



Calendar System

Design Report

Cenk Dogruer - s2875144

Zsombor Ivanyi - s2809265

Muhammet Beyoglu - s2827255

Kazi Rifat Hasan - s28727733

Omar Alaaeldin Badreldin Moustafa - s2879263

Cosmin Ana - s2628503

Table of Contents

Table of Contents.....	1
1- Introduction	4
2- User Requirements	5
2.1 User Stories (including MoSCoW)	5
2.1.1 Schedule Engine.....	6
2.1.2 Customer Event Sign-up.....	6
2.1.3 Public Booking Pages	7
2.1.4 Secondary Calendar Functions	7
2.2 Non-functional requirements.....	8
2.2.1. Functional Suitability.....	8
2.2.2. Flexibility	8
2.2.3. Usability.....	8
2.2.4. Reliability	8
2.2.5. Security	9
2.2.6. Maintainability	9
2.2.7. Portability	9
2.3 Communication	10
2.4 Risk Assessment	11
3- Design	12
3.1 Use Case	12
3.2 Architectural Design.....	13
3.2.1 Lower-Level Design	15
3.3 Database Design	19
3.3.1 Initial Design.....	19
3.3.2 Final Design	21
3.3.3 Summary	24
4- Planning.....	25
4.1 Project Phases and Timeline.....	25
4.1.1 Week 1 - Onboarding & Setup	25

4.1.2 Week 2 - Requirements & Initial Design	25
4.1.3 Week (3-4) - System & Database Design	25
4.1.4 Week (5-8) - Implementation	26
4.1.5 Week (5-9) - Testing & Integration	26
4.1.6 Week 10 - Finalization	26
4.2 Tools and Coordination	26
4.3 Timeline Overview (Gantt-style table)	27
5- Implementation	28
5.1 Technology Stack	28
5.1.1 Calendar-specific Libraries	28
5.1.2 External APIs	29
5.2 Mock-up	29
5.3 Firebase (Backend)	31
5.4 Calendar Page (Frontend)	32
5.4.1 React's Component-Based Approach	32
5.4.2 State Management	33
5.4.3 Type Checking with Typescript	35
5.4.4 Component Hierarchy	36
5.4.5 Components Explored in Detail	37
5.5 Limitations	42
5.5.1 Conditional fetching	42
5.5.2 Other limitations	43
6- Security Design	44
6.1 User Authentication	44
6.2 User Input Sanitization	44
6.3 Database permissions	44
6.3.1 Employee Permissions:	44
6.3.2 Customer Permissions:	44
7- Maintainability	45
8- Testing	48
8.1 Test Cases	48
8.1.1 Test Case 1	48
8.1.2 Test Case 2	50

8.1.3 Test Case 3	51
8.1.4 Test Case 4	54
8.1.5 Test Case 5	56
8.1.6 Test Case 6	57
8.1.7 Test Case 7	59
8.2 Summary of Test Cases	61
9- Evaluation	61
9.1- Individual Contribution	62
9.2- Evaluation of User Stories	63
9.2.1 User Stories Met	63
9.2.2 User Stories Removed	64
9.2.3 User Stories Not Met	65
10- Reflection	66
10.1 Challenges	66
10.1.1 Working with an Existing Codebase	66
10.1.2 Ambiguity in Type Specifications and Component Communication	66
10.1.3 Iterative Changes in Data Fetching Logic	66
10.1.4 Frequent Changes in UI Requirements	66
10.1.5 Concurrency and Race Condition Handling	67
10.2 Future Work	67
11- References	68

1- Introduction

NullSpace is a software development company which provides real-time business management for coaching businesses, combining booking, communication, and customer management into a single integrated system. The product has not yet been released onto the market, and it is still under development. It is managed by two co-founders: Jasper van Amerongen and Aleksandra Ignatovic. Since it is a start-up company without any customer database currently, there is no factual relevant data from customers, but only information about potential customers that the start-up aims to attract.

The central and significant feature of this platform is the calendar system which drives bookings, customer engagement, staff planning, and revenue. Although there are already made calendar systems that companies use (i.e. Google Calendar, Apple Calendar), NullSpace aims to integrate company meetings, customer sign-ups, payment models, queueing, and public booking pages to this calendar. Even though there is a basic calendar in NullSpace currently, it lacks the ability to handle complex scheduling logic.

The main concern of not using other calendar systems is that the calendar system that is to be implemented inside the project involves specific features for the company (e.g., queueing, queue time, allowing customer groups, specific locations, kicking / waitlisting someone for the event...). Therefore, it is necessary to have the implementation of a calendar system which already has the features of the most used calendar systems, such as Google Calendar and Apple Calendar, and to add extra functionality and specialize for the company's needs.

Developing such a custom calendar system also ensures full control over data management, security, and scalability. Since NullSpace handles sensitive customers, product, and business information, relying on third-party calendar integrations could introduce data breach risks and limitations in customization. By building the system in-house, the company can design the architecture to support real-time synchronization, automated conflict detection, and adaptive scheduling based on staff availability and customer demand. This approach not only enhances reliability and performance but also allows future extensions, such as analytics dashboards, AI-driven booking recommendations, and integration with the company's payment and communication modules.

2- User Requirements

To implement such a calendar system for the company's needs, the company has opened 4 different epic requirements - which also have many sub-requirements - for our project structure. These epics mainly focus on the improvements and specifications for the project structure.

According to their priority and relevance, the epics are in respective order:

- **Schedule Engine** – the backbone for event creation, editing, assignments, and recurrence handling.
- **Customer Event Sign-up** – ensures smooth and transparent participation flows for customers.
- **Public Booking Pages** – enables external booking functionality through secure widgets.
- **Secondary Calendar Functions** – enriches the scheduling system with supporting features (i.e. queues, opening hours, and group management)

These projects will work together to replace the legacy calendar system and operate closely with other parts of the platform, such as payments, customer information, and public booking workflows. The result will be a scheduling system that is more flexible, scalable, and easy to use, and that meets the demands of coaching firms and their clients.

However, not all epics hold the same level of significance. The **Schedule Engine** and **Customer Event Sign-up** are the core components of this project since they form the foundation of the scheduling system's functionality. The **Schedule Engine** has the highest priority due to its central role in managing and coordinating the scheduling logic, then the **Customer Event Sign-up** follows in significance, supporting user interaction and event participation. Finally, the **Public Booking Pages** and **Secondary Calendar Functions** are considered lower in priority, serving as complementary features that enhance user experience and accessibility.

2.1 User Stories (including MoSCoW)

The company has shared many user stories to showcase the requirements that need to be done for the project itself. Therefore, the main structure of the project revolves around these 4 epics and their user stories. The significance of these user stories is stated in MoSCoW principle.

2.1.1 Schedule Engine

This epic mainly focuses on the schedule engine and how the scheduling is handled on the website and in the Firestore (where the data is stored). Therefore, this epic and sub requirements of this epic is the backbone of this whole project. Hence, the requirements under this epic are mostly obligatory for us to have.

- Admin users can plan and edit recurring events. **(Must have)**
- Admin users can plan and edit non-recurring events. **(Must have)**
- Admin users can assign or change hosts of an event. **(Must have)**
- Admin users can specify a room for an event. **(Must have)**
- Admin users can enable a queue for an event. **(Should have)**
- Admin users can tag categories for events. **(Should have)**
- Admin users can change customer groups for events. **(Must have)**
- Admin users can select the payment model for an event. **(Could have)**
- Admin users can sign up individual customers for an event. **(Should have)**
- Admin users can sign up customer groups for an event. **(Should have)**
- Admin users can make an event private. **(Must have)**
- Admin users can create recurrent events. **(Must have)**
- Customers can see available events in the calendar. **(Must have)**
- Customers can see the events they joined. **(Must have)**
- Admin users can see specific events associated with specific admins. **(Must have)**
- Admin users can see all events in the calendar. **(Must have)**
- Admin users can see occupancy of all locations. **(Should have)**
- Admin users can see occupancy of other admins. **(Should have)**
- All users can see basic information of an event. **(Must have)**
- All users can see more detailed information about an event. **(Must have)**
- All users can differentiate events by color. **(Should have)**

2.1.2 Customer Event Sign-up

This epic mainly focuses on the customer to be able to sign up. It is the second most important part of the project structure for users to independently sign up and join such events.

- Customers to see queue status. **(Should have)**

- Customers to see if they joined the queue. **(Should have)**
- Customer to see host's picture. **(Could have)**

2.1.3 Public Booking Pages

This epic mainly focuses on the customer being able to see public booking pages of admins and to request an event based on the admin's timeslot. This epic is not priority but offers room to extend the project structure if there is more free time left for implementation.

- Admin users to set the public booking page constraints. **(Could have)**
- Customers to book a timeslot with an admin who created a public booking page. **(Could have)**

2.1.4 Secondary Calendar Functions

This epic involves secondary operations which are not vital for the core functionality but a room to work on for the calendar system. The functionalities inside this epic cannot be generalized but they are self-explanatory.

- System to update the queue. **(Could have)**
- System to notify the customer of their queue movement. **(Could have)**
- System to notify the customers when they are running up the queue. **(Could have)**
- Admin users to set freeze time of the queue. **(Could have)**
- Admin users to set/edit opening hours of a location. **(Could have)**
- Rooms within the location to inherit opening hours. **(Could have)**
- Admin users to create/edit a room. **(Could have)**
- Admin users to mark a location unavailable. **(Could have)**
- Rooms automatically become unavailable. **(Could have)**
- Admin users to create and edit customer groups. **(Could have)**

2.2 Non-functional requirements

The non-functional requirements for the NullSpace Calendar System are defined in alignment with the ISO/IEC 25010 software quality standards. These requirements ensure that the calendar system is reliable, efficient, secure, and user-friendly, while remaining maintainable and compatible within the broader NullSpace ecosystem.

2.2.1. Functional Suitability

- **FR-1:** The system shall provide complete and accurate scheduling functionalities consistent with defined user requirements, including event creation, editing, and recurrence handling.
- **NFR-2:** The system shall ensure correctness of displayed event information, including time zones, locations, and participant data.

2.2.2. Flexibility

- **NFR-3:** The calendar system shall function seamlessly across all modern browsers (Chrome, Safari, Edge, Firefox) and mobile devices.

2.2.3. Usability

- **NFR-4:** The interface shall follow a familiar layout similar to popular calendar tools (e.g., Google Calendar, Apple Calendar) to reduce the learning curve.
- **NFR-5:** The application shall maintain consistent design patterns aligned with NullSpace's UI/UX guidelines and accessibility standards (WCAG 2.1 Level AA).

2.2.4. Reliability

- **NFR-6:** Scheduled events shall be backed up automatically at least once every 24 hours.
- **NFR-7: Concurrency Control on Event Sign-up**
The system shall implement concurrency-safe mechanisms during event sign-up operations to prevent race conditions. When two or more customers attempt to sign up for the same event simultaneously and only one spot is available, only one customer shall be successfully registered, and the remaining customer(s) shall be automatically added to the waitlist.

2.2.5. Security

- **NFR-8:** Only authenticated users shall be able to view or modify events according to their role (admin or customer).
- **NFR-9:** Sensitive data (e.g., customer information) shall not be stored in the client-side cache.

2.2.6. Maintainability

- **NFR-10:** The source code shall follow the company's TypeScript/React coding standards and include inline documentation.
- **NFR-11:** The system shall maintain modular architecture to enable independent updates of calendar components.

2.2.7. Portability

- **NFR-12:** The system shall be deployable across different environments (development, staging, production) with minimal configuration changes.
- **NFR-13:** The application shall be platform-independent and run on all major operating systems (Windows, macOS, Linux).

2.3 Communication

The communication is established through WhatsApp and Discord channels that are specifically created for the NullSpace development team including the founder of the company (Jasper van Amerongen). Progress is evaluated and discussed weekly by scrum meetings every Monday and Wednesday between 10:00-18:00 at Incubase.

Git Branches: For the separation of work and not having merge conflicts, several branches are made to have a clean working structure. They are derived from the epic branch and every time a branch is complete and fully functional, it is merged into the epic branch. The so-called branches are:

Schedule-Engine: The epic branch to be merged onto at the end

Create-Event-Fullscreen: The branch which involves the creation, editing, and viewing of the event in fullscreen view.

User-Event-Signup: The branch made for user event signups (self-explanatory).

Event-Card-Popup: The branch made for working on the event card pop-up on the calendar when a specific event is clicked.

Security-Permissions: The branch used to develop optimized and secure conditional fetching.

2.4 Risk Assessment

We also looked at possible risks that could slow down or cause problems in our project. For each risk, we considered ways to reduce the chance of it happening or to handle it if it did happen.

Risk	Likelihood	Impact	Mitigation
Technical issues with Firebase	High	Medium	Consult NullSpace dev team, research alternatives early.
Delays in backend–frontend integration	Medium	High	Early mock API testing, iterative integration.
Team communication gaps	Low	Medium	Bi-weekly office meetings, daily WhatsApp/Discord updates.
Time pressure near deadline	Medium	High	Continuous report writing (every week) to avoid backlog.

Figure 1: Risk assessment table.

3- Design

The design section consists of use cases and structure of the calendar system, low-level designs, and database design. Since it makes up the general structure of the project, this section is the most significant prior to the implementation.

3.1 Use Case

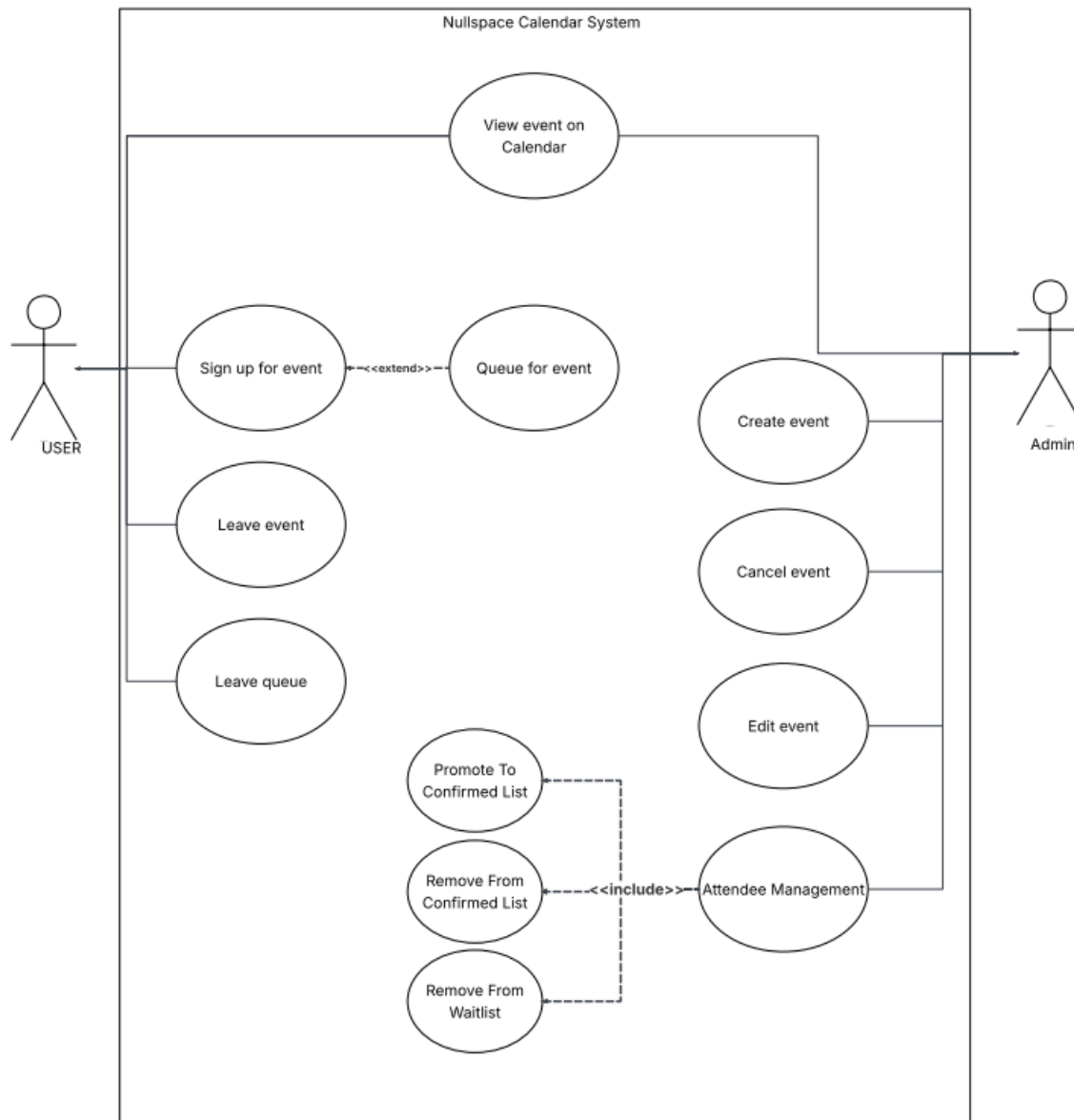


Figure 2: Use case diagram of the main features.

This use case diagram shows all features of the Nullspace Calendar System, both for users and for admins. In the system, users can view events on the main calendar page. Users can also sign up for events, or if the events are already full and a queue is enabled, users can then queue for the events. Finally, users may also leave the event or queue. Admins on the other hand have a more administrative role. They can also view the events, but within the events themselves admins are able to edit the data (start-end time, recurrence, location data etc.) and cancel the events, either for one event, upcoming events, or for the entire event series. Admins are also able to create events through the event creation button. Attendee management is also done through the event editing, where admins can promote people from the queue to the event, remove people from the queue, or remove people from the event.

3.2 Architectural Design

The Nullspace Calendar web application is structured around a set of modular components that are integrated throughout the codebase. Each component encapsulates a distinct feature or service, ensuring maintainability and separation of concerns. These components are referred to as contexts within the system architecture. The following diagram illustrates the various contexts utilized across the application.

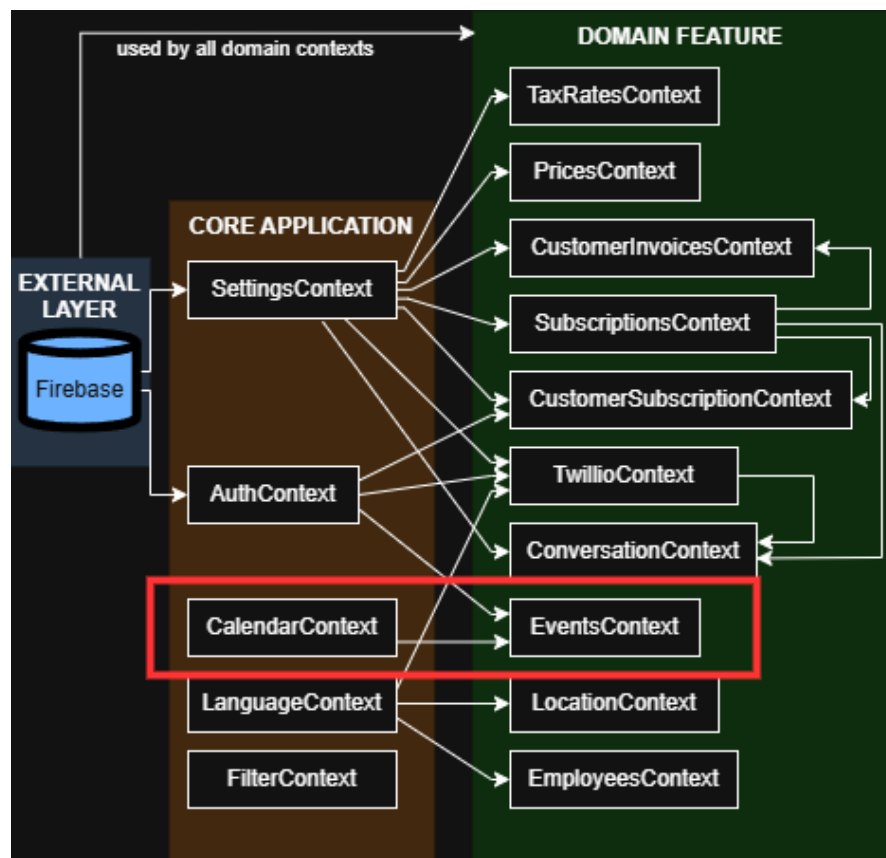


Figure 3: Component diagram showcasing the components and their relations.

The external layer in this case is Firebase, which is providing data to all of the components, except calendar, language and filter contexts. Some components are accessing firebase directly, some using other contexts to get data from the database.

Here is an explanation of what each context does:

- **Settings Context:** Manages application settings including server settings, banners, entitlements, and FAQs.
- **Language Context:** Manages language selection and translations across the app.
- **Tax Rates Context:** Manages tax rate data fetched from Stripe for commerce purposes.
- **Prices Context:** Manages Stripe price and product data for commerce features.
- **Customer Invoices Context:** Manages customer invoice data from Stripe, providing invoice retrieval and update capabilities.
- **Subscriptions Context:** Manages Stripe subscription data for users, including fetching and updating subscription status.
- **Customer Subscription Context:** Provides customer subscription details fetched from Stripe for authenticated users.
- **Twilio Context:** Provides Twilio client instances for messaging and conversations.
- **Conversations Context:** Manages customer support conversation data using Twilio Conversations API.
- **Locations Context:** Manages location data including location names and addresses, providing access to location information throughout the app.
- **Employees Context:** Manages employee data including roles and contact information.
- **Filter Context:** Provides filtering functionality for collections of data, managing filter states with URL integration. Was not used by any other context in the end.

The component our team had to work on was mainly the events context. This component includes provider scripts, which can be used anywhere in the project, just as any other context.

- **Calendar context** is responsible for the calendar object in the schedule engine page. Manages calendar date, scope, and navigation functionalities.
- **Events context** is responsible for event fetching and pushing. Data types and structure are specified here too. Manage event-related data including events, occurrences, and user signups, with role-based data access. Explained in more detail in section [5.3.2](#).
- **Auth context** was not modified by our team, however, it is responsible for user authentication and has been effectively utilized throughout the application to determine the role and the id of the user.

3.2.1 Lower-Level Design

Below are several diagrams to explain the page interactions that we have implemented in the project. These diagrams are made for a better understanding and an overview of the pages. They include user/admin interaction with event popover, event inspection, editing and deletion in full screen, and admin attendee management.

3.2.1.1 User Interacting with Event Popover

This activity diagram illustrates the interactions between the user, system, and the Firestore database when the user interacts with an event popover in the application. It covers viewing event details, signing up for an event, leaving an event, and viewing the full event page. The process starts when the user chooses and clicks on an event from the calendar, which then the system fetches event data from Firestore, which Firestore retrieves the data and returns it to the system. The system displays a popover with basic information and an option to view full event details. The user has the option to sign up or leave the event if signed up, depending on the choice and a confirmation message is displayed.

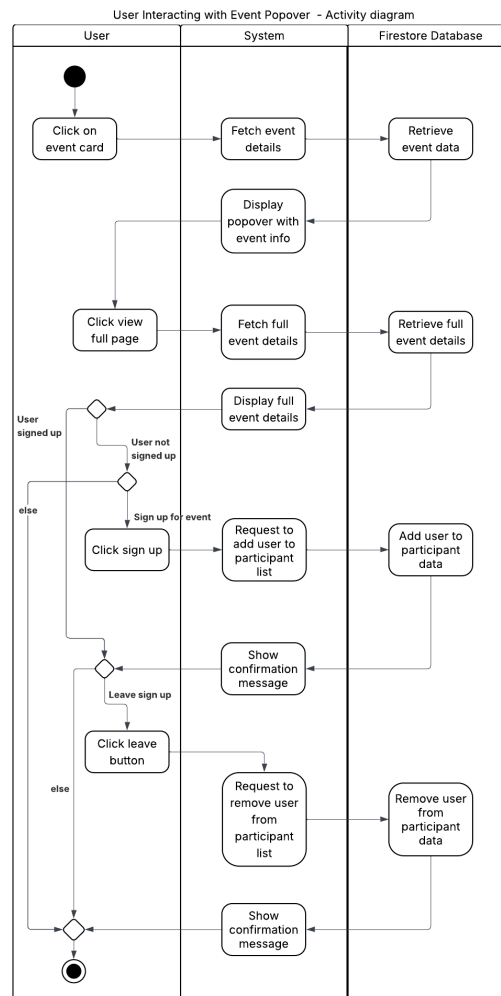


Figure 4: Activity diagram showcasing user interaction.

3.2.1.2 Admin Interacting with Event Popover

This activity diagram illustrates the interactions between the admin and the system to view and manage event details stored in the Firestore database. When the admin chooses and clicks on an event from the calendar, it prompts the system to fetch event details from the database and display them in a popover. The admin has the option to view the full event page, edit or cancel event(s). Depending on the choice of editing, the admin updates the relevant event fields, and the system validates the input before sending the changes back to Firestore. Once the update is successful, a confirmation message is displayed.

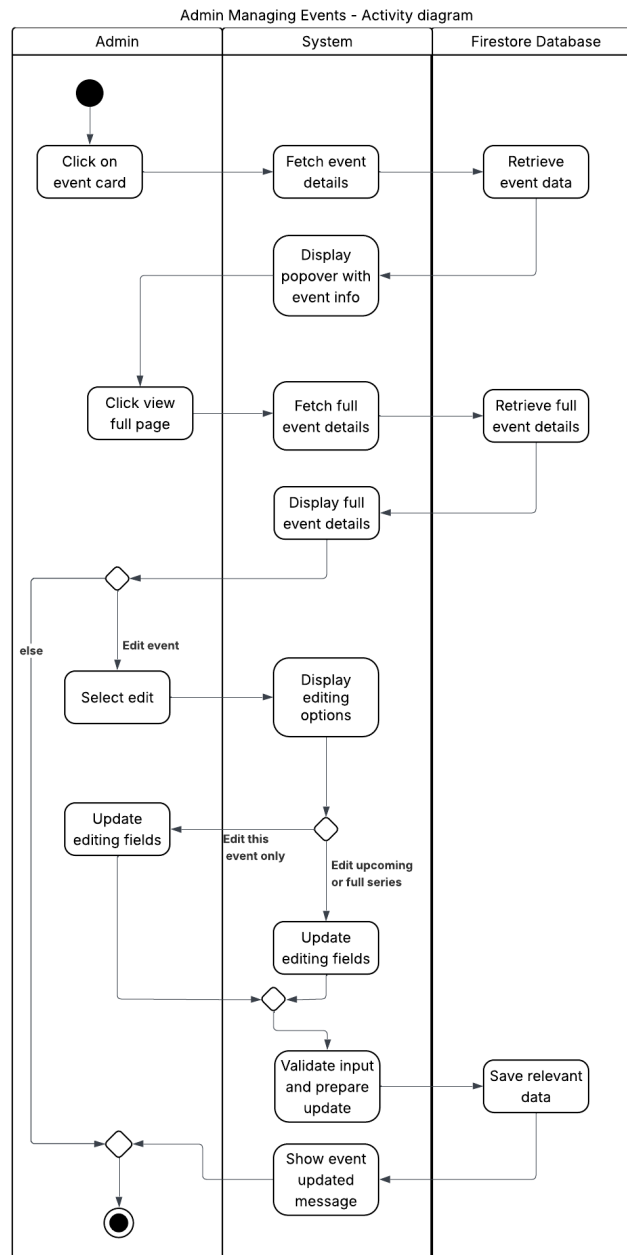


Figure 5: Activity diagram showcasing admin interaction.

3.2.1.3 Event Inspection, Creation, and Deletion in Full-screen view

This state machine diagram illustrates how the admin interacts with the full-screen view of the events. First, the admin presses the new event button to create it. Then, after filling in the fields, the event is published. If the admin thinks it is done, then the state comes to an end. Otherwise, the admin can inspect the event anytime. After the admin inspects the event, the admin can decide to either change the event or be done with it. If an admin decides to edit the page, he/she can either discard the changes or apply the changes. If the admin decides to discard the changes, it goes back to the inspection view. If the admin decides to apply the changes, it goes to the published state once again.

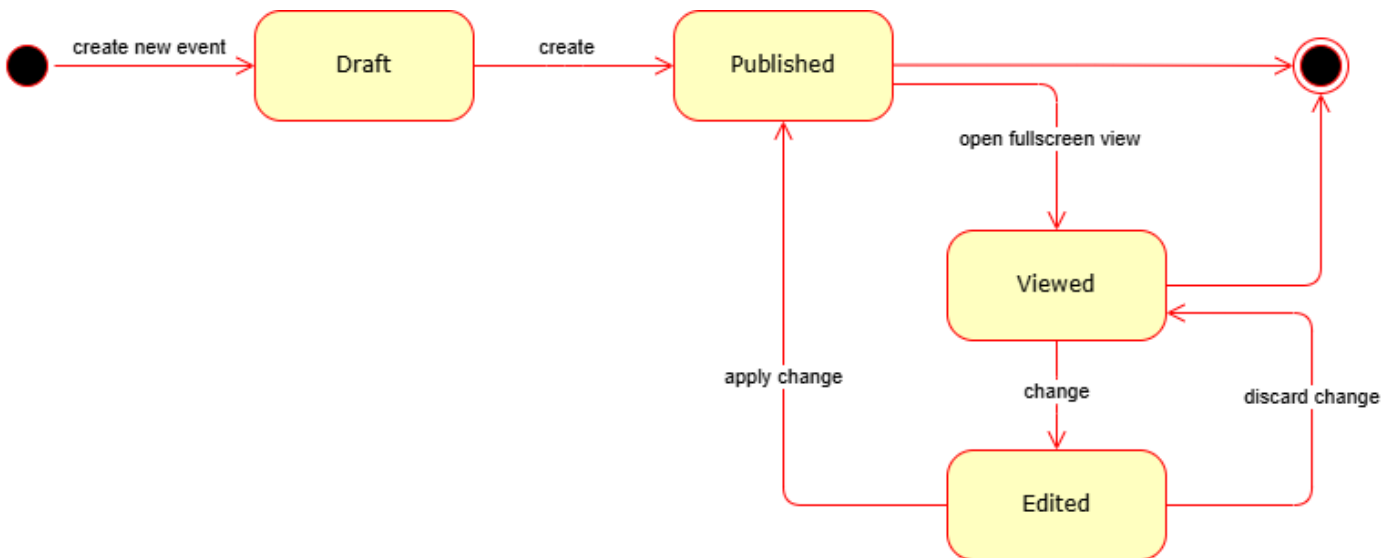


Figure 6: State machine diagram showcasing event interaction

3.2.1.4 Admin Attendee Management

This activity diagram shows the process of attendee management by an admin. Once an admin is on the main calendar page and clicks on an event card, they can click on the button to open the event in full screen, or they can press the edit button, and will be brought to the event details editing page. On this page, at the very bottom, they can see an Attendee List for the event, separated by two sections, Confirmed Attendees and Waitlist. If an attendee is present on the waitlist, the admin is able to promote them from the waitlist to the confirmed list or remove them from the queue altogether. On the Confirmed Attendees list, the admin is also able to remove an attendee.

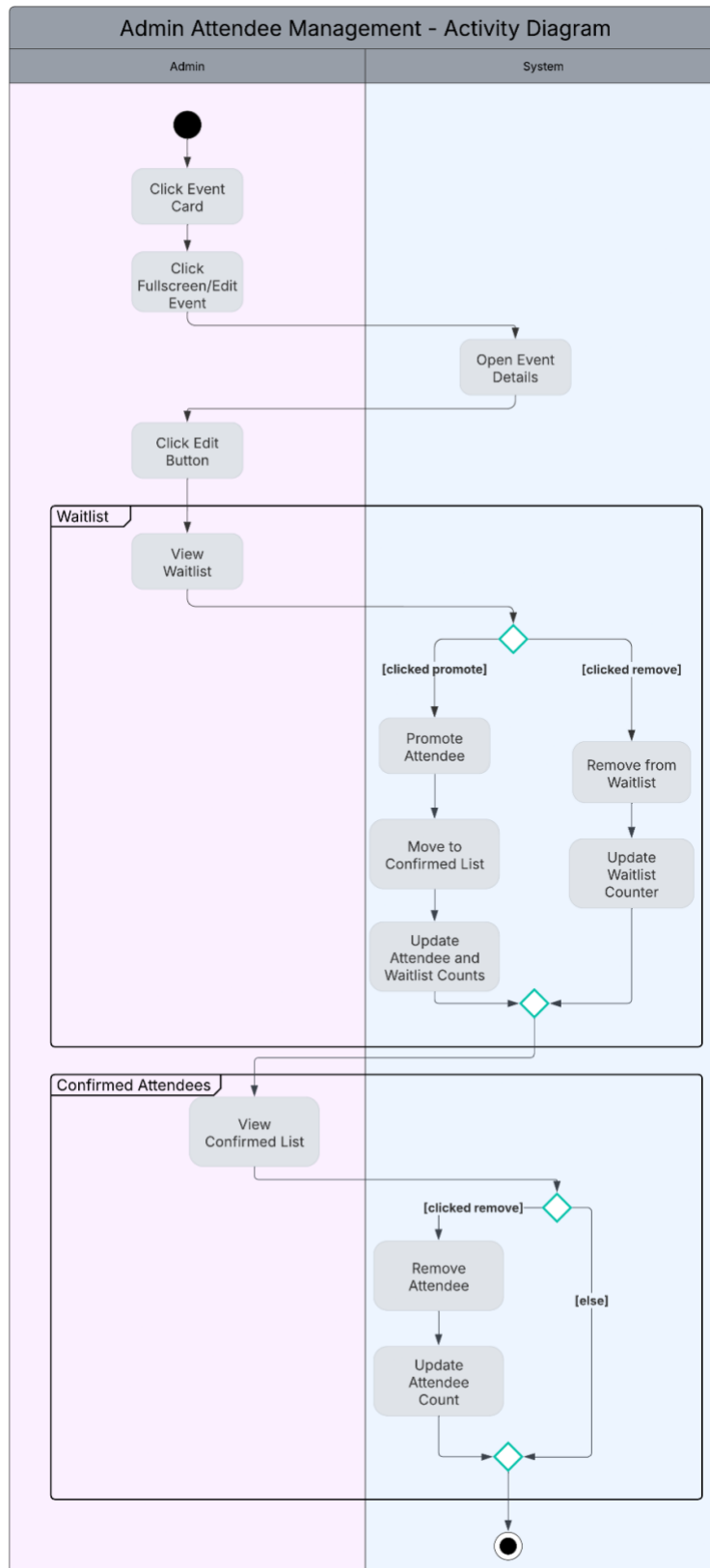


Figure 7: Activity diagram showcasing attendee management.

3.3 Database Design

This section involves the contents for the initial and final version of the database design of the calendar system.

The database design was the most vital part of our project structure since it involves how an event was created, updated, and deleted in a database. It involves how the attendee is queued, joined, and cancelled to participate in the event. It also involves how the host sets up the parameters for the event including: title, description, start date, end date, color (to distinguish the event on the calendar), location, visibility, recurrent events, queue, and the capacity of the event. To explain some contexts in detail (since some of these contexts are self-explanatory e.g. title, description...), definition of these parameters is given below:

Visibility: An event can be either public or private. If it is public, the event can be seen by every user. However, if it is set to private, only **allowed users** and **allowed groups** can see this event.

Recurrent Event: An event can be set to occur more than once. So, it can be set to recur on a specific day of the month, specific day of the week, once in a given number of days/weeks/months/years depending on the user's choice.

Capacity: An event can have a limited capacity of attendees, and this number can be set by the host.

Queue: An event can have queue enabled, where, if it has a limited capacity and some attendees decide not to go to the event, those in the waiting list can join the event in order. The queue limit can also be set to a specific number.

3.3.1 Initial Design

The first version of our database was made to keep the main information of each event. It was simple and focused only on creating and showing events without any advanced features like recurrence, queues, or capacity limits. The goal was to make a working system that allowed events to be created, stored, and shown on the calendar.

In this version, we had four main collections in Firestore: events, employees, locations and customerGroups.

Events: This collection stored all data related to an event, such as the title, description, start time, end time, and visibility (public or private). Each document was one single event. There were no subcollections inside each event, which means no separate occurrences or signups yet.

Employees: This collection stored the data of the hosts or organizers of events, such as their name, role, and address information.

Locations: This collection included information about where events take place. Each document had details like the city, country, and location name.

CustomerGroups: This collection was added to organize users into groups. For example, a group could be “Students” or “Trainers.” These groups helped to control which users could see private events.

Each event document had the following fields:

- **title** – the name of the event
- **description** – a short description about the event
- **startAt and endAt** – when the event starts and ends
- **visibility** – to choose if the event is public or private
- **createdBy** – the ID of the user who made the event
- **color** – used to visually mark the event on the calendar
- **createdAt and updatedAt** – timestamps for tracking changes

At this stage, every event was created separately. If the host wanted to make the same event happen every week, they had to create it again manually for each date.

Initial Database Design

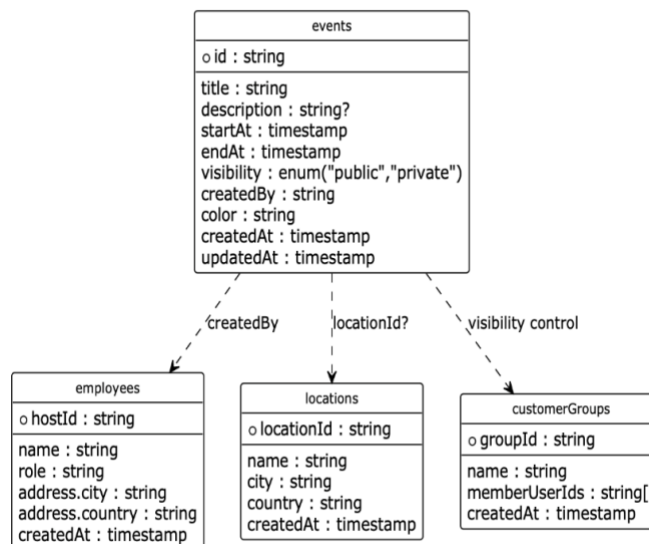


Figure 8: Initial database design, designed by the client.

Limitations of the Initial Design:

- **No Recurring Events:**
 - Events could not repeat automatically. Every event had to be created one by one.
- **No Queue or Capacity:**
 - There was no way to limit how many people could join an event or manage a waiting list.
- **No Subcollections:**
 - Signups or attendance were not stored under events, so we could not easily track who joined each event.
- **No Status Tracking:**
 - Events could not be marked as cancelled, closed, or completed.
- **Limited Links Between Collections:**
 - Events were not directly linked to rooms or hosts, so it was hard to know which host managed which event.
- **Security:**
 - All events, occurrences and signups were fetched for any user without constraints.

3.3.2 Final Design

As the project developed, it became clear that the calendar system required a more comprehensive and flexible data structure capable of handling complex event management scenarios. The final database design therefore extends the initial version by introducing a more normalized schema with stronger relationships between entities, improved scalability, and advanced features such as recurrence, queueing, and capacity management.

In this final version, the design revolves around the Event entity as the central element, from which other entities like EventInstance, Occurrence, Signup, and Room are derived or connected. The structure ensures that both administrative and customer operations such as scheduling, sign-ups, and queueing handling can be efficiently managed while maintaining clear data relationships.

3.3.2.1 Core Entities:

- **Event:**

The core entity represents an abstract event definition. It contains general information such as **title, description, duration, visibility, capacity, and recurrence rules (rrule)**. An event may also specify associated **hosts (hostIds)**, **assigned room (roomId)**, and applicable booking policies. Each event can be public or private, with visibility controlled through **allowedUserIds** and **allowedGroupIds**.

- **EventInstance:**

This entity represents a specific realization of an event at a given time. Instances are derived either from recurrence rules or manually created one-time events. They inherit core properties from their parent **Event** but store unique attributes such as actual start and end times, assigned hosts, and location information. This structure enables the system to support both recurring and one-off events without duplicating base data.

- **Occurrence**

Each occurrence corresponds to a single scheduled session of an event, generated based on its recurrence configuration. Occurrences contain fields such as **startAt**, **endAt**, and **color**, and reference the corresponding **eventId** and **locationId**. They serve as the bridge between event definitions and user sign-ups.

- **Signup**

Represents the relationship between a user and a specific occurrence. It includes the **status** (e.g., **CONFIRMED**, **CANCELED**, **QUEUED**), timestamps, and a link to both the user and occurrence. This structure supports queue management, allowing users to automatically join the waiting list when an event reaches its capacity.

- **Room & BookingPolicy:**

Rooms provide physical or virtual spaces where events occur. Each room has an associated **BookingPolicy** that defines constraints such as time, maximum booking range, and slot size margins. This allows for flexible control of room usage and scheduling rules.

- **Location & Address:**

Locations and addresses encapsulate geographical information. Each location includes detailed address data (street, city, postal code, country). This supports map integration for event visualization.

- **Employee:**

Employees act as event hosts or administrators. Their records contain contact information, assigned roles, and links to hosted events. This connection enables event filtering by host and proper permission management.

3.3.2.2 Improvements:

- **Structure of Hierarchies**

Events now generate occurrences and instances, creating a clear parent-child relationship that simplifies recurrence handling and data querying.

- **Increased Adaptability**

The introduction of **BookingPolicy**, **Room**, and **Location** entities allows the system to accommodate multi-location scheduling and enforce specific organizational constraints.

- **Management of Queues and Capacity**

Each event can define capacity and queue limits, enabling real-time queue handling and automatic promotion of users when spots become available.

- **Enhanced Role Separation and Security**

The use of **allowedUserIds**, **allowedGroupIds**, and **Role** ensures that access control is tightly managed within the database level.

- **Scalability and Consistency**

The final model supports recurring events, multi-host collaboration, and location scheduling while maintaining efficient data retrieval through structured references.

3.3.3 Summary

This final scheme successfully integrates recurrence, location, and queueing features while preserving extensibility for future additions such as room availability tracking or external booking integration. It forms a stable and scalable backbone for the entire calendar system, aligning with the functional and non-functional requirements set out by NullSpace.

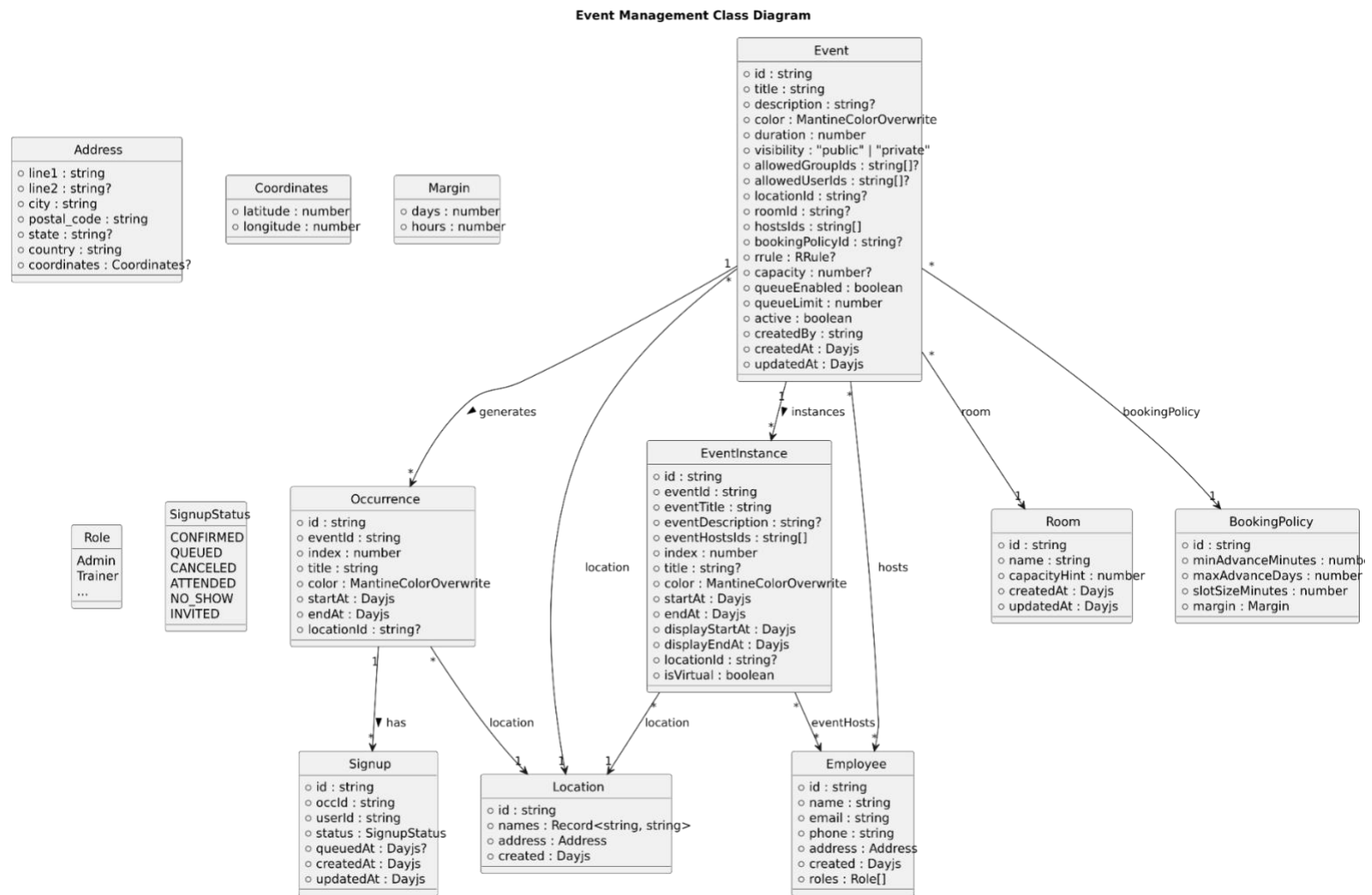


Figure 9: Class diagram of the data scheme

4- Planning

To facilitate the design and development processes. The team adopted a plan to onboard, design, implement, and test the product. In addition, we decided on what tools to use for goal-alignment and the recurring meetings we would have with our supervisors. Finally, we detailed out, in a Gantt chart, the timeline of the project.

4.1 Project Phases and Timeline

This project followed a 10-week schedule defined by NullSpace, with weekly collaboration both at the company office and remotely. Each week combined technical work with ongoing report writing.

Week 1 - Onboarding & Setup

- Held an introduction with the team members and the Company Supervisor (founder of NullSpace).
- Set up the project environment (GitHub, Jira, Miro, Firebase).
- Began to familiarize ourselves with the existing codebase and requirements provided by NullSpace.
- Searched for and confirmed the project supervisor.

Week 2 - Requirements & Initial Design

- Analyzed requirements and scheduling needs.
- Created the first draft of the database schema and backend architecture.
- Fixed small bugs in the existing website to improve tech stack intuition.
- Held an initial feedback session with the client.
- Started drafting the design report.
- Discussed possible edge cases and further clarified the scope with the client.

Week (3-4) - System & Database Design

- Complete the detailed NoSQL schema design.
- Define API endpoints for scheduling features.
- Document recurrence rules and booking flows.
- Continue design report writing and incorporate supervisor feedback.
- Purge irrelevant calendar-related components, functions, and database collections.

Week (5-8) - Implementation

- Develop backend features, including:
- Event creation and recurrence handling.
- Multi-location booking and user availability checks.
- Queue/waitlist logic and event visibility options.
- Integrate with Firebase
- Perform continuous unit testing and debugging.
- Add weekly updates to the design report.

Week (5-9) - Testing & Integration

- Integrate the system with the existing React frontend.
- Conduct performance testing.
- Validate edge cases and user scenarios.
- Make final iterations on the design report.

Week 10 - Finalization

- Review and clean up the codebase.
- Prepare the final documentation and handover to NullSpace.
- Finalize the design report and presentation.

4.2 Tools and Coordination

We also utilized coordination tools that facilitate the development of the project. We needed a task management tool that allows us to view all the epics and requirements. In addition, we were given a set of other tools provided by the client. Finally, we mention the applications we used to communicate on a day-to-day basis.

- **Task Management:** Jira (backlog, sprint planning). This enterprise software granted us access to the list of requirements provided by the client. It also allowed direct branching and feature-tracking.
- **Design & Brainstorming:** Miro (system diagrams, workflows). In the first phase, this was used for an abstract overview of the design of the system.

- **Version Control:** Git and GitHub. We utilized these to allow for parallel work, as well as keeping track of the code's history.
- **Communication:** Google Meet, Discord, WhatsApp. These were employed for arranging meetings, discussing specific code/implementation issues, and other academic arrangements.
- **In-person collaboration:**
 - 2 days per week at the NullSpace office: for bi-weekly standups, technical discussions, and help with integration with existing modules.
 - 1 day per week with the supervisor: to develop and refine the Design report.
- **Documentation:** Shared Google Drive.

4.3 Timeline Overview (Gantt-style table)

By combining weekly progress with continuous documentation and close coordination with both the Company Supervisor and the project supervisor, the project stayed on track and was completed within the planned 10-week schedule. In the following sections, we overview our undertaking as a result of our planning.

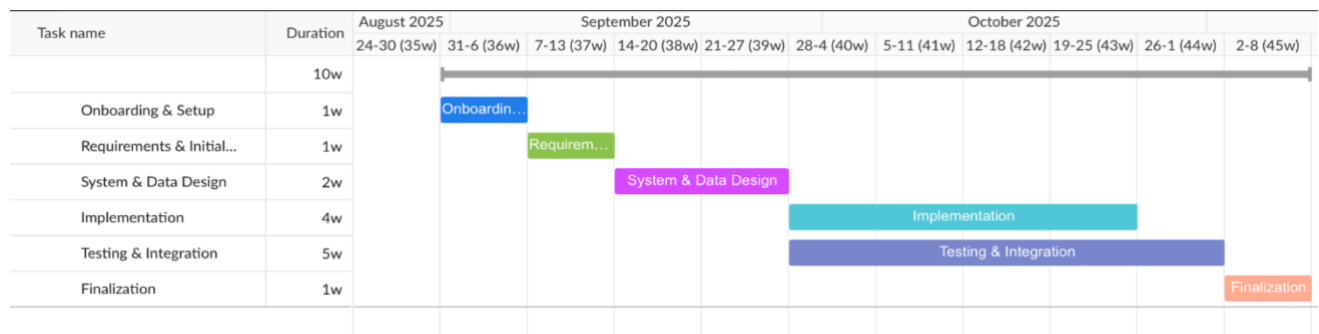


Figure 10: Gantt chart for organization

5- Implementation

In this section, we explore how we implemented the calendar product in a lower-level format. We outline the technologies, external Application Programming Interfaces (APIs), and code structure. We explain how we implemented certain features in more detail, and what compromises we made to be able to develop those features.

5.1 Technology Stack

The client has already built the rest of the Nullspace application using React and Typescript. As such, to maintain compatibility with the rest of the codebase, we used the same stack for our implementation.

React is a Javascript framework that allows a greater degree of freedom and simplified manipulation of the DOM (Document Object Model). With its component system, we can modify individual elements in a page, including the underlying business logic, in a manner that adopts low coupling and high cohesion.

React comes bundled with Typescript. As such, we get access to a strict type of system which allows us to properly define variables and constants for improved maintainability and cohesiveness.

Thanks to Firebase's versatility, a standalone backend server and database are not needed. Below, we overview how we utilized these technologies, and what limitations/challenges we faced when using them to build the calendar system.

5.1.1 Calendar-specific Libraries

Our calendar implementation makes use of these critical libraries:

- [day.js](#): is a lightweight JS date library used throughout the calendar system for date/time manipulation and formatting. Without it, the platform would not be able to display event time, generate cards on the calendar, allow admins to create and edit events, etc.
- [rrule.js](#): is a JS library that implements the iCalendar RFC specification (5545) for recurring events. In this codebase, it powers the entire event recurrence system, which was an essential requirement provided by the client. This library allows us to store event information in a manner so efficient that we can populate the calendar with event recurrences that span decades. In the "Frontend" section, we dive deeper into how it's used to populate the calendar with events.
- [Mantine v6](#): is a styling library that we use to structure and style the calendar. It is used to customize every single calendar component. As this was already used by the client

throughout the codebase, in order to maintain harmony across the entire product suite, we also had to use this for our components. This is a core part of the usability requirement; customers should not be confused by inconsistent element design on the platform.

5.1.2 External APIs

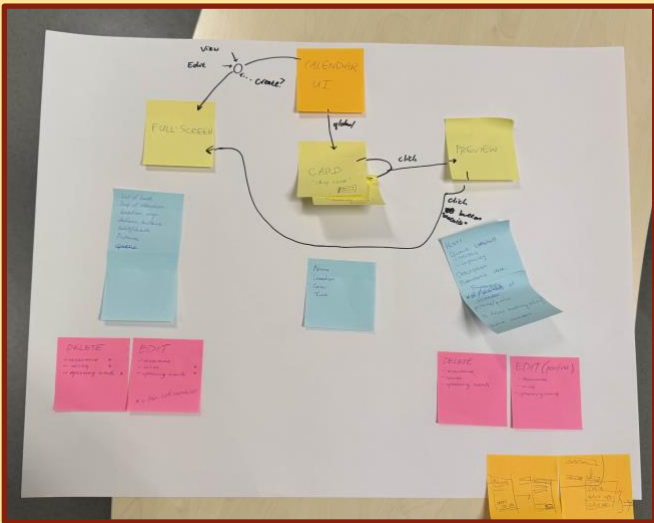
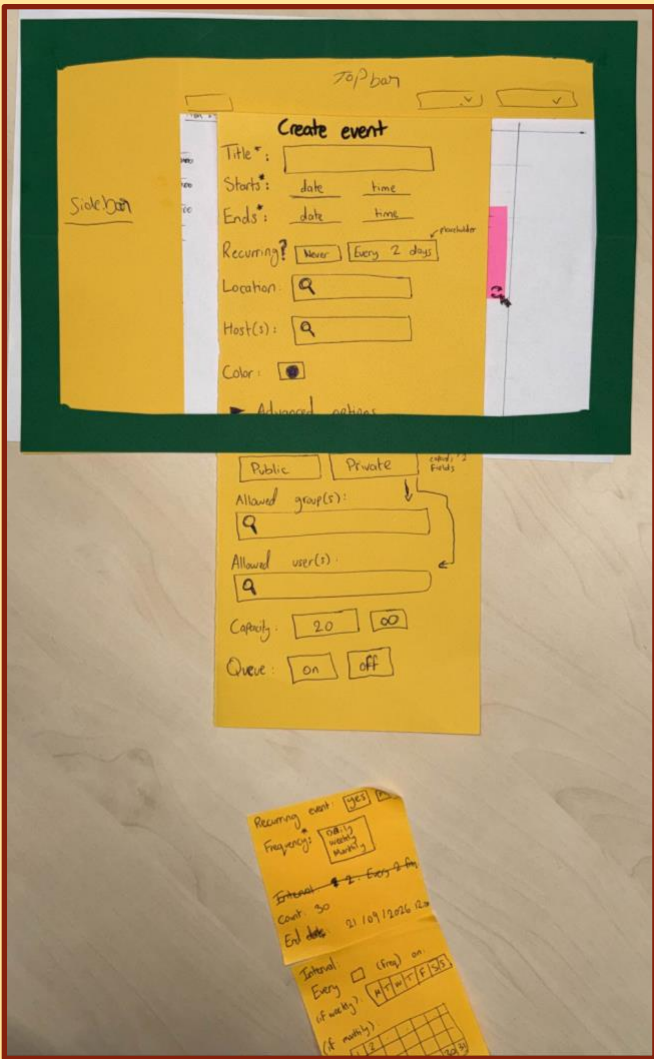
We also utilize external APIs already used by the client throughout the NullSpace application:

- **Stripe**: is a Software-as-a-Service (SaaS) platform that focuses on payments. In our implementation, it stores customer information linked to Firebase user IDs via metadata. This information is used throughout the calendar to display attendee names and emails. More importantly, for the next iteration of this product, this integration will be important for businesses using the NullSpace calendar to charge their customers' subscriptions, lock events behind paywalls, etc.
- **Google Maps**: is integrated to display interactive maps showing event locations. The system fetches pre-determined location data (with coordinates) from Firestore, then renders embedded maps with markers in the event details page.

5.2 Mock-up

The mock-up and frontend design of how the website is going to look for the calendar system was designed by our group during week 5. A session was made among the group members, the client (Jasper Van Amerongen) and the design company.

The company, "PAP!", helped us design the calendar system for both usability and for the aesthetics of the calendar system. They pointed out the parts in which we need to take in consideration the usability of the user and suggested some parts which need more clarification in context.



30

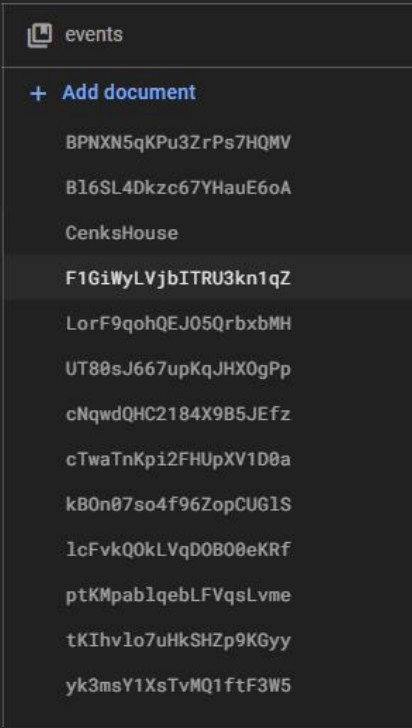
5.3 Firebase (Backend)

Firebase is Google's Backend-as-a-Service (BAAS) platform that replaces traditional server infrastructure with an integrated suite of cloud services. Instead of managing separate servers, databases, and APIs, Firebase provides authentication, a realtime database, and serverless cloud functions that connect directly to the frontend through a SDK. For the calendar, this means user authentication, event storage, and live data synchronization for event creation, registration, queuing, etc.

Firestore is Firebase's NoSQL document database that organizes data hierarchically rather than in SQL tables. Which makes optimizations like database normalization unnecessary; this allows us to dynamically alter the database according to evolving technical requirements. The calendar uses a three-tier structure:

- `/events/{eventId}` stores base information (title, recurrence rules, capacity, etc.),
- It contains the `/occurrences/{occurrenceId}` subcollection for specific instances.
- Inside each occurrence document lives a `/signups/{signupId}` subcollection for tracking registrations.

Here are what the collections, documents, and subcollections look like in Firestore:

Events collection	Nonrecurring event document (<code>rrule</code> is set to null here):	<code>rrule</code> object (only for recurrent events):
	<pre> allowedGroupIds: null allowedUserIds: null bookingPolicyId: null capacity: null color: "gold" createdAt: 23 October 2025 at 21:56:54 UTC+2 createdBy: "VVZZmFVm0cUrqKzlrG7Rzl5x55I2" description: null duration: 60 hostsIds 0 "VVZZmFVm0cUrqKzlrG7Rzl5x55I2" locationId: "mB1J9LLzIEf6VlBCrSsX" queueEnabled: false queueLimit: 1 roomId: null rrule: null status: "CANCELED" title: "CrossFit Gauntlet" updatedAt: 24 October 2025 at 13:56:52 UTC+2 visibility: "public" </pre>	<pre> rrule byeaster: null byhour: [17] byminute: [0] bymonth: null bymonthday bymonthday: [] byweekday: null bysecond: [0] bysetpos: null byweekday: ["TU", "TH", "SA"] byweekno: null byyearday: null count: null dtstart: 16 October 2025 at 19:00:00 UTC+2 freq: "WEEKLY" interval: 1 tzid: null until: null wkst: 0 </pre>

Occurrence document

58JqG7R3xbZ2WqTsZtX0

+ Start collection

signups

+ Add field

color: "blue"
 endAt: 6 December 2025 at 10:00:00 UTC+1
 index: 43
 locationId: null
 startAt: 6 December 2025 at 09:00:00 UTC+1
 status: "CANCELED"
 title: "New StartAt with end"

Signup document

D799XOrG0UNYtMzXNXFx

+ Start collection

+ Add field

createdAt: 29 October 2025 at 13:05:37 UTC+1
 status: "CANCELED"
 updatedAt: 29 October 2025 at 13:05:39 UTC+1
 userId: "NHDnZBnLcqha0ayJ4l8xzJkWHW33"

5.4 Calendar Page (Frontend)

In this subsection, we delve deeper into the ‘heaviest’ part of the implementation. Because we do not have a dedicated backend per se, functionality that would normally exist in the backend now lives on the client-side (also known as the frontend). We explain what React and Typescript are, how we used them to develop components that acted as features, and how the code is structured.

5.4.1 React’s Component-Based Approach

React’s primary advantage lies in its component-based architecture, which enables developers to build complex user interfaces from small, isolated, and reusable pieces of code. Each component ‘encapsulates’ its own structure (markup), behavior (logic), and presentation (styling).

In the calendar system, this philosophy manifests clearly: an **EventCard** component is responsible solely for displaying a single event on the calendar, while the **RenderedCalendar** component orchestrates the overall calendar layout. If the design of event cards needs to change, we modify only the **EventCard** component without touching the rendering logic elsewhere.

React's **virtual DOM** further enhances performance by intelligently updating only the parts of the page that have changed, rather than re-rendering the entire interface. For a calendar displaying potentially hundreds of event instances across a month view, this selective rendering is crucial for maintaining responsiveness.

5.4.2 State Management

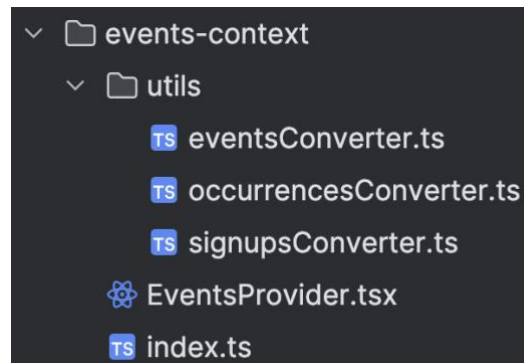
The calendar system's data flows through a **context/provider** pattern, the source of event data in our case is the **EventsProvider**. This architecture addresses a fundamental challenge in React apps: prop drilling - the tedious process of passing data through multiple components to get to a single component at the bottom of the hierarchy. A provider, such as the **EventsProvider** allows us to tap into the latest data without verbosity.

5.4.2.1 The Events Context Structure

The 'events context' serves as the application's single source of truth for calendar data. It consists of three main parts:

- Type definition (**index.ts**): Establishes a Typescript specification for all calendar-related types:
 - **EventDocument**: Represents a parent event document with all the event data, settings, and most importantly, the recurrence rule object (rrule). This document does not represent a single event at a point in time. However, its child (**OccurrenceDocument**) does.
 - **OccurrenceDocument**: Represents a specific instance of an event at a single point in time. The occurrence is an "instantiated" event instance, which means it was not virtually rendered using rrule, but instead lives in the database.
 - **SignupDocument**: Represents a single user's signup details.
 - **EventInstanceDocument**: A unified type used purely for rendering. It is used to render both virtually generated event instances and occurrences.
- **EventsProvider**: Manages realtime data synchronization with Firebase:
 - Listens to three Firestore collections: events, occurrences, and user signups.
 - Implements visibility querying: private events are shown only to authorized users.
 - Creates efficient lookup structures (**eventsMap**, **occurrencesMap**, **userSignupsByOccurrence**) to avoid O(n) searches.
 - Provides this data to all child components through the **<EventsContext.Provider>** component.

- Firestore Converters: Bridge the gap between database representation and application logic (detailed in next section)
- Any component within the calendar system can use this centralized data by calling `useEvents()`.



5.4.2.2 Scalable Data Fetching Strategy

As explained in the implementation section, EventsProvider is responsible for fetching data from firestore. However, as initially expected, fetching requires conditional queries instead of just using traditional and simple collection/collection group queries.

- **Conditional querying:** Due to security and optimization reasons, instead of filtering on the front end level, conditional querying is used to fetch data that the user is supposed to see. This is done in the following way:
 - If the user is an employee
 - Fetch all events as a collection
 - This way, occurrences are fetched as a collection group.
 - If the user is a customer
 - Fetch events with conditional queries
 - Conditions:
 - Events either public,
 - or the user is included in the allowed user's array to fetch even if the event is private.
 - This way, each fetched event has a snapshot to listen to its occurrences and fetch them.
- **Pagination:** Another optimization technique used is pagination, which essentially fetches only those instances that are inside the view window of the user.
 - Events: There are 3 types of events, window checks are performed accordingly
 - Recurring with end
 - Recurring without end
 - Non-recurring

5.4.3 Type Checking with Typescript

Typescript's integration provides compile-time guarantees that prevent entire categories of runtime errors. The calendar system leverages Typescript's type system extensively, particularly when handling the dual representation of event data:

- Firestore storage format:
 - Dates stored as Firebase Timestamp objects
 - RRule stored as plain Javascript objects (POJO) with serialized values
 - No RRule methods available
- Frontend storage format:
 - Dates as Dayjs objects (providing .format(), .add(), .isBefore() methods)
 - RRule as an actual RRule instance
 - Deserialized RRule enums (weekdays and event frequency)

We do not store two different types of the same document for this sake. Instead, we use Typescript's conditional types to elegantly store a single type:

```
export type EventDocument<T = Dayjs, RR = true> = {  
  // ... other fields  
  rrule: RR extends true  
    ? RRule // Runtime: full RRule instance  
    : { freq: RRuleFrequency, ... } // Storage: serialized object  
  createdAt: T // Either Dayjs or Timestamp  
}
```

5.4.4 Component Hierarchy

Below, we outline the exact structure of the codebase. `NullSpaceCoreApp` handles routing between the different calendar paths. The `Calendar.tsx` component stores all of the relevant calendar components. The `EventDetailsPane.tsx` component, since it has its own url path, operates as a standalone component with its relevant child components.

Routing:

- **NullSpaceCoreApp** (Routing layer)
 - `/app/client/calendar` → `Calendar`
 - `/app/client/calendar/:eventId:ocId` → `EventDetailsPane`
 - `/app/admin/calendar` → `Calendar`
 - `/app/admin/calendar/new` → `EventDetailsPane`
 - `/app/admin/calendar/:eventId:ocId` → `EventDetailsPane`

Calendar View:

- **Calendar** (`/pages/Calendar.tsx`)
 - **CalendarProvider** (Context: date, scope state)
 - **RenderedCalendar** (Orchestrates rendering, generates instances)
 - **CalendarNavigation** (Date picker, view switcher)
 - **ColumnHeaders** (Day/date labels - only for day/week views)
 - **ColumnCalendar** (Day/Week vertical timeline view)
 - **EventCard** (Single event display)
 - **EventPopover** (Click handler → detailed popup)
 - Join/Leave button
 - Event details (time, location, capacity)
 - **MonthCalendar** (Month grid view)
 - **MonthEventPopover** (Compact event button)
 - **EventPopover** (Same detailed popup component)

Event Details View:

- **EventDetailsPane** (</calendar/eventId/occlid>)
 - Event form fields (title, description, location, etc.)
 - Recurrence configuration (RRule builder)
 - Google Maps display (location visualization)
 - **Attendee Management Section** (Admin only)
 - Confirmed attendees list
 - **AttendeeRow** × N (Individual attendee with remove button)
 - Waitlist
 - **AttendeeRow** × N (Individual attendee with promote/remove buttons)
- Supporting Components
- **AttendeeRow** (Reusable attendee display)
 - Avatar with initials
 - Name and email
 - Action buttons (Promote/Remove)

5.4.5 Components Explored in Detail

In this part, we explore components of the calendar in a visual manner. For clarity's sake, we present the employee's **view in dark mode**, and the client's **view in light mode**, albeit both dark and light modes are available to admins and clients.

Column Calendar

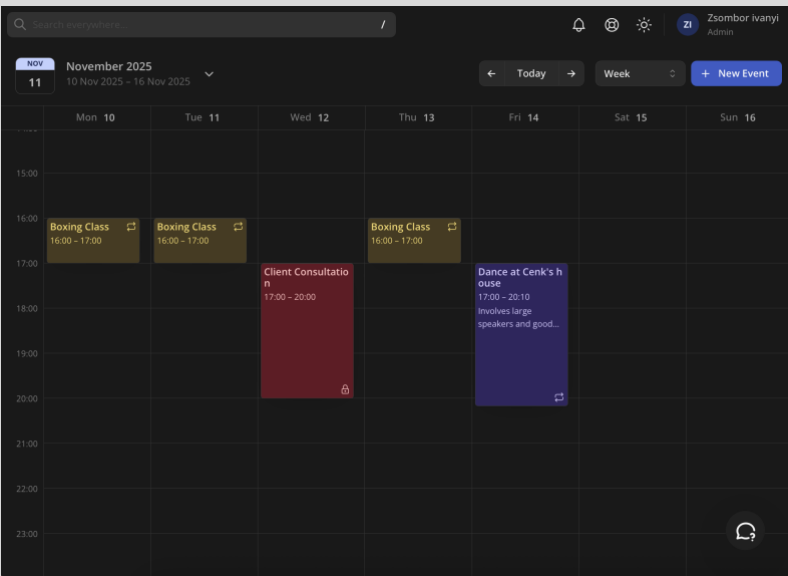


Figure 12: Employee view.

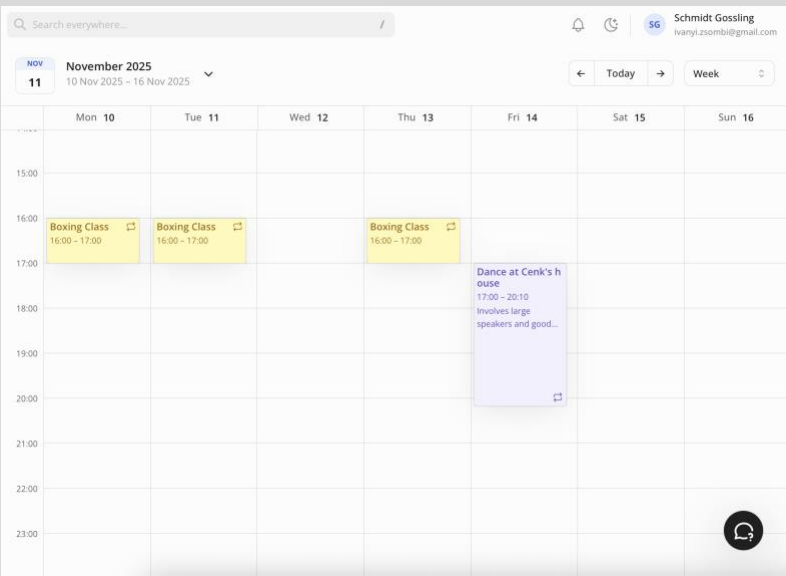


Figure 13: Customer view.

These screenshots show the calendar’s weekly view. Notice that the private event ‘Client Consultation’ is not visible to the customer (right side) because they were not invited.

Event Card

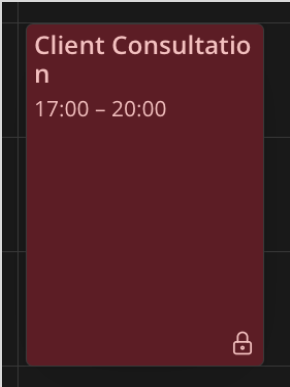


Figure 14A13: Employee view of a private event card.

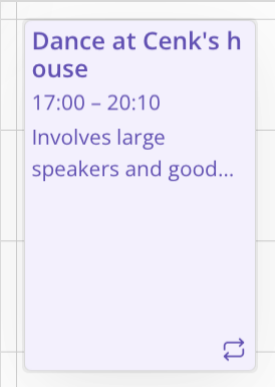


Figure 14B: Recurrent event card.

On the left side, the previously explained private event is visible, while on the right side we can see a recurring event. Recurrence and the private state are indicated by symbols in the bottom-right corner of the event card.

Event Popover (Admin and Customer views)

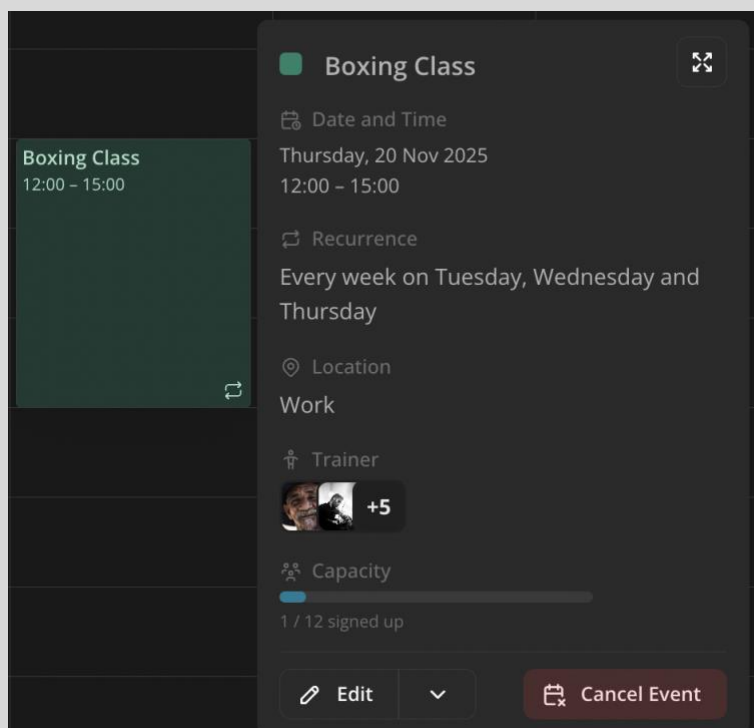


Figure 15: Employee view.

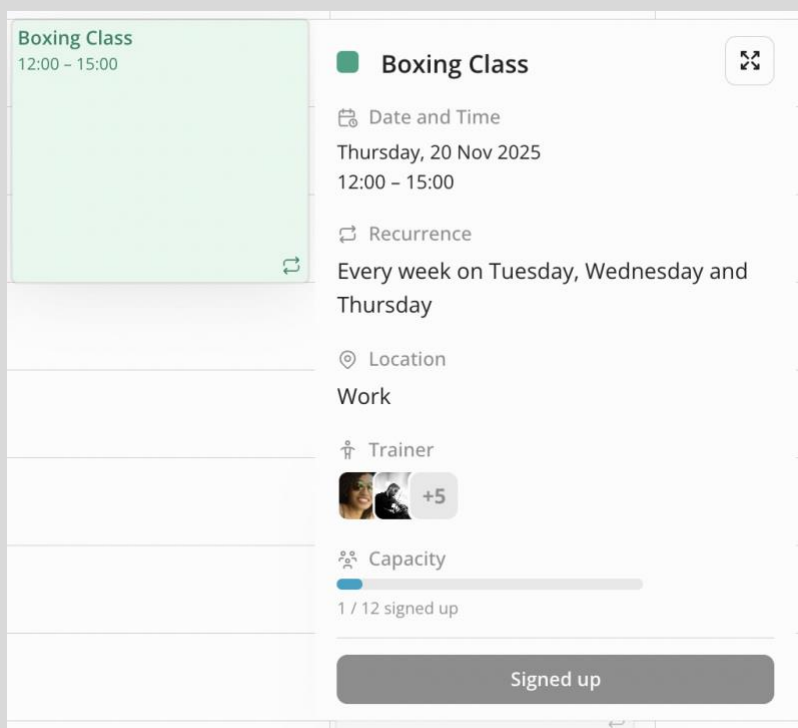


Figure 16: Customer view.

The customer and employee versions of the popover are shown. To open the popover, the user must click on the event card. In the customer screenshot, the popover of an already signed-up user is visible, while in the employee version, options to cancel or edit the event are available. Both versions include a full screen button.

Month Calendar

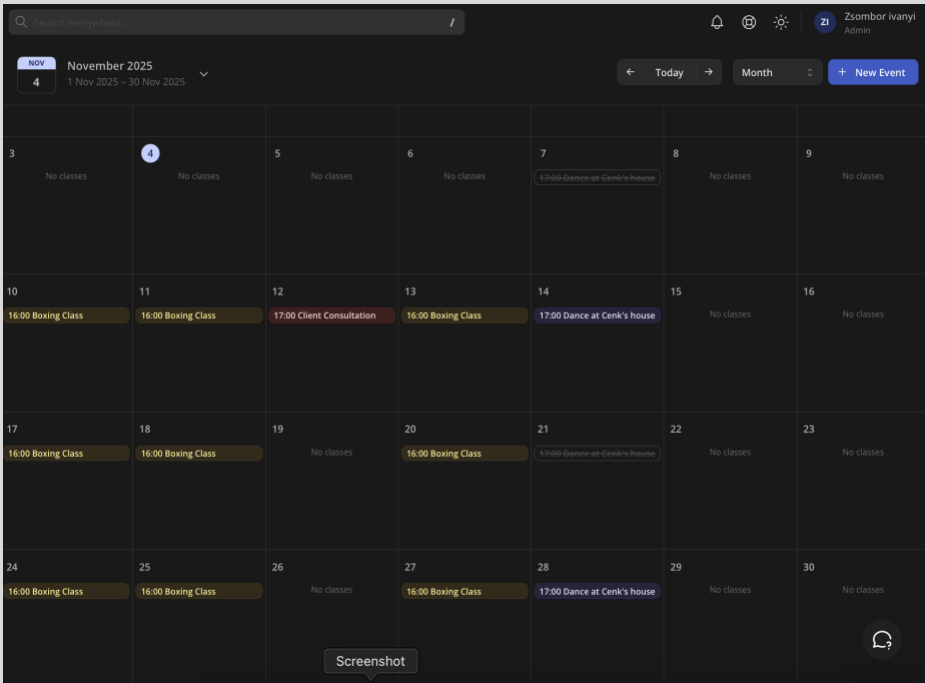


Figure 17: Employee month calendar view.

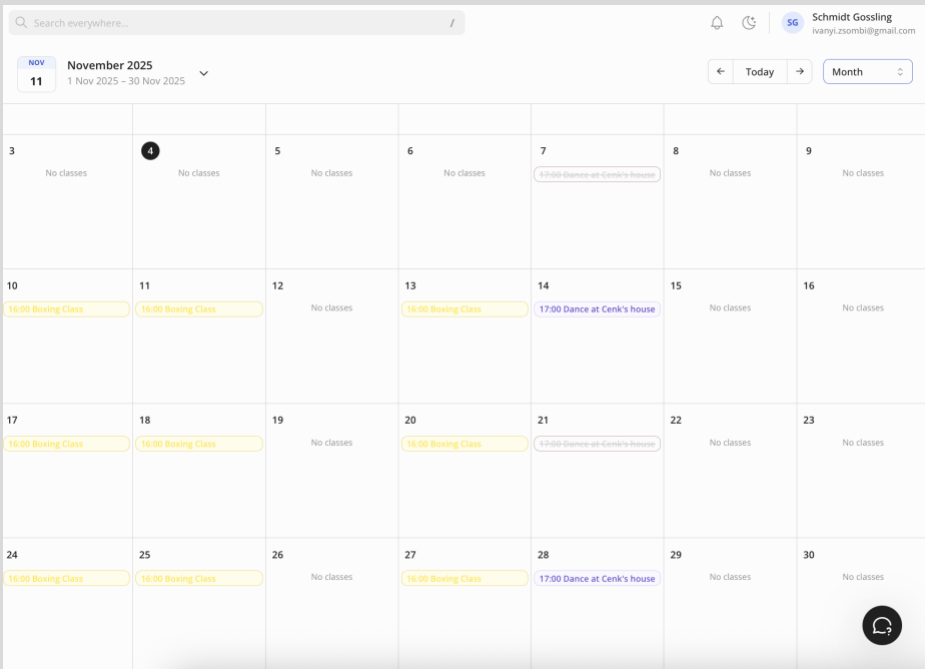


Figure 18: Customer month calendar view

These screenshots show the calendar’s monthly view. Notice that the private event ‘Client Consultation’ here is also not visible to the customer.

Event Details Pane (Creation / Editing)

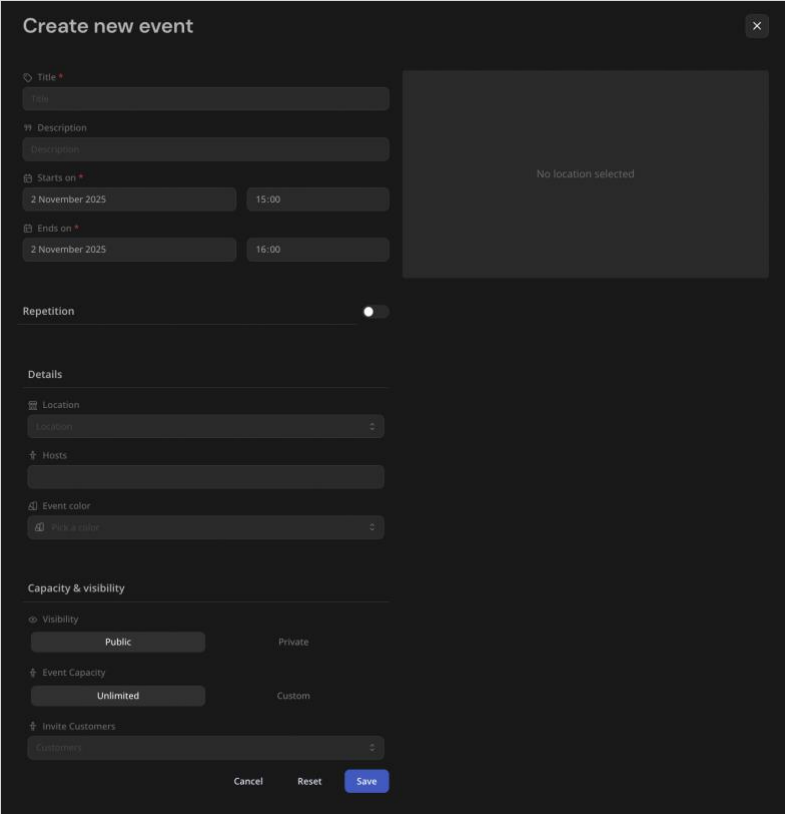


Figure 19: Event creation page.

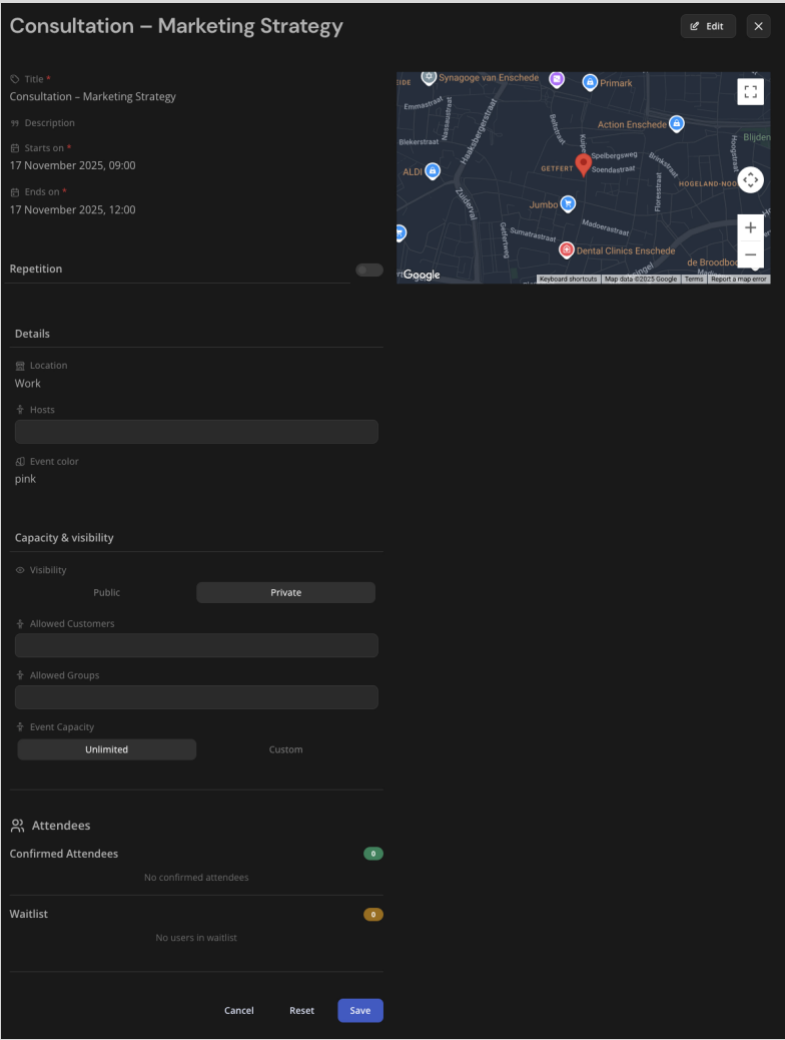
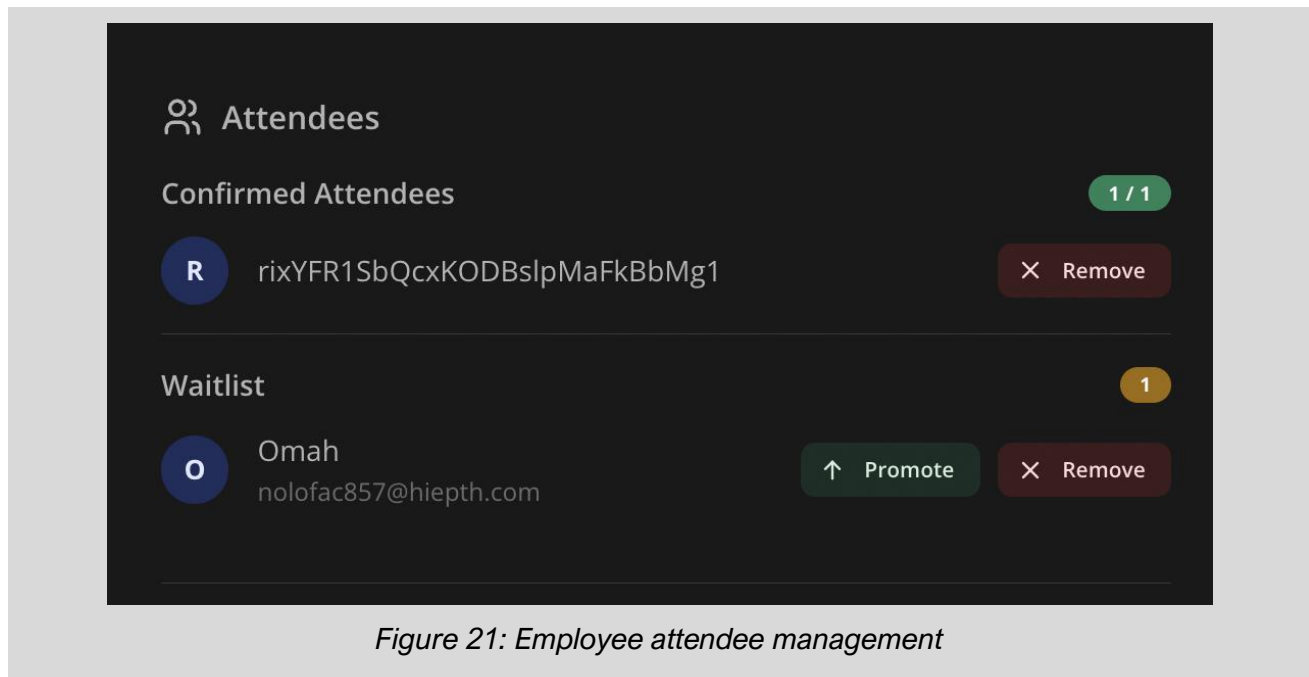


Figure 20: Event editing full page.

After the employee clicks on the full screen button on the popover, the editing page of the event is loaded. This page is quite like the event creation page, which opens when the new event button is pressed in any calendar view.

Attendee Management (Admin)

*Figure 21: Employee attendee management*

Employees can kick out signed up users from an event and promote users from the queue to the state signed up. Queue promotion is automatic by default.

5.5 Limitations

In this section, limitations will be explained that our team faced during implementation.

5.5.1 Conditional fetching

Data modeling and access are shaped by Firestore's constraints. We maintain two-way parent "<>" child relationships by fetching from both sides, which increases read counts and risks consistency drift if one side is updated without the other.

Event fetching with collection group queries is not possible due to firestore query limitations when using complex conditions. For this reason, events must be fetched through 3 different collection queries using separate listeners (snapshots) for each event type (non-recurring, recurring with end, recurring without end). Firestore does not let 2 relational equivalences check on 2 separate fields. The higher complexity level of the conditions is due to having pagination and visibility conditions in the front end when performing queries.

For example, applying these two relational checks in one query is not possible, since *rrule* and *startAt* are two different fields:

```
where("rrule.until", ">", windowStart),  
where("startAt", "<=", windowEnd),
```

Due to this, recurring events with an end date will be fetched even if the start date is after the window ends. This was proven to not cause a huge performance plummet, since the main optimization issue concerns the events in the past, while the weight of fetching future recurrent events with an end date won't cause huge problems.

Due to the same limitation explained above, occurrence fetching suffers the same fate in the case of the user being a customer. The most efficient collection group querying is not possible either. The best solution was proven to be the creation listeners for each event that was already fetched. If the user is an employee, all occurrences can be fetched through a collection group query within the scope window without any limitation.

5.5.2 Other limitations

Daylight Saving Time primarily impacts recurring series that span the DST change. A weekly "9:00" event that exists before and after the switch can drift by an hour because the series is anchored to a single UTC timestamp. Due to the time constraint, we were not able to rewrite the whole event rendering system to be compatible with storing local time as opposed to UTC timestamps.

Some behaviors are client-driven due to product and platform limits. Certain actions depend on the customer's subscription being loaded on the client, most notably: fetching the customer's name, adding them to an allow list, etc. Signup status is currently set from the frontend for speed, which shifts trust to the client and must be backed by strict rules or a server function to prevent spoofing. Queue movement is handled in the frontend as well, without a cloud function to apply side effects and enforce invariants.

6- Security Design

In our project, security was a major concern, with careful attention given to ensuring secure data exchanges across all components. Throughout development, the team tried to consistently follow established security conventions and best practices.

6.1 User Authentication

All users must be authenticated to access the web application. The product is gated by signing up page, which if the user goes through, only then can access the calendar schedule engine page. A subscription model will be implemented by the client in the future, which will connect the right customers to the right events.

6.2 User Input Sanitization

To prevent code injection vulnerabilities common in SQL-based databases, input sanitization is typically required. With Firestore's non-SQL architecture and native SDK, input sanitization occurs automatically, as all data passed through the SDK is securely handled. This significantly lowers the risk of security breaches related to injection attacks.

6.3 Database permissions

To ensure users can only view events they are authorized to access, all user and event data in Firestore is restricted to entitled users. Data security is enforced by a set of rules configured on the Firebase server, specifying read and write permissions. A core rule is that only authenticated users are permitted to access the database, providing a fundamental layer of protection against unauthorized access.

6.3.1 Employee Permissions:

- **Event** fetching: Employees can read all events and write to those they have created.
- **Occurrence** fetching: Employees can read all occurrences and write to those belonging to events they have created.
- **Signup** fetching: Employees can read all signups and write new signups.

6.3.2 Customer Permissions:

- **Event** fetching: Customers can view events only if they are public, if the customer is invited, or if the customer has previously signed up for the event.
- **Occurrence** fetching: Customers can view occurrences only if the related event is public, if the customer is invited, or if the customer has signed up for at least one occurrence of that event.
- **Signup** fetching: Customers can view only their own signups.

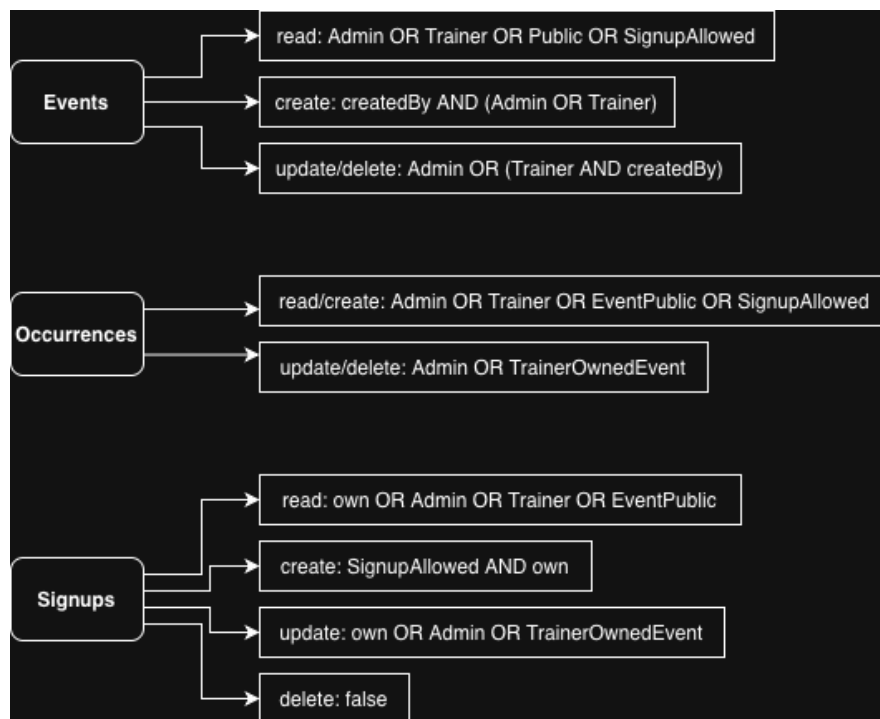


Figure 22: Diagram showcasing fetching rules.

7-Maintainability

We built the calendar system with a focus on selective maintainability, using deliberate architectural patterns and targeted documentation practices. Our code is strong in key areas like transaction handling, type safety, and data conversion. However, we tend to rely more on inline comments and logging than on comprehensive documentation.

The system's structure: featuring clearly defined read-process-write phases, JSDoc on utility functions, namespaced logging, and type-safe serializers. This shows intentional design aimed at clarity. That said, these standards aren't applied consistently. Some complex algorithms lack comments, the EventPopover component has grown beyond maintainable limits at 481 lines, and we currently have no automated tests covering business-critical logic.

This section highlights specific examples where maintainability was thoughtfully addressed, as well as areas where additional investment could significantly improve the system's long-term resilience.

Transaction phase clarity:

Sequential actions are clearly labeled for developers to understand what each part does.

```
packages/nullspace-core/src/features/calendar/handlers/handleEventTrigger.ts

await runTransaction(firebase.firestore, async transaction => {
  // 1. READ PHASE - All reads must happen before writes
  const eventSnap = await transaction.get(eventRef.withConverter(eventsConverter))
  const [existingSignupSnapshot, allSignupsSnapshot, queueSnapshot] = await Promise.all([...])

  // 2. COMPUTE PHASE - Determine signup status based on capacity
  let signupStatus: "CONFIRMED" | "QUEUED" = "CONFIRMED"
  if (event.capacity !== null && confirmedCount >= event.capacity) {
    signupStatus = "QUEUED"
  }

  // 3. WRITE PHASE - Atomic write
  transaction.update(signupDocRef, { status: signupStatus, ... })
})
```

JSDoc:

We use JSDoc to label each function, its purpose, the parameters it takes in, and the value it returns.

```
packages/nullspace-core/src/features/calendar/handlers/promoteNextQueued.ts

/**
 * Finds and promotes the next queued user to CONFIRMED
 * Used by both handleLeave and updateSignupStatus
 * @param transaction - Firestore transaction
 * @param occurrenceRef - Reference to the occurrence document
 * @returns true if a user was promoted, false otherwise
 */
export async function promoteNextQueued(
  transaction: Transaction,
  occurrenceRef: DocumentReference,
): Promise<boolean> {
  // Find next queued user (oldest first)
  const queueQuery = query(signupsCol, where("status", "==", "QUEUED"), orderBy("queuedAt", "asc"), limit(1))
  ...
}
```

Type-safe serialization

To ensure developers do not get confused about the type system as objects propagate through the codebase, we ensure that objects are serialized with the right types from the root.

```
packages/nullspace-core/src/contexts/events-context/utils/eventsConverter.ts

function serialiseRRuleFreq(freq: Frequency | null): RRuleFrequency {
  switch (freq) {
    case Frequency.YEARLY: return "YEARLY"
    case Frequency.MONTHLY: return "MONTHLY"
    // ... with safe fallback
    default: return "MONTHLY"
  }
}

// Safe date computation with fallback
const startDate =
  rrule?.options.dtstart ??
  (startAt ? startAt.toDate() : new Date()) // fallback if startAt undefined
```

Helper functions

Helper functions can be used in different components - eliminating the need for duplicate code and verbosity.

```
packages/nullspace-core/src/features/calendar/components/rendered-calendar/RenderedCalendar.tsx

// Allows occurrences to span more than one day or go past midnight
function splitInstanceByDay(eventInstance: EventInstanceDocument): EventInstanceDocument[] {
  ...
  displayStartAt: eventInstance.startAt, // keep original
  displayEndAt: eventInstance.endAt, // keep original
}

function toEventInstance(event: EventDocument, occurrence: OccurrenceDocument, isVirtual: boolean): EventInstanceDocument {
  ...
}
```

Namespaced logging for debugging

In the browser console, developers can see how components store/pass data when using different parts of the calendar. This approach has clear labeling where any issue pertaining to a component can be easily identified.

```
console.log("[handleJoin] Starting join for event:", eventInstance.eventTitle)
console.log("[EventsProvider] Fetching all occurrences to admin within window:", events.map(e => e.id))
console.log("[RenderedCalendar] Event ${event.id}: ${occurrences.length} occurrences")
console.log("[promoteNextQueued] Promoted user:", nextInQueue.data().userId)
```

8- Testing

In this phase of the project, only Alpha Testing was conducted. Since the system is still under active development and has not yet been released to external users, testing was performed internally by the development team and project stakeholders. The purpose of **Alpha Testing** is to validate the core functionality of the system in a controlled environment before it is exposed to real users.

Alpha Testing is a type of internal acceptance testing performed by the developers and internal staff before releasing the product to external users or beta testers. It focuses on verifying that all core functionalities operate as intended, identifying defects, and ensuring that the product meets the specified functional and non-functional requirements. According to ISO/IEC/IEEE 29119-1:2022 (Software Testing – Concepts and Definitions), alpha testing serves as an initial validation activity conducted in a controlled environment prior to external evaluation.

8.1 Test Cases

8.1.1 Test Case 1

Test Case ID: TC-01

Title: Event Creation

Objective: Verify that an admin can successfully create an event with valid parameters and that invalid date ranges trigger an error message.

Preconditions: The admin is logged in and has access to the calendar page.

Test steps:

1. Navigate to the calendar page.
2. Click on the “**New Event**” button.
3. Fill in all required fields with valid data.

4. Set the start date after the end date → expect an error message.
5. Set the event to private.
6. Correct the dates and submit the form.

Expected Result:

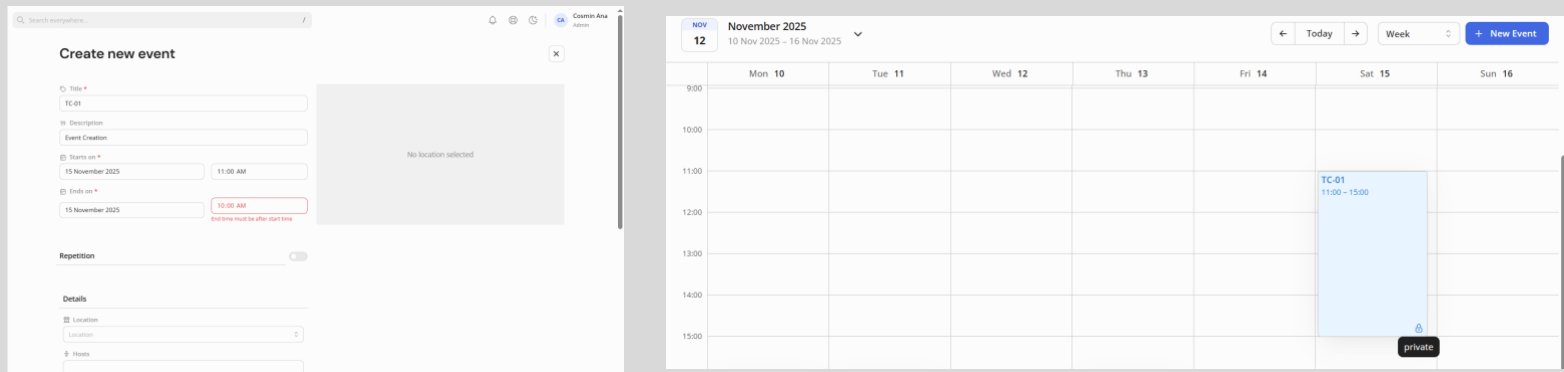
- The system displays an error when start date > end date.
- When corrected, the event appears on the calendar with all details visible (title, description, location, start date, end date, etc.).

Actual Result: Works as expected

Status: Pass

Related User Stories:

- Admin users can make an event private. **(Must have)**



Figures 23: Results for test case 1.

8.1.2 Test Case 2

Test Case ID: TC-02

Title: Event Inspection (Fullscreen)

Objective: Verify that a user can open an event in fullscreen mode and view all related details.

Preconditions:

- The user is logged into the system.
- At least one event exists on the calendar.

Test steps:

1. Navigate to the calendar page.
2. Click on an event card to open its popover.
3. Select the **Fullscreen View** icon.

Expected Result:

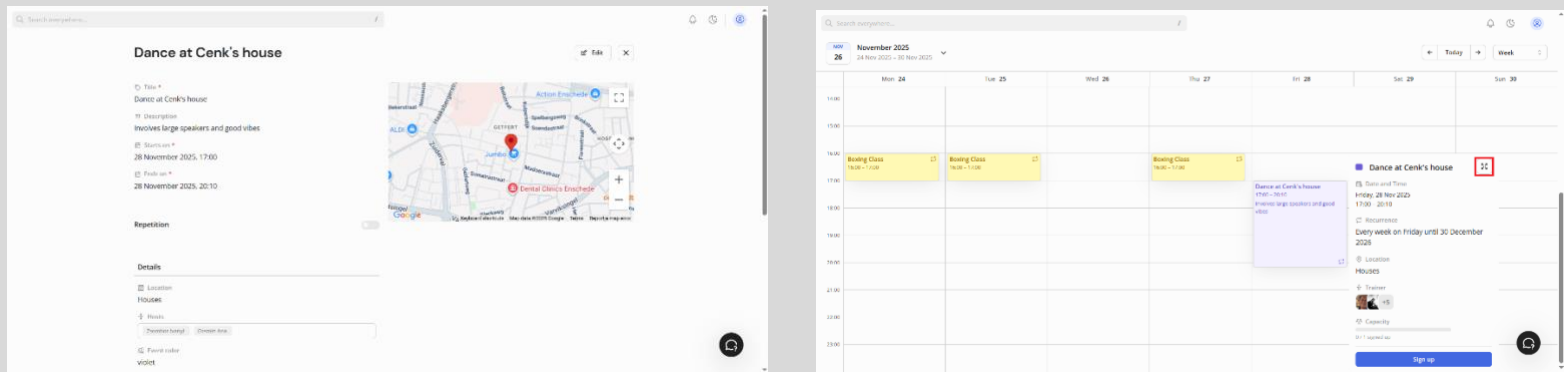
- The event is viewed in fullscreen with all event information displayed.

Actual Result: Works as expected

Status: Pass

Related User Stories:

- Admin users can see the event in fullscreen view. **(Must have)**



Figures 24: Results for test case 2.

8.1.3 Test Case 3

Test Case ID: TC-03

Title: Event Editing (Fullscreen)

Objective: Verify that an admin can open an event in fullscreen mode and edit its information successfully.

Preconditions:

- The user is logged in as an admin.
- At least one event exists on the calendar.

Test steps:

1. Navigate to the calendar page.
2. Click on an event card to open its popover.

3. Select the **Fullscreen View** icon.
4. Click the **Edit** option.
5. Modify one or more event fields (e.g., title or location).
6. Save the changes.

Expected Result:

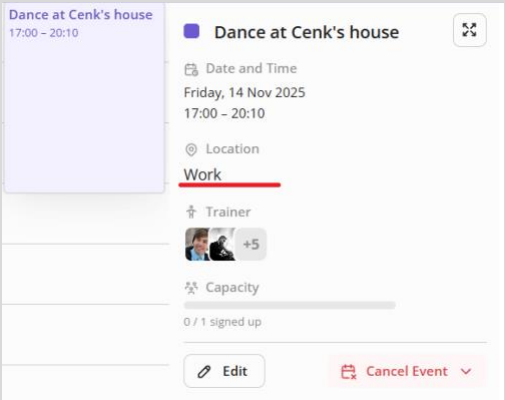
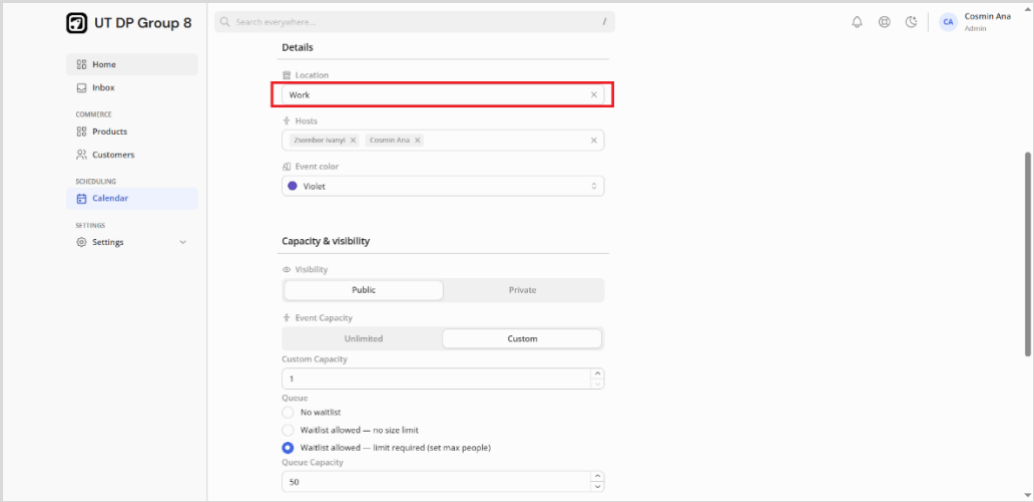
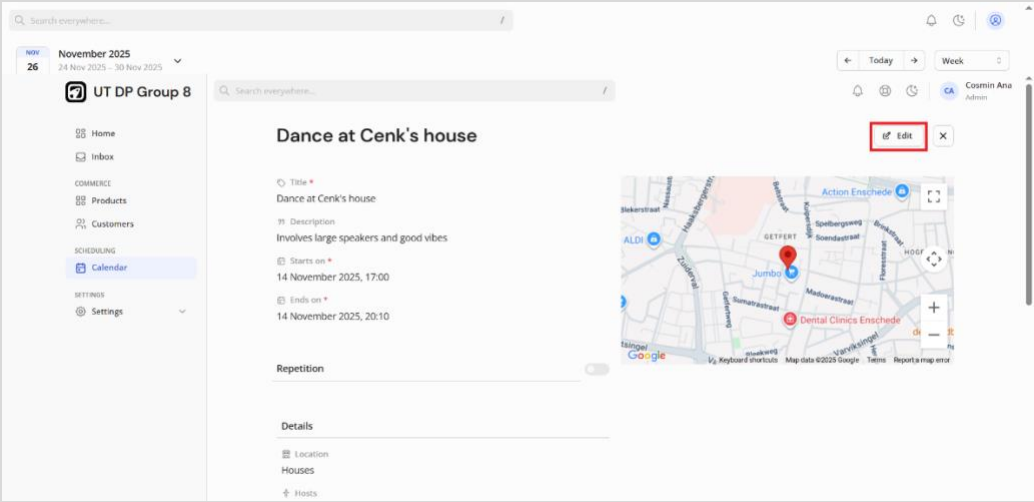
- The system updates the event with the new details and displays them correctly in the calendar.

Actual Result: Works as expected

Status: Pass

Related User Stories:

- Admin users can edit the event in fullscreen view. **(Must have)**



Figures 25: Results for test case 3.

8.1.4 Test Case 4

Test Case ID: TC-04

Title: Recurrent Event Creation (Daily / Yearly)

Objective: Verify that an admin can create daily and yearly recurring events without errors.

Preconditions:

- The user is logged in as an admin.

Test steps:

1. Navigate to the event creation form.
2. Set up a recurring event to repeat every **3 days** with **no end date**.
3. Submit the event.
4. Repeat the process for a **yearly recurrence**, set to repeat every **2 years**.

Expected Result:

- Daily recurring events appear in the calendar every 3 days.
- Yearly recurring events appear every 2 years as configured.

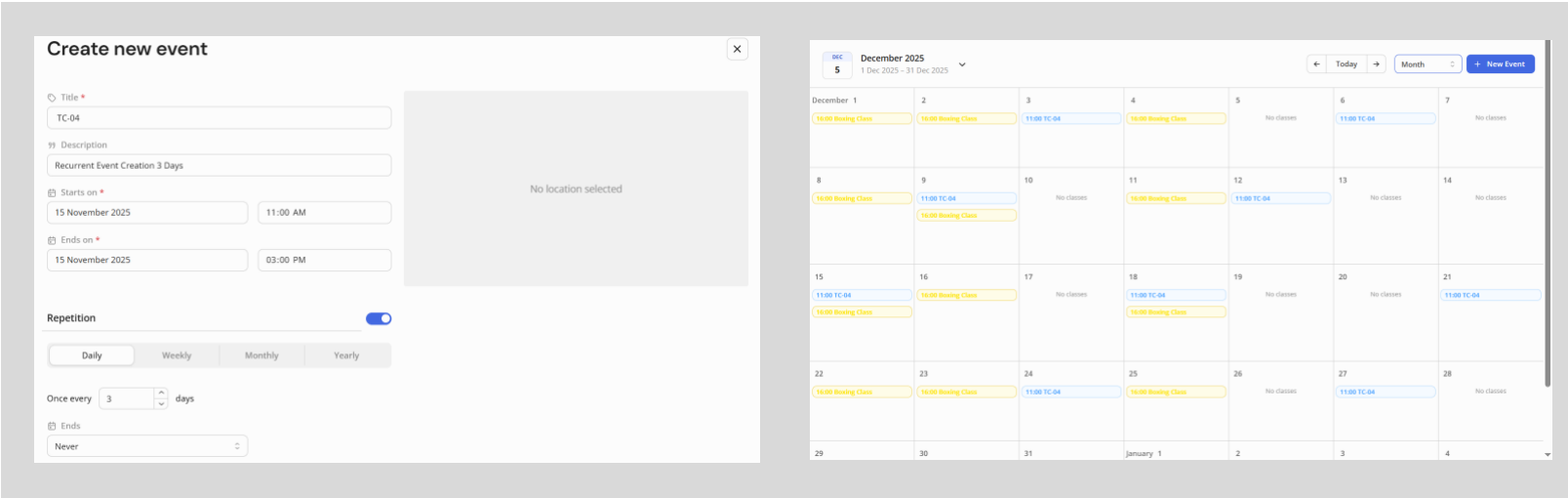
Actual Result: Works as expected

Status: Pass

Related User Stories:

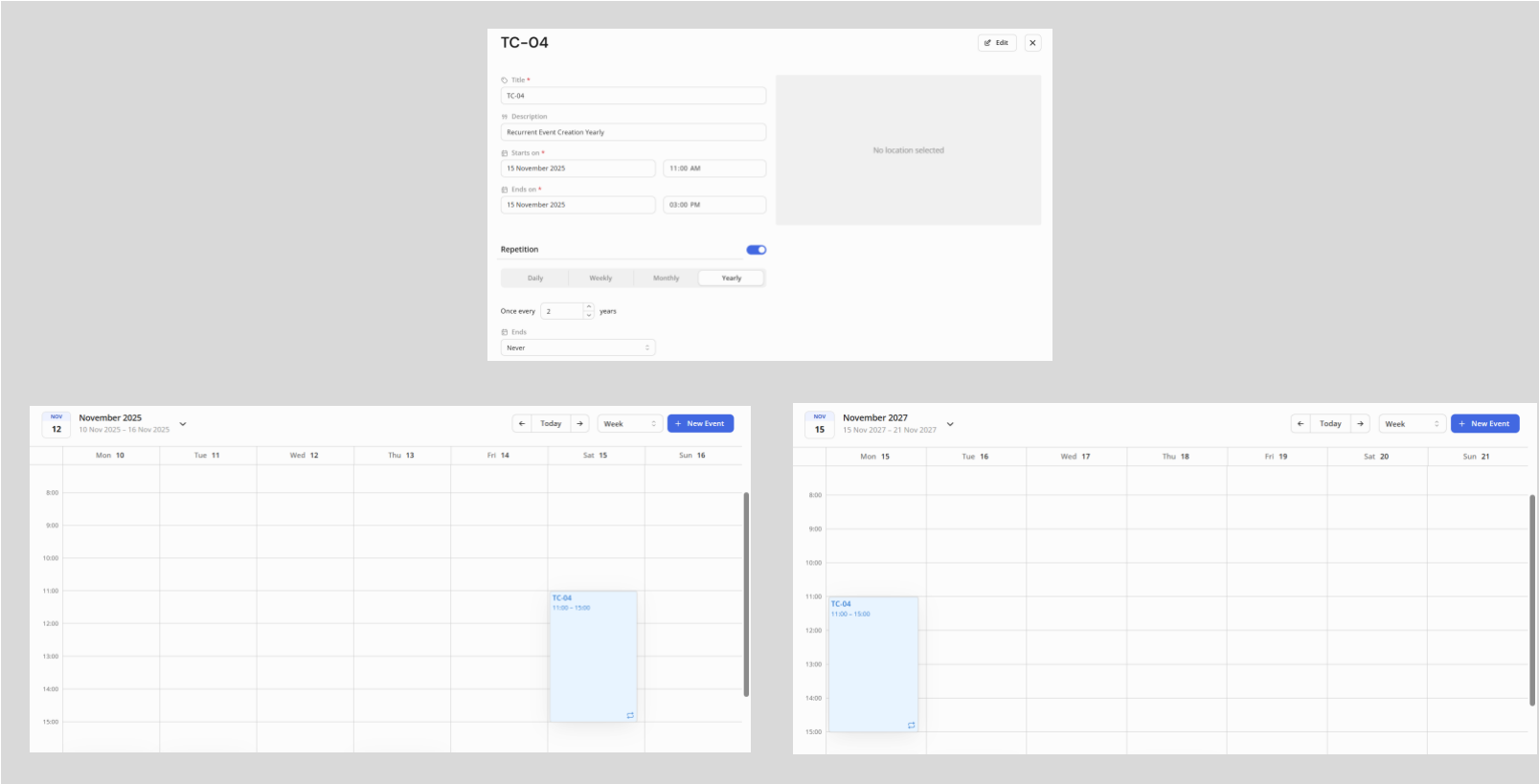
- Admin users can create recurrent events. **(Must have)**

Every 3 days



Figures 26: 3 day result for test case 4.

Every 2 years



Figures 27: Yealy result for test case 4.

8.1.5 Test Case 5

Test Case ID: TC-05

Title: Recurrent Event Creation (Weekly)

Objective: Verify that an admin can create weekly recurring events with multiple days selected.

Preconditions:

- The user is logged in as an admin.

Test steps:

1. Navigate to the event creation form.
2. Configure a recurring event to repeat every **2 weeks** on **Fridays and Tuesdays**.
3. Submit the event.

Expected Result:

- Recurring events appear in the calendar every 2 weeks on the specified days (Friday and Tuesday).

Actual Result: Works as expected

Status: Pass

Related User Stories:

- Admin users can create recurrent events. **(Must have)**

Create new event ✕

Title *
TC-05

Description
Recurrent Event Creation (Weekly)

Starts on *
15 November 2025 11:00 AM

Ends on *
15 November 2025 03:00 PM

Repetition ☑

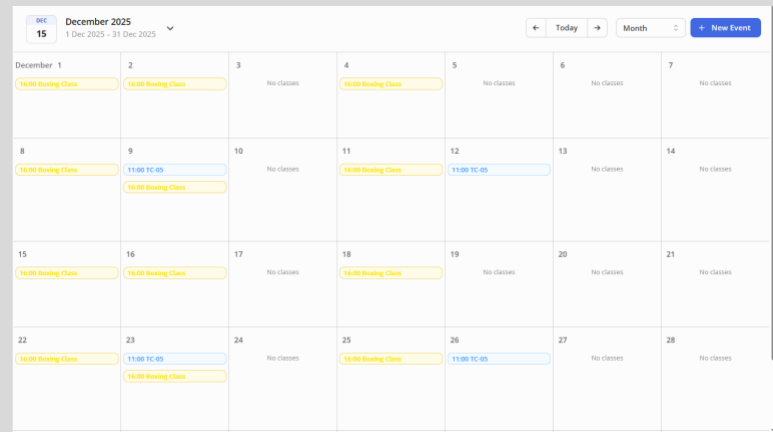
Daily Weekly Monthly Yearly

Once every 2 weeks

M T W T F S S

Ends
Never

No location selected



Figures 28: Results for test case 5.

8.1.6 Test Case 6

Test Case ID: TC-06

Title: Recurrent Event Creation (Monthly)

Objective: Verify that an admin can create monthly recurring events with both date-based and weekday-based rules.

Preconditions:

- The user is logged in as an admin.

Test steps:

- Navigate to the event creation form.
- Configure a recurring event to repeat every **2 months** on the **6th, 8th, and 21st** days.
- Submit the event.
- Configure another recurring event to repeat every **3 months** on the **3rd Sunday**.



5. Submit the event.

Expected Result:

- The system creates and displays all events correctly on the specified days and months.

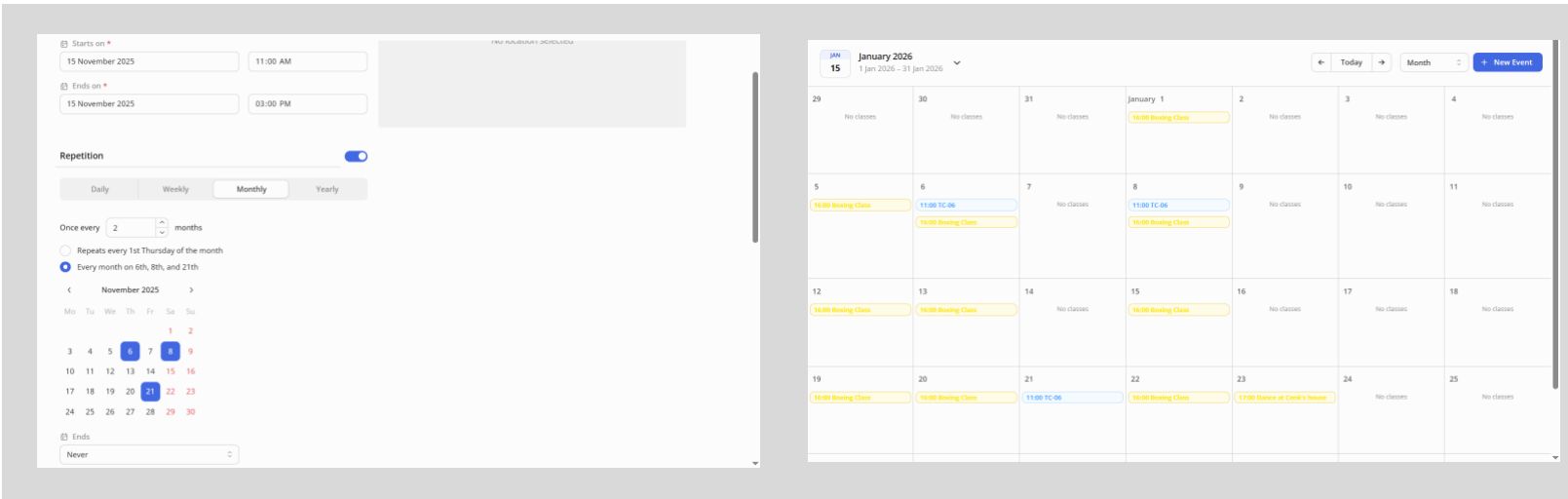
Actual Result: Works as expected

Status: Pass

Related User Stories:

- Admin users can create recurrent events. **(Must have)**

Every 2 months



Figures 29: 2 months results for test case 6.

Every 3 months

Repetition 🔵

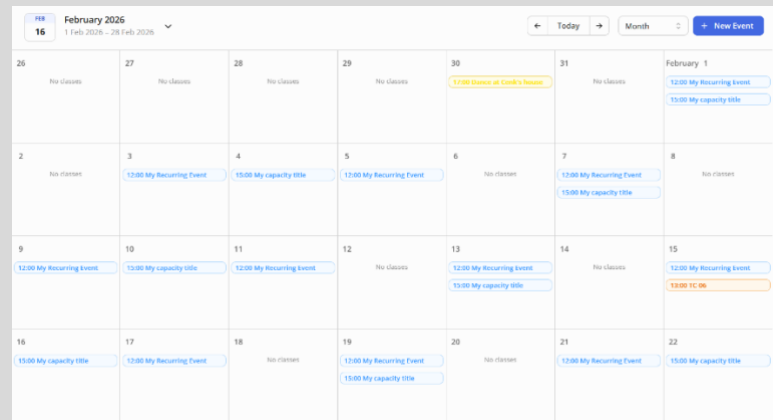
☐ Daily
 ☐ Weekly
 ☒ Monthly
 ☐ Yearly

Once every months

☒ Repeats every 3rd Sunday of the month
☐ Every month on 16th

Pick a reference date (determines Nth weekday)

☐ Ends



Figures 30: 3 months results for test case 6.

8.1.7 Test Case 7

Test Case ID: TC-07

Title: Event Cancellation

Objective: Verify that an admin or trainer can cancel events, and with that having users unable to sign up anymore to any occurrence under the canceled event.

Preconditions:

- The user is logged in as an admin or trainer.

Test steps:

- Click on one of the event cards, which opens a pop-over.



2. Click cancel event.

Expected Result:

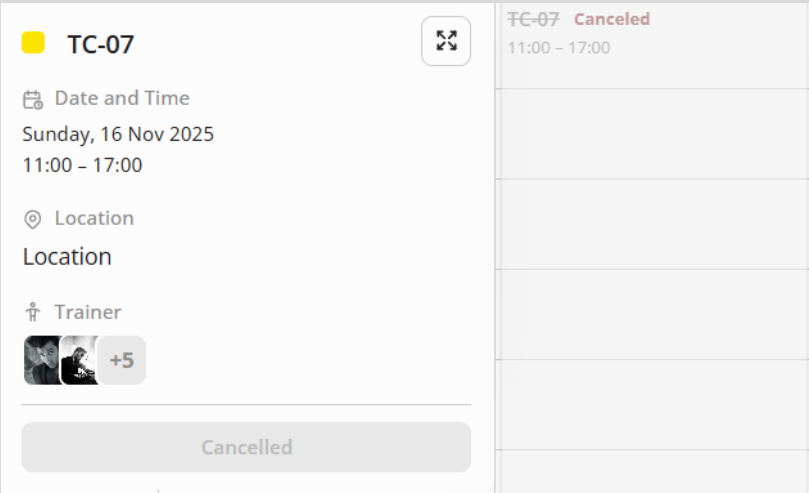
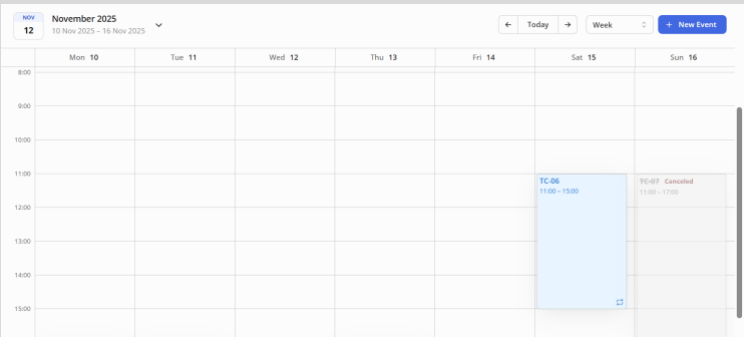
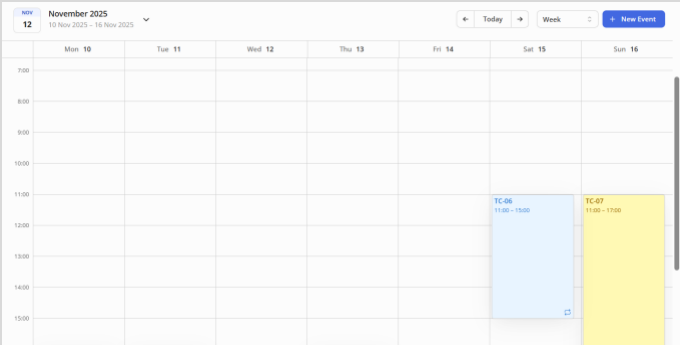
- The system will display the event as canceled.

Actual Result: Works as expected

Status: Pass

Related User Stories:

- Admin users can cancel events. **(Must have)**



Figures 31: Results for test case 7.

8.2 Summary of Test Cases

Test Case ID	4ey9ou8	Type	Result
TC-01	Event Creation	Functional	Pass
TC-02	Event Inspection (Fullscreen)	Functional	Pass
TC-03	Event Editing (Fullscreen)	Functional	Pass
TC-04	Recurrent Event Creation (Daily / Yearly)	Functional	Pass
TC-05	Recurrent Event Creation (Weekly)	Functional	Pass
TC-06	Recurrent Event Creation (Monthly)	Functional	Pass
TC-07	Event Cancelation	Functional	Pass

9-Evaluation

This section includes the individual contributions and the evaluation of the user stories that were defined at the beginning of the project. Therefore, it holds significant importance as it distinguishes between individual and group work, while also reflecting the progress of each user story.

9.1- Individual Contribution

Feature	Team members					
	Cenk	Zsombor	Muhammet	Omar	Kazi	Cosmin
-						
Type architecture	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>			
Database design	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>			
Data converters	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>			
Firestore rules		<input checked="" type="checkbox"/>				
SDK front-end fetching	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>			
Fetching logic		<input checked="" type="checkbox"/>				
Security and optimization		<input checked="" type="checkbox"/>				
Event creation	<input checked="" type="checkbox"/>				<input checked="" type="checkbox"/>	
Event cards			<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Event popover			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Fullscreen event edit, and viewing	<input checked="" type="checkbox"/>				<input checked="" type="checkbox"/>	
Event virtualization and recurrence	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	
Occurrence instantiation			<input checked="" type="checkbox"/>			

Event signups			✓			
Event capacity handling	✓		✓		✓	
Queuing system			✓			
Attendee handling	✓		✓		✓	
Event location map integration			✓			
Routing			✓			
Purging and code cleanup		✓	✓	✓	✓	✓
Bug testing	✓	✓	✓	✓	✓	✓
Report writing	✓	✓	✓	✓	✓	✓

9.2- Evaluation of User Stories

This section is about the evaluation of the user stories and which ones worked and which ones did not. It consists of 3 parts: user stories met, user stories removed, and user stories not met. In detail, each of these subsections are explained by why they are implemented and why not.

9.2.1 User Stories Met

This section outlines the user stories that were successfully implemented during the development of the project. Each story represents a functional requirement derived from stakeholder needs and system goals. Meeting these user stories demonstrates that the essential features of the calendar system (which includes event creation, management, customer interaction, and visualization) have been effectively realized. The focus was primarily on “**Must have**” and “**Should have**” requirements, ensuring that the core

functionalities of the scheduling and booking processes are stable, usable, and aligned with stakeholder expectations. These successfully met stories validate that the foundation of the system operates as intended and can support future enhancements.

- Admin users can plan and edit recurring events. **(Must have)**
- Admin users can plan and edit non-recurring events. **(Must have)**
- Admin users can assign or change hosts of an event. **(Must have)**
- Admin users can enable a queue for an event. **(Should have)**
- Admin users can change customer groups for events. **(Must have)**
- Admin users can sign up individual customers for an event. **(Should have)**
- Admin users can sign up customer groups for an event. **(Should have)**
- Admin users can make an event private. **(Must have)**
- Customers can see available events in the calendar. **(Must have)**
- Customers can see the events they joined. **(Must have)**
- Admin users can see specific events associated with specific admins. **(Must have)**
- Admin users can see all events in the calendar. **(Must have)**
- All users can see basic information about an event. **(Must have)**
- All users can see more detailed information about an event. **(Must have)**
- All users can differentiate events by color. **(Should have)**
- Customers to see queue status. **(Should have)**
- Customers to see if they joined the queue. **(Should have)**
- System to update the queue. **(Could have)**

9.2.2 User Stories Removed

This section presents the user stories that were intentionally removed from the scope of the project following discussions with the stakeholder. These features were deprioritized because they were either no longer relevant to the current business objectives, introduced unnecessary complexity, or were deferred to future development phases. Although these stories were not implemented, they are documented here to maintain transparency in the development process and to clearly show how project priorities evolved over time. Including these removed stories provides a comprehensive record of the decision-making process and helps justify scope adjustments throughout the project lifecycle.

- Admin users can select the payment model for an event. **(Won't have)**
- Admin users can see occupancy of all locations. **(Won't have)**
- Admin users can see occupancy of other admins. **(Won't have)**
- Customer to see host's picture. **(Won't have)**

9.2.3 User Stories Not Met

This section lists the user stories that were not completed within the project timeline due to time constraints, technical limitations, or workload distribution. While these requirements were not implemented in the current iteration, they were classified as lower priority (“**Could have**” or features associated with less critical epics, such as Public Booking Pages and Secondary Calendar Functions). Their absence does not affect the overall functionality or stability of the system’s core operations. These user stories can be considered for future development cycles once the main system components are fully stabilized and optimized.

- Admin users can specify a room for an event. (**Could have**)
- Admin users can tag categories for events. (**Could have**)
- Admin users to set the public booking page constraints. (**Could have**)
- Customers to book a timeslot with an admin who created a public booking page. (**Could have**)
- System to update the queue. (**Could have**)
- System to notify the customer of their queue movement. (**Could have**)
- System to notify the customer when they are running up the queue. (**Could have**)
- Admin users to set freeze time of the queue. (**Could have**)
- Admin users to set/edit opening hours of a location. (**Could have**)
- Rooms within the location to inherit opening hours. (**Could have**)
- Admin users to create/edit a room. (**Could have**)
- Admin users to mark a location unavailable. (**Could have**)
- Rooms to automatically become unavailable. (**Could have**)
- Admin users to create and edit customer groups. (**Could have**)

10- Reflection

This section outlines the challenges faced throughout the project and the potential future work that could extend the functionality already implemented by our team. These parts are important for analyzing the factors that held our team back and for reflecting on possible improvements for the future.

10.1 Challenges

Throughout the development of this project, our team encountered several challenges which influenced the pace and direction of our implementation. These challenges ranged from adapting to an existing and complex codebase to managing changing client requirements and design choices. The following subsections outline the key difficulties our group has faced and how we addressed them during the development process.

10.1.1 Working with an Existing Codebase

We believe the first challenge was that we had to hop into an already well created product and only create main functionalities. It was quite hard in the beginning to navigate through such a complex and vast code base, especially having the fact that most of us have never worked with React and we did not have the opportunity to choose environments. But after the first weeks, fixing some minor bugs and browsing through the code base, we finally got familiar with it.

10.1.2 Ambiguity in Type Specifications and Component Communication

We iterated extensively with the client to refine the type specification, clarifying how different components interacted and what information needed to be exchanged between them. The specification evolved continuously, as we adjusted fields to address newly identified constraints, whether uncovered during development or introduced by the client throughout the process.

10.1.3 Iterative Changes in Data Fetching Logic

In addition, signup fetching was constantly changed: where we first implemented fetching of all signups for all users. Then, we shifted to fetching signups only on popover open. Finally, we settled for fetching all the signups of the current customer, with conditional fetching for relevant signups on popover open.

10.1.4 Frequent Changes in UI Requirements

Another major challenge arose from the frequent design changes requested by the stakeholder for the event creation interface. The user interface (UI) of the calendar's event

creation page underwent several revisions throughout the development process, often in response to evolving client expectations and feedback. These changes included modifications to layout structure, input field arrangements, form flow, and the visibility of certain features.

10.1.5 Concurrency and Race Condition Handling

Lastly, time critical actions, particularly the signing up feature, involved intricate handling of race conditions. In this particular implementation, due to time constraints, we could not use cloud functions. Instead, we decided to use Firestore's transaction function that acts as a lock on the database when a signup is being written to the database.

10.2 Future Work

The requirements of the client have altered throughout the project and therefore, the stakeholder has removed the notification system and public booking page from the requirements due to time constraints. Therefore, for the future of this calendar application, a notification system and a public booking page can be implemented. Maybe as a viewing, several different themes (coloring for the display) can be made for the viewing of the calendar system or make it customizable for the person. Also, an implementation for adding an image option for the display of the event on calendar might well also be implemented.

Functionality wise, the calendar system already incorporates all the possible recurrent events, all the possible subsections in detail - such as allowed customers, queue limit, signed up customers, etc. However, additionally a tag for events, a specification for rooms inside locations can be implemented. Adjusting the occupancy of the rooms and setting up a limit for them can be also an option to extend.

11- References

- ISO/IEC/IEEE. (2021). *ISO/IEC/IEEE 29119-3:2021 – Software and systems engineering — Software testing — Part 3: Test documentation*. International Organization for Standardization.
- ISO/IEC 25002:2024. (2024). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality model overview and usage (Edition 1)*. International Organization for Standardization / International Electrotechnical Commission. <https://www.iso.org/standard/78175.html>
- NullSpace – All-in-one business management software for SMEs – <https://nullspace.cloud/>