Sprint 3 Retrospective - Project Jimmy

Links to our GitHub Repo, Heroku Deployment, Pivotal Tracker, Slack workspace, and Code Climate:

- Github Repo - https://github.com/tamu-edu-students/jimmy-gym-buddy-finder
- Deployment URL - https://jimmy-buddy-finder-f97708d96ef8.herokuapp.com/
- Pivotal Tracker - https://www.pivotaltracker.com/n/projects/2721606
- Slack Workspace - https://app.slack.com/client/T07P2NT2ZM1/C07P00FFRGD
- Code Climate - https://codeclimate.com/github/tamu-edu-students/jimmy-gym-buddy-finder

Dates of the Sprint:

21st October 2024 to 3rd November 2024

Information about team members and contributions:

Team Member	Contribution	Tasks
Kuan-Ru Huang	13%	• Proof of Concept (POC) for Persistent Chat Feature
Wei-Chien Cheng	14%	• Notification Triggering on Match
Yash Phatak	15%	• Profile Matching UI with User Preferences and Filters • CSS Bug Fixes and Mobile Responsiveness for Profile Matching
Mrunmay Deshmukh	15%	• Fitness Profile Activity Preferences • UI and CSS Enhancements as per client feedback

Kushal Lahoti	15%	• Profile Matching Algorithm and Database Integration
Barry Liu	15%	• Private Chat UI Interface
ChuanHsin Wang	13%	• Notification Dialog Box in UI for Match Notifications

Sprint Goal:

The primary goal of this sprint was to enhance the core user experience of the gym buddy finder application by improving user interaction and communication features while addressing key functional and design issues and conducting a Proof of Concept (POC) for new functionality.

1. **Feature Implementation:** The sprint focused on implementing the following features – the profile matching backend, a notification system for matched users, and a private chat UI.
    a. The **profile matching feature** is fully connected to the backend, ensuring that users can view actual profiles based on their preferences, with relevant filters and sorting algorithms in place.
    b. A **notification system**, where both the notification backend and the notification UI are implemented, allowing users to receive match notifications and mark them as read.
    c. A **mock private chat UI** is implemented, allowing users to initiate communication with their matched gym buddies.
2. **UI/UX and CSS Styling Improvements**: A significant portion of the sprint focused on resolving **CSS-related bugs**, especially on the **profile matching page**, ensuring that the interface is fully responsive and visually appealing across devices. The application is now adjusted to improve the mobile experience, including proper alignment, font scaling, and consistent button styling. Additionally, we addressed minor UI issues raised by client feedback, such as **font colors, button styling, and background adjustments**.
3. **POC for Persistent Chat Feature**: A **Proof of Concept (POC)** was conducted to explore how a **persistent chat feature** can be implemented in the Ruby on Rails application. This helped to determine the best approach for the future implementation of a robust chat feature.

The application has improved profile matching functionality, better design, a notification system, and a solid foundation for future chat integration, providing an overall richer and more responsive user experience.

User stories: A total of 9 user stories have been completed in this sprint.

Feature: Fitness Profile Activity Preferences
As a user,
I want to manage my fitness profile by adding my activity preferences,
So that I can find workout partners that match my specific fitness goals and interests.

Scenario 1: Add Preferred Activities to Fitness Profile
Given: The user is on the fitness profile settings page.
When: The user selects preferred workout activities such as weightlifting, running, yoga, etc.
Then: The preferences should be saved and reflected in the user's fitness profile.

Scenario 2: Set Fitness Buddy Preferences
Given: The user wants to specify their preferences for workout partners.
When: The user selects options such as desired workout partner experience level, location, availability times, or preferred workout intensity.
Then: These preferences should be saved and used to match with potential gym buddies.

Scenario 3: Update or Remove Activity Preferences
Given: The user wants to update or remove existing activity preferences.
When: The user modifies their choices for fitness activities or buddy preferences.
Then: The updated preferences should replace the old ones and reflect on the profile accordingly.

Feature: Profile Matching Algorithm and Database Integration
As a developer,
I want to implement a backend that connects to the actual database and maintains profile matching data for specific users,
So that the system can display relevant user profiles based on specific matching, sorting, and filtering criteria for each user.

Scenario 1: Implement Profile Matching Algorithm

Given: Each user has fitness preferences and a history of interactions (matched, skipped, or blocked profiles).
When: The user browses the profiles.
Then: The backend should implement an algorithm that:

- Sorts and filters profiles based on activity preferences, location, workout timings, and experience level.
- Excludes profiles that have been blocked or already matched.
- Prioritizes profiles that are a closer match to the user's criteria.

Scenario 2: Maintain User Interaction Records
Given: The system needs to track the user's interactions with other profiles.
When: The user skips, blocks, or matches with a profile.
Then: The backend should store this interaction data in the database, ensuring that the same profiles are not shown again unnecessarily in the future.

Scenario 3: Query the Database for Relevant Profiles
Given: The backend needs to fetch profiles from the database.
When: The system queries the database for relevant user profiles.
Then: The profiles should be filtered and sorted based on user preferences and past interactions using a dynamic algorithm that adapts to user behavior.

Scenario 4: Handle Skipped, Blocked, and Matched Profiles
Given: The backend needs to manage and respect user choices for skipped, blocked, and matched profiles.
When: The user performs any of these actions on profiles.
Then: The corresponding records should be updated, and the backend will exclude these profiles from future queries unless preferences change.

Feature: Profile Matching UI with User Preferences and Filters
As a developer,
I want to connect the mock UI for the profile matching feature to the implemented backend and add functionality for users to select preferences and filters,
So that the system can display relevant user profiles based on dynamic criteria and preferences.

Scenario 1: Connect Mock UI to Backend for Displaying Profiles
Given: The user interacts with the profile-matching UI (e.g., browsing/swiping profiles).
When: The user opens the matching interface.
Then: The system should query the backend database and display actual user profiles that match the user's preferences and filters.

Scenario 2: Enable User to Select Preferences for Matching
Given: The user wants to refine the profiles they see based on specific fitness preferences.
When: The user selects filters such as workout type, location, experience level, and available times.
Then: The system should dynamically adjust the displayed profiles, only showing profiles that match the selected criteria.

Scenario 3: Save User's Preferences and Apply Filters in Backend Queries
Given: The user has set specific preferences and filters.
When: The backend queries the database for user profiles.
Then: The backend should apply these preferences and filters, ensuring that only the most relevant profiles are returned for display.

Scenario 4: Update Displayed Profiles Based on User Interactions
Given: The user interacts with displayed profiles by skipping, blocking, or matching.
When: The user takes an action on a profile.
Then: The backend should update the user’s interaction history, and the displayed profiles should reflect this, excluding skipped or blocked users in future matches.

Scenario 5: Implement Real-Time Filtering of Profiles

Given: The user wants to update their preferences in real time.
When: The user modifies filters or preferences (such as changing location or workout type).
Then: The system should re-query the backend and immediately update the list of displayed profiles without requiring a page refresh.

Feature: CSS Bug Fixes and Mobile Responsiveness for Profile Matching
As a developer,
I want to resolve CSS-related bugs on the profile matching page and improve its appearance on mobile-sized interfaces,
So that the page looks visually appealing and functions properly across different screen sizes, especially mobile devices.

Scenario 1: Fix CSS Bugs on Profile Matching Page
Given: There are existing CSS issues that affect the layout or appearance of the profile-matching page.
When: The developer inspects the page for broken styles or misaligned elements.
Then: The CSS bugs should be fixed to ensure a clean and functional design on desktop and mobile screens.

Scenario 2: Ensure Proper Layout on Mobile Devices
Given: The user views the profile matching page on a mobile device.
When: The page is rendered on smaller screen sizes (e.g., mobile phones).
Then: The layout should automatically adjust to fit the mobile interface without cutting off content or misplacing elements.

Scenario 3: Implement Responsive Design for Profile Cards and Buttons
Given: Profile cards and action buttons (such as swiping or matching) need to adapt to mobile screens.
When: The user views these elements on mobile devices.
Then: The profile cards and buttons should resize and align correctly to ensure easy interaction on smaller screens.

Feature: UI and CSS Enhancements as per client feedback

As a developer,
I want to resolve minor UI and CSS issues based on the client's feedback,
So that the application looks more visually appealing and meets the client's design expectations, including improvements to font color, buttons, and background styling.

Scenario 1: Update Font Color as per Client's Feedback
Given: The client has provided feedback regarding the font color on certain pages.
When: The developer reviews and adjusts the font color to match the client's preferences.
Then: The updated font color should enhance readability and align with the overall design aesthetic.

Scenario 2: Adjust Button Styles for Consistency
Given: The client has requested changes to button styles (e.g., size, color, hover effects).
When: The developer modifies the button CSS to reflect the feedback.
Then: The buttons across the application should have a consistent style that improves the user experience and aligns with the design theme.

Scenario 3: Improve Background Styling Based on Client Feedback
Given: The client has requested changes to the background (e.g., color, pattern, or texture).
When: The developer updates the background styling to match the requested changes.
Then: The background should complement the overall UI design and enhance the visual experience for users.

Scenario 4: Ensure Visual Consistency Across All Pages
Given: Multiple pages require UI consistency following the client's feedback.
When: The developer makes adjustments to fonts, buttons, and background elements.
Then: The UI should maintain a cohesive design across all pages, ensuring a seamless user experience.

Feature: Notification Triggering on Match
As a developer,
I want to implement notification-triggering logic in the backend,
So that whenever two users match with each other, both users are notified of the match.

Scenario 1: Trigger Notification When Users Match
Given: Two users swipe right or express interest in each other.
When: Both users' preferences align, resulting in a match.
Then: The backend should trigger notifications to both users, informing them of the successful match.

Scenario 2: Store Notification Details in Database
Given: A match occurs between two users.
When: The notification is triggered.
Then: The notification details (e.g., match date, user IDs) should be stored in the database for future reference and tracking.

Scenario 3: Ensure Notifications Are Sent to Both Users
Given: A match is confirmed between two users.
When: The backend processes the match.
Then: Notifications should be sent to both users simultaneously, ensuring that both are informed of the match at the same time.

Feature: Notification Dialog Box in UI for Match Notifications
As a user,
I want to see all my match notifications in a dialog box,
So that I can easily track new matches and mark notifications as read.

Scenario 1: Display Notification Dialog Box
Given: The user has match notifications.
When: The user clicks on the notification icon or opens the notifications menu.
Then: A dialog box should appear, displaying all notifications, with unread notifications clearly marked as new.

Scenario 2: Mark Notification as Read
Given: The user has unread notifications.
When: The user clicks or interacts with an individual notification.

Then: The notification should be marked as read, updating its status in the backend and UI so it no longer appears as new.

Scenario 3: Segregate Read and Unread Notifications
Given: The user has both read and unread notifications.
When: The user opens the notification dialog.
Then: Unread notifications should appear at the top of the list and be visually distinguished (e.g., bold or highlighted), while read notifications are listed below or in a separate section.

Scenario 4: Store Read/Unread Status in the Backend
Given: The user marks a notification as read.
When: The action is taken.
Then: The read/unread status should be updated and stored in the backend, so the notification's state persists across sessions.

Scenario 5: Handle Empty Notification State
Given: The user has no notifications.
When: The user opens the notification dialog box.
Then: The dialog should display a message like "No new notifications" to indicate there are no available match notifications.

Feature: Private Chat UI Interface
As a user,
I want to have a private chat interface with another user,
So that I can communicate with my matched gym buddies directly through the app.

Scenario 1: Display Chat Interface Between Two Users
Given: The user has matched with another user.
When: The user clicks on the “Chat” button for the match.
Then: A private chat window or section should open, displaying an empty message field and previous conversation history (if any).

Scenario 2: Send and Display Messages in the Mock UI
Given: The user types a message into the chat input field.
When: The user presses "Send."
Then: The message should be displayed in the chat window on the user's side, and a corresponding "received" message box should appear

on the other side (mocked for UI purposes).

Scenario 3: Display Chat History (Mocked Data)
Given: The user opens the chat interface with a matched user.
When: There are previously sent or received messages.
Then: The chat window should display the message history in a scrollable format, with the sender and receiver's messages aligned on opposite sides of the window.

Scenario 4: Handle Empty Chat State
Given: The user opens the chat window for the first time.
When: There is no message history between the users.
Then: The chat window should display a placeholder message, such as "Start the conversation!" or a blank space for the chat input.

Scenario 5: Responsive Design for Chat UI
Given: The user is accessing the app on different devices.
When: The user opens the chat interface on a mobile or tablet.
Then: The chat UI should adapt to the screen size, ensuring the input field and chat window are easy to use on both mobile and desktop devices.

Feature: Proof of Concept (POC) for Persistent Chat Feature
As a developer,
I want to carry out a Proof of Concept (POC) to implement a **persistent chat feature** in the Ruby on Rails application,
So that I can evaluate the feasibility of storing and retrieving chat messages between users for long-term use.

Scenario 1: Setting up Basic Chat Functionality
Given: The developer has set up models for users and chat messages,
When: Two users interact with each other through the chat interface,
Then: The system should allow messages to be sent and stored in a database for future retrieval.

Scenario 2: Evaluating Database Schema for Persistent Storage
Given: The chat messages need to be stored for persistence,
When: The developer designs the database schema for storing messages (e.g., sender, receiver, message content, timestamp),

Then: The schema should allow efficient retrieval and querying of past conversations.

Scenario 3: Testing Real-Time Chat Capabilities
Given: The POC involves exploring real-time communication,
When: The developer integrates ActionCable or WebSockets for live updates,
Then: The chat feature should support real-time message exchange between users.

Scenario 4: Exploring Data Retention and Archiving
Given: Users may want to retain chat history for future reference,
When: The POC considers data retention policies and archiving strategies,
Then: The system should offer options for retaining or archiving old conversations.

Sprint Achievements:

This sprint made remarkable progress and included a total of 9 user stories added. Key features were implemented across various areas focused on enhancing the fitness profile, profile matching, notifications, and chat functionality.

1. Fitness Profile Activity Preferences

- Added user ability to manage fitness activity preferences.
- Enabled settings for workout buddy preferences, allowing users to select partner criteria.
- Implemented options for users to update or remove activity preferences, reflecting changes in their fitness profile.

2. Profile Matching Algorithm and Database Integration

- Created a backend profile matching algorithm that sorts and filters profiles based on user preferences.
- Maintained interaction history (matched, skipped, blocked) to ensure personalized profile suggestions.

3. Profile Matching UI with User Preferences and Filters

- Connected UI to backend for real-time display of matching profiles.
- Enabled users to refine profile matches with filters (e.g., activity type, location).

- Implemented real-time profile updates based on user interactions and preference adjustments.

4. CSS Bug Fixes and Mobile Responsiveness

- Fixed CSS issues on the profile matching page.
- Ensured proper mobile layout for seamless interaction on smaller screens.
- Applied responsive design to profile cards and buttons.

5. UI and CSS Enhancements (Client Feedback)

- Updated font colors, button styles, and background based on client feedback.
- Ensured consistent UI across all pages, improving overall visual experience.

6. Notification Triggering on Match

- Developed notification logic for matched users, with real-time notifications for both parties.
- Stored notification details in the database for tracking and future reference.

7. Notification Dialog Box in UI

- Created a notification dialog box to display unread and read match notifications.
- Enabled marking notifications as read and persisted read/unread status in the backend.
- Included handling of an empty state with appropriate messaging.

8. Private Chat UI Interface

- Developed a private chat interface for matched users.
- Enabled message sending and mock display in the UI.
- Designed a responsive chat interface suitable for both mobile and desktop.

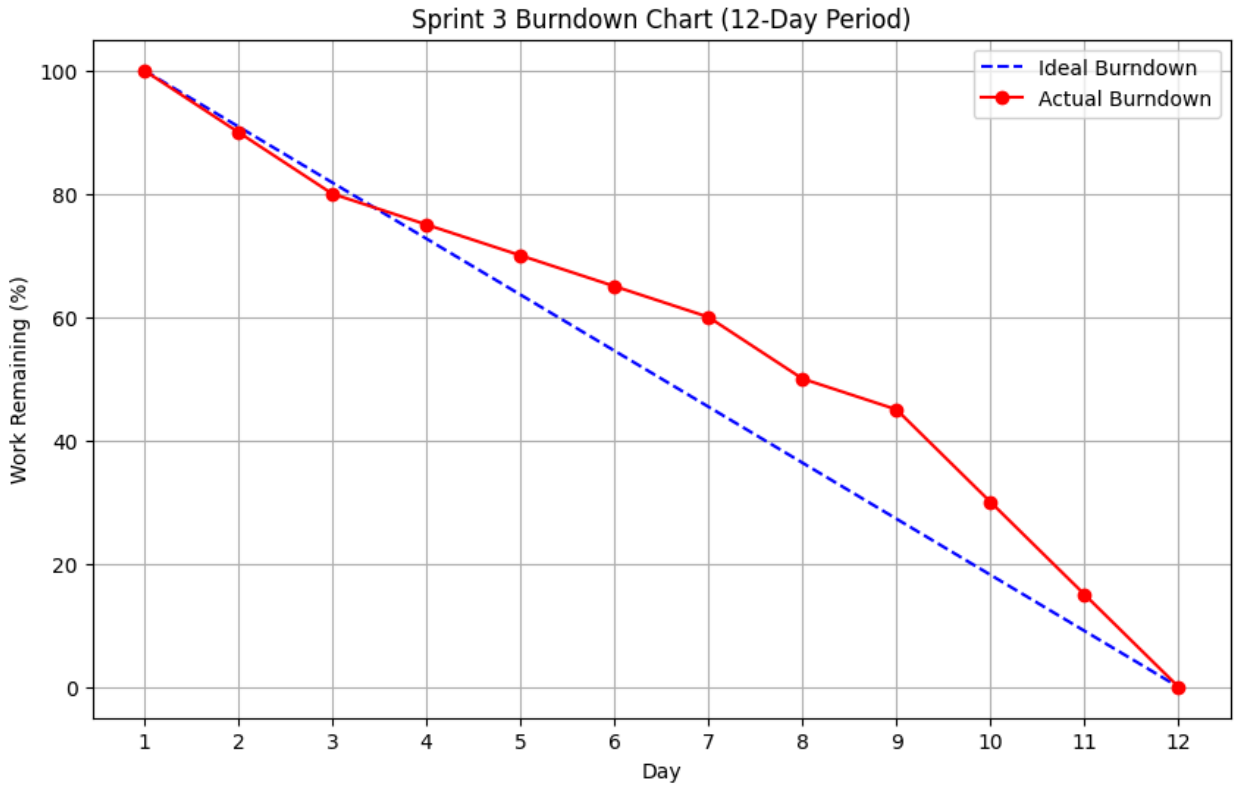
9. Proof of Concept (POC) for Persistent Chat Feature

- Implemented basic chat functionality with message storage in a database.
- Tested database schema for long-term message storage.
- Explored real-time messaging with ActionCable and retention options for chat history.

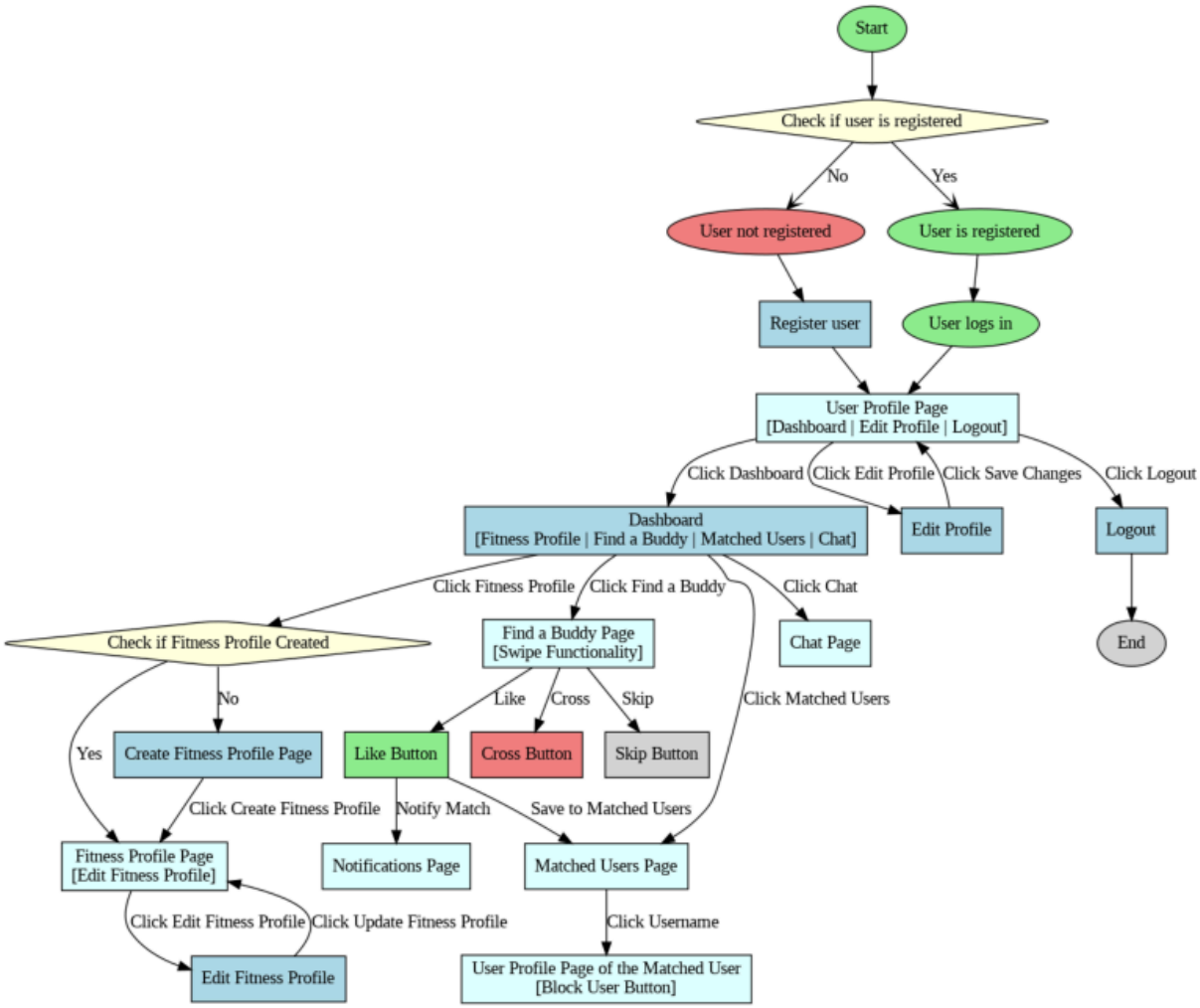
Sprint Backlog Items and Status:

The current sprint has no remaining backlog items. All sprint goals have been completed. The team has achieved everything planned for this sprint.

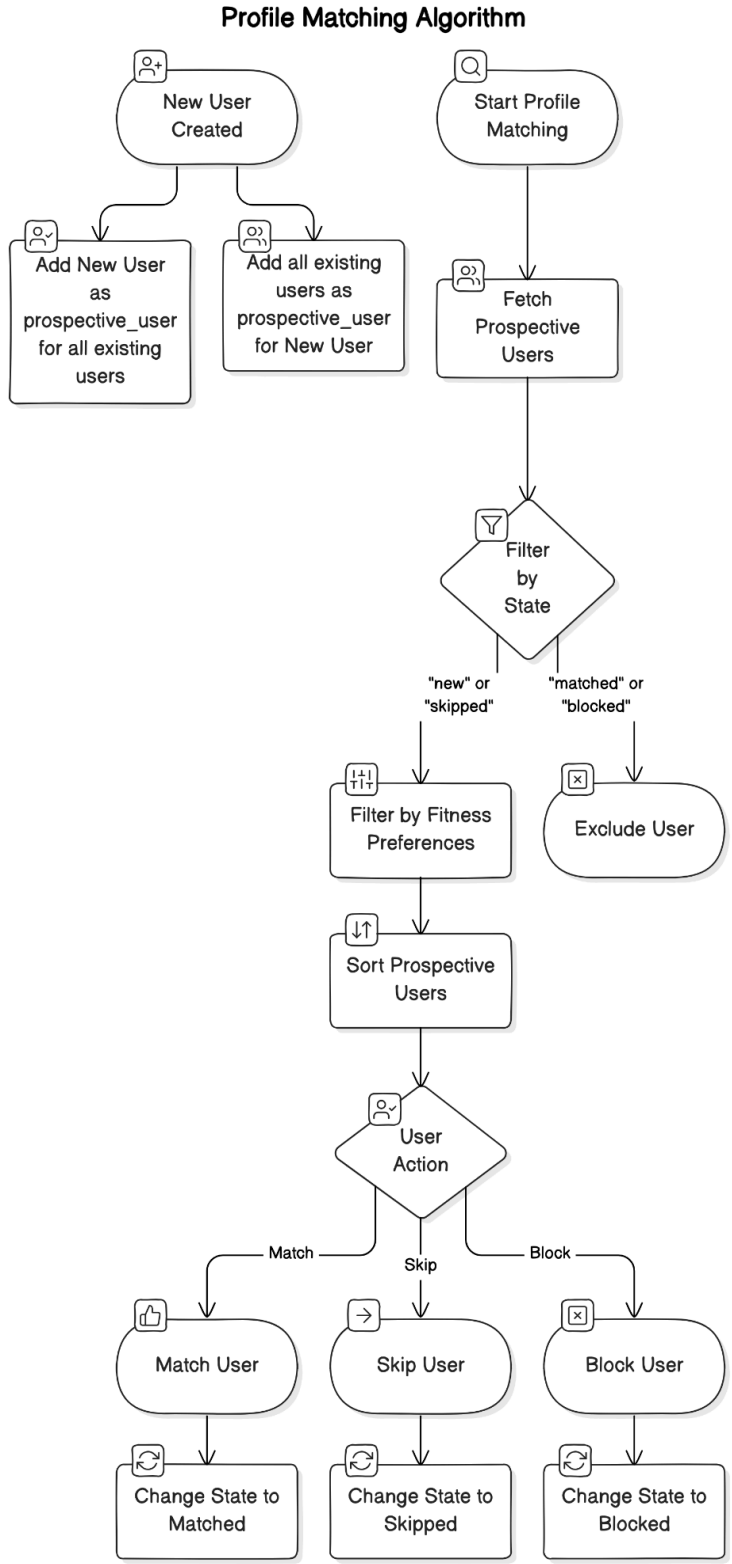
Burndown:



Design Diagram:



Profile Matching Algorithm Design Diagram:



Documentation of Changes:

We did not incorporate any changes, and everything was implemented as per the plan.

Evaluation of Code and Test Quality:

Our project has received an overall quality rating of A, signifying strong compliance with coding standards and best practices. We leveraged **SimpleCov** and **CodeClimate** to assess both our code and test quality, focusing specifically on metrics like coverage, code smells, and adherence to style guidelines.

- **SimpleCov Score**: Our current test coverage, as measured by SimpleCov, stands at **96.54%** for RSpec and **92.51%** for Cucumber scenarios. Combined, these yield an **98.56%** overall coverage across both RSpec and Cucumber, indicating a thorough approach to testing. Each team member initially developed individual test cases and Cucumber scenarios for their respective features, with all tests passing during their feature testing.
- **Code Smells**: Our analysis revealed **0 code smells** and **2 Duplications**. While minimal, addressing this will enhance code readability and maintainability. Although our project has achieved an B rating, we are committed to resolving the remaining code smell and further improving test coverage in future iterations to ensure a high-quality, sustainable codebase.

Customer Meeting - Demo for Sprint 3 MVP:

Date: 23rd October, 2024 & 30th October, 2024
Time: 10 am - 10.30 am
Place: Zoom Call

Client Meeting Summary:

In our recent client meeting, we presented the progress achieved in this sprint, demonstrating the new features and improvements implemented across various sections of the app, including the fitness profile, profile matching, notifications, and chat functionalities. We highlighted the enhancements made based on prior client feedback, particularly in the UI design, which were well-received.

The client was pleased with the advancements, particularly noting the refined profile matching and fitness profile features. They commended the addition of activity preferences in the fitness profile, which allows users to better define workout buddy criteria, adding a personalized dimension to the user experience.

Regarding the profile matching feature, the client appreciated the integration of user preferences and real-time updates, as well as the added CSS adjustments that improve mobile responsiveness. This progress aligns closely with their vision, as it ensures a visually consistent and responsive experience across devices.

The client also expressed enthusiasm for the newly implemented notification system, especially the real-time matching notifications, which they felt would enhance engagement. They were also pleased with the private chat feature, which now supports message persistence, giving users a more reliable means to maintain their communication history.

On the client's suggestion, we discussed adding a "block" feature. It was confirmed that this functionality would be accessible both in the profile matching view and after a match, granting users more control over interactions. Additionally, they proposed a sequence for displaying users in the profile matching section: unmatched users first, followed by previously skipped ones, which we plan to incorporate for improved user flow.

The client was satisfied with the overall sprint achievements, which included nine significant user stories focused on essential enhancements. Moving forward, we will prioritize the remaining UI updates they suggested to further refine the app's interface, focusing on improving intuitiveness and visual consistency.

This meeting reinforced our alignment with the client's vision, and we are excited to move forward with their continued support and feedback in the upcoming sprint.

Bdd and Tdd:

Bdd:

profile_swipe.feature

```
# features/profile_swipe.feature

Feature: Profile Swiping

  Background:
    Given I am logged in as a user for profile swipe

  Scenario: View profile swipe page
    When I visit the profile swipe page
    Then I should see the profile swipe container
    And I should see the action buttons
```

profile_swipe_steps

```ruby
# features/step_definitions/profile_swipe_steps.rb

require 'rspec/expectations'
require 'rspec/mocks'

# In features/step_definitions/profile_swipe_steps.rb

Given("I am logged in as a user for profile swipe") do
    @user = FactoryBot.create(:user, :complete_profile)
    page.set_rack_session(user_id: @user.id)
  end

# In features/step_definitions/profile_swipe_steps.rb

When("I visit the profile swipe page") do
    allow(UserMatchesController).to receive(:get_prospective_users).and_return([
      { 'id' => 1, 'username' => 'user1', 'age' => 25, 'fitness_profile' => { 'activities_with_experience' => 'Running:Intermediate' } },
      { 'id' => 2, 'username' => 'user2', 'age' => 30, 'fitness_profile' => { 'activities_with_experience' => 'Swimming:Beginner' } }
    ])
    visit matching_profileswipe_path(@user.id)
  end

Then("I should see the profile swipe container") do
  expect(page).to have_css('.profile-cards')
end

And("I should see the action buttons") do
  expect(page).to have_css('button[onclick="handleAction(\'block\')"]')
  expect(page).to have_css('button[onclick="handleAction(\'skip\')"]')
  expect(page).to have_css('button[onclick="handleAction(\'like\')"]')
end
```

The purpose of this feature is to allow users to browse through potential workout partners by swiping through their profiles. Users can view basic information about other users, such as username, age, and fitness activities with experience levels. This functionality helps users connect with compatible partners based on shared fitness interests, enhancing the social aspect of the application.

Scenario Covered:

View Profile Swipe Page:

This scenario ensures that a logged-in user can access the profile swipe page and interact with it as intended.

The user should be able to see the profile swipe container displaying prospective user profiles.

Action buttons for "like," "skip," and "block" should be present, allowing the user to express their interest or disinterest in each profile.

user_matches.feature

```
Feature: User Matches
  As a user
  I want to manage my potential matches
  So that I can find suitable workout partners

  Background:
    Given I am logged in as a user

  Scenario: View prospective users
    When I request to view prospective users
    Then I should see a list of filtered prospective users

  Scenario: Match with a prospective user
    Given there is a prospective user
    When I match with the prospective user
    Then I should see a success message
    And a match should be created in the database

  Scenario: Skip a prospective user
    Given there is a prospective user
    When I skip the prospective user
    Then I should see a success message
    And a skip record should be created in the database

  Scenario: Block a prospective user
    Given there is a prospective user
    When I block the prospective user
    Then I should see a success message
    And a block record should be created in the database

  Scenario: Attempt to match with oneself
    When I try to match with myself
    Then I should see an error message

  Scenario: Attempt to skip oneself
    When I try to skip myself
    Then I should see an error message

  Scenario: Attempt to block oneself
    When I try to block myself
    Then I should see an error message
```

```
Scenario: Mutual match creates notifications
  Given there is a prospective user who has matched with me
  When I match with the prospective user
  Then I should see a success message
  And notifications should be created for both users

Scenario: Failed match
  Given there is a prospective user
  And the match will fail to save
  When I match with the prospective user
  Then I should see a failure message

Scenario: Failed skip
  Given there is a prospective user
  And the skip will fail to save
  When I skip the prospective user
  Then I should see a failure message

Scenario: Failed block
  Given there is a prospective user
  And the block will fail to save
  When I block the prospective user
  Then I should see a failure message

Scenario: View filtered prospective users
  Given there are multiple prospective users with various profiles
  When I request to view prospective users
  Then I should see a list of filtered prospective users
  And the list should only include users matching my preferences
```

user_matches_steps

```ruby
require 'rack/test'
require 'rspec/mocks/standalone'
World(RSpec::Mocks::ExampleMethods)
World(Rack::Test::Methods)

Given("I am logged in as a user") do
  @user = FactoryBot.create(:user, :complete_profile)
  page.set_rack_session(user_id: @user.id)
end

When("I request to view prospective users") do
  visit "/users/#{@user.id}/prospective_users.json"
  @response_body = JSON.parse(page.body)
end

Then("I should see a list of filtered prospective users") do
  expect(page.status_code).to eq(200)
  expect(@response_body).to be_an(Array)
end

Given("there is a prospective user") do
  @prospective_user = FactoryBot.create(:user, :complete_profile)
end

When("I match with the prospective user") do
  @current_action = "match"
  visit "/users/#{@user.id}/match/#{@prospective_user.id}"
end

When("I skip the prospective user") do
  @current_action = "skip"
  page.driver.post "/users/#{@user.id}/skip/#{@prospective_user.id}"
end

When("I block the prospective user") do
  @current_action = "block"
  page.driver.post "/users/#{@user.id}/block/#{@prospective_user.id}"
end
```

```ruby
Then("I should see a success message") do
  expect(page.status_code).to eq(200)

  success_messages = {
    "match" => "Matched successfully.",
    "skip" => "Skipped successfully.",
    "block" => "Blocked successfully."
  }

  expected_message = success_messages[@current_action]
  expect(JSON.parse(page.body)["message"]).to eq(expected_message)
end

And("a match should be created in the database") do
  expect(UserMatch.find_by(user_id: @user.id, prospective_user_id: @prospective_user.id, status: "matched")).to be_present
end

And("a skip record should be created in the database") do
  expect(UserMatch.find_by(user_id: @user.id, prospective_user_id: @prospective_user.id, status: "skipped")).to be_present
end

And("a block record should be created in the database") do
  expect(UserMatch.find_by(user_id: @user.id, prospective_user_id: @prospective_user.id, status: "blocked")).to be_present
end

When("I try to match with myself") do
  visit "/users/#{@user.id}/match/#{@user.id}"
end

Then("I should see an error message") do
  expect(page.status_code).to eq(422)
  error_message = JSON.parse(page.body)["error"]
  expect(error_message).to match(/You cannot (match|skip|block) yourself/)
end

Given("there is a prospective user who has matched with me") do
  @prospective_user = FactoryBot.create(:user, :complete_profile)
  UserMatch.create(user_id: @prospective_user.id, prospective_user_id: @user.id, status: "matched")
end

And("notifications should be created for both users") do
  expect(Notification.find_by(user: @user, matched_user: @prospective_user)).to be_present
  expect(Notification.find_by(user: @prospective_user, matched_user: @user)).to be_present
end
```

```ruby
When("I try to skip myself") do
  page.driver.post "/users/#{@user.id}/skip/#{@user.id}"
end

When("I try to block myself") do
  page.driver.post "/users/#{@user.id}/block/#{@user.id}"
end

Given("the match will fail to save") do
  allow_any_instance_of(UserMatch).to receive(:save).and_return(false)
end

Given("the skip will fail to save") do
  allow_any_instance_of(UserMatch).to receive(:save).and_return(false)
end

Given("the block will fail to save") do
  allow_any_instance_of(UserMatch).to receive(:save).and_return(false)
end

Then("I should see a failure message") do
  expect(page.status_code).to eq(422)
  expect(JSON.parse(page.body)["error"]).to include("Failed to")
end

Given("there are multiple prospective users with various profiles") do
  # Set up the current user's fitness profile
  @user.fitness_profile.update!(
    age_range_start: 25,
    age_range_end: 35,
    gender_preferences: "Female,Non-binary",
    gym_locations: "Gym A,Gym B",
    activities_with_experience: "Running:Intermediate|Swimming:Beginner",
    workout_schedule: "Morning=06:00-08:00|Evening=18:00-20:00",
    workout_types: "Cardio,Strength"
  )

  # Create a matching user
  @matching_user = FactoryBot.create(:user,
    age: 30,
    gender: "Female",
    username: "MatchingUser"
  )
```

```
  @matching_user.fitness_profile.update!(
    age_range_start: 25,
    age_range_end: 35,
    gender_preferences: "Male,Non-binary",
    gym_locations: "Gym A",
    activities_with_experience: "Running:Advanced|Swimming:Intermediate",
    workout_schedule: "Morning=06:00-08:00",
    workout_types: "Cardio"
  )

  # Create a non-matching user
  @non_matching_user = FactoryBot.create(:user,
    age: 40,
    gender: "Male",
    username: "NonMatchingUser"
  )
  @non_matching_user.fitness_profile.update!(
    age_range_start: 35,
    age_range_end: 45,
    gender_preferences: "Female",
    gym_locations: "Gym C",
    activities_with_experience: "Yoga:Beginner",
    workout_schedule: "Afternoon=14:00-16:00",
    workout_types: "Flexibility"
  )

  # Create UserMatch entries
  UserMatch.create!(user_id: @user.id, prospective_user_id: @matching_user.id, status: "new")
  UserMatch.create!(user_id: @user.id, prospective_user_id: @non_matching_user.id, status: "new")
end

Then("the list should only include users matching my preferences") do
  response_body = JSON.parse(page.body)
  puts "Response body: #{response_body}"
  puts "Matching user ID: #{@matching_user.id}"
  puts "Non-matching user ID: #{@non_matching_user.id}"
  expect(response_body.length).to eq(1)
  expect(response_body.first["id"]).to eq(@matching_user.id)
end
```

The **User Matches** feature allows users to manage potential workout partners by viewing, matching, skipping, and blocking prospective users. It also includes handling for special cases, such as preventing self-matching and managing mutual matches, to ensure the user experience is seamless and intuitive. This feature helps users connect with others based on shared fitness preferences, facilitating meaningful connections within the app.

Scenarios Covered

1. **View Prospective Users**:

   Allows the user to request and see a list of filtered prospective users based on their profile preferences.

2. **Match with a Prospective User**:

   Enables the user to select a match, creating a record in the database and showing a success message.

3. **Skip a Prospective User**:

   Lets the user skip a profile they are not interested in, with a skip record created in the database and a success message displayed.

4. **Block a Prospective User**:

   Provides the option to block a user, removing them from future matches and recording this in the database.

5. **Attempt to Match/Skip/Block Oneself**:

   Handles cases where the user attempts to match, skip, or block their profile by mistake, returning an error message to prevent such actions.

6. **Mutual Match Creates Notifications**:

   Ensures that if a user matches with someone who has already matched with them, notifications are created for both users.

7. **Failed Match, Skip, or Block Actions**:

   Simulates failure in saving a match, skip, or block action to confirm that an error message is returned.

8. **View Filtered Prospective Users**:

   Ensures that only profiles matching the user's preferences are shown, with non-matching profiles excluded from the list.

TDD:

fitness_profile_spec

```ruby
require 'rails_helper'

RSpec.describe FitnessProfile, type: :model do
  let(:user) { create(:user) }

  it 'is valid with valid attributes' do
    fitness_profile = FitnessProfile.new(
      fitness_goals: 'Lose weight',
      workout_types: 'Running',
      gender: 'Male',
      age_range_start: '18',
      age_range_end: '28',
      gym_locations: 'Student Rec Center, Polo Road Rec Center',
      gender_preferences: 'Male, Female',
      activities_with_experience: 'Soccer:Amateur|Basketball:Beginner',
      workout_schedule: 'Monday=06:00-07:00|Wednesday=08:00-09:00',
      user: user
    )
    expect(fitness_profile).to be_valid
  end

  it 'is not valid without a fitness goal' do
    fitness_profile = FitnessProfile.new(
      fitness_goals: nil,
      gym_locations: 'Student Rec Center, Polo Road Rec Center',
      user: user
    )
    expect(fitness_profile).not_to be_valid
  end

  it 'is not valid without a user' do
    fitness_profile = FitnessProfile.new(
      fitness_goals: 'Lose weight',
      gym_locations: 'Student Rec Center, Polo Road Rec Center',
      user: nil
    )
    expect(fitness_profile).not_to be_valid
  end

  describe '#activities_array' do
    it 'returns an array of activities with experience levels' do
      fitness_profile = FitnessProfile.new(
        activities_with_experience: 'Soccer:Amateur|Basketball:Beginner'
      )
```

```ruby
    expected_result = [
      { 'activity' => 'Soccer', 'experience' => 'Amateur' },
      { 'activity' => 'Basketball', 'experience' => 'Beginner' }
    ]
    expect(fitness_profile.activities_array).to eq(expected_result)
  end

  it 'returns an empty array if activities_with_experience is nil' do
    fitness_profile = FitnessProfile.new(activities_with_experience: nil)
    expect(fitness_profile.activities_array).to eq([])
  end
end

describe '#schedule_hash' do
  it 'returns a hash of workout days with start and end times' do
    fitness_profile = FitnessProfile.new(
      workout_schedule: 'Monday=06:00-07:00|Wednesday=08:00-09:00'
    )
    expected_result = {
      'Monday' => { 'start' => '06:00', 'end' => '07:00' },
      'Wednesday' => { 'start' => '08:00', 'end' => '09:00' }
    }
    expect(fitness_profile.schedule_hash).to eq(expected_result)
  end

  it 'returns an empty hash if workout_schedule is nil' do
    fitness_profile = FitnessProfile.new(workout_schedule: nil)
    expect(fitness_profile.schedule_hash).to eq({})
  end
end

describe '#workout_types_array' do
  it 'returns an array of workout types' do
    fitness_profile = FitnessProfile.new(workout_types: 'Running, Swimming, Cycling')
    expected_result = [ 'Running', 'Swimming', 'Cycling' ]
    expect(fitness_profile.workout_types_array).to eq(expected_result)
  end

  it 'returns an empty array if workout_types is nil' do
    fitness_profile = FitnessProfile.new(workout_types: nil)
    expect(fitness_profile.workout_types_array).to eq([])
  end
```

```ruby
  end

  describe '#gym_locations_array' do
    it 'returns an array of gym locations' do
      fitness_profile = FitnessProfile.new(gym_locations: 'Student Rec Center, Southside Rec Center')
      expected_result = [ 'Student Rec Center', 'Southside Rec Center' ]
      expect(fitness_profile.gym_locations_array).to eq(expected_result)
    end

    it 'returns an empty array if gym_locations is nil' do
      fitness_profile = FitnessProfile.new(gym_locations: nil)
      expect(fitness_profile.gym_locations_array).to eq([])
    end
  end

  describe '#gender_preferences_array' do
    it 'returns an array of gender preferences' do
      fitness_profile = FitnessProfile.new(gender_preferences: 'Male, Female')
      expected_result = [ 'Male', 'Female' ]
      expect(fitness_profile.gender_preferences_array).to eq(expected_result)
    end

    it 'returns an empty array if gender_preferences is nil' do
      fitness_profile = FitnessProfile.new(gender_preferences: nil)
      expect(fitness_profile.gender_preferences_array).to eq([])
    end
  end
end
```

The purpose of this test suite is to ensure that the **FitnessProfile** model is working as expected. It verifies that:

1. FitnessProfile objects are valid only when essential attributes are present.
2. Model methods correctly parse and transform data fields into more usable structures, such as arrays and hashes.
3. Each aspect of the fitness profile, such as goals, workout preferences, locations, and schedules, behaves correctly, enabling reliable data handling in the application.

Test Breakdown

1. **Validation Tests**:
    - **Valid Fitness Profile**: Confirms that a FitnessProfile is valid when all required attributes (like fitness_goals, workout_types, user, etc.) are provided.
    - **Missing Fitness Goal**: Ensures that the profile is invalid if the fitness_goals attribute is missing.
    - **Missing User**: Ensures that the profile is invalid if the user association is missing.
2. **Method-Specific Tests**:

- **#activities_array**:
  - Tests that activities_with_experience correctly returns an array of activity and experience level pairs.
  - Checks for an empty array if activities_with_experience is nil.
- **#schedule_hash**:
  - Confirms that workout_schedule is parsed into a hash with each workout day, including start and end times.
  - Ensures that an empty hash is returned if workout_schedule is nil.
- **#workout_types_array**:
  - Verifies that workout_types converts into an array of individual workout types.
  - Checks for an empty array if workout_types is nil.
- **#gym_locations_array**:
  - Ensures that gym_locations is split into an array of individual gym locations.
  - Returns an empty array if gym_locations is nil.
- **#gender_preferences_array**:
  - Confirms that gender_preferences is parsed into an array of gender options.
  - Returns an empty array if gender_preferences is nil.

user_match _job_spec

```ruby
# spec/jobs/user_match_job_spec.rb
require 'rails_helper'

RSpec.describe UserMatchJob, type: :job do
  let!(:user) { create(:user) }
  let!(:prospective_user1) { create(:user) }
  let!(:prospective_user2) { create(:user) }

  it 'creates matches for each prospective user with the "new" status' do
    UserMatchJob.perform_now(user)
    expect(UserMatch.where(user_id: user.id, status: 'new').pluck(:prospective_user_id)).to contain_exactly(prospective_user1.id, prospective_user2.id)
  end

  it 'creates reciprocal matches for each prospective user with the "new" status' do
    UserMatchJob.perform_now(user)
    expect(UserMatch.where(user_id: prospective_user1.id, prospective_user_id: user.id, status: 'new')).to exist
    expect(UserMatch.where(user_id: prospective_user2.id, prospective_user_id: user.id, status: 'new')).to exist
  end

  it 'does not create a match for the user with themselves' do
    UserMatchJob.perform_now(user)
    expect(UserMatch.where(user_id: user.id, prospective_user_id: user.id)).not_to exist
  end

  it 'does not duplicate existing matches' do
    UserMatch.create!(user_id: user.id, prospective_user_id: prospective_user1.id, status: 'new')
    expect {
      UserMatchJob.perform_now(user)
    }.not_to change { UserMatch.where(user_id: user.id, prospective_user_id: prospective_user1.id).count }
  end
end
```

The purpose of this test suite is to ensure that **UserMatchJob** correctly manages the creation of match records between users and prospective users, based on certain conditions. The job is expected to create new matches, avoid duplicates, establish reciprocal matches, and prevent self-matching. By thoroughly testing these scenarios, the suite helps maintain the reliability of the matchmaking feature within the application.

Test Breakdown

1. **Match Creation Test**:
    - **Purpose**: Verifies that UserMatchJob creates new match records with a "new" status for each prospective user.
    - **Test**: After running the job, it checks that the job has created matches with the "new" status for both prospective_user1 and prospective_user2.
2. **Reciprocal Match Creation Test**:
    - **Purpose**: Ensures that the job creates reciprocal match records (where each prospective user has a match record with the main user).
    - **Test**: After the job runs, it confirms that reciprocal match records are present for both prospective_user1and prospective_user2, with user as the prospective user for each.
3. **Self-Matching Prevention Test**:
    - **Purpose**: Confirms that the job does not create a match between a user and themselves.
    - **Test**: Checks that no UserMatch record is created with both user_id and prospective_user_id set to user.id.
4. **Duplicate Match Prevention Test**:
    - **Purpose**: Ensures that if a match already exists, the job does not create a duplicate.
    - **Test**: Sets up an existing match with prospective_user1 and then verifies that running the job does not increase the count of matches for that user-prospective user pair.