CS673 Software Engineering

Team 3 - PlanningJam 

Software Design Document

Team Member	Role(s)	Signature	Date
Ashley	Requirement Leader	*AshleySachdeva*	9/22
David	Team Leader	*DavidMetraux*	9/22
Donjay	Configuration	*Donjay Barit*	9/22
Jason	QA	*JasonLee*	9/22
Haolin	Design and Implementation Leader	*Haolin Yang*	9/22

Revision history

Version	Author	Date	Change
0.0.0	ALL	9/22	CREATED
1.0.1	Haolin Yang	10/09/2025	Removed Unimplemented features in the class diagram
2.0.0	Donjay	10/12/2025	Updated Database design
2.1.0	Haolin Yang	10/13/2025	Add User Profile to the Class Diagram

Introduction

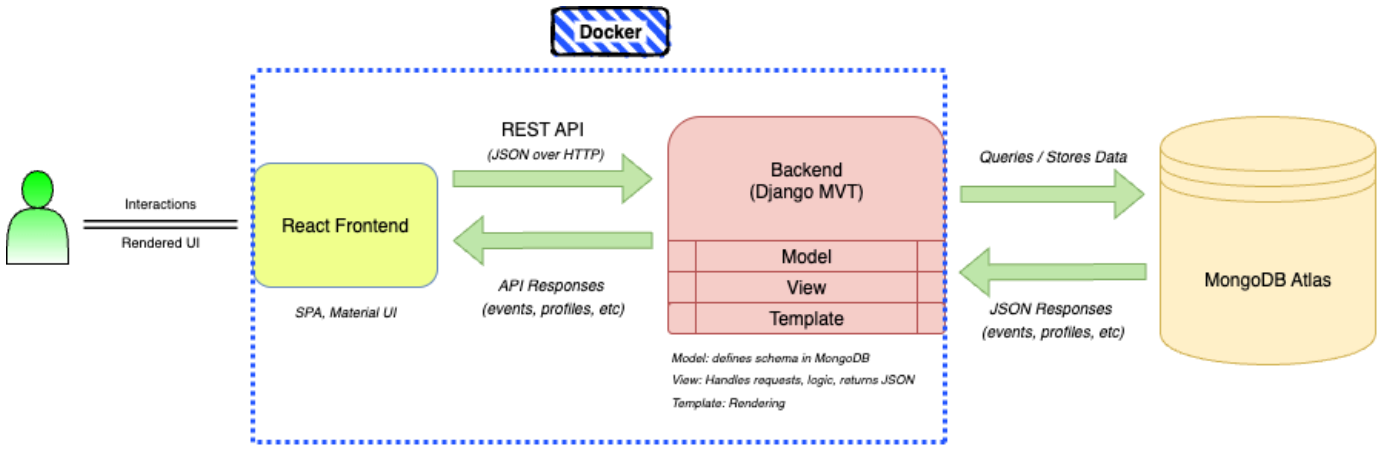
This document serves as a guide for developers, testers, and stakeholders by detailing the system architecture, class models, database schema, user interface design, security framework, and core business logic.

The system emphasizes usability, reliability, and scalability while fostering an intuitive user experience. To achieve these goals, we adopted a React frontend and a Django backend, connected through RESTful APIs, with MongoDB serving as the primary database. This combination provides flexibility, modularity, and efficient data handling.

The design goals of PlanningJam are:

- User-Centric Experience: Provide a clean and accessible interface for creating and managing plans, groups, and friendships.
- Scalability and Modularity: Use a component-based frontend and a decoupled backend to support development and future feature expansion.
- Reliability and Security: Ensure secure authentication, data integrity, and resilience against common threats.
- Reusability and Maintainability: Apply design patterns and architectural principles that promote clean code, separation of concerns, and easier long-term maintenance.

- Software Architecture



https://app.diagrams.net/#G1tMTP-RLervRX6Ddrq2YARJRp7QPlQDRD#%7B%22pageId%22%3A%22C46RRvvLz8JMO92MqmRO%22%7D

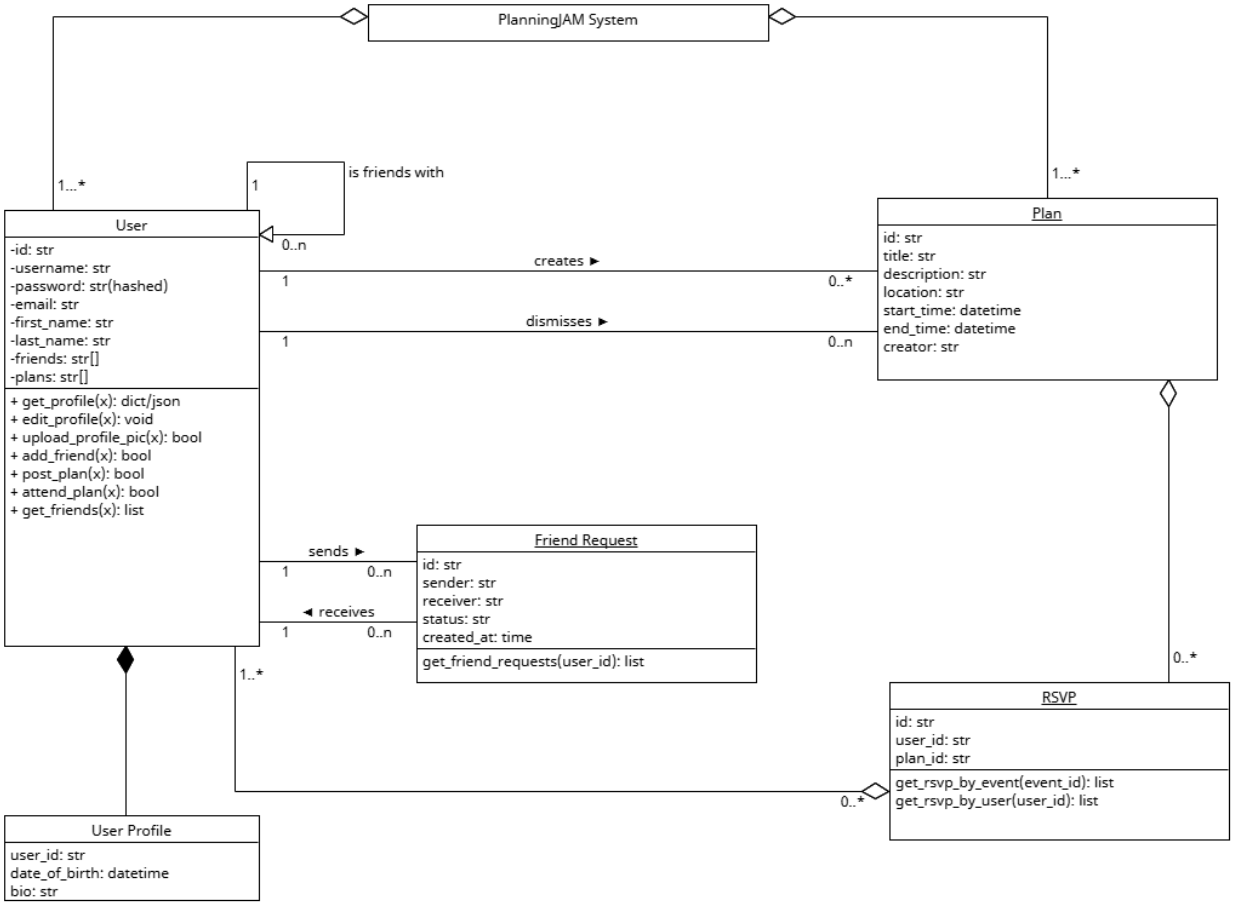
We are using React as the frontend and Django as a backend. The two components are connected using Rest API's. We use MongoDB to actually store the data.

Similar to most applications, we are using MVC. Except, since it is Django, it is Model-View-Template. The Model of Django sets the design of the database, and we're using MongoDB as our database. The user sends commands via Django's View (which is similar to a controller). This mostly uses Python to return HTTP responses. The Template is how we form what the user sees, and we are combining that with React.

React enables single-page applications (SPAs) with smooth, real-time updates. The structure of components allows us to make components and reuse them, and lets us use React's Material UI lets us have a quicker time developing UI elements.

Django serves as a reliable backend service, while React can be deployed as static files through CDNs for speed. The decoupled nature reduces server load and allows us to use Docker efficiently.

- Class Diagram



User:

Role: Primary actor. Owns content (plans, groups, profile), forms friendships, and participates in plans.
Key fields: id, username, email, password (hashed), first_name, last_name, friends[], plans[], groups[], profile_pic.
Core behaviors:

Profile: get_profile(), edit_profile(), upload_profile_pic().

Social graph: add_friend(target_user_id) creates a Friend Request (or auto-accepts if mutual pending); get_friends().

Plans: post_plan(plan_data) creates a Plan with creator = user.id; attend_plan(plan_id) routes through RSVP to mark intent.

Validations & invariants:

username unique, email unique; password stored as hash.

Cannot befriend self; no duplicate friendships; friendship is symmetric once accepted.

Deleting a user should cascade/soft-delete owned artifacts or reassign ownership per policy.
Relationships:

1–* creates Plan; – attends Plan via RSVP.

0..n ↔ 0..n friends (symmetric).

1–* sends/receives Friend Request; can dismiss requests.

1-1 owns profile

User Profile:

Role: A supplement class to the User class

Key fields: id, date_of_birth, bio

Core behaviors:

Queries (getting a user's profile or another user's profile)

Update (changing bio or date of birth, additional fields are allowed)

Relationships:

1–1 owned by User

Plan:

Role: The event/hangout unit users coordinate around.
Key fields: id, title, description, location, start_time, end_time, creator.
Core behaviors:

Lifecycle: create (by creator), edit (by owner/admins), cancel/archive, visibility controls (public/group/private).

Participation: read RSVPs, attendee list, capacity checks/waitlist.

Messaging/updates: emits Notification on key changes (time, location, host updates).

Calendar: syncs 1:1 with Calendar entry.

Tagging/filtering: associates with one or more Tags for search & discovery.
Validations & invariants:

start_time < end_time; title non-empty; creator must be a valid User.

Editing rules respect ownership/role (creator or group admins).
Relationships:

– RSVP (each RSVP ties one User to this Plan).

*–1..n Tag (plan belongs to one or more tags).

RSVP:

Role: Join table capturing a user's response to a plan.
Key fields: id, user_id, plan_id, (typically also status = {going, maybe, not_going}, timestamp, optional comment).
Core behaviors:

Queries: get_rsvp_by_event(event_id), get_rsvp_by_user(user_id).

Upserts: change status without creating duplicates.

(user_id, plan_id) unique; user/plan must exist; status is controlled vocabulary.
Relationships:

Many-to-one to User and Plan (implements the many-to-many attendance between them).

Friend Request:

Role: Workflow object to establish symmetric friendship between two users.
Key fields: id, sender, receiver, status, created_at.
Core behaviors:

Creation: sender → receiver creates pending.

Acceptance/Decline: on accept, add each user to the other's User.friends; on decline/dismiss, close request.

Listing: get_friend_requests(user_id) returns inbound & outbound pending requests.
Validations & invariants:

No self-requests; no duplicate pending requests between the same pair; friendship must not already exist.
Relationships:

1–* from User (sends), 1–* to User (receives).

PlanningJAM System (System Boundary):

Role: Context boundary that contains all components and interfaces to external services.
Scope & concerns:

Auth & security: session/JWT, password hygiene, rate limiting, audit logs.

Data & consistency: transactions for friendship acceptance and RSVP upserts; idempotent operations.

Search & discovery: tag indexing; text search on plan titles/descriptions; time & location filters.

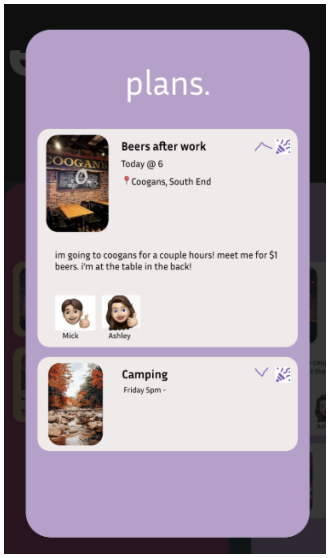
Privacy & access control: plan visibility (public, friends, group, private); owner/admin permissions.

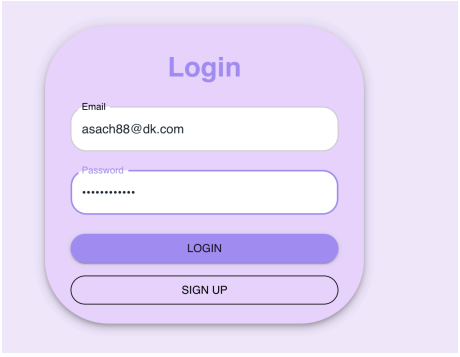
Reliability: background jobs (reminders), retries, dead-letter queues; clock skew tolerance for reminders.

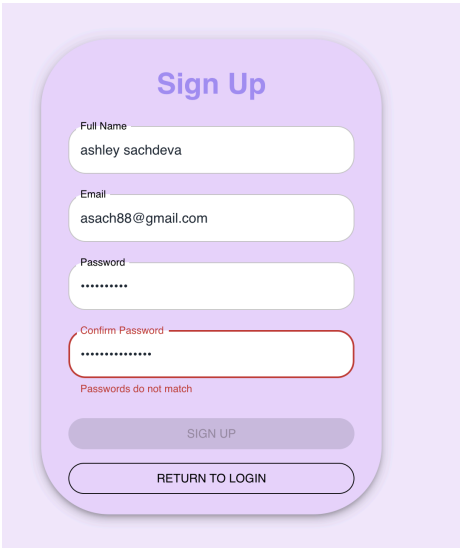
- ## UI Design

The application follows a clean and consistent design language with a pastel color palette, rounded components, and simple typography for readability. The UI emphasizes ease of use, with forms that provide clear validation and error handling to guide users through interactions. Navigation is straightforward and consistent across the application, ensuring that users can move between features without confusion. Interactive elements such as buttons, input fields, and cards are designed to be intuitive and responsive,

creating a smooth user experience. The overall design balances simplicity with functionality to make the system both approachable and effective.







- ## Database Design

The application uses MongoDB as its database, which is a NOSQL database that employs a document-oriented data model allowing for flexible and dynamic data structures. MongoDB is used for its flexible schema and the ability to scale horizontally as the data for Planning Jam evolves and changes over time.

The database contains the following collections for the application:

User collection:
The user collection stores user profiles and authentication information. Key fields include

username, first and last name, date of birth, and bio as well as authentication related data. Supported operations for the collection include retrieving a user by their unique ID, retrieving a list of users, and updating a user's public profile information.

Plans collection:
The plans collection stores plan event information and includes the following key fields: the plan title, description, location, start and end times. Supported operations for the collection include creating a plan, updating or deleting a specific plan by its unique ID, retrieving a list of plans based on specific criteria and retrieving a plan by its unique ID.

RSVP collection:
The rsvp collection stores the relationship between a user and their commitment to attend a plan. Supported operations for the collection includes retrieving a list of reservations made by the logged in user, retrieving a reservation information based on an ID, or deleting a reservation with a given ID.

Dismissal collection:
The dismissals collection stores a relationship between a user and the plans they decide to opt out from their feed or notifications. Supported operations for the collection includes retrieving a list of plans a user has dismissed from their feed, adding an entry for a dismissed plan, and removing dismissed plans.

Friends collection:
The friends collection stores the relationship and statuses between users. Each record shows the connection between users and whether their friendship status is either pending or accepted based on request from either user. Supported operations for the collection includes retrieving a list of friends for the current user, removing a friend, and creating and updating the documents for creating a friend relationship.

● Security Design

A comprehensive security strategy is deployed to provide a multi-layered defensive security across all stages of the software development lifecycle – through code, application, deployment, infrastructure, DNS and database. Each layer is designed to protect and minimize a range of threats and attacks and proactively mitigate such risks.

Code Security
Developers are expected to keep security in mind when writing or reviewing codes, ensuring that no sensitive data or private keys are hard coded or exposed publicly. As an added measure, Github's AI code scanning and Dependabot tools provide a way to inform developers of any exposed secret keys and vulnerabilities to dependent packages in the application.

Application Security
Communication between the frontend and backend applications are done through an API. Data are secured and encrypted through HTTPS/SSL protocols during transmission to protect data from being intercepted. JWT authentication is used to secure user sessions where tokens are set to expire in 1 hour to minimize the risks of stolen tokens from being used for any extended period.

Deployment Security
Github Action is utilized to perform a CI/CD to the staging and production environments. Throughout the CI/CD pipeline, secret keys and environment variables are securely managed, preventing them from being hard-coded or exposed during the deployment of docker images and logging into the servers via SSH to run the docker builds.

Infrastructure Security
The application's backend and frontend are both served through Digital Ocean's droplets, a Virtual Private Server (VPS) containing an Ubuntu linux server. Security principles are implemented to harden the server from any potential threats of attacks and mitigate such risks.

The following steps are taken to secure and harden the servers:

- Securing server access with SSH key-based authentication and disabling password authentication.
- Disallowing root user login to the servers to prevent direct administrative access.
- Setting up firewalls to only permit necessary ports and allowed IP addresses.
- Continuously perform security updates and patch upgrades to the operating system from vulnerabilities.
- Hardening the system with AppArmour.
- Track and audit access to the servers by monitoring and logging user activities.
- Regularly scheduling backups of data that can easily be restored in the event of a security incident or data loss.

DNS Security
The application's DNS is hosted by Cloudflare where domains are proxied, allowing the servers' IP addresses to be hidden. Cloudflare also provides protection against DDoS attacks, SSL certificates and strict SSL/TLS encryptions services.

Database Security
The database is hosted with MongoDB Atlas and inherits all security features from the service provider that includes TLS and encrypted data. Additionally, only whitelisted IP addresses will have direct access to the database, which is likely to only be from the backend server, reducing any attacks from unauthorized IPs.

- ## Business Logic and/or Key Algorithms - Take a look at this later

  In this section, you shall describe any key algorithms used in your software system, either in terms of pseudocode or flowchart, or sequence diagrams.

- ## Design Patterns

Frontend (React):

- **Builder Pattern**: Used in constructing UI components such as plan creation forms and profile editing screens. This allows us to incrementally add fields (ex. plan title, description, tags, date/time) and render them dynamically. It makes the UI more flexible and reduces the possibility of duplication across different components
- **Observer Pattern (via React state management)**: React's *useState* and *useEffect* hooks implement an observer-style pattern where UI components automatically re-render when state changes (like when new plans appear on the feed or a friend request is accepted)
- **Container/Presentational Pattern**: React components are split between container components that manage data fetching and business logic, and presentational components that focus purely on rendering UI. This separation makes it easier to test and maintain code.

Backend (Django/Python):

- **Factory Pattern**: Applied in creating different types of responses or objects (ex. building JSON responses for API endpoints, instantiating user or plan objects). This pattern helps encapsulate object creation logic and makes it easier to expand features without rewriting core logic
- **Model-View-Template (MVT) Pattern**: Django's built-in architecture follows MVT, separating database models (MongoDB schemas), view logic (request handling in Python), and templates. Even though React replaces many templates, the separation still ensures backend logic remains clean and independent from presentation.
- **Repository Pattern (with MongoDB)**: Used to abstract database interactions from the business logic. This helps us centralize all queries to MongoDB, making the system easier to maintain and migrate if schemas change

- ## Rest APIs

  Authentication uses JWT, when users log in they receive an access token (valid for 1 hour) and, optionally, a refresh token

- The access token is included in every request via the Authorization header (Bearer <token>)
- Once the token expires, the user must re-authenticate or use a refresh token to obtain a new one

Currently all API endpoints are served under the backend's root URL:

Authentication

- Register a new user: **POST/register/**
    - Registers a new user and issues JWT tokens (access + refresh)
- Retrieve authenticated user's profile: **GET/profile/**
    - Requires a valid JWT access token
- User logs in: **POST/login**
    - Authenticate user, return access + refresh tokens
- User logs out: **POST/logout**
    - Blacklist/expire the refresh token (optional though since access tokens expire hourly)

Users & Friends

- Retrieve another user's public profile: **GET/users/{id}**
- Update own profile: **PUT/users/{id}**
- Send a friend request: **POST/friends/{id}/request/**
- Accept a friend's request: **POST/friends/{id}/accept/**
- List all friends: **GET/friends/**

Plans

- Create a new plan: **POST/plans/**
- List all visible plans (filterable): **GET/plans/**
- Get details of a plan: **GET/plans/{id}/**
- Edit a plan: **PUT/plans/{id}**
- Delete a plan: **DELETE/plans/{id}**

RSVP

- RSVP "Yes" to a Plan: **POST/plans/{id}/rsvp/**
- List all users who RSVP'd to a plan: **GET/plans/{id}/rsvp/**

Notifications

- Retrieve user notifications to plans: **GET/notifications/**

- Any Additional Topics you would like to include.

- AI usage Log

  You are allowed and even encouraged to use AI tools to help you generate the project idea, plan it and build it, but you need to clearly describe 1) What tools were used? 2) for what specific tasks and 3) Is it helpful? 4) how did you evaluate or modify AI-generated content? Additionally, you should submit the exported AI chat history as an appendix or share that with the instructor and facilitators.

Tools	Who	Tasks	helpful	Evaluation/modification	links
ChatGPT	David	Software Architecture	Relatively.	I didn't just list out, used sections to justify software architecture	https://chatgpt.com/share/68cee637-f510-8012-a7ef-d9e7acd2e2a5
ChatGPT	Donjay	Security	Yes	Evaluate recommended steps to take to secure a linux server that may be helpful for the project	https://chatgpt.com/share/68cf0dd0-d320-8000-a5a5-abfa2dbaa07e
ChatGPT	Jason	Design Patterns /Rest APIs	Yes	Used as a resource to fill out sections as a template	https://chatgpt.com/share/68d07fff-7e28-8012-a200-c964b3a09f7e
ChatGPT	Haolin	Derive description from existing Class Diagram	Yes	Used as a template to improve the existing class diagram. Also changed some of the generation to better fulfill our design goal.	https://chatgpt.com/share/68d09429-d3f4-8008-ba42-65cad24ead20

- References

  https://www.geeksforgeeks.org/python/django-project-mvt-structure/
  https://docs.djangoproject.com/en/5.2/faq/general/

- Glossary