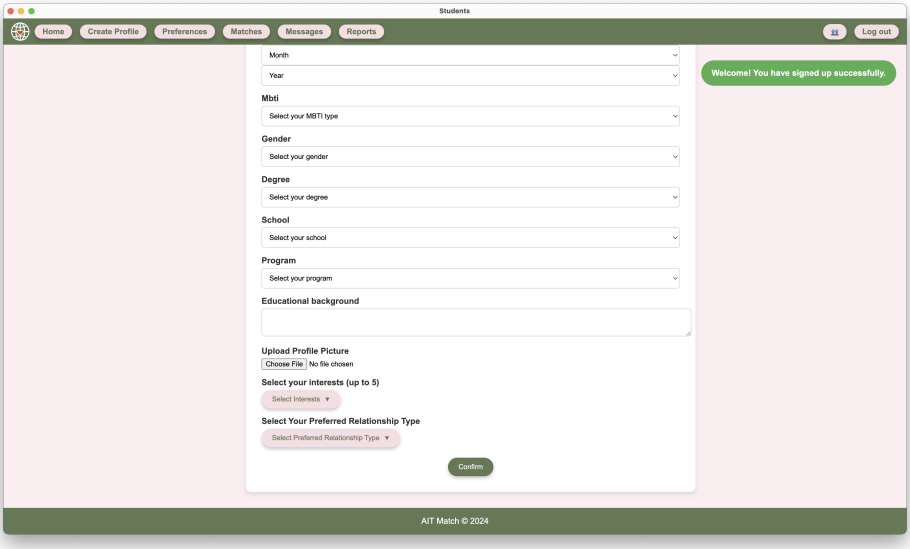

Figure 5.9 *Profile Creation Page (Continued).*

5.2.8 Profile: Validation of Profile Creation

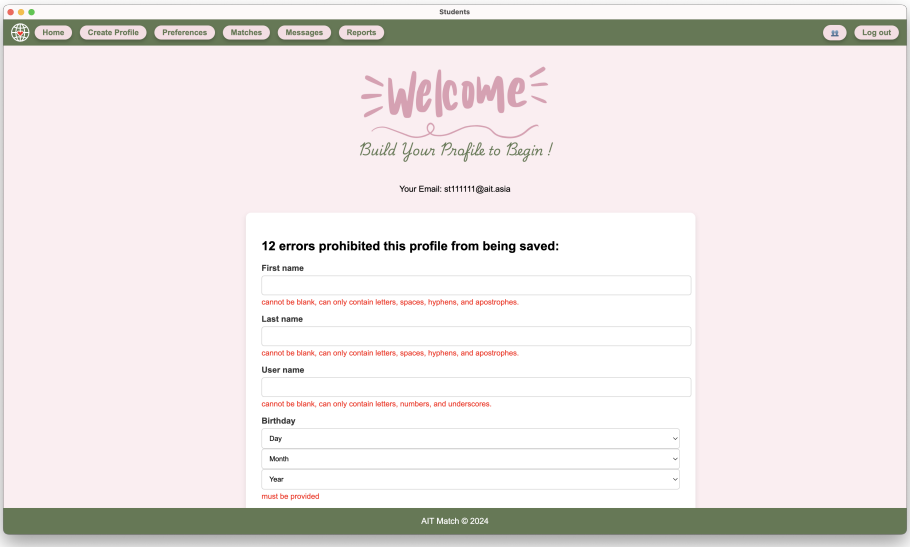

Figure 5.10 *Validation of Profile Creation.*

5.2.9 Profile: Validation of Profile Creation (Continued)

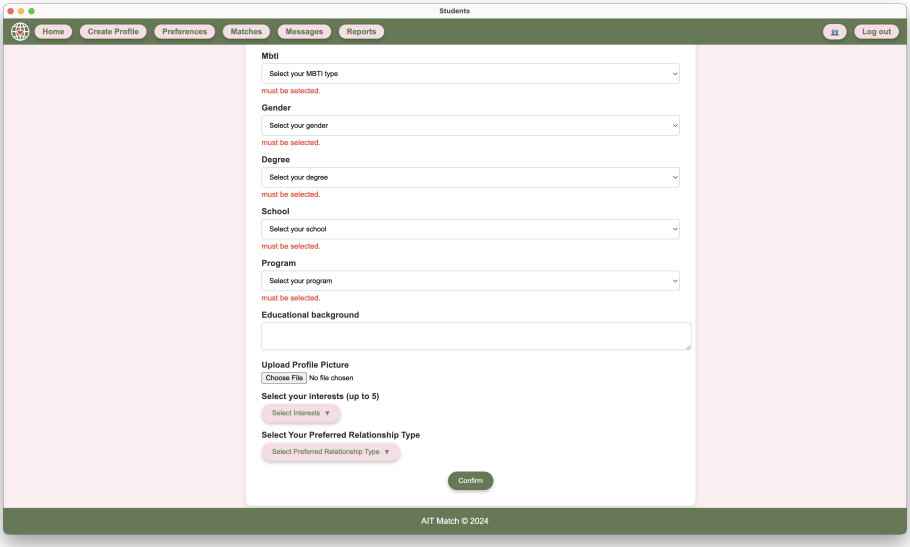

Figure 5.11 *Validation of Profile Creation (Continued).*

5.2.10 Profile: Profile Edit Page

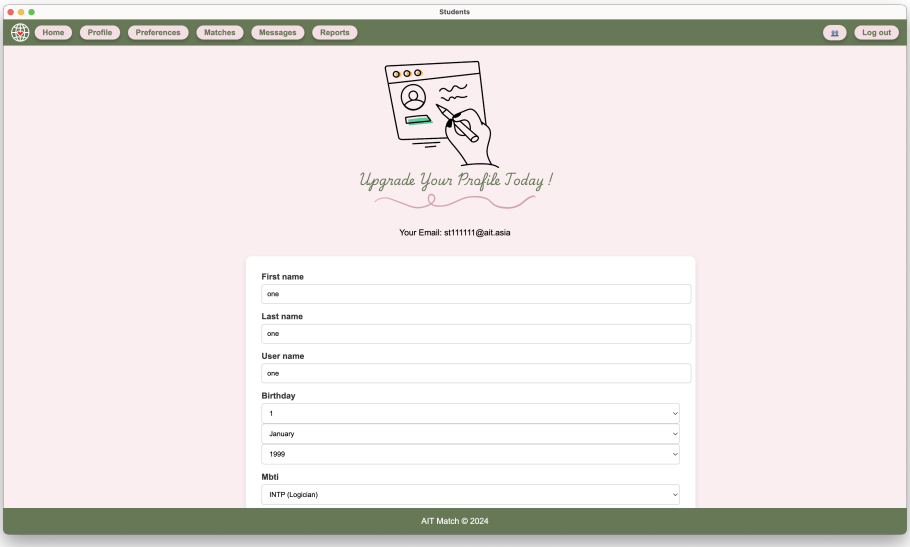

Figure 5.12 *Profile Edit Page.*

5.2.11 Profile: Profile Edit Page (Continued)

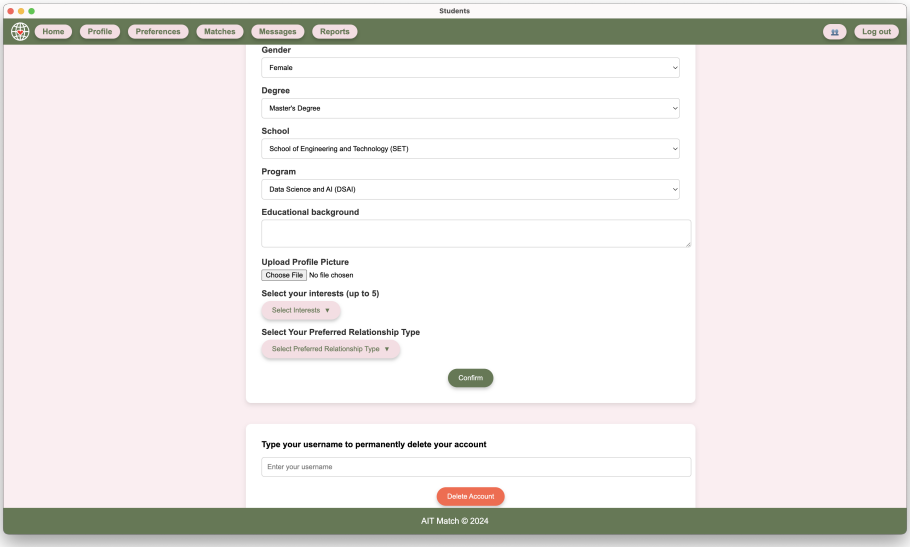

Figure 5.13 *Profile Edit Page (Continued).*

5.2.12 Profile: Profile Show Page

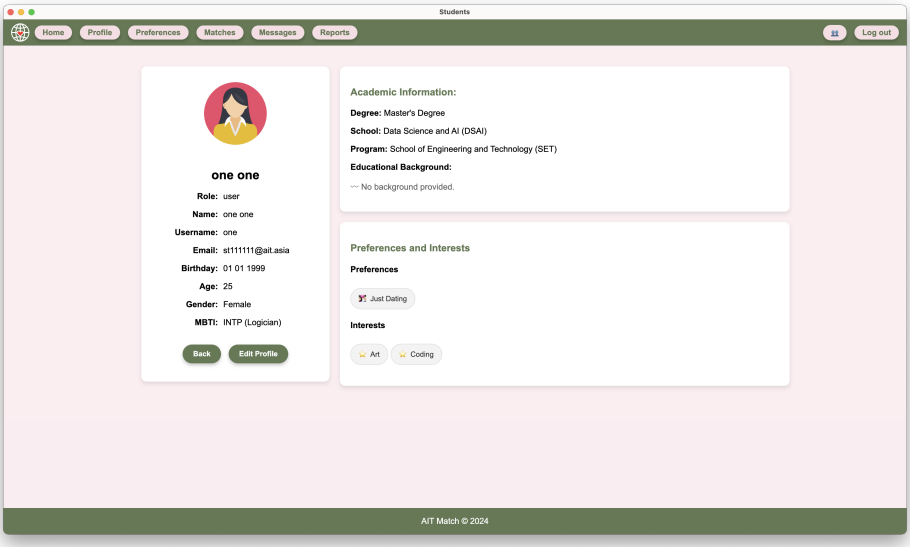

Figure 5.14 *Profile Show Page.*

5.2.13 Profile: Profile Index Page

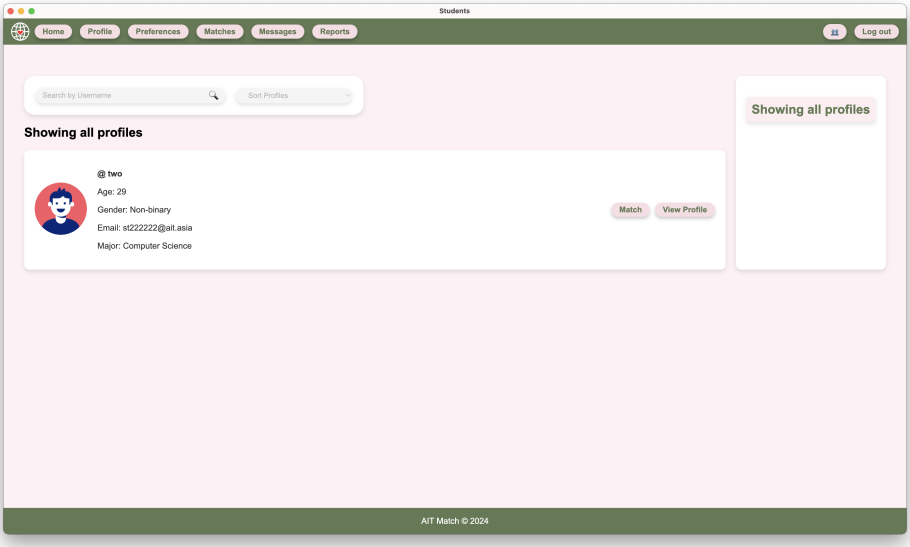

Figure 5.15 *Profile Index Page.*

5.2.14 Preference: Preference Setup Page

Figure 5.16 *Preference Setup Page.*

5.2.15 Preference: Preference Setup Page (Continued)

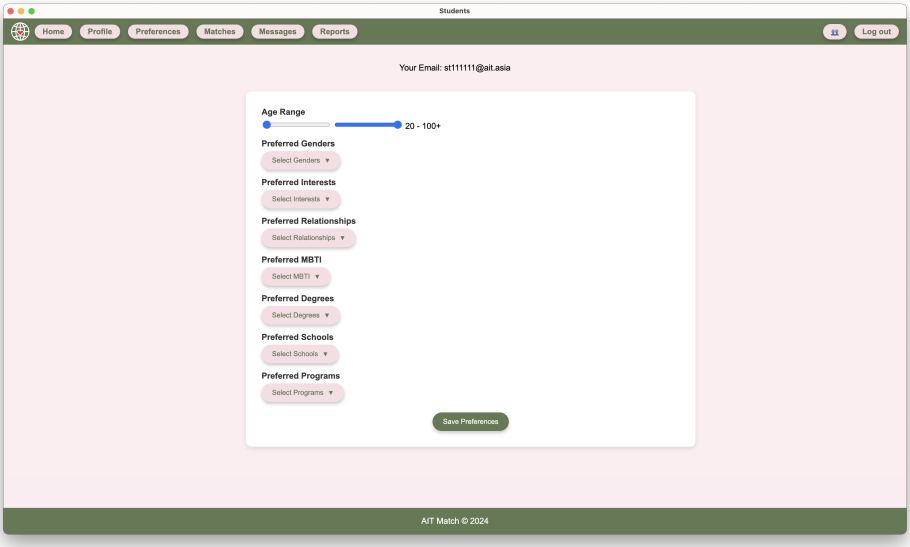

Figure 5.17 *Preference Setup Page (Continued).*

5.2.16 Match: No Match Requests Page

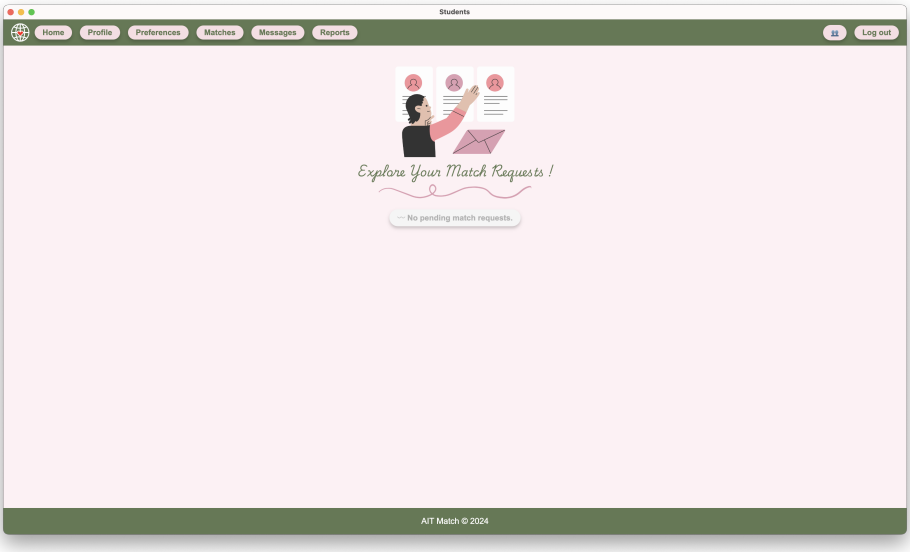

Figure 5.18 *No Match Requests Page.*

5.2.17 Match: Match Requests Page

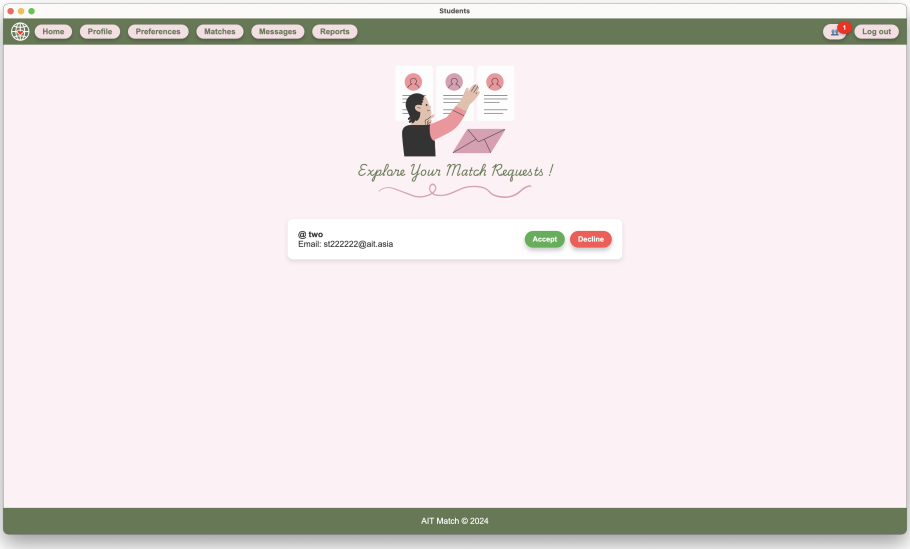

Figure 5.19 *Match Requests Page.*

5.2.18 Match: No Matched Profiles Page

Figure 5.20 *No Matched Profiles Page.*

5.2.19 Match: Matched Profiles Page

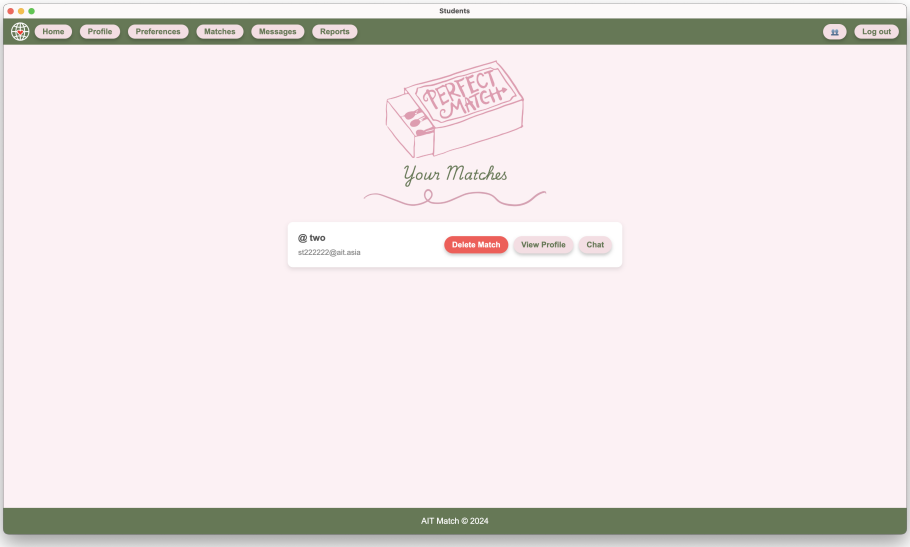

Figure 5.21 *Matched Profiles Page.*

5.2.20 Conversation: No Conversations Page

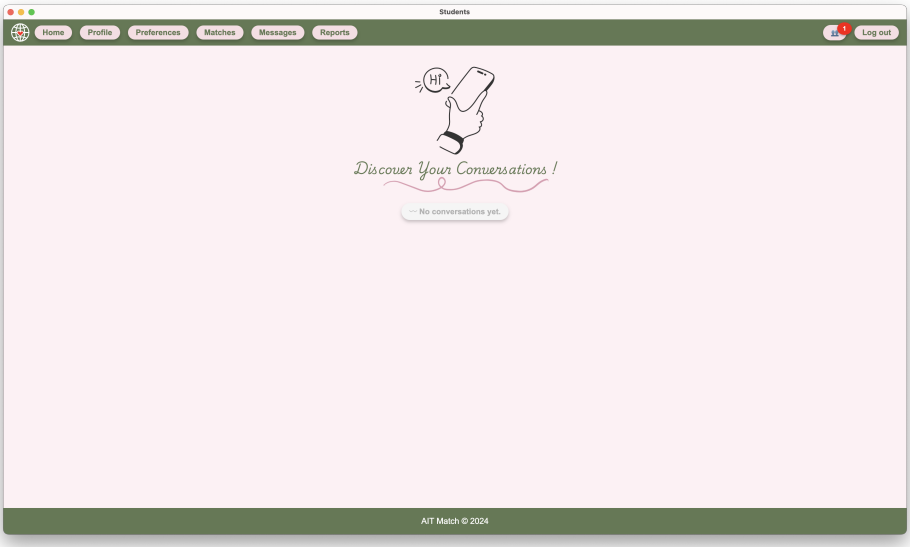

Figure 5.22 *No Conversations Page.*

5.2.21 Conversation: Conversations Page

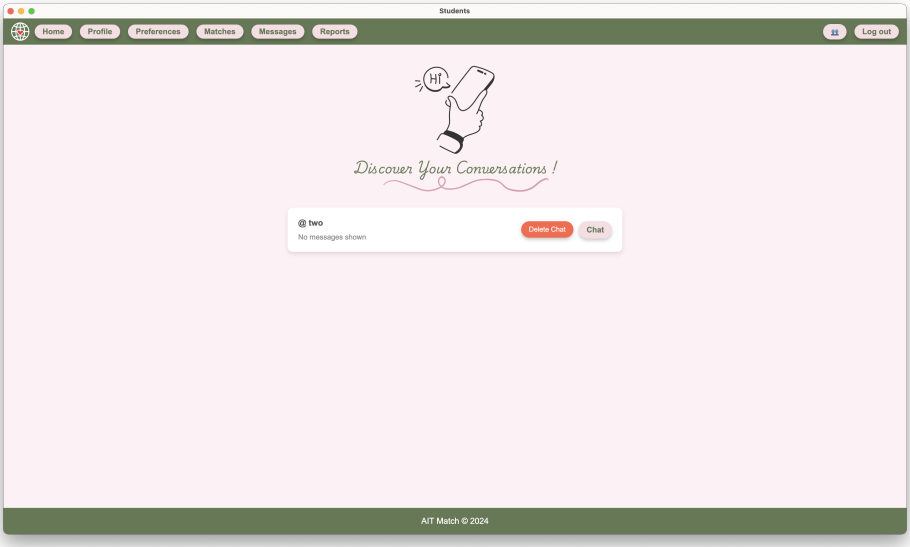

Figure 5.23 *Conversations Page.*

5.2.22 Chat: Chat Room Page

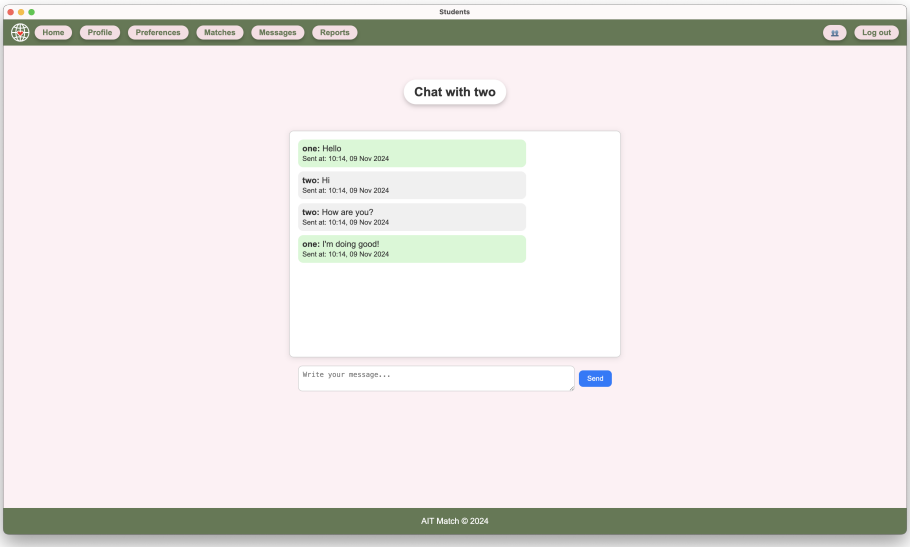

Figure 5.24 *Chat Room Page.*

5.2.23 Report: Reported Profile Page

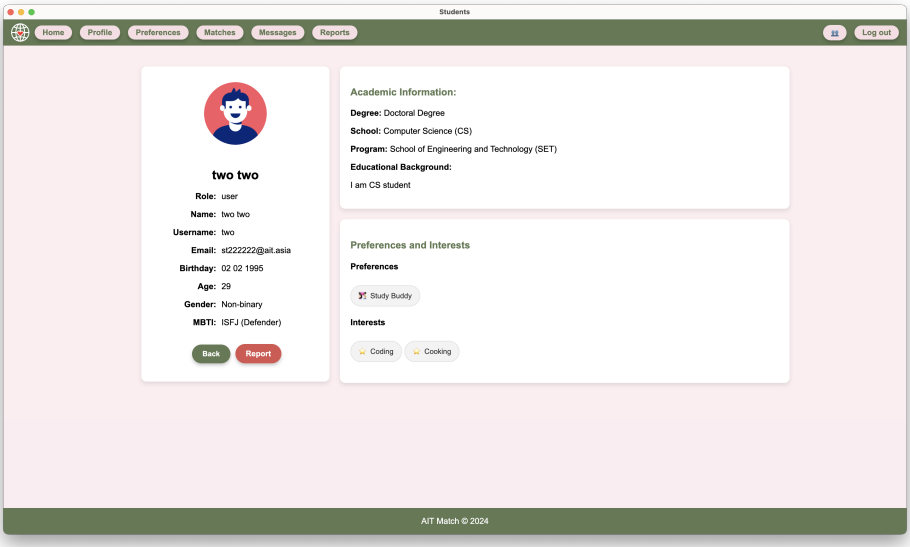

Figure 5.25 *Reported Profile Page.*

5.2.24 Report: New Report Page

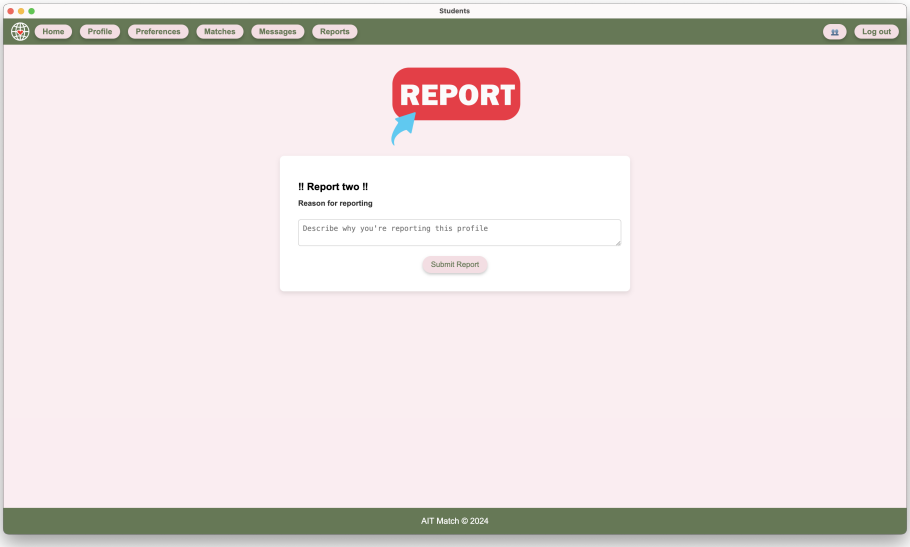

Figure 5.26 *New Report Page.*

5.2.25 Report: No Report Page

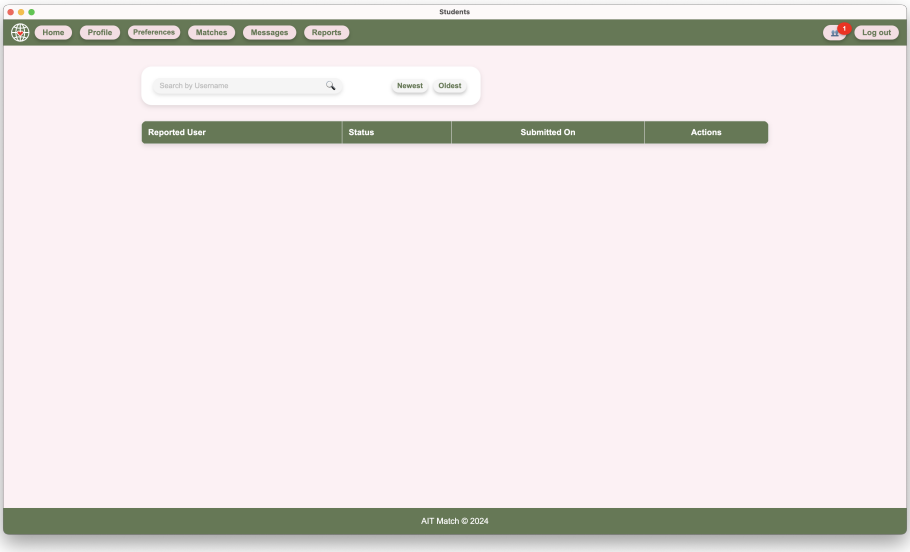

Figure 5.27 *No Report Page.*

5.2.26 Report: Report Index Page

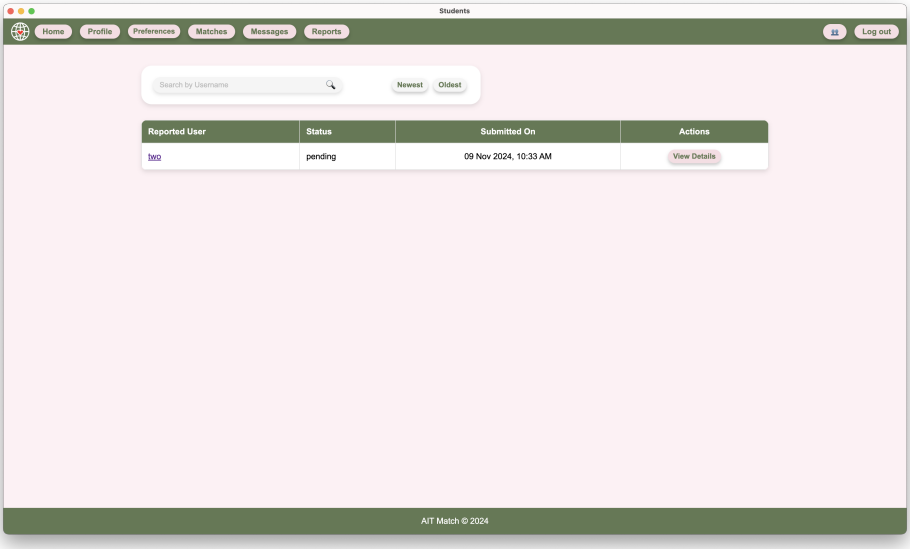

Figure 5.28 *Report Index Page.*

5.2.27 Report: Report Show Page

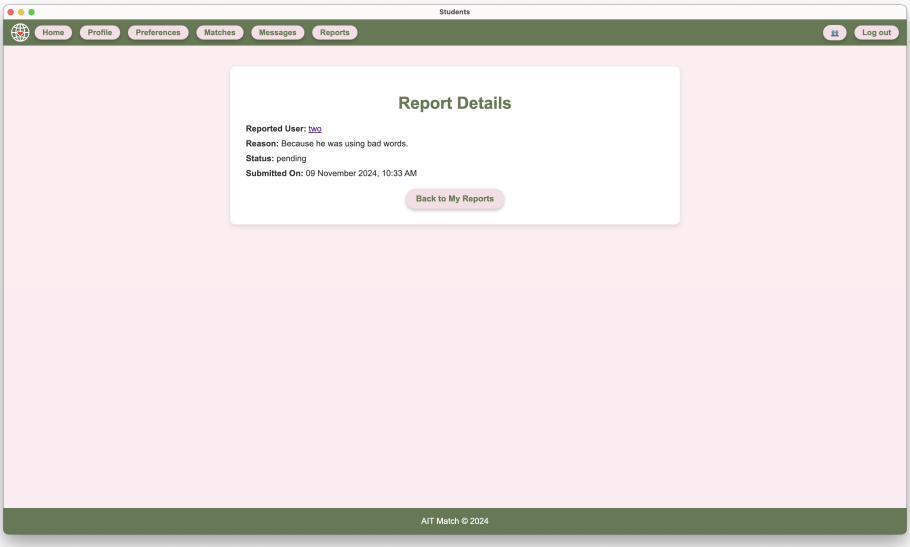

Figure 5.29 *Report Show Page.*

5.2.28 Admin Panel: Users Management Page

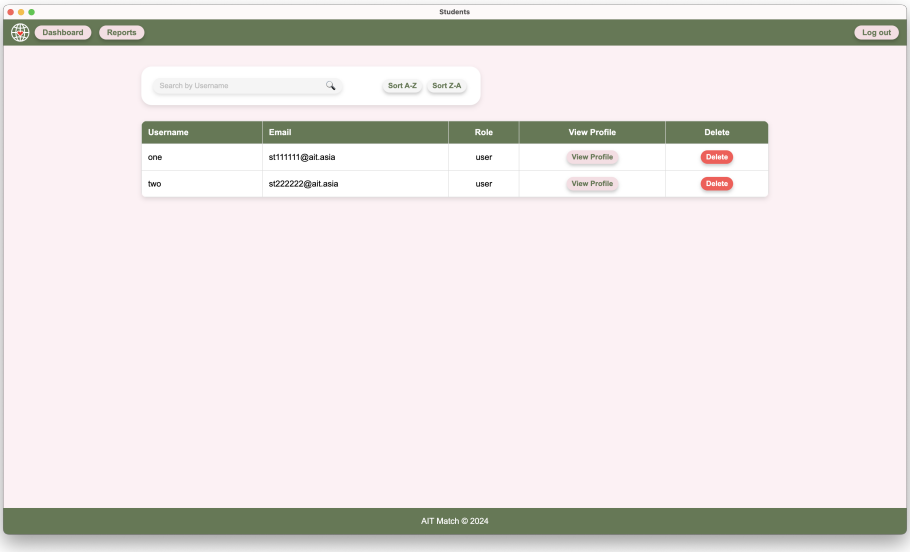

Figure 5.30 *Admin Users Management Page.*

5.2.29 Admin Panel: Reports Management Page

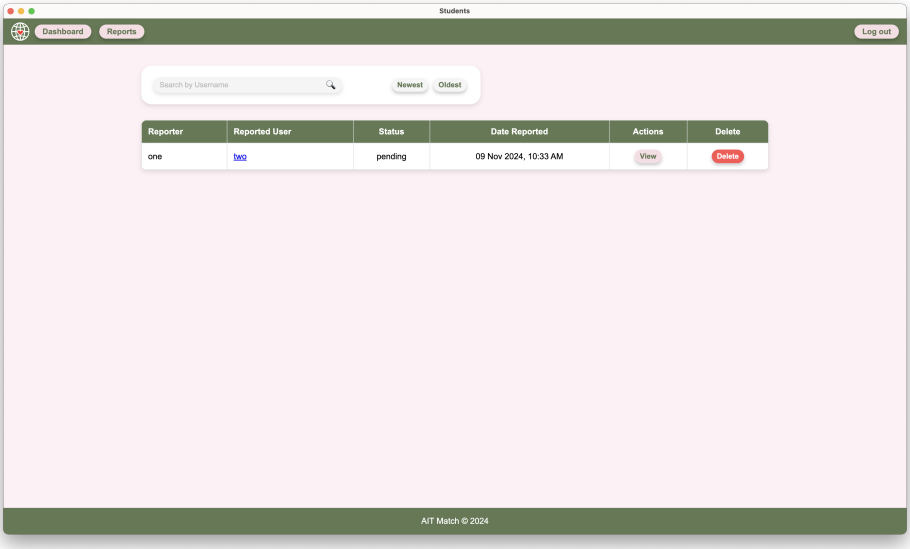

Figure 5.31 *Admin Reports Management Page.*

CHAPTER 6
PROJECT EVALUATION

6.1 Challenges

The development and deployment of the AIT Match application encountered several challenges, including:

6.1.1 Google Authentication and SMTP Email:

Integrating Google authentication and configuring SMTP for email notifications presented technical hurdles that required careful attention to security protocols. According to Google Support (2024), as of September 30, 2024, Google will no longer support less secure apps that only use a username and password for authentication, particularly for Google Workspace accounts. This change means that using a simple username and password combination with SMTP to send emails from Gmail may no longer work due to the deprecation of this form of authentication for security reasons. Consequently, I faced significant challenges verifying my sender email for password resets in Devise, as this required adapting to new security measures that complicate the setup process for sending emails.

6.1.2 Google Drive Integration for Profile Pictures:

Initially, I attempted to use Google Drive links for hosting profile pictures; however, this proved to be impractical due to various restrictions imposed by Google, such as limitations on sharing and CORS (Cross-Origin Resource Sharing) policies, according to Google Drive (2024). Consequently, I shifted to using Imgur for image hosting. Unfortunately, this transition also presented challenges, particularly in a production environment where access is limited to the CSIM server via the CSIM Wi-Fi. This setup effectively blocks Imgur's IP addresses, preventing users from accessing images hosted on Imgur while using the application. As a result, I opted to implement Active Storage for image management, as it provides a more seamless user experience and circumvents these access issues.

6.1.3 ActiveStorage Performance Issues:

The implementation of ActiveStorage for managing user-uploaded images resulted in slower website performance. According to Rails Team (2024), this slowdown can be attributed to several factors, including the overhead associated with processing and storing images, which can lead to increased load times and reduced responsiveness of the application. Specifically, when images are uploaded, ActiveStorage needs to create multiple variants for different display requirements, which requires additional processing time and storage space. Moreover, the reliance on external storage services can introduce latency, especially if network conditions are suboptimal. These performance issues necessitated optimizations to ensure a smoother user experience

6.1.4 Performance Issues Due to Excessive Queries

One significant challenge encountered in the AIT Match platform is the slow performance of the website, particularly when handling user queries. The filtering system, which allows users to specify multiple criteria when searching for profiles, results in a large number of database queries. As users request more complex filters, the system performs additional queries to fetch the required data. This can lead to increased load times and reduced responsiveness, negatively impacting the overall user experience.

Additionally, when users are setting up their profiles or preferences and need to fill in user information from the dataset in the database, the system also experiences slow performance. Retrieving and populating data from the database for these forms can further exacerbate load times, making the user experience cumbersome. Optimizing these queries and implementing more efficient data retrieval methods will be necessary to enhance the performance of the application.

6.2 Future Work

In anticipation of continued growth and user engagement, several enhancements and new features are proposed to further improve the AIT Match platform:

6.2.1 Expansion to Other Universities

Plans are underway to broaden the platform's reach by making it available to students from other universities, thereby enhancing the user base and community engagement. Currently, the application is limited to users from the Asian Institute of Technology (AIT), but future developments could extend accessibility to other institutions across Thailand, facilitating a more diverse community of learners.

6.2.2 User Ban System

Developing a system to ban users instead of deleting their accounts provides a more flexible approach to managing user behavior. Currently, administrators can only delete profiles deemed inappropriate, which is a permanent action. By introducing a ban feature, administrators can temporarily restrict access to the chat function or place a user in a timeout for a designated period, allowing for corrective actions without permanently removing users from the platform.

6.2.3 Filter Inappropriate Words in Chat

Implementing filters to prevent users from typing inappropriate words in the chat system is essential for maintaining a respectful environment. This feature will actively screen messages to block any offensive language or derogatory terms, ensuring that all users can communicate in a safe and constructive manner.

6.2.4 Chat Functionality between Users and Admin

Enabling direct communication channels between users and administrators will significantly enhance support and user experience. Currently, users who submit reports have no means to inquire about the status or outcomes of their submissions. Likewise, administrators lack a mechanism to provide updates or feedback on the reports. Implementing a chat feature or comment section for each report will foster open communication, allowing users to seek clarifications and administrators to convey updates effectively.

6.2.5 Addition of Events, Blogs, Posts, and Comments

Integrating features for users to create and share events, blogs, posts, and comments will promote greater engagement and interaction within the community. This functionality would empower users to contribute to discussions, share experiences, and foster a lively social atmosphere, making the platform not just a dating app but also a hub for community interaction.

6.2.6 More Customizable Interests

Allowing users to define a wider range of customizable interests will enhance matching accuracy and user satisfaction. At present, users can only select from preset options defined by developers, which may not fully capture their unique preferences. Future iterations should incorporate features that allow users to add and modify their interests, leading to more tailored connections and experiences.

6.2.7 Chat with Sending Images and Emojis

Currently, the chat functionality is limited to text messaging, which may feel monotonous for users. Future updates could include support for sending images and emojis, making conversations more dynamic and engaging. This enhancement would allow users to express themselves more fully and interact in a fun, creative manner, thereby enriching the overall chat experience.

REFERENCES

Bcrypt. (1999). *Bcrypt: Password hashing function for ruby.* Retrieved from https://github.com/codahale/bcrypt-ruby

Bumble. (2014). *Bumble: Dating, friends and business networking.* Retrieved from https://bumble.com

Coffee-Meets-Bagel. (2012). *Coffee meets bagel: Meaningful connections and authentic dating.* Retrieved from https://www.coffeemeetsbagel.com

Datamatch. (1994). *Datamatch: Harvard's premier free dating service.* Retrieved from https://www.datamatch.me

Devise. (2009). *Devise: Flexible authentication solution for rails.* Retrieved from https://github.com/heartcombo/devise

Docker, I. (2013). *Docker: Empowering app development for developers.* Retrieved from https://www.docker.com

Facebook. (2004). *Facebook: A global social networking platform.* Retrieved from https://www.facebook.com

Facebook. (2020). *Facebook campus: Connect with your college community.* Retrieved from https://www.facebook.com/campus

Force, I. I. E. T. (2011). *The websocket protocol.* Retrieved from https://datatracker.ietf.org/doc/html/rfc6455 (RFC 6455)

Friendsy. (2012). *Friendsy: The college-only social network.* Retrieved from https://www.friendsyapp.com

Google Drive. (2024). *Google drive help.* Retrieved from https://support.google.com/drive/answer/2452654?hl=en

Google Support. (2024). *Less secure apps & your google account.* Retrieved from https://support.google.com/accounts/answer/6010255?hl=en

Group, P. G. D. (1997). *Postgresql: The world's most advanced open source relational database.* Retrieved from https://www.postgresql.org

Hinge. (2013). *Hinge: Designed to be deleted.* Retrieved from https://hinge.co

International, E. (1993). *Javascript: Ecmascript language specification.* Retrieved from https://262.ecma-international.org

LinkedIn. (2003). *Linkedin: Professional networking, jobs, and career development.* Retrieved from https://www.linkedin.com

ORM_Adapter. (2010). *Orm adapter: Adapter for various orm systems in rails.* Retrieved from https://github.com/ianwhite/orm_adapter

Rails Team. (2015). *Actioncable: Real-time websocket framework in rails.* Retrieved from https://guides.rubyonrails.org/action_cable_overview.html

Rails Team. (2024). *Active storage overview.* Retrieved from https://guides.rubyonrails.org/active_storage_overview.html

Railties. (2008). *Railties: Interface between the rails framework and ruby gems.* Retrieved from https://github.com/rails/rails/tree/main/railties

Sanfilippo, S., & Labs, R. (2009). *Redis: In-memory data structure store.* Retrieved from https://redis.io

(SMTP), S. M. T. P. (1982). *Smtp: Internet standard for electronic mail transmission.* Retrieved from https://tools.ietf.org/html/rfc5321

Team, R. (1995). *Ruby: A dynamic, open source programming language.* Retrieved from https://www.ruby-lang.org

Team, R. (2004). *Ruby on rails: Web application framework.* Retrieved from https://rubyonrails.org

Tinder. (2012). *Tinder: Dating, make friends and meet new people.* Retrieved from https://www.gotinder.com

W3C. (1993). *Html: Hypertext markup language.* Retrieved from https://www.w3.org/TR/html52/

W3C. (1996). *Css: Cascading style sheets.* Retrieved from https://www.w3.org/Style/CSS/Overview.en.html

Warden. (2009). *Warden: Middleware for authentication in rack applications.* Retrieved from https://github.com/wardencommunity/warden