

# **Design Project Report**

University of Twente

Supervisor: dr.ir. B.J. van der Zwaag (Berend-Jan)

Coordinator: dr.ir. R. Langerak (Rom)

April 18, 2025

## **Group 13 Members:**

Daniel Chitoraga s2987309

Martin Demirev s2965046

Dragos Erhan s2940124

Ion Tulei s2928787

Alexandru Verhovetchi s2958716

# **Data Analysis and Visualization of Project Room Occupancy in the University Library**

## **Project Proposal**

The University Library has 43 bookable project rooms via TimeEdit. The project rooms are also equipped with different types of sensors, such as presence sensors. With the new booking system TimeEdit, more data analysis and visualization opportunities can be created. The project rooms are a 'scarce resource' and are intensively used. Could a visualization of bookings and occupancy help students make better use of the project rooms?

In the project students will cooperate with Library and ICT departments on combining booking data with sensor data. In this project, students can work on sensor technology, data analysis, and creating a user interface. The goals would be to 1. Get a better understanding of the booking behavior of the project rooms in the Library and 2. Create a visualization of this data for both Library staff and students. Information gathered in this project will assist broader projects on occupancy-related projects on campus.

## **Abstract**

This report describes a system that analyzes and visualizes project room occupancy in the University of Twente's Vrijhof library. It combines booking data from TimeEdit with real-time sensor readings collected by sensors located in project rooms. The report begins with project planning and scope. It defines functional and non-functional requirements and outlines key use cases and stakeholders. A more in-depth project description follows, with an explanation of the realized server and client design. The dedicated testing chapters report results on the realized tests (unit, manual, system, etc.). Key modules, such as data ingestion, data processing, and user interface components, are described. Later chapters detail how the team has integrated TimeEdit and sensor data. The report concludes with lessons learned and suggestions for future enhancements.

## Table of Contents

<b>Abstract.....</b>	<b>3</b>
<b>Table of Contents.....</b>	<b>4</b>
<b>1. Project Description.....</b>	<b>7</b>
General Description.....	7
Data Sources.....	7
Backend.....	7
Database.....	7
Authorization.....	8
Frontend.....	8
MazeMap.....	8
Environment Variables.....	8
<b>2. Project Planning.....</b>	<b>9</b>
<b>3. Functional Requirements.....</b>	<b>11</b>
<b>4. Non-functional Requirements.....</b>	<b>13</b>
<b>5. User Stories.....</b>	<b>14</b>
<b>6. Metrics and Conventions.....</b>	<b>16</b>
<b>7. Use Cases.....</b>	<b>18</b>
Use Case 1: Real-Time Room Analysis.....	18
Use Case 2: Historical Booking Analysis.....	18
Use Case 3: Real-Time Room Monitoring.....	19
Use Case 4: Presence Detection and Discrepancy Identification.....	19
Use Case 5: Room Navigation.....	20
Use Case 6: Sensors Raw-Data Output.....	20
<b>8. Stakeholders.....</b>	<b>21</b>
Staff.....	21
Students.....	21
<b>9. Complications and Limitations.....</b>	<b>22</b>
Complications.....	22
Functional Requirements Dependencies.....	23
Non-functional Requirements Dependencies.....	23
Limitations.....	23
<b>10. Diagrams.....</b>	<b>25</b>
Database Design.....	25
Server Class Diagrams.....	27
Hardware Design Diagram.....	37
Sequence Diagrams.....	39

State Machine Diagrams.....	42
<b>11. Analysis Visuals List.....</b>	<b>45</b>
1. Booking Frequency By Start Time.....	45
2. Booking Frequency By Created Time.....	45
3. Booked and Unoccupied; Booked and Occupied; Unbooked and Unoccupied; Unbooked and Occupied + Unknown [Staff Only].....	45
4. Room Capacity vs TimeEdit Given People Count Discrepancies [Staff Only].....	46
5. Cancelled vs Not Cancelled Bookings.....	46
6. Booking Rate Visualisation.....	46
7. Booking Durations.....	46
8. Occupancy Rate Visualisation.....	47
9. Booking Lead Time.....	47
10. Real-time Occupancy Monitoring.....	47
11. Sensor Data Monitoring [Staff Only].....	47
12. Humidity/Temperature Monitoring - Extra from other students (implemented). 47	
13. Modified Bookings - Extra (not implemented).....	48
14. Under/normally/over-populated rooms [Staff Only] - Won't Do (complications) 48	
15. Sensor Data Discrepancies and Confidence [Staff Only] - Won't Do (complications).....	48
<b>12. API List.....</b>	<b>49</b>
I. General definitions.....	49
II. Blueprint /api/bookings (located in bookings_api.py).....	51
III. Blueprint /api (located in occupancy_api.py) & WebSocket events.....	54
<b>13. Test Schedule.....</b>	<b>59</b>
<b>14. Test Strategy.....</b>	<b>61</b>
1. Scope and Overview.....	61
2. Types of Testing.....	61
<b>15. Test Plan.....</b>	<b>63</b>
1. Unit Testing.....	63
2. Manual Testing.....	65
3. Integration Testing.....	67
4. User Acceptance Testing (UAT).....	68
5. System Testing.....	69
6. Performance Testing.....	71
7. Security Testing.....	72
<b>16. User Acceptance Testing Report.....</b>	<b>75</b>
1. Introduction.....	75

2. Objectives.....	75
3. Test Scope.....	75
4. Test Participants.....	77
5. The Questionnaire.....	77
6. Test Results & Key Findings.....	77
<b>17. Manual Testing Report - MT-02 Sensor Data.....</b>	<b>81</b>
1. Dependencies Generation and Database Insertion.....	81
2. Occupancy and Discrepancy APIs (occupancy_api.py).....	82
3. Occupancy and Discrepancy WebSockets (sockets.py).....	82
<b>18. Manual Testing Report - MT-03 TimeEdit.....</b>	<b>83</b>
1. Data Retrieval and Insertion (bookings_api.py and timeedit_retrieve_insert.py).....	83
2. Bookings APIs (bookings_api.py).....	84
3. Data Structuring (timeedit_structure.py).....	84
4. Data Grouping (timeedit_utils_analysis.py).....	85
5. Data Filtering (timeedit_filters.py).....	85
6. Data Analysis (timeedit_analysis.py).....	85
<b>19. External Interfaces Report.....</b>	<b>86</b>
1. MazeMap.....	86
2. TimeEdit.....	87
3. Sensors.....	87
<b>20. Authorization.....</b>	<b>88</b>
<b>21. Recommendations.....</b>	<b>89</b>
Comparative Graphs.....	89
Improved Occupancy Detection with Advanced Sensors.....	89
Parquet over JSON and CSV for data storage.....	90
Optimization.....	90
<b>22. Reflection.....</b>	<b>91</b>
I. Technical - Database Design.....	91
II. Project Experience and Teamwork.....	91
<b>23. Individual Contributions.....</b>	<b>93</b>
<b>Appendix.....</b>	<b>94</b>
1. Front End.....	94
2. Test results and visuals.....	97
3. Meeting Notes and Project Planning.....	129
<b>References.....</b>	<b>1</b>

## **1. Project Description**

The following section describes the main components of the occupancy monitoring system and how they work together.

### **General Description**

The goal of this project is to create a platform to monitor and visualize how students at the University of Twente use the Vrijhof library's project rooms. It combines bookings from TimeEdit with readings from sensors placed in project rooms to analyze and reveal occupancy patterns. The system aims to give library staff and students accurate insights into room usage, find discrepancies between reservations and actual presence, and guide users to the desired room via integrated map navigation.

### **Data Sources**

One of the main goals of the system is to perform data analysis regarding the occupancy of the project rooms in the Vrijhof library. The required data for the analysis can be categorized into booking data and occupancy data. The system ingests booking records from the TimeEdit API and sensor occupancy data from the university's sensor server. By aligning timestamps and filtering out non-working hours, the system creates a data set for historic and real-time data.

### **Backend**

The backend runs on Flask and uses SQLAlchemy models to map data into a PostgreSQL database. A background task polls real-time sensor and TimeEdit data and aggregates readings into hourly records. This data is then processed and served by REST endpoints, delivering such results as discrepancy summaries, (booking/occupancy) frequency counts, real-time room status, and others. The backend also defines a WebSocket channel used for live client updates via Flask-SocketIO.

### **Database**

The database schema contains multiple tables defined by SQLAlchemy models. The Room table stores identifiers, capacity, location, and geographic data. The Sensor table links each sensor to its room and records its type (ERS Eye or Nighthawk). The TimeEditBooking table holds one record per reservation, including timestamps, room identifiers, and status flags.

## **Authorization**

The application uses Google OAuth to restrict sensitive data to authorized personnel. The backend verifies the users' staff status before granting access to detailed analytics. Unauthorized users, such as UT students, are allowed to view real-time occupancy and status of the rooms alongside some analysis.

## **Frontend**

On the client side, a JavaScript application communicates via REST and Socket.IO with the backend. It opens a WebSocket connection to receive room occupancy updates as they occur, and it issues HTTP requests for user-given queries (filtering data by date, time, room, or sensor type). The interface renders interactive charts and analysis maps, and it makes use of the browser's local storage to cache results for optimized performance.

## **MazeMap**

The system integrates with the MazeMap API to generate navigation paths towards project rooms. MazeMap is also used for displaying color maps of rooms based on their booking and occupancy frequencies.

## **Environment Variables**

API credentials and code constants reside in environment variables and have to be specified by hand. This was done to keep sensitive data out of the source code and make it easier to modify important thresholds.



## 2. Project Planning

At the start of this project, our team conducted an introductory meeting with the clients, structured as an interview-style discussion. We followed a predefined list of questions (see [Appendix 3.1](#)) to gain a deeper understanding of their project proposal, objectives, and expectations. This initial meeting also served to establish a framework for future collaboration - we agreed to hold weekly meetings to ensure consistent progress and alignment.

During these weekly sessions, we:

- Addressed new questions and clarifications,
- Documented key insights and client feedback,
- Presented our progress and discussed challenges,
- Brainstormed new ideas and solutions.

To maintain organization, we created a dedicated meeting document for each session, recording all suggestions, answers, and action items. An example of such documentation can be found in [Appendix 3.2](#).

### Team Organization and Workflow

For seamless communication, we utilized Discord, structuring it with:

- A general channel for important updates and announcements,
- A dedicated channel for meeting schedules and details,
- A repository channel for storing essential documents and resources.

Task allocation and progress tracking were managed via Trello, where we:

- Assigned individual task lists to each team member,
- Outlined weekly objectives to ensure steady progress,
- Maintained a master planning board with overarching milestones.

See [Appendix 3.3](#) for an example of weekly task allocation in Trello.

For code management, our team has used GitHub following the standards described in the Metrics & Conventions section.

## Project Phases and Timeline

Our project followed a structured timeline, divided into distinct phases:

### Week 1 – Group Formation & Requirements Elicitation

- Established team roles and responsibilities.
- Conducted initial client discussions to define project scope.

### Weeks 2–4 – Design Phase

- Explored design choices, including UI/UX prototypes.
- Developed system diagrams and mockups.
- Conducted preliminary acceptance testing.

### Weeks 5–7 – Implementation

- Focused on backend and frontend development.
- Integrated MazeMap functionality.

### Weeks 8–9 – Integration & Testing

- Merged frontend and backend components.
- Addressed additional ("COULD DO") requirements.
- Performed comprehensive system and security testing.

### Week 10 – Report Finalization

- Refined the final report.
- Added reflective analysis on project outcomes and learnings.

See [Appendix 3.4](#) for general planning in Trello.

## Conclusion

Through structured planning, consistent client communication, and agile task management, our team completed the project from initial requirements gathering to final implementation and reporting. The iterative feedback process and well-defined workflows ensured alignment with client expectations while allowing flexibility for adjustments as needed.

3. Functional Requirements	MoSCoW Priority
1. Analyze booking behavior of <b>historic</b> data and provide insights (e.g., frequency of bookings, peak usage times) after a user's query.	M
2. Provide Library staff with advanced visualizations, based on <b>sensor</b> and <b>TimeEdit booking data</b> in <b>real-time</b> , to monitor room usage (e.g., analysis maps, trend charts).	M
3. Implement presence detection by evaluating the data collected from the sensors.	M
4. Correlate sensor data and TimeEdit booking data to identify discrepancies. (booked and occupied, booked and unoccupied, unbooked and occupied, unbooked and unoccupied).	M
5. Provide filtering options to simplify data exploration and visualization (e.g., by room size, time slot, occupancy status).	S
6. Implement navigation to a room (using MazeMap).	S
7. Send reminders to users by email about their upcoming bookings.	C
8. Count the number of people in a room using sensors (or integrate an existing project if applicable).	C
9. Record and analyze if and when the number of people using a room matches its capacity: underpopulated/normal/overpopulated.	C

10. Notify students if a room becomes available due to cancellations or no-shows when everything is fully booked (within a reasonable time range).	W
11. Detect overstays if the occupancy duration is larger than the booking duration.	W
12. Notify students when their count in a room deviates (significantly) from the room capacity, and inform them about the availability of rooms with a more suitable capacity that they can move to.	W

4. Non-functional Requirements	MoSCoW Priority
1. The system must be scalable to accommodate a growing number of users, sensors, and bookings without performance degradation.	M
2. The system must ensure data privacy and comply with GDPR when handling booking and sensor data.	M
3. The system should have an uptime of at least 99.5% to ensure availability for library staff and students.	M
4. The system should have an intuitive and user-friendly UI (receive a score of 4.5 out of 5 on a user-friendliness survey), ensuring ease of navigation for both students and library staff.	M
5. The system should be designed in a modular way to allow future expansion, such as additional analytics features or AI-based predictions.	M
6. The dashboard should update in real-time (or near real-time) to reflect the latest occupancy and booking status.	S
7. The system must be compatible with existing infrastructure, including TimeEdit and the sensor network.	S
8. System performance should allow room occupancy updates to be processed within 5 seconds of data collection.	S
9. The system should be accessible on multiple devices (desktop, tablet, mobile) with a responsive design.	S
10. The system should provide role-based access control, ensuring that only authorized users can access specific data (e.g., only staff can view full room usage history).	C

User	5. User Stories	MoSCoW Priority
Staff	1. As a staff member, I want to view room usage trends derived from historical data, including peak usage times and occupancy rates, so that I can schedule maintenance during the least impactful periods and adjust opening hours. I should be able to access reports within 10 seconds of query submission.	M
Staff	2. As a staff member, I want to see a live dashboard displaying the real-time booking and occupancy status of all rooms, updating every 5 seconds so that I can monitor room utilization efficiently.	M
Staff	3. As a staff member, I want to see a list of the most frequently booked/used rooms, segmented by features (e.g., capacity, room type) so that I can identify which attributes contribute to room preference. The list should be available within 10 seconds of query submission.	M
Staff	4. As a staff member, I want to see a list of the most frequently booked/used rooms, segmented by features (e.g., capacity, room type) so that I can look into any issues with those rooms (e.g., malfunctioning screens, heaters, etc.). The list should be available within 10 seconds of query submission.	M
Staff	5. As a staff member, I want to access real-time data on the occupancy status of any booked room through sensor readings, updating at least every 10 seconds, so I can check if the room is used appropriately based on booking details.	M
Staff	6. As a staff member, I want to view advanced visualizations, such as analysis maps and trend charts, that display real-time room usage and booking data, updating every 5 seconds, so I can efficiently monitor occupancy.	S
Staff	7. As a staff member, I want to generate a weekly summary showing the percentage of booked rooms that were not occupied, so I can assess booking reliability and adjust policies if needed.	S
Staff	8. As a staff member, I want to see the number of people present in each room when it was booked and/or unbooked, calculated over a selectable time range so that I can identify underutilized or overcrowded rooms.	C

Student	9. As a student, I want to receive an automated email reminder at least 24 hours before my booking starts so I do not forget my reservation.	C
Student	10. As a student, I want to access booking behavior insights, such as the time in advance that students make reservations, so that I can plan my bookings more effectively. This data should be updated weekly.	C
Student	11. As a student, I want to receive a notification within 1 minute if a room matching my criteria (time, capacity) becomes available due to a cancellation, so that I can book it before someone else does.	W
All	12. As a user, I want to see a graph showing the distribution of booking times in relation to the start time (e.g., how far in advance most users book) so I can understand booking patterns. This should be accessible on demand.	S
All	13. As a user, I want to access a live occupancy status view of all rooms, updating every 10 seconds, so that I can easily find an available study space.	C
All	14. As a user, I want to receive step-by-step navigation instructions via MazeMap to my booked room within 10 seconds of request submission so that I can locate it easily.	C

Metric Title	6. Metrics and Conventions
1. <b>Response Time</b>	Duration between the collection, processing and display of data on the system interface. The system must ensure that this duration does not exceed <b>10 seconds</b> .
2. <b>Uptime</b>	Percentage of time the system is operational and available for use over a specified period. The system must maintain an uptime of at least <b>99.5%</b> to ensure high availability for users.
3. <b>Scalability</b>	The system's ability to accommodate an increasing number of users, sensors, and bookings without experiencing performance degradation. The system should be designed to scale efficiently for at least <b>500 users</b> .
4. <b>Discrepancy Detection Accuracy</b>	The system's ability to correctly identify inconsistencies, such as rooms that are booked but unoccupied or unbooked but occupied. The system must achieve an accuracy of at least <b>90%</b> in detecting such discrepancies.
5. <b>User Satisfaction</b>	The measurement is conducted through periodic surveys to assess users' perceptions of the system's interface, responsiveness, and features. The target satisfaction score is at least <b>4 out of 5</b> , where 1 is not user-friendly and responsive, and 5 is a completely intuitive platform.
6. <b>Task Success Rate</b>	Percentage of users who can complete key tasks (e.g., viewing occupancy status) without requiring assistance. The system must achieve a task success rate of at least <b>95%</b> .
7. <b>Error Rate</b>	Frequency at which users encounter errors (e.g., incorrect navigation or input mistakes) while interacting with the system. The system must maintain an error rate of less than <b>5%</b> .
8. <b>Data Privacy</b>	Ensure compliance with GDPR and other relevant data protection regulations.



9. <b>Access Control</b>	Implement role-based access control (RBAC) to ensure that only authorized users can access sensitive data.
10. <b>Endpoint Compliance Rate</b>	<p>The percentage of API endpoints that meet all of the following checks in a single test run:</p> <ul style="list-style-type: none"><li>• Return the correct HTTP status code</li><li>• Produce a response matching the expected data schema</li><li>• Format empty responses correctly (i.e. {} rather than an error)</li></ul>
1. <b>Coding Standards</b>	Follow consistent coding conventions (e.g., naming conventions, indentation, commenting) to ensure maintainability and readability of the codebase.
2. <b>API Conventions</b>	Use RESTful API design principles for all system integrations (e.g., with TimeEdit, MazeMap). Ensure APIs are well-documented and versioned.
3. <b>UI/UX Conventions</b>	<p>Adhere to established UI/UX design principles, such as:</p> <ul style="list-style-type: none"><li>A. Consistent navigation and layout across all pages.</li><li>B. Use of intuitive icons and labels.</li><li>C. Responsive design for compatibility with desktop, tablet, and mobile devices.</li></ul>
4. <b>Data Formatting</b>	Use standardized date/time formats (e.g., ISO 8601) and consistent units of measurement (e.g., capacity in number of people, time in 24-hour format).
5. <b>Git Conventions</b>	Create branches for each feature or bug fix, with descriptive names, and use clear committing messages.

## 7. Use Cases

This section presents the user flow during the diverse cases of application practical usage.

### Use Case 1: Real-Time Room Analysis

**Description:** UT staff and students can inspect the state of the rooms in real-time, be able to sort the list and filter by its parameters.

**Scenario:**

1. Click on the “Lists => Rooms” section in the navigation bar.
2. See the list of the rooms and select the needed subsection (all, available or booked).
3. Apply any available filters or sorting to see particular rooms.

**Result:** Stakeholders can easily see and interact with all project rooms inside the library.

### Use Case 2: Historical Booking Analysis

**Description:** Library staff can analyze historical booking data to identify trends, such as peak usage times and frequency of bookings, to optimize room scheduling and maintenance.

**Scenario:**

1. Staff logs into the system and navigates to the "Dashboard" or any “Analytics => Bookings” section.
2. Chooses a time range and room(s) for analysis.
3. Views visualizations such as line charts, bar graphs, pie charts and analysis maps showing booking frequency and peak usage times.
4. Uses the insights to schedule maintenance during low-usage periods or adjust opening hours.

**Result:** Staff can make data-driven decisions to improve room utilization and maintenance scheduling.

### **Use Case 3: Real-Time Room Monitoring**

**Description:** Library staff can monitor room occupancy and booking status in real-time using advanced visualizations like analysis maps and trend charts.

**Scenario:**

1. Staff logs into the system and navigates to the "Dashboard" or any "Analytics => Occupancy" section.
2. View an analysis map and other trending charts showing the real-time occupancy status of all rooms.
3. Applies filters to see detailed information, including current occupancy and booking status.
4. Uses the information to manage room availability and address any discrepancies.

**Result:** Staff can make data-driven decisions to improve room utilization and maintenance scheduling.

### **Use Case 4: Presence Detection and Discrepancy Identification**

**Description:** The system uses sensor data to detect room occupancy and identify discrepancies between booked and actual room usage.

**Scenario:**

1. Sensors continuously collect data on room occupancy and send it to the system.
2. The system correlates sensor data with booking data to identify discrepancies (e.g., booked but unoccupied, unbooked but occupied).
3. Staff logs into the system and navigates to the "Dashboard" or any "Analytics => Discrepancies" section.
4. Staff can monitor for any discrepancies and take appropriate action, such as investigating no-shows or unauthorized usage.

**Result:** Improved accuracy in room usage tracking and better management of room resources.

### **Use Case 5: Room Navigation**

**Description:** Students can use the system to navigate to their booked rooms using integrated navigation tools like MazeMap.

**Scenario:**

1. A student navigates to the “Navigation” section.
2. Selects a booked room in the search select input.
3. The system integrates with MazeMap to provide step-by-step navigation instructions to the room.
4. The student follows the instructions to find the room easily.

**Result:** Students can efficiently locate their booked rooms, enhancing their overall experience.

### **Use Case 6: Sensors Raw-Data Output**

**Description:** Authorized staff members can monitor in real-time the output of sensors in a table format.

**Scenario:**

1. A staff member logs into the system.
2. Click on the “Lists => Sensors” section in the navigation bar.
3. The member can easily see all the details related to the room's humidity, temperature, and occupancy, synchronized with the sensor ID and the room ID.
4. Staff can apply filters and/or sorting for relevant results.

**Result:** Staff can easily see the raw output from the sensors in real-time.

## **8. Stakeholders**

This section discusses the roles, responsibilities, requirements, and concerns of the project's stakeholders: university staff and students.

### **Staff**

The library staff oversees room management and resource allocation, ensuring the efficient use of study spaces. They also monitor room occupancy and optimize resources based on booking trends.

The staff analyzes booking and occupancy data through TimeEdit and sensors to identify room usage patterns. They also maintain system functionality, address booking discrepancies, and ensure compliance with institutional policies.

The staff requires accurate, real-time room occupancy data, advanced visualizations, and historical analytics for decision-making. They are concerned with system reliability, scalability, GDPR compliance, and the accuracy of sensor data in detecting presence and people count.

### **Students**

Students are the primary users of the TimeEdit booking system and are responsible for reserving, occupying, and vacating study rooms. They rely on TimeEdit data to make informed decisions about room availability.

Students book rooms according to their study needs but do not always adhere to occupancy rules or update their bookings if plans change. Sometimes they are forced to use rooms that do not fit their needs due to the unavailability of other, more suitable ones, or TimeEdit limitations, or, in the worst case, have no room to use at all due to full booking.

TimeEdit does not provide information on trends such as most frequently occupied or free study spaces, causing less informative student choices. Students need a user-friendly system with real-time occupancy updates, trend analyses and visualizations, room navigation, booking reminders, and other notifications.

## 9. Complications and Limitations

This section presents the potential technical and conceptual limitations and complications of the project.

### Complications

These logical complications imply that implementing the listed features cannot happen without assumptions, eroding their usefulness in providing meaningful data. Hence, the related requirements will likely not be realized.

1. **Overstay detection:** Detect if anyone overstays their booking duration. An overstay is determined if a room has not been empty after people have been present during the booking duration. Limitations:
  - a. If a room is booked, but people who have not made the booking occupy it instead (over the booking duration), it is not an overstay but a no-show.
  - b. It must be determined what time after the booking time frame ends is reasonable to be considered as an overstay.
2. **Delays:** Mark a booking as delayed when there is no presence after the booking time frame begins. Limitations:
  - a. It must be determined what time after the booking time frame begins is reasonable to be considered a delay, e.g., 15 minutes.
  - b. It could be that people who have not made the booking occupy the room.
3. **Availability notifications:** Send notifications to students who have not been able to book a room because everything is full. The only way to implement this is by detecting that they have been active on TimeEdit but have not booked anything due to the unavailability of all project rooms. Hence, this depends on:
  - a. If TimeEdit allows activity monitoring;
  - b. If we have access to user emails;
  - c. If they intended to book a room and were not just observing.
4. **TimeEdit rigged data points:** Upon observation of TimeEdit “People Count” data, several unrealistic group sizes were encountered, e.g., a group size of 100 (ref. the screenshot below). Hence, these rigged data points are modified to be 0 for negative inputs (very large positive integers convert to negative) and 100 for large positive inputs. These data points are included in the discrepancies’ bar and pie charts, but marked as “Invalid” in the discrepancies table.

## Functional Requirements Dependencies

Please note that the W requirements are excluded due to the [Complications](#).

TimeEdit: Requirements 1, 2, 4, 5, 7, 8

Sensors: Requirements 2, 3, 4, 6, 7

MazeMap: Requirements 9

## Non-functional Requirements Dependencies

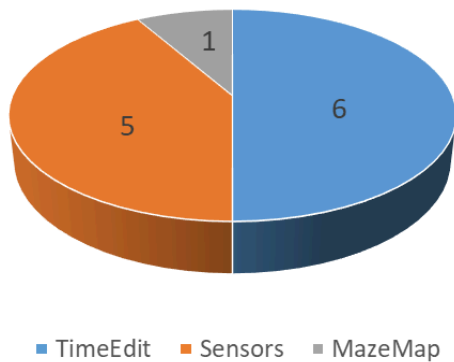
TimeEdit: Requirements 3, 6, 7

Sensors: Requirements 3, 6, 7, 8

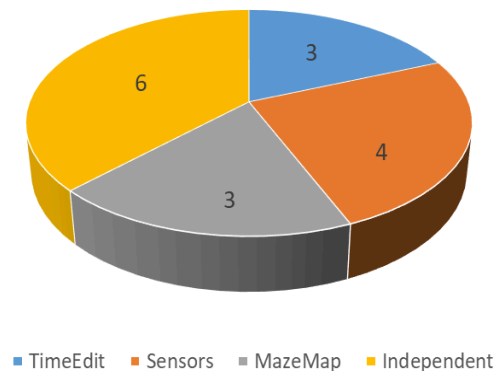
MazeMap: Requirements 3, 6, 7

Independent: Requirements 1, 2, 4, 5, 9, 10

Count of Functional Requirements Dependencies



Count of Non-functional Requirements Dependencies



## Limitations

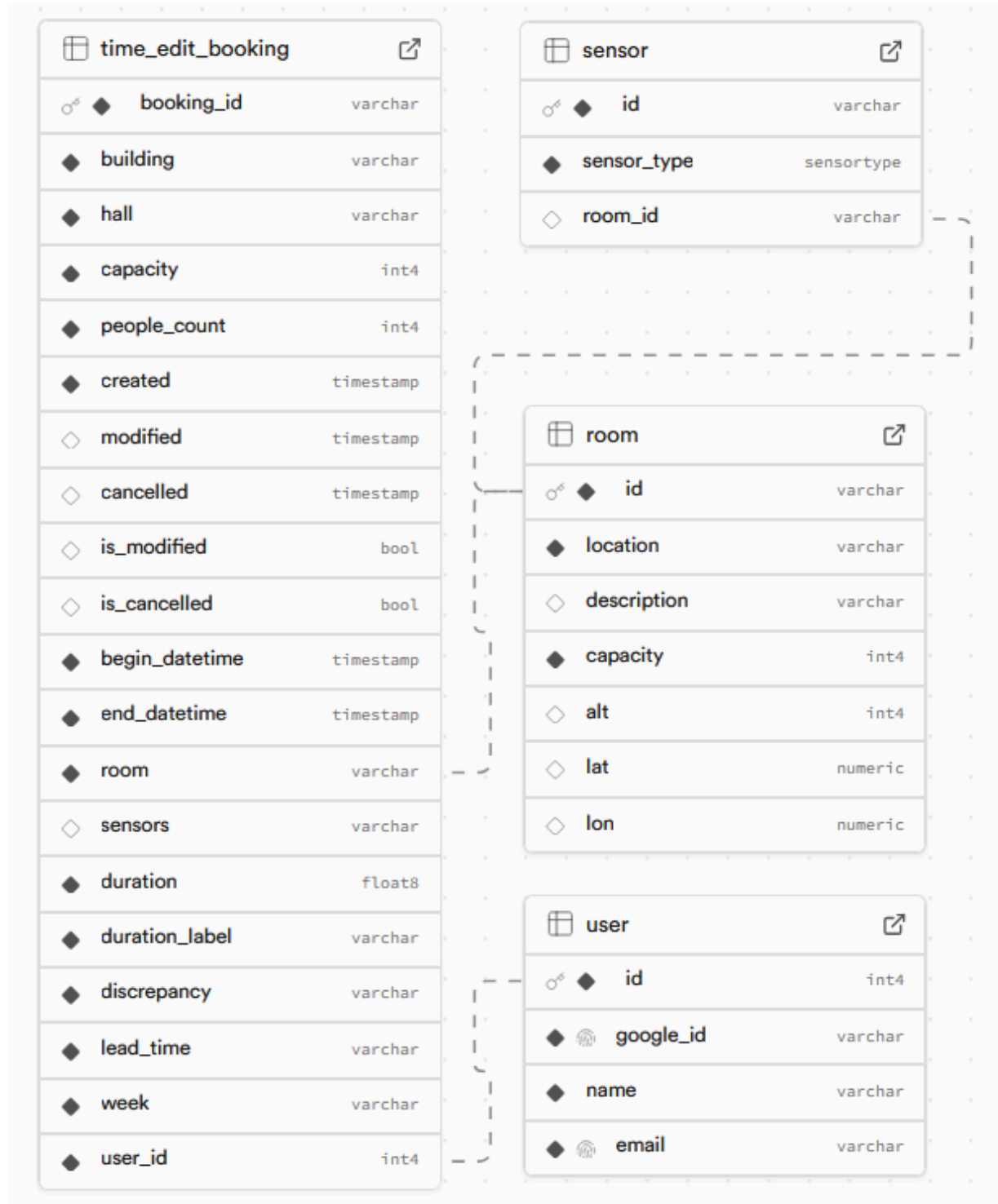
1. **Users:** Since user information, such as an email address, is not included in the TimeEdit API data, the User database table will contain only fake data.
2. **Notifications:** Since no real user data is available and handling emails raises privacy concerns, we will likely not be able to and have permission to work with them.

3. **People Count** (from sensors): While the TimeEdit “People Count” field is always present in data, the People Count by sensors is not. Neither the Nighthawk, nor the ErsEye sensors provide data about it. Hence, we will likely not be able to implement the requirements related to it too.



## 10. Diagrams

### Database Design



The system comprises several key components that work together to manage room bookings, track sensor data, monitor room occupancy, and send notifications to users. The main entities in the system are Sensor, Room, User, OccupancyHours, TimeEditBooking, and Notification.

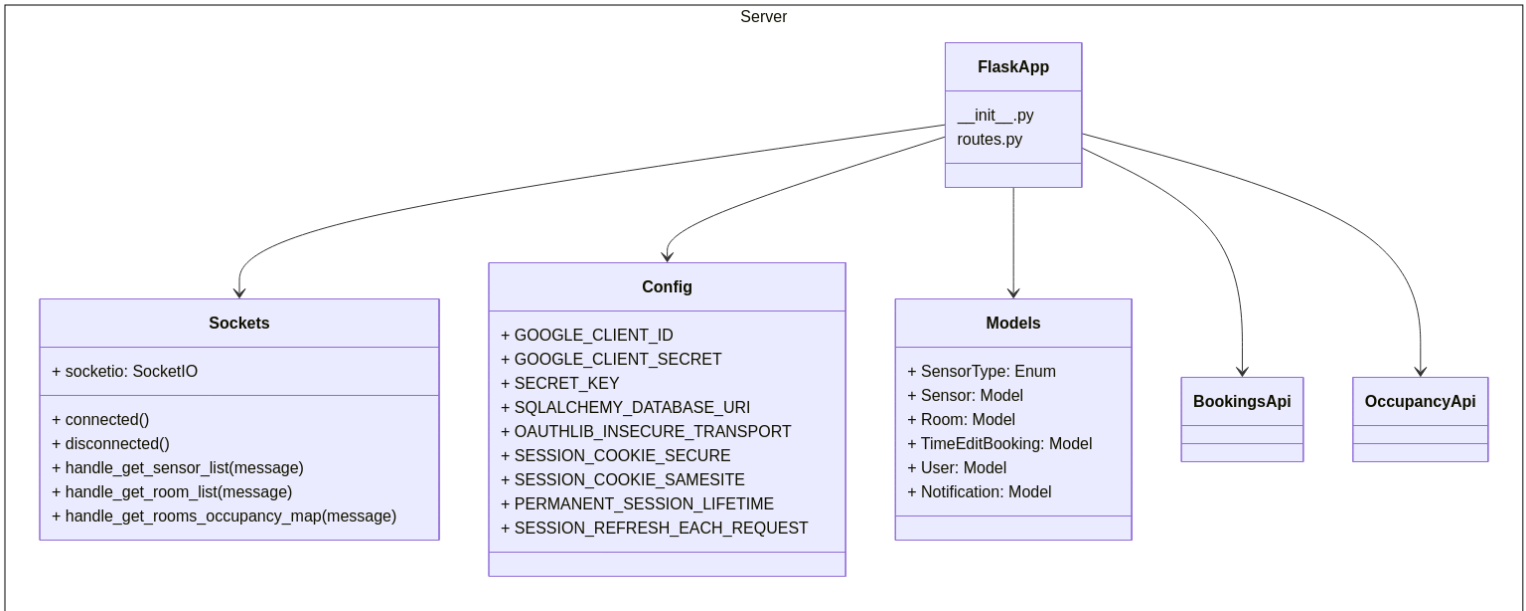
**Sensor Table:** Represents a physical sensor device installed in a room. Each sensor has a unique SensorID and is associated with a specific room via RoomID.

**Room Table:** Represents a physical room that users can book (available on TimeEdit). Each room has a unique RoomID, Location, Capacity, positioning attributes, and Description. Rooms are linked to sensors and bookings.

**User Table:** Represents a mock class of the University of Twente individuals who can book rooms and receive notifications. Each user has a unique UserID, along with Email, Password, and Role (Student, Staff). The table contains only “fake” user data due to [limitations](#).

**TimeEditBooking Table:** This represents a booking made by a user for a specific room. This information is retrieved from TimeEdit and contains already structured data instead of raw TimeEdit API data.

## Server Class Diagrams



This class diagram showcases the structure of the classes in our server implementation written in [Flask](#). It encapsulates all the backend components that manage data polling, processing, and communication via REST endpoints and WebSockets. Below is a detailed explanation of each class.

### 1. FlaskApp

This is the core of the server. Technically it is represented by two files: `__init__.py` and `routes.py`. **FlaskApp** creates the Flask application instance, loads the configuration, and initializes extensions (database, login manager, CORS, Flask-SocketIO). It also registers the two blueprints - **BookingsApi** under `/api/bookings` and **OccupancyApi** under `/api/`.

### 2. Config

This class defines constant variables used for Google OAuth (Google sign-in), session settings, and others.

### 3. Models

Here we define our SQLAlchemy ORM layer:

- **SensorType** is a Python **Enum** for the two sensor types: ERS Eye and Nighthawk.
- **Room** contains data about rooms: name, location, capacity, geographical coordinates. It is linked with the **Sensor** table through the **sensors** relationship.
- **Sensor** contains a listing of active sensors associated with rooms.

- **TimeEditBooking** contains booking records, linking back to the **Room** and **User** tables.
- **User** table holds Google profile data used for Google OAuth.
- **Notification** table was supposed to contain data about user notifications. Due to time constraints, this was never used.

#### 4. BookingsApi

This blueprint handles all booking-related data under **/api/bookings/**. See below for a more detailed description.

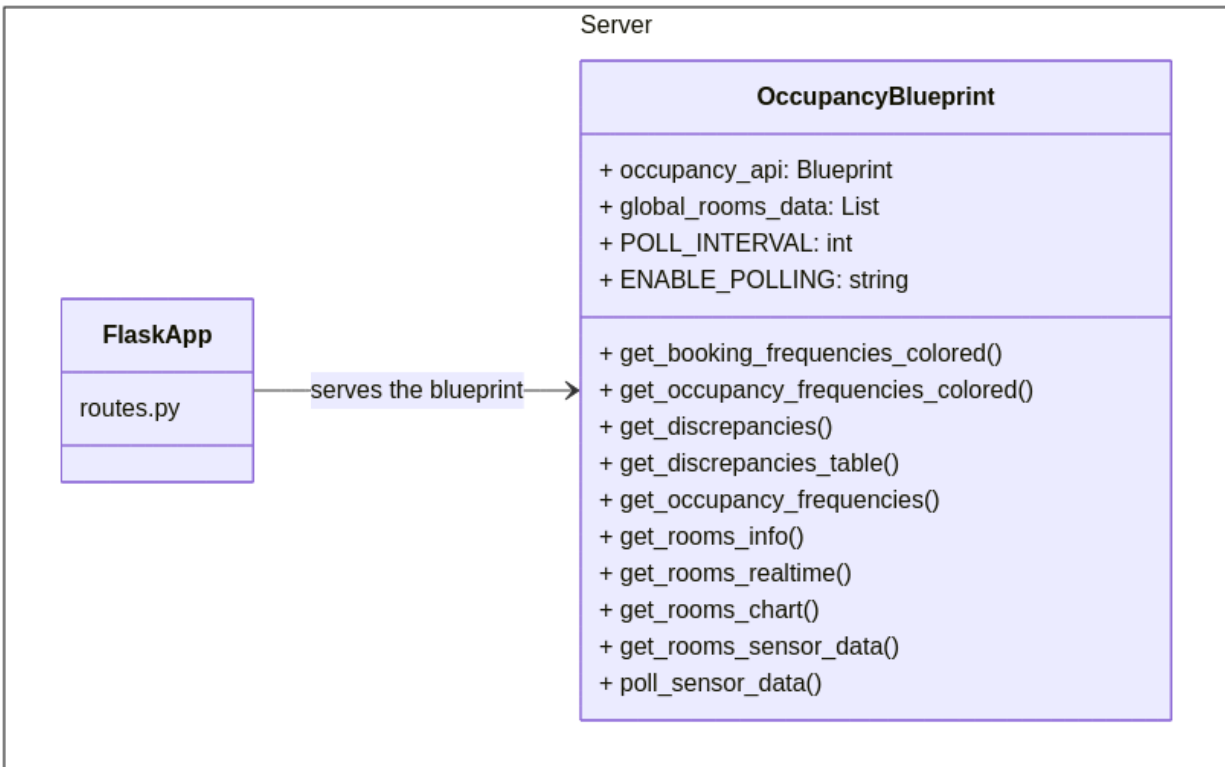
#### 5. OccupancyApi

This blueprint serves under **/api** and covers room occupancy, discrepancies, and room color mappings. See below for a more detailed description.

#### 6. Sockets

This class uses Flask-SocketIO to add a WebSocket layer alongside HTTP. It listens for client connections and disconnections. It also defines three events - **getSensorList**, **getRoomList**, and **getRoomsOccupancyMap** - each of which parses a query string, applies the same filters as the HTTP endpoints, and returns JSON data directly over the socket connection.

### Occupancy Blueprint



The **OccupancyBlueprint** (*occupancy\_api.py*) handles routes and tasks that deal with room occupancy, discrepancies, and room color mappings. It creates a Flask Blueprint called **occupancy\_api** (used by *routes.py*) and reads environment settings:

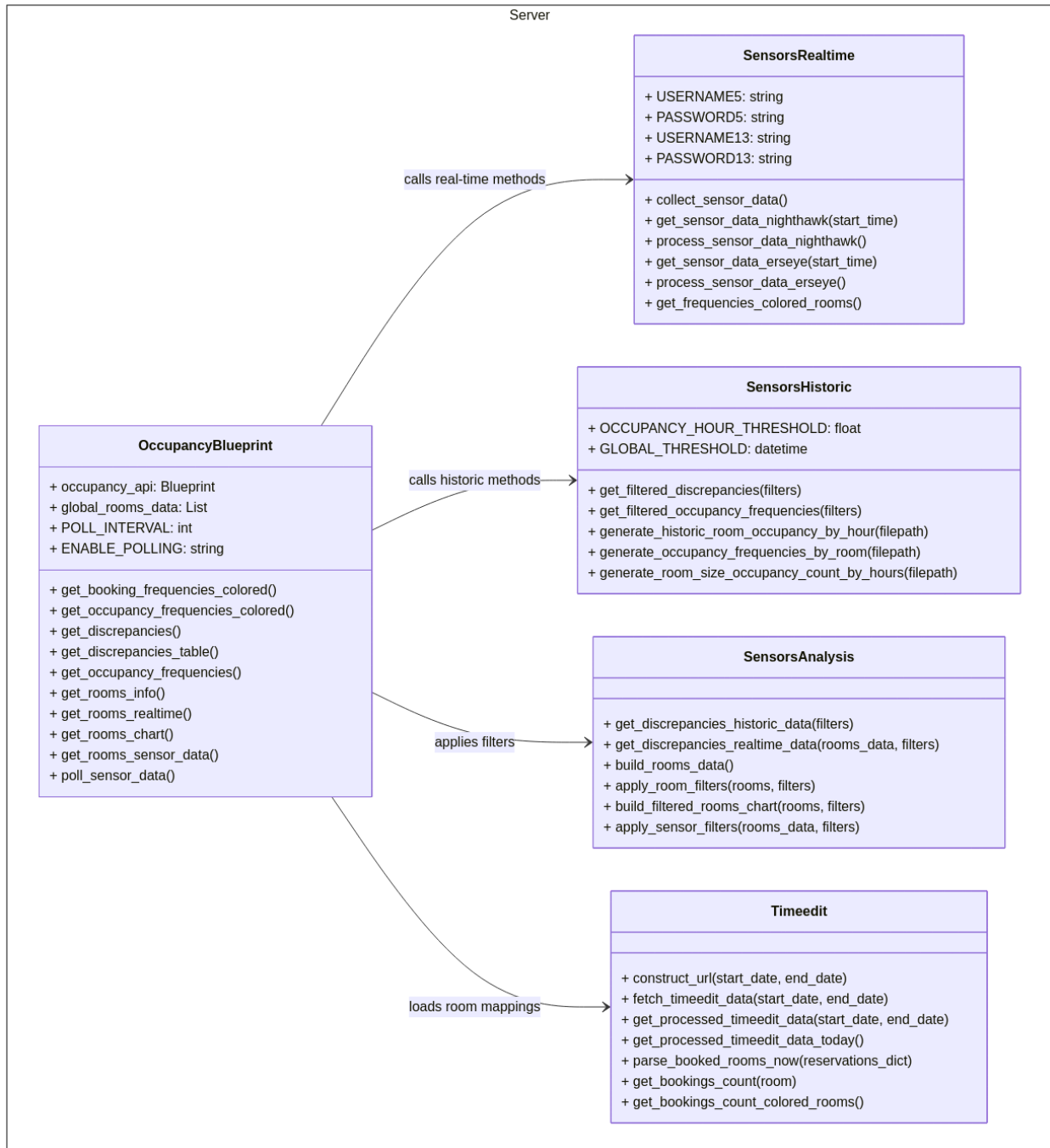
- **POLL\_INTERVAL**: how often to refresh the in-memory cache (real-time sensor data)
- **ENABLE\_POLLING**: whether to start a background thread for polling or not

The cache is a simple list named **global\_rooms\_data**. A function called **poll\_sensor\_data()** runs in its own thread (when polling is enabled), calls **build\_rooms\_data()**, updates **global\_rooms\_data**, then sleeps for the next interval.

All of these HTTP endpoints are served by this blueprint:

1. **Color codes** for MazeMap views (*/booking-frequencies/colorcodes/*, */occupancy-frequencies/colorcodes/*)
2. **Discrepancies** endpoints (*/discrepancies/* and */discrepancies/table/*)
3. **Occupancy frequencies** (*/occupancy-frequencies/*)
4. **Room data** for stati info, real-time status, charting, and sensor tables (*/rooms/info/*, */rooms/real-time/*, */rooms/table/*, */sensors/table/*)

Each endpoint reads query parameters, runs them through common filter checks, and then returns a JSON result.



There are four key modules **OccupancyBlueprint** makes use of.

### 1. **SensorsRealtime** (*sensors\_realtime.py*)

This module polls the university's sensor ingest service for the latest ERS Eye and Nighthawk readings. It defines:

- HTTP calls to each sensor endpoint
- ***process\_sensor\_data\_{sensorname}*** to process exported data

- ***collect\_sensor\_data*** to combine both sensor data into a single room-indexed dict
- A color-coding helper (***get\_frequencies\_colored\_rooms***) that maps historic occupancy counts to hex colors.

## 2. **SensorsHistoric** (***sensors\_historic.py***)

This module works with stored JSON files of historic sensor dumps. It:

- Loads / generates hourly room occupancy
- Filters data by date, time, and ISO week (YYYY-Www)
- Processes and analyses data for discrepancies and frequencies

## 3. **SensorsAnalysis** (***sensors\_analysis.py***)

This module merges sensor data with TimeEdit bookings and defines helper functions used by the **OccupancyBlueprint**. It:

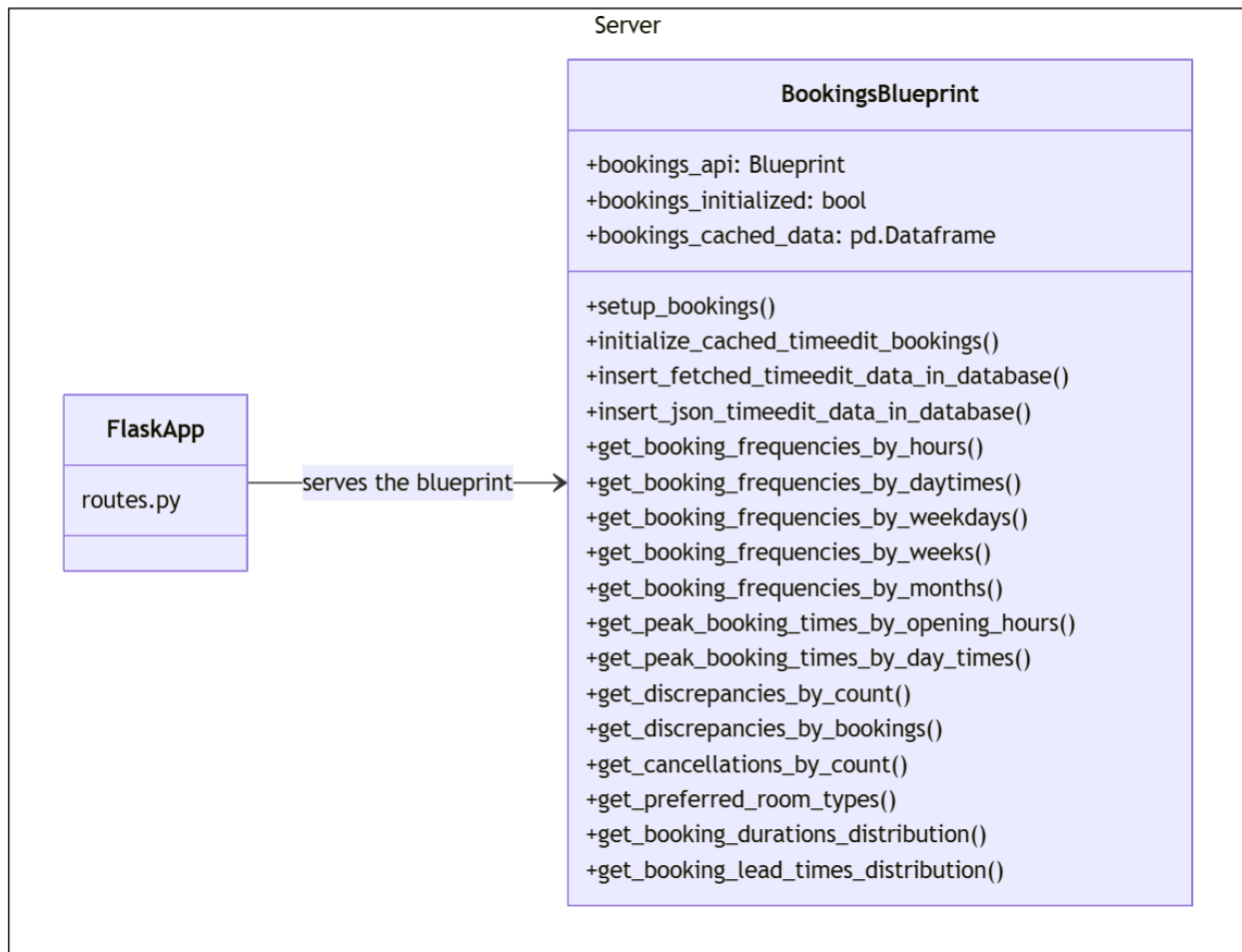
- Builds a list of real-time room occupancy and booking status
- Builds discrepancies lists based on historic sensor data and historic TimeEdit bookings
- Defines filtering functions for the occupancy and discrepancies endpoints (***apply\_room\_filters***, ***apply\_sensor\_filters***)

## 4. **Timeedit** (***timeedit.py***)

This module fetches and decodes TimeEdit reservation data. It defines:

- ***get\_processed\_timeedit\_data\_today*** - to retrieve bookings from TimeEdit for today
- ***parse\_booked\_rooms\_now*** - to find the currently booked rooms

## Bookings Blueprint



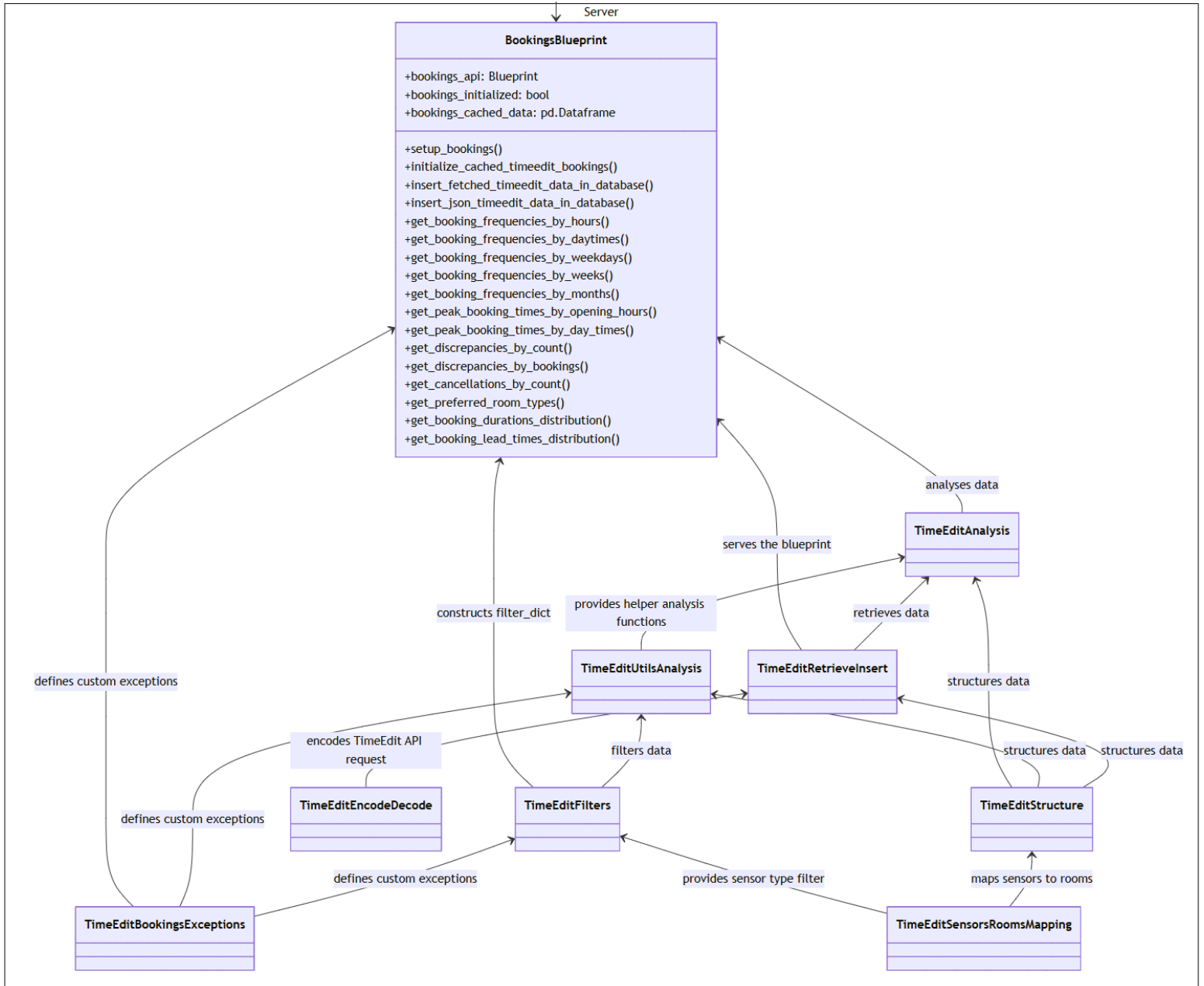
This diagram shows the structure and responsibilities of the **server-side components** of a Flask-based web application handling booking data. The **FlaskApp** serves this blueprint and manages routing. It includes a dictionary of routes and simple connection methods.

At the center is the **BookingsBlueprint** class, which manages everything related to bookings:

- The **setup\_bookings()** and **initialize\_cached\_timeedit\_bookings()** methods handle the **caching mechanism** using a Parquet file. They load this data into `bookings_cached_data` and set `bookings_initialized` to True. This caching ensures that data is only loaded once when the server starts, as Flask doesn't natively support a one-time-on-startup execution, so it's handled via a workaround in `before_app_request`.



- The following functions - **insert\_fetched\_timeedit\_data\_in\_database()** and **insert\_json\_timeedit\_data\_in\_database()**, are responsible for inserting data into the database. This can be done either by fetching from the **TimeEdit API** or by reading an existing **parquet file**.
- The rest of the methods (all beginning with **get\_**) correspond to **API endpoint functions**. These endpoints provide booking-related insights and statistics, such as frequency distributions, cancellation counts, preferred room types, discrepancies, and lead times.



## 1. BookingsBlueprint

This is the main class that defines the Flask Blueprint for booking-related API endpoints as presented previously.

## 2. TimeEditRetrieveInsert

Handles retrieving and inserting data from the TimeEdit API and cached Parquet files.

- Acts as a data gateway for the rest of the system.
- It is mainly related to the TimeEditBooking database table, but that can be easily extended to cover other tables in the future.
- Provides data to both analysis and caching components.

### **3. TimeEditAnalysis**

Responsible for analyzing booking data, such as processing frequencies, durations, lead times, discrepancies, etc.

- The 'main' file used by the BookingsBlueprint to process data before returning it via API endpoints.

### **4. TimeEditUtilsAnalysis**

Offers helper functions to support TimeEditAnalysis, mainly with data grouping either by a time key and unit or a non-time key. The method **get\_grouped\_data()** is the core of the TimeEditBooking analysis.

### **5. TimeEditFilters**

Constructs and applies filters to booking data based on user-provided query parameters.

- Plays a key role in creating the filter\_dict used by endpoints.
- Connects directly to exceptions (e.g., when a combination of filters causes empty data) and utility classes.

### **6. TimeEditEncodeDecode**

Encodes requests to the TimeEdit API. It can also decode TimeEdit API request URLs, but that functionality is not utilized.

### **7. TimeEditStructure**

Structure raw data from TimeEdit into usable formats - formatting or parsing raw booking entries.

- Used in both retrieval and analysis flows.
- Supports readable and structured interpretation of raw booking data.

## 8. TimeEditSensorsRoomsMapping

Maps sensor types to rooms. In the context of the BookingBlueprint, it is only meaningful for the [sensorType filter](#).

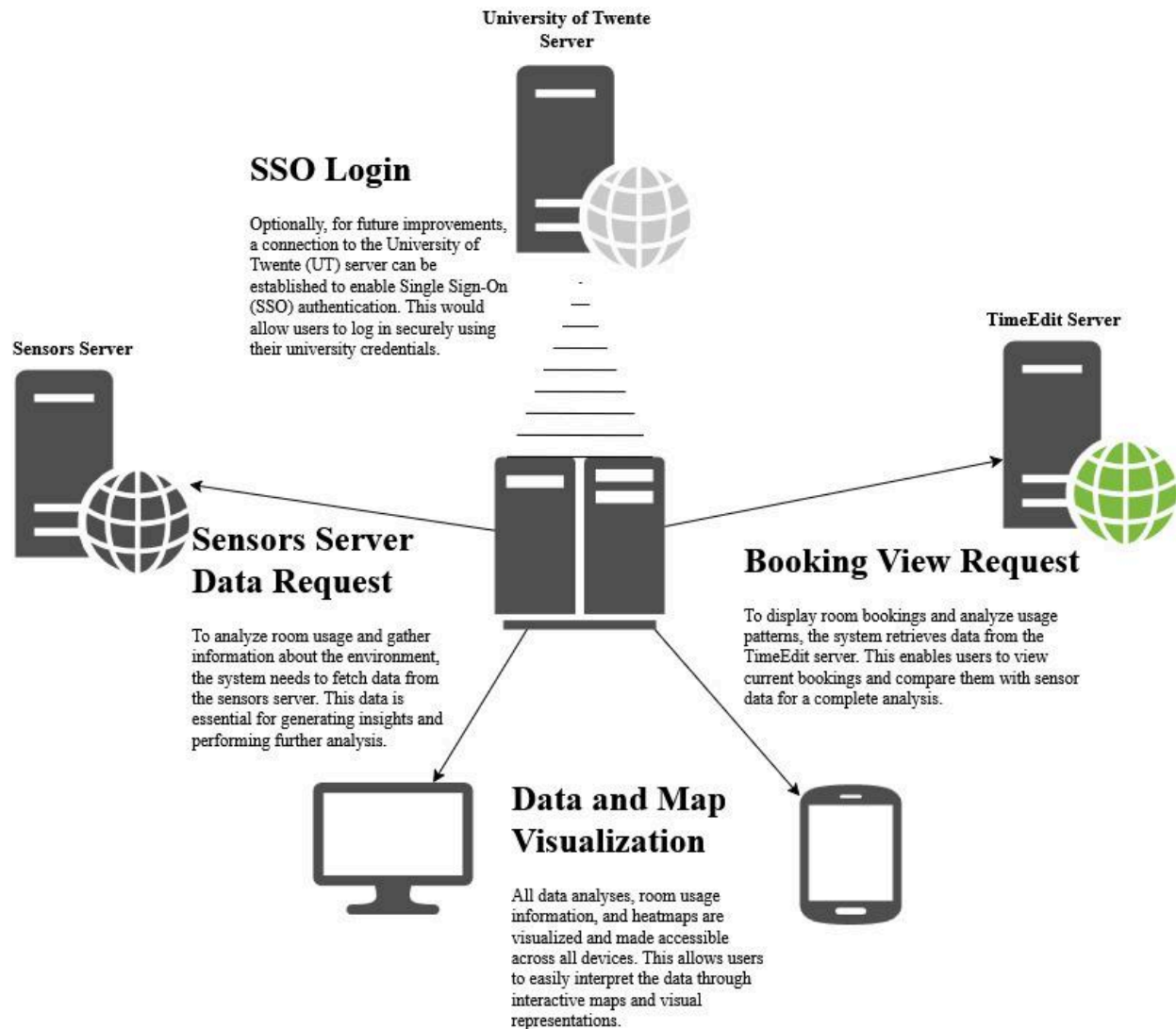
## 9. TimeEditBookingsExceptions

Defines custom exceptions used throughout the booking system.

- Ensures consistent error handling across filter construction, API calls, and analysis steps.

Note: **TimeEditUtilsAnalysis** and **TimeEditRetrieveInsert** are not connected. The arrows between them are from **TimeEditEncodeDecode** and **TimeEditStructure**.

## Hardware Design Diagram



**SSO Login:** Optionally, for future improvements, a connection to the University of Twente (UT) server can be established to enable Single Sign-On (SSO) authentication. This would allow users to log in securely using their university credentials.

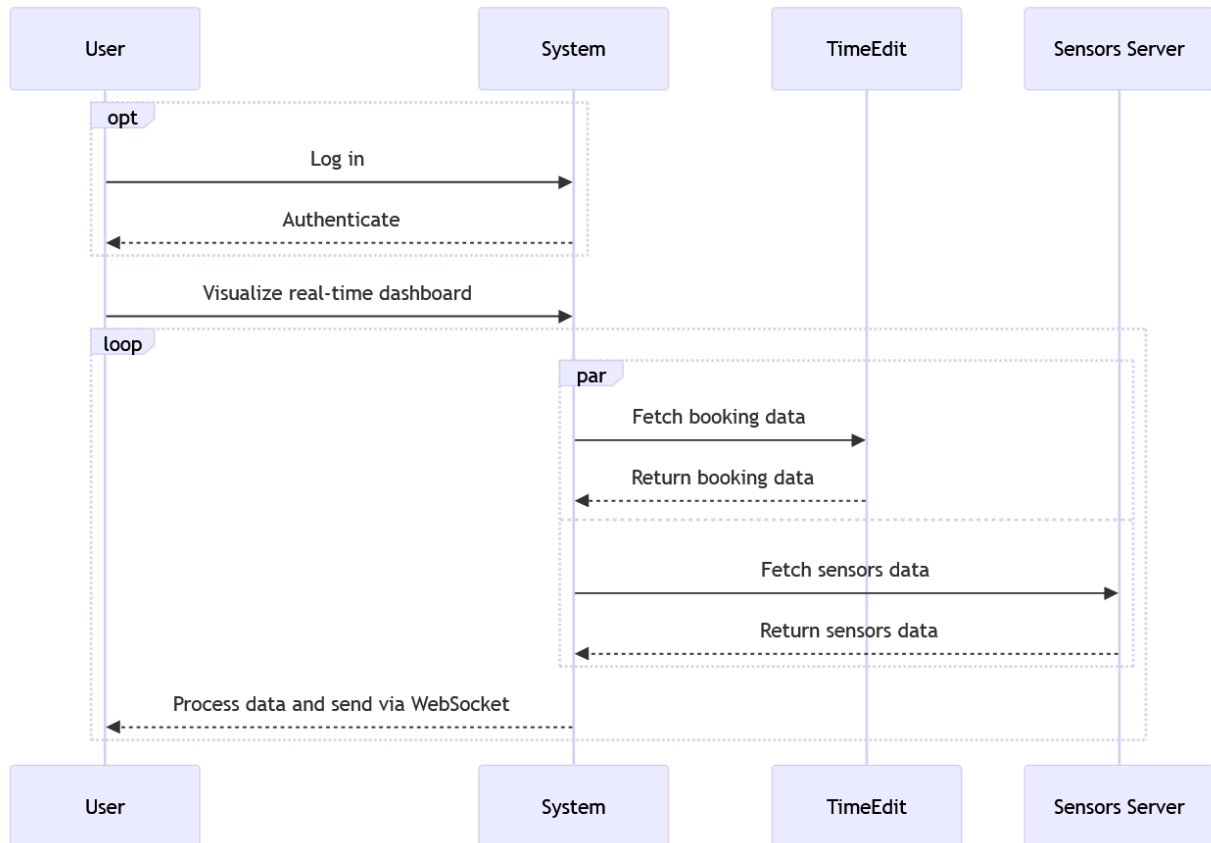
**Sensors Server Data Request:** To analyze room usage and gather information about the environment, the system needs to fetch data from the sensors server. This data is essential for generating insights and performing further analysis.

**Booking View Request:** To display room bookings and analyze usage patterns, the system retrieves data from the TimeEdit server. This enables users to view current bookings and compare them with sensor data for a complete analysis.

**Data and Map Visualization:** All data analyses, room usage information, and analysis maps are visualized and made accessible across all devices. This allows users to easily interpret the data through interactive maps and visual representations.

## Sequence Diagrams

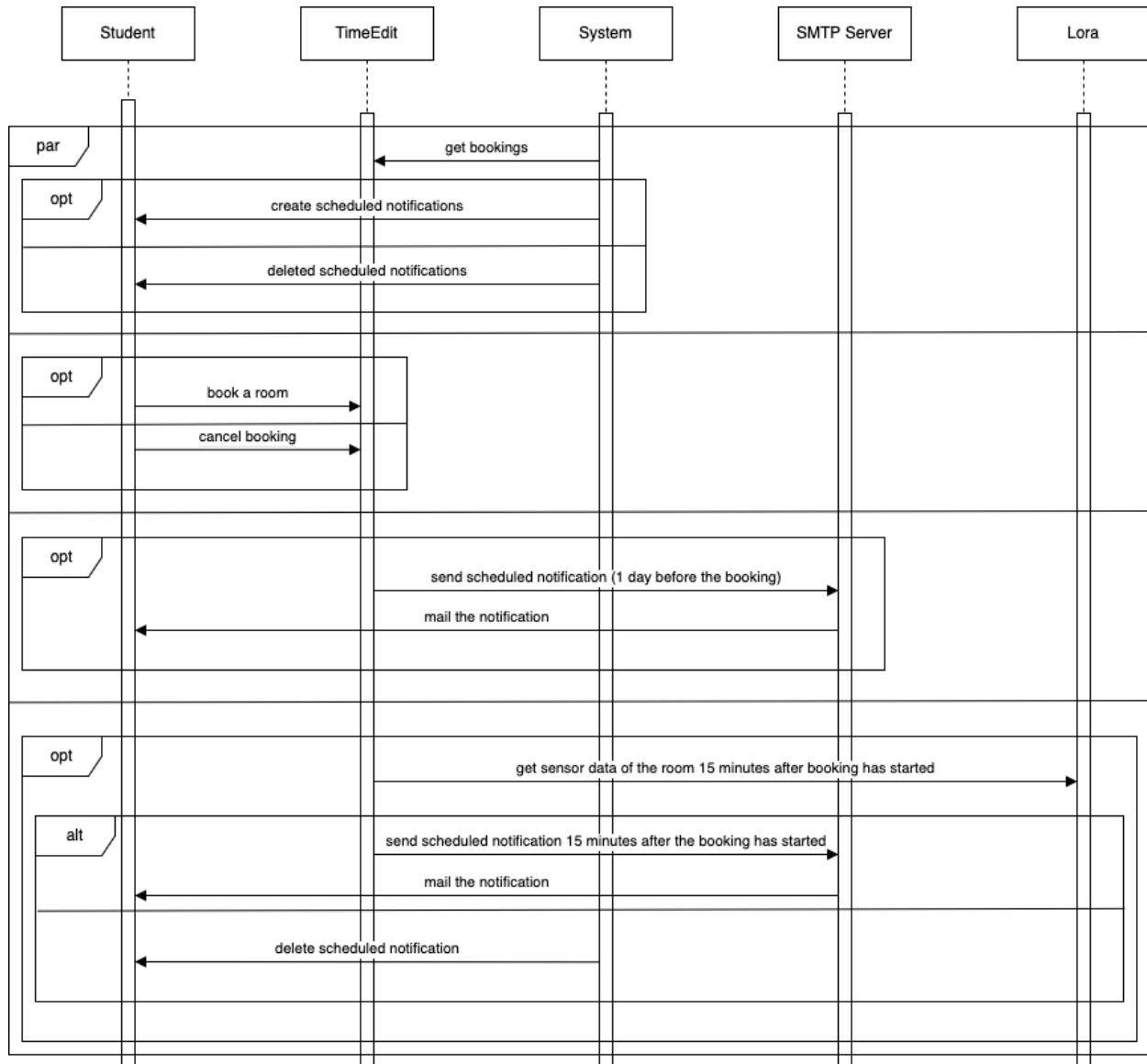
### Real-time Room Monitoring



The diagram showcases the scenario when a user wants to access real-time data regarding rooms (occupancy and booking data). The user can optionally log in, if he is a staff user (this is required only for visualizing the table with raw sensor readings).

After logging in, the user accesses the dashboard in order to visualize real-time data about rooms. The client sends a request to the server asking for this data. The server is in a continuous state of polling most recent data regarding bookings (from TimeEdit) and sensor readings (from the UT's server). After processing the data, the server forwards it to the client via WebSockets.

## Student Booking Notification [Inapplicable]



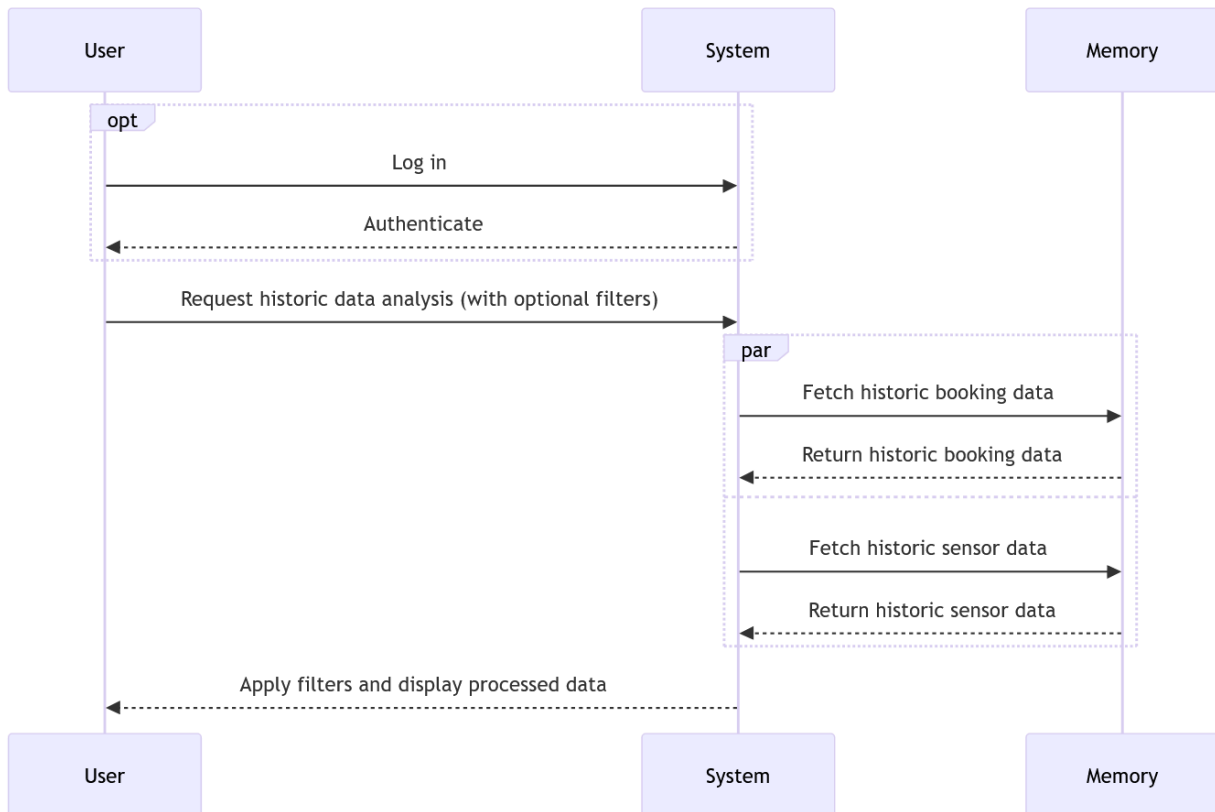
This diagram describes all possible scenarios related to the booking notifications. There are a couple of things happening in parallel. A student can book a room and/or they can cancel a booking. At the same time, the system fetches bookings on a regular basis from TimeEdit and if new bookings are created, it creates scheduled notifications for those bookings. Similarly, for canceled bookings, the system deletes the scheduled notifications it created when those bookings were made.

In parallel, the system sends to the students notifications via mail through an SMTP server when the booking is due in 24hrs. Also, if a booking has started and 15 minutes



after it started the sensors do not detect any motion, another notification is sent to the students. Otherwise, this last scheduled notification is deleted.

## Historic data analysis

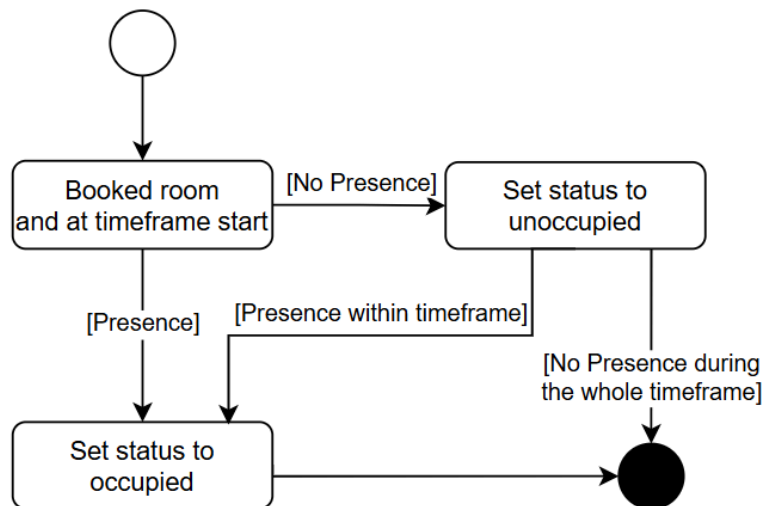


This diagram describes how users can request historic data analyses. Optionally, before sending any requests, the user can authenticate into the system because some endpoints have restricted access; these endpoints are meant to give more insights to staff users. The user can request historic data and optionally indicate filters. The system will then retrieve relevant booking data and sensor data from the historic data stored in memory and process it by applying filters. Finally, the resulting data is being displayed to the user.

## State Machine Diagrams

**Disclaimer:** The diagrams do not include functionalities and features mentioned in the C and W Functional Requirements.

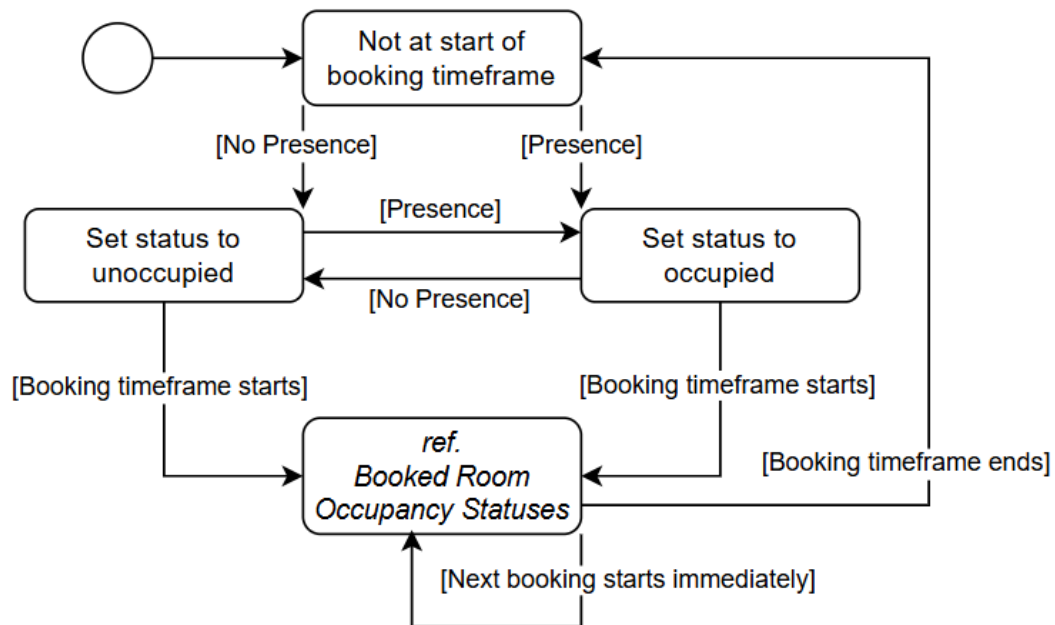
### Diagram Booked Room Occupancy Statuses



This state machine diagram represents the occupancy status handling of a booked room. The system begins when a room is booked and the booking time frame starts. At this point, the system checks whether any presence is detected in the room.

If no presence is detected at the start and remains absent throughout the entire time frame, the final status is set to “unoccupied”. If presence is detected at any time within the booking time frame, the final status becomes “occupied”.

## Diagram Unbooked Room Occupancy Statuses



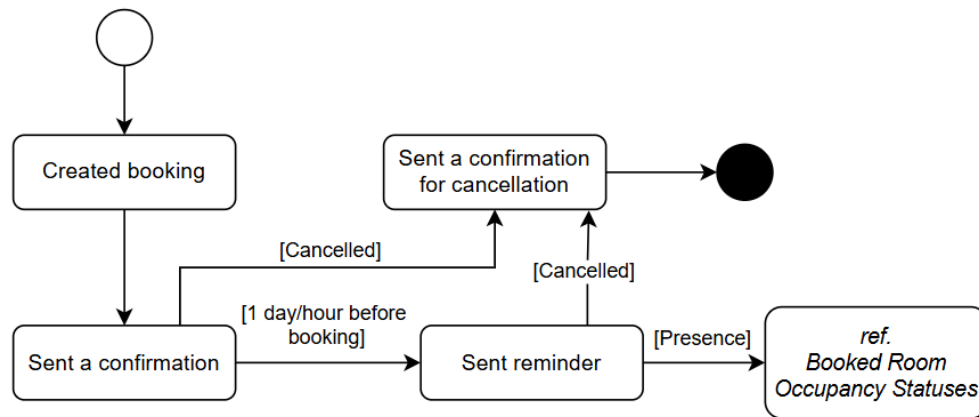
This state machine diagram represents the occupancy status handling of an unbooked room. The system begins (and repeats) when the current time is not the start of any booking timeframe of the given room. Next, the system checks whether any presence is detected in the room.

If no presence is detected at the particular sensor reading, the temporary status is set to “unoccupied” until the next reading. If presence is detected, the temporary status becomes “occupied” until the next reading. In this way, the occupancy status can be indefinitely changed from occupied to unoccupied and vice versa until the booking timeframe of the room starts.

During the booking time frame, the process is the same as described in [Booked Room Occupancy Statuses](#). After that, if another booking starts immediately after the current one ends, the system enters the state machine [Booked Room Occupancy Statuses](#). This can happen indefinitely until no booking begins when the previous one ends, repeating the whole process.

## Diagram Notifications [[Inapplicable](#)]

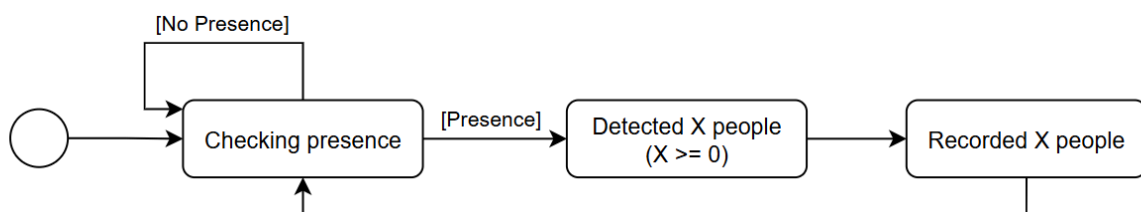
**Disclaimer:** The notifications will be implemented only if we have access to the user emails.



This state machine diagram represents the process of managing notifications. The process starts when a booking is created, triggering a confirmation to be sent. If the booking is canceled at any point, a cancellation confirmation is sent, and the process ends.

A reminder is issued one day or one hour (subject to change) before the booking. If the booking is not canceled, the system enters the state machine [Booked Room Occupancy Statuses](#).

## Diagram People Count [[Inapplicable](#)]



The “People Count” process starts after presence is detected in a room. Then, it runs indefinitely, checking for presence and keeping track of the count of people (it can still be 0 if its confidence rate is (very) low) when presence is detected in the room.

## 11. Analysis Visuals List

The “[Staff Only]” titles indicate that they will be accessible only to staff members. Moreover, some charts or graphs have room for *extensions* where the extensions have the lowest priority and may only be reproducible with fake data within the scope of our project. There are also 3 charts (13, 14, 15) that were not implemented due to time constraints, [complications and limitations](#).

Please note that this list and the [API List](#) complement each other.

### 1. Booking Frequency By Start Time

- Explanation: Number of booking instances within a time unit (hour, day time, weekday, week, month).
- Type: Line Chart, Bar Chart, Pie Chart
- Data Needed:
  - Booking data from TimeEdit

### 2. Booking Frequency By Created Time

- Explanation: Number of created booking instances within a time unit (hour, day time).
- Type: Bar Chart, Line Chart, Pie Chart
- Data Needed:
  - Booking data from TimeEdit

### 3. Booked and Unoccupied; Booked and Occupied; Unbooked and Unoccupied; Unbooked and Occupied + Unknown [Staff Only]

- Explanation: Opposing the 4 cases of booked and/or occupied rooms. An addition could be a fifth case - “Unknown” - when sensor data is unavailable.
- Type: Pie Chart / Table
- Data Needed:
  - Booking data from TimeEdit
  - Sensor data

#### **4. Room Capacity vs TimeEdit Given People Count Discrepancies**

**[Staff Only]**

- Explanation: Compare the discrepancies between room capacity and TimeEdit's People Count. Highlight the discrepancies by displaying them in a table containing all bookings.
- Type: Table (per booking) / Pie Chart and Bar Chart (per room type)
- Data Needed:
  - Booking data from TimeEdit

#### **5. Cancelled vs Not Cancelled Bookings**

- Explanation: See how many bookings have been cancelled and not cancelled.
- Type: Bar Chart, Pie Chart
- Data Needed:
  - Booking data from TimeEdit

#### **6. Booking Rate Visualisation**

- Explanation: Based on the number of bookings. Extension: For example, focusing on two types of rooms: 1-person rooms (VR171-VR180) and all other rooms, and visualizing whether there is a difference in preferred rooms.
- Type: Analysis Map (floor levels) / Pie Chart, Bar Chart
- Data Needed:
  - Booking data from TimeEdit

#### **7. Booking Durations**

- Explanation: Show the popularity of different booking durations by their booking counts.
- Type: Bar Chart (bar per week, month)
- Data Needed:
  - Booking data from TimeEdit

## 8. Occupancy Rate Visualisation

- Explanation: Display the occupancy rate (aka popularity) based on sensor data related to presence detection.
- Type: Analysis Map (floor levels) / Pie Chart (room types)
- Data Needed:
  - Sensor-based room occupancy data

## 9. Booking Lead Time

- Explanation: Check how far in advance users book rooms.
- Type: Bar Chart (showcasing a combined time range per bar, e.g., < 1 hour, 1-4 hours, 4-8 hours, 8-16 hours, 16-48 hours, >48 hours in advance)
- Data Needed:
  - Booking data from TimeEdit

## 10. Real-time Occupancy Monitoring

- Explanation: Check the occupancy of the rooms in real time by viewing the occupancy status. Extensions could be displaying the people count and booking status.
- Type: Analysis Map / Table
- Data Needed:
  - Sensor-based room occupancy data - presence and people count
  - Booking data from TimeEdit

## 11. Sensor Data Monitoring [Staff Only]

- Explanation: Display the raw outputs of the sensors in a list.
- Type: Table
- Data Needed:
  - Sensor-based room occupancy data - presence and people count

## 12. Humidity/Temperature Monitoring - Extra from other students (implemented)

- Explanation: Check the humidity and temperature of the rooms in real time.
- Type: Table

- Data Needed:
  - Sensor-based room temperature/humidity data
  - Room IDs

### **13. Modified Bookings - Extra (not implemented)**

- Explanation: See what bookings have been modified.
- Type: Bar Chart, Pie Chart
- Data Needed:
  - Booking data from TimeEdit

### **14. Under/normally/over-populated rooms [Staff Only] - Won't Do (complications)**

- Explanation: View reports of rooms being under/normally/over-populated per room type and room ID. Extension: Correlate these with the conditions of all other rooms in the library at the times of over/under-population (the library was fully booked, the given floor was full, the rooms of the same type were taken, etc.). Please note that this will likely not be reproducible within the scope of our project, but we can still create fake test cases and mention it for future projects.
- Type: Pie Chart (per room types) / Table (recordings per Room ID containing the population status and datetime)
- Data Needed:
  - Sensor-based room occupancy data - presence and people count
  - Booking data from TimeEdit

### **15. Sensor Data Discrepancies and Confidence [Staff Only] - Won't Do (complications)**

- Explanation: Compare detected discrepancies in sensor readings. View the calculated confidence in the different data. Includes only rooms with 2 or more sensors.
- Type: Table (to check particular readings) / Pie chart (ratio of confidence in different sensor types)
- Data Needed:
  - Sensor-based room occupancy data



## 12. API List

Please refer to the section [Analysis Visuals List](#) for detailed explanations of the numbered diagrams (in the subsections [II.](#) and [III.](#)) by mapping their corresponding numbers.

### I. General definitions

#### 1. Time Units

- **Hours:**
  - **Booking data:** data is grouped by **2** hours - from 08:00, 10:00, 12:00, ..., up to 22:00. For example, if the given group's **name** is "08:00", its **value** will take into account all datapoints between (including) "08:00" and (excluding) "10:00". The same applies to all other hour groups.
  - **Occupancy data:** data is grouped into **1** hour chunks - from 08:00, 09:00, 10:00, ..., up to 23:00. For example, if the given group's **name** is "08:00", its **value** will take into account all datapoints between (including) "08:00" and (excluding) "09:00". The same applies to all other hour groups.
- **Durations:** "<1h" (strictly less than 1h), "1-2h" (includes 1h and 2h), "2-3h" (includes 3h), "3-4h" (includes 4h), ">4h" (strictly greater than 4h).
- **Lead times:** "<1h" (strictly less than 1h), "1-4h" (includes 1h and 4h), "4-8h" (includes 8h), "8-16h" (includes 16h), "16-48h" (includes 48h), ">48h" (strictly greater than 48h).
- **Daytimes:** data is grouped by parts of the day:
  - Morning (08:00 - 12:00);
  - Afternoon (12:00 - 16:00);
  - Evening (16:00 - 20:00);
  - Night (20:00 - 23:59).
- **Weekdays** or **days:** the names of the days in the week - Monday, Tuesday, ..., Sunday.
- **Weeks:** format - [ yyyy-Wpq ] - e.g., 2024-W41 or 2025-W05.
- **Months:** format - [ monthName yyyy ] - e.g., January 2025 or October 2024.

#### 2. All Filters

- **startTime** and **endTime**
  - Type: string

- Format: [ HH:MM ]
- Please note that data is generally filtered (when **startTime** and/or **endTime** are not provided) by approximate opening hours - **startTime default** “08:00” and **endTime default** “23:59”. That is because the opening hours of the library sometimes change.
- **startDate** and **endDate**
  - Type: string
  - Format: [ yyyy-mm-dd ]
- **week**
  - Type: string
  - Format: [ yyyy-Wpq ]
- **bookingDuration**
  - Type: string
  - Values: “<1”, “1-2” (including 1 and 2 hours), “2-3” (including 3 hours), “3-4” (including 4 hours), “>4”. Note that there is no ‘h’ for the filter.
- **room**
  - Type: string
  - Values: all rooms present in TimeEdit
- **capacity**
  - Type: string
  - Values: “1”, “2-3”, “>3”
- **floor**
  - Type: int
  - Values: 1, 2, 3, 4, 5
- **sensorType**
  - Type: string
  - Values: “nighthawk”, “erseye”

### 3. Discrepancy (difference) types for Diagram 4 (Capacity vs. People Count)

- **Invalid:** if  $\text{people\_count} \leq 0$  or  $(\text{capacity} * 3) < \text{people\_count}$
- **Smaller:** if  $\text{capacity} > \text{people\_count}$
- **Equal:** if  $\text{capacity} == \text{people\_count}$
- **Larger:** if  $\text{capacity} < \text{people\_count}$

## II. Blueprint `/api/bookings` (located in *bookings\_api.py*)

### **Diagram 1**

The data is grouped by start time (“Begin datetime”) by a given time unit.

#### **1. `/api/bookings/start/hours/`**

- Invalid filters: -
- Example output:  
[{'name': '08:00', 'value': 93}, {'name': '10:00', 'value': 149}, {'name': '12:00', 'value': 219}, {'name': '14:00', 'value': 183}, {'name': '16:00', 'value': 186}, {'name': '18:00', 'value': 116}, {'name': '20:00', 'value': 35}, {'name': '22:00', 'value': 0}]

#### **2. `/api/bookings/start/daytimes/`**

- Invalid filters: -
- Example output:  
[{'name': 'Morning', 'value': 242}, {'name': 'Afternoon', 'value': 402}, {'name': 'Evening', 'value': 302}, {'name': 'Night', 'value': 35}]

#### **3. `/api/bookings/start/weekdays/`**

- Invalid filters: startDate, endDate
- Example output:  
[{'name': 'Monday', 'value': 111}, {'name': 'Tuesday', 'value': 106}, {'name': 'Wednesday', 'value': 125}, {'name': 'Thursday', 'value': 127}, {'name': 'Friday', 'value': 106}, {'name': 'Saturday', 'value': 19}, {'name': 'Sunday', 'value': 31}]

#### **4. `/api/bookings/start/weeks/`**

- Invalid filters: week
- Example output:  
[{'name': '2024-W40', 'value': 514}, {'name': '2024-W41', 'value': 467}]

#### **5. `/api/bookings/start/months/`**

- Invalid filters: week

- Example output:  
[{'name': 'October 2024', 'value': 981}]

## **Diagram 2**

The data is grouped by timestamp (“Created”) by a given time unit.

### **1. /api/bookings/created/hours/**

- Invalid filters: -
- Example output:  
[{'name': '08:00', 'value': 69}, {'name': '10:00', 'value': 145}, {'name': '12:00', 'value': 147}, {'name': '14:00', 'value': 167}, {'name': '16:00', 'value': 154}, {'name': '18:00', 'value': 93}, {'name': '20:00', 'value': 75}, {'name': '22:00', 'value': 39}]

### **2. /api/bookings/created/daytimes/**

- Invalid filters: -
- Example output:  
[{'name': 'Morning', 'value': 223}, {'name': 'Afternoon', 'value': 314}, {'name': 'Evening', 'value': 247}, {'name': 'Night', 'value': 142}]

## **Diagram 4**

**1. /api/bookings/discrepancies/bar-chart/** - show the capacity types and their discrepancy counts

- Format: [ capacity\_type: discrepancy\_count ]
- Invalid filters: capacity, room, floor
- Example output:  
[{'name': 'Small (1 person)', 'value': 6}, {'name': 'Medium (2-3 people)', 'value': 68}, {'name': 'Large (>3 people)', 'value': 640}]

**2. /api/bookings/discrepancies/table/** - display a booking per row and highlight the discrepancy. The included columns are: *booking\_id*, *capacity*, *people\_count*, *difference* (the discrepancy type), *room*.

- Invalid filters: -
- Example output:  
[{'booking\_id': '56315', 'capacity': 7, 'people\_count': 2, 'difference': 'Smaller', 'room':

```
'VR275R'}, {'booking_id': '51994', 'capacity': 7, 'people_count': 4, 'difference':  
'Smaller', 'room': 'VR275C'}, {'booking_id': '53256', 'capacity': 7, 'people_count': 7,  
'difference': 'Equal', 'room': 'VR275T'}, {'booking_id': '56569', 'capacity': 7,  
'people_count': 4, 'difference': 'Smaller', 'room': 'VR275J'}, {'booking_id': '56245',  
'capacity': 7, 'people_count': 3, 'difference': 'Smaller', 'room': 'VR275O'},  
{ 'booking_id': '56262', 'capacity': 2, 'people_count': 2, 'difference': 'Equal', 'room':  
'VR257'}}
```

### **Diagram 5**

The data returns the counts of Cancelled and Not Cancelled bookings.

#### **1. /api/bookings/cancellations/**

- Invalid filters: -
- Example output:  
[{'name': 'Cancelled', 'value': 117}, {'name': 'Not Cancelled', 'value': 864}]

### **Diagram 6**

The data is in the format: [ capacity\_type: booking\_count ].

#### **1. /api/bookings/preferred/room-types/**

- Invalid filters: capacity, room, floor
- Example output:  
[{'name': 'Small (1 person)', 'value': 45}, {'name': 'Medium (2-3 people)', 'value': 181},  
{ 'name': 'Large (>3 people)', 'value': 755}]

### **Diagram 7**

The data is in the format: [ duration\_label: booking\_count ].

#### **1. /api/bookings/durations/**

- Invalid filters: bookingDuration
- Example output:  
[{'name': '<1h', 'value': 21}, {'name': '1-2h', 'value': 634}, {'name': '2-3h', 'value': 129},

```
{'name': '3-4h', 'value': 180}, {'name': '>4h', 'value': 17}]
```

### **Diagram 9**

The data is in the format: [ lead\_time\_label: booking\_count ].

#### **1. /api/bookings/lead-times/**

- Invalid filters: -
- Example output:  
[{'name': '<1h', 'value': 966}, {'name': '1-4h', 'value': 0}, {'name': '4-8h', 'value': 0},  
{'name': '8-16h', 'value': 4}, {'name': '16-48h', 'value': 6}, {'name': '>48h', 'value': 5}]

### **III. Blueprint /api (located in *occupancy\_api.py*) & WebSocket events**

### **Diagram 3**

For the chart, the data is in the format: [ discrepancy\_type: discrepancy\_count ] (historic data).

For the table, data is grouped per room name and timestamp (historic data). There can be multiple entries for the same room but with different timestamps.

#### **1. /api/discrepancies/**

- Invalid filters: bookingDuration
- Example output:  
[{"name": "Unbooked/Unoccupied", "value": 4922}, {"name": "Unbooked/Occupied",  
"value": 1497}, {"name": "Booked/Occupied", "value": 1066}, {"name":  
"Booked/Unoccupied", "value": 296}]

#### **2. /api/discrepancies/table/**

- Invalid filters: bookingDuration
- Example output:  
[{"capacity": 0, "discrepancy": "Unbooked/Unoccupied", "name": "VR275A",  
"sensor\_type": "Erseye", "timestamp": "2025-04-18 15:36"}, {"capacity": 7,  
"discrepancy": "Unbooked/Unoccupied", "name": "VR275G", "sensor\_type":

```
"Erseye&Nighthawk", "timestamp": "2025-04-18 15:36"}, {"capacity": 1,
"discrepancy": "Unbooked/Unoccupied", "name": "VR174", "sensor_type":
"Erseye&Nighthawk", "timestamp": "2025-04-18 15:36"]}]
```

## **Diagram 8**

For the map, the data is in the format [ room\_name: hex\_colorcode ]

For the chart, data is grouped per room type [ room\_type: frequencies\_count ].

### **1. /api/occupancy-frequencies/colorcodes/**

- Invalid filters: none; this endpoint does not take filters into consideration.
- Example output:  
{ "VR170": "#d1ff26", "VR171": "#aaff4d", "VR172": "#f8f500", "VR173": "#5aff9d",  
"VR174": "#8aff6d", "VR175": "#ffd000", "VR176": "#deff19", "VR177": "#ffae00",  
"VR178": "#ff9f00", "VR179": "#ff8d00", "VR180": "#30ffc7", "VR191B": "#caff2c",  
"VR193A": "#80ff77", "VR193B": "#9dff5a", "VR193D": "#ceff29", "VR193E": "#ffcc00",  
"VR193F": "#a7ff50", "VR193G": "#c4ff33", "VR193I": "#deff19", "VR193J": "#a7ff50",  
"VR193K": "#deff19", "VR193L": "#b4ff43", "VR193N": "#77ff80", "VR247": "#c4ff33",  
"VR248": "#8aff6d", "VR256": "#d1ff26", "VR257": "#beff39", "VR258": "#5dff9a",  
"VR259": "#000080", "VR261": "#0014ff", "VR262": "#001cff", "VR275A": "#19ffde",  
"VR275B": "#ff7300", "VR275C": "#f8f500", "VR275E": "#e40000", "VR275G":  
"#800000", "VR275H": "#ffd000", "VR275J": "#ffd300", "VR275K": "#ff7e00",  
"VR275L": "#ebff0c", "VR275M": "#e1ff16", "VR275O": "#ffb200", "VR275P":  
"#e4ff13", "VR275Q": "#ffc100", "VR275R": "#ff7e00", "VR275T": "#f1fc06" }

### **2. /api/occupancy-frequencies/**

- Invalid filters: bookingDuration
- Example output:  
[{"name": "Small(1 person)", "value": 3448}, {"name": "Medium(2-3 people)", "value":  
1651}, {"name": "Large(>3 people)", "value": 10568}]

## Diagram 10

For the map, data is grouped per room name { room\_name: data }. The “data” contains information about the booked status, occupancy status, room capacity, and associated sensors and their raw data.

For the table, data is returned as a list of dictionaries, each dictionary containing data about a room [ {room\_data} ].

This endpoint has associated WebSocket events. The server is listening for the events and expects to receive the filters to be applied (similar to the endpoint).

### 1. /api/rooms/real-time/

- **WebSocket event: “getRoomsOccupancyMap”**
- Invalid filters: booking\_duration, start\_date, start\_time, end\_date, end\_time, week
- Example output:

```
{"VR170": {"booked": 0, "capacity": "7", "erseye": {"geo": {"alt": "1", "lat": "52.243968963623", "lon": "6.85338068008423"}, "humidity": 47.0, "light": 232.0, "motion": 0.0, "occupancy": 0.0, "sensor_id": "A81758FFFE0362B3", "sensor_type": "Erseye", "temperature": 19.4, "time": "2025-04-18T16: 07: 55.775753+00: 00", "vdd": 3596.0}, "occupancy": 0}, "VR171": {"booked": 0, "capacity": "1", "erseye": {"geo": {"alt": "1", "lat": "52.2439422607422", "lon": "6.85342788696289"}, "humidity": 43.0, "light": 7.0, "motion": 0.0, "occupancy": 0.0, "sensor_id": "A81758FFFE0362B4", "sensor_type": "Erseye", "temperature": 19.2, "time": "2025-04-18T16: 08: 02.306076+00: 00", "vdd": 3591.0}, "occupancy": 0}, }
```

### 2. /api/rooms/table/

- **WebSocket event: “getRoomList”**
- Invalid filters: booking\_duration, start\_date, start\_time, end\_date, end\_time, week
- Example output:

```
[{"booking": "NotBooked", "capacity": 0, "name": "VR261", "occupancy": "Available", "total_bookings": 0}, {"booking": "NotBooked", "capacity": "2", "name": "VR257", "occupancy": "Available", "total_bookings": 541}, {"booking": "NotBooked", "capacity": "7", "name": "VR275O", "occupancy": "Available", "total_bookings": 565}, ]
```



## **Diagram 11**

The data is returned as a list grouped by sensor name. Each entry contains the sensor\_id, its associated room, and the sensor values.

This endpoint has an associated WebSocket event. The server is listening for the event and expects to receive the filters to be applied (similar to the endpoint).

### **1. /api/sensors/table/**

- **WebSocket event: “getSensorList”**
- Invalid filters: booking\_duration, start\_date, start\_time, end\_date, end\_time, week
- Example output:  

```
[{"occupancy": 0, "room": "VR275L", "sensor_id": "A81758FFFE048577", "timestamp": "2025-04-18 16:07", "type": "Erseye", "value": {"geo": {"alt": "2", "lat": "52.2439384460449", "lon": "6.85386037826538"}, "humidity": 41.0, "light": 11.0, "motion": 0.0, "occupancy": 0.0, "sensor_id": "A81758FFFE048577", "sensor_type": "Erseye", "temperature": 19.4, "time": "2025-04-18T16: 07: 21.913381+00: 00", "vdd": 3660.0}}, {"occupancy": 0, "room": "VR275Q", "sensor_id": "A81758FFFE048579", "timestamp": "2025-04-18 16:07", "type": "Erseye", "value": {"geo": {"alt": "2", "lat": "52.2437705993652", "lon": "6.85387706756592"}, "humidity": 43.0, "light": 0.0, "motion": 0.0, "occupancy": 0.0, "sensor_id": "A81758FFFE048579", "sensor_type": "Erseye", "temperature": 18.9, "time": "2025-04-18T16: 07: 30.557314+00: 00", "vdd": 3653.0}}, ]
```

## **Diagram 12**

For this diagram, the previous endpoint (diagram 11) is reused. Humidity and temperature are part of the data collected by sensors. This diagram is joined on the client side with diagram 11 to be displayed in a single table.

### **1. /api/sensors/table/**

- Invalid filters: booking\_duration, start\_date, start\_time, end\_date, end\_time, week
- Example output:  

```
[{"occupancy": 0, "room": "VR275L", "sensor_id": "A81758FFFE048577", "timestamp": "2025-04-18 16:07", "type": "Erseye", "value": {"geo": {"alt": "2", "lat": "52.2439384460449", "lon": "6.85386037826538"}, "humidity": 41.0, "light": 11.0, "motion": 0.0, "occupancy": 0.0, "sensor_id": "A81758FFFE048577", "sensor_type": "Erseye", "temperature": 19.4, "time": "2025-04-18T16: 07: 21.913381+00: 00", "vdd":
```

```
3660.0}}, {"occupancy": 0, "room": "VR275Q", "sensor_id": "A81758FFFE048579",  
"timestamp": "2025-04-18 16:07", "type": "Erseye", "value": {"geo": {"alt": "2", "lat":  
"52.2437705993652", "lon": "6.85387706756592"}, "humidity": 43.0, "light": 0.0,  
"motion": 0.0, "occupancy": 0.0, "sensor_id": "A81758FFFE048579", "sensor_type":  
"Erseye", "temperature": 18.9, "time": "2025-04-18T16: 07: 30.557314+00: 00",  
"vdd": 3653.0}}, ]
```

### 13. Test Schedule

A well-tested environment is crucial for multiple reasons. One of the most fundamental use cases is testing new features as they are gradually added to the project. Testing provides confidence in the correctness of our implementation while also helping us identify hidden bugs. Additionally, it ensures that any code changes-whether a new feature or a refactor not introduce unintended side effects.

Starting from week 4, during the design phase, we will proceed with user acceptance testing to understand and document the final key steps and ensure platform UI user-friendliness before any implementation steps. After we begin implementation, testing will be an integral part of our development process. Initially, we will focus on unit testing for each component: frontend, backend, TimeEdit, and MazeMap, before any integration takes place. These tests will verify that individual modules function correctly in isolation. In parallel, we will manually test the available sensors to understand how to process and manage data effectively. This phase will help us establish a solid foundation before moving on to integration testing.

By week 6, we will begin integration testing while continuing to write unit tests for new features. Integration testing will validate that different components work together as expected. For example, backend integration tests will simulate API calls under various conditions to ensure the correct responses are returned. A key scenario includes testing user permissions when retrieving room occupancy data and verifying that users without proper privileges cannot access restricted information. This phase is essential to confirm that interactions between system components are seamless and reliable.

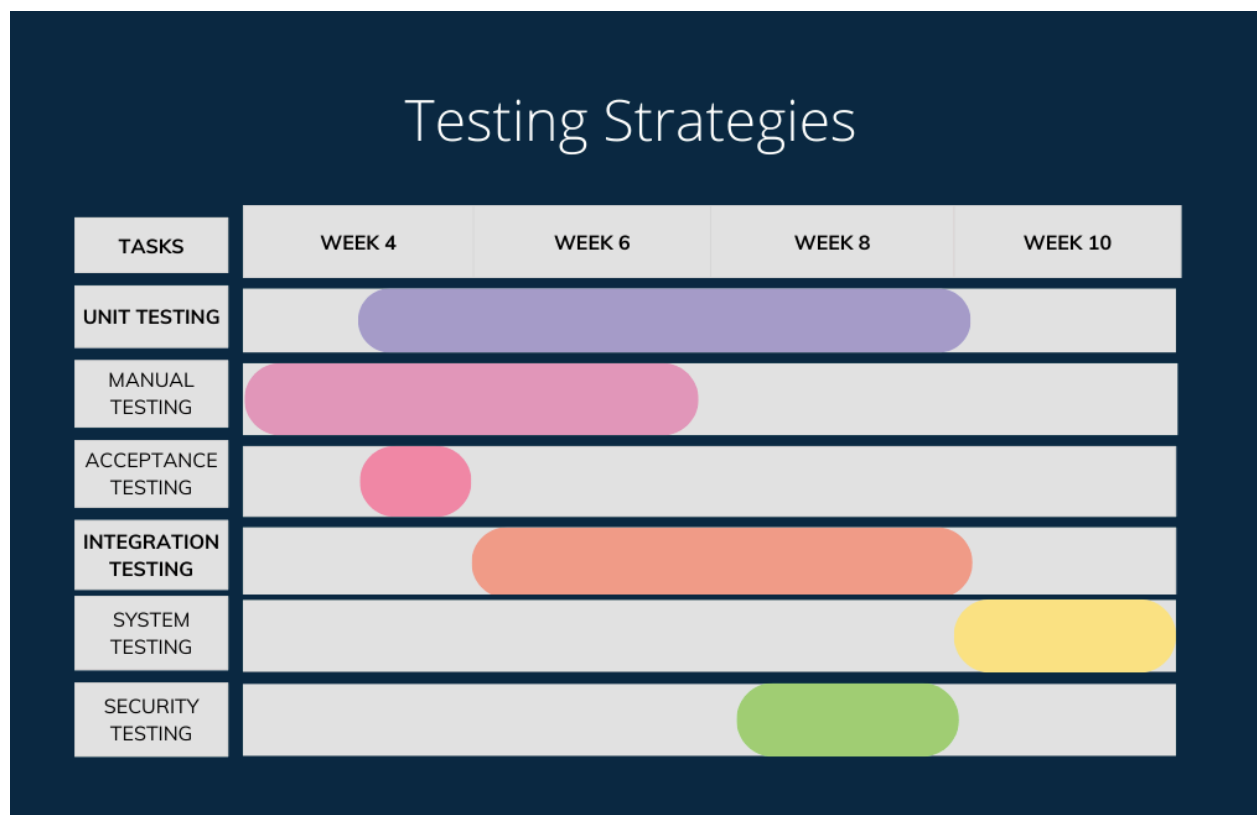
Also, by weeks 6-7, we will focus on integration testing for TimeEdit and MazeMap to ensure smooth interaction with external systems. The goal is to verify that booking data from TimeEdit is correctly processed and correlated with sensor data and that navigation functionality through MazeMap works without errors. This will be a critical phase, as these integrations directly impact core system features such as room booking accuracy, real-time monitoring, and navigation support.

As we approach an MVP, around week 8, we will introduce end-to-end (E2E) testing to ensure the system behaves as expected from a user's perspective. Using Cypress, we will simulate real-world interactions, such as logging in, navigating the platform, and performing key actions. This stage will help us catch potential issues that unit and integration tests might have missed, ensuring the overall user experience remains smooth and functional. Additionally, we will address any final bug fixes and performance optimizations before deployment.

Beyond automated testing, we will enforce code reviews as an additional quality assurance measure. As GitLab defines it, "A code review is a peer review of code that helps developers ensure or improve the code quality before they merge and ship it." Each team member will have specific responsibilities, and every feature will be reviewed by another developer before being merged. Code reviews ensure that best practices, such as clean code principles and handling edge cases, are followed. This process also fosters collaboration and helps identify issues that automated tests might not detect.

A feature will only be merged once it has passed all necessary tests and has been reviewed by at least one team member. This structured approach ensures that our system remains robust, maintainable, and scalable, minimizing unexpected failures and technical debt.

By following this test plan, we will systematically validate our implementation at different stages, allowing us to build a stable and reliable system while maintaining high development standards.



## 14. Test Strategy

### 1. Scope and Overview

The **Test Strategy** defines the high-level testing approach for verifying the requirements of the system. The primary goal is to verify that all Must and the majority of Should requirements are met, while lower-priority requirements (Could, Won't) undergo limited or no testing, depending on the amount of reserved time. Our testing phases include Unit Testing, Manual Testing, Integration Testing, User Acceptance Testing, System Testing, Performance Testing, and Security Testing.

### 2. Types of Testing

#### 2.1 Unit Testing

The objective of unit testing is to verify the correctness of individual modules or components of the system (e.g., front-end pages, back-end endpoints & processing, sensor data processing, etc.). Each module will be tested with a variety of automated tests (as long as it is possible and feasible). The goal is at least 90% code coverage on Must and Should features.

#### 2.2 Manual Testing

The objective of manual testing is to test use cases and edge cases that might not be fully covered by automated tests. Test scenarios resembling real-world interactions (e.g., staff checking data correlation of selected rooms, students receiving navigation towards an empty room, etc.). Testers will be required to describe each step's outcome and check if it matches the expected results.

#### 2.3 Integration Testing

The goal of integration testing is to confirm that different modules (e.g., presence detection, TimeEdit data visualization, advanced analytics, etc.) work together successfully and exchange data as expected. Modules will be combined progressively and tested respectively. This testing must detect any inconsistent data states or module communication failures.

#### 2.4 User Acceptance Testing (UAT)

The goal of acceptance testing is to validate the system's usability, navigation flow, and overall user satisfaction by having test users interact with Figma prototypes. Quantitative

data (performance time for specific tasks) and qualitative feedback (user appeal, intuitiveness) are used to decide on the final design. Test users will be provided with a set of tasks and scenarios resembling real-life interactions (e.g., navigate to page X, find element Y, etc.).

## **2.5 System Testing**

The objective of system testing is to validate the complete system functionality. Testers will execute end-to-end scenarios covering Must and Should features with real (preferably) or realistic data.

## **2.6 Performance Testing**

The objective of performance testing is to assess the system's speed, throughput, and resource utilization (e.g., CPU, memory) under varying connection and request loads. This testing must ensure the complete validation of Must and Should non-functional requirements, as well as establish existing physical limitations.

## **2.7 Security Testing**

The objective of security testing is to ensure the system complies with security requirements (e.g., role-based access control, secure handling of stored data, etc.). The testing will attempt to exploit potential vulnerabilities in authentication, data exchange, session management, or input sanitization. It will also be tested to ensure that only authorized users (e.g., library staff) can access sensitive analytics data.

## 15. Test Plan

### 1. Unit Testing

For unit testing, we used the pytest and unittest libraries in Python. We used unittest to mock authenticated API requests and pytest to keep the testing process simple and clear.

#### UT-01: Bookings Data Retrieval

- **Description of the Test Process:** The test suite for the bookings API verified multiple endpoints that handle the retrieval of booking related data. These included grouping by hours, daytimes, weekdays, weeks, and months. The tests also included cases with invalid filters to check system robustness. The process focused on the shape and integrity of the returned data, making sure it matched the expected formats, rather than on the correctness of the data content.
- **Result of Test:** All tests passed. The API returned structured results consistently, even when filters were incorrect or missing. The system handled input variations gracefully. Results are displayed in [Appendix 2.1](#).
- **Conclusion (Pass/Fail):** Pass.

#### UT-02: Bookings Utils Functions

- **Description of the Test Process:** The utility functions were tested to confirm their ability to process booking data for structured formatting, filtering, encoding, decoding, and label mapping. The tests checked that outputs met expected structural patterns and correctly handled edge cases like empty data or invalid input.
- **Result of Test:** Every function returned valid, well-structured outputs, and handled both valid and invalid inputs as expected. Functions showed resilience against poorly formed data. Results are displayed in [Appendix 2.2](#).
- **Conclusion (Pass/Fail):** Pass.

#### UT-03: Occupancy Data Retrieval

- **Description of the Test Process:** The occupancy API was tested for its ability to fetch sensor-based occupancy data. Tests covered scenarios with correct parameters, incorrect ones, and absence of parameters, to evaluate the stability and response of the system under various conditions. The focus was on

confirming the returned data structure and ensuring the system remained responsive.

- **Result of Test:** The API returned structured data in all tested cases, including cases with incorrect or missing parameters. All outputs met the required format. Results are displayed in [Appendix 2.3](#).
- **Conclusion (Pass/Fail):** Pass.

#### UT-04: Occupancy Utils Functions

- **Description of the Test Process:** The tests verified helper functions responsible for URL construction, room parsing, and data handling within the occupancy system. Various input formats, including both valid and malformed data, were used to confirm that the functions returned the correct structure and failed safely when expected. The lower number of tests for these files reflects the complexity of the functions and the challenges of working with real-time, dynamic values that change with each execution.
- **Result of Test:** The utility functions were tested against dynamic and unpredictable data formats. Despite this, the functions consistently produced correctly structured outputs. Results are displayed in [Appendix 2.4](#).
- **Conclusion (Pass/Fail):** Pass.

#### UT-05: User Authentication Logic

- **Description of the Test Process:** The tests covered the basic authentication flow, including logging in, logging out, user authorization (both existing and new), and authentication checks for both valid and invalid sessions. The tests verified whether the correct response structures were returned for each use case.
- **Result of Test:** All authentication routes produced correct responses for both successful and unsuccessful login attempts, new and existing users, and authenticated versus unauthenticated access. Results are displayed in [Appendix 2.5](#).
- **Conclusion (Pass/Fail):** Pass.



## 2. Manual Testing

### MT-01: Maze Map Navigation

- **Requirement(s):**
  - **F4.** Implement navigation to a room (using MazeMap).
- **Metric:**
  - Task Success Rate ( $\geq 95\%$ )
  - UI/UX Conventions (Use of intuitive icons and labels)
- **Description of the Test Process:** Team members will manually navigate through the user interface, select the room they want to navigate to, and make observations. Their observations, errors, and overall satisfaction will be recorded.
- **Pass Criteria / Bound:** A task succession rate of 95% for receiving an intuitive path to the selected room will result in a pass.
- **Result of Test:** After having received a list of project rooms from the server API, the user can use the search select input and select the room. The starting point of the path is the library entrance. The user is also offered an intuitive button to toggle floors in case the path includes multiple floors. After conducting the test, all rooms have been indicated with a correct path leading to them. An example is presented in [Appendix 2.7](#).
- **Conclusion (Pass/Fail):** Pass.

### MT-02: Sensor Data Processing and Presence Detection

- **Requirement(s):**
  - **F3.** Implement presence detection by evaluating the data collected from the sensors.
  - **F4.** Correlate sensor data and TimeEdit booking data to identify discrepancies.
- **Metric:**
  - Discrepancy Detection Accuracy (%)
  - Response Time ( $\leq 10$  seconds)
- **Description of the Test Process:** Let the system analyze the continuous stream of real-time data from sensors and TimeEdit. For every update issued by the server, verify that on the user interface, the system:
  1. Updates the occupancy state of rooms based on sensor data (lists and map).

2. Flags any discrepancy case (booked & occupied, booked & unoccupied, unbooked & occupied, unbooked & unoccupied).

Please refer to the [Sensor Data Manual Testing](#) section for more details.

- **Pass Criteria / Bound 1:** The module must consistently detect room occupancies and have a discrepancy detection accuracy of at least 90% across test runs.
- **Pass Criteria / Bound 2:** The module must provide **real-time** data updates every 10 seconds and have a **historic** data analysis response time in under 10 seconds.
- **Result of Test:** The system is only interpreting the readings recorded by the sensors, thus, we are unable to confidently confirm the full correctness of the system. However, while monitoring the state of the rooms on the dashboard and in person, they were aligning. The system correctly identified discrepancies between sensor data and booking data from TimeEdit in 100% of cases. Initial time to load the map displaying room occupancy on the dashboard's main page (including other analyses affecting performance): 9-10 seconds. Subsequent map updates communicated through WebSockets: between 2-3 seconds and 4-5 seconds. Please note that this is the time between when the data was processed by the server and displayed by the client.
  - [Diagram 3](#) (discrepancies): 4-5 seconds.
  - [Diagram 10](#) (real-time occupancy): 9-10 seconds for the map and 4-5 seconds for the table.
- **Conclusion (Pass/Fail): Pass**

### MT-03: TimeEdit Data Retrieval and Processing

- **Requirement(s):**
  - **F1.** Analyze booking behavior of historic data and provide insights (e.g., frequency of bookings, peak usage times) after a user's query.
  - **F2.** Provide Library staff with advanced visualizations, based on TimeEdit booking data in real-time.
- **Metric:**
  - Response Time ( $\leq$  10 seconds)
- **Description of the Test Process:** Execute several queries that fetch and process historic booking data (occupancy status, usage frequency charts, etc.). Measure the time taken for the booking data to be retrieved, parsed, and made available for visualization.

Please refer to the [TimeEdit Manual Testing](#) section for more details.

- **Pass Criteria / Bound:** The booking data processing must complete within 10 seconds and return data in the expected format.
- **Result of Test:**
  - Dashboard initial loading (all analysis at once): 9-10 seconds;
  - Subsequent Dashboard requests (all analysis at once): from 2-3 seconds to 4-5 seconds, depending on the filters (mainly the date and time ranges).
  - [Diagram 4](#) table: 7-8 seconds.
- **Conclusion (Pass/Fail): Pass.**

### 3. Integration Testing

#### IT-01: Booking Data Integration Test

- **Metric:**
  - Endpoint Compliance Rate
  - Response Time **Variation** (average  $\leq 2$  seconds)
- **Description of the Test Process:** Verify the integration of the booking data module by testing all booking-related API endpoints with valid and invalid parameters. The test checks: authentication requirements, data format compliance, filter handling, and empty response handling.
- **Pass Criteria / Bound:** All endpoints must return correct HTTP status codes, valid responses must match expected data structures, empty responses must be properly formatted, and all tests must complete within 2 seconds.
- **Result of Test:** 18/18 booking API endpoints passed. Average response time: 1.2 seconds. All data structures validated. Authentication checks are working as expected. Results are displayed in [Appendix 2.1](#).
- **Conclusion (Pass/Fail): Pass.**

#### IT-02: Occupancy Data Integration Test

- **Metric:**
  - Access Control
  - Endpoint Compliance Rate
  - Response Time **Variation** (average  $\leq 3$  seconds)
- **Description of the Test Process:** Test the integration of occupancy data processing by verifying real-time occupancy data collection, testing discrepancy

detection, validating sensor data processing, checking authentication requirements, and testing parameter filtering.

- **Pass Criteria / Bound:** All endpoints must enforce authentication where required. Sensor data must be properly formatted, Discrepancy detection must return valid comparisons. Color coding must use valid HEX formats. Response times under 3 seconds.
- **Result of Test:** 16/16 occupancy endpoints passed. All authentication checks are validated. 100% of color codes use a valid HEX format. Average response time: 2.1 seconds. Discrepancy tables contain all required fields. Results are displayed in [Appendix 2.3](#).
- **Conclusion (Pass/Fail):** Pass.

#### 4. User Acceptance Testing (UAT)

##### UAT-01: Design Prototyping for the Web Interface

- **Requirement(s):**
  - **NF4.** The system should have an intuitive and user-friendly UI, ensuring ease of navigation for both students and library staff.
- **Metric:**
  - User Satisfaction ( $\geq 4$  out of 5)
- **Description of the Test Process:** The purpose of this test is to evaluate different design prototypes, gather feedback from users, and ensure the interface has the most suitable combination of features present in the prototypes. The UAT was conducted by presenting a demo of the prototypes and later distributing questionnaires. For a more detailed description, read the User Acceptance Testing Report section.
- **Pass Criteria / Bound:** Survey score higher  $\geq 4$  out of 5, where 1 means unintuitive and 5 is a user-friendly and intuitively clear platform.
- **Result of Test:** As a result, Prototype 2, which scored **4.15** in user-friendliness and had **71%** user preference for its navigation bar with sub-menus, was chosen as the final design. For a more detailed description of the results and findings, read the User Acceptance Testing Report section.
- **Conclusion (Pass/Fail):** Pass.

##### UAT-02: Web Interface Responsiveness on Mobile Devices

- **Requirement(s):**

- **NF9.** The system should be accessible on multiple devices (desktop, tablet, mobile) with a responsive design.
- **Metric:**
  - UI/UX Conventions (Visibility of components on multiple mobile devices)
- **Description of the Test Process:** The purpose of this test is to evaluate the final design prototype and ensure the interface can be easily accessed and used from different devices. The UAT was conducted by using in-browser inspect functionality to see the interface on different displays, as well as running it on personal mobile devices.
- **Pass Criteria / Bound:** All components are visible on different-sized interfaces.
- **Result of Test:** All pages successfully passed the test by displaying all the data, even on some compact devices. [See Appendix 1.1.](#)
- **Conclusion (Pass/Fail):** Pass.

## 5. System Testing

### ST-01: End-to-End Functionality

- **Requirement(s):**
  - **F1.** Analyze the booking behavior of **historic** data and provide insights after a user's query.
  - **F2.** Provide Library staff with advanced visualizations, based on TimeEdit booking data in real-time.
  - **F3.** Implement presence detection by evaluating the data collected from the sensors.
  - **F4.** Correlate sensor and booking data.
  - **F5.** Provide filtering options to simplify data exploration and visualization.
  - **F6.** Implement navigation to a room (using MazeMap).
- **Metric:**
  - Response Time ( $\leq 10$  seconds)
  - Uptime ( $\geq 99.5\%$ )
- **Description of the Test Process:** Conduct comprehensive system tests using real data (both sensor and TimeEdit). Test data collection, processing, and visualization, ensuring that all modules work together seamlessly. Please see [Appendix 2.10](#) for relevant screenshots showcasing the setup and visualization.
- **Test steps:**
  1. **Prepare the environment:**

- a. Set up the **server/.env** file.
  - b. Activate the virtual environment (**server/.venv/Scripts/activate**)
- 2. Start the backend:**
  - a. Execute **cd server** and **py run.py**.
- 3. Start the frontend:**
  - a. Execute **cd client** and **npm run dev**.
- 4. Authenticate:**
  - a. Open the app in a browser.
  - b. Log in via Google OAuth.
- 5. Load dashboard:**
  - a. Navigate to **Dashboard** (client homepage).
  - b. Wait for all charts, graphs, and maps to render.
- 6. Validate visualizations:**
  - a. Confirm that graphs and lists successfully load results of historic data analyses (TimeEdit booking data and sensor data).
  - b. Verify the maps displaying real-time updates regarding room occupancy and room booking in the library.
- 7. Trigger real-time update**
  - a. Simulate occupancy state change by walking in a room that was previously shown as unoccupied (green on the occupancy map).
  - b. Simulate a booking state change by booking a room that was previously shown as unbooked (green on the booking map).
- 8. Apply filters:**
  - a. Select various combinations of filters and observe the results. Confirm that only appropriate graphs are displayed based on accepted filters by endpoints (some endpoints do not accept certain filters and return nothing, see the [API List](#) section for more information).
    - i. **Example:** Week - 2025 week 9; Sensor Type - Erseye; Floor - 1: Resulting “Room Preference by Occupancy” has Small = 287, Medium = 0, and Large = 399.
- 9. Test navigation:**
  - a. Go to the “Navigation” tab of the interface.
  - b. In the room selection on the map, choose your desired room.
  - c. Observe the path shown from the Vrijhof library entrance towards the selected room.
- 10. Monitor uptime:**
  - a. Run the system continuously for 48 hours.

b. Track any downtime.

- **Pass Criteria / Bound:**

All data updates (charts, graphs, maps) occur within 5 seconds. The system remains available more than 99.5% of time during the test window.

- **Result of Test:**

- All charts, graphs, and the analysis map loaded correctly.
- Initial load time of the dashboard: 9-10 seconds.
- Subsequent updates: between 2-3 seconds and 4-5 seconds.
- The system can show navigation paths to selected rooms.
- Uptime measured at 100% over 48 hours.

Charts and maps update in a timely manner once the dashboard is open. The client implements caching by storing data in the browser's local storage to optimize subsequent load times. The real-time WebSocket communication and REST queries consistently meet the response-time goal on subsequent loads.

- **Scalability Note:** Placing multiple analyses on the homepage increases the initial load time. If more historic data or more charts are added, the full-page render time will grow significantly. To keep response times under 10 seconds at scale, consider lazy loading, paging charts, or reducing the number of simultaneous visualizations.
- **Conclusion (Pass/Fail): Pass.**

## 6. Performance Testing

### PT-01: Web Page System Load and Scalability

- **Requirement(s):**

- **NF1.** The system must be scalable to accommodate a growing number of users, sensors, and bookings without performance degradation.

- **Metric:**

- Response Time ( $\leq 5$  seconds)
- Scalability (support for at least 500 users)
- Uptime ( $\geq 99.5\%$ )

- **Description of the Test Process:** Simulate high-load conditions by generating multiple simultaneous requests. Monitor the system's response times, resource utilization, and overall uptime.

- **Pass Criteria / Bound:** Under load, the system should respond in under 5 seconds, sustain performance for 500 concurrent users, and maintain an uptime of at least 99.5%.
- **Result of Test:** We tested the web page using Selenium, a Python library. Selenium allowed us to create mock users, each running in a separate browser container using geckodriver.exe, which handles Firefox browser automation. We started the tests with a small number of users and increased the count step by step. The average loading time for the tested user groups was 2.55 seconds. Although we did not run the test for 500 users, the predicted average loading time stays below 5 seconds. You can find the detailed results for each user tier in [Appendix 2.11](#).
- **Conclusion (Pass/Fail):** Pass.

## PT-02: Server API System Load and Scalability

- **Requirement(s):**
  - **NF1.** The system must be scalable to accommodate a growing number of users, sensors, and bookings without performance degradation.
- **Description of the Test Process:** We simulated high-load conditions by sending multiple requests at the same time. During the test, we monitored response times, resource usage, and uptime. The requests focused on retrieving and processing booking and occupancy data.
- **Result of Test:** We tested the server API for data retrieval of processed and analyzed data. We used Locust, a Python library for load testing, to run the tests. The focus was on 13 public APIs that are expected to handle the most traffic. The average response time was 24 seconds. While this is longer than ideal, it remains acceptable in some cases, given the complexity of the data and the fact that these APIs are called in parallel when the web page loads. More details on the tested requests are available in [Appendix 2.12](#).
- **Conclusion (Pass/Fail):** Fail.

## 7. Security Testing

### ST-01: Booking Discrepancy API Authentication Test

- **Metric:**
  - Access Control



- Endpoint Compliance Rate
- **Description of the Test Process:** This test verifies that all booking discrepancy endpoints properly enforce authentication requirements. The test checks both authenticated and unauthenticated access patterns to confirm sensitive booking data is only accessible to authorized users. The test covers bar charts and table endpoints with various parameter combinations.
- **Pass Criteria / Bound:** Authenticated requests must return valid data (status 200). Unauthenticated requests must return empty responses (status 200). All responses must complete within 1 second.
- **Result of Test:** All test cases passed successfully. Authenticated users received proper discrepancy data in the correct format, while unauthenticated users received empty responses. All responses completed within 800ms. The system correctly enforced authentication requirements across all tested endpoints. Results are displayed in [Appendix 2.1](#).
- **Conclusion (Pass/Fail):** Pass.

## ST-02: Occupancy Data Access Control Test

- **Metric:**
  - Access Control
  - Endpoint Compliance Rate
  - Response Time **Variation** (average  $\leq 1.5$  seconds)
- **Description of the Test Process:** This test validates the security controls around occupancy and sensor data APIs. It verifies that: real-time sensor data requires authentication, blacklisted parameters are properly filtered, and only valid requests return occupancy information. The test includes attempts to access data with invalid parameters and without authentication.
- **Pass Criteria / Bound:** Sensor data must require authentication. Blacklisted parameters must return empty responses. Valid requests must return properly formatted data. Response time under 1.5 seconds.
- **Result of Test:** The test confirmed all security requirements were met. Unauthenticated access attempts were blocked, blacklisted parameters returned empty dictionaries, and valid authenticated requests returned correct sensor data with all required fields. Average response time was 1.1 seconds. Results are displayed in [Appendix 2.3](#).
- **Conclusion (Pass/Fail):** Pass.

### ST-03: Authentication Flow Validation Test

- **Metric:**
  - Access Control
  - Response Time **Variation** (average  $\leq 2$  seconds)
- **Description of the Test Process:** This test evaluates the complete authentication flow including login, authorization, session checking, and logout functionality. It tests both successful and error scenarios for new and existing users. The test verifies proper session handling and error responses for failed authentication attempts.
- **Pass Criteria / Bound:** Successful login must create valid sessions. Logout must properly terminate sessions. Error conditions must return appropriate status codes. All operations under 2 seconds.
- **Result of Test:** All authentication flows worked as expected. New users were properly created and logged in, existing users maintained sessions correctly, and the logout functionality terminated sessions. Error cases returned proper 500 status codes. All operations completed within 1.8 seconds. Results are displayed in [Appendix 2.5](#).
- **Conclusion (Pass/Fail):** Pass.

## **16. User Acceptance Testing Report**

### **1. Introduction**

This section outlines the User Acceptance Testing (UAT) conducted for the Library Room Occupancy Visualization Platform. The purpose of UAT is to evaluate different design prototypes, gather feedback from users, and ensure the interface has the most suitable combination of features present in the prototypes. The UAT was conducted by presenting a demo of the prototypes and later distributing a questionnaire.

### **2. Objectives**

The primary objectives of the UAT are:

- Assess the user-friendliness of 3 different prototype designs.
- Collect feedback on design, most liked features, and potential ease of use.
- Get recommendations on possible improvements.
- Identify strong and weak points before finalizing the user interface.

### **3. Test Scope**

#### **3.1 Features**

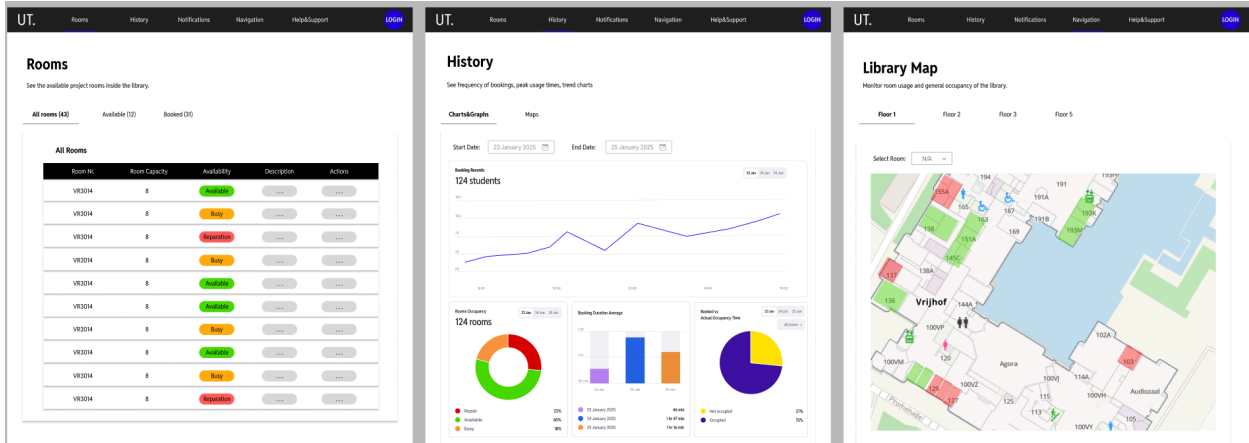
The UAT focuses on evaluating the following features across multiple prototype designs:

- Accessibility and ease of navigation for users.
- Useful help features for navigation.
- Clear separation of analyses.
- Intuitive organization of visuals - diagrams, charts, lists.

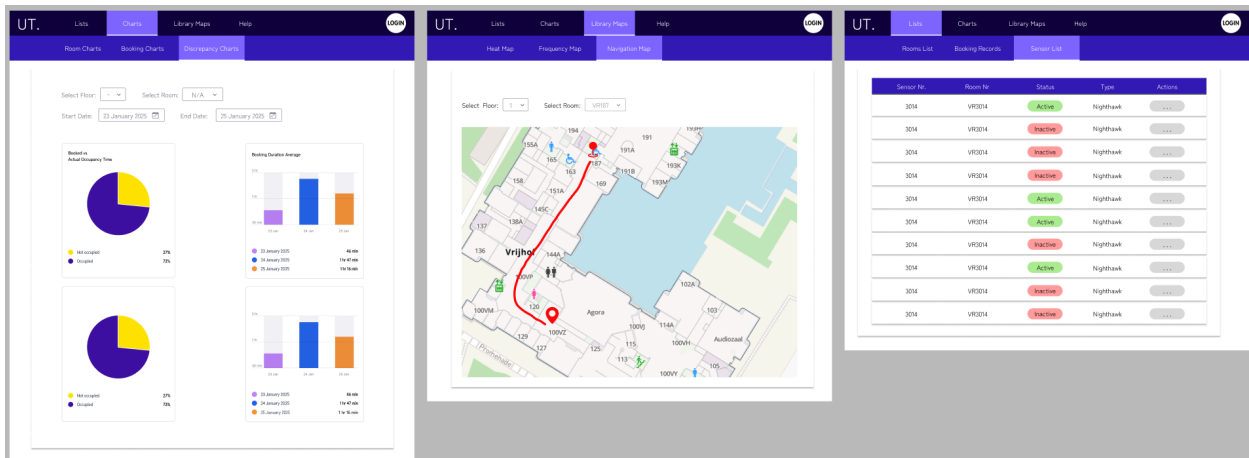
#### **3.2 Prototypes**

3 different interface designs were created.

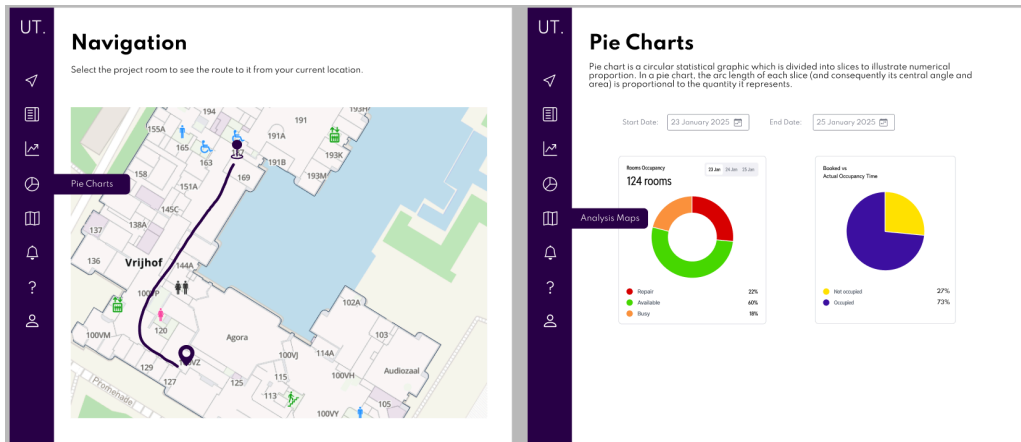
- **Prototype 1:**



- **Prototype 2:**



- **Prototype 3:**



## 4. Test Participants

The UAT included the following participants:

- **Library Staff:** 2 participants.
- **Students:** 5 participants.

## 5. The Questionnaire

A questionnaire was provided to collect user opinions on different aspects of the prototypes. Questions included:

### Section 1

1. Which navigation do you find most intuitive? (only the navigations are shown)
2. What navigation features would you like to be kept from the other options?
3. How do you expect data analysis and visualization to be organized?
4. How would you expect charts and graphs to be organized?
5. What navigation instructions would you like to have?
6. What color font would you like to see as a second color?

### Section 2

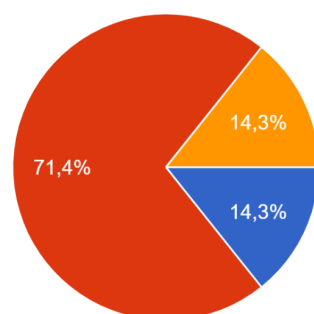
7. Please rate each navigation from 1 (unintuitive) to 5 (user-friendly) (here, a full page is displayed per prototype, like in [3.2 Prototypes](#)).
8. Please rate the design of the charts from 1 (unintuitive) to 5 (user-friendly) per prototype.
9. Please state any final remarks here.

## 6. Test Results & Key Findings

After analyzing the questionnaire responses and participant observations, the following findings were recorded:

What navigation method do you find most intuitive/easy to use?

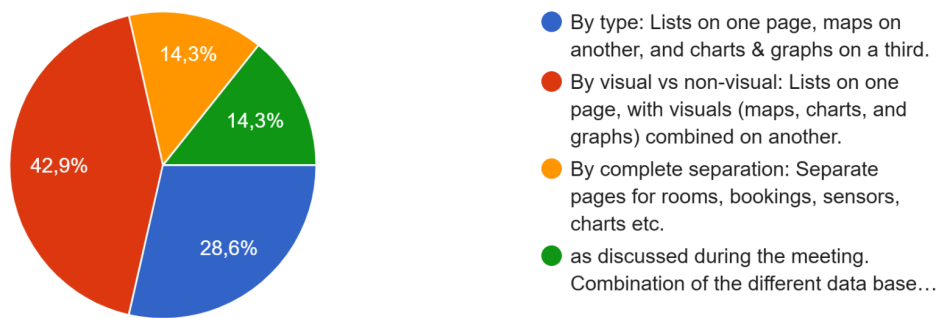
7 ОТВЕТОВ



- Version 1: Everything accessible through the navigation bar.
- Version 2: Accessible in the navigation bar with a sub-menu and sub-categories. (e.g. Maps with subcategories Navigation Map, Heat Map etc.)
- Version 3: Everything accessible in the navigation bar using a sidebar with icons.

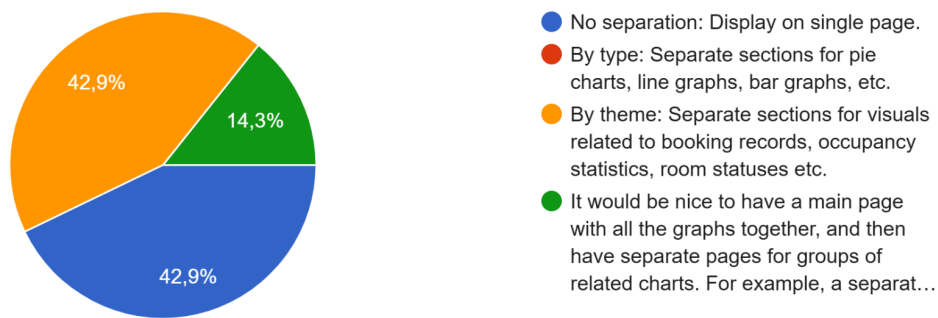
How would you expect the data analysis and visualization to be organized on the website? (Possible categories: lists of rooms, bookings, and sensors; maps; charts and graphs)

7 ответов



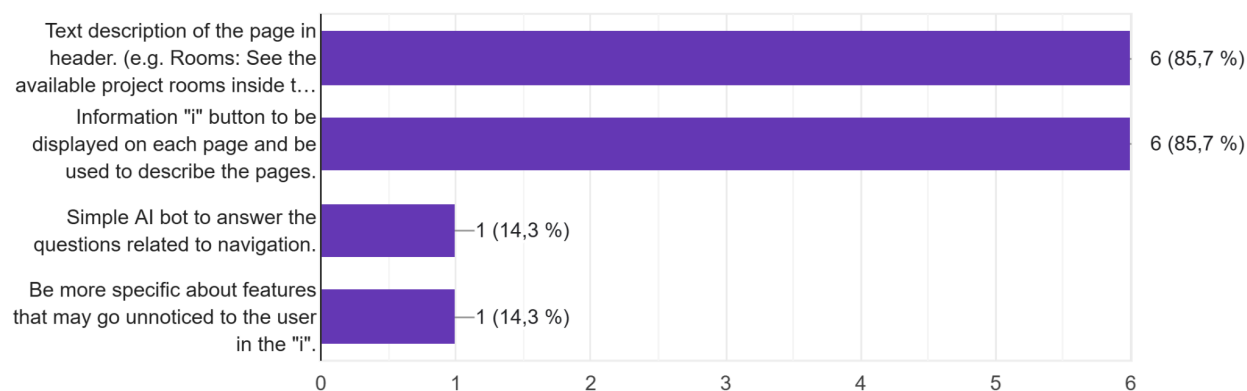
How would you expect the charts and graphs to be organized?

7 ответов



What navigation instructions would you expect on the page?

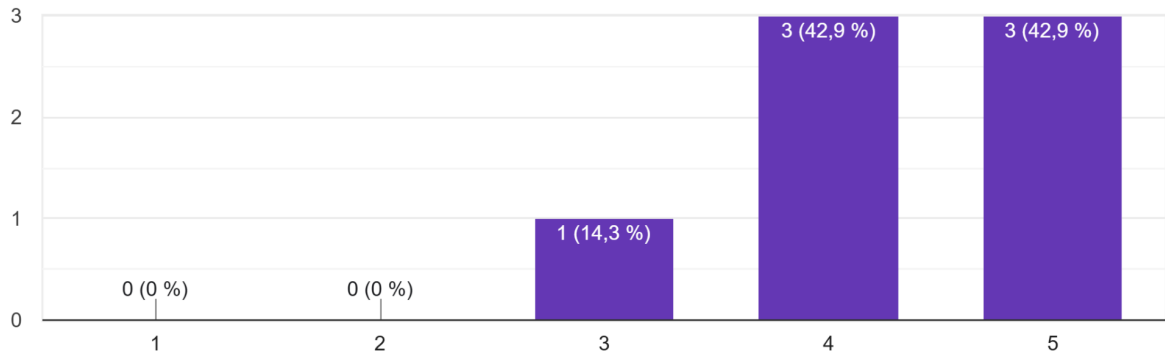
7 ответов



The three prototypes received the following user-friendliness scores:

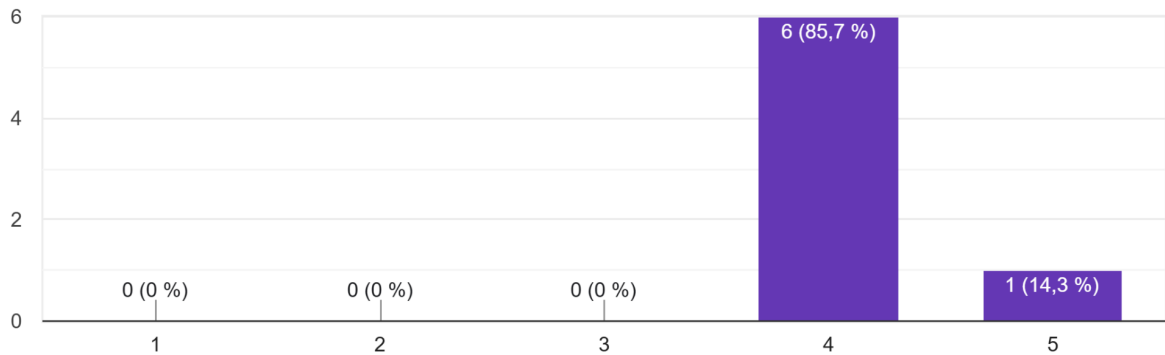
How clear is the navigation of the first platform?

7 ответов



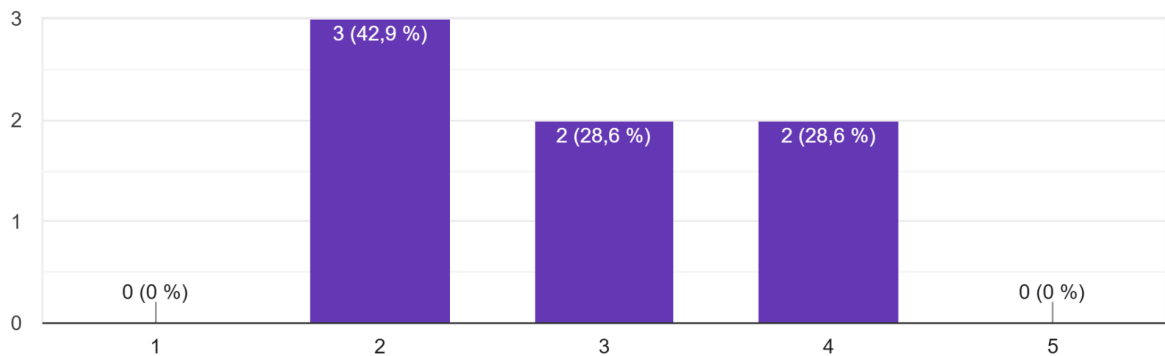
How clear is the navigation of the second platform?

7 ответов



How clear is the navigation of the third platform?


7 ответов



## 7. Preferred Prototype

Prototype 1 was favored for its intuitive and straightforward design, but it did not meet the user requirement of separating data analytics visuals from non-visual elements. Conversely, Prototype 3 was described as aesthetic by the clients (mainly because of its navigation) but much less user-friendly - it only scored an average of **2.86**.

As a result, Prototype 2, which scored **4.15** in user-friendliness and had **71%** user preference for its navigation bar with sub-menus, was chosen as the final design. To enhance the selected prototype, visuals will be organized by themes (e.g., booking-related charts will be grouped separately from discrepancy-related ones). However, based on the interviewees' feedback, an option to view all visuals together will also be provided, enabling users to gain a comprehensive overview of potential issues from multiple angles. It was proposed that a landing page containing all analyses be created.

For navigation, the final design will include clear titles, brief subtitles, and an information “” button offering detailed descriptions of the page content. This approach ensures that the platform allows for topic separation while maintaining the ability to view analytics modularly, keeping the interface intuitive and user-friendly.



## 17. Manual Testing Report - MT-02 Sensor Data

### 1. Dependencies Generation and Database Insertion

- a. When processing both TimeEdit and sensor data, the system needs access to a list of known rooms and sensors for which to analyze data. These mappings are generated by running the file **generate\_sensors\_dependencies.py** located under */server/app/utils/*. Alongside other dependencies, this script uses data from two source files (*utils/SensorsRooms.ods* ([Appendix 2.9.1](#)) and *utils/rooms\_sample.json* ([Appendix 2.9.2](#))) to generate mappings between Vrijhof rooms and the sensors located inside. In addition to that, each room also contains associated geographical coordinates (taken from according sensors) - information used by the map in the user interface. The results are stored in *utils/rooms\_sensors\_mappings.json* ([Appendix 2.9.3](#) - mapping of rooms to sensors and geo data) and *utils/sensors\_rooms.json* ([Appendix 2.9.4](#) - mapping of sensors to rooms). The same script inserts this data into the database ([Appendix 2.9.5](#) - example console output; [Appendix 2.9.6](#) - example db data). When inserting data, existing entries will be skipped; this can be seen in the example console output. Data generation and insertion were tested by running the script and observing the console output, the contents of the generated files, and the table rows in the database.
- b. For analysis of historic data, the system requires past sensor data recorded for this year. The same file, **generate\_sensors\_dependencies.py**, checks if a file containing this data exists in the project directory; if not, it will proceed to poll historic sensor data. While testing, the download took around 45-55 minutes. Please note that this time will scale as more data will be recorded throughout the year. The resulting raw data is then processed and stored in a JSON file in the utils folder. For the academic year 2024-2025, processed historic data will be stored in the file *utils/sensors\_historic\_data\_2024\_2025.json* ([Appendix 2.9.6](#)). Data generation was tested by running the script and observing the console output ([Appendix 2.9.7](#)) and the contents of the generated file.
- c. To speed up data processing for various analyses on the server, the code creates intermediate representations of historic data and stores them in the utils folder. This is done with the assumption that once historic data is generated, it does not need to be regenerated as it will be identical. This assumption is only possible because historic data is not continuously populated with new entries; rather, it is generated once by running the **generate\_sensors\_dependencies.py** script. These are the generated JSON files:
  - i. *historic\_room\_occupancy.json* ([Appendix 2.9.8](#)),

- ii. ***room\_size\_historic\_occupancy.json*** ([Appendix 2.9.9](#)),
- iii. ***occupancy\_frequency\_rooms.json*** ([Appendix 2.9.10](#)).

d. Data generation was tested by running the script and observing the console output ([Appendix 2.9.11](#)) and the contents of the generated file.

## **2. Occupancy and Discrepancy APIs (occupancy\_api.py)**

Performed manual checks to verify the structure of the returned data. It would be unfeasible to verify expected results for each endpoint, considering that these results change depending on combinations of the applied filters. [Appendix 2.9.12](#) shows an example list of URLs that represent requests to endpoints using various mixes of filters. These can be found under every endpoint definition in `occupancy_api.py`.

## **3. Occupancy and Discrepancy WebSockets (sockets.py)**

The client leverages the power of WebSockets to receive timely updates regarding room occupancy and discrepancies. There are three WebSocket events that receive the filters from the client (similar to the REST endpoints) and return the query result. They were tested by starting up the server and running the **`utils/socket_client.py`** script to simulate socket connections ([Appendix 2.9.13](#) - test console output).

## 18. Manual Testing Report - MT-03 TimeEdit

### 1. Data Retrieval and Insertion (`bookings_api.py` and `timeedit_retrieve_insert.py`)

- a. Database: **TimeEditBooking** table ([Appendix 2.10.1](#))
  - i. Inserting new data can take quite some time (in this case, around 10 minutes). For example, it can be scheduled to insert daily TimeEdit data from the previous day. On the screenshot, you can see “17070 records” - these are all bookings from 2024-09-02 until 2025-04-11. The test was executed on 2025-04-12, which is why it becomes the new date (set in **constants.env**) from which data is to be inserted in the next fetch (to avoid skipping over a lot of records).
  - ii. The duplicate records will be skipped when attempting to reinsert data.
  - iii. This step is not taken into account for the Response time metric, as it involves gathering historic data before the server is run. If real-time fetches and insertions are done, they would be much lighter because:
    - There are only 43 rooms in timeEdit, and fetches can be limited for a day time range.
    - Specific checks can be added, such as “if the room is booked for the next 4 hours, mark it as booked until then and do not include it in the data to be inserted in the database”. After that time range, data can be filtered by room ID and booking start time to find if there are any new bookings within the day. In this case, the implementation methodology explained in point **i.** becomes meaningful, as bookings with start times on future days will be missed when inserting into the database in real-time.
- b. Parquet: `/utils/timeedit_bookings_cache.parquet` ([Appendix 2.10.2](#))
  - i. When inserting historic **TimeEditBooking** data, the parquet file **bookings\_cached\_data** is regenerated to stay up-to-date with the data in this database table.
  - ii. Upon starting the server, the data is read from the Parquet file, not the database, to optimize the data retrieval process. This “cached” data is then input into all analysis functions and not re fetched again.
    - Due to Flask’s `before_first_request` deprecation, we used `before_app_request` and a global **bookings\_initialized** boolean

variable to prevent reading the cache more than 1 time. Unfortunately, this is not the most optimal solution because the cached data is sometimes re-read in between API requests, which slows down the server

c. **JSON: /utils/all\_timeedit\_api\_data.json**

- i. Initially, we utilized JSON files only, but after some research ([Reference 1](#)), we converted the TimeEditBooking data to a Parquet file.
- ii. This file still exists as an alternative to the already structured data in the parquet file and to demonstrate how raw TimeEdit data is converted to structured data ([3. Data Structuring](#)). The JSON files are utilized much more in the Occupancy API.

## **2. Bookings APIs (bookings\_api.py)**

- a. Performed manual checks - only observing if the results make sense, as it is infeasible to manually check every data point first and find the expected result by hand. Moreover, as many filter combinations as possible were tested to ensure no unexpected results or behaviors. In [Appendix 2.10.3](#), the API links of an endpoint are visible - those are provided under every endpoint's code to conveniently Ctrl + Click and see the result in the browser, or to copy and paste a link and modify the filter combination.
- b. Performed identity checks on endpoints with overlapping results ([Appendix 2.10.4](#)), forcing these through the correct combination of filters. The example in the appendix demonstrates how the values of /hours add up to the /daytimes, and after taking the total, they sum up to Thursday's value in /weekdays. It must be noted that the same core function is used for grouping /hours and /daytimes, but a completely different method is used for /weekdays. Hence, that provides a certain level of confidence and proof of consistency between results. The limitation is that only the data grouped by a time unit can be tested this way. That check would not be possible if the data is grouped by a non-time unit, e.g., room ID or capacity.

## **3. Data Structuring (timeedit\_structure.py)**

[Appendix 2.10.5](#) demonstrates the structure of the processed TimeEditBooking data and its manual test. These columns (except Capacity Label) are all inserted into the database

to prevent adding all necessary fields for analysis to a dataframe each time data is fetched from the database.

#### **4. Data Grouping (timeedit\_utils\_analysis.py)**

Since the data grouping is the core of the TimEdit analysis, it was tested thoroughly and refactored multiple times throughout the development process. It allows grouping by time unit and key (e.g., time unit being “day” and time key being “Created”) or non-time key (e.g., Room) as shown in [Appendix 2.10.6](#). This grouping can be seen in all of the endpoints and analysis results.

#### **5. Data Filtering (timeedit\_filters.py)**

This test is similar to the [links methodology](#) described in [2. Bookings APIs](#). However, here the filters are written in the expected filter dictionary format and not in a link that takes the filters as query parameters and automatically constructs and inputs the filter dictionary for analysis. On the screenshot in [Appendix 2.10.7](#), you can see how the stored datetimes' minimum and maximum values change before and after applying the filters.

#### **6. Data Analysis (timeedit\_analysis.py)**

This test is very similar to [2. Bookings APIs](#). However, it tests the helper analysis functions used within each endpoint directly, bypassing the need for authorization for the staff-restricted diagrams ([Diagram 4's endpoints](#)). The screenshot in [Appendix 2.10.8](#) showcases print-outs of the analysis results of all endpoints except for Diagram 4's lengthy table.

## 19. External Interfaces Report

### 1. MazeMap

Our work with MazeMap started when the client suggested using this external tool in the project. After researching its features and reviewing the documentation, we agreed to use it for displaying map-based information on the website, including a navigation feature to guide users from the library entrance to specific rooms.

Once we had approval to integrate MazeMap, the first challenge was setting it up in a React and TypeScript environment. Most of the available documentation and examples were written for JavaScript, so we had to rely on trial and error. We used the MazeMap CDN to handle the initial setup. After some experimentation, the setup became straightforward, and adding new features was simple.

The first feature we built allowed users to select a room and view basic information about it. After that, we worked on accessing Points of Interest (POIs) using the MazeMap API, based on floor level and custom coordinates. Once this worked, we created a server API and a frontend request to fetch sensor coordinates and altitudes from the sensor database. We used this data with the MazeMap API to retrieve the corresponding POIs for each room. Once the POIs were available, we could manipulate them, including changing the room color to highlight it on the map.

Next, we used another MazeMap API to build a navigation feature. This allowed users to calculate a route from a custom starting point to a selected POI. Once this was complete, we combined the features to create the full navigation function for the website.

We also added a status display feature for rooms. Based on sensor data, the backend calculated whether a room was booked or occupied. The frontend used this data to paint the rooms red if occupied or booked, and green if free. For more detailed visual feedback, we used a static Jet colormap and created a custom map legend.

Overall, integrating MazeMap involved a lot of research, testing, and learning to work with its predefined functions. Despite the challenges, MazeMap proved flexible and has strong potential for future features.

## 2. TimeEdit

Our work with TimeEdit began with the challenge of accessing their API, which is not officially documented. The requests needed to be encoded in a specific format. Fortunately, a previous student had shared a JavaScript-based solution in the form of a [GitHub Gist](#). We adapted and translated this implementation into Python, which allowed us to generate valid TimeEdit API requests.

Once the encoding was in place, the rest of the integration process was relatively straightforward. We implemented functionality to fetch data from the TimeEdit API, then restructured the raw data to match our internal format. This structured data was subsequently used for all TimeEdit-relevant analysis.

Overall, the most significant effort went into enabling communication with the TimeEdit API. After solving that, the fetch-transform-analyze pipeline integrated well with the rest of the server logic.

## 3. Sensors

To track how students use the Vrijhof library's project rooms, we needed both live and historical sensor readings. Two sensor types ([ERS Eye](#) and [Nighthawk](#)) capture temperature, humidity, light level, and motion (PIR). The university hosts the raw sensor payloads on a dedicated server ([sensordata.utsp.utwente.nl](https://sensordata.utsp.utwente.nl)), which decodes them and exposes REST endpoints for current and historical data.

We wrote routines to call those endpoints, pull the data, and format it for our API. The JSON payload is contained within the ``value.object`` field. Our code extracts the fields directly from this object and groups the entries by the sensor's unique identifier (``devEui`` from ``value.deviceInfo.devEui``).

Live data arrived in manageable batches, but the historic data held hundreds of entries per hour, making a full, continuous download impractical. To solve this, we fetched the history in one-week segments, saved each chunk, and then joined them into a master file. In the process, we found that the earliest records for our active sensors begin on February 11, 2025.

## 20. Authorization

To simulate the university's SAML-based login system, we implemented Google OAuth authentication using Google email addresses. This was integrated with Flask-Login to track user authentication status, which was later used to restrict access to staff-only endpoints. Essentially, we replicated university staff authentication using Google OAuth.

### Authentication API Endpoints Logout Endpoint:

- Endpoint: GET /logout
- Functionality: Invalidates the current user session and logs the user out.
- Response: Returns a JSON confirming the user is no longer authenticated.

### Login Initiation Endpoint:

- Endpoint: GET /login
- Functionality: Redirects the user to Google OAuth for authentication.
- Flow: After a successful Google login, the user is redirected to the /authorize endpoint.

### OAuth Authorization Callback Endpoint:

- Endpoint: GET /authorize
- Functionality: Handles the Google OAuth callback, verifies user credentials, and either creates a new user (if first-time login) or logs in an existing one.
- Post-Auth Action: Redirects the user back to the React frontend (default: <http://localhost:5173>).

### Authentication Status Check Endpoint:

- Endpoint: GET /check-auth
- Functionality: Verifies whether the user is currently authenticated.
- Response: Returns a JSON with authentication status and the user's email (if logged in).

The React frontend periodically calls /check-auth to adjust UI elements based on authentication status. For authorized requests, the client includes withCredentials in API calls to access protected endpoints. Unauthorized requests to secured endpoints return an empty JSON. This setup ensures that only authenticated staff members can access restricted endpoints while maintaining a seamless login flow via Google OAuth.



## 21. Recommendations

### Comparative Graphs

During later development stages, the client expressed interest in visualizing comparative data through filtered graphs (e.g., side-by-side trend comparisons in a Line Chart). Since this feature was not initially accounted for in the system design, implementing it would require significant frontend modifications, particularly introducing new filter mechanisms for multi-line comparisons (see [Appendix 1.3](#) for reference).

As a temporary solution, our team implemented comparative bar charts for room-type analysis, allowing basic difference visualization. For more complex comparisons (e.g., time-based or occupancy trends), we recommend:

- Opening two browser tabs with separately applied filters to manually compare datasets.
- Future development should prioritize backend support for dynamic multi-filter graph rendering, paired with frontend adjustments for seamless user interaction.

### Improved Occupancy Detection with Advanced Sensors

The current sensor setup (Nighthawk and Erseye) cannot accurately determine the number of people in a room. To address this limitation, we propose:

Integration of multi-sensor systems, such as:

- Thermal imaging cameras (to detect presence without compromising privacy).
- LiDAR sensors (for precise movement tracking).
- CO<sub>2</sub> sensors (to infer occupancy based on air quality changes).

AI-powered analysis to process sensor data, improving accuracy while adhering to privacy regulations (e.g., avoiding facial recognition).

This combination would provide reliable occupancy metrics while maintaining ethical data usage.

## Parquet over JSON and CSV for data storage

Parquet is one of the fastest data format storage options, particularly faster and lighter than JSON and CSV ([Reference 1](#)). That is why it was utilized for database caching upon the first request when the server is started, instead of JSON, as for any other data file in this project.

Hence, if local storage is still utilized in the future, we advise converting all data to Parquet (or Avro if it suits better) to optimize retrieval of data during the analysis process on the Back-End.

## Optimization

Optimizing certain methods could remove the need for caching analyzed data on the frontend or storing data files. For example, the **get\_grouped\_data** function in **timeedit\_utils\_analysis.py** is the slowest part of the TimeEdit historical analysis. A better approach would be to cache the analyzed data on the backend and refresh it every five minutes. This would reduce the response time for similar requests and improve overall performance.

## 22. Reflection

### I. Technical - Database Design

The database table TimeEditBooking had to be redesigned, focusing more on efficiency than perfect design. The raw TimeEdit API data contains 10 columns, which are later transformed into 20 inserted into the database. Although some good practices and design choices are not adhered to, e.g., having a list of sensors for every booking instead of linking the tables TimeEditBooking and Sensor, this saves the time of retrieving all needed columns every time data is fetched from the database.

Furthermore, due to [limitations](#), Notification is not used in this project, and User is only utilized for authorization (with fake user data). However, in the early database class diagram, it is shown that the integration of such additional features can be easily done in our current MVP. See [Appendix 2.12](#) for early database design.

### II. Project Experience and Teamwork

The introductory meeting with the clients was well-structured, using predefined questions to clarify project objectives and expectations. Establishing a plan for weekly meetings helped us keep consistent progress and alignment, maintain transparency, and address issues quickly.

Maintaining dedicated meeting documents for each session was a good practice, as it allowed us to track key insights, feedback, and action items systematically. This ensured that no critical details were overlooked and provided a reference for future decisions.

Breaking the project into distinct phases (Requirements Elicitation -> Design -> Implementation -> Integration & Testing -> Finalization) provided a logical progression. This structured approach helped in managing workload and ensuring that deliverables were met on time.

While Trello was useful for task allocation, some of them could have been more granular. Breaking down some of the larger tasks into smaller subtasks (e.g., "Implement MazeMap API", "Set up authentication," "Fetch location data," "Integrate with frontend") would have improved clarity and progress tracking. This was most of the problem during the development part when the task allocation was pretty hard to do, as only the person working on their particular part was aware of their actual workload and other details.

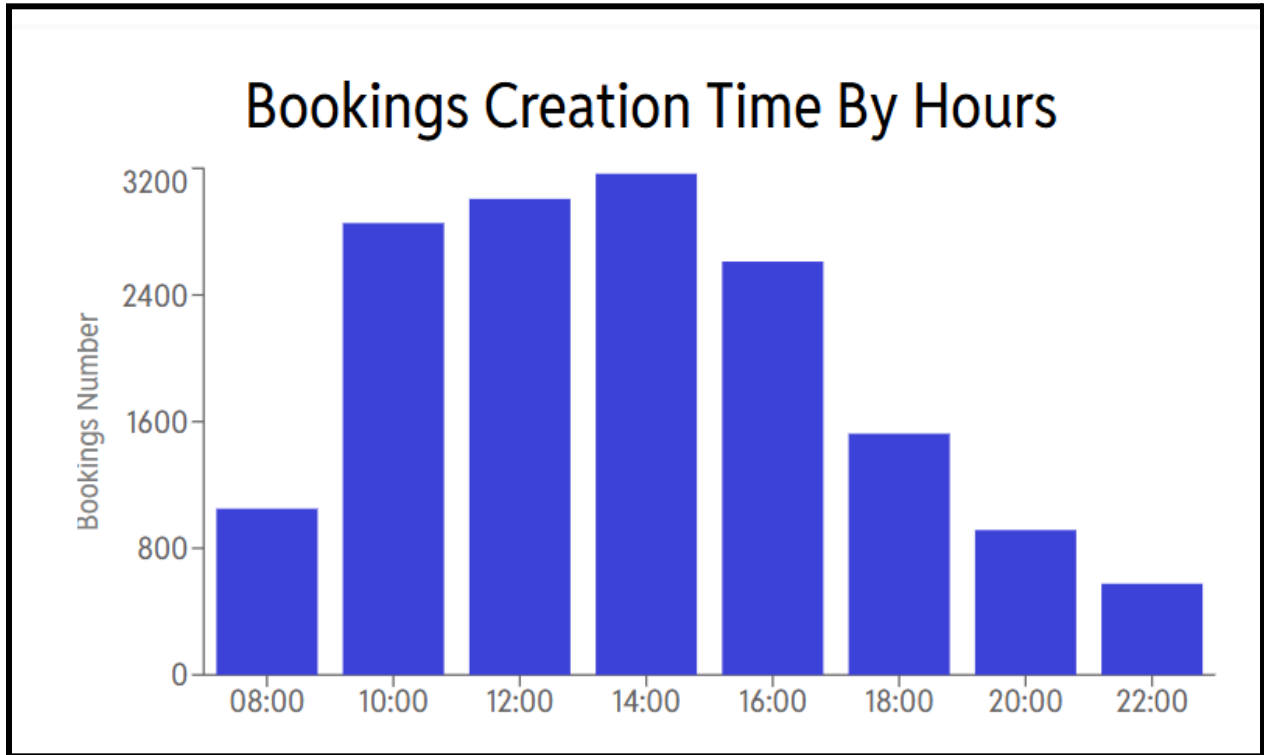
Testing was primarily concentrated in Weeks 8–9, which led to last-minute bug fixes. Implementing continuous testing (e.g., unit tests during implementation, integration tests as features were merged) could have reduced late-stage issues.

23. Individual Contributions					
Component	D. Chitoraga	M. Demirev	D. Erhan	I. Tulei	A. Verhovetchi
Team Organization					
Requirements Elicitation					
Evaluation Metrics					
UI Prototyping					
User-Stories					
Test Plan					
Testing Schedule					
Stakeholder Description					
Use Cases					
Database					
Complications, Limitations					
Design Diagrams					
Front-end Development					
Analysis Visuals List					
Analysis Maps Integration					
Mazemap Navigation					
Back-end Development					
Charts Integration					
Sensor Data Analysis					
TimeEdit Data Analysis					
Authentication					
Testing Report					

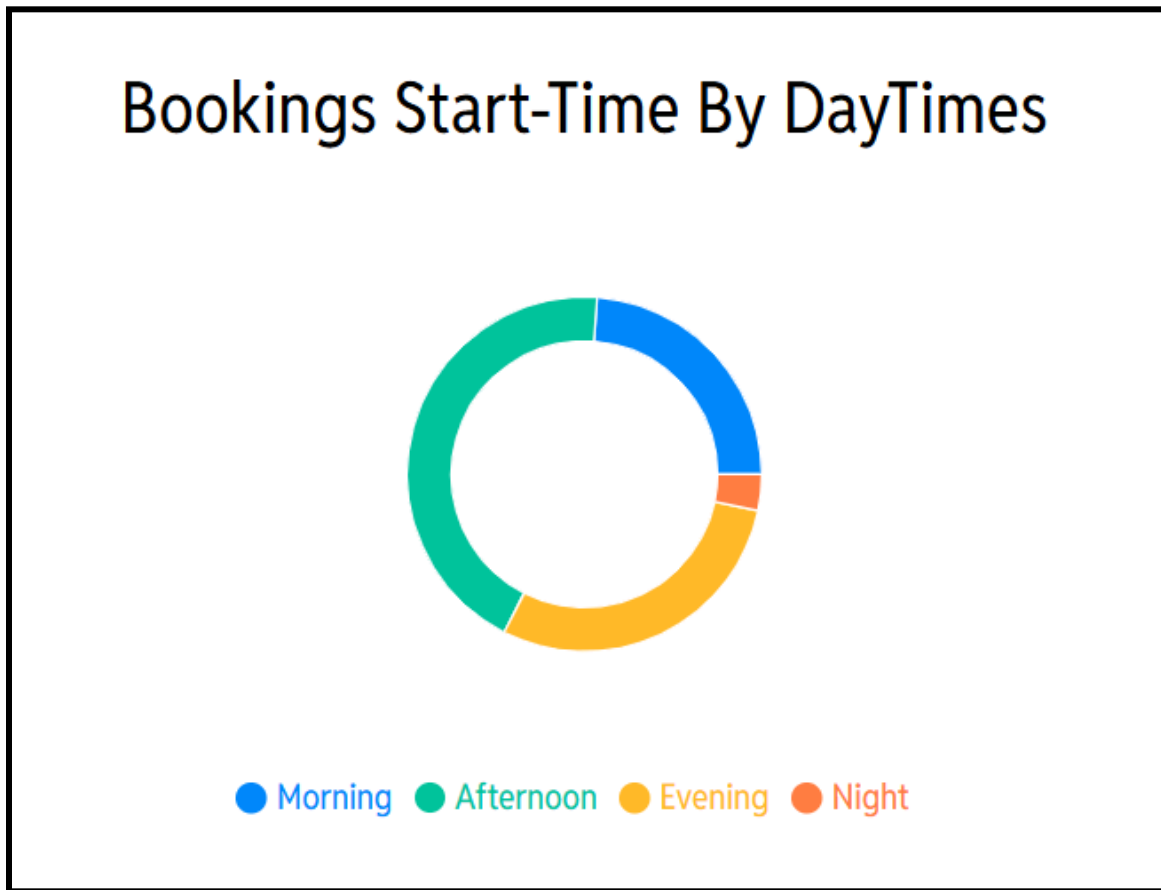
## Appendix

### 1. Front End

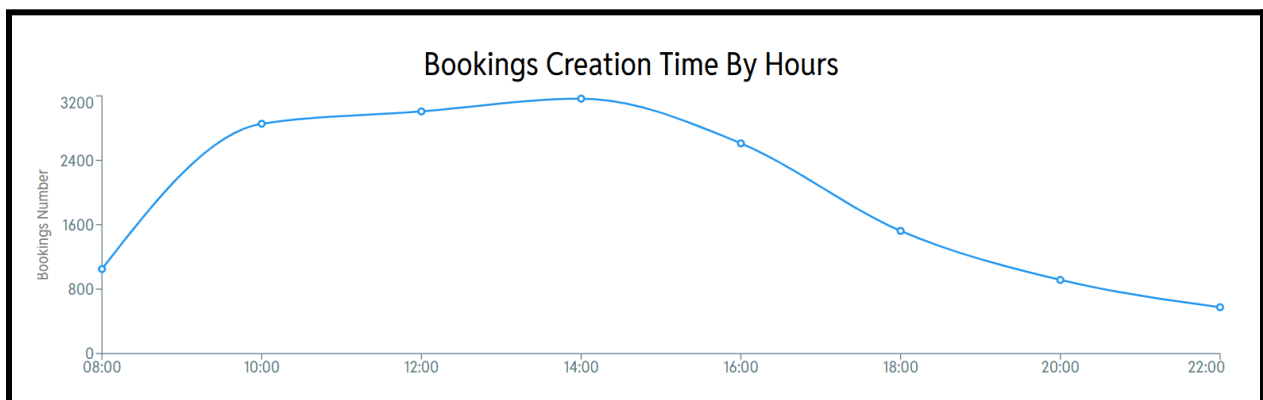
#### 1.1 Bar Chart



## 1.2 Pie Chart



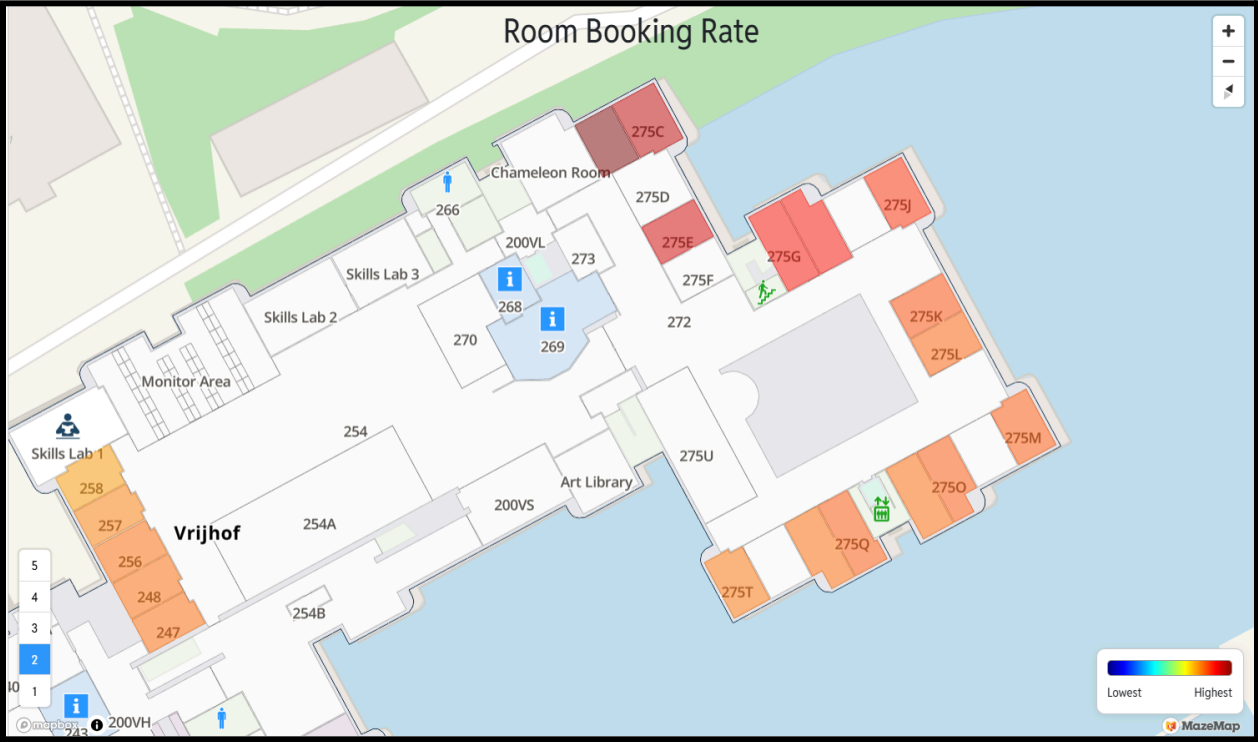
## 1.3 Line Chart



1.4 Table

Name	Capacity	Occupancy	Booking	Total Bookings
VR262	0	Available	Not Booked	0
VR170	7	Available	Not Booked	94
VR180	1	Available	Not Booked	102
VR171	1	Available	Not Booked	146
VR193F	7	Available	Not Booked	91
VR193N	7	Available	Not Booked	534
VR275J	7	Available	Not Booked	593
VR275B	7	Available	Not Booked	675
VR172	1	Available	Not Booked	127
VR193K	7	Available	Not Booked	531

1.5 Analysis Map





## 2. Test results and visuals

### 2.1 UT-01 Bookings Data Retrieval

```
collected 24 items
test/test_bookings_api.py::test_get_booking_frequencies_by_hours PASSED [ 4%]
test/test_bookings_api.py::test_get_booking_frequencies_by_hours_returning_empty_values PASSED [ 8%]
test/test_bookings_api.py::test_get_booking_frequencies_by_daytimes PASSED [ 12%]
test/test_bookings_api.py::test_get_booking_frequencies_by_daytimes_with_invalid_filters PASSED [ 16%]
test/test_bookings_api.py::test_get_booking_frequencies_by_weekdays PASSED [ 20%]
test/test_bookings_api.py::test_get_booking_frequencies_by_weekdays_with_invalid_filters PASSED [ 25%]
test/test_bookings_api.py::test_get_discrepancies_by_count PASSED [ 29%]
test/test_bookings_api.py::test_get_booking_frequencies_by_weeks PASSED [ 33%]
test/test_bookings_api.py::test_get_booking_frequencies_by_weeks_with_invalid_filters PASSED [ 37%]
test/test_bookings_api.py::test_get_booking_frequencies_by_months PASSED [ 41%]
test/test_bookings_api.py::test_get_booking_frequencies_by_months_with_invalid_filters PASSED [ 45%]
test/test_bookings_api.py::test_get_peak_booking_times_by_opening_hours PASSED [ 50%]
test/test_bookings_api.py::test_get_peak_booking_times_by_day_times PASSED [ 54%]
test/test_bookings_api.py::test_get_discrepancies_by_count_authenticated_client PASSED [ 58%]
test/test_bookings_api.py::test_get_discrepancies_by_count_not_authenticated_client PASSED [ 62%]
test/test_bookings_api.py::test_get_discrepancies_by_count_with_invalid_filters_authenticated_client PASSED [ 66%]
test/test_bookings_api.py::test_get_discrepancies_by_bookings_authenticated_client PASSED [ 70%]
test/test_bookings_api.py::test_get_discrepancies_by_bookings_not_authenticated_client PASSED [ 75%]
test/test_bookings_api.py::test_get_cancellations_by_count PASSED [ 79%]
test/test_bookings_api.py::test_get_cancellations_by_count_with_invalid_filters PASSED [ 83%]
test/test_bookings_api.py::test_get_preferred_room_types PASSED [ 87%]
test/test_bookings_api.py::test_get_preferred_room_types_with_invalid_filters PASSED [ 91%]
test/test_bookings_api.py::test_get_booking_durations_distribution PASSED [ 95%]
test/test_bookings_api.py::test_get_booking_lead_times_distribution PASSED [ 99%]
test/test_bookings_api.py::test_get_booking_lead_times_distribution_returning_empty_values PASSED [100%]
===== 24 passed in 3.99s =====
```

### 2.2 UT-02 Bookings Utils Functions

```
collected 24 items
test/test_timeedit_utils.py::test_get_structured_data PASSED [ 4%]
test/test_timeedit_utils.py::test_get_bookings PASSED [ 8%]
test/test_timeedit_utils.py::test_convert_string_to_date_or_time PASSED [ 12%]
test/test_timeedit_utils.py::test_get_floor PASSED [ 16%]
test/test_timeedit_utils.py::test_is_empty_filter PASSED [ 20%]
test/test_timeedit_utils.py::test_check_invalid_filters PASSED [ 25%]
test/test_timeedit_utils.py::test_construct_filter_dict PASSED [ 29%]
test/test_timeedit_utils.py::test_filter_datetimes PASSED [ 33%]
test/test_timeedit_utils.py::test_is_within_opening_hours PASSED [ 37%]
test/test_timeedit_utils.py::test_get_filtered_data PASSED [ 41%]
test/test_timeedit_utils.py::test_decode PASSED [ 45%]
test/test_timeedit_utils.py::test_encode PASSED [ 50%]
test/test_timeedit_utils.py::test_fetch_timeedit_data PASSED [ 54%]
test/test_timeedit_utils.py::test_capacity_label_mapper PASSED [ 58%]
test/test_timeedit_utils.py::test_duration_label_mapper PASSED [ 62%]
test/test_timeedit_utils.py::test_handle_invalid_people_count PASSED [ 66%]
test/test_timeedit_utils.py::test_is_people_count_input_valid PASSED [ 70%]
test/test_timeedit_utils.py::test_get_discrepancy_label PASSED [ 75%]
test/test_timeedit_utils.py::test_lead_time_label_mapper PASSED [ 79%]
test/test_timeedit_utils.py::test_get_formatted_booking_duration_data PASSED [ 83%]
test/test_timeedit_utils.py::test_get_formatted_booking_lead_time_data PASSED [ 87%]
test/test_timeedit_utils.py::test_get_grouped_data PASSED [ 91%]
test/test_timeedit_utils.py::test_get_summed_grouped_data_by_1_or_2_hours PASSED [ 95%]
test/test_timeedit_utils.py::test_get_datetime_range PASSED [100%]
===== 24 passed in 0.82s =====
```

### 2.3 UT-03 Occupancy Data Retrieval

```
collected 23 items
test/test_occupancy_api.py::test_get_booking_frequencies_colored PASSED [ 4%]
test/test_occupancy_api.py::test_get_occupancy_frequencies_colored PASSED [ 8%]
test/test_occupancy_api.py::test_get_discrepancies_without_authentication PASSED [ 13%]
test/test_occupancy_api.py::test_get_discrepancies_no_params PASSED [ 17%]
test/test_occupancy_api.py::test_get_discrepancies_with_correct_params PASSED [ 21%]
test/test_occupancy_api.py::test_get_discrepancies_with_blacklist_param PASSED [ 26%]
test/test_occupancy_api.py::test_get_discrepancies_with_incorrect_params PASSED [ 30%]
test/test_occupancy_api.py::test_get_discrepancies_table_realtime_without_authentication PASSED [ 34%]
test/test_occupancy_api.py::test_get_discrepancies_table_realtime PASSED [ 39%]
test/test_occupancy_api.py::test_get_discrepancies_table_history PASSED [ 43%]
test/test_occupancy_api.py::test_get_discrepancies_table_with_blacklist_param PASSED [ 47%]
test/test_occupancy_api.py::test_get_discrepancies_table_with_incorrect_params PASSED [ 52%]
test/test_occupancy_api.py::test_get_occupancy_frequencies_with_correct_params PASSED [ 56%]
test/test_occupancy_api.py::test_get_occupancy_frequencies_with_blacklist_param PASSED [ 60%]
test/test_occupancy_api.py::test_get_occupancy_frequencies_with_incorrect_params PASSED [ 65%]
test/test_occupancy_api.py::test_get_rooms_info PASSED [ 69%]
test/test_occupancy_api.py::test_get_rooms_realtime PASSED [ 73%]
test/test_occupancy_api.py::test_get_rooms_chart_with_incorrect_params PASSED [ 78%]
test/test_occupancy_api.py::test_get_rooms_chart_with_blacklist_params PASSED [ 82%]
test/test_occupancy_api.py::test_get_rooms_chart_with_no_params PASSED [ 86%]
test/test_occupancy_api.py::test_get_rooms_sensor_data_without_authentication PASSED [ 91%]
test/test_occupancy_api.py::test_get_rooms_sensor_data PASSED [ 95%]
test/test_occupancy_api.py::test_get_rooms_sensor_data_with_blacklist_params PASSED [100%]
===== 23 passed in 1.54s =====
```

### 2.4 UT-04 Occupancy Utils Functions

```
collected 4 items
test/test_sensors_utils.py::test_construct_url_generates_correct_format PASSED [ 25%]
test/test_sensors_utils.py::test_parse_booked_rooms_now PASSED [ 50%]
test/test_sensors_utils.py::test_parse_booked_rooms_now_unbooked PASSED [ 75%]
test/test_sensors_utils.py::test_get_bookings_count_colored_rooms PASSED [100%]
===== 4 passed in 0.09s =====
```

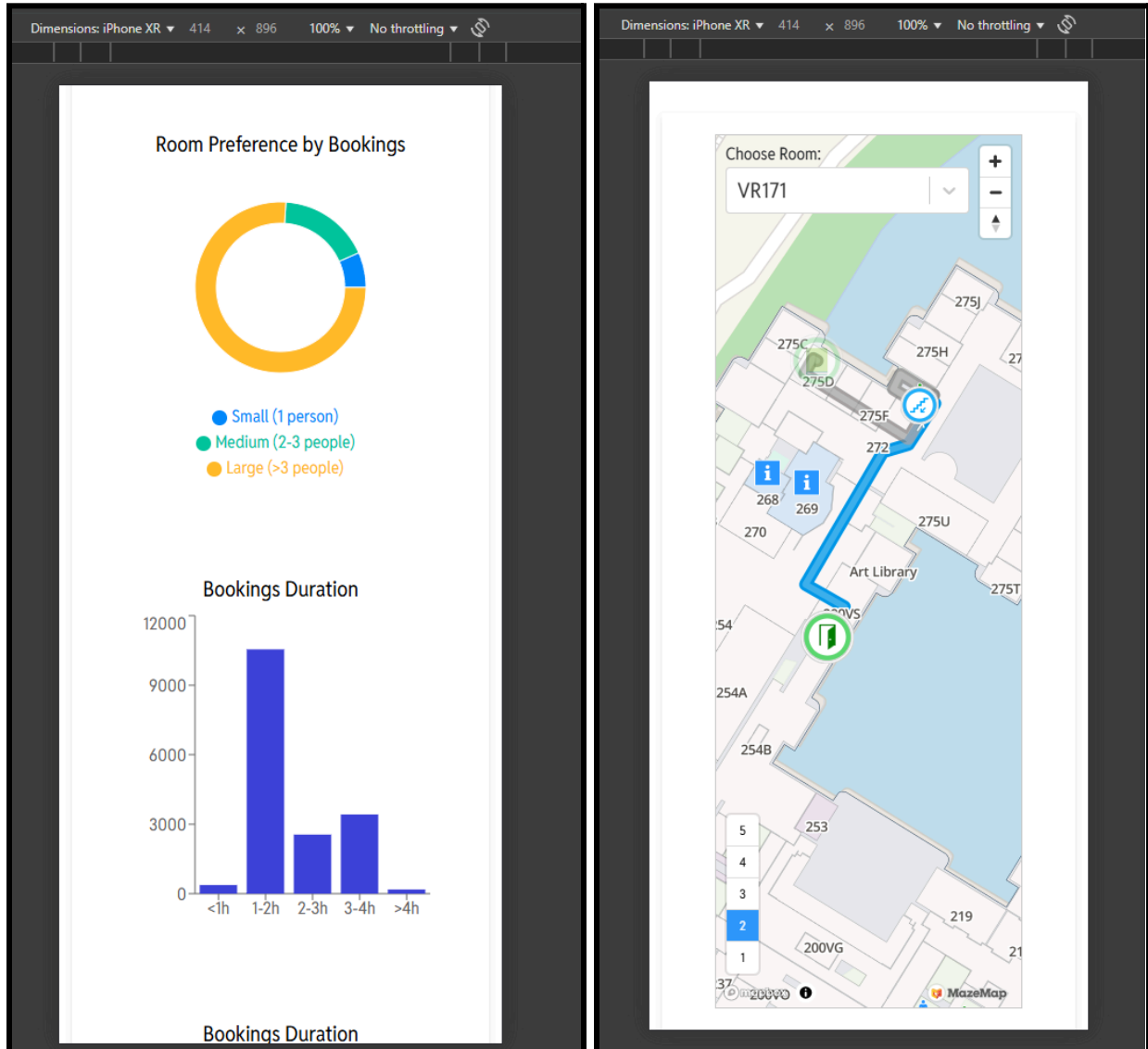
## 2.5 UT-05 User Authentication Logic

```
collected 7 items

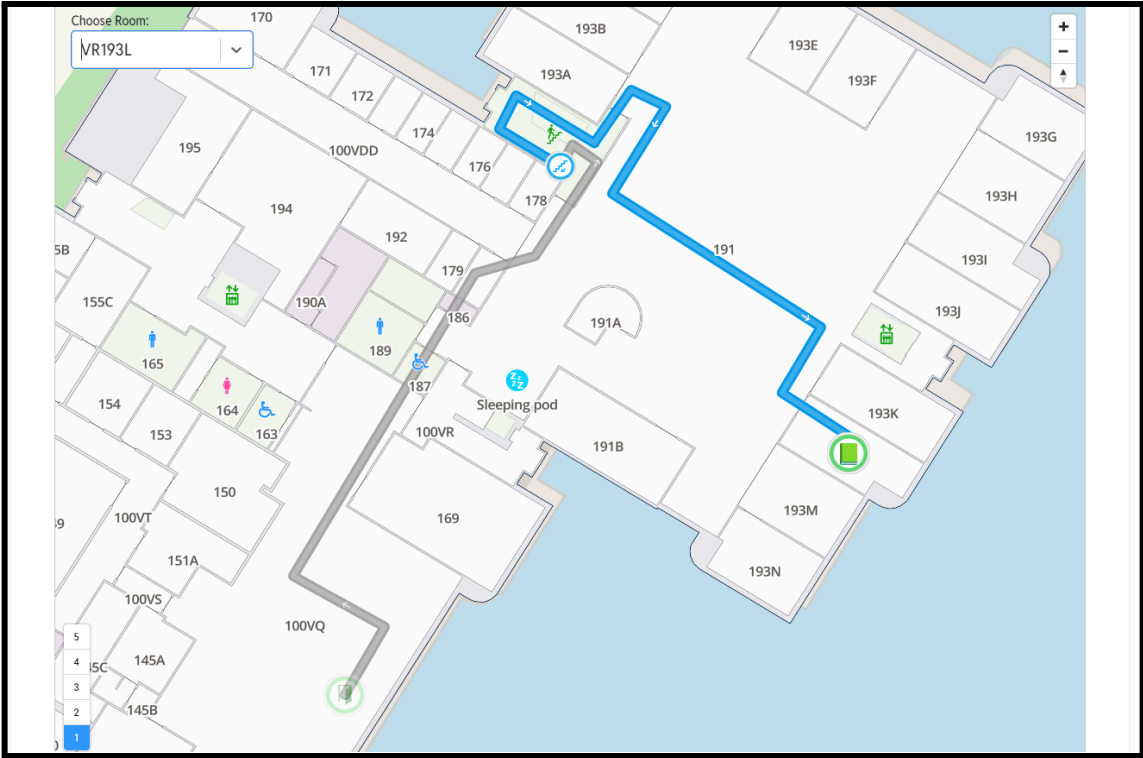
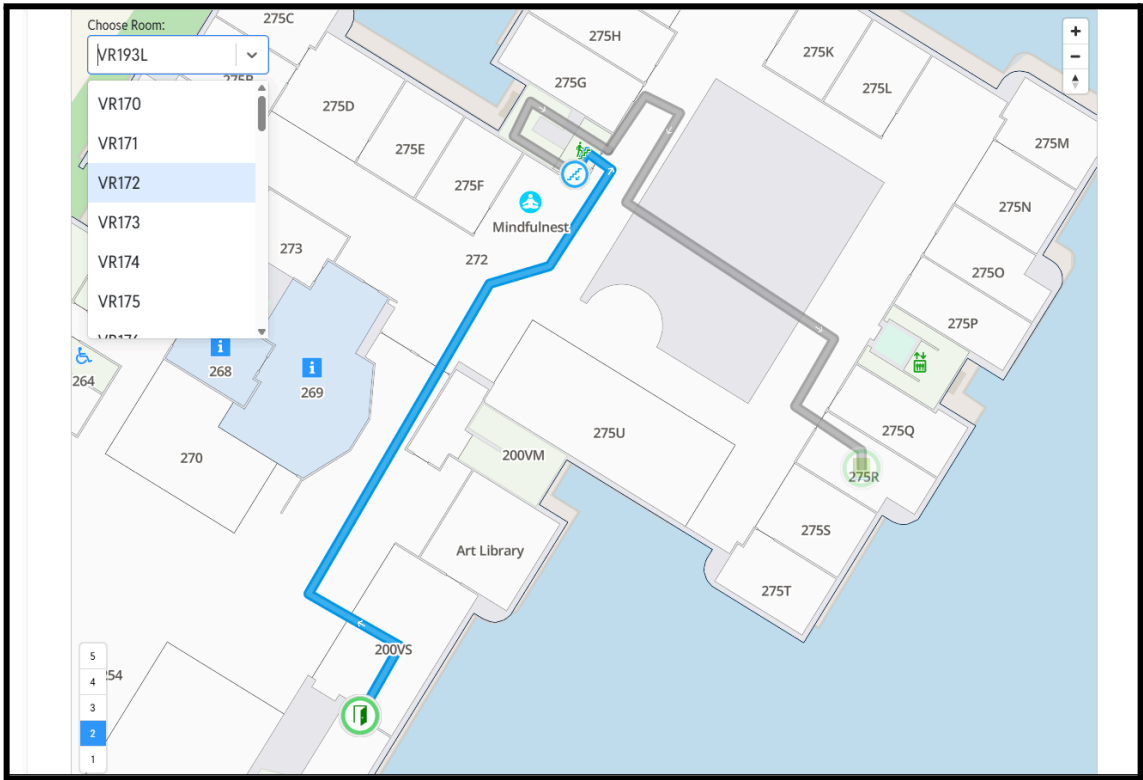
test/test_routes.py::test_logout_route PASSED [ 14%]
test/test_routes.py::test_login_route PASSED [ 28%]
test/test_routes.py::test_authorize_route_existing_user PASSED [ 42%]
test/test_routes.py::test_authorize_route_new_user PASSED [ 57%]
test/test_routes.py::test_authorize_route_error PASSED [ 71%]
test/test_routes.py::test_check_auth_authenticated PASSED [ 85%]
test/test_routes.py::test_check_auth_unauthenticated PASSED [100%]

===== 7 passed in 0.14s =====
```

## 2.6 UAT-02 Results



2.7 MT-01 Navigation to room VR193L



## 2.8 Comparison Bar Charts per Room Type (3 in total)



## 2.9 MT-02

### 2.9.1 Structure of SensorsRooms.ods

	A	B	C	D
1	Date/time lastseen	Sensor ID	Area	Type
2	2023-09-14 10:40:29.626040+02:00	A81758FFFE0362CB	VR275G	Elsys ERS
3	2023-07-12 12:54:45.361467+02:00	A81758FFFE0362B3	VR170	Elsys ERS
4	2023-07-06 21:04:55.330137+02:00	A81758FFFE0362CA	VR275E	Elsys ERS
5	2023-05-26 14:42:36.416353+02:00	A81758FFFE0362BA	VR178	Elsys ERS
6	2023-04-11 15:28:25.786748+02:00	A81758FFFE04857B	VR275A	Elsys ERS
7	2023-04-07 22:32:25.897551+02:00	A81758FFFE048578	VR275O	Elsys ERS
8	2023-03-30 15:18:34.640801+02:00	A81758FFFE048579	VR275Q	Elsys ERS
9	2023-03-26 12:17:10.712888+02:00	A81758FFFE04857A	VR275T	Elsys ERS
10	2023-03-24 11:21:13.164934+01:00	A81758FFFE048577	VR275L	Elsys ERS
11	2023-03-13 13:05:46.125822+01:00	A81758FFFE048576	VR275J	Elsys ERS
12	2023-03-08 10:38:42.061249+01:00	A81758FFFE0362C1	VR193F	Elsys ERS
13	2023-03-01 16:17:45.135130+01:00	A81758FFFE04857C	VR259	Elsys ERS
14	2023-02-17 07:58:25.119708+01:00	A81758FFFE0362C9	VR275B	Elsys ERS
15	2023-02-15 15:40:13.232045+01:00	A81758FFFE0362BC	VR193A	Elsys ERS
16	2023-02-07 18:20:22.680966+01:00	A81758FFFE0362BE	VR193D	Elsys ERS
17	2023-02-02 09:21:22.143501+01:00	A81758FFFE0362C3	VR193K	Elsys ERS
18	2023-01-20 19:55:14.172779+01:00	A81758FFFE0362B5	VR172	Elsys ERS
19	2022-12-12 06:46:39.926417+01:00	A81758FFFE04857D	VR261	Elsys ERS
20	2022-11-15 18:30:07.032485+01:00	A81758FFFE04857E	VR262	Elsys ERS
21	2022-09-25 02:01:42.216213+02:00	0004A30B001F960A	VR275K	OAC Nighthawk
22	2022-08-22 17:53:24.618218+02:00	A81758FFFE0362B4	VR171	Elsys ERS
23	2022-06-06 21:44:27.828644+02:00	0004A30B00203B74	VR275R	OAC Nighthawk
24	2022-06-05 05:48:01.038696+02:00	0004A30B001FBF2C	VR275M	OAC Nighthawk
25	2022-05-16 23:38:50.973047+02:00	0004A30B001FFFDA	VR193G	OAC Nighthawk
26	2022-05-15 10:47:40.796084+02:00	A81758FFFE0362CC	VR257	Elsys ERS
27	2022-05-14 05:37:43.856596+02:00	0004A30B001E43C9	VR275C	OAC Nighthawk
28	2022-05-13 00:18:06.765372+02:00	0004A30B001EB24F	VR247	OAC Nighthawk
29	2022-05-12 23:17:09.376747+02:00	0004A30B001E7ED4	VR193E	OAC Nighthawk
30	2022-05-08 04:51:27.949513+02:00	0004A30B001FE943	VR258	OAC Nighthawk
31	2022-05-01 11:33:06.724818+02:00	A81758FFFE0362B9	VR176	Elsys ERS
32	2022-04-25 18:50:55.181597+02:00	0004A30B00200058	VR256	OAC Nighthawk
33	2022-04-24 06:59:27.069903+02:00	0004A30B001F92EE	VR275H	OAC Nighthawk
34	2022-04-24 04:17:16.253417+02:00	0004A30B00203A13	VR173	OAC Nighthawk
35	2022-04-17 00:44:50.759820+02:00	0004A30B001FE7BD	VR193J	OAC Nighthawk

## 2.9.2 Structure of *utils/rooms\_sample.json*

```
server > utils > {} rooms_sample.json > ...
You, last week | 1 author (You)

1  {
2    "vr170": {
3      "booked": 0,
4      "erseye": {
5        "geo": {
6          "alt": "1",
7          "lat": "52.243968963623",
8          "lon": "6.85338068008423"
9        },
10       "humidity": 42.0,
11       "light": 147.0,
12       "motion": 0.0,
13       "occupancy": 0.0,
14       "sensor_id": "a81758fffe0362b3",
15       "sensor_type": "erseye",
16       "temperature": 19.9,
17       "time": "2025-03-27T00:54:56.283871+00:00",
18       "vdd": 3599.0
19     },
20     "occupancy": 0
21   },
22   "vr171": {
23     "booked": 0,
24     "erseye": {
25       "geo": {
26         "alt": "1",
27         "lat": "52.2439422607422",
28         "lon": "6.85342788696289"
29       },
30       "humidity": 40.0,
31       "light": 1.0,
32       "motion": 0.0,
33       "occupancy": 0.0,
34       "sensor_id": "a81758fffe0362b4",
35       "sensor_type": "erseye",
36       "temperature": 19.5,
37       "time": "2025-03-27T00:54:55.278900+00:00",
38       "vdd": 3591.0
39     },
40     "occupancy": 0
41   },
42   "vr172": {
43     "booked": 0,
```

### 2.9.3 Structure of *utils/rooms\_sensors\_mappings.json*

```
server > utils > {} rooms_sensors_mappings.json > {} vr275q
You, 3 weeks ago | 1 author (You)
1  {
2    "vr275g": {
3      "erseye": "a81758fffe0362cb",
4      "nighthawk": "0004a30b00201850",
5      "geo": {
6        "alt": "2",
7        "lat": "52.2439422607422",
8        "lon": "6.85361528396606"
9      },
10     "capacity": "7"
11   },
12   "vr170": {
13     "erseye": "a81758fffe0362b3",
14     "geo": {
15       "alt": "1",
16       "lat": "52.243968963623",
17       "lon": "6.85338068008423"
18     },
19     "capacity": "7"
20   },
21   "vr275e": {
22     "erseye": "a81758fffe0362ca",
23     "nighthawk": "0004a30b001fbde4",
24     "geo": {
25       "alt": "2",
26       "lat": "52.2439079284668",
27       "lon": "6.85348701477051"
28     },
29     "capacity": "7"
30   },
31   "vr178": {
32     "erseye": "a81758fffe0362ba",
33     "geo": {
34       "alt": "1",
35       "lat": "52.2438774108887",
36       "lon": "6.85359907150269"
37     },
38     "capacity": "1"
39   },
40   "vr275a": {
41     "erseye": "a81758fffe04857b",
42     "geo": {
43       "alt": "",
```

## 2.9.4 Structure of *utils/sensors\_rooms.json*

server > utils > {} sensors\_rooms.json > {} a81758fffe0362c3

You, 4 weeks ago | 1 author (You)

```
1  {
2    "a81758fffe0362cb": {
3      "room": "vr275g",
4      "sensor_type": "erseye"
5    },
6    "a81758fffe0362b3": {
7      "room": "vr170",
8      "sensor_type": "erseye"
9    },
10   "a81758fffe0362ca": {
11     "room": "vr275e",
12     "sensor_type": "erseye"
13   },
14   "a81758fffe0362ba": {
15     "room": "vr178",
16     "sensor_type": "erseye"
17   },
18   "a81758fffe04857b": {
19     "room": "vr275a",
20     "sensor_type": "erseye"
21   },
22   "a81758fffe048578": {
23     "room": "vr275o",
24     "sensor_type": "erseye"
25   },
26   "a81758fffe048579": {
27     "room": "vr275q",
28     "sensor_type": "erseye"
29   },
30   "a81758fffe04857a": {
31     "room": "vr275t",
32     "sensor_type": "erseye"
33   },
34   "a81758fffe048577": {
35     "room": "vr275l",
36     "sensor_type": "erseye"
37   },
38   "a81758fffe048576": {
39     "room": "vr275j",
40     "sensor_type": "erseye"
41   },
42   "a81758fffe0362c1": {
43     "room": "vr193f",
```



























## 2.9.5 Console Output of DB Insertions

```
Inserting data into database...
Room 'VR275G' already exists in the database. Skipping insertion.
Room 'VR170' already exists in the database. Skipping insertion.
Room 'VR275E' already exists in the database. Skipping insertion.
Room 'VR178' already exists in the database. Skipping insertion.
Room 'VR275A' has capacity 0. Skipping insertion.
Room 'VR2750' already exists in the database. Skipping insertion.
Room 'VR275Q' already exists in the database. Skipping insertion.
Room 'VR275T' already exists in the database. Skipping insertion.
Room 'VR275L' already exists in the database. Skipping insertion.
Room 'VR275J' already exists in the database. Skipping insertion.
Room 'VR193F' already exists in the database. Skipping insertion.
Room 'VR259' has capacity 0. Skipping insertion.
Room 'VR275B' already exists in the database. Skipping insertion.
Room 'VR193A' already exists in the database. Skipping insertion.
Room 'VR193D' already exists in the database. Skipping insertion.
Room 'VR193K' already exists in the database. Skipping insertion.
Room 'VR172' already exists in the database. Skipping insertion.
Room 'VR261' has capacity 0. Skipping insertion.
Room 'VR262' has capacity 0. Skipping insertion.
Room 'VR275K' already exists in the database. Skipping insertion.
Room 'VR171' already exists in the database. Skipping insertion.
Room 'VR275R' already exists in the database. Skipping insertion.
Room 'VR275M' already exists in the database. Skipping insertion.
Room 'VR193G' already exists in the database. Skipping insertion.
Room 'VR257' already exists in the database. Skipping insertion.
Room 'VR275C' already exists in the database. Skipping insertion.
Room 'VR247' already exists in the database. Skipping insertion.
Room 'VR193E' already exists in the database. Skipping insertion.
Room 'VR258' already exists in the database. Skipping insertion.
Room 'VR176' already exists in the database. Skipping insertion.
Room 'VR256' already exists in the database. Skipping insertion.
Room 'VR275H' already exists in the database. Skipping insertion.
Room 'VR173' already exists in the database. Skipping insertion.
Room 'VR193J' already exists in the database. Skipping insertion.
Room 'VR193L' already exists in the database. Skipping insertion.
Room 'VR275P' already exists in the database. Skipping insertion.
Room 'VR174' already exists in the database. Skipping insertion.
Room 'VR191B' already exists in the database. Skipping insertion.
Room 'VR193N' already exists in the database. Skipping insertion.
Room 'VR193I' already exists in the database. Skipping insertion.
Room 'VR175' already exists in the database. Skipping insertion.
Room 'VR193B' already exists in the database. Skipping insertion.
Room 'VR177' already exists in the database. Skipping insertion.
Room 'VR179' already exists in the database. Skipping insertion.
Room 'VR248' already exists in the database. Skipping insertion.
Room 'VR180' already exists in the database. Skipping insertion.
Rooms inserted successfully.
```

Sensor with id 'A81758FFFE0362BA' already exists. Skipping sensor 'A81758FFFE0362BA'.  
Room with id 'VR275A' not found in database. Skipping sensor 'A81758FFFE04857B'.  
Sensor with id 'A81758FFFE048578' already exists. Skipping sensor 'A81758FFFE048578'.  
Sensor with id 'A81758FFFE048579' already exists. Skipping sensor 'A81758FFFE048579'.  
Sensor with id 'A81758FFFE04857A' already exists. Skipping sensor 'A81758FFFE04857A'.  
Sensor with id 'A81758FFFE048577' already exists. Skipping sensor 'A81758FFFE048577'.  
Sensor with id 'A81758FFFE048576' already exists. Skipping sensor 'A81758FFFE048576'.  
Sensor with id 'A81758FFFE0362C1' already exists. Skipping sensor 'A81758FFFE0362C1'.  
Room with id 'VR259' not found in database. Skipping sensor 'A81758FFFE04857C'.  
Sensor with id 'A81758FFFE0362C9' already exists. Skipping sensor 'A81758FFFE0362C9'.  
Sensor with id 'A81758FFFE0362BC' already exists. Skipping sensor 'A81758FFFE0362BC'.  
Sensor with id 'A81758FFFE0362BE' already exists. Skipping sensor 'A81758FFFE0362BE'.  
Sensor with id 'A81758FFFE0362C3' already exists. Skipping sensor 'A81758FFFE0362C3'.  
Sensor with id 'A81758FFFE0362B5' already exists. Skipping sensor 'A81758FFFE0362B5'.  
Room with id 'VR261' not found in database. Skipping sensor 'A81758FFFE04857D'.  
Room with id 'VR262' not found in database. Skipping sensor 'A81758FFFE04857E'.  
Sensor with id '0004A30B001F960A' already exists. Skipping sensor '0004A30B001F960A'.  
Sensor with id 'A81758FFFE0362B4' already exists. Skipping sensor 'A81758FFFE0362B4'.  
Sensor with id '0004A30B00203B74' already exists. Skipping sensor '0004A30B00203B74'.  
Sensor with id '0004A30B001FBF2C' already exists. Skipping sensor '0004A30B001FBF2C'.  
Sensor with id '0004A30B001FFFDA' already exists. Skipping sensor '0004A30B001FFFDA'.  
Sensor with id 'A81758FFFE0362CC' already exists. Skipping sensor 'A81758FFFE0362CC'.  
Sensor with id '0004A30B001E43C9' already exists. Skipping sensor '0004A30B001E43C9'.  
Sensor with id '0004A30B001EB24F' already exists. Skipping sensor '0004A30B001EB24F'.  
Sensor with id '0004A30B001E7ED4' already exists. Skipping sensor '0004A30B001E7ED4'.  
Sensor with id '0004A30B001FE943' already exists. Skipping sensor '0004A30B001FE943'.  
Sensor with id 'A81758FFFE0362B9' already exists. Skipping sensor 'A81758FFFE0362B9'.  
Sensor with id '0004A30B00200058' already exists. Skipping sensor '0004A30B00200058'.  
Sensor with id '0004A30B001F92EE' already exists. Skipping sensor '0004A30B001F92EE'.  
Sensor with id '0004A30B00203A13' already exists. Skipping sensor '0004A30B00203A13'.  
Sensor with id '0004A30B001FE7BD' already exists. Skipping sensor '0004A30B001FE7BD'.  
Sensor with id '0004A30B001FEA11' already exists. Skipping sensor '0004A30B001FEA11'.  
Sensor with id '0004A30B001FC9E3' already exists. Skipping sensor '0004A30B001FC9E3'.  
Sensor with id '0004A30B00204A53' already exists. Skipping sensor '0004A30B00204A53'.  
Sensor with id '0004A30B002031E3' already exists. Skipping sensor '0004A30B002031E3'.  
Sensor with id '0004A30B001E4019' already exists. Skipping sensor '0004A30B001E4019'.  
Sensor with id 'A81758FFFE0362C5' already exists. Skipping sensor 'A81758FFFE0362C5'.  
Sensor with id '0004A30B00201850' already exists. Skipping sensor '0004A30B00201850'.  
Sensor with id 'A81758FFFE0362C2' already exists. Skipping sensor 'A81758FFFE0362C2'.  
Sensor with id '0004A30B001FE21C' already exists. Skipping sensor '0004A30B001FE21C'.  
Sensor with id '0004A30B001FFFC A' already exists. Skipping sensor '0004A30B001FFFC A'.  
Sensor with id '0004A30B00204802' already exists. Skipping sensor '0004A30B00204802'.  
Sensor with id '0004A30B001FBDE4' already exists. Skipping sensor '0004A30B001FBDE4'.  
Sensor with id '0004A30B001E9B95' already exists. Skipping sensor '0004A30B001E9B95'.  
Sensor with id '0004A30B001BA3B3' already exists. Skipping sensor '0004A30B001BA3B3'.  
Sensor with id 'A81758FFFE0362C7' already exists. Skipping sensor 'A81758FFFE0362C7'.  
Sensor with id 'A81758FFFE0362B7' already exists. Skipping sensor 'A81758FFFE0362B7'.  
Sensor with id 'A81758FFFE0362BB' already exists. Skipping sensor 'A81758FFFE0362BB'.  
Sensors inserted successfully.

## 2.9.5 Example Database Rooms-Sensors Rows

 <b>id</b> <small>varchar...</small>	<b>location</b> <small>varchar</small>	<b>capacity</b> <small>int4</small>	<b>description</b> <small>varchar</small>	<b>alt</b> <small>int4</small>	<b>lat</b> <small>numeric</small>	<b>lon</b> <small>numeric</small>
VR170	Vrijhof	7	EMPTY	1	52.24396896362300	6.85338068008423
VR171	Vrijhof	1	EMPTY	1	52.24394226074220	6.85342788696289
VR172	Vrijhof	1	EMPTY	1	52.24393081665040	6.85346078872681
VR173	Vrijhof	1	EMPTY	1	52.24392318725590	6.85348701477051
VR174	Vrijhof	1	EMPTY	1	52.24391174316410	6.85350942611694
VR175	Vrijhof	1	EMPTY	1	52.24390411376950	6.85353183746338
VR176	Vrijhof	1	EMPTY	1	52.24389648437500	6.85355424880981
VR177	Vrijhof	1	EMPTY	1	52.24388885498050	6.85357666015625
VR178	Vrijhof	1	EMPTY	1	52.24387741088870	6.85359907150269
VR179	Vrijhof	1	EMPTY	1	52.24384307861330	6.85353279113770
VR180	Vrijhof	1	EMPTY	1	52.24383544921880	6.85355472564697
VR191B	Vrijhof	7	EMPTY	1	52.24375534057620	6.85365629196167
VR193A	Vrijhof	7	EMPTY	1	52.24394226074220	6.85361433029175
VR193B	Vrijhof	7	EMPTY	1	52.24396514892580	6.85364246368408
VR193D	Vrijhof	7	EMPTY	1	52.24402236938480	6.85370874404907
VR193E	Vrijhof	7	EMPTY	1	52.24395751953120	6.85381126403809
VR193F	Vrijhof	7	EMPTY	1	52.24393844604490	6.85385847091675
VR193G	Vrijhof	7	EMPTY	1	52.24391174316410	6.85400056838989
VR193I	Vrijhof	7	EMPTY	1	52.24385070800780	6.85394763946533
VR193J	Vrijhof	7	EMPTY	1	52.24382400512700	6.85392665863037
VR193K	Vrijhof	7	EMPTY	1	52.24377059936520	6.85387516021729

 id varchar	sensor_type sensortype	 room_id varchar	
0004A30B001BA3B3	NIGHTHAWK	VR179	
0004A30B001E4019	NIGHTHAWK	VR191B	
0004A30B001E43C9	NIGHTHAWK	VR275C	
0004A30B001E7ED4	NIGHTHAWK	VR193E	
0004A30B001E9B95	NIGHTHAWK	VR177	
0004A30B001EB24F	NIGHTHAWK	VR247	
0004A30B001F92EE	NIGHTHAWK	VR275H	
0004A30B001F960A	NIGHTHAWK	VR275K	
0004A30B001FBDE4	NIGHTHAWK	VR275E	
0004A30B001FBF2C	NIGHTHAWK	VR275M	
0004A30B001FC9E3	NIGHTHAWK	VR275B	
0004A30B001FE21C	NIGHTHAWK	VR175	
0004A30B001FE7BD	NIGHTHAWK	VR193J	
0004A30B001FE943	NIGHTHAWK	VR258	
0004A30B001FEA11	NIGHTHAWK	VR193L	
0004A30B001FFFC A	NIGHTHAWK	VR193B	
0004A30B001FFFD A	NIGHTHAWK	VR193G	
0004A30B00200058	NIGHTHAWK	VR256	
0004A30B00201850	NIGHTHAWK	VR275G	
0004A30B002031E3	NIGHTHAWK	VR174	
0004A30B00203A13	NIGHTHAWK	VR173	

## 2.9.6 Structure of *utils/sensors\_historic\_data\_2024\_2025.json*

```
server > utils > {} sensors_historic_data_2024_2025_old.json
26735409      {
26735410          "sensor_id": "a81758fffe048578",
26735411          "sensor_type": "erseye",
26735412          "time": "2025-02-19T04:56:16.217326+00:00",
26735413          "occupancy": 0.0,
26735414          "temperature": 17.2,
26735415          "humidity": 20.0,
26735416          "light": 3.0,
26735417          "motion": 0.0,
26735418          "vdd": 3675.0,
26735419          "geo": {
26735420              "alt": "2",
26735421              "lat": "52.2438468933105",
26735422              "lon": "6.85394906997681"
26735423          }
26735424      },
26735425      {
26735426          "sensor_id": "a81758fffe0362cc",
26735427          "sensor_type": "erseye",
26735428          "time": "2025-02-19T04:56:27.125650+00:00",
26735429          "occupancy": 0.0,
26735430          "temperature": 17.2,
26735431          "humidity": 25.0,
26735432          "light": 2.0,
26735433          "motion": 0.0,
26735434          "vdd": 3547.0,
26735435          "geo": {
26735436              "alt": "2",
26735437              "lat": "52.2434883117676",
26735438              "lon": "6.85303449630737"
26735439          }
26735440      },
26735441      {
26735442          "sensor_id": "a81758fffe04857e",
26735443          "sensor_type": "erseye",
26735444          "time": "2025-02-19T04:56:28.066616+00:00",
26735445          "occupancy": 0.0,
26735446          "temperature": 19.4,
26735447          "humidity": 19.0,
26735448          "light": 1.0,
26735449          "motion": 0.0,
26735450          "vdd": 3678.0,
26735451          "geo": {
```

## 2.9.7 Historic Data Generation - Console Output

```
Checking for raw historic data files...
Raw historic sensor data not detected.
The generation of this data may take several hours.
Starting...
Processing chunk: 2024-09-01T00:00:00Z to 2024-09-07T23:59:59Z
Saving data for 2024-09-01T00:00:00Z - 2024-09-07T23:59:59Z
Started loading nighthawk data...
Started loading erseye data...
Processing nighthawk data...
Processing erseye data...
Processing chunk: 2024-09-08T00:00:00Z to 2024-09-14T23:59:59Z
Saving data for 2024-09-08T00:00:00Z - 2024-09-14T23:59:59Z
Started loading nighthawk data...
Started loading erseye data...
Processing nighthawk data...
Processing erseye data...
Processing chunk: 2024-09-15T00:00:00Z to 2024-09-21T23:59:59Z
Saving data for 2024-09-15T00:00:00Z - 2024-09-21T23:59:59Z
Started loading nighthawk data...
Started loading erseye data...
Processing nighthawk data...
Processing erseye data...
Processing chunk: 2024-09-22T00:00:00Z to 2024-09-28T23:59:59Z
Saving data for 2024-09-22T00:00:00Z - 2024-09-28T23:59:59Z
Started loading nighthawk data...
Started loading erseye data...
Processing nighthawk data...
Processing erseye data...
Processing chunk: 2024-09-29T00:00:00Z to 2024-10-05T23:59:59Z
Saving data for 2024-09-29T00:00:00Z - 2024-10-05T23:59:59Z
Started loading nighthawk data...
```

## 2.9.8 Structure of *utils/historic\_room\_occupancy.json*

```
server > utils > {} historic_room_occupancy.json > {} 2025-02-11T14:00:00+00:00 > {} vr275j
1  {
2    "2025-02-11T14:00:00+00:00": {
3      "vr275g": {
4        "occupied": 1,
5        "occ_ratio": 0.3076923076923077,
6        "avg_temperature": 15.141153846153847,
7        "avg_humidity": 28.923076923076923,
8        "avg_light": 57.472692307692306,
9        "capacity": "large"
10     },
11     "vr275e": {
12       "occupied": 1,
13       "occ_ratio": 0.42857142857142855,
14       "avg_temperature": 13.350571428571428,
15       "avg_humidity": 26.571428571428573,
16       "avg_light": 90.76228571428571,
17       "capacity": "large"
18     },
19     "vr275h": {
20       "occupied": 0,
21       "occ_ratio": 0.2,
22       "avg_temperature": 14.677000000000001,
23       "avg_humidity": 31.8,
24       "avg_light": 10.407,
25       "capacity": "large"
26     },
27     "vr193g": {
28       "occupied": 0,
29       "occ_ratio": 0.0,
30       "avg_temperature": 14.8575,
31       "avg_humidity": 28.0,
32       "avg_light": 4.3175,
33       "capacity": "large"
34     },
35     "vr256": {
36       "occupied": 1,
37       "occ_ratio": 0.625,
38       "avg_temperature": 15.77125,
39       "avg_humidity": 27.75,
40       "avg_light": 28.663750000000004,
41       "capacity": "medium"
42     },
43     "vr275a": {
```

### 2.9.9 Structure of *utils/room\_size\_historic\_occupancy.json*

```
server > utils > {} room_size_historic_occupancy.json > {} 2025-02-11T19:00:00+00:00

14114   "2025-04-15T13:00:00+00:00": {
14115     "small": {
14116       "occupied": 10,
14117       "total": 10
14118     },
14119     "medium": {
14120       "occupied": 4,
14121       "total": 5
14122     },
14123     "large": {
14124       "occupied": 26,
14125       "total": 27
14126     }
14127   },
14128   "2025-04-15T14:00:00+00:00": {
14129     "small": {
14130       "occupied": 10,
14131       "total": 10
14132     },
14133     "medium": {
14134       "occupied": 4,
14135       "total": 5
14136     },
14137     "large": {
14138       "occupied": 25,
14139       "total": 27
14140     }
14141   },
14142   "2025-04-15T15:00:00+00:00": {
14143     "small": {
14144       "occupied": 10,
14145       "total": 10
14146     },
14147     "medium": {
14148       "occupied": 4,
14149       "total": 5
14150     },
14151     "large": {
14152       "occupied": 22,
14153       "total": 27
14154     }
14155   },
14156   "2025-04-15T16:00:00+00:00": {
```



### 2.9.10 Structure of *utils/occupancy\_frequency\_rooms.json*

```
server > utils > {} occupancy_frequency_rooms.json > ...
You, 3 days ago | 1 author (You)
1  [
2    {
3      "name": "vr275g",
4      "value": 887
5    },
6    {
7      "name": "vr275e",
8      "value": 781
9    },
10   {
11     "name": "vr275h",
12     "value": 723
13   },
14   {
15     "name": "vr256",
16     "value": 639
17   },
18   {
19     "name": "vr193g",
20     "value": 635
21   },
22   {
23     "name": "vr258",
24     "value": 578
25   },
26   {
27     "name": "vr275c",
28     "value": 666
29   },
30   {
31     "name": "vr275m",
32     "value": 649
33   },
34   {
35     "name": "vr191b",
36     "value": 631
37   },
38   {
39     "name": "vr176",
40     "value": 631
41   },
42   {
43     "name": "vr179",
```

### 2.9.11 Intermediate Data Generation - Console Output

```
Checking for raw historic data files...
Raw historic sensor data detected, filepath=./utils/sensors_historic_data_2024_2025.json

generate_rooms_sensors_mappings
Results saved to ./utils/rooms_sensors_mappings.json

generate_sensors_rooms
Results saved to ./utils/sensors_rooms.json

generate_historic_room_occupancy_by_hour
Results saved to ./utils/historic_room_occupancy.json

generate_room_size_occupancy_count_by_hours
Results saved to ./utils/room_size_historic_occupancy.json

generate_occupancy_frequencies_by_room
Results saved to ./utils/occupancy_frequency_rooms.json

Finished generating dependencies.

Inserting data into database...
Room 'VR275G' already exists in the database. Skipping insertion.
```

### 2.9.12 Occupancy API Testing URLs

```
# http://127.0.0.1:5000/api/discrepancies/
# http://127.0.0.1:5000/api/discrepancies/?floor=2&sensorType=nighthawk
# http://127.0.0.1:5000/api/discrepancies/?room=VR170&capacity=3
# http://127.0.0.1:5000/api/discrepancies/?startDate=2025-03-01&endDate=2025-03-05
# http://127.0.0.1:5000/api/discrepancies/?bookingDuration=1-2 # returns {}
# http://127.0.0.1:5000/api/discrepancies/?week=2025-W10&sensorType=erseye

@occupancy_api.route("/discrepancies/", methods=['GET'])
```

### 2.9.13 WebSockets Testing - Client Output

```
○ Connected to server!
○ Sending getSensorList request with: ?capacity=1
  Received response for getSensorList: 12
  Sending getRoomList request with: ?capacity=1
  Received response for getRoomList: 10
  Sending getRoomsOccupancyMap request with: ?capacity=1&sensorType=nighthawk
  Received response for getRoomsOccupancyMap: 6
```

## 2.10 MT-03

### 2.10.1 Database Insertion

- Before insertion: 0 existing bookings and 17070 inserted bookings
- After insertion: 17070 existing bookings and 0 inserted bookings

```
Skipped 0 existing bookings.
Inserted 17070 records successfully.
    New start date: 20250412
127.0.0.1 - - [12/Apr/2025 19:23:10] "GET /api/bookings/insert/fetch/ HTTP/1.1" 200 -
Skipped 17070 existing bookings.
Inserted 0 records successfully.
    New start date: 20250412
```

### 2.10.2 Reading the Parquet file before the first API request

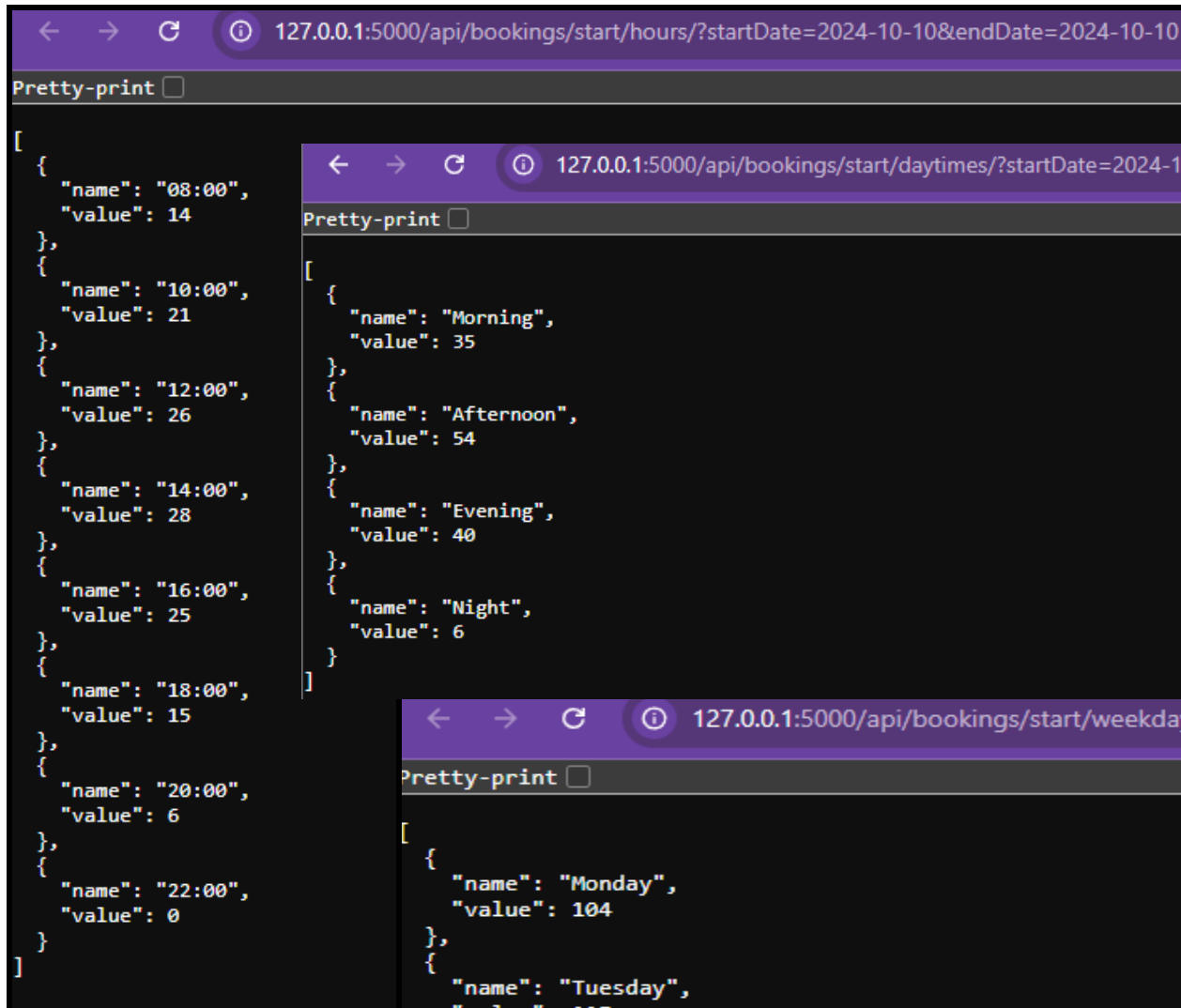
```
Real-time sensor data polling disabled!
To enable, set ENABLE_POLLING=True in /server/.env
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
Real-time sensor data polling disabled!
To enable, set ENABLE_POLLING=True in /server/.env
* Debugger is active!
* Debugger PIN: 130-318-255
Loaded 17070 rows from Parquet file.
127.0.0.1 - - [17/Apr/2025 23:46:16] "GET /api/bookings/start/hours/?sensorType=&startT
me=&week=&bookingDuration=&room=&capacity&floor= HTTP/1.1" 200 -
```

### 2.10.3 Bookings API Testing Links

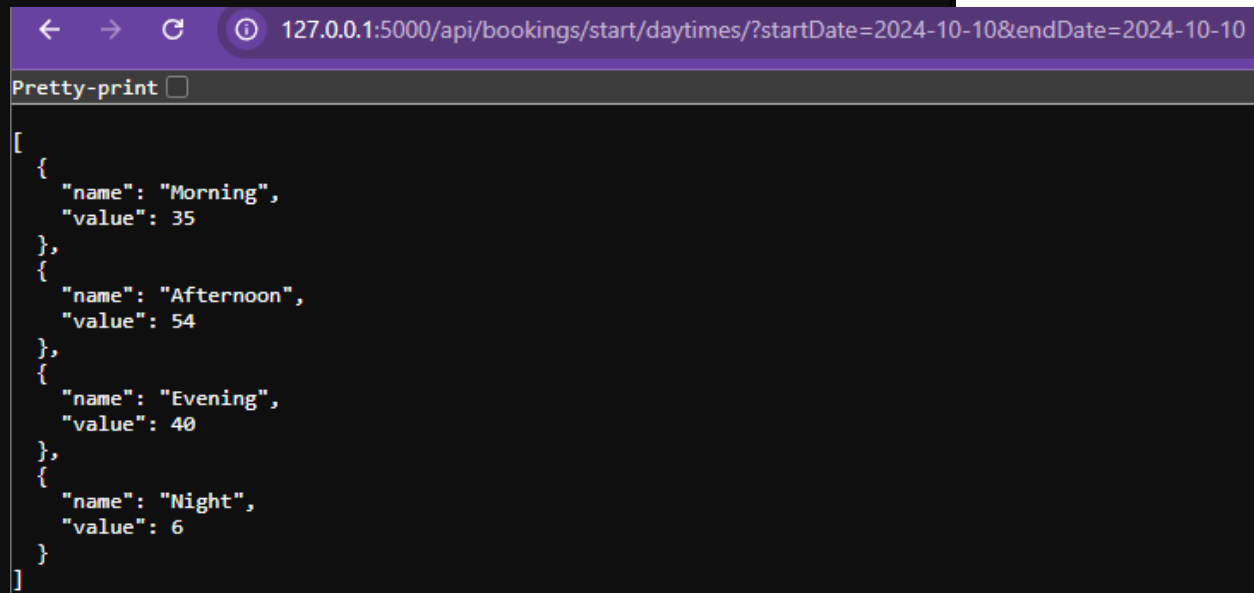
```
# http://127.0.0.1:5000/api/bookings/start/hours/?startDate=2024-10-10&endDate=2024-10-10
# http://127.0.0.1:5000/api/bookings/start/hours/?sensorType=&startDate=&endDate=&startTime=&endTime=&week=&bookingDur
# http://127.0.0.1:5000/api/bookings/start/hours/?sensorType=&startDate=2024-10-10&endDate=2024-10-10&startTime=&endTime=
# http://127.0.0.1:5000/api/bookings/start/hours/?capacity=2-3&sensorType=&startTime=&endTime=&week=&bookingDuration=&
# http://127.0.0.1:5000/api/bookings/start/hours/?capacity=>3&sensorType=&startTime=&endTime=&week=&bookingDuration=&
# http://127.0.0.1:5000/api/bookings/start/hours/?capacity=1&sensorType=&startTime=&endTime=&week=&bookingDuration=&
# http://127.0.0.1:5000/api/bookings/start/hours/?capacity=1&sensorType=nighthawk&startTime=&endTime=&week=&bookingDur
# http://127.0.0.1:5000/api/bookings/start/hours/?capacity=1&sensorType=erseye&startTime=&endTime=&week=&bookingDurati
# Triggers EmptyFilteredData exception:
# http://127.0.0.1:5000/api/bookings/start/hours/?startDate=2024-10-10&endDate=2024-10-10&floor=1&room=275B
```

## 2.10.4 Checking identity between APIs with overlapping results

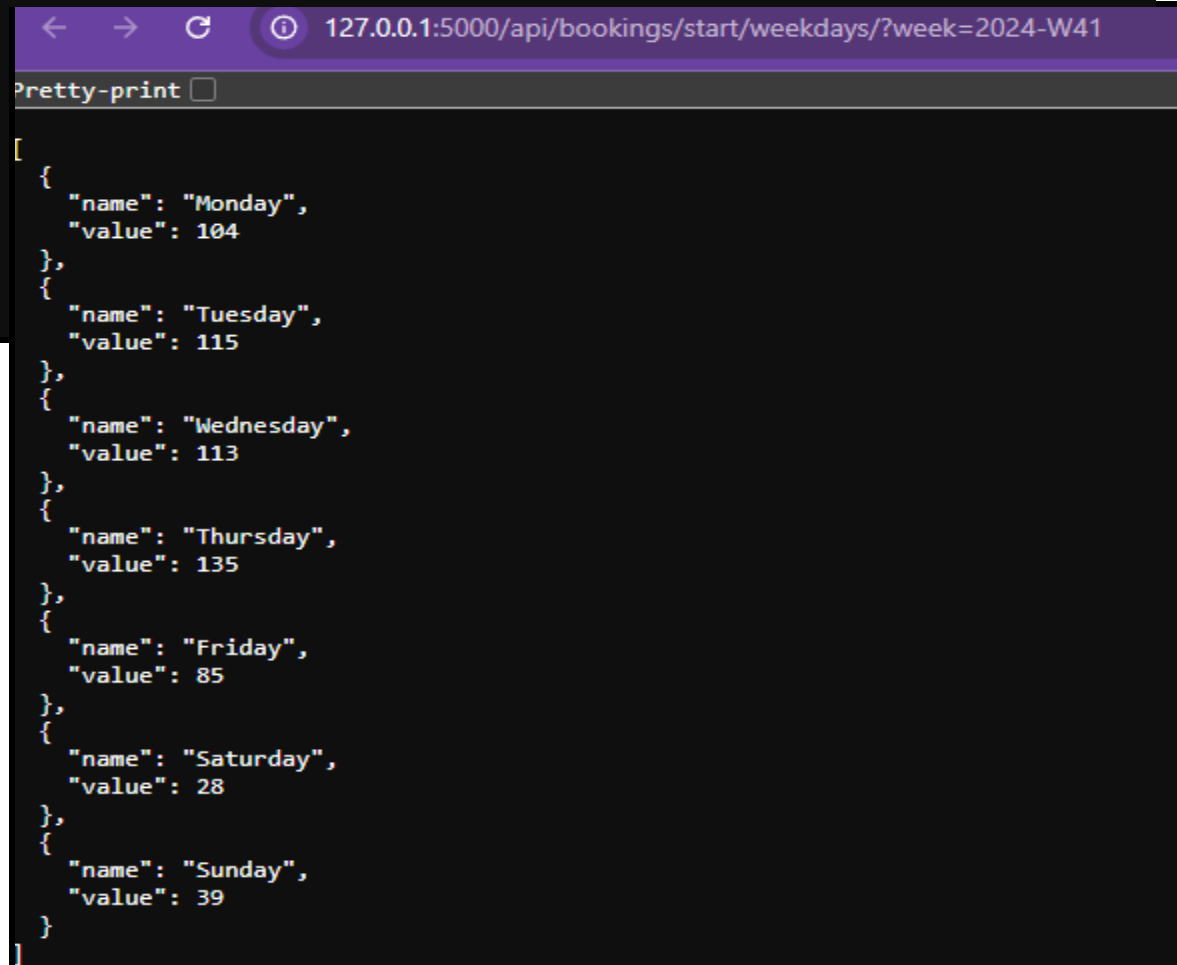
2024-10-10 is the Thursday of week 2024-W41.



```
[
  {
    "name": "08:00",
    "value": 14
  },
  {
    "name": "10:00",
    "value": 21
  },
  {
    "name": "12:00",
    "value": 26
  },
  {
    "name": "14:00",
    "value": 28
  },
  {
    "name": "16:00",
    "value": 25
  },
  {
    "name": "18:00",
    "value": 15
  },
  {
    "name": "20:00",
    "value": 6
  },
  {
    "name": "22:00",
    "value": 0
  }
]
```



```
[
  {
    "name": "Morning",
    "value": 35
  },
  {
    "name": "Afternoon",
    "value": 54
  },
  {
    "name": "Evening",
    "value": 40
  },
  {
    "name": "Night",
    "value": 6
  }
]
```



```
[
  {
    "name": "Monday",
    "value": 104
  },
  {
    "name": "Tuesday",
    "value": 115
  },
  {
    "name": "Wednesday",
    "value": 113
  },
  {
    "name": "Thursday",
    "value": 135
  },
  {
    "name": "Friday",
    "value": 85
  },
  {
    "name": "Saturday",
    "value": 28
  },
  {
    "name": "Sunday",
    "value": 39
  }
]
```

## 2.10.5 Data Structuring

```
Index(['Building', 'Hall', 'Capacity', 'People Count', 'Created', 'Modified',
      'Cancelled', 'Booking ID', 'Capacity Label', 'Begin datetime',
      'End datetime', 'Is Modified', 'Is Cancelled', 'Room', 'Sensors',
      'Duration', 'Duration Label', 'Discrepancy', 'Lead Time', 'Week',
      'User ID'],
      dtype='object')
Timestamp columns:
Index(['Created', 'Modified', 'Cancelled', 'Begin datetime', 'End datetime'], dtype='object')
Count of Cancelled columns:
Is Cancelled
False    13497
True      2282
Name: count, dtype: int64
User ID
1    15779
Name: count, dtype: int64
Capacity Label
>3    11923
2-3    2721
1      1135
Name: count, dtype: int64
```

	Building	Hall	Capacity	People Count	...	Discrepancy	Lead Time	Week	User ID
0	Vrijhof	Library	1	1	...	Equal	<1h	2024-W36	1
1	Vrijhof	Library	7	5	...	Smaller	<1h	2024-W36	1
2	Vrijhof	Library	1	1	...	Equal	<1h	2024-W36	1
3	Vrijhof	Library	2	1	...	Smaller	<1h	2024-W36	1
4	Vrijhof	Library	3	2	...	Smaller	<1h	2024-W36	1

```
[5 rows x 21 columns]
```

## 2.10.6 Data Grouping

Filters:

startTime: 08:00, endTime: 23:59, startDate: 2024-10-10, endDate: 2024-10-10

	Room	Sum
--	------	-----

0	VR171	1
---	-------	---

1	VR172	2
---	-------	---

2	VR173	2
---	-------	---

3	VR178	1
---	-------	---

4	VR191B	1
---	--------	---

5	VR193J	7
---	--------	---

6	VR193K	6
---	--------	---

7	VR193L	5
---	--------	---

8	VR193N	4
---	--------	---

9	VR247	4
---	-------	---

10	VR248	5
----	-------	---

11	VR256	5
----	-------	---

12	VR257	7
----	-------	---

13	VR258	7
----	-------	---

14	VR275B	6
----	--------	---

15	VR275C	6
----	--------	---

16	VR275E	5
----	--------	---

17	VR275F	5
----	--------	---

18	VR275G	5
----	--------	---

19	VR275H	8
----	--------	---

20	VR275J	4
----	--------	---

21	VR275K	6
----	--------	---

22	VR275L	5
----	--------	---

23	VR275M	4
----	--------	---

24	VR275O	8
----	--------	---

25	VR275P	5
----	--------	---

26	VR275Q	3
----	--------	---

27	VR275R	5
----	--------	---

28	VR275T	3
----	--------	---

## 2.10.7 Data Filtering

```
---- Structured Data ----
0    2024-09-02 09:00:00
1    2024-09-02 09:00:00
2    2024-09-02 09:15:00
3    2024-09-02 10:00:00
4    2024-09-02 10:00:00
Name: Begin datetime, dtype: datetime64[ns]
15774 2025-03-31 17:00:00
15775 2025-03-31 17:00:00
15776 2025-03-31 17:30:00
15777 2025-04-01 14:30:00
15778 2025-04-01 18:30:00
Name: End datetime, dtype: datetime64[ns]

Applying filters:
startTime: 08:00, endTime: 23:59, startDate: 2024-10-01, endDate: 2024-10-10

---- Filtered Data ----
2997 2024-10-01 09:00:00
2998 2024-10-01 09:00:00
2999 2024-10-01 09:15:00
3000 2024-10-01 09:30:00
3001 2024-10-01 09:30:00
Name: Begin datetime, dtype: datetime64[ns]
3973 2024-10-10 23:00:00
3974 2024-10-10 23:00:00
3975 2024-10-10 23:00:00
3976 2024-10-10 23:00:00
3977 2024-10-10 22:45:00
Name: End datetime, dtype: datetime64[ns]
```

## 2.10.8 Data Analysis

```
Loaded 17070 rows from Parquet file.
##### DIAGRAM 1 #####
Hours
[{'name': '08:00', 'value': 93}, {'name': '10:00', 'value': 149}, {'name': '12:00', 'value': 219}, {'name': '14:00', 'value': 183}, {'name': '16:00', 'value': 186}, {'name': '18:00', 'value': 116}, {'name': '20:00', 'value': 35}, {'name': '22:00', 'value': 0}]
Daytimes
[{'name': 'Morning', 'value': 242}, {'name': 'Afternoon', 'value': 402}, {'name': 'Evening', 'value': 302}, {'name': 'Night', 'value': 35}]
Weekdays
[{'name': 'Monday', 'value': 111}, {'name': 'Tuesday', 'value': 106}, {'name': 'Wednesday', 'value': 125}, {'name': 'Thursday', 'value': 127}, {'name': 'Friday', 'value': 106}, {'name': 'Saturday', 'value': 19}, {'name': 'Sunday', 'value': 31}]
Days
[{'name': '2024-W40', 'value': 514}, {'name': '2024-W41', 'value': 467}]
Months
[{'name': 'October 2024', 'value': 981}]

##### DIAGRAM 2 #####
Opening hours
[{'name': '08:00', 'value': 69}, {'name': '10:00', 'value': 145}, {'name': '12:00', 'value': 147}, {'name': '14:00', 'value': 167}, {'name': '16:00', 'value': 154}, {'name': '18:00', 'value': 93}, {'name': '20:00', 'value': 75}, {'name': '22:00', 'value': 39}]
Day Times
[{'name': 'Morning', 'value': 223}, {'name': 'Afternoon', 'value': 314}, {'name': 'Evening', 'value': 247}, {'name': 'Night', 'value': 142}]

##### DIAGRAM 4 #####
Bar Chart
[{'name': 'Small (1 person)', 'value': 6}, {'name': 'Medium (2-3 people)', 'value': 68}, {'name': 'Large (>3 people)', 'value': 640}]

##### DIAGRAM 5 #####
[{'name': 'Cancelled', 'value': 117}, {'name': 'Not Cancelled', 'value': 864}]

##### DIAGRAM 6 #####
[{'name': 'Small (1 person)', 'value': 45}, {'name': 'Medium (2-3 people)', 'value': 181}, {'name': 'Large (>3 people)', 'value': 755}]

##### DIAGRAM 7 #####
[{'name': '<1h', 'value': 21}, {'name': '1-2h', 'value': 634}, {'name': '2-3h', 'value': 129}, {'name': '3-4h', 'value': 180}, {'name': '>4h', 'value': 17}]

##### DIAGRAM 9 #####
[{'name': '<1h', 'value': 966}, {'name': '1-4h', 'value': 0}, {'name': '4-8h', 'value': 0}, {'name': '8-16h', 'value': 4}, {'name': '16-48h', 'value': 6}, {'name': '>48h', 'value': 5}]
```



2.11 ST-01

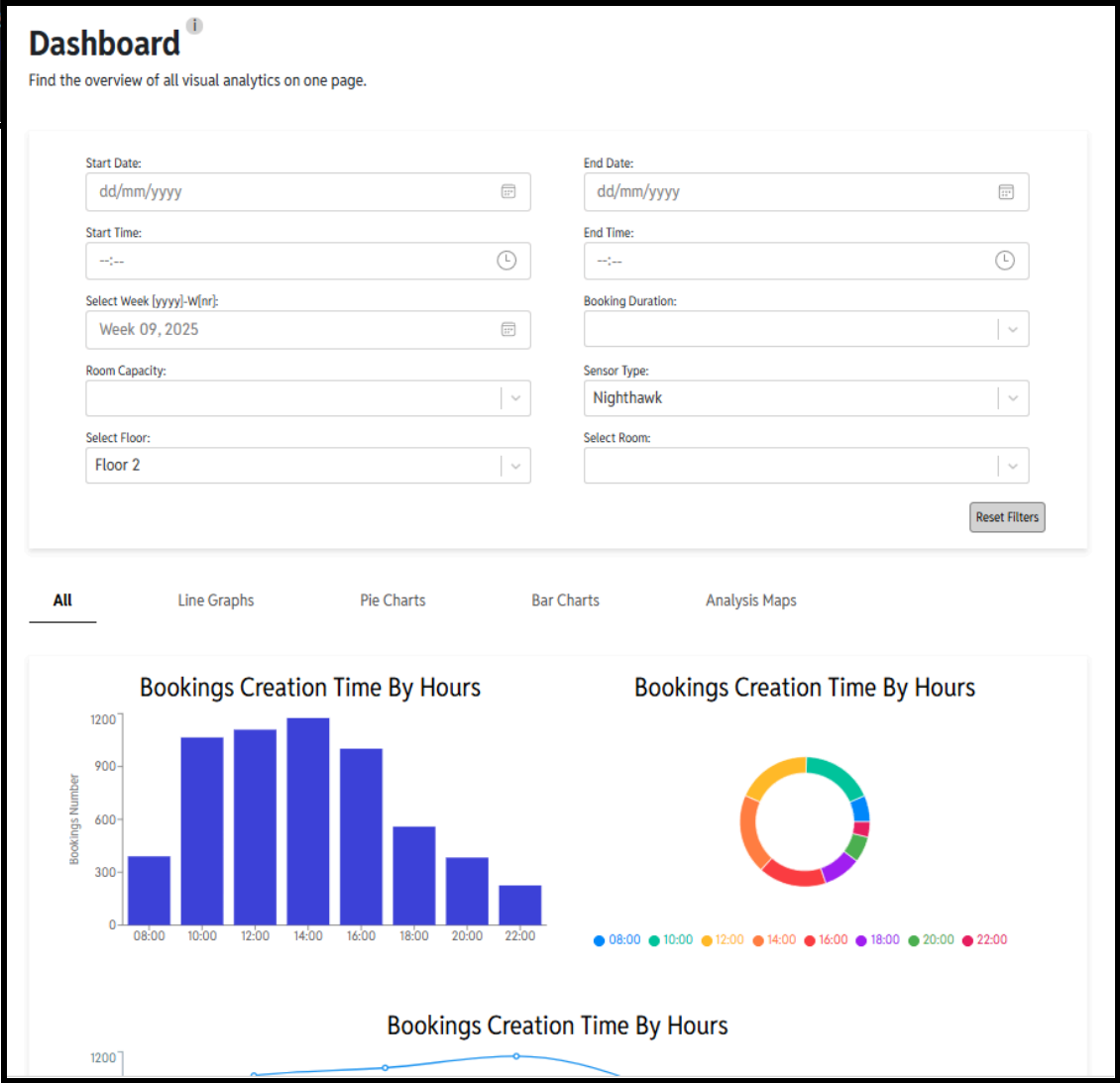
```
PS C:\My-Files\git\lisa\lisa-library> cd server
● PS C:\My-Files\git\lisa\lisa-library\server> .\.venv\Scripts\activate
○ (.venv) PS C:\My-Files\git\lisa\lisa-library\server> py run.py
Started polling background task.
  * Restarting with stat
Started polling background task.
  * Debugger is active!
  * Debugger PIN: 130-440-469
Polled sensor data len: 47
```

```
PS C:\My-Files\git\lisa\lisa-library> cd client
PS C:\My-Files\git\lisa\lisa-library\client> npm run dev

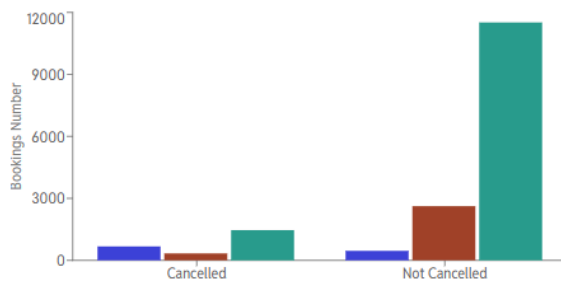
> webapp@0.0.0 dev
> vite

VITE v6.2.5 ready in 2091 ms

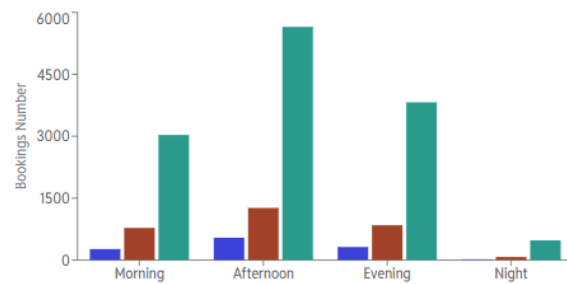
→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```



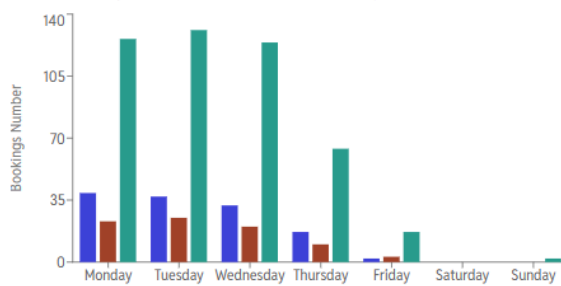
### Booking Cancellations Comparison



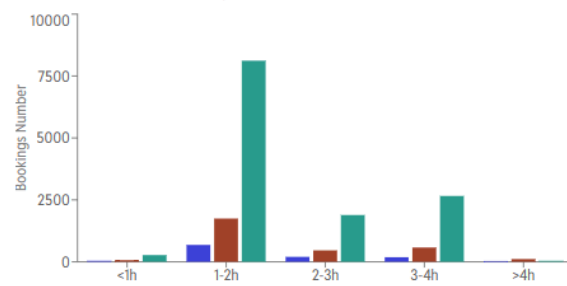
### Booking Start-Time (DayTime) Comparison



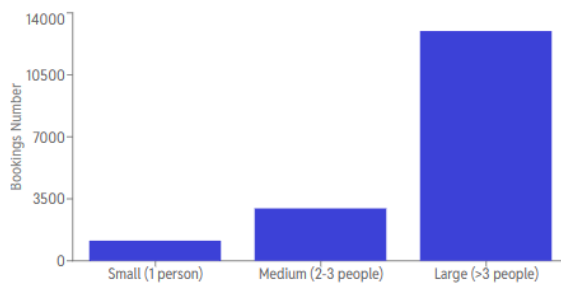
### Booking Start-Time (WeekDays) Comparison



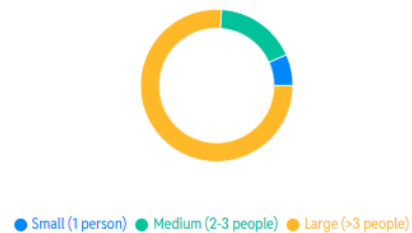
### Bookings Time Comparison



### Room Preference by Bookings



### Room Preference by Bookings

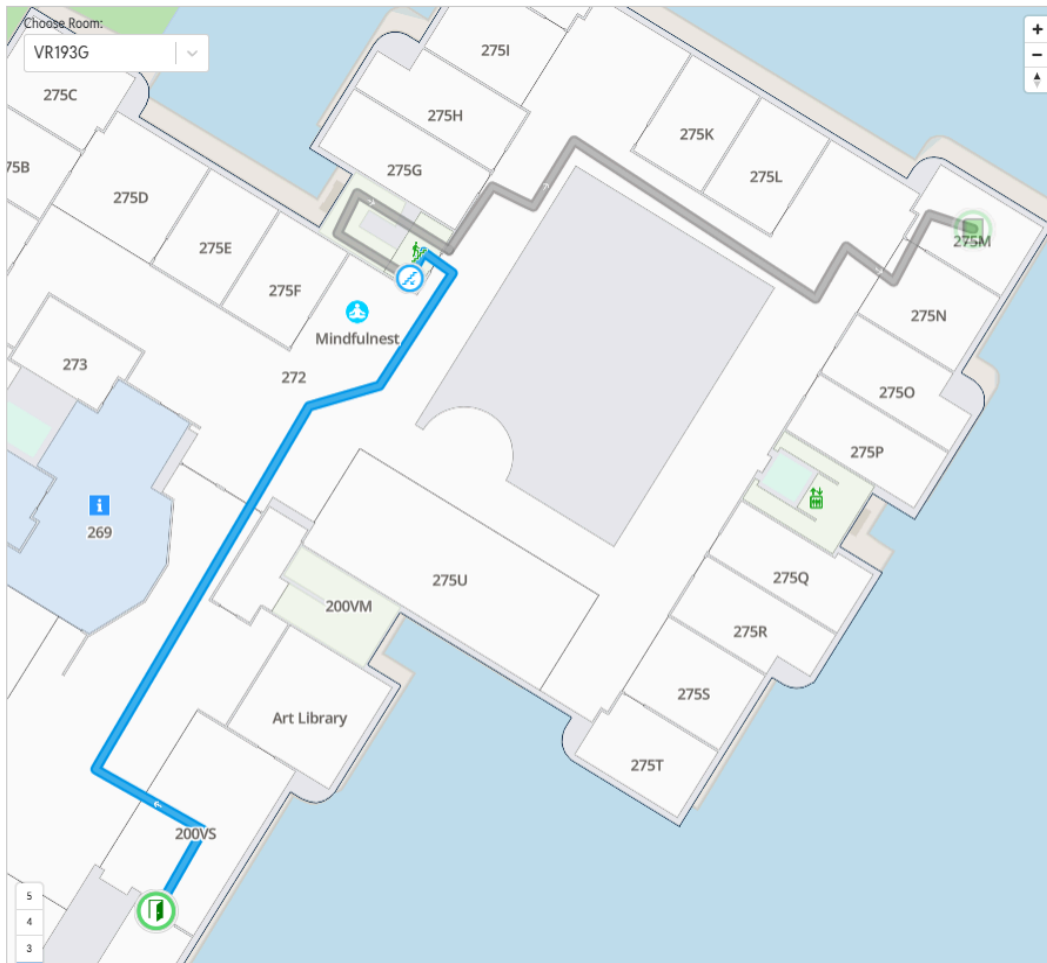




## Navigation

Navigate easily through the

Select the room you want to navigate to. The start point is the entrance of the Vrijhof library 2nd floor. To switch the floors on the map route, click on the "Stairs" icon.



## 2.12 Old Database Design



## 2.13 PT-01 Web Page System Load and Scalability

```
Checking if server is reachable at http://localhost:5173/...  
Server is up and responding.
```

```
Starting simulation with 10 users...
```

```
[User 1] Interaction completed in 2.47 seconds.  
[User 2] Interaction completed in 2.81 seconds.  
[User 3] Interaction completed in 2.81 seconds.  
[User 4] Interaction completed in 2.77 seconds.  
[User 5] Interaction completed in 2.67 seconds.  
[User 6] Interaction completed in 2.54 seconds.  
[User 7] Interaction completed in 2.68 seconds.  
[User 8] Interaction completed in 2.54 seconds.  
[User 9] Interaction completed in 2.43 seconds.  
[User 10] Interaction completed in 2.77 seconds.
```

```
Average time spent: 2.65 seconds
```

```
All users have completed their session.
```

```
Checking if server is reachable at http://localhost:5173/...  
Server is up and responding.
```

```
Starting simulation with 20 users...
```

```
[User 1] Interaction completed in 2.36 seconds.  
[User 2] Interaction completed in 2.46 seconds.  
[User 3] Interaction completed in 2.64 seconds.  
[User 4] Interaction completed in 3.23 seconds.  
[User 5] Interaction completed in 2.57 seconds.  
[User 6] Interaction completed in 2.56 seconds.  
[User 7] Interaction completed in 2.66 seconds.  
[User 8] Interaction completed in 2.70 seconds.  
[User 9] Interaction completed in 2.47 seconds.  
[User 10] Interaction completed in 2.59 seconds.  
[User 11] Interaction completed in 2.64 seconds.  
[User 12] Interaction completed in 2.46 seconds.  
[User 13] Interaction completed in 2.50 seconds.  
[User 14] Interaction completed in 2.54 seconds.  
[User 15] Interaction completed in 2.56 seconds.  
[User 16] Interaction completed in 2.49 seconds.  
[User 17] Interaction completed in 2.51 seconds.  
[User 18] Interaction completed in 2.54 seconds.  
[User 19] Interaction completed in 2.46 seconds.  
[User 20] Interaction completed in 2.43 seconds.
```

```
Average time spent: 2.57 seconds
```

```
Checking if server is reachable at http://localhost:5173/...
Server is up and responding.
```

```
Starting simulation with 30 users...
```

```
[User 1] Interaction completed in 2.39 seconds.
[User 2] Interaction completed in 2.59 seconds.
[User 3] Interaction completed in 2.70 seconds.
[User 4] Interaction completed in 2.38 seconds.
[User 5] Interaction completed in 2.55 seconds.
[User 6] Interaction completed in 2.41 seconds.
[User 7] Interaction completed in 2.53 seconds.
[User 8] Interaction completed in 2.64 seconds.
[User 9] Interaction completed in 2.63 seconds.
[User 10] Interaction completed in 2.70 seconds.
[User 11] Interaction completed in 2.56 seconds.
[User 12] Interaction completed in 2.53 seconds.
[User 13] Interaction completed in 2.54 seconds.
[User 14] Interaction completed in 2.54 seconds.
[User 15] Interaction completed in 2.61 seconds.
[User 16] Interaction completed in 2.48 seconds.
[User 17] Interaction completed in 2.60 seconds.
[User 18] Interaction completed in 2.52 seconds.
[User 19] Interaction completed in 2.55 seconds.
[User 20] Interaction completed in 2.44 seconds.
[User 21] Interaction completed in 2.53 seconds.
[User 22] Interaction completed in 2.44 seconds.
[User 23] Interaction completed in 2.63 seconds.
[User 24] Interaction completed in 2.52 seconds.
[User 25] Interaction completed in 2.54 seconds.
[User 26] Interaction completed in 2.56 seconds.
[User 27] Interaction completed in 2.56 seconds.
[User 28] Interaction completed in 2.42 seconds.
[User 29] Interaction completed in 2.49 seconds.
[User 30] Interaction completed in 2.42 seconds.
```

```
Average time spent: 2.53 seconds
```

```
Checking if server is reachable at http://localhost:5173/...
Server is up and responding.
```

```
Starting simulation with 50 users...
```

```
Average time spent: 2.53 seconds
```

```
All users have completed their session.
```

## 2.14 PT-02 Server API System Load and Scalability

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/api/booking-frequencies/colorcodes/	10	0	25000	50000	50000	25949.58	798	49548	976	0	0
GET	/api/bookings/created/daytimes/	10	0	57000	58000	58000	52254.5	11224	57525	203	0	0
GET	/api/bookings/created/hours/	10	0	47000	64000	64000	48893.87	37106	63904	385	0	0
GET	/api/bookings/start/daytimes/	11	0	61000	61000	61000	50975.03	6379	61153	202	0	0
GET	/api/bookings/start/hours/	16	0	25000	63000	63000	30128.13	6165	63433	384	0.1	0
GET	/api/bookings/start/months/	10	0	34000	61000	61000	36830.04	33445	61162	444	0	0
GET	/api/bookings/start/weekdays/	10	0	29000	59000	59000	32124.3	2408	59260	347	0	0
GET	/api/bookings/start/weeks/	10	0	23000	39000	39000	24034.89	3758	39274	1599	0	0
GET	/api/occupancy-frequencies/	10	0	1600	58000	58000	7216.39	365	57726	186	0	0
GET	/api/occupancy-frequencies/colorcodes/	10	0	350	35000	35000	3773.29	56	34629	1042	0	0
GET	/api/rooms/info/	20	0	5300	12000	12000	6281.13	705	12421	7612	0	0
GET	/api/rooms/real-time/	10	0	1400	56000	56000	6794.4	98	55642	3	0	0
GET	/api/rooms/table/	10	0	11	710	710	81.09	10	711	3	0	0
	Aggregated	147	0	19000	61000	63000	24135.52	10	63904	1445.48	0.1	0



### 3. Meeting Notes and Project Planning

#### 3.1 Week 1 Meeting - Key Questions & Notes

Question	Client Response/Notes
Q1: Current system implementation?	The 1st project focused on people counting. Discussed limitations of prior solutions.
Q2: Preferred requirements format (e.g., MoSCoW)?	Draft requirements discussed; focus on data visualization and derived insights.
Q3: Final assessment priorities?	Informal proof of concept preferred. Presentation + slides required.
Q4: Data visualization preferences?	Combination of real-time and historical data analysis needed.
Q5: TimeEdit's capabilities for data analysis?	No API; JSON data available for processing.
Q6: Technical preferences (frameworks, languages)?	Independent website preferred (not embedded in TimeEdit).
Q7: Testing strategies?	Limited to library sensors; motion/heat sensors available.
Q8: Key stakeholders?	Staff (historical data) vs. students (real-time insights).
Q9: Ethical considerations?	Focus on anonymized data; no personal identifiers.
Q10: Meeting schedule?	Weekly in-person meetings confirmed.

### 3.2 Week 4 Meeting - Key Notes

#### Sensors & Data

- Older data retrieved first (filtering required; max 4096 records).
- Motion sensor false positives observed under low light.

#### Design Feedback

- Prototype 3 (violet theme) selected for intuitiveness and depth.
- Graphs consolidated into a single dashboard for clarity.
- Descriptions + info icons preferred over AI-generated text.

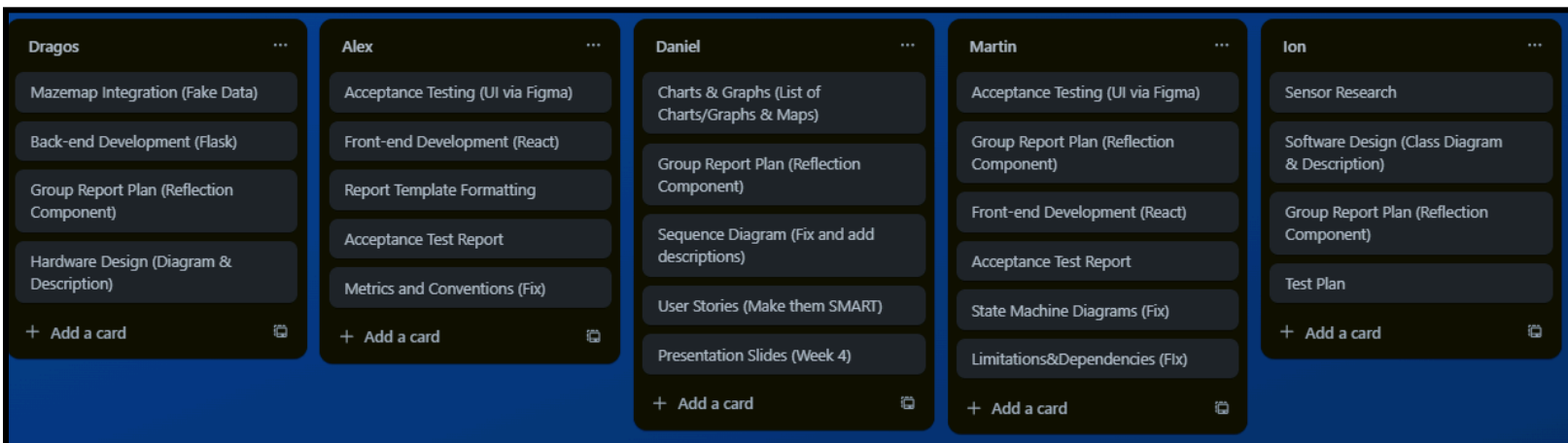
#### Data Analysis Priorities

- For students: Unified, trustworthy results (combined sensor data).
- For staff: Detailed comparisons with confidence metrics.
- Investigate discrepancies between sensor readings (e.g., "no movement" vs. light status).

#### Additional Notes

- The Python project shared by the client contained errors; required debugging.
- TimeEdit data accessible via JSON (no direct API).

### 3.3 Trello Week 4 - Task Allocation



### 3.4 Trello - General Planning

**Week 1**

- ✓ Group Formation
- ✓ Project Selection
- ✓ Requirements Discussion
- ✓ Supervisor Selection
- + Добавить карточку

**Week 2**

- ✓ Non-functional Requirements
- ✓ Functional Requirements
- ✓ UI Draft (Figma)
- ✓ User-Stories
- + Добавить карточку

**Week 3**

- ✓ Testing Strategy
- ✓ Stakeholders (Roles, Responsibilities, Concerns)
- ✓ Use Cases (Description)
- ✓ Week 2 Feedback Implementation
- ✓ Testing Schedule
- ✓ Metrics and Conventions
- ✓ State Machine Diagrams
- ✓ Database Design (Class Diagram & Description)
- ✓ System Description (Limitations, Constraints, Dependencies, Risks)
- ✓ Sequence Diagrams
- + Добавить карточку

**Week 4**

- ✓ Acceptance Testing (UI via Figma)
- ✓ Acceptance Testing (Report)
- ✓ Front-end Development (React)
- ✓ Week 3 Feedback Implementation
- ✓ Group Report Plan (Reflection)
- ✓ Charts, Graphs & Maps (List)
- ✓ Mazemap Integration (Fake Data)
- ✓ Back-end Development (Flask)
- ✓ Hardware Design (Diagram & Description)
- ✓ Software Design (Class Diagram & Description)
- + Добавить карточку

**Week 5**

- ✓ Charts & Graphs Integration (Fake Data)
- ✓ TimeEdit Integration (Fake Data)
- ✓ Sensor Integration (Data Retrieval)
- ✓ Front-end Development (React)
- ✓ Mazemap Integration (Fake Data)
- ✓ Back-end Development (Flask)
- ✓ Week 4 Feedback Implementation
- ✓ Initialize DB
- ✓ Test Plan
- + Добавить карточку

**Week 6**

- ✓ Reflection Report
- ✓ Week 5 Feedback Implementation
- ✓ Front-end Development (React)
- ✓ Sensor Integration (Data Analysis)
- ✓ TimeEdit Integration (Data Retrieval)
- ✓ Charts & Graphs Integration (Fake Data)
- ✓ Back-end Development (Flask)
- ✓ Mazemap Integration (Real Data)
- + Добавить карточку

**Week 7**

- ✓ Authentication - Google OAuth
- ✓ Week 6 Feedback Implementation
- ✓ Front-end Development (React)
- ✓ Back-end Development (Flask)
- ✓ TimeEdit Integration (Data Analysis)
- ✓ Charts & Graphs Integration (Real Data)
- ✓ Sensor Integration (Data Analysis)
- ✓ Mazemap Integration (Real Data)
- + Добавить карточку

**Week 8**

- ✓ Charts & Graphs (Real Data)
- ✓ TimeEdit Integration (Additional Requirements - COULD DO)
- ✓ Mazemap Integration (Additional Requirements - COULD DO)
- ✓ Week 7 Feedback Implementation
- API List (endpoint name, filters to be accepted, description, response example, unit test, isWebSocket/real-time)
- + Добавить карточку

**Week 9**

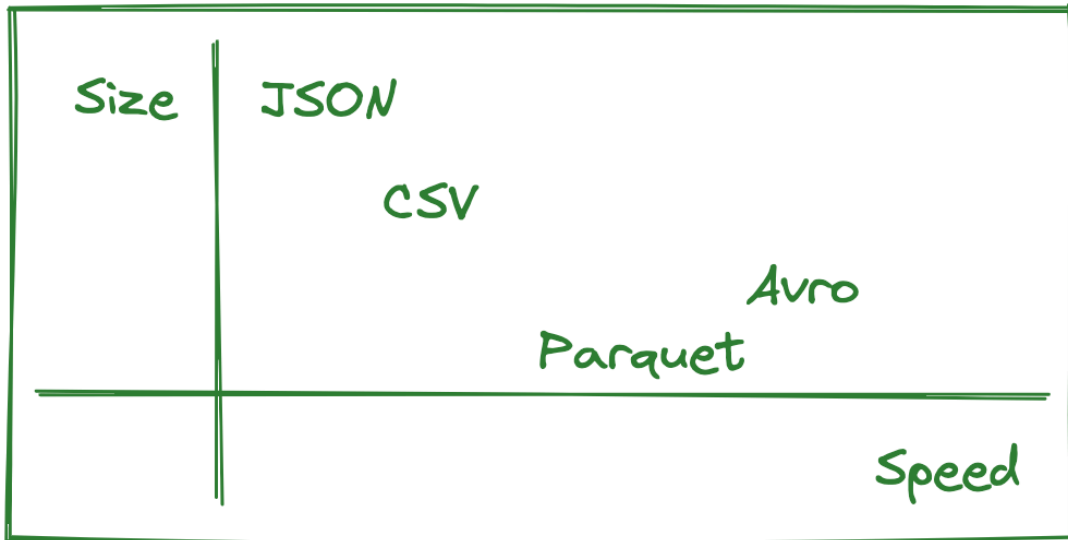
- ✓ Week 8 Feedback Implementation
- System Testing (Described in Testing Strategy W3)
- Security Testing (Report)
- Testing Report (Results & Outcomes)
- External Interfaces Report (TimeEdit, Mazemap etc.)
- Design Choices (Report and Reflection)
- Project Planning (Report and Reflection)
- Prototype/MVP Description (Report)
- + Добавить карточку

**Week 10**

- ✓ Final Poster Development
- ✓ Week 9 Feedback Implementation
- ✓ Individual Contributions
- ✓ Recommendations (Report: comparison graphs, sensors, other project related thing (limitations/dependencies))
- References (Find as many possible: MazeMap documentation, other sources, design choices based on smth may be example)
- Report Refinement
- Final Presentation
- + Добавить карточку

## References

### 1. Data storage formats - size and speed comparison



Source: [CSV vs Parquet vs JSON vs Avro - datacrump.com](https://datacrump.com/CSV-vs-Parquet-vs-JSON-vs-Avro/)

### 2. Flask

Source: [flask.palletsprojects.com/en/stable](https://flask.palletsprojects.com/en/stable/)

### 3. Flask-Login

Source: [flask-login.readthedocs.io/en/latest](https://flask-login.readthedocs.io/en/latest)

### 4. Flask-SocketIO

Source: [flask-socketio.readthedocs.io/en/latest](https://flask-socketio.readthedocs.io/en/latest)

### 5. MazeMap Documentation

Source: [MazeMap JS API 2](https://mazemap.com/docs/js-api-2)

### 6. Mermaid (diagram design)

Source: [Mermaid Live Editor](https://mermaid.live/)

## **7. OAuth Google**

Source: [developers.google.com/identity/protocols/oauth2](https://developers.google.com/identity/protocols/oauth2)

## **8. python-dotenv**

Source: [github.com/theskumar/python-dotenv](https://github.com/theskumar/python-dotenv)

## **9. PostgreSQL**

Source: [www.postgresql.org](https://www.postgresql.org)

## **10. React**

Source: <https://react.dev/learn/thinking-in-react>

## **11. Recharts**

Source: <https://recharts.org/en-US/>

## **12. Supabase PostgreSQL**

Source: [Supabase](https://supabase.com)

## **13. SQLAlchemy**

Source: [www.sqlalchemy.org](https://www.sqlalchemy.org)

## **14. TimeEdit**

Source: [TimeEdit](https://timeedit.com)

## **15. TimeEdit API encode and decode GitHub gist by Eli Saado**

Source: [timeedit scrambling · GitHub](https://gist.github.com/eli-saado/1111111)

## **16. UT Sensors Server**

Source: [sensordata.utsp.utwente.nl](https://sensordata.utsp.utwente.nl)