CS673 Software Engineering

Team 4: Kickaas

Software Design Document

| **Team Member:** | **Role(s):** | **Signature:** | **Date:** |
| --- | --- | --- | --- |
| Javier | Team Leader | *Javi* | 09/25/2025 |
| Vamsi | Requirement Leader | *Vamsi* | 10/13/2025 |
| Johannes | Design and Implementation Leader | *Johannes* | 10/13/2025 |
| Saniya | Configuration Leader | *Saniya* | 10/14/2025 |
| Yuanfei (Keni) | QA Leader | *Keni* | 10/14/2025 |
| Shreya | Security Leader | *Shreya* | 10/13/2025 |

**Revision History**

| **Version:** | **Author:** | **Date:** | **Change:** |
| --- | --- | --- | --- |
| 0 | Team | 09/25/2025 | Initial Draft |
| 1 | Team | 10/28/2025 | Feedback From Iteration 1 |
| 2 | Team | 11/15/2025 | Update to Match New Features |

[Introduction](#_87t9hln2vjz0)

[Design Goals](#_pdq8k15ismua)

[Software Architecture](#_3ipvmjgn6clp)

[Frontend Architecture](#_x1mdg3yiguz)

[Architecture Overview](#_4yj9wukanug)

[Router (App Shell) Layer](#_mt8lz4svizyq)

[UI Components Layer](#_y5ogsmixnccx)

[State and Data Layer](#_7r3yf3pk1yo1)

[Services (API/SDK) Layer](#_9l43mjsnriq)

[Utilities Layer](#_91zx2p9odfzc)

[Request Flow](#_sycuxeifevb4)

[Backend Architecture](#_hkpol6315von)

[Architecture Overview](#_kuhmxjcfridb)

[Model Layer](#_nemjcbpja1wa)

[Router Layer](#_zanpvhvu7l6d)

[Service Layer](#_yp4sothdv4m3)

[Database Layer](#_wkehuuc9iu0o)

[Request Flow](#_tc7ofv1z6wth)

[Overall Folder Structure](#_3672ajjwx14g)

[Class Diagram](#_ky60nv8suxxm)

[Domain Models](#_nc3sy9vofl1d)

[Backend Layers](#_pff7x5m06bnq)

[Dependencies](#_e1qmweoae1wd)

[UI Design](#_7ucksmkf6rzx)

[Landing Page](#_qxfkccwq8v0l)

[Create an Event](#_45rjflfqr1ah)

[Sign Up/Login](#_ys9cl9yfa2mz)

[Database Design](#_tcmuor4nl1kz)

[Entity Relationship Diagram (ERD)](#_evag2a64l7v5)

[Tables and Relationships](#_urlq8qcvhnn6)

[Users](#_ex0cmsy3skje)

[Events](#_rskxbvc8ljvt)

[EventAttendees](#_7iba0mvyidb9)

[Categories](#_4ja33db2lny)

[Referential Integrity and Constraints](#_v8uxmtkoxq4s)

[Design Considerations](#_kdhjqj7kn62f)

[Security Design](#_x18fj36s1121)

[Authentication & Authorization](#_tmjm0jcxknrr)

[Data Protection](#_w6jhvkgd3uyg)

[API & Input Security](#_sgmk8xd08e7f)

[Frontend Security](#_x59eghwl26x3)

[Payments](#_g4p9l293iqja)

[CI/CD Security (GitHub Actions)](#_bi08bhz6wb9)

[Rest APIs](#_9a2dvlqtk0j2)

[User Endpoints](#_huljc3jvlhow)

[POST /users - Create User](#_ujszmgkuwsk3)

[GET /users - Get Users](#_a04v5y52wsgo)

[PATCH /users - Update Users](#_sqrsmj8psd3c)

[DELETE /users/{user\_id} - Delete User](#_c2h86jv8z6wo)

[Event Endpoints](#_20jgmslda7uh)

[POST /events - Create Event](#_blmmdjk3izrz)

[GET /events - Get Events](#_vtx4z5pbwf4z)

[PATCH /events - Update Events](#_nszw224ns0n0)

[DELETE /events/{event\_id} - Delete Event](#_65ca58igwpcc)

[Attendee Endpoint](#_77ibq5v48gkb)

[POST /attendees - Create Attendee](#_tknh1i3m2jya)

[GET /attendees - Get Attendees](#_y6r1nhci0g9v)

[PATCH /attendees - Update Attendee](#_atpnwvec1cdl)

[DELETE /attendees/{attendee\_id} - Delete Attendee](#_y0qz8cneit01)

[AI Usage Log](#_161vmjht87fy)

[References](#_iy9hzopoo8e1)

[Glossary](#_w1tz6tj2bb0m)

# **Introduction**

This Software Design Document (SDD) provides a comprehensive technical blueprint for the Event Manager and Planner application, a full-stack web solution designed to streamline event creation, management, and attendance tracking for individuals and organizations. This document serves as the authoritative guide for the development team, outlining the system architecture, component interactions, data models, and implementation strategies that will guide the construction of a robust and scalable event management platform.

The Event Manager application addresses the growing need for efficient event coordination tools by providing a unified platform where users can seamlessly transition between organizing their own events and discovering and attending events created by others. Unlike traditional event management systems that separate organizer and attendee functionalities, this application employs a unified role model that empowers every authenticated user to both create and participate in events from a single account.

## Design Goals

The primary design goals of the Event Manager system are structured around delivering a high-quality, maintainable, and user-centric solution:

**Scalability and Performance:** The system is architected to support concurrent usage by hundreds of users while maintaining responsive performance. The backend employs a layered architecture with FastAPI and PostgreSQL to ensure efficient data processing, while the frontend leverages Next.js with TypeScript for optimal rendering and user experience.

**Security and Reliability:** Security is paramount in the design, with Clerk authentication services providing robust user management, HTTPS encryption for all data transmission, and comprehensive input validation at multiple layers. The system implements transaction-based database operations with rollback capabilities to ensure data integrity and consistency.

**Maintainability and Extensibility:** The codebase follows modern software engineering practices with a clear separation of concerns across frontend and backend layers.

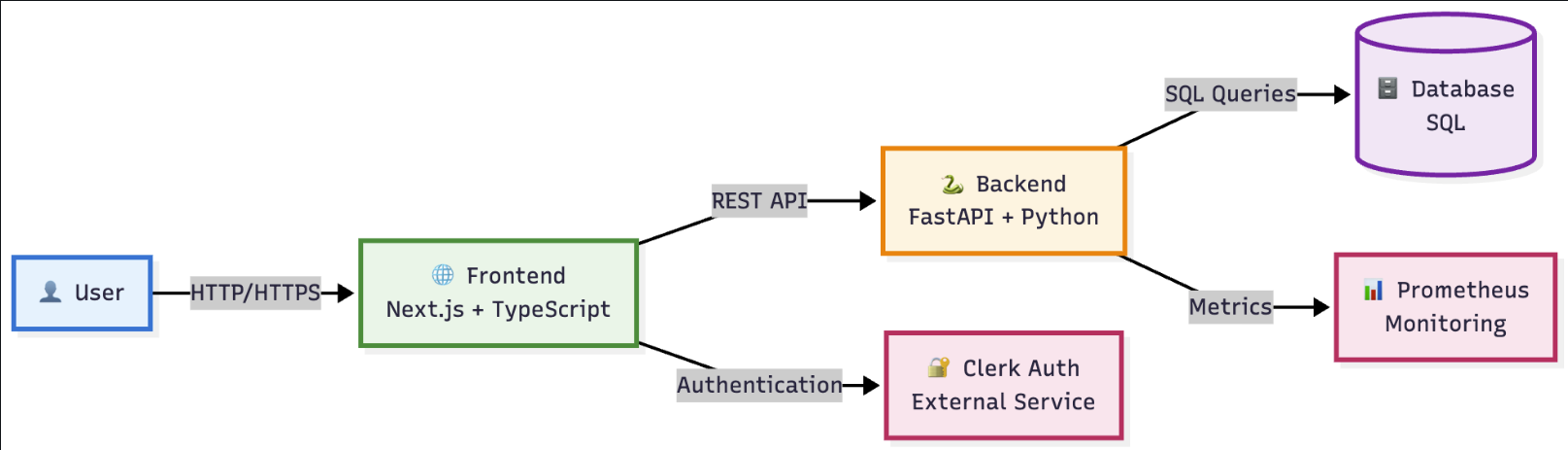
**User Experience and Accessibility:** The application prioritizes intuitive user interactions through responsive design, real-time feedback mechanisms, and accessibility compliance. The unified role model eliminates the complexity of managing separate accounts for different user types, creating a seamless experience for event management and participation.

**Integration:** The system is designed to integrate with external services including payment processors (Stripe/PayPal), communication platforms for ticket delivery, and mapping services for location-based features. The RESTful API architecture ensures compatibility with future integrations and third-party applications.

This document details how these design goals are achieved through specific architectural decisions, technology choices, and implementation patterns that collectively deliver a comprehensive event management solution suitable for both personal and organizational use cases.

# **Software Architecture**

The Event Manager app aims to help individuals and organizations manage and track their events. In order to do so, the team's proposed solution is a full stack application. The team plans to provide this service through a website connected to a backend REST API that provides the needed endpoints for the user to manage and RSVP to events, among other functionalities. The frontend is built using Next.js with TypeScript to deliver a responsive and user-friendly interface for event management. The backend will be composed of 3 main components: the REST API, the SQL database, and monitoring services. The REST API will provide all the functionality needed by the frontend through various endpoints to enable users to control their events. The SQL database will provide a storage solution for information within the application. Additionally, the application integrates Clerk authentication services to handle user management and security, while Prometheus monitoring ensures system reliability and performance tracking. Please see **Figure 1** below for a diagram of the full stack application's design.



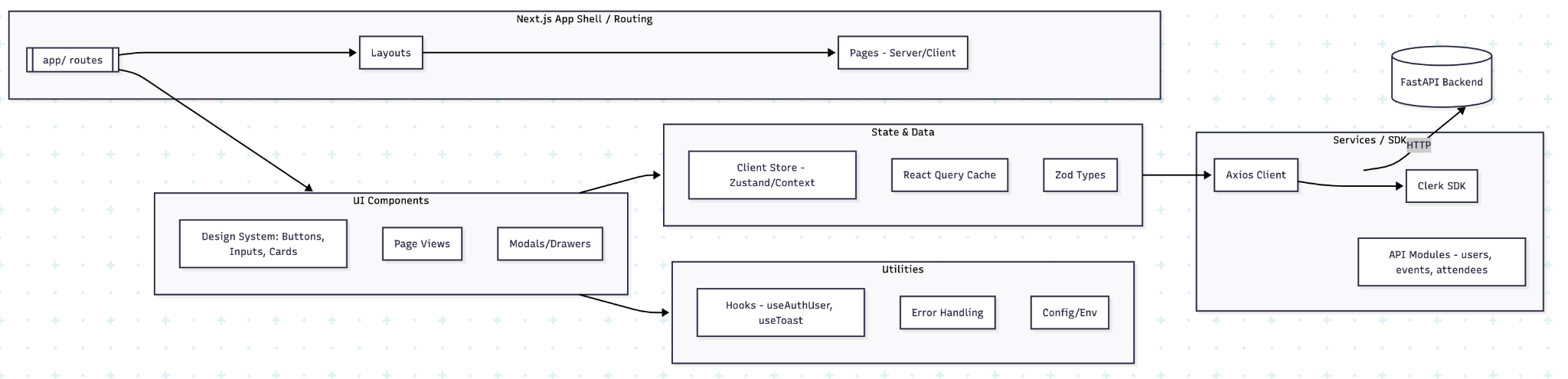
**Figure 1: Full Stack Application Diagram**

## Frontend Architecture

The frontend architecture will be structured in clearly defined layers: Routes, UI components, State & data, services (API/SDK) and utilities. This layered architecture promotes separation of concerns, maintainability and scalability. The frontend is built with Next.js (App router) and Typescript, styled with Tailwind CSS, authenticated via Clerk, and communicates to the backend using Axios.

### Architecture Overview

**Figure 2** below illustrates the overall frontend architecture and the relationships between the different layers:



**Figure 2: Frontend Architecture Overview**

### Router (App Shell) Layer

The Routes layer defines navigation, layouts and server vs. client component boundaries using the Next.js App router.

* Layouts (app/(public)/layout.tsx), app/(protected)/layout.tsx) encapsulate headers, nav, and global providers (e.g. QueryClientProvider, ClerkPRovider, Toaster)
* Routing splits public cs protected areas: Home, Docs, Sign-in/Sing-up, Discover (public) vs Events, Profile (protected)
* Use server components by default for data-fetching pages that can be rendered on the server, and Client Components where interactivity/state is required (forms, modals, etc)
* Page Guards use Clerk middleware and server helpers to redirect unauthenticated users.

### UI Components Layer

The UI layer houses all visual building blocks

* Design System primitives
  + Button, input, select, Modal, Card, Skeleton
    - Generated using shadcdn/ui
* Composites
  + NavBar, sidebar, EventCard, Attendee List, pagination
* Patterns
  + Empty states, loading states, Error boundaries
* Accessability
  + Semantic HTML, ARIA attributes
* Styling
  + Tailwind CSS with a theme token file for colors/spacing

### State and Data Layer

The layer centralizes client-side state management and data validation

* React Query
  + For server data fetching, caching, retries, etc
* Context for UI state (drawer open, wizard step) that should not live in URL or React Query
* Zod
  + Schemas for runtime validation of API responses and forms
* URL state
  + For shareable filters/sorts via search params; uses SearchParams

### Services (API/SDK) Layer

All backend interaction and third-party SDK calls live here

* Axios instance with baseURL from NEXT\_PUBLIC\_BACKEND\_URL, JSON interceptors and Clerk token
* API modules that expose type functions
  + services/user.ts
  + services/events.ts
  + services/attendeed.ts
* Error normalization
  + Map HTTP errors to a consistent shape for UI consumption
* Retries & timeouts: sensible defaults for HTTP requests and REact Query

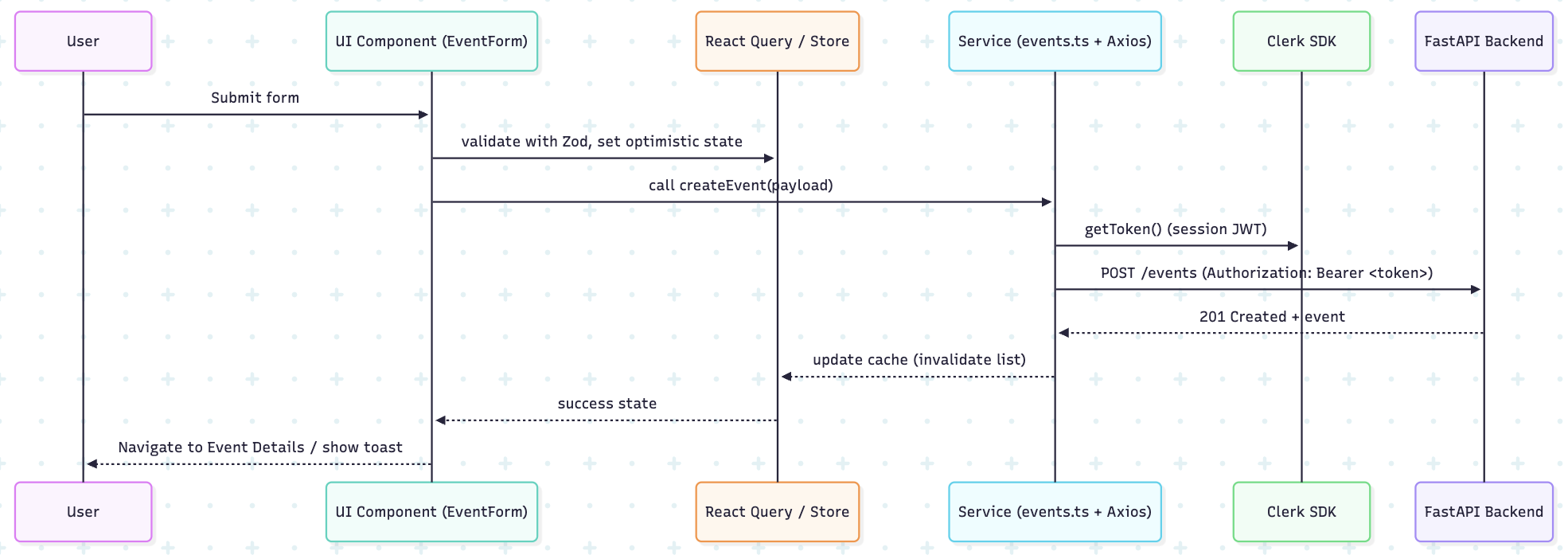
### Utilities Layer

Helpers and hooks that do not fit elsewhere

* Auth helpers for client/server tokens
* Formatter
  + Dates, currency
* Error & Toast hooks
  + sonner

### Request Flow

**Figure 3** illustrates how a typical request flows through the frontend architecture:



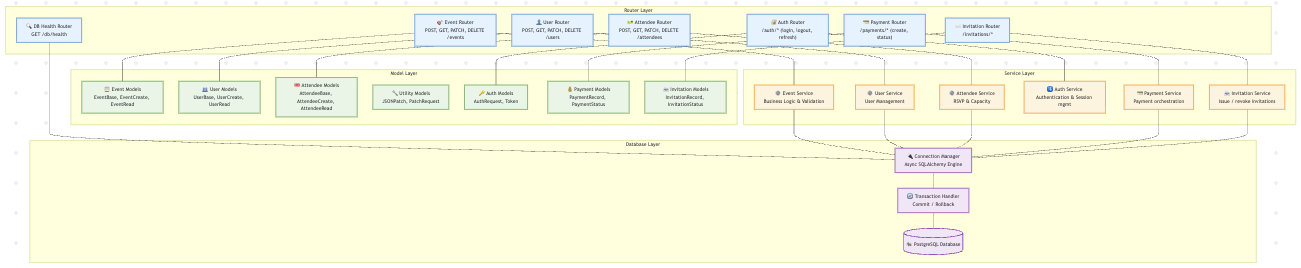
**Figure 3: Frontend Flow Diagram**

## Backend Architecture

The backend will be structured in 4 main layers: model, router, service, and database. This layered architecture promotes separation of concerns, maintainability, and scalability. The backend is built using FastAPI with Python and connects to a PostgreSQL database through SQLAlchemy ORM. Please see a detailed description of each layer in the sections below.

### Architecture Overview

The following diagram illustrates the overall backend architecture and the relationships between the different layers:



**Figure 4: Backend Architecture Overview**

### Model Layer

The model layer will contain the main models used by the REST API, implemented using Pydantic for data validation and serialization. This will help the team define the models needed for each of the endpoints, including Events, Users, Attendees, and utility models for operations like JSON Patch. Defining them in separate files as objects allows the team to address the minimal information needed to create entities while offering flexibility to add fields to models without having to update larger amounts of code, as the models defined are passed from function to function. Each entity follows a consistent pattern with Base, Create, and Read models, providing clear contracts for API requests and responses. The models include comprehensive validation logic to ensure data integrity at the application level before reaching the database.

### Router Layer

FastAPI strongly encourages the router layer as a best practice. This layer allows the entry point for the endpoints to be separated from the FastAPI app configuration, as well as from the service main logic. The routers are organized by domain (events, users, attendees) and include a dedicated health check router for database connectivity monitoring. Each router defines endpoints without containing business logic – instead, it consists of the endpoint definition, request/response models, and a call to the service layer which contains each endpoint's logic. This type of structure encourages separation of responsibility between the different layers/components and enforces that developers first have the models and routers in place before jumping into the service logic, which can be cumbersome if the endpoint body, query, or path parameters are not thought carefully beforehand. The routers also handle HTTP-specific concerns such as status codes, error responses, and API documentation through FastAPI's automatic OpenAPI generation.

### Service Layer

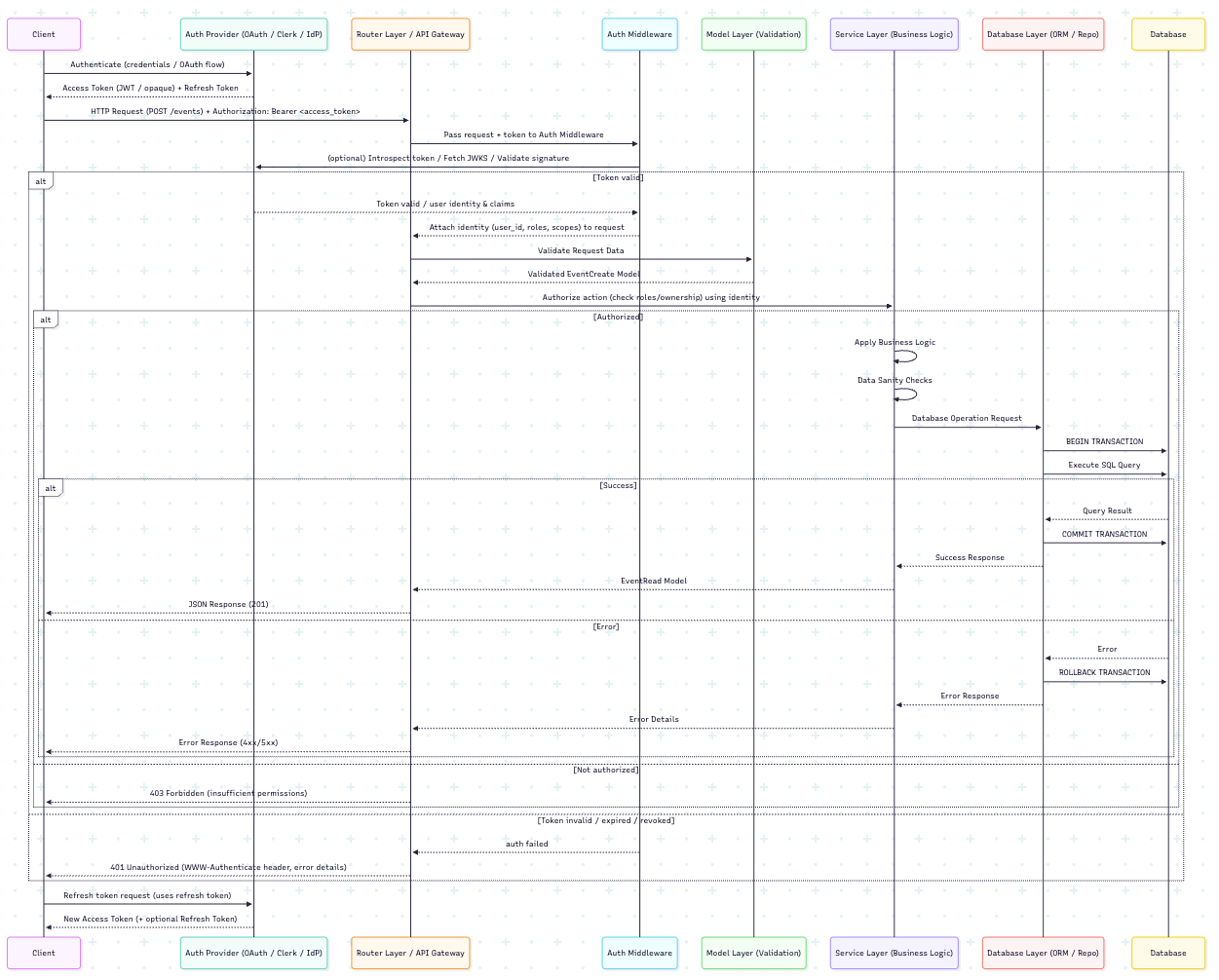
The service layer will take care of the functionality each endpoint needs to provide. Triggered by the routers, the service functions will ensure data sanity and consistency, apply business rules and validation logic, add any necessary modifications, and make use of the database layer to select, create, update, or delete information in the database. This layer encapsulates all domain-specific business logic, such as event capacity validation, user permission checks, and attendee status management. Once all the logic for each respective endpoint is completed, the service logic will return the information the router is expected to return to the user in order to minimize the logic contained in each router. The service layer also coordinates complex operations that may involve multiple database entities and ensures that business rules are consistently applied across different endpoints.

### Database Layer

This layer is responsible for setting up and maintaining the connection to the PostgreSQL database as well as managing all database interactions. Built on top of SQLAlchemy's async engine, this layer provides a robust foundation for database operations. This layer will contain all inserts, updates, selects, and deletes needed to handle events, users, and attendees through well-defined CRUD operations. The team will wrap all database interactions in transactions in order to avoid data corruption and inconsistencies if an error occurs midway through an SQL statement execution. This is possible as transactions support the rollback feature which allows databases to return to the previous clean state before the error occurred. The database layer also includes connection pooling for optimal performance, health monitoring capabilities, and proper error handling to ensure system reliability.

### Request Flow

The following sequence diagram illustrates how a typical request flows through the backend architecture:



**Figure 5: Backend Flow Diagram**

## Overall Folder Structure

Please see the folder structure [here](https://github.com/BUMETCS673/cs673f25a2project-cs673a2f25team5?tab=readme-ov-file#-overall-folder-structure) in the README file in the repo.

# **Class Diagram**

This diagram depicts the architecture of an event management system, which is built to manage user operations, event organization, attendee tracking, and system health checks. The system is structured into three main components, with well-defined dependencies between layers to ensure modularity and ease of maintenance.

## Domain Models

These models define the structure and validation rules for core data entities in the system:

* User-related models: UserBase contains fundamental attributes like first\_name, last\_name, and email. It is extended by UserCreate (used for receiving user creation input) and UserRead (used for returning user data with auto-generated fields such as user\_id, created\_at, and updated\_at).
* Event-related models: EventBase captures event details including event\_name, event\_datetime, and relationships to UserBase (the event creator) and Category(the event type). It is inherited by EventCreate (for event creation input) and EventRead (for outputting event data with event\_id).
* Attendee models: AttendeeBase tracks the relationship between users and events, using the AttendeeStatus enumeration (with values RSVPed, Maybe, and Not\_Going) to indicate participation intent. AttendeeCreate and AttendeeReadextend AttendeeBase for input and output purposes, respectively.

## Backend Layers

The system uses a layered architecture to separate concerns:

* Routers: Handle incoming API requests. RouterUsers, RouterEvents, and RouterAttendees provide endpoints for CRUD (Create, Read, Update, Delete) operations (e.g., POST /users to create a user, DELETE /events/:id to delete an event). RouterHealth exposes a GET /health endpoint to check system status.
* Services: Contain the business logic. ServiceUsers, ServiceEvents, and ServiceAttendees implement core functionalities like createUser(), listEvents(), and patchAttendees().
* DBLayer: Manages interactions with the database, including connection pooling, transactions, and CRUD operations. All services depend on this layer to persist or retrieve data.

## Dependencies

* Routers depend on their corresponding service layers (e.g., RouterUsers relies on ServiceUsers to handle user-related business logic).
* Services depend on domain models (for data validation), DBLayer (for data storage), and utility classes like JSONPatchOperation and PatchRequest (for handling partial updates to resources).

This architecture ensures a clear flow of data: API requests are received by routers, passed to services for business logic processing, and then forwarded to DBLayer for data persistence—with all data validated by Pydantic models at each step.

# 

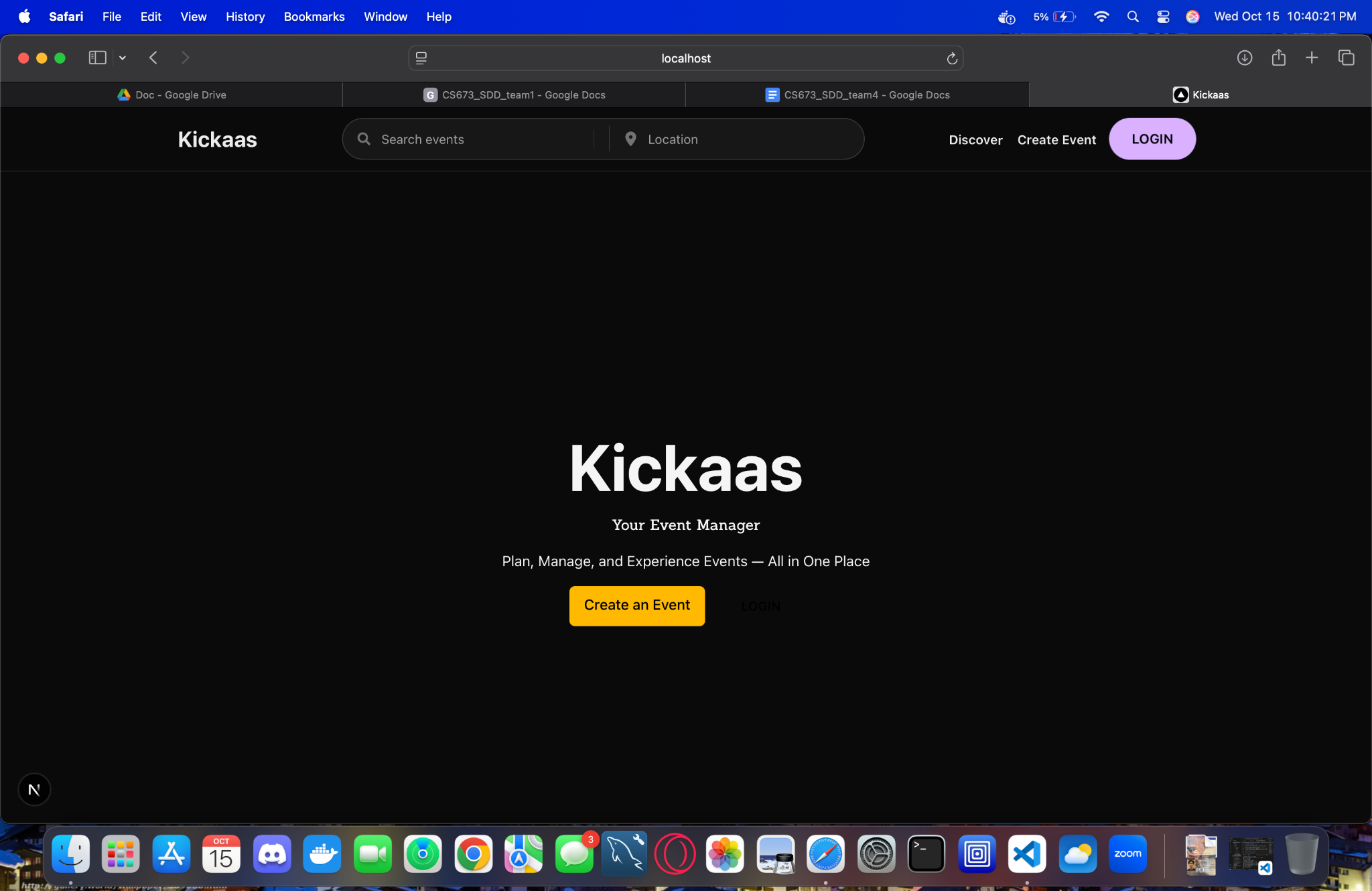
**Figure 6: Class Diagram for Events, Users, and Attendees**

# **UI Design**

The Event Manager’s UI design uses a Next.js framework, Tailwind CSS features, and React Font Awesome 6 (fa6) Icons which allow for a more modern and customizable application. Our main pages are shown below in **Figures 7-10** these include the landing page, create an event form, and the sign up/login form, respectively. These pages include several key features including:

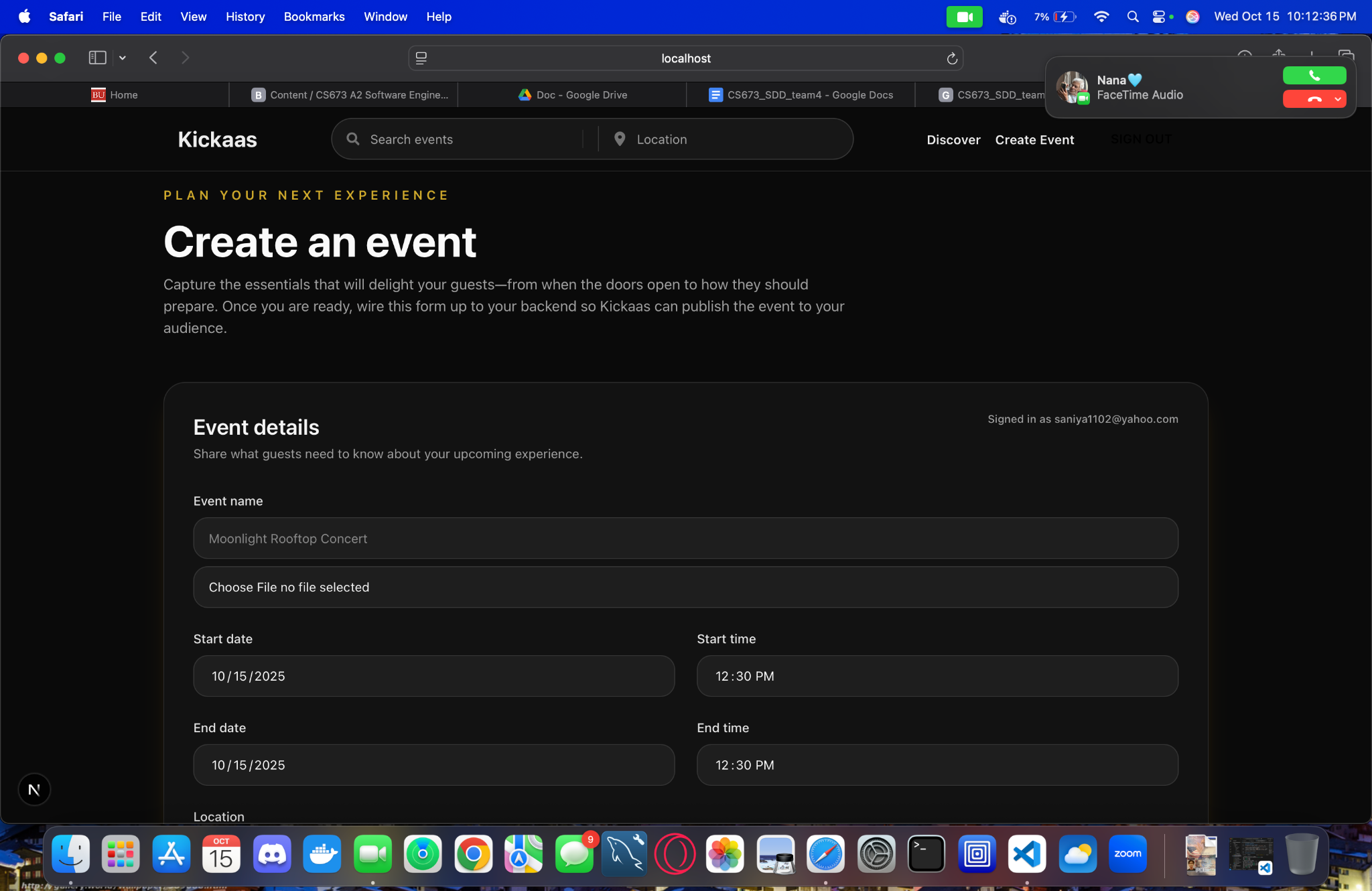
* Landing Page:
  + Header
    - Search Bar
      * Name
      * Location
    - Create Event Tab
    - Events Tab
    - Profile Tab
    - Sign Up/Login Tab
  + Create an Event Button
* Create an Event Form:
  + Event Name Field
  + Category Field
  + Start Date Field
  + Start Time Field
  + End Date Field
  + End Time Field
  + Location Field
  + Capacity Field
  + Price Field
  + Description Field
  + Save Event Draft Button
* Sign Up/Login Form:
  + Sign In with Google Button
  + Email Field
  + Login Button

## Landing Page

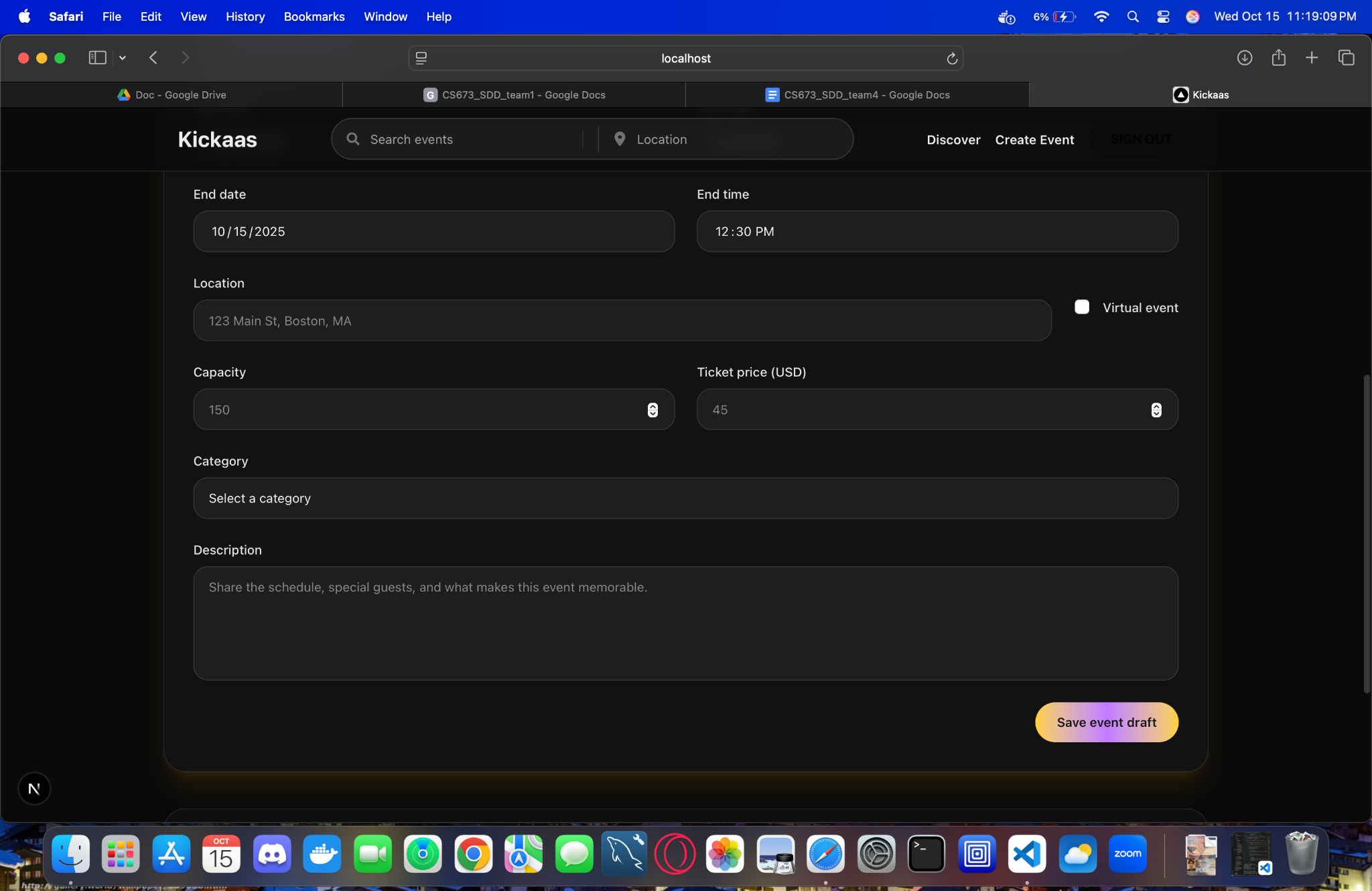


**Figure 7: Landing Page**

## Create an Event

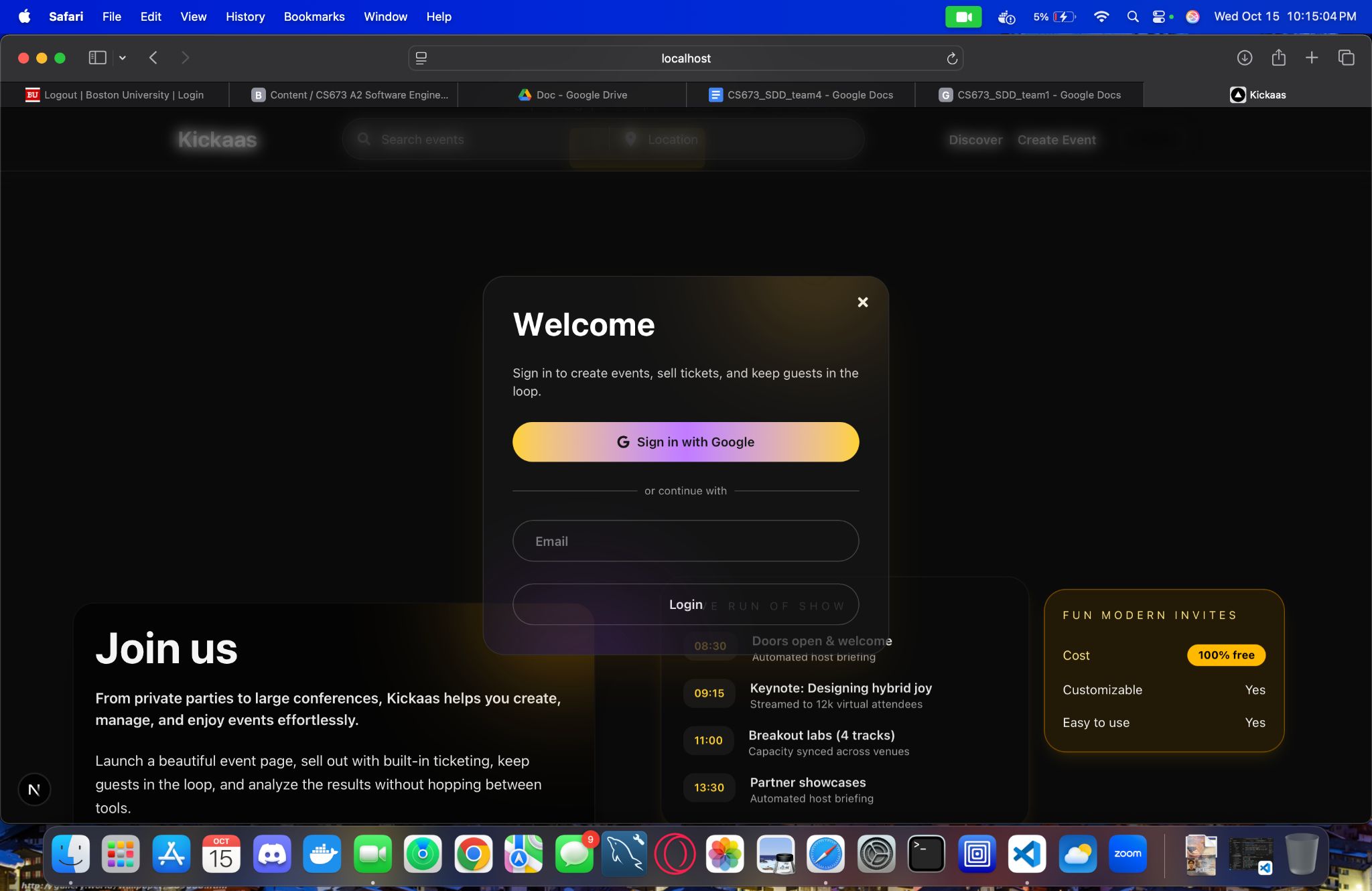


**Figure 8: Create an Event Form**

****

**Figure 9: Create an Event Form pt. 2**

## Sign Up/Login



**Figure 10: Sign Up/Login Form**

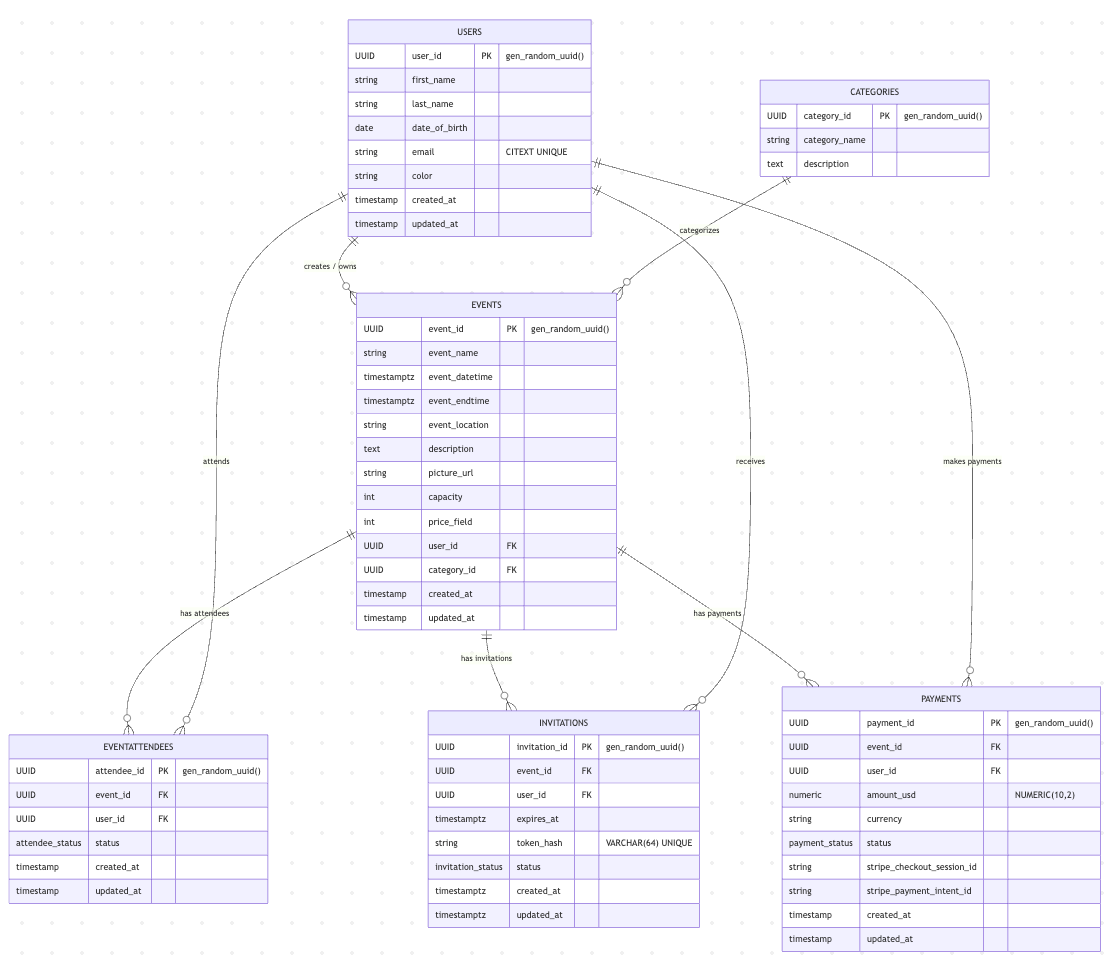
# **Database Design**

The Event Manager application's backend leverages a relational database model implemented with PostgreSQL to ensure data integrity, scalability, and efficient query performance. The database schema is designed to support core functionalities such as user management, event creation, categorization, and attendee registration, while maintaining clear relationships and enforcing referential integrity through foreign key constraints.

The design follows industry best practices for normalized relational schemas, enabling flexible querying, easy future extensibility, and robust transaction support. Database migrations and schema evolution are managed through SQL scripts and SQLAlchemy ORM definitions, ensuring consistency across development and production environments.

## Entity Relationship Diagram (ERD)

The following ERD illustrates the primary tables and their relationships within the system:



**Figure 11: Event Manager Database ERD Diagram**

ERD Relationships

1. Users to events (one to many)
   1. Each **User** can create many **Events**, but each **Event** is created by exactly **one User**.
2. Categories → Events (One-to-Many)
   1. Each **Category** can have multiple **Events**, but each **Event** belongs to one **Category**.
3. **Events ↔ Event\_Attendees** (Many-to-Many)
   1. An **Event** can have many **EventAttendees** attending. A **User** can attend many **Events**.

## Tables and Relationships

### Users

The Users table stores all registered user profiles, capturing essential information such as name, email, date of birth, and profile attributes.

Key fields:

* user\_id (UUID, primary key)
* first\_name, last\_name
* email (unique)
* date\_of\_birth
* color (optional profile attribute)
* created\_at, updated\_at (timestamps)

Each user may create multiple events and register for any event as an attendee.

### Events

The Events table records all planned events, linking each event to its creator (organizer) and its category.

Key fields:

* event\_id (UUID, primary key)
* event\_name, event\_datetime, event\_endtime
* event\_location, description, picture\_url
* capacity, price\_field
* user\_id (foreign key, references Users)
* category\_id (foreign key, references Categories)
* created\_at, updated\_at (timestamps)

Events are uniquely identified and connected to both their organizer and a category, supporting efficient filtering and search.

### EventAttendees

The EventAttendees table tracks attendee registrations for each event, enabling RSVP functionality and capturing attendance status.

Key fields:

* attendee\_id (UUID, primary key)
* event\_id (foreign key, references Events)
* user\_id (foreign key, references Users)
* status\_attendee\_status (e.g., RSVPed, Maybe, Not Going)
* created\_at, updated\_at (timestamps)

This junction table establishes a many-to-many relationship between users and events, allowing users to register for multiple events and events to have multiple attendees.

### Categories

The Categories table enables event categorization, supporting search and organization.

Key fields:

* category\_id (UUID, primary key)
* category\_name
* description

Categories provide a way to group and filter events by type or theme.

## Referential Integrity and Constraints

* Foreign key constraints are enforced on event\_id, user\_id, and category\_id fields to maintain data consistency.
* Cascading rules ensure that deleting a user or event appropriately updates related records.
* Unique constraints on primary keys and certain fields (e.g., email) prevent duplication.

## Design Considerations

* Scalability: UUIDs are used as primary keys for distributed scalability and uniqueness.
* Extensibility: The schema allows for easy addition of new fields, such as event analytics, ticketing, or payment integration.

The database design ensures reliable data storage and efficient access for all application features, while supporting future enhancements and integrations.

# **Security Design**

This section outlines how the Event Management system ensures security across all layers. It explains how authentication, data protection, and input validation are handled, along with measures for frontend safety and CI/CD pipeline security. Each component has been designed to safeguard user data, prevent unauthorized access, and maintain the integrity of the system during development and deployment.

## Authentication & Authorization

### Frontend

This workflow shows how a user signs up or logs in using Google through Clerk. Clerk takes care of the login process and then securely tells the Next.js app that a new user was created. Next.js verifies that this message came from Clerk and then proceeds to send the user’s basic information to the backend API to save it. Once that’s done, the user is logged in and can access private pages in the app.

This workflow permits users to sign up with Google via Clerk. When a new user is created in Clerk, Clerk sends a signed webhook to the Next.js app. The Next.js route verifies the signature, then syncs the new user to the backend (/create-user/). Middleware protects private routes while allowing the webhook through. All requests from outside the application are automatically blocked through Next.js but in the middleware file, I specifically allowed requests to come from Clerk.

Security Key points:

* Uses verifyWebhook to validate the signature from Clerk (required).
* Only acts on user.created; all other events are logged and ignored.
* Posts a minimal payload { email, firstName, lastName } to the backend.

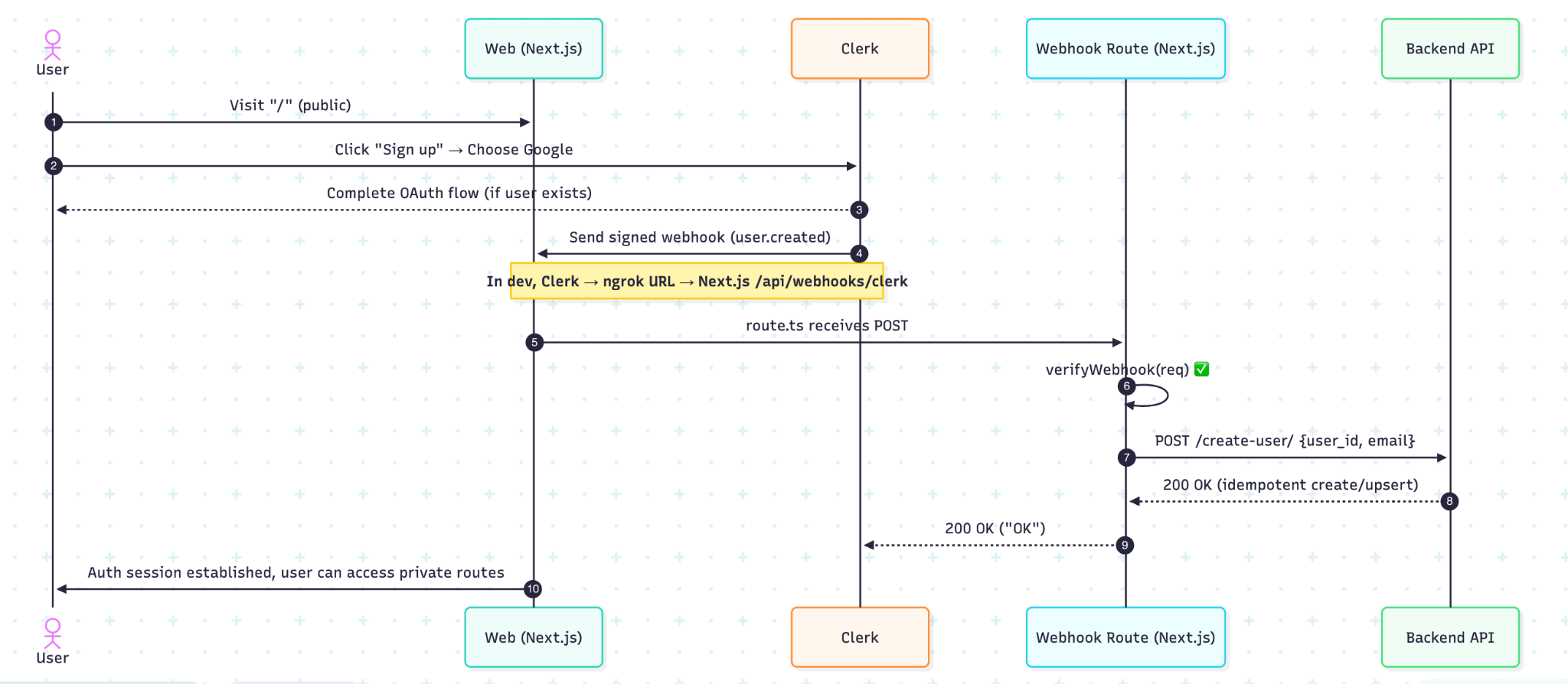
ENV varirables:

* CLERK\_PUBLISHABLE\_KEY
* CLERK\_SECRET\_KEY
* CLERK\_WEBHOOK\_SIGNING\_SECRET ← used by verifyWebhook
* NEXT\_PUBLIC\_BACKEND\_URL ← e.g., http://localhost:8000/ on dev

Point the Clerk webhook to the public ngrok URL: https://<ngrok-subdomain>.ngrok.io/api/webhooks/clerk

Sub domain auto rotates on restart, so Clerk webhooks have to be changed every time. Currently I just have it always running on my laptop (probably awful for my battery, but I can't be asked to change it every time)

On prod since our URL is always going to the same, changing webhook URL on clerk is not required.



**Figure 12: Frontend Authentication Flow Diagram**

### Backend

The backend enforces authentication for all protected REST endpoints by requiring a Clerk-issued session/token in the Authorization header. A lightweight auth dependency (get\_current\_user) extracts the Bearer token and, when Clerk auth is enabled, verifies it using Clerk’s JWKS/verification endpoints or the Clerk SDK (checks signature, issuer/audience, expiry and email\_verified). On success the dependency injects a typed user payload into route handlers; on failure it fails fast with 401 (WWW-Authenticate header) or 503 if the verification service is unavailable.



**Figure 13: Backend Authentication Flow Diagram**

## Data Protection

* All environment secrets such as database URLs, API keys, and Clerk credentials are securely stored in GitHub Actions Secrets or local .env files that are ignored by Git.
* In development, the FastAPI app runs using:

**uv run uvicorn app.main:event\_manager\_app --reload**

* This uses HTTP locally.

## API & Input Security

This section explains how the backend validates, sanitizes, and protects incoming data.

* **FastAPI + Pydantic** automatically validate and structure request data.
* API endpoints only accept defined fields, rejecting malformed input.
* Verified user data comes from **Clerk webhooks** instead of direct authentication.
* Database queries use **parameterized access** through the service layer to prevent SQL injection.

## Frontend Security

The Next.js frontend uses Clerk for secure user authentication and session management. Middleware (middleware.ts) protects private routes like /discover and /onboarding while allowing webhooks and static assets to remain public. Clerk manages login, logout, and session cookies automatically, reducing exposure to manual token handling. Next.js also escapes user input in JSX by default, helping prevent cross-site scripting (XSS). Environment variables (like Clerk keys and backend URLs) are stored in .env.local files which are git-ignored for safety.

This section describes how the frontend handles user sessions and protects private routes.

* The **Next.js frontend** uses **Clerk** for authentication and session management.
* [middleware.ts](http://middleware.ts) restricts access to private routes like /discover and /onboarding.
* **Clerk** automatically manages login, logout, and session cookies.
* **Next.js** safely escapes JSX output to prevent XSS attacks.
* Environment variables (Clerk keys, backend URLs) are stored in .env.local and ignored by Git.

## Payments

Stripe acts as the trusted third-party payment processor. The backend never stores or handles raw card information. Instead, it generates a Stripe Checkout Session that safely collects payment details. After the user completes payment, Stripe notifies the backend via a signed webhook, ensuring payment records remain accurate even if the user closes their browser or loses internet connection.

The high-level flow is as follows:

* Frontend requests a Checkout Session (amount, event\_id, user\_id, email).
* Backend creates a Payment record in “created” state.
* Backend creates a Stripe Checkout Session and stores the session ID.
* The user is redirected to the Stripe-hosted payment page.
* Stripe sends a signed webhook to the backend after payment succeeds.
* Backend updates the payment record to “succeeded”.

## CI/CD Security (GitHub Actions)

This section covers automated security checks and safe deployment in the CI/CD pipeline.

* **GitHub Actions** automate builds and tests.
* **Gitleaks** scans the repo for secrets using .gitleaks.toml
* **Semgrep** checks backend and frontend code for vulnerabilities.
* **pip-audit** scans Python dependencies for known security issues.
* **Hadolint** ensures Dockerfiles follow security best practices.
* **npm audit** can be added to the frontend workflow to catch JS package vulnerabilities.
* All keys and tokens are securely stored in **GitHub Actions Secrets**.

# **Rest APIs**

The Event Manager application leverages RESTful API architecture as the communication standard between the frontend and backend systems. REST (Representational State Transfer) provides an industry-standard approach that ensures scalability, maintainability, and platform independence. This architectural choice allows the system to support multiple client applications without requiring backend modifications – whether accessing the service through a web browser, mobile application, or third-party integrations, the backend API remains consistent and reusable. If the team ever decides to develop native mobile applications or alternative frontend frameworks, the existing FastAPI backend can seamlessly support these new clients without any structural changes, maximizing development efficiency and code reuse.

The REST API follows conventional HTTP methods and status codes, providing intuitive and predictable interactions for developers. Each endpoint is designed with clear resource-based URLs and consistent request/response patterns, making the API easy to understand, test, and integrate. The FastAPI framework automatically generates comprehensive OpenAPI documentation accessible at /docs, providing interactive testing capabilities and detailed schema definitions for all endpoints.

## User Endpoints

### POST /users - Create User

Creates a new user account in the system with validation for required fields including name, email, and date of birth. The endpoint accepts a JSON payload conforming to the UserCreate model and returns the created user with system-generated identifiers and timestamps. Input validation ensures data integrity with appropriate error responses for invalid or duplicate data.

### GET /users - Get Users

Retrieves a list of users with optional filtering capabilities. Supports query parameters for filtering based on any user field for broad user listing. Returns a JSON array of UserRead objects containing user details while maintaining privacy by excluding sensitive information. Supports pagination and filtering to handle large user bases efficiently.

### PATCH /users - Update Users

Enables partial user profile updates using JSON Patch operations for granular field modifications. This approach allows clients to update specific user attributes without requiring full object replacement, reducing bandwidth and improving performance. Supports batch operations for updating multiple user records simultaneously while maintaining data consistency.

### DELETE /users/{user\_id} - Delete User

Removes a user account from the system using the unique user identifier. Implements proper cascade deletion logic to handle associated user data including created events and attendee records. Returns the deleted user object for confirmation and audit purposes, ensuring data integrity throughout the deletion process.

## Event Endpoints

### POST /events - Create Event

Creates a new event with comprehensive validation for event details including name, datetime, location, and capacity constraints. Accepts rich event data including descriptions, images, pricing, and categorization. Validates business rules such as ensuring end time occurs after start time and capacity limits are positive. Returns the complete event object with system-generated identifiers ready for attendee registration.

### GET /events - Get Events

Retrieves a list of events with optional filtering capabilities. Supports query parameters for filtering based on any event field for broad event listing. Returns comprehensive event information suitable for display in event listings and detailed views.

### PATCH /events - Update Events

Provides flexible event modification capabilities using JSON Patch operations for precise field updates. Enables event organizers to update event details, modify capacity, change locations, or adjust timing while maintaining referential integrity with existing attendee registrations. Supports bulk operations for managing multiple events efficiently.

### DELETE /events/{event\_id} - Delete Event

Removes an event from the system with proper handling of associated attendee registrations and related data. Implements authorization checks to ensure only event creators can delete their events. Returns the deleted event object and handles notification workflows for registered attendees affected by the event cancellation.

## Attendee Endpoints

### POST /attendees - Create Attendee

Registers a user's attendance for a specific event, creating the relationship between users and events. Validates event capacity constraints, prevents duplicate registrations, and supports different attendance statuses (RSVPed, Maybe, Not Going). Enables the core RSVP functionality that connects event organizers with potential attendees.

### GET /attendees - Get Attendees

Retrieves a list of attendees with optional filtering capabilities. Supports query parameters for filtering based on any attendee field for broad attendee listing. Supports analytics queries for understanding event popularity and attendance patterns across the platform.

### PATCH /attendees - Update Attendee

Allows modification of attendance status and related attendee information using JSON Patch operations. Enables users to change their RSVP status or update attendance preferences while maintaining accurate event capacity tracking and organizer visibility into registration changes.

### DELETE /attendees/{attendee\_id} - Delete Attendee

Removes an attendee registration, effectively canceling a user's participation in an event. Updates event capacity tracking and provides confirmation of the cancellation. Maintains audit trail of attendance changes while ensuring accurate attendee counts for event planning purposes.

## Payment Endpoints

### POST /payments/checkout-session – Create Checkout Session

Creates a Stripe Checkout Session for an event payment. The backend validates the event and user, prevents duplicate successful payments, and creates a new payment record with status created. It then contacts Stripe to generate a secure hosted checkout page and stores the resulting session ID. The endpoint returns the checkout URL that the frontend uses to redirect the user to Stripe for payment.

### POST /payments/webhook – Process Stripe Webhooks

Handles payment status updates sent by Stripe. The backend verifies the webhook signature to ensure authenticity and processes events such as checkout.session.completed. When a valid event is received, the corresponding payment record is updated (e.g., marked as succeeded). This ensures payment status remains accurate even if the user does not return to the application after paying.

## Invitations Endpoints

# **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:** |
| --- | --- | --- | --- | --- | --- |
| Copilot | Javier | Create overall architecture diagram using mermaid | Very helpful | Copilot provided a great starting point which I then refined further with specific details. | Can’t find how to share an open link with copilot |
| ChatGPT | Vamsi | Modified DB standards and learned about why relational databases should be used. | helpful | ChatGPT provided db column naming conventions and datatype conventions. It also allowed me to fix errors with some datatypes and get the db running on EC2 | The link has expired and does not work. |
| Copilot | Javier | Create backend flow diagram using mermaid | Very helpful | Copilot provided a great starting point which I then refined further with specific details. | Can’t find how to share an open link with copilot |
| Copilot | Javier | Provide introduction template | Helpful | Copilot provided a great template for the introduction section. I then had to adapt it to our style, structure, and project. | Can’t find how to share an open link with copilot |
| Copilot | Javier | Provide template for the REST API section | Very helpful | Copilot provided a great level of detail, therefore, I barely had to modify the content. The main thing was to adapt the style to my writing. | Can’t find how to share an open link with copilot |
| ChatGPT | Shreya | Create security design section | Very helpful | ChatGPT provided proper guidance on implementation of security design. | Link does not work |
| Copilot | Johannes | Created information for PR requests | Very helpful | Copilot gave a summarized review on the PR request I made | Can’t find how to share an open link with copilot |
| ChatGPT | Johannes | Created documentation for Mermaid diagrams of high-level system design | Very Helpful | ChatGPT created user diagrams to explain the structure of the project | <https://chatgpt.com/g/g-p-68cac87d6e9481918745b5f7f40a443c-even-planner-manager/c/68e52858-f844-8325-b98d-417b2203f014> |
| Cursor | Johannes | Fixed backend failing to connect to PostgreSQL on localhost:5432, | Not that Helpful | I couldn’t run the backend on local so was forced to switch to docker build instead | Can’t find how to share an open link with cursor |

# **References**

# **Glossary**