EVENT MANAGEMENT SYSTEM

Software System Design

CSCI-P465/565 (Software Engineering II)

Project Team

Aishwarya Kore

Shanti Jangam

Yudai Hirata

Tushar Khatri

Tribhangin Dichpally

---

1. Introduction

TeamUp is built on modern web development principles, focusing on user experience, scalability, and maintainability. Our approach is centered around agile methodologies, enabling iterative development and continuous improvement based on user feedback. The architecture follows a client-server model with React-based frontend and Node.js backend, allowing for separation of concerns and independent scaling of system components.
Security is integrated at every layer, ensuring protection of user data and transactions. Our carefully selected technology stack—React, TypeScript, Node.js, Express.js, and PostgreSQL—provides a robust foundation for development, offering both performance and flexibility. This comprehensive approach positions TeamUp for success in the current market while allowing for future growth and feature expansion.

1.1 System Description:

TeamUp is a comprehensive sports venue management platform designed to connect venue owners with sports enthusiasts. By leveraging a modern, responsive interface built with React and TypeScript, and a robust Node.js/Express.js backend with PostgreSQL database, TeamUp aims to streamline the process of discovering, booking, and managing sports venues and activities.

The platform addresses common pain points in the sports venue management ecosystem, providing a centralized location for users to find available venues based on location, sport type, and time availability. For venue owners, it offers powerful tools to list facilities, manage bookings, and communicate with users.
In this sprint, the primary focus is on the following initiatives:

In this sprint the primary focus is on the following initiatives:

1. Implementing a secure and user-friendly authentication system with multi-factor authentication.
2. Creating a comprehensive venue listing and booking functionality.
3. Developing search capabilities with filters for sport name, venue name, and location.
4. Finalizing the Web Design Template for the Application.
5. Connecting Backend with Databases by creating and managing well-structured Models and Schemas.

These targeted improvements are designed to refine TeamUp's core features, with particular emphasis on user experience, security, and establishing a platform that connects sports enthusiasts with venue owners efficiently.

1.2 Design Evolution

1.2.1 Design Issues

The design of TeamUp was driven by several key issues and requirements, including:

- Scalability: To accommodate a growing user base and venue inventory without performance degradation
- Security: To protect user data and ensure secure transactions with authentication mechanisms including OAuth and two-factor authentication
- User Experience: To provide an intuitive, responsive interface accessible across various devices
- Real-time Communication: To facilitate immediate interaction between users and venue owners
- Location Services: To enable venue discovery based on proximity and geographic data
- Payment Integration: To support secure transaction processing for venue bookings

1.2.2 Candidate Design Solutions

During the initial design phase, several architectural patterns and technologies were discussed:

- Frontend Frameworks: React, Angular, and Vue.js were evaluated for building the user interface
- State Management: Redux, Context API, and MobX were considered for managing application state
- Backend Architecture: Monolithic, microservices, and serverless architectures were assessed. Packages such as Zod, Multer & AJV were finalized for specific purposes.
- Database Solutions: Relational (PostgreSQL, MySQL) vs. NoSQL (MongoDB) options were compared.
- Authentication Strategies: JWT-based authentication, OAuth providers, and custom authentication systems were evaluated.
- Deployment Models: Traditional server deployment vs. container-based solutions vs. serverless options.

1.2.3 Design Solution Rationale

After thorough evaluation, the development team selected a modern stack with React, TypeScript, and Vite for the frontend, Node.js and Express.js for the backend, and PostgreSQL for data storage. This decision was based on several factors:

- React and TypeScript: Provides component-based architecture for modular development, strong typing for code reliability, and excellent developer tooling.
- Vite: Offers faster development cycles with hot module replacement and optimized builds.
- Node.js and Express.js: Enables JavaScript across the stack, simplifying development and maintenance.
- PostgreSQL: Provides robust relational database capabilities with excellent support for geographic data, which is essential for location-based venue search.
- Material-UI: Offers a comprehensive set of pre-built, customizable UI components that follow the Material Design principles.
- Redux Toolkit: Simplifies state management with a more opinionated approach than plain Redux.

This combination offers a balance of modern development practices, performance, and maintainability, addressing the core design issues while providing flexibility for future expansion.

1.3 Design Approach

1.3.1 Methods

The development of TeamUp is guided by a structured yet flexible approach, incorporating best practices from agile methodologies to ensure adaptability and continuous improvement.

- Agile Development: We've adopted an agile approach with two-week sprints, allowing for regular reassessment and adjustment of project priorities.
- User-Centered Design: Development decisions are informed by user needs and feedback, ensuring the platform addresses real-world use cases.
- Iterative Process: Features are developed incrementally, with regular testing and refinement cycles.
- Component-Based Architecture: The system is built using modular, reusable components that can be developed and tested independently.
- Continuous Integration/Continuous Deployment (CI/CD): Automated pipelines ensure code quality and facilitate regular releases.

1.3.2 Standards

- Coding and Design Standards: We adhere to well-established coding conventions and design standards to ensure code quality and maintainability. This includes following the Agile Manifesto principles, which advocate for working software, customer collaboration, and responding to change.
- UI/UX Best Practices: Our design is informed by best practices in UI/UX, ensuring that each iteration enhances the user experience. We prioritize accessibility, usability, and intuitive design, consistent with Agile user-centric focus.
- Component Modularity: In line with Agile iterative approach, our system architecture and components are designed for modularity and flexibility, allowing for easier updates and refinements with each iteration.

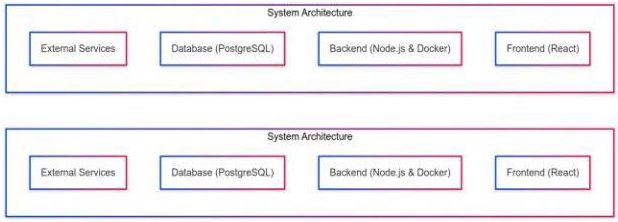
1.3.3 Tools

For the RentItAll.com project, we leverage a selection of essential tools to streamline our design and development processes:

- JIRA: This project management tool helps us organize and track all aspects of the project, from initial planning to final delivery. It enables the creation of detailed project roadmaps, sprint plans, and task tracking to ensure timely progress and clear communication among team members.
- GitHub: As our primary version control system, GitHub facilitates collaborative coding, allowing team members to contribute, review, and merge code seamlessly. It maintains a comprehensive version history, aiding in change management and issue tracking to enhance code quality and collaboration.
- Jenkins: This CI/CD tool is critical for automating the testing and deployment processes. Jenkins runs our code through rigorous tests with each update and automates deployment, ensuring that our application is always in a deployable state and reducing manual intervention.
- Figma: Used for UI/UX design, Figma allows our designers to create detailed and interactive prototypes. This tool provides a collaborative environment for designing and iterating on user interfaces, offering a clear visual guide for developers to implement the UI, enhancing the user experience.
- GitHub Actions: Our go-to CI/CD tool for automating builds, tests, and deployments. It streamlines development by integrating directly with GitHub, reducing manual work & ensuring smooth, fast deliveries.

2. Introduction

2.1 System Design



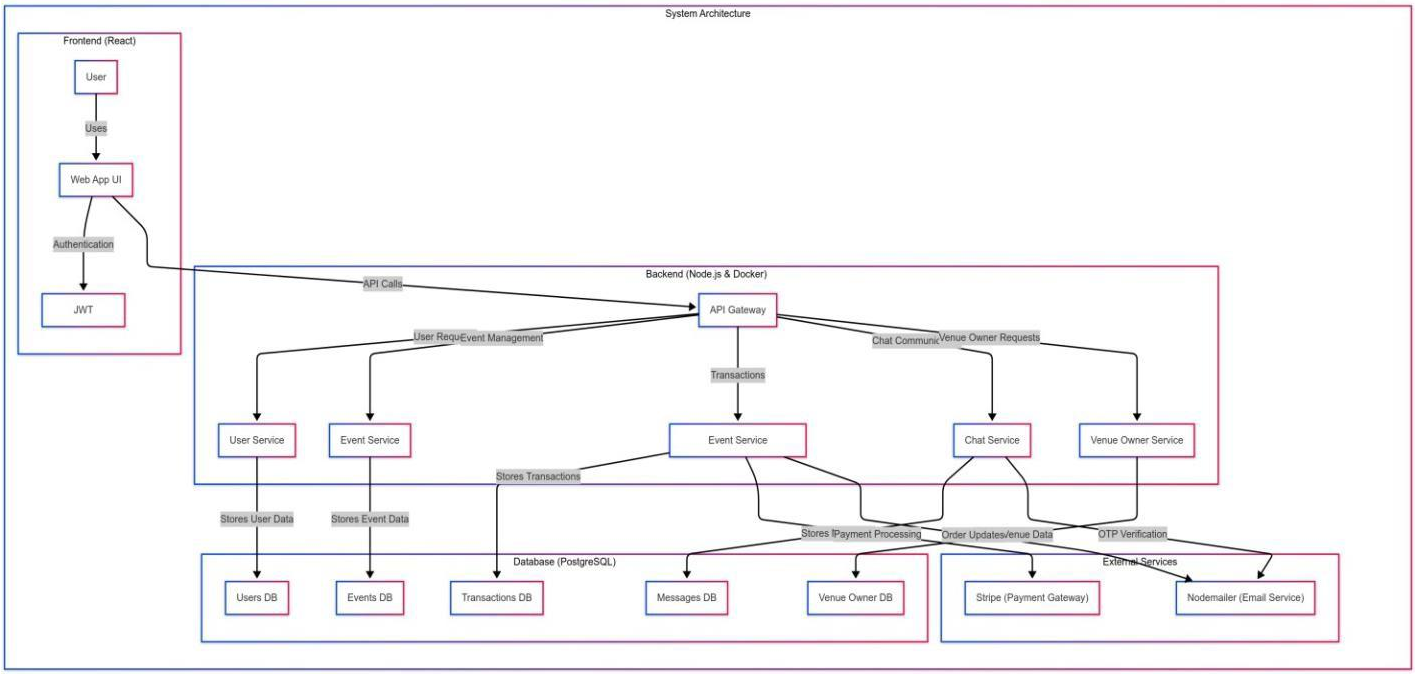


Fig: System Design Architecture of Team Up

TeamUp uses a React and Material UI frontend for a responsive UI and a microservices backend for modular development. An API gateway routes requests to specific services, which communicate through HTTP calls. Data is managed in PostgreSQL via Supabase, ensuring scalability and reliability. The chat system will be implemented through sockets.This design supports efficient development, scalability, and a seamless user experience.

2.2 External Interfaces

TeamUp integrates several external interfaces, each serving specific functions and enhancing the overall capabilities of the system. Below is a detailed description of these interfaces and their interactions with our system:

- JWT for Authentication:
  - Description: JSON Web Token (JWT) is used for secure, stateless authentication, allowing users to log in and access resources without repeated authentication.
  - Relationship with System: Provides a scalable and efficient authentication mechanism by issuing and validating tokens instead of maintaining server-side sessions.
  - Communication Details: Upon successful login, the system generates a signed JWT containing user details and permissions. This token is included in each request for authentication and authorization. The server validates the token on every request, ensuring security. All communication occurs over HTTPS to protect data integrity and confidentiality.

- Email Notifications via Nodemailer and Google Account:
  - Description: Currently, the Nodemailer package with a Google account is used for sending out email notifications for various user interactions such as account verification and alerts.
  - Relationship with System: Supports communication with users outside the platform interface.
  - Communication Details: SMTP protocol is used for sending emails. The data exchanged includes email content templates, recipient addresses, and SMTP configuration details. Secure authentication with OAuth2 is used for accessing the Google account.

- Stripe for Payment Processing (Not yet finalized):
  - Description: (Planned) Stripe will be integrated to handle secure online payments, subscriptions, and refunds.
  - Relationship with System: Will manage all financial transactions within the platform, providing seamless checkout experience.
  - Communication Details: (Planned) The integration will use Stripe's APIs over HTTPS, exchanging information like payment amounts, currency, and customer details. Stripe's webhooks will be used for real-time notifications on transaction status.

3. Component Design (Will be completed with upcoming functionality development)

➢ User Authentication

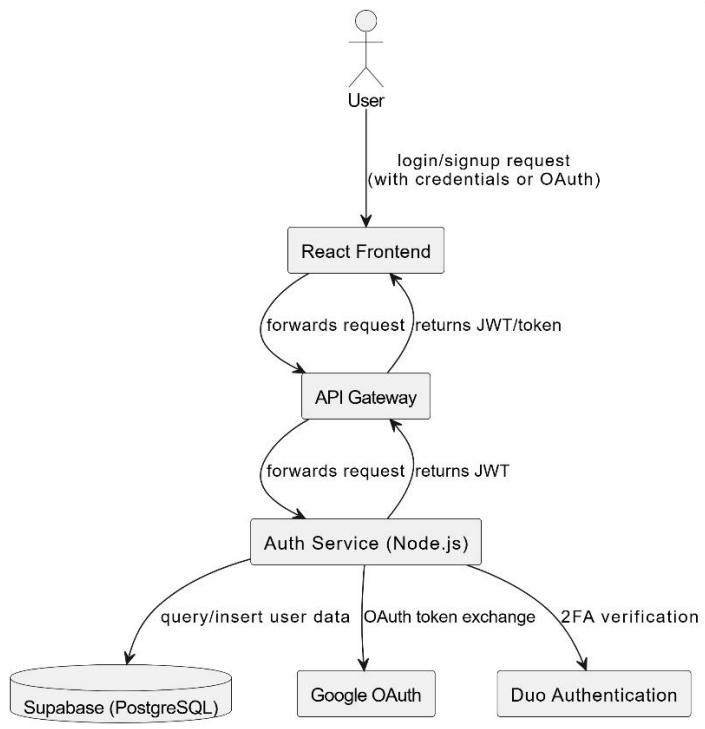
- **<u>Component Name:</u>**
  Users Authentication

- **<u>Component Description:</u>**

This component handles the **secure authentication and authorization** of users within the application. It supports multiple authentication flows, including:
Traditional credential-based login using email and password
OAuth-based login via trusted third-party identity providers like **Google**
Multi-factor authentication (MFA) through **Duo Authentication**, which adds an additional layer of security by requiring users to verify their identity using a second factor such as a mobile device prompt.

- **<u>Responsible Development Team Member:</u>**
  Tushar khatri, Yudai Hirata, Shanti Jangam, Aishwarya Kore

- **<u>Component Diagram:</u>**



➢

- **<u>Component User Interface:</u>**

TeamUp

Discover Tailored Venues.
Sign Up for personalized recommendations today!

Create Account

Sign Up with Google | Sign Up with Facebook

OR

Role

Username

Email

Password

I'm not a robot

Create Account

Already have an account? Log in

TeamUp

Discover Tailored Venues.
Sign Up for personalized recommendations today!

Login

Login with Google | Login with Facebook

OR

Role
User

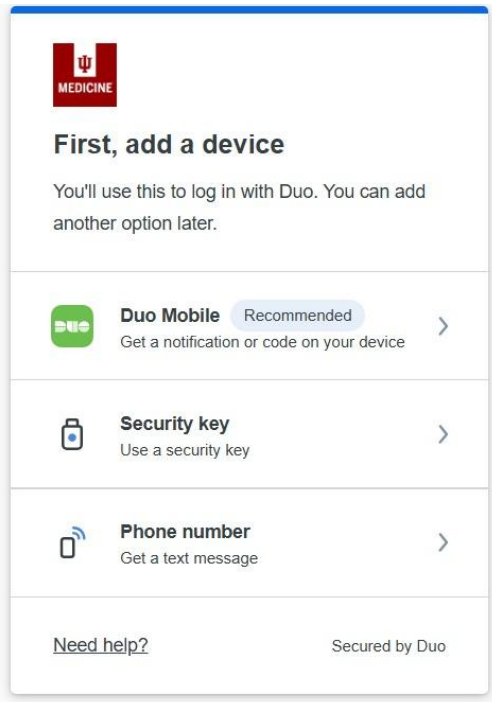
Email
virat.koFFsdd1dssshli18@gmail.com

Password
••••••••••••••••

I'm not a robot

Login

Don't have an account? Sign up



- **<u>Component Objects:</u>**

Component	Description
GatewayController	Handles incoming authentication-related HTTP requests from the frontend (e.g., /login, /register) and delegates them to appropriate services such as ProxyService.
ProxyService	Acts as a forwarding layer using Axios to communicate with the actual **Auth Service** microservice. Manages request payloads, headers (e.g., JWT), and response handling.
Route Definitions	Express Router objects that define and bind HTTP endpoints (e.g., POST /auth/login, POST /auth/register, GET /auth/profile) to controller methods.
HTTP Client (Axios)	Enables inter-service communication from the Gateway to the Auth Service. Also used for exchanging tokens with OAuth providers and Duo.
JWT Handler	Generates and verifies JWT tokens used for stateless session management. Tokens are issued on successful login or registration and validated for protected routes.
OAuth Handler	Manages OAuth flows for providers like **Google**, handling redirection, token exchange, and user profile fetching.
Duo Handler	Orchestrates the **multi-factor authentication (MFA)** flow using Duo's API: initiates the second-factor challenge and verifies completion before final login.
Supabase Interface	Handles direct interactions with the Supabase database for storing and retrieving user credentials, roles, and metadata.

- **Component Interfaces:**

1. Frontend → Gateway Interface

Endpoint	Method	Payload	Description
/auth/register	POST	{email, password, role}	Registers a new user. Role can be user or venue_owner.
/auth/login	POST	{email,password}	Authenticates an existing user. On success, issues JWT and optionally triggers Duo MFA.
/auth/duo/init	POST	{user_id}	*(If enabled)* Initiates Duo MFA challenge. Returns Duo session token.
/auth/duo/callback	POST	{sig_response}	Verifies Duo response. Completes authentication.

2. Gateway → Auth Service Interface (Internal API)

Internal Route	Method	Payload	Description
http://auth-service/register	POST	{email, password, role}	Forwards registration data to Auth service.
http://auth-service/login	POST	{email, password}	Verifies credentials and returns JWT if valid.
http://auth-service/duo/init	POST	{ user_id }	Triggers Duo challenge setup for 2FA.
http://auth-service/duo/verify	POST	{sig_response}	Validates Duo signature to complete login.

- **Component Error Handling:**

The authentication system implements robust error handling at the **Gateway** level to ensure graceful failure and clear communication with the frontend.

Error Handling Strategy (Gateway Service)

- ✅ **Axios Error Wrapping**:
  All outbound HTTP requests (e.g., to Auth service) are wrapped in try-catch blocks to intercept and manage errors gracefully.
- 📄 **Status + Message Forwarding**:
  If the error object contains a response (e.g., from Axios), the gateway extracts:
  - statusCode: error.response.status
  - message: error.response.data.message
    And returns them directly to the frontend to maintain clarity.
- ❗ **Fallback Handling**:
  If the error does **not** include a response (e.g., network failure, internal crash):
  - Responds with 500 Internal Server Error
  - Uses a generic message: "Something went wrong. Please try again later."
- 🛡️ **Security**:
  - Internal stack traces and error objects are **not exposed** to clients.
  - URLs, service names, and raw exception data are sanitized before sending back any response.

➢ User Booking

- **Component Name:**
  Venue Listing

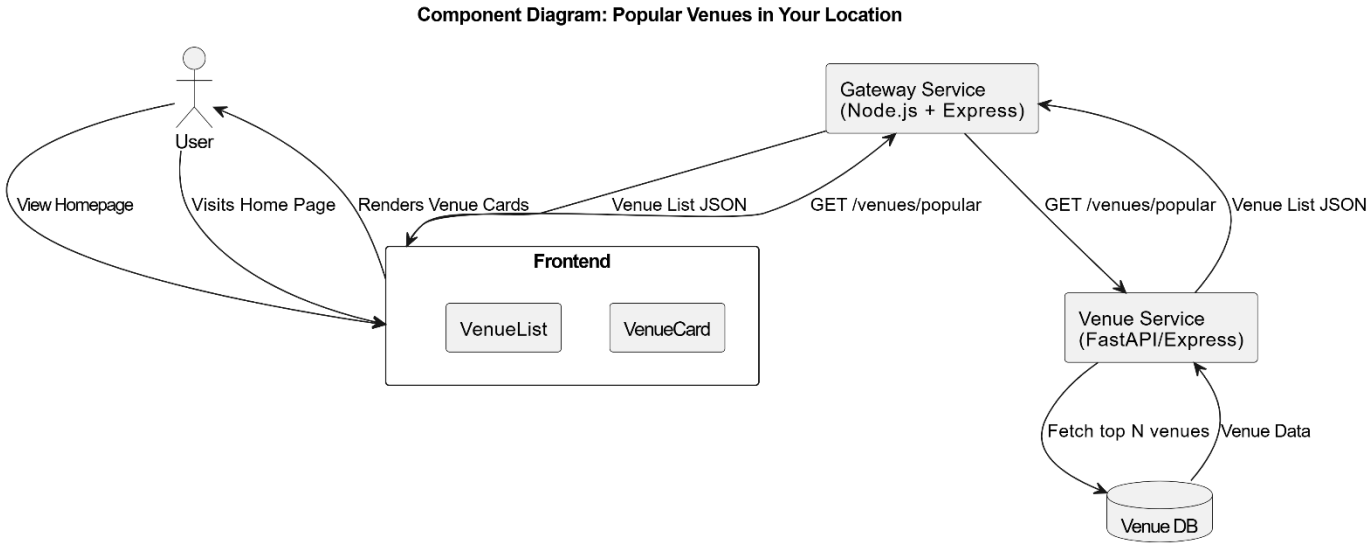
- **Component Description:**

  This component allows logged-in users to view all the venues they have previously booked. It displays booking details such as venue name, location, booking date, time slot, and payment status. The component fetches data securely using the user's session token and supports pagination if bookings exceed a certain number.

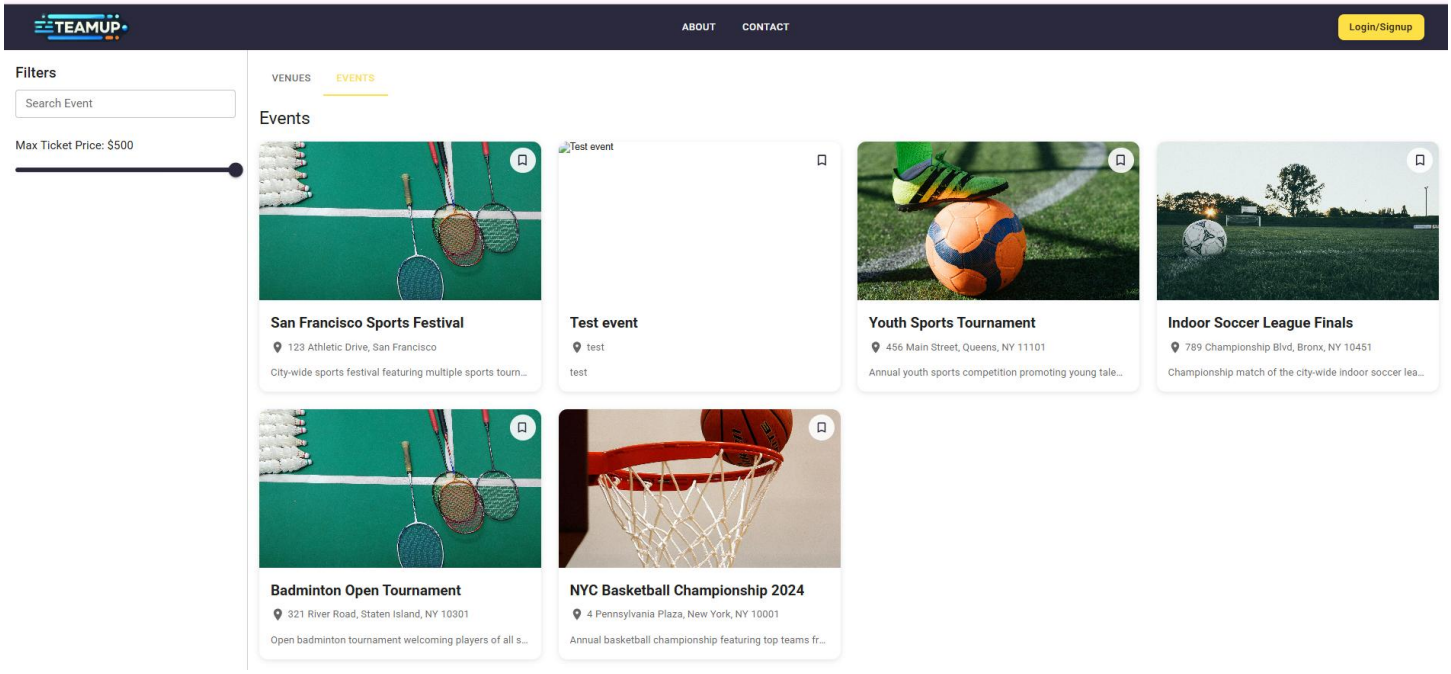
- **Responsible Development Team Member:**

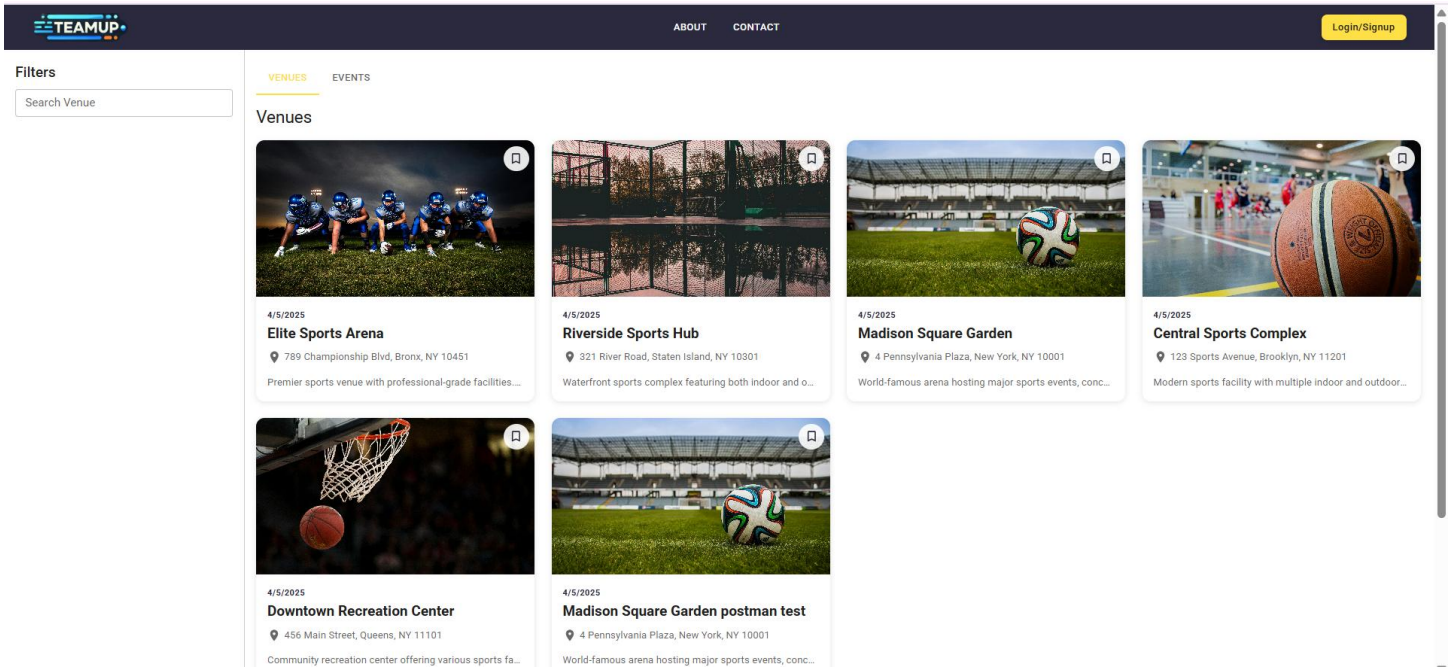
  Frontend Team — React Component Developer
  Backend Team — Booking API Developer

- **Component Diagram:**



- **Component User Interface:**





Book Courts

321 River Road, Staten Island, NY 10301

Venue Description

Waterfront sports complex featuring both indoor and outdoor facilities. Beautiful views and modern amenities.

Available Sports

Basketball | Badminton | Table Tennis

Hosted by

JW

Joy Williams

Venue Owner

Chat with Host

Location

Court Information

Price per court varies by sport type:

- Basketball: ₹1600
- Badminton: ₹900
- Table Tennis: ₹700

- **Component Objects:** User Booking Section

BookingCard.tsx
This component displays an individual booking record. It receives props such as `venueName`, `image`, `date`, `slot`, `status`, and `amount`, and visually represents this data in a compact, user-friendly card. It supports status badges and may include interactive UI elements like a bookmark icon or cancel button depending on booking state.

BookingList.tsx
The BookingList component acts as a container that maps over a list of booking objects and renders each one using the `BookingCard` component. It is responsible for layout management, ensuring responsiveness across different screen sizes. This component also handles conditional rendering to show appropriate messages or loading states.

EmptyStateMessage.tsx
This component is rendered when no bookings are returned from the backend. It typically displays a message like "No bookings yet" along with a prompt or illustration to guide the user. It helps maintain a clean UI even when the user has no bookings.

Loader.tsx
The Loader component provides visual feedback while booking data is being retrieved. It may use a spinner or skeleton loader to indicate that content is being loaded asynchronously.

ErrorBanner.tsx
The ErrorBanner component displays error messages when the booking fetch fails due to issues like server errors, unauthorized access, or network problems. It may include a retry option and ensures that users are clearly informed without exposing technical details.

- **Component Interfaces:**

1. Frontend → Gateway Interface

Endpoint	Method	Payload	Description
/bookings/user	GET	userId=UUID (query param)	Fetches all bookings made by a specific user (both venue & event).
/bookings/details/:bookingId	GET	bookingId as URL param	Fetches full details for a specific booking.
/bookings/cancel/:bookingId	POST	{reason: string}	Cancels a specific booking and updates status in backend.

2. Gateway → Booking Service Interface (Internal)

Endpoint	Method	Payload	Description
http://booking-service/user	GET	?userId=UUID	Retrieves all bookings for the user.
http://booking-service/details/:id	GET	bookingId in URL	Returns details about the selected booking including venue, timeslot, and status.
http://booking-service/cancel/:id	POST	{reason: string}	Updates booking status to "cancelled", logs reason, and updates DB.

3. Booking Service → Supabase (PostgreSQL)

- Fetches data from:
  - venue_bookings table
  - event_bookings table
- Joins with:
  - venues, events, and users to show human-readable info.
- Updates booking status (e.g., active, cancelled, completed).
- Logs timestamp and cancellation metadata.

- **<u>Component Error Handling:</u>**

1. **Empty Bookings**
   If the backend returns an empty array, the UI displays a user-friendly message:
   "You haven't made any bookings yet."
2. **Authentication Errors (401/403)**
   Unauthorized or expired sessions redirect the user to the login screen and clear tokens from storage.
3. **Network / Server Failures**
   Any Axios/network errors (e.g., timeout, 5xx errors) trigger an error banner with retry options.
4. **Missing Venue or Event Info**
   If venue/event metadata is incomplete or missing, fallback UI is shown (e.g., placeholder image or "info unavailable" message).
5. **Cancellation Failures**
   If a booking cancel request fails, a toast notification informs the user:
   "Unable to cancel booking at the moment. Please try again later."
6. **Loader State**
   While fetching bookings, a spinner is displayed. If the request takes too long, a timeout message is shown.

➢ Payment Processing

- **<u>Component Name:</u>**

  Payment Service (Stripe Integration)

- **<u>Component Description:</u>**

The **Payment Service** manages all financial transactions initiated when users book events or venues. It acts as the central system for initiating, validating, and confirming payments securely.

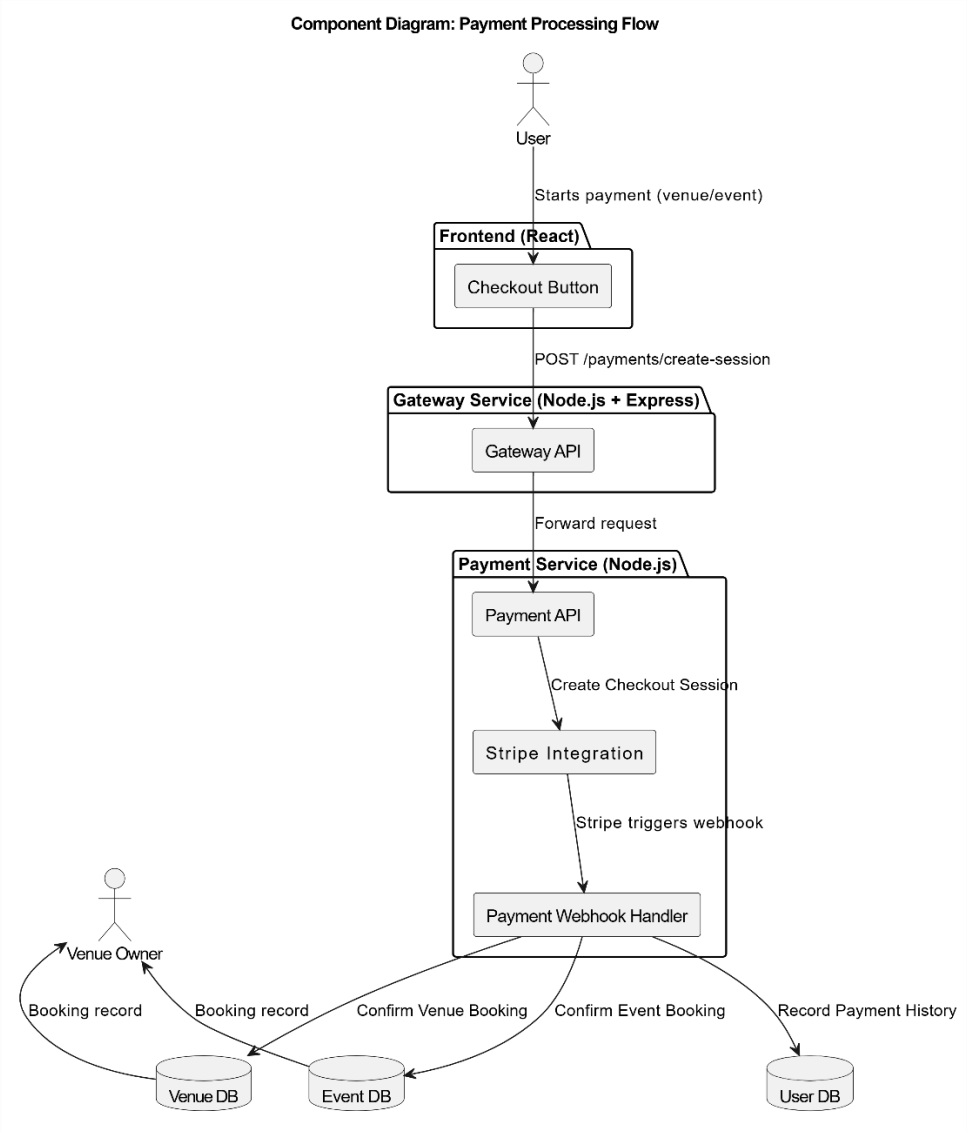
This service is built using **Node.js** and integrates with **Stripe** to handle real-time payments with reliability and PCI compliance.

Key responsibilities include:

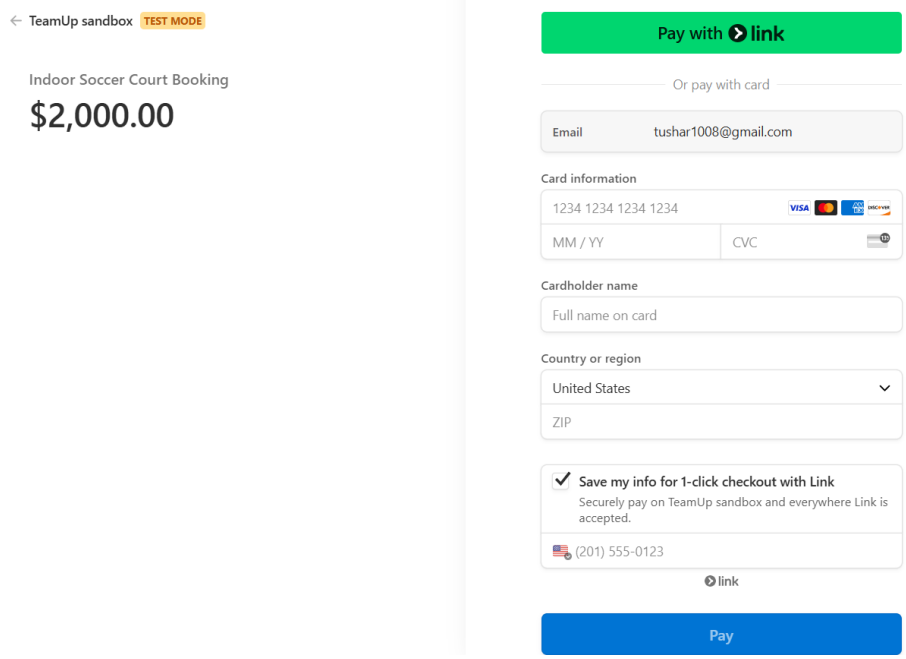
- **Creating Stripe Checkout Sessions**
  When a user initiates a booking, the service validates the booking details (e.g., price, user ID, venue/event ID) and creates a corresponding **Stripe checkout session**.
- **Webhook Handling for Payment Status**
  Upon transaction completion, Stripe sends a webhook notification:
  - On **successful payment**, the service:
    - Confirms the booking in the respective event or venue database.
    - Updates the internal payment history log for the user.
    - Notifies the venue owner (and optionally the user) via the **Email Service** with booking details.
  - On **payment failure or timeout**, the service:
    - Flags the transaction as failed in the system.
    - Sends an email alert to the user explaining the failure reason and includes retry instructions.
    - Ensures no booking is created unless payment is verified as successful.
- **Email Notification Integration**
  The service connects with a dedicated **Email Service microservice**, triggering:
  - **Booking Confirmation Emails** upon payment success.
  - **Payment Failure Alerts** for incomplete or declined transactions.
- **Security & Validation**
  - All Stripe interactions are server-side to ensure secure processing.
  - Webhook signatures are verified using Stripe's secret key.
  - Booking data is re-validated before confirmation to prevent spoofed requests.

- **<u>Responsible Development Team Member:</u>**
  Tribhangin Dichpally

- **Component Diagram:**



- **Component User Interface:**



- **Component Objects:**

1. PaymentSessionRequest
This object is used when initiating a Stripe Checkout Session from the frontend via the gateway and payment service.

userId: a string representing the unique identifier of the user initiating the payment.
bookingType: a string that can be either 'venue' or 'event', indicating what the user is booking.
bookingId: a string identifying the specific venue or event being booked.
amount: a number representing the total payment amount to be charged.
currency: a string representing the currency in which the payment will be made (e.g., "usd").
metadata *(optional)*: an object containing any additional key-value pairs to track contextual info (e.g., timestamp, owner ID).

2. PaymentWebhookPayload

This object is automatically sent from Stripe to the backend webhook endpoint after a checkout session completes.

type: a string representing the event type (e.g., "checkout.session.completed").
data.object.id: a string representing the Stripe session or payment intent ID.
data.object.metadata: an object including:
- **userId**: string identifier of the customer who made the payment.
- **bookingType**: either 'venue' or 'event'.
- **bookingId**: the ID of the booked entity.

data.object.payment_status: status of the transaction ("paid", "unpaid", or "failed").

3. BookingConfirmationRecord

This object is created in the backend after successful payment confirmation.

- **userId**: the identifier of the user who completed the booking.
- **bookingId**: the ID of the venue or event that was booked.
- **bookingType**: the type of the booking, either 'venue' or 'event'.
- **status**: booking confirmation status, typically set to 'confirmed'.
- **paymentIntentId**: the Stripe-generated payment intent or session ID.
- **timestamp**: ISO-formatted timestamp of when the booking was finalized.

Component Interfaces:

- **1. CreateCheckoutSessionRequest**
  Description: Sent from frontend to gateway to begin checkout.
  Fields:
  userId: string
  booking Type: 'venue' | 'event'
  bookingId: string
  amount: number
  currency: string

- **2. CheckoutSessionResponse**
  Returned From Backend:

  {
  sessionUrl: string; // Stripe-hosted checkout URL
  }

- **3. StripeWebhookPayload**
  Received by backend webhook handler Contains payment status, booking, and Stripe session metadata to update backend.

- **Component Error Handling:**

Scenario	StatusCode	Response Message
Missing userId or bookingId	400	"Missing required fields"
Invalid booking type	400	"Invalid booking type"
Stripe API error	500	"Stripe checkout creation failed"
Webhook payload missing or malformed	400	"Invalid Stripe webhook payload"
Payment not completed (e.g. canceled)	409	"Payment not completed"
DB update failure after payment	500	"Could not confirm booking"

➢ Venue Owner Event Management

- **Component Name:** Venue & Event Management (Venue Owner Panel)

- **Component Description:**

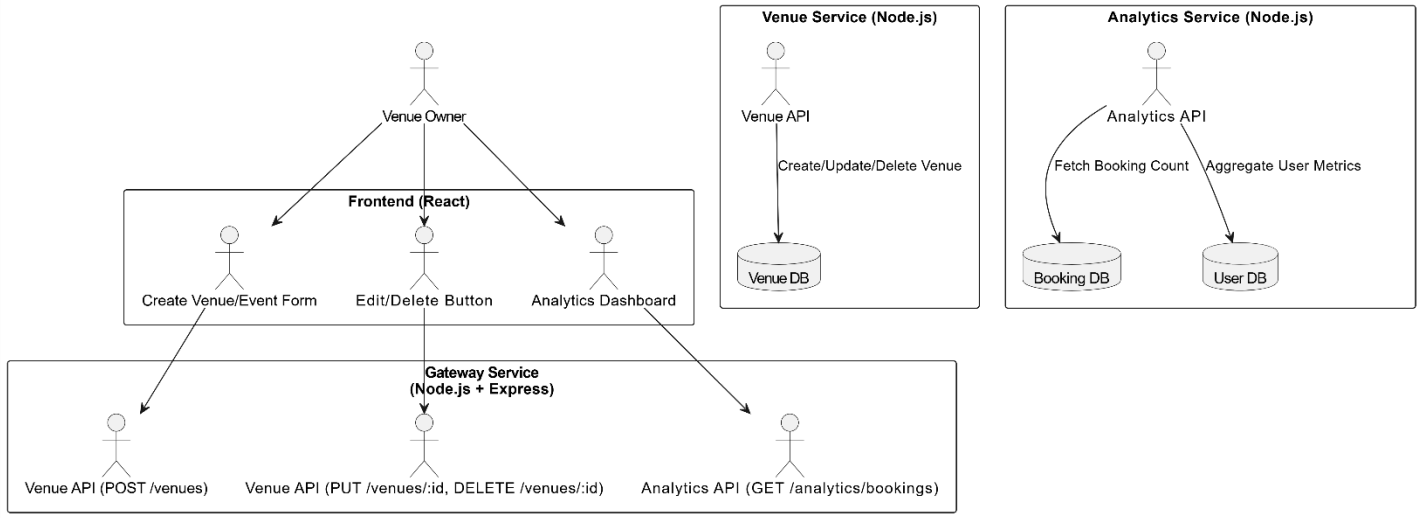
This component enables venue owners to create and manage their venues and events in a multi-step, guided workflow. It is designed to simplify data entry and ensure accurate event listing creation. The process is broken into clearly segmented stages—Venue Details, Image Upload, Sports & Pricing, and Review—enhancing user experience and data validation.

Each step handles specific functionality:

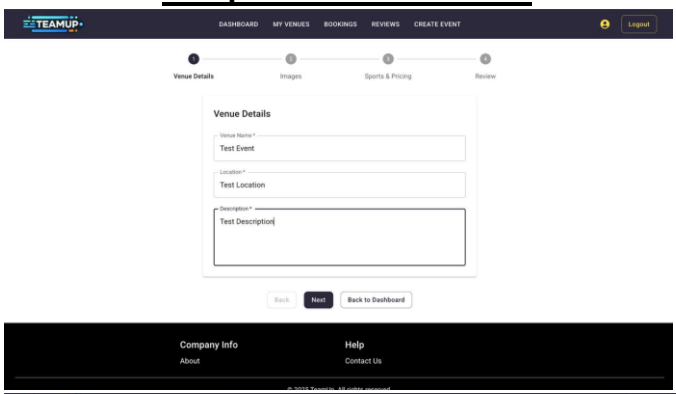
Step 1: Collects core venue/event metadata (name, location, description).
Step 2: Allows image upload and preview with validation.
Step 3: Enables adding sport types, court prices, and total court capacities.
Step 4: Displays a comprehensive review of all submitted details for confirmation.

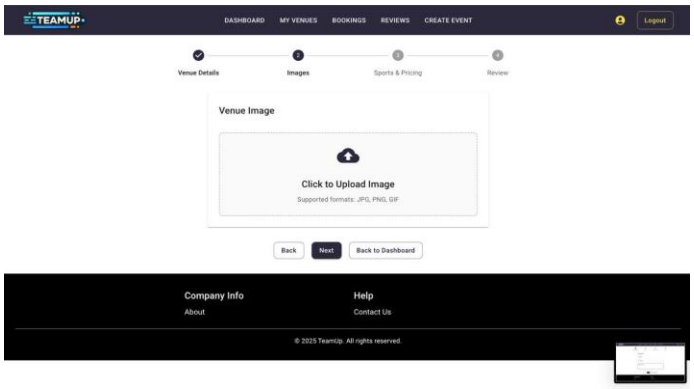
- **Responsible Development Team Member:**
  Tribhangin Dichpally
  Shanti Jangam

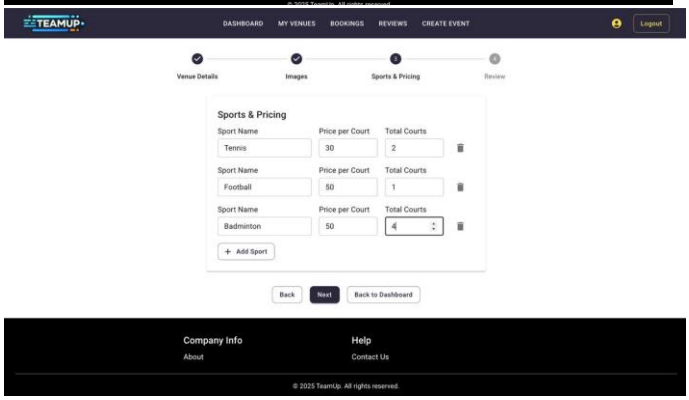
- **Component Diagram:**

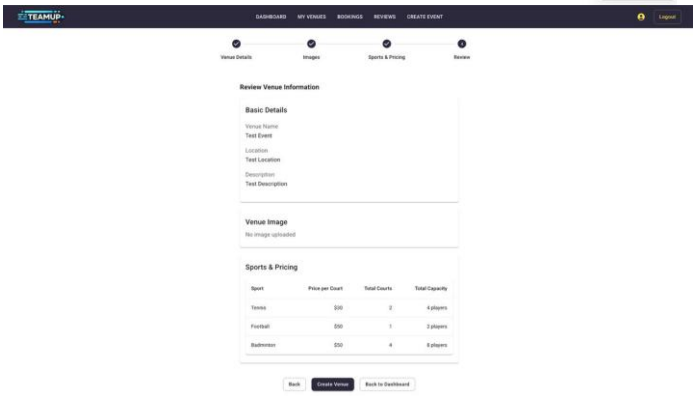


- **Component User Interface:**









- **Component Objects:**

1. VenueForm

Collects input for venue name, location, description, images, and pricing.
Used for both creation and editing of venues.

2. EventForm

Similar to VenueForm, but includes fields for event date, time, ticket price, and capacity.
Supports image upload and preview.

3. VenueServiceController

Handles incoming HTTP requests related to venue creation, update, and deletion.
Exposes endpoints like POST /venues, PUT /venues/:id, and DELETE /venues/:id.

4. EventServiceController

Manages API routes for event creation and editing.
Validates input and triggers image uploads and DB interactions.

5. VenueModel / EventModel

ORM or database models representing venue and event records.
Maps data to Venue DB and stores fields like name, description, images, sports & pricing.

6. AnalyticsDashboard

UI component for venue owners to view insights about their listings.
Displays metrics like number of users booked, booking trends, and revenue summaries.

7. AnalyticsController

Fetches statistics from the Booking DB and User DB.
Returns aggregated results to the frontend in real-time.

8. BookingDB (Database)
Stores all user bookings, linked to venue/event IDs and timestamps.
Used for generating analytics and booking history.
9. Supabase Storage / UploadService
Manages image uploads for venue and event media.
Returns URLs stored in the respective models.
10. AuthMiddleware
Ensures only authorized venue owners can perform CRUD operations on their listings.
Verifies JWT tokens before processing requests.

- **Component Interfaces:**

Frontend => Gateway Service

Endpoint	Method	Payload	Description
/venues	POST	{name, location, description, pricing[] }	Create a new venue entry. Requires authentication token.
/venues/:id	PUT	{updatedFields}	Update details for a specific venue.
/venues/:id	DELETE	–	Delete a venue. Authorization token required.
/events	POST	{name, date, price, location, capacity }	Create a new event with given parameters.
/events/:id	PUT	{ updated_Fields }	Update an existing event.
/events/:id	DELETE	–	Delete a specific event. Authorization token required.
/analytics/venue/:id	GET	–	Fetch metrics like number of bookings for the selected venue.

Gateway Service => Backend

Endpoint	Method	Description
POST /internal/venues	POST	Proxy request to create a venue.
PUT /internal/venues	PUT	Proxy request to update venue data.
DELETE /internal/venues	DELETE	Proxy request to delete a venue.
GET /internal/analytics/venue/:id	GET	Fetch analytics for dashboard.

- **<u>Component Error Handling:</u>**

Form Validation: Frontend prevents submission if required fields (name, location, pricing) are missing.

Image Upload Failures: Displays “Upload Failed” message and retry option if Supabase rejects the image.

401 Unauthorized: Redirects to login if JWT is invalid or expired.

403 Forbidden: Displays “Access Denied” when a user tries to edit/delete another owner’s venue.

500 Internal Server Error: Shows banner “Something went wrong. Please try again later.”

Empty States: If no venues or events exist, shows a friendly message with a “Create New” call to action.

Booking Analytics Failures: Displays fallback placeholder or loading spinner in case metrics fail to load.

➢ Additional Feature Services

- **Component Name:**

  Bookmark & Venue Location Display Service

- **Component Description:**

Bookmark Service: The Bookmark Service allows authenticated users to save and manage their favorite venues and events for quick access. This functionality enhances user personalization by letting users "bookmark" any venue or event they find interesting. On the frontend, a reusable Bookmark Icon component is embedded into event and venue cards, as well as detail pages. These icons toggle state based on user interaction. When clicked, the component makes a request to the backend to update the bookmark status. The backend handles these updates using a dedicated microservice that interacts with venue_bookmarks and event_bookmarks tables in Supabase. All operations pass through the API Gateway for centralized authentication and routing. The system supports adding, removing, and retrieving bookmarks per user, and handles separate bookmark types for events and venues.

Map Service: The Map Service provides dynamic geolocation and visualization functionality across the application, allowing users to see the physical location of venues and events on an interactive map. It is built using Leaflet.js and integrated into React using the react-leaflet library. The map component (MapComponent.tsx) is fully reusable and supports configurable coordinates, zoom levels, and custom markers. It is embedded in venue and event detail pages to show the exact location based on user-provided addresses. The service uses OpenStreetMap's Nominatim API to perform geocoding (address → coordinates) and reverse geocoding (coordinates → address). To ensure compliance with usage limits, rate limiting and caching mechanisms are implemented. In the case of invalid or missing location data, the map component gracefully displays a fallback message. Overall, the Map Service significantly improves the UX by providing spatial context for event and venue browsing.

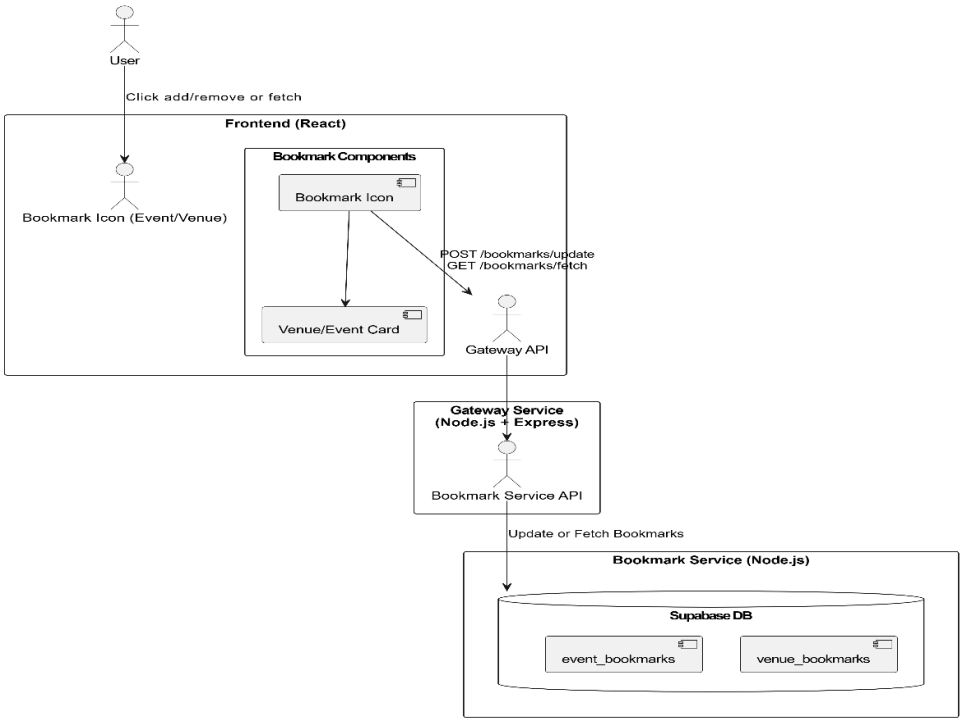
- **Responsible Development Team Member:**

  Bookmark -> Tushar Khatri

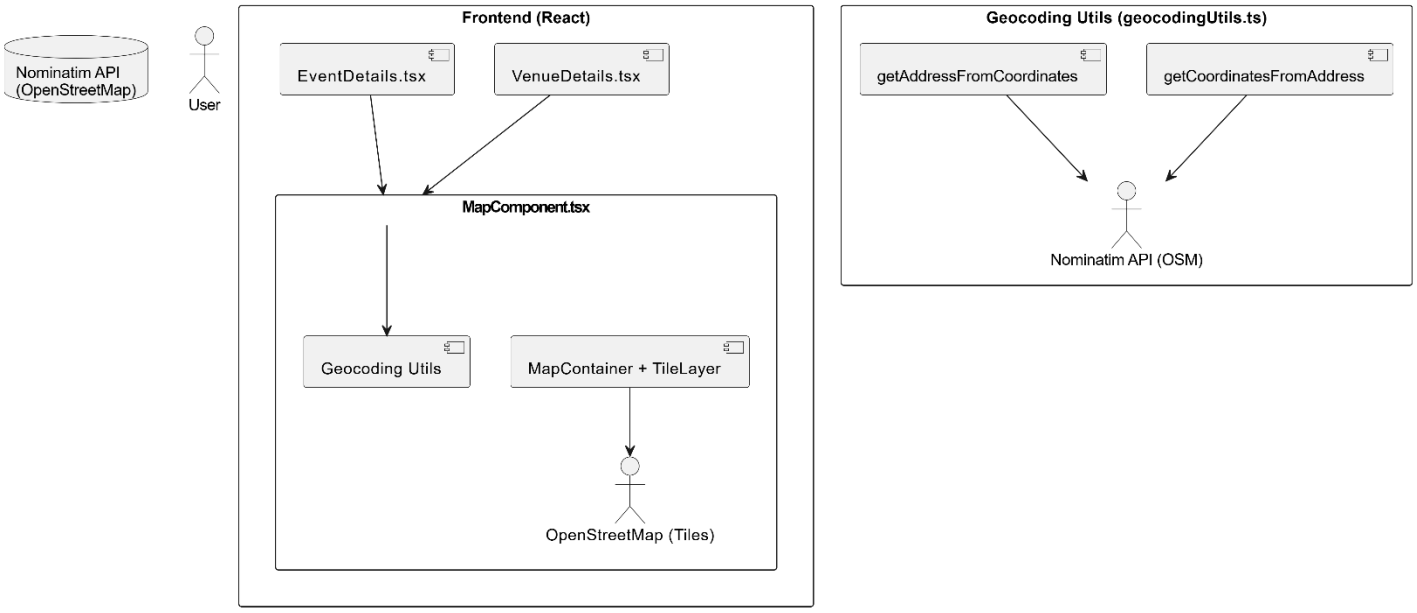
  Maps -> Shanti Jangam

Component Diagram:

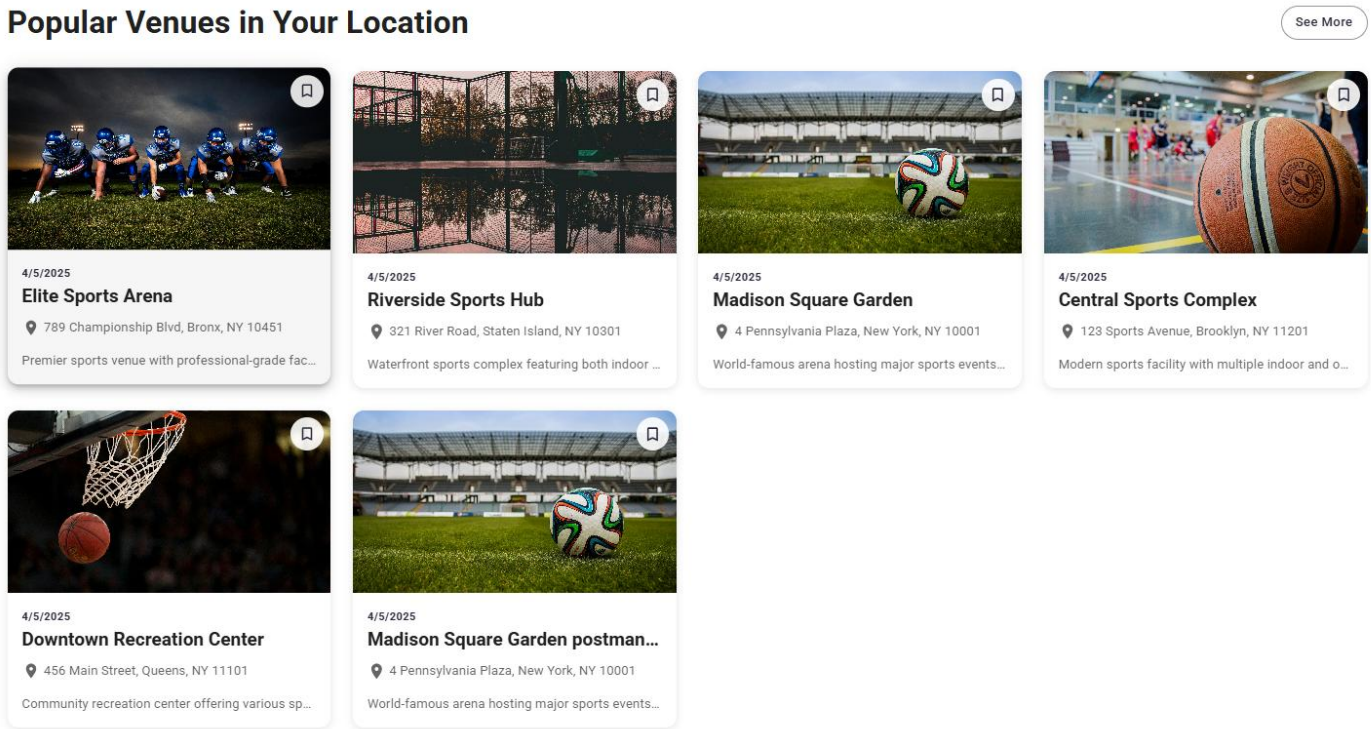
Bookmark Service:

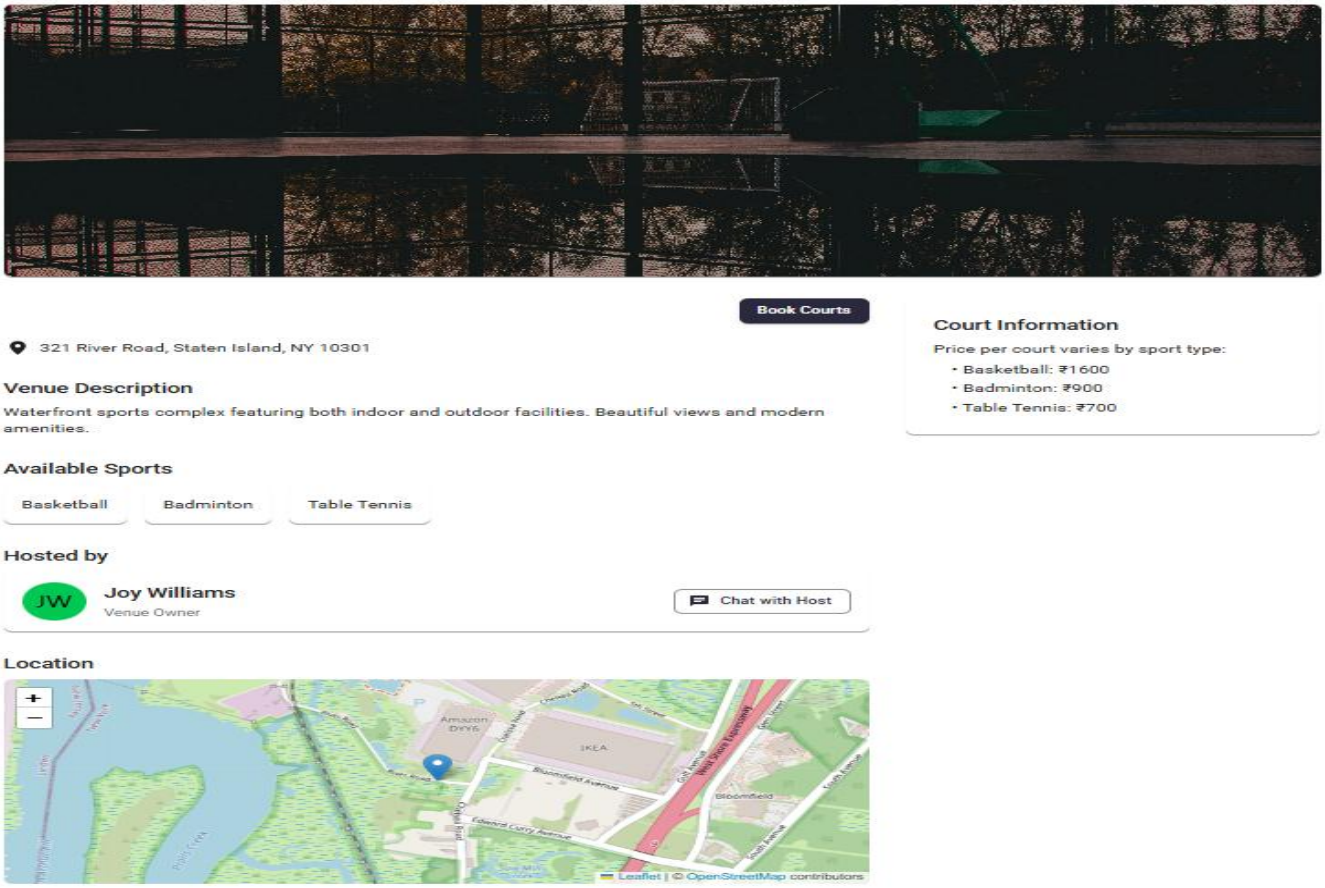


Map Service



- **<u>Component User Interface:</u>**





- **Component Objects:**

Bookmark ->
The Bookmark Service allows users to bookmark both events and venues for future access. This is implemented through multiple components. On the frontend, BookmarkButton.tsx is a small toggle icon that appears on venue and event cards and within their detail pages. It triggers backend updates when users add or remove bookmarks. The BookmarkList.tsx component is used to display all bookmarked venues or events for a user in a categorized format. On the backend, bookmark actions are handled by an Express-based microservice that communicates with Supabase tables: event_bookmarks and venue_bookmarks, both of which store references to the user ID and the respective item ID (event or venue). These services are routed through the API Gateway to enforce authentication and streamline communication.

Map Service ->
The Map Service consists primarily of two core components: the MapComponent.tsx file and the geocodingUtils.ts utility module. MapComponent.tsx is a reusable React component that leverages react-leaflet to render interactive OpenStreetMap views. It accepts configurable props such as center coordinates, zoom levels, dimensions, and marker positions, enabling it to be embedded across different parts of the application. The geocodingUtils.ts module provides asynchronous functions to convert physical addresses into geographic coordinates (getCoordinatesFromAddress) and vice versa (getAddressFromCoordinates) using OpenStreetMap's Nominatim service. These components are tightly integrated within VenueDetails.tsx and EventDetails.tsx, ensuring that maps are dynamically rendered as soon as location data becomes available.

- **Component Interfaces:**

Bookmark Service ->
The frontend interacts with the Gateway using two core endpoints. The POST /bookmarks/update endpoint accepts a payload containing the type (event or venue), id, and action (add or remove), enabling toggle behavior. The GET /bookmarks/fetch endpoint retrieves all bookmark entries associated with the currently logged-in user. Internally, the Gateway routes these requests to the Bookmark Service via Axios. The Bookmark Service processes these requests using /internal/bookmarks/update and /internal/bookmarks/fetch, which directly manipulate or retrieve entries from the venue_bookmarks and event_bookmarks tables. All requests are authenticated via JWT before execution.

Map Service ->
The map system exposes clearly defined TypeScript interfaces for interaction. The MapComponentProps interface defines the structure for map configuration, including center coordinates (lat, lng), optional zoom level, marker positions, height, and width. The utility functions follow standard asynchronous API signatures. getCoordinatesFromAddress(address: string) returns a Promise that resolves to latitude and longitude, while getAddressFromCoordinates(lat, lng) returns a reverse-geocoded address string. These interfaces encapsulate the abstraction needed to fetch and display dynamic map data while keeping implementation loosely coupled.

- **Component Error Handling:**

Bookmark Service -> The Bookmark Service includes user-friendly error handling at every level of the stack. On the client side, bookmark actions are blocked for unauthenticated users and redirect to the login page. Optimistic UI feedback is provided via toasts/snackbars indicating bookmark success or failure. On the backend, all requests are wrapped in try-catch blocks. Duplicate inserts are prevented through Supabase constraints and conflict resolution. If the fetch request returns an empty result set, a fallback message such as "No bookmarks yet" is displayed. Network issues or service failures are surfaced to the frontend through standardized error responses.

Map Service -> The map system includes robust error handling to account for geocoding failures, invalid inputs, and API rate limits. If the geocoding utility fails to return valid coordinates, the component displays a fallback message such as "Map Unavailable" or suppresses the map view entirely. A 1-second delay is enforced between Nominatim API requests to comply with their rate-limiting policy. Component state caching prevents repeated API calls on re-renders, and placeholder markers are used in case location coordinates are malformed or undefined. User feedback includes loading indicators while fetching location data and gracefully degrades functionality when geolocation fails.

4. Revision History

Revision	Date	Description
Initial Planning & Setup	**03/01/2025**	**Finalized database schema, designed UI/UX in Figma, and configured environment setup.**
Authentication & Microservices Design	**03/22/2025**	**Completed authentication (JWT, Duo), set up microservices structure with Gateway API.**
Core Feature Development (Frontend & Backend)	**04/05/2025**	**Developed user bookings, venue/event creation, Stripe integration, and email support.**
Final Touches: Feature Enhancements & Deployment	**4/18/2025**	**Added bookmarks, map view, analytics; completed integration and deployment to cloud.**