California State University, Fullerton

Department of Computer Science

MINGLE

Design Document

CPSC 411 – iOS Mobile Application Development

Team Name: Mingle

Roszhan Raj Meenakshi Sundhresan (roszhan.2684@csu.fullerton.edu)
Tanmay Chaudhari (tanmay_2106@csu.fullerton.edu)
Vanessa Iwaki-Honore (vanessa_ih@csu.fullerton.edu)
Sunny Palaco (sunnypalaco@csu.fullerton.edu)

Version: v1.0

Contents

List of Figures

1 Introduction

1.1 Problem Context and Motivation

University life is full of informal events: birthday parties, movie nights, game tournaments, potlucks, club meetings, and last-minute study sessions. Most of these gatherings are coordinated using generic messaging platforms such as WhatsApp, iMessage, Discord, or Instagram DMs. While these tools are excellent for chatting, they are not designed for structured event management.

Because of this mismatch, students often face the following problems:

- Event details get buried in long chat histories, making it hard to find the latest information.
- Attendees repeatedly ask the same questions:
  - "What time does it start again?"
  - "Where exactly is it?"
  - "Who else is coming?"
- Hosts cannot easily track who is actually planning to attend, who is unsure, and who has declined.
- There is no simple way for a student to open an app and quickly see everything happening soon in a clean, organized view.

These pain points can lead to:

- Lower event turnout because people forget or miss details.
- Confusion when plans change last minute (time or location changes).
- Extra mental load on hosts, who have to keep answering the same questions and manually tracking attendees.

Mingle is motivated by the need for a student-friendly, event-focused tool that reduces this chaos and makes social planning smoother.

1.2 Proposed Solution: Mingle

To address these issues, we propose **Mingle**, an iOS mobile application built with Swift and SwiftUI. Mingle focuses specifically on the creation, discovery, and management of social events for students.

At a high level, Mingle provides:

- A simple event creation flow for hosts:
  - Set a title, date, time, location, description, tags, and optional capacity.

- A clear event browsing experience for guests:
  - See upcoming events in a structured list.
  - Tap into any event for full details.
- A straightforward RSVP system:
  - Mark yourself as Going, Maybe, or Not Going.
  - Allow the host to see how many people are planning to attend.
- Optional capacity rules:
  - Limit the number of "Going" attendees using a maximum capacity field.

The emphasis is on:

- **Clarity** – all critical information (what, when, where, who) on one screen.
- **Simplicity** – minimal steps to create an event or respond as a guest.
- **Student context** – optimized for small to medium-sized gatherings common in university life.

1.3 Scope and Objectives of the Project

The scope of Mingle is intentionally focused and realistic for a course project in iOS development. Mingle is designed as a single-device application (no live backend) that still demonstrates strong software design and implementation.

The project aims to:

- Model a real-world problem (student event coordination) and convert it into a concrete app.
- Demonstrate core iOS concepts:
  - Swift language fundamentals (data types, functions, control flow, collections).
  - Object-oriented design (structs, classes, properties, error handling).
  - SwiftUI layouts, navigation, and state management.
- Implement key features needed for an event app, including:
  - Event creation, listing, details, and RSVP logic.
  - Sorting and filtering events using data structures and algorithms.
  - Optional capacity enforcement to prevent overbooking.
- Produce a complete deliverable for the course:
  - A working iOS application.
  - A clean, well-structured GitHub repository.

  - This design document and a demo video walking through all major functionalities.

Out of scope for this version are:

- Real-time synchronization across multiple devices.
- User authentication with a remote server.
- Push notifications using external services.

However, the architecture is designed so these could be added in a future version.

1.4 Target Users and Usage Scenarios

Mingle is designed around two main user roles:

- **Hosts**
  - Students who create and organize events such as birthday parties, game nights, study sessions, club meetings, or socials.
  - They need a fast, reliable way to enter event details once, share an event, and see who is planning to attend.
- **Guests**
  - Students who receive invites or browse upcoming events.
  - They want to quickly scan what events are happening soon, open an event to see full details, RSVP with one tap, and optionally change their response later.

Typical usage scenarios include:

- A host creating a Game Night event, setting capacity to 10 people, and sharing it with friends.
- A guest opening Mingle on a Friday afternoon and checking "What events are happening this weekend?".
- A host checking attendee counts to decide how much food or snacks to buy.

1.5 Team Composition and Collaboration

Mingle was developed as a group project by a team of four students. The project structure intentionally supports collaboration, both technically and organizationally.

Each team member contributed to different aspects of the app:

- **UI/UX & SwiftUI Layout**: Designing screen flows, arranging views using stacks and lists, and ensuring the interface is intuitive for students.

- **Data Model & Business Logic**: Designing the `Event`, `User`, and `EventRSVP` models, implementing the MVVM architecture, and writing the RSVP and capacity logic.
- **State Management & Navigation**: Setting up `TabView`, `NavigationStack`, view models, and shared state across screens.
- **Testing, Polish & Documentation**: Running the app through different scenarios, fixing edge cases, preparing the GitHub repo, and writing this design document.

The team collaborated using GitHub for source control, regular check-ins to divide tasks and review progress, and shared design references for naming conventions and coding standards.

2 Learning Outcomes Alignment

This section explains how the Mingle project satisfies the course learning outcomes. Each subsection maps a requirement from the assignment to concrete design and implementation choices in the app.

2.1 Designing an iOS Application that Solves a Real-World Problem

Outcome: Design an iOS mobile application written in Swift that solves a real-world problem.

Mingle addresses a realistic, everyday problem faced by university students: organizing and joining social events efficiently without losing important details in chat threads.

Concretely, Mingle:

- Centralizes event information into a dedicated app.
- Allows users to see what the event is, when and where it takes place, and who is going.
- Replaces disorganized chat threads with a clean, structured list of upcoming events.

From a software design perspective, Mingle shows that the team can identify real-world pain points, translate them into features and UI flows, and implement those features as a functioning iOS application.

2.2 Basic Swift Programming Constructs

Outcome: Design and write Swift code that makes appropriate use of basic programming constructs (data types, constants, variables, operators, expressions, control flow, functions, methods, and closures).

Throughout the codebase, we use:

- **Data types**: `String`, `Int`, `Bool`, `Date`, and custom enums like `RSVPStatus`.
- **Constants and variables** using `let` and `var`.

- **Control flow**: `if`/`else`, `guard`, `switch`, and `for`-loops.
- **Functions and methods**: helper methods such as `createEvent(...)` and `rsvp(to:as:for:)`.
- **Closures**: used in SwiftUI button handlers and high-order methods like `filter` and `sorted`.

These constructs are integrated into logic for event creation, listing, filtering, and RSVP behaviour.

2.3 Object-Oriented Programming in Swift

Outcome: Design and write Swift code that makes appropriate use of object-oriented programming constructs (classes, inheritance, structures, properties, collections, and error handling).

In Mingle:

- Domain models (`Event`, `User`, `EventRSVP`) are structs conforming to `Identifiable` and `Codable`.
- `EventViewModel` is a class conforming to `ObservableObject`, with `@Published` properties such as `events` and `rsvps`.
- Collections (`[Event]`, `[EventRSVP]`) are encapsulated within view models.
- Error handling via `do--try--catch` is used when encoding and decoding persistent data (when enabled).

This separation between structs (data) and classes (logic) matches modern Swift design principles.

2.4 Data Structures and Algorithms

Outcome: Design and implement software that makes appropriate use of data structures and algorithms to solve a well-posed computational problem using Swift.

Mingle uses:

- Arrays to store events and RSVPs.
- Filtering to obtain upcoming events or events for a given tag.
- Sorting to order events chronologically by date.
- Counting and searching to implement capacity checks and RSVP updates.

These algorithms are simple but sufficient to solve the core problem of managing events and responses.

2.5 SwiftUI Views, Layouts, and Interactions

Outcome: Implement visual designs and interactions using SwiftUI.

Mingle's UI includes:

- Layout primitives (`VStack`, `HStack`, `ZStack`, `ScrollView`, `List`).
- Form controls (`TextField`, `TextEditor`, `DatePicker`, `Button`).
- Navigation components (`TabView`, `NavigationStack`, `NavigationLink`).
- State management with `@State`, `@StateObject`, `@ObservedObject`, and `@EnvironmentObject`

These constructs create a responsive, declarative UI that updates automatically when underlying data changes.

2.6 Readable and Maintainable Swift Code

Outcome: Write readable Swift code.

We follow:

- PascalCase for type names and camelCase for variables and functions.
- Logical grouping of files into `Models`, `ViewModels`, and `Views`.
- Short, targeted comments for non-trivial logic (for example, RSVP capacity enforcement).
- Avoiding deeply nested structures by using helper functions and subviews.

2.7 Group Collaboration and Contributions

Outcome: Contribute to the development of a group mobile application project.

Responsibilities were divided among team members but integrated into a single coherent project using GitHub for branching, merging, and code review. All four members contributed to design decisions, implementation, testing, and documentation.

2.8 Version Control and Final Code Delivery

The final codebase is maintained in a GitHub repository:

- The `main` branch contains the final working version of the app.
- The repository includes a `README` with build instructions.
- Commit history shows contributions from all team members.

2.9 Design Document and Demo Submission

This design document and a demo video complete the final deliverables. The demo presents all major flows—creating events, viewing lists, exploring details, RSVP actions, and the My Events screen—and ties them back to the learning outcomes.

3 Requirements

This section specifies what Mingle must do (functional requirements) and how it should behave (non-functional requirements).

3.1 Functional Requirements (FR)

FR-1: Manage Current User Profile

Mingle must maintain a `User` instance representing the current user. The profile stores at least a display name and may also include email and avatar URL. This user acts as host when creating events and as guest when RSVPing. The user's name appears on events they host and can be used in attendee lists.

FR-2: Create Events

The app must provide a *Create Event* screen where users can input:

- Event title (required).
- Event description.
- Date and time (required).
- Location name (required).
- Optional maximum capacity.
- Optional tags.

On save, the app validates required fields, constructs a new `Event` with a unique ID and `hostId` equal to the current user's ID, appends it to the event list, and navigates back to the list view. If validation fails, the event is not created and a clear error message is displayed.

FR-3: View Events (All Events & My Events)

Mingle must provide:

- An **All Events** view showing all events with basic summary information (title, date/time, location, host).
- A **My Events** view showing only events where `hostId == currentUser.id` and allowing navigation to the Create Event screen.

Events should be sorted in ascending order by date/time and may exclude past events from the default view.

FR-4: View Event Details

Tapping an event navigates to an Event Detail screen displaying:

- Title and host name.

- Date and time.
- Location name.
- Full description text.
- Tags (if any).
- Capacity information: maximum capacity (if set) and current Going count.
- The current user's RSVP status.

Past events can be visually marked as already completed.

FR-5: RSVP to Events

The Event Detail screen must provide options to RSVP as Going, Maybe, or Not Going. When the user selects a status:

- The app creates or updates an `EventRSVP` record linking event ID, user ID, status, and timestamp.
- The UI refreshes to highlight the selected status.
- If the event has a `maxCapacity`, the app counts existing Going RSVPs and blocks additional Going responses once capacity is reached, showing an "Event is full" message.

Changing from Going to another status must reduce the effective Going count.

FR-6: Event Filtering and Tags (Optional)

If implemented, Mingle allows users to select a tag and see only events that include that tag in their `tags` array. Clearing the filter restores the full upcoming event list.

FR-7: Local Data Persistence (Optional)

If persistence is enabled:

- Events and RSVPs are encoded using `Codable` and stored (for example in `UserDefaults` or a JSON file).
- On launch, the app attempts to load and decode saved data.
- Missing or corrupted data is handled gracefully without crashing the app.

FR-8: In-App Help Chat (Optional)

Mingle may include a simple in-app chat assistant that helps users understand how to use the app. The assistant answers common questions about creating events, RSVPing, and navigating screens.

The assistant:

- Responds with short, straightforward explanations.
- Assumes user questions are related to Mingle unless stated otherwise.

- Treats “event” and “party” as the same concept for clarity.

This feature exists to reduce confusion and repetitive questions without changing any core app behavior.

3.2 Non-Functional Requirements (NFR)

NFR-1: Usability and User Experience

The UI must be simple, uncluttered, and intuitive. Key actions (such as browsing events and RSVP) should be reachable within one or two taps. Forms must use appropriate controls (e.g., `DatePicker`) and provide clear labels and error messages.

NFR-2: Performance and Responsiveness

The app should feel responsive for typical workloads (dozens of events). Event lists and detail views must load and update smoothly. Creating an event or changing RSVP status should immediately update relevant screens.

NFR-3: Reliability and Robustness

The app must handle:

- Empty states when no events exist.
- Optional fields such as capacity being nil.
- Invalid input by preventing event creation rather than crashing.

If persistence is used, file or decoding errors should be caught and handled.

NFR-4: Maintainability and Extensibility

Separation into models, view models, and views should make it straightforward to add features like authentication, backend connectivity, notifications, or more advanced filtering in future versions without major rewrites.

NFR-5: Readability and Code Quality

Consistent naming, a clean file structure, and targeted comments are required so that new contributors and reviewers can understand the project quickly.

NFR-6: Collaboration and Version Control

All work must be committed to a shared GitHub repository. The final stable version resides on the `main` branch, with meaningful commit messages reflecting incremental development.

4 System Design & Architecture

Mingle is implemented as a single iOS app using Swift, SwiftUI, and the MVVM architectural pattern.

4.1 High-Level Overview

Mingle consists of three main layers:

1. **Model Layer** – core data structures (`Event`, `User`, `EventRSVP`, `RSVPStatus`).
2. **ViewModel Layer** – business logic and state (`EventViewModel`, optionally `UserViewModel`).
3. **View Layer** – SwiftUI screens (`ContentView`, `HomeView`, `MyEventsView`, `EventDetailView`, `CreateEventView`, `ProfileView`, and reusable components like `EventCardView`).

Data flows in a unidirectional cycle: View $\rightarrow$ ViewModel $\rightarrow$ Model $\rightarrow$ View. Views observe view models using `@ObservedObject` or `@EnvironmentObject`.

4.2 Architectural Pattern: MVVM in Mingle

- Models are pure data types implemented as Swift structs.
- ViewModels are classes conforming to `ObservableObject`, exposing `@Published` properties to trigger UI updates.
- Views are SwiftUI structs that bind to view model data and call view model methods in response to user interactions.

This separation keeps business logic out of views and promotes testability.

4.3 Responsibilities of Each Layer

4.3.1 Model Layer

- Represent domain concepts and their fields.
- Provide small, self-contained computed properties (e.g., `isPast`).
- Avoid UI or persistence responsibilities.

4.3.2 ViewModel Layer

- Maintain application state (events, RSVPs, current user, filters).
- Implement business logic (create events, filter and sort events, perform RSVP updates and capacity checks).
- Optionally handle saving and loading from persistent storage.

4.3.3 View Layer

- Render data on screen using SwiftUI layout primitives.
- Handle user interactions (button taps, text input, navigation).
- Delegate non-trivial logic to view models.

4.4 Module and File Organization

The Xcode project is organized into:

- **Models/**: `Event.swift`, `User.swift`, `EventRSVP.swift`, `RSVPStatus.swift`.
- **ViewModels/**: `EventViewModel.swift`, optionally `UserViewModel.swift`.
- **Views/**: screen-level views (`ContentView.swift`, `HomeView.swift`, `MyEventsView.swift`, `EventDetailView.swift`, `CreateEventView.swift`, `ProfileView.swift`) and reusable components (`EventCardView.swift`, RSVP button views, etc.).

4.5 Data Flow: Key Use Cases

4.5.1 Viewing Upcoming Events

1. `ContentView` initializes `EventViewModel` using `@StateObject`.
2. The view model loads initial events (sample data or persisted data).
3. `HomeView` observes the view model and reads `upcomingEvents`, which filters and sorts `events`.
4. `HomeView` renders the result in a `List` or `ScrollView` using `EventCardView`.

4.5.2 Creating an Event

1. From `MyEventsView`, the user taps a "+" button.
2. `CreateEventView` appears with form fields bound to `@State` variables.
3. On tapping "Save", `CreateEventView` calls `viewModel.createEvent(...)`.
4. The view model validates input, constructs a new `Event`, appends it to `events`, and optionally persists data.
5. Because `events` is `@Published`, SwiftUI refreshes `MyEventsView` and `HomeView`.

4.5.3 RSVPing to an Event

1. From a list, the user taps an event to open `EventDetailView`.
2. `EventDetailView` queries the view model to determine the current user's RSVP status and counts.
3. When the user selects an RSVP option, the view calls `rsvp(to:as:for:)`.
4. The view model performs capacity checks, updates or creates an `EventRSVP`, and publishes changes.
5. The UI updates to reflect the new status and counts.

4.6 Frameworks and Dependencies

Mingle uses only standard Apple frameworks:

- Swift Standard Library for core types and algorithms.
- SwiftUI for declarative UI and state management.
- Combine (indirectly) via `ObservableObject` and `@Published`.
- Foundation for date handling and basic utilities.

4.7 Architectural Benefits for a 4-Person Team

The MVVM architecture and clear file structure support:

- Parallel development on different layers.
- Reduced merge conflicts because UI, models, and logic live in separate files.
- Easier onboarding for any new team member or reviewer.

5 Data Model

The data model defines Mingle's core entities and how they relate to one another.

5.1 User Model

```
struct User: Identifiable, Codable {
    var id: String
    var name: String
    var email: String?
    var avatarURL: String?
}
```

`User` represents an individual using the app, primarily the current user. The `id` uniquely identifies the user and is referenced by events and RSVPs.

5.2 Event Model

```
struct Event: Identifiable, Codable {
    var id: String
    var hostId: String
    var title: String
    var description: String
    var date: Date
```

```
    var locationName: String
    var maxCapacity: Int?
    var tags: [String]
    var createdAt: Date

    var isPast: Bool {
        date < Date()
    }

    var isUpcoming: Bool {
        !isPast
    }
}
```

`Event` is the core entity representing a gathering, with fields for host, title, description, date, location, optional capacity, tags, and creation time. Computed properties `isPast` and `isUpcoming` help filter events.

5.3 RSVPStatus Enum

```
enum RSVPStatus: String, Codable {
    case going
    case maybe
    case notGoing
    case none
}
```

`RSVPStatus` represents a user's RSVP state for an event and is used in both logic and UI.

5.4 EventRSVP Model

```
struct EventRSVP: Identifiable, Codable {
    var id: String
    var eventId: String
    var userId: String
    var status: RSVPStatus
    var updatedAt: Date
}
```

`EventRSVP` links a user to an event with a specific RSVP status and timestamp, enabling many-to-many relationships between users and events.

5.5 Relationships Between Models

- One user can host many events (`Event.hostId` points to `User.id`).
- One user can RSVP to many events (`EventRSVP.userId`).
- One event can have RSVPs from many users (`EventRSVP.eventId`).

This forms a many-to-many relationship between users and events mediated by `EventRSVP`.

5.6 Example Data Instances

```
let currentUser = User(
    id: "user-123",
    name: "Alex",
    email: "alex@example.com",
    avatarURL: nil
)

let event = Event(
    id: "event-456",
    hostId: currentUser.id,
    title: "Friday Game Night",
    description: "Board games and snacks. Bring your favorite game!",
    date: Date().addingTimeInterval(60 * 60 * 24), // tomorrow
    locationName: "Apartment 304",
    maxCapacity: 10,
    tags: ["Party", "Games"],
    createdAt: Date()
)

let rsvp = EventRSVP(
    id: "rsvp-789",
    eventId: event.id,
    userId: currentUser.id,
    status: .going,
    updatedAt: Date()
)
```

5.7 Extensibility of the Data Model

The data model is designed for extension with fields such as:

- `isPublic` (for public vs invite-only events).
- Location coordinates for map integration.
- Event cover images.
- Friend lists or preference tags for users.

6 Core Algorithms & Logic

This section describes the main algorithms and logical steps that power Mingle. Although the app is relatively small, it still relies on clear, well-structured logic to handle event creation, listing, filtering, and RSVP management.

6.1 Event Creation

Goal. Take user input from the event creation form and turn it into a valid `Event` object that appears in the app.

Inputs

From `CreateEventView`, the user provides:

- Title
- Description
- Date and time (via `DatePicker`)
- Location
- Optional capacity
- Optional tags (e.g., “Party”, “Study Session”)

Process

1. **Validate required fields**
    - Ensure that key fields such as title, date/time, and location are not empty.
    - If any required field is missing, the `ViewModel` does not create the event and the view shows an appropriate message or alert.
2. **Prepare the date and metadata**
    - The selected `Date` from the `DatePicker` is used directly as the event’s start time.
    - The `createdAt` timestamp is set to the current date/time using `Date()`.

3. **Create a new `Event` instance**
    - The `ViewModel` constructs a new `Event` with:
        - A unique `id` (e.g., `UUID().uuidString`).
        - `hostId` equal to the current user's `id`.
        - All other fields based on the user's input.
4. **Update the events collection**
    - The new event is appended to `events`, a `@Published var events: [Event] = []` inside `EventViewModel`.
    - Because `events` is `@Published`, SwiftUI automatically refreshes the UI in views that depend on it (`HomeView`, `MyEventsView`).
5. **Optional: Persist data**
    - If persistence is implemented, the updated list of events is encoded (for example, to JSON) and saved to local storage (such as `UserDefaults` or a file).

Output

- The event immediately appears in *My Events*, and if appropriate, also in *All Events*.
- The user is navigated back to the previous screen, giving a smooth flow.

6.2 Event Listing & Sorting

All events are stored in the `ViewModel` as:

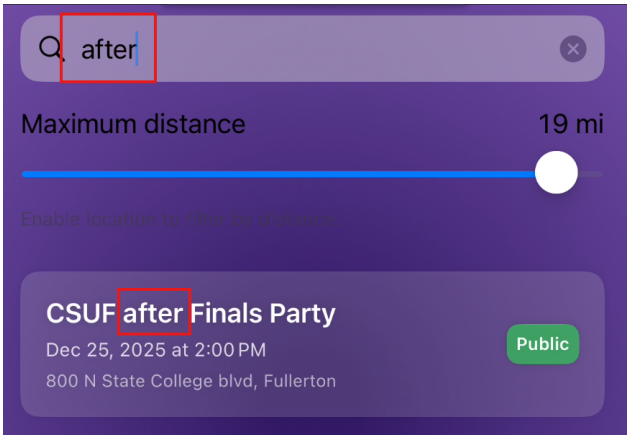
```
@Published var events: [Event] = []
```

To show events in a useful order, Mingle uses:

- Filtering to show only upcoming events.
- Sorting to order them chronologically.

Upcoming Events Logic

```
var upcomingEvents: [Event] {
    events
        .filter { $0.date >= Date() }
        .sorted { $0.date < $1.date }
}
```



Explanation

- `filter { $0.date >= Date() }` removes events that are already in the past.
- `sorted { $0.date < $1.date }` orders the remaining events from earliest to latest.

This approach uses standard Swift array operations (`filter`, `sorted`) and keeps the logic in one place (the `ViewModel`). Views like `HomeView` and `MyEventsView` simply read `upcomingEvents` and render the results.

6.3 Event Filtering by Tag or Category

To support more focused browsing, Mingle can optionally filter events by category (for example, "Party", "Study", "Club").

Filtering Function

```
func events(forTag tag: String?) -> [Event] {
    guard let tag = tag, !tag.isEmpty else { return upcomingEvents }
    return upcomingEvents.filter { $0.tags.contains(tag) }
}
```

Explanation

- If `tag` is `nil` or an empty string, the function returns all upcoming events.
- Otherwise, it filters `upcomingEvents` to only those where the `tags` array contains the selected tag.
- This is implemented as a simple linear search over the event list, which is efficient enough for the small number of events expected in this app.

The view can use `events(forTag:)` whenever the user selects a filter option, updating the displayed list accordingly.

6.4 RSVP Logic & Capacity Check

RSVP handling is one of the most important algorithms in Mingle. It must:

- Update the user's RSVP status.
- Enforce capacity constraints.
- Keep the counts of Going/Maybe attendees correct.

High-Level Steps When a User RSVPs

1. Check if the selected status is `.going` and whether the event has a `maxCapacity`.
2. Count the number of `Going` RSVPs for that event.
3. If the event is already full, reject the new `Going` RSVP and inform the user.
4. Otherwise, create or update the `EventRSVP` record for the current user and event.
5. Reflect the updated status and counts in the UI.

Pseudo-code Implementation

```
func rsvp(to event: Event, as status: RSVPStatus, for user: User) -> Bool {
    // 1. Capacity check for "Going"
    if status == .going, let capacity = event.maxCapacity {
        let goingCount = rsvps.filter {
            $0.eventId == event.id && $0.status == .going
        }.count
        if goingCount >= capacity {
            return false // event is full
        }
    }

    // 2. Find existing RSVP for this user and event
    if let index = rsvps.firstIndex(where: {
        $0.eventId == event.id && $0.userId == user.id
    }) {
        // 3. Update existing record
        rsvps[index].status = status
        rsvps[index].updatedAt = Date()
    } else {
        // 4. Create a new RSVP record
        let newRSVP = EventRSVP(
            id: UUID().uuidString,
```

```
                eventId: event.id,
                userId: user.id,
                status: status,
                updatedAt: Date()
            )
            rsvps.append(newRSVP)
        }

        return true
}
```

Data Structures & Algorithms Used

- **Filtering** (`filter`) – to count `Going` RSVPs for a specific event.
- **Searching** (`firstIndex(where:)`) – to find the current user's existing RSVP for that event.
- **Updating collections** – by modifying or appending `EventRSVP` elements in an array.

This logic ensures that Mingle behaves correctly even when users change their RSVP status multiple times.

7 User Interface & Navigation (SwiftUI)

The user interface of Mingle is built entirely with SwiftUI. This section describes the main screens, how navigation is structured, and which SwiftUI constructs are used to implement layouts and interactions.

7.1 Main Screens

Mingle consists of several primary screens, each responsible for a specific part of the user flow.

1. **HomeView (All Events)**
    - Displays a scrollable list of upcoming events available to the user.
    - Uses a `List` or `ScrollView` combined with `LazyVStack` to show multiple events efficiently.
    - Each row or card uses a reusable `EventCardView`, which typically includes:
        - Event title
        - Date and time
        - Location name
        - Optional host name or tag badges
    - Tapping a card navigates to `EventDetailView`.

MyEventsView

- Shows only the events where `hostId == currentUser.id`.
- Allows the user to quickly review everything they are hosting.
- Includes a prominent "+ Create Event" button that:
    - Opens `CreateEventView` via a `NavigationLink` or `.sheet`.

EventDetailView

- Shows full information about a single event:
    - Title, description, date, time, location, and tags.
    - Capacity and attendee counts, if applicable.
- Provides RSVP buttons for *Going* / *Maybe* / *Not Going*.
- Shows the current user's RSVP status clearly (for example, a highlighted button or label).
- Optionally displays counts of Going/Maybe attendees.

CreateEventView

- A form-based screen where hosts enter:
    - Title (`TextField`)

  - Description (`TextEditor`)
  - Date & time (`DatePicker`)
  - Location (`TextField`)
  - Optional capacity (`TextField` or `Stepper`)
  - Optional tags (`TextField` or tag selector)
- Includes "Save" and "Cancel" buttons:
  - "Save" calls `EventViewModel.createEvent(...)`.
  - On success, the view dismisses and returns to *My Events*.

ProfileView

- Displays the current user's information:
  - Name
  - Optional email or avatar
- May optionally allow simple editing of profile fields.

These screens together cover the main flows of discovering, creating, and managing events in the app.

7.2 Navigation Structure

Navigation in Mingle is organized around a root `TabView`, with navigation stacks inside each tab.

Root Structure

- A `TabView` with three main tabs:
  1. **Home** – All Events
  2. **My Events** – hosted events and event creation
  3. **Profile** – user profile information
- Each tab typically wraps its content in a `NavigationStack`, which allows push-based navigation to detail screens.

Example Flows

- **Home tab**
  - `HomeView` (list of events)
  - Tap event → `EventDetailView`
- **My Events tab**
  - `MyEventsView` (list of hosted events + "+" button)
  - Tap "+" → `CreateEventView`

- Tap existing event → `EventDetailView`

This structure is simple, consistent, and familiar to iOS users, while keeping navigation logic easy to maintain.

7.3 SwiftUI Constructs Used

Mingle makes use of a range of SwiftUI components and patterns to implement its user interface.

Layout

- `VStack` – vertical stacking of text, buttons, and sections.
- `HStack` – horizontal layout for badges, labels, and button groups.
- `ZStack` – layering background elements behind content when needed.
- `ScrollView` – for screens with content that might not fit on one page.
- `List` – for dynamic lists of events.

UI Components

- `Text` – for titles, descriptions, labels, and counts.
- `Image` – for icons or potential profile/event images.
- `Button` – for actions such as Create, Save, and RSVP.
- `TextField` – single-line input (title, location, tags).
- `TextEditor` – multi-line input (description).
- `DatePicker` – selecting date and time.
- `NavigationStack` / `NavigationView` – for hierarchical navigation.
- `NavigationLink` – for tapping rows to navigate to detail screens.
- `TabView` – for root-level tabbed navigation.

State Management

- `@State` – for local, view-specific state (text fields, toggles, selected tag).
- `@StateObject` – for creating and owning a `ViewModel` instance in a root view.
- `@ObservedObject` – for reading a `ViewModel` when it is created elsewhere.
- `@EnvironmentObject` – for sharing a `ViewModel` across many views via the environment.

Modifiers

- `.sheet` – to present modal screens (e.g., `CreateEventView`) when needed.
- `.alert` – to show validation errors or "Event is full" messages.
- `.padding()`, `.font()`, `.foregroundColor()` – for basic spacing and visual styling.

Together, these constructs allow Mingle to present a declarative, reactive UI that responds instantly to changes in the underlying data.

8 Implementation Details

8.1 Programming Constructs

Mingle uses standard Swift constructs (variables, constants, collections, control flow, helper methods, closures) in combination with SwiftUI's declarative patterns. Models are structs; view models are classes conforming to `ObservableObject`.

8.2 Error Handling and Validation

- Input validation prevents events from being created with missing required fields.
- Logical checks enforce capacity rules and avoid inconsistent RSVP states.
- When persistence is used, `do--try--catch` guards encoding and decoding to prevent crashes due to corrupted data.

8.3 Persistence (If Included)

Two modes are supported:

- **In-memory only**: all data is lost when the app closes, sufficient for demo purposes.
- **UserDefaults/JSON**: `[Event]` and `[EventRSVP]` are encoded to JSON on updates and decoded on launch.

8.4 Screens and Diagrams (Placeholders)

The following figures currently act as placeholders for the system architecture and key UI screens of Mingle. They indicate where the final diagrams and screenshots will appear in the completed document. Before submission, each placeholder image file should be replaced with a high–resolution architecture diagram or an actual app screenshot, ensuring that all figure captions, labels, and references remain consistent.

8.4.1 System Architecture Diagram

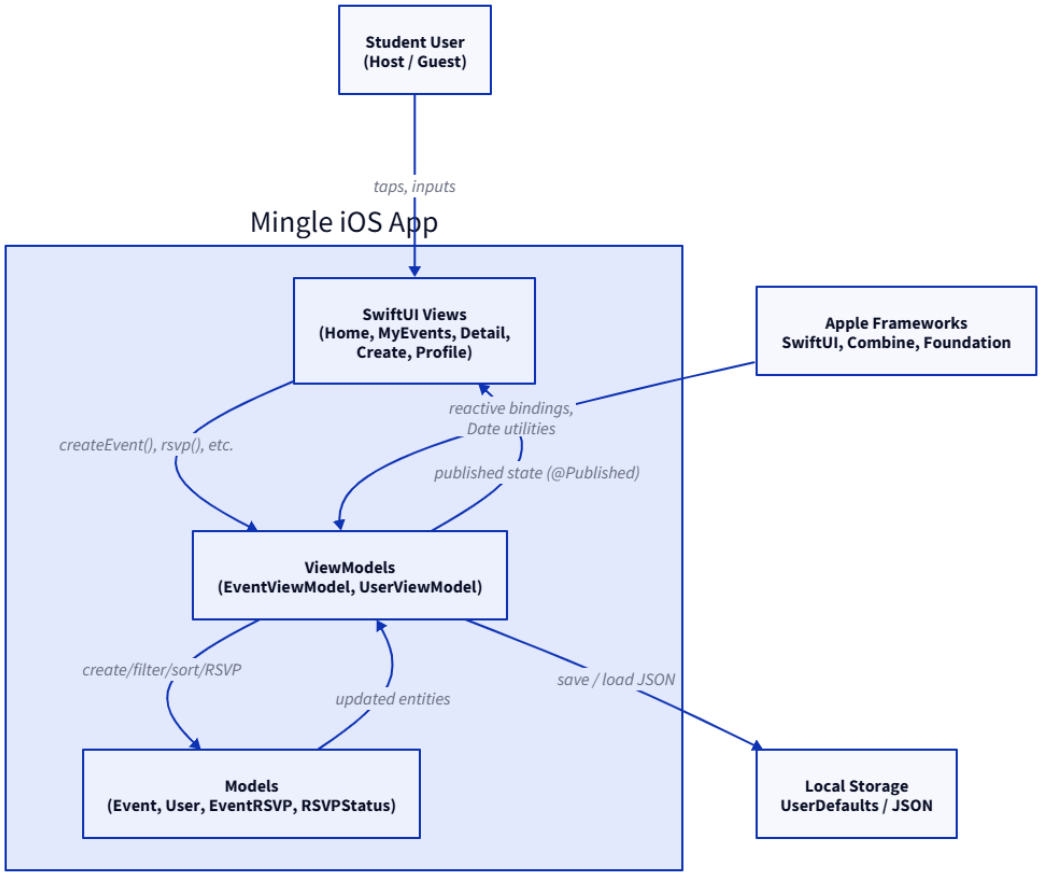


Figure 1: System Architecture Diagram for Mingle

8.4.2 User Flow Diagram

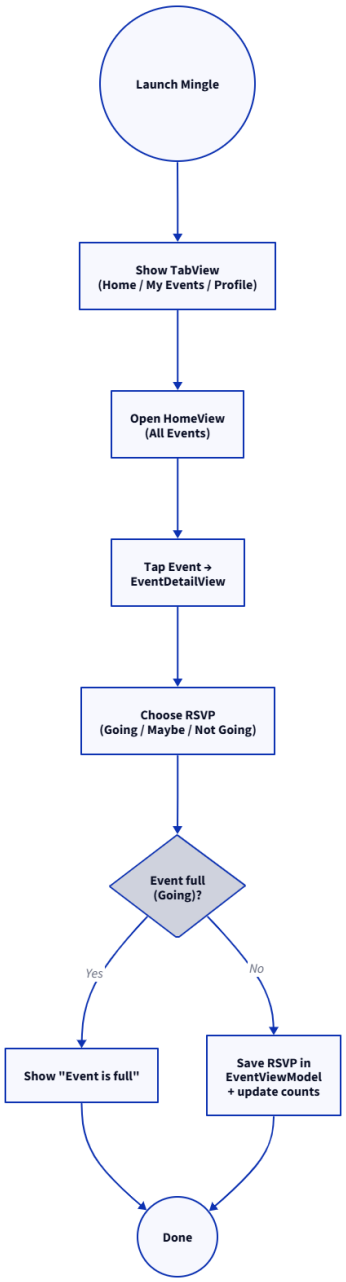
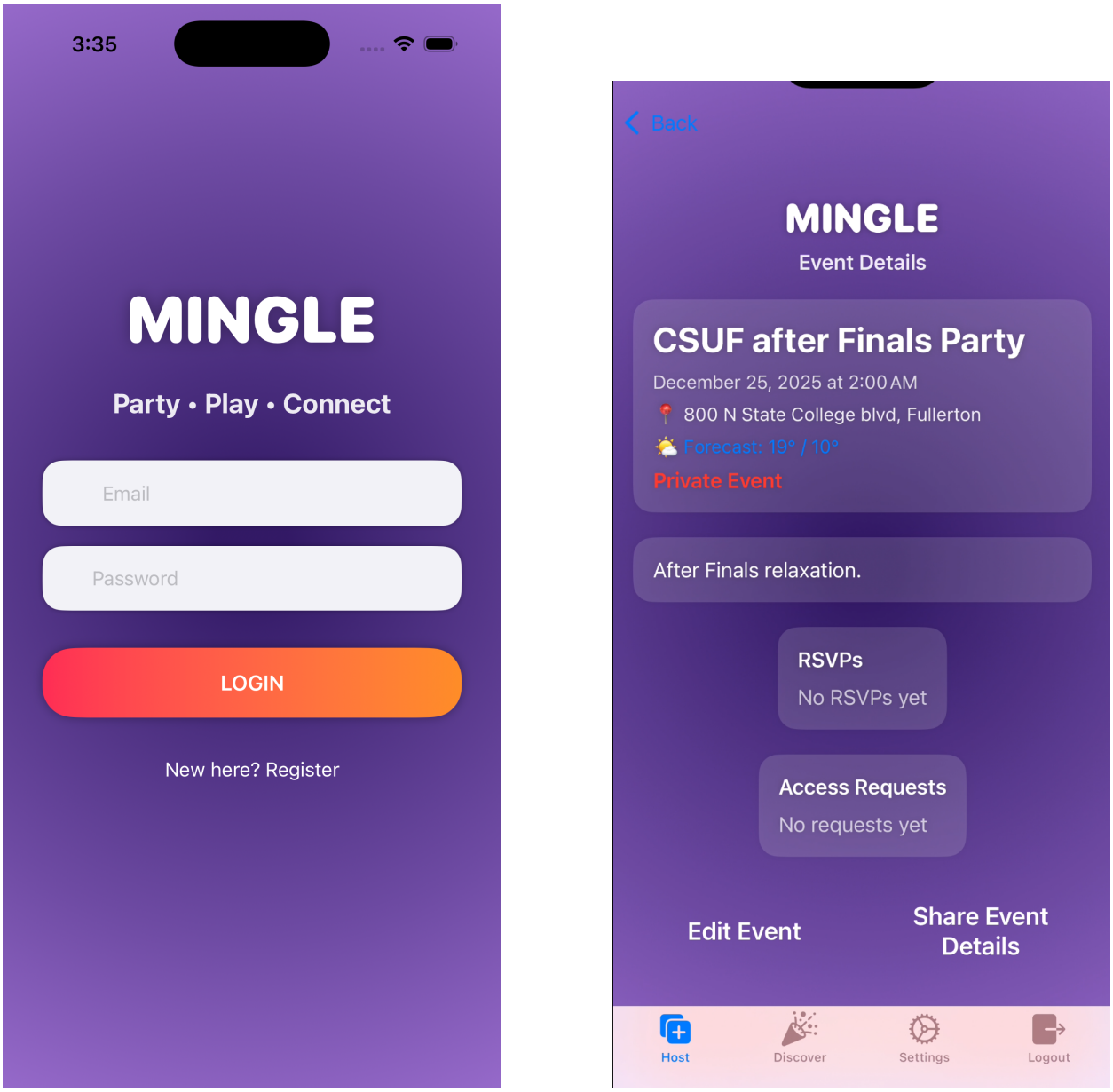


Figure 2: User Flow Diagram for Mingle

8.4.3 Results Screens – Home and Event Detail

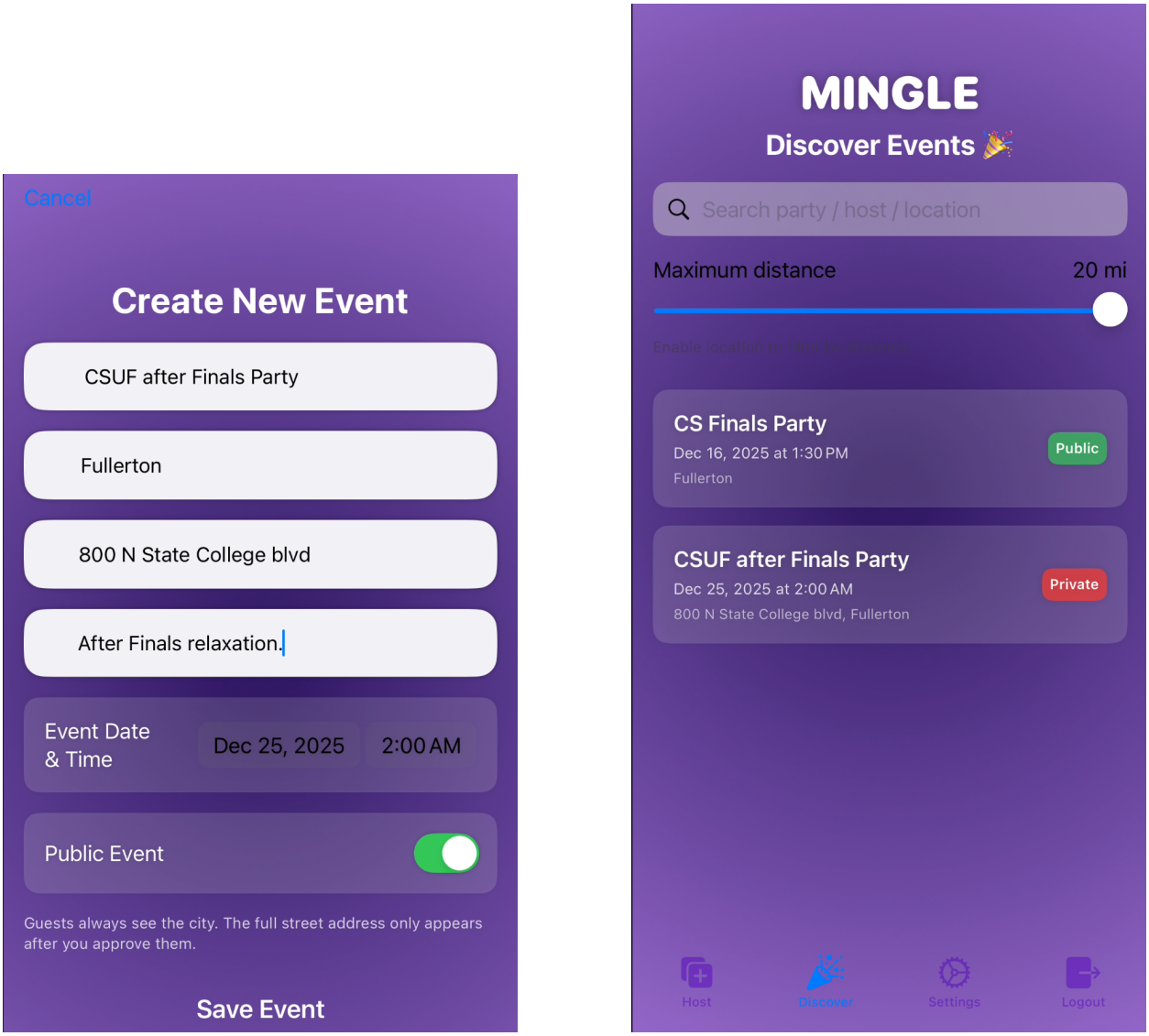


(a) Home Screen – All Events

(b) Event Detail Screen

Figure 3: Mingle home and event detail screens.

8.4.4 Results Screens – Create Event and Discover

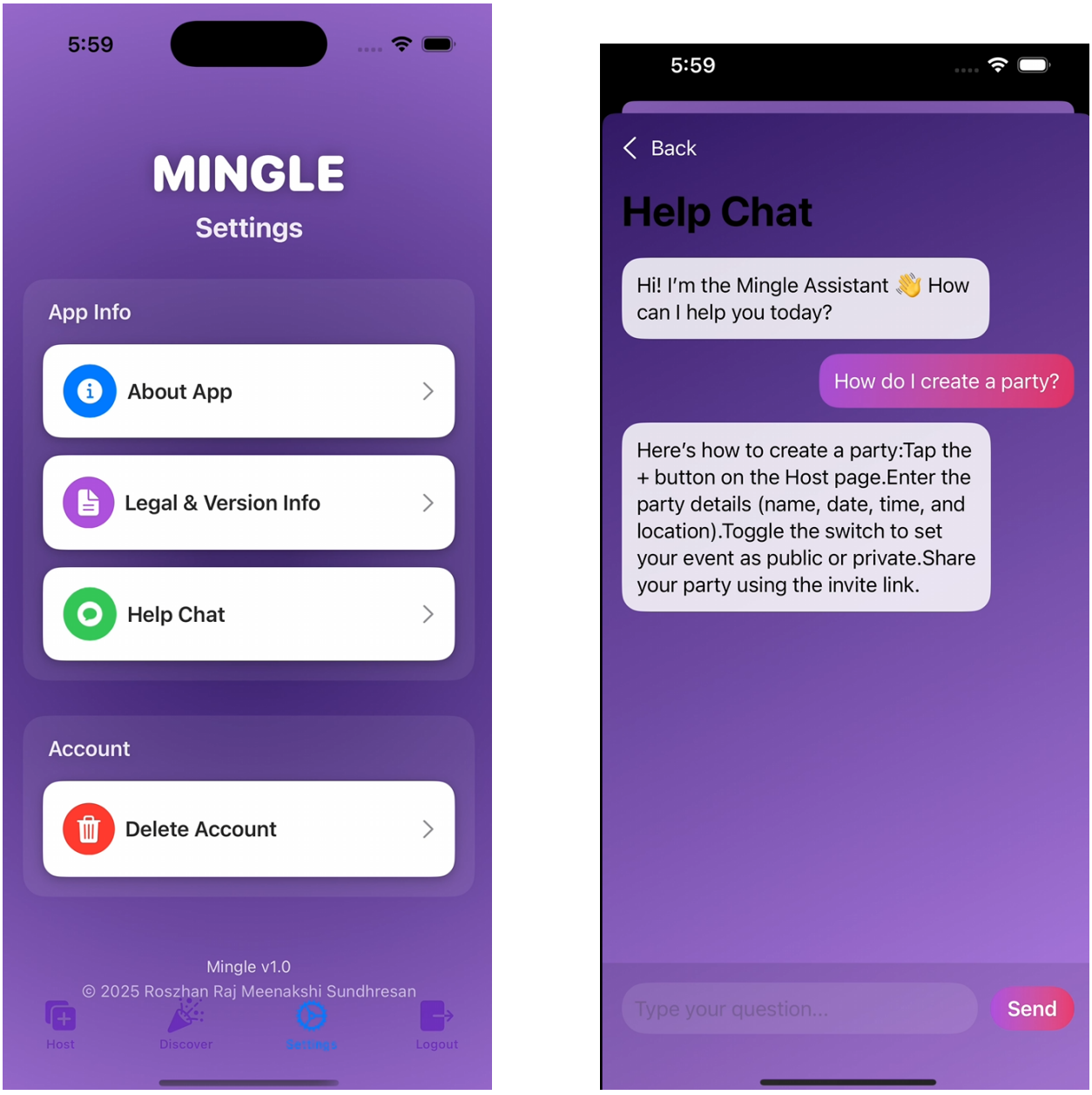


(a) Create Event Screen

(b) Discover Screen

Figure 4: Create event and discover events screens.

8.4.5 Results Screens – Profile and AI Chat Support



(a) Profile Screen

(b) AI Chat Support Screen

Figure 5: Profile screen and AI-powered chat support screen.

9 Testing & Evaluation

Testing for Mingle focuses on manual functional testing and verifying that the app behaves correctly in both typical and edge–case scenarios.

9.1 Manual Testing Scenarios

The team manually exercised the following flows:

1. **Event Creation**
    - Create an event with all required fields filled correctly and confirm that it appears in both *My Events* and *All Events*.
    - Attempt to create an event with a missing title or location and verify that the app prevents creation and shows appropriate feedback.
2. **Event Listing and Sorting**
    - Create multiple events with different dates and times.
    - Confirm that upcoming events appear sorted from earliest to latest.
    - Check that past events are either hidden from the upcoming list or clearly marked as past.
3. **Navigation**
    - From Home: open an event, view `EventDetailView`, then navigate back and confirm smooth navigation without crashes.
    - From My Events: tap "+ Create Event", create an event, and ensure the new event is visible immediately when returning.
4. **RSVP Functionality**
    - For an event without capacity limits, change the RSVP status between Going, Maybe, and Not Going and verify that the UI always reflects the current status.
    - For an event with `maxCapacity`, simulate multiple users (or test with different user IDs in code) to fill the event with Going RSVPs, then attempt to add an additional Going RSVP and confirm that it is rejected.
5. **My Events Filtering**
    - Create events as the current user and (optionally) simulate other hosts.
    - Verify that *My Events* shows only events where `hostId == currentUser.id`.
6. **AI Help Assistant**

- Open the AI help feature from the Settings view and confirm that the chat interface loads correctly.
- Enter common user questions such as how to create an event or how RSVPs work and verify that the assistant returns relevant, concise guidance.
- Test empty input, short greetings, and follow-up questions to confirm the assistant responds appropriately without crashing or duplicating introductory messages.
- Verify that AI responses display correctly in the chat UI, including line breaks and emphasized text when applicable.

These tests ensure that each core feature behaves as expected from a user's perspective.

9.2 Edge Cases and Observations

In addition to the main flows, the team considered several edge cases:

1. **Past Events**
   - Events whose date is strictly less than `Date()` should either be excluded from the upcoming list or visually distinguished as past events.
2. **No Capacity Specified**
   - When `maxCapacity` is `nil`, the app treats the event as having unlimited capacity and the RSVP logic skips capacity checks.
3. **Changing RSVP Status**
   - Switching from Going to Maybe or Not Going must decrement the effective Going count so that no "stale" attendee counts remain.
4. **Empty State**
   - When there are no events at all, `HomeView` and `MyEventsView` should display a friendly "No events yet" message instead of a blank screen.

9.3 Limitations

The current version of Mingle relies on local, manual data entry and does not connect to a real backend service. As a result, some scenarios—such as time zone differences, concurrent edits from multiple devices, and server–side validation—are out of scope for this release and are left as future work.

10 Team Roles & Contributions

Team Members

- Roszhan Raj Meenakshi Sundhresan (roszhan.2684@csu.fullerton.edu)
- Tanmay Chaudhari (tanmay_2106@csu.fullerton.edu)
- Vanessa Iwaki-Honore (vanessa_ih@csu.fullerton.edu)
- Sunny Palaco (sunnypalaco@csu.fullerton.edu)

Individual Contributions

Roszhan Raj Meenakshi Sundhresan

- Led SwiftUI implementation of core flows (`HomeView`, `MyEventsView`, `EventDetailView`, `CreateEventView`).
- Integrated UI with `EventViewModel` and ensured smooth navigation.
- Connected RSVP interactions to the underlying logic.
- Coordinated overall project structure and contributed substantially to this design document.

Tanmay Chaudhari

- Designed and implemented data models (`Event`, `User`, `EventRSVP`, `RSVPStatus`).
- Implemented business logic in `EventViewModel`, including event creation, filtering, sorting, and capacity checks.
- Assisted with debugging and verifying integration between ViewModel methods and SwiftUI views.

Vanessa Iwaki-Honore

- Focused on UI/UX design, including colors, typography, layout spacing, and visual polish.
- Created and refined reusable components such as `EventCardView`.
- Conducted usability testing and improved error messages and empty states.
- Reviewed and refined sections of the design document and prepared visuals for the demo.

Sunny Palaco

- Implemented and refined the in-app help chat feature, including UI behavior and response formatting.
- Added the in-app AI help entry point through the Settings view and wired it to the chatbot UI using AIService and the Help Chat view.

- Owned navigation structure and shared state setup using `TabView` and `NavigationStack`.
- Integrated screens and components into a cohesive user flow.
- Managed GitHub workflows (branching, merging, keeping `main` stable).
- Assisted with testing RSVP edge cases and My Events filtering.

All team members participated in regular check-ins, manual testing, bug fixing, and demo preparation.

11 GitHub Repository & Submission

The complete Mingle source code is pushed to the `main` branch of the team's GitHub repository.

- All Swift and SwiftUI project files are included.
- A `README.md` file provides setup and run instructions.
- This design document and a link to the demo video may also be included.

Repository URL: https://github.com/VanessaIH/party_planner.git

12 Conclusion & Future Work

Mingle demonstrates how a focused iOS application built with Swift and SwiftUI can address a real, student-centered problem: organizing and attending social events in a structured, reliable way. By concentrating on the everyday pain points of college students—lost event details, unclear headcounts, and scattered communication—the project transforms an informal, messy workflow into a clean, well-defined mobile experience.

From a functional perspective, Mingle successfully delivers the core features of a lightweight event management platform tailored for students:

- Hosts can quickly create events with clear titles, descriptions, dates, times, locations, tags, and optional capacity limits.
- Guests can effortlessly browse upcoming events, open detailed views, and respond using intuitive Going/Maybe/Not Going controls.
- The separation between *All Events* and *My Events* provides both a global overview and a personal dashboard for the current user.
- Basic capacity enforcement ensures that events with limited space remain manageable.

Looking forward, Mingle provides a solid foundation for future development. Potential enhancements include:

- **Authentication and backend integration** to support real multi-user scenarios, cross-device synchronization, and durable storage of events and RSVPs.
- **Event sharing and invitations** via deep links, QR codes, or integration with campus systems so that events can be easily distributed.
- **Notifications and reminders** (local and push) to reduce no-shows and keep attendees informed about schedule or location changes.
- **Map integration** to visualize event locations and provide directions directly from the app.
- **In-event communication**, such as chat or comment threads, to coordinate carpools, potlucks, or last-minute updates.
- **Advanced discovery and personalization**, including richer filters, recommendation algorithms, and personalized feeds based on attendance history and interests.

Overall, Mingle meets the objectives of the CPSC 411 project by solving a real problem with a thoughtful design, a clear architecture, and a working implementation. At the same time, it lays down a strong architectural and conceptual base that could grow into a more complete event platform for students in future iterations.