Intelligent Floor Plan Management System for a Seamless Workspace Experience

-A case study by Mir Razee

Introduction

As organisations strive to foster collaboration and productivity and adapt to dynamic workforce needs, the significance of a well-designed and efficiently managed floor plan becomes essential. With the advent of flexible work arrangements and remote working tools, it's crucial for companies to use their space well. This case study looks at how one company decided to improve its workspace by using a smart Floor Plan Management System.

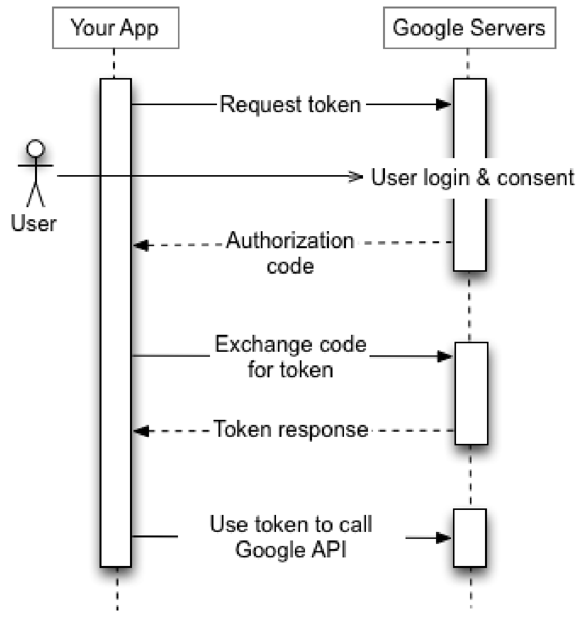
Furthermore, the case study provides a comprehensive overview of the solution, outlining its key features, how it can be implemented, integration with existing technologies, challenges that might come up in the future, as well as the reliability of the system.

Motivation & Objectives

The motivation is quite simple: to foster a more dynamic, collaborative, and efficient work environment. Recognising the evolving nature of workspaces and the diverse needs of its workforce, we need to leverage technology to optimise its floor plan management. The goal is to create a workplace that not only adapts to the changing needs of the employees but also maximises the utility of available resources. Of course, this is to be done in an organised manner with zero risk of conflicts or unintended changes in a reliable and scalable manner.

Methodology

Authentication:
We can use the OAuth 2.0 framework to authenticate users using the accounts that their organisation uses, such as OAuth APIs for Google or Outlook, and then save that information, including role hierarchy in the org. This can then be used to implement which user can access which areas of the office and book rooms accordingly. More information about the OAuth framework can be found here.
One such example is given below for Google API for OAuth



Workflow for authentication using OAuth[1]

Classes:
The following classes can be used to implement the system:

- Users- email, role, team, isAdmin
- Room/desk- ID, capacity, location, status, attributes like contains PC, whiteboard,etc
- Booking- ID, roomID, list of users, timestamp, priority

Floor plan management:
The users who have been assigned as admins can access a portal to upload floor plans.
We can analyse floor plans and convert them into numbers to make them easy to work with. Data can be extracted from a CAD model/floor plan image and converted into room coordinates. Then, we define each floor as a grid and assign unique IDs to each room. Each block in the grid can be numbered with this ID. This room/seat ID can denote the type of room or seat. For example, a standing desk with a PC can be given an ID PCStD, while a fixed desk with two monitors can be given an ID M2SiD, and similarly for meeting rooms and paths.

```
function createFloorPlanWithIDs(coordinates):
    # Find the dimensions of the floor plan
    maxX = findMaxCoordinate(coordinates, 'x')
    maxY = findMaxCoordinate(coordinates, 'y')

    # Initialize a 2D grid
    grid = initializeGrid(maxX, maxY)

    # Place rooms on the grid with unique IDs
    for roomId from 1 to length(coordinates):
        roomCoordinate = coordinates[roomId - 1]
        roomX = roomCoordinate.x
        roomY = roomCoordinate.y
        roomWidth = roomCoordinate.width
        roomHeight = roomCoordinate.height

        # Mark the cells corresponding to the room with its unique ID
        markCellsWithID(grid, roomX, roomY, roomWidth, roomHeight, roomId)

    return grid

# Function to mark cells with a unique room ID based on room coordinates
function markCellsWithID(grid, x, y, width, height, roomId):
    for i from x to (x + width - 1):
        for j from y to (y + height - 1):
            grid[i][j] = roomId  # Use the room ID as the unique
identifier
```

Meeting room bookings:
Each user will be able to see only those rooms that the admin has given access to, and the other rooms can be greyed out. We maintain a bookings object which can store all the booking information like the User ID, Room ID, booking timestamp, time range of the meeting, etc.
We also need to maintain a room object, which stores all the information like its ID, type, attendees, location, etc.
The user booking the room (meeting host) will be asked to add the names of all the invited members. This has a two-fold function: firstly, it will count the number of people expected to attend the meeting (and hence the minimum size of room required), and secondly, it will give the system an idea of the locations of the invitees' desks. Then, the system can use their desk locations and number of attendees to suggest a meeting room that is available at a close and equal distance from all attendees and of minimum size but large enough to accommodate all attendees(Best-fit algorithm). Functionality can also be added to send an automated mail invitation (and later, reminders) to all the invited members through this data collected initially from the host.

The room suggestion algorithm is as follows,
First, find the ideal room IDs for the meeting according to user requirements, such as PR30 or WB5 (Room with projector for max 30, Room with whiteboard for max 5 people, etc). Then, we can simply iterate the grid from the median team location to find all the available rooms. We can display these available rooms by marking them green and marking other rooms as red or grey(incompatible & inaccessible resp.). Finally, we binary search the ideal meeting location and suggest the 3 closest rooms to the location.

```
# Function to retrieve a list of available rooms based on meeting size
function getAvailableRooms(grid, meetingSize):
    availableRooms = []

    for each room in grid:
        if room.capacity >= meetingSize and room.status == "available":
            availableRooms.append(room)

    return availableRooms
```

```
# Function to find the closest 3 rooms to the meeting location using binary search
function findClosestRoomsBinarySearch(sortedAvailableRooms, meetingLocation):
    closestRooms = []  # Initialize an empty list to store closest rooms

    # Binary search to find the initial room closest to the meeting location
    initialRoomIndex = binarySearch(sortedAvailableRooms, meetingLocation)

    # Check rooms around the initial room for the closest 3 rooms
    for i from max(0, initialRoomIndex - 1) to min(initialRoomIndex + 1, length(sortedAvailableRooms) - 1):
        closestRooms.append(sortedAvailableRooms[i])

    return closestRooms
```

Conflict Resolution:
Based on the coordinates of conflict regions, the system can generate a Floor Plan layout highlighting the conflict regions in red or any colour to mark out the conflict.

Conflict resolution can be done in multiple ways, and the following mechanisms can be used in decreasing order of priority;

1) Business priority: This can be a company-defined parameter used to represent the importance of the task, and these tasks should be prioritised over anything else

2) User Hierarchy: The user with higher authority should be able to override existing plans and upload their own. The lower priority task can be shifted to a similar type of room/seat.

3) Timestamp: The exact time at which the user has uploaded the plan; the earlier plan is given priority over the latter one.

This resolution can be made automatically or manually based on the needs of the organisation. The lower-priority bookings will be notified of the change and will be able to modify the booking if needed.

Dynamic updates:
To include Dynamic updates, we can implement a system that continuously monitors and updates the available meeting rooms based on real-time data. This can be achieved by running a continuous loop that periodically refreshes the grid data to stay updated with the latest bookings and room capacities. Now, this is a tradeoff that needs to be taken into consideration; more frequent refreshes will present more accurate data to the user but increase the load on the system, whereas a less frequent update will not show the most latest data, but the overall system will run faster.

```
# Function to dynamically update meeting room suggestions as bookings
occur and capacities change
function dynamicMeetingRoomSuggestions(grid, meetingSize,
meetingLocation):
    # Monitor changes in real-time (e.g., using event listeners or
periodic checks)
    while true:
        # Retrieve the latest grid data, including bookings and room
capacities
        grid = getGridData()

        # Get available rooms based on the latest data
        availableRooms = getAvailableRooms(grid, meetingSize)

        if length(availableRooms) == 0:
            print("No available rooms for the given meeting size.")
        else:
            # Find the closest available room
            closestRoom = findClosestRoom(availableRooms, meetingLocation)

            if closestRoom is not null:
                print("Suggested Meeting Room:", closestRoom)
            else:
                print("No suitable meeting room found.")

        # Sleep for a specified interval before checking for updates again
        sleep(UPDATE_INTERVAL)
```

Version control:
Implementing version control involves tracking changes made to the booking data over time, managing different versions, and providing mechanisms to revert to or merge versions. This can be done by storing all the relevant information related to bookings and storing the timestamp or version number. Then, when a version needs to be retrieved, we can revert all the bookings that have been done till that point in time.

Analytics

Analysing metrics to evaluate the system's performance is very important to optimise workflow and improve efficiency. The following metrics can be used to monitor and improve the work plan in the office:

Utilisation of the office space:
This can be calculated using the simple formula

$$Utilisation = \frac{TimeUsed}{TotalTime} * 100$$

$$TotalUtilisation = \frac{\sum RoomUtilisation}{TotalRooms}$$

We can calculate the time usage of each room using the bookings class, iterating on all the bookings in a time range and finally calculating the utilisation value.
Representing this data visually would be great to showcase which areas are less frequently used and how they can be improved in the upcoming time. This can be done by colour-coding the regions as per the utilisation %. For example, 0%-50% as red, 90%-100% as green, and so on.

Other metrics like satisfaction of users with the meeting room/seats and feedback from the users can also be taken into consideration.

Time & Space complexity:
The time complexity of most algorithms suggested is O(N) per user, where N is the number of rooms per floor. Assuming a worst-case scenario of 500-1000 concurrent users for one floor, this amounts to operations in the order of 10^6, which modern-day systems can easily handle in milliseconds.

References

[1] Using OAuth 2.0 to Access Google APIs
https://developers.google.com/identity/protocols/oauth2