Intelligent Floor Plan Management System for a Seamless Workspace Experience

-A case study by Vaibhav Jaiswani

Introduction

In the pursuit of enhancing collaboration, productivity, and adaptability to evolving workforce demands, the importance of a thoughtfully designed and effectively managed floor plan cannot be overstated.

In response to the changing landscape of work, marked by flexible arrangements and the widespread use of remote tools, companies are compelled to optimize their spatial resources. This case study delves into the proactive steps taken by a particular company to enhance its workspace through the implementation of an intelligent Floor Plan Management System.

Moreover, the case study offers a comprehensive exploration of the solution, delineating its key features, the methodology for implementation, seamless integration with existing technologies, potential challenges on the horizon, and the overall reliability of the system.

Motivation & Objectives

The motivation behind this initiative is straightforward: to cultivate a work environment that is dynamic, collaborative, and efficient. Acknowledging the ever-changing nature of workspaces and the varied requirements of its workforce, there is a need to harness technology for the optimization of floor plan management. The objective is to establish a workplace that not only adjusts to the evolving needs of employees but also optimizes the utilization of available resources. This optimization is to be executed in an organized fashion, ensuring there is no risk of conflicts or unintended changes, and with a focus on reliability and scalability.

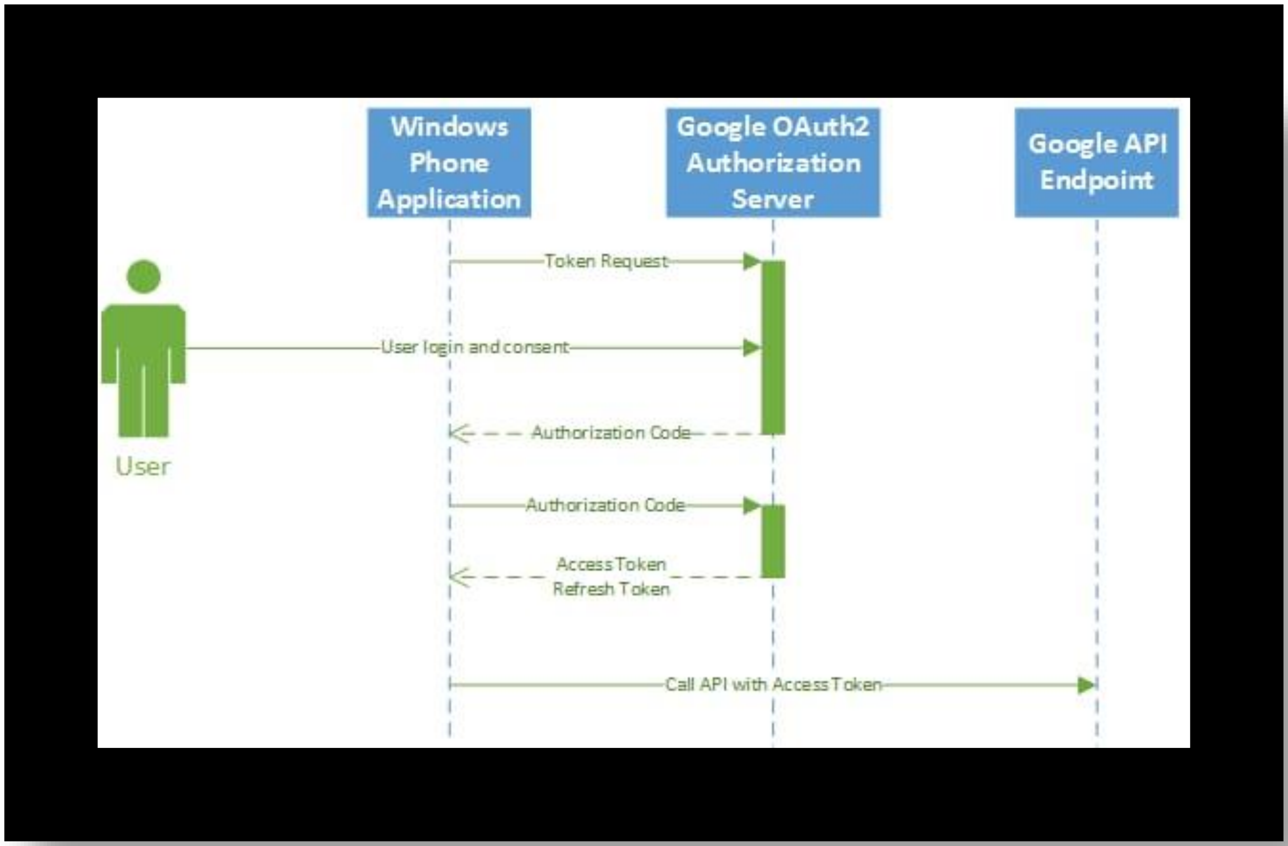
Methodology

Authentication:

The OAuth 2.0 framework can be employed to authenticate users through their organization's accounts, utilizing OAuth APIs like those for Google or Outlook. Subsequently, the gathered information, including the organizational role hierarchy, can be stored. This stored data can then be utilized to determine user access to different office areas and facilitate room bookings based on their permissions. 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]

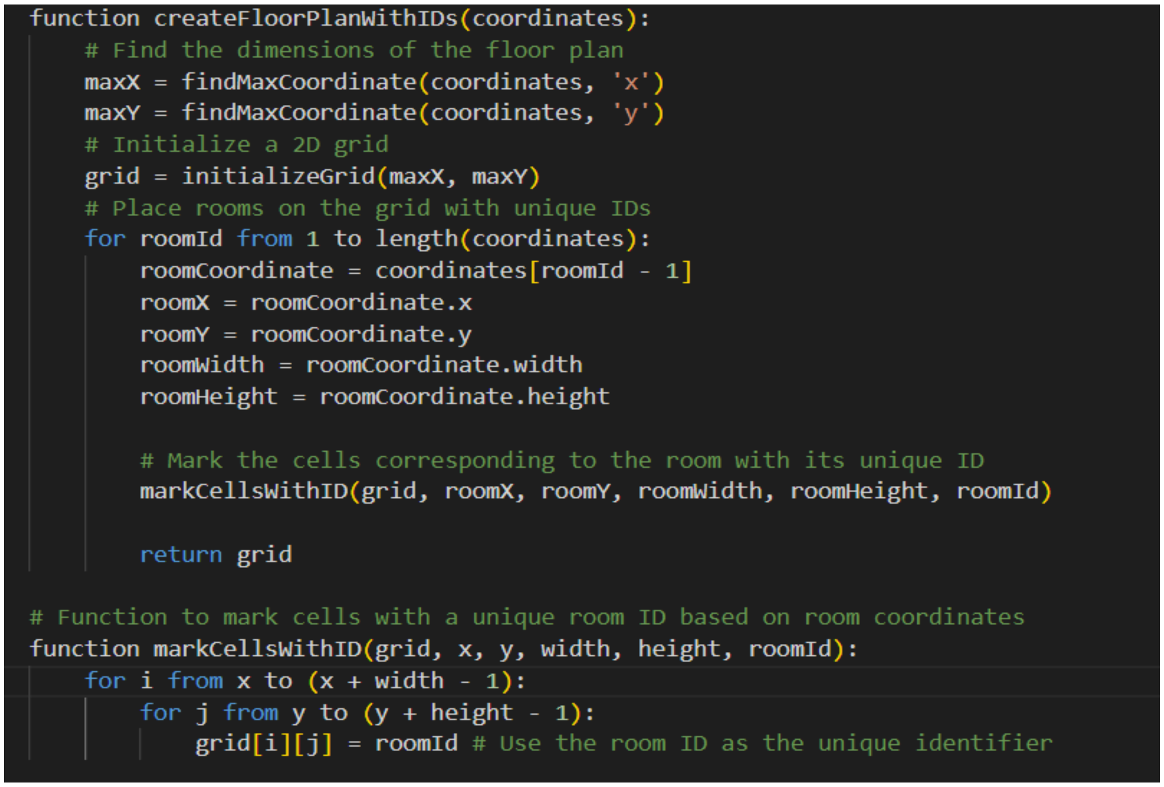
Classes:

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:

Users designated as administrators will have access to a portal for uploading floor plans. The floor plans can undergo analysis, transforming visual information into numerical data for enhanced practicality. Utilizing a CAD model or floor plan image, data extraction is performed to obtain room coordinates. Subsequently, each floor is conceptualized as a grid, and unique IDs are assigned to individual rooms. The grid blocks are then numbered using these IDs, where each room or seat ID carries significance regarding its type. For instance, a standing desk with a PC might be assigned the ID PCStD, while a fixed desk with two monitors could be designated as M2SiD. This categorization extends to meeting rooms and pathways as well.



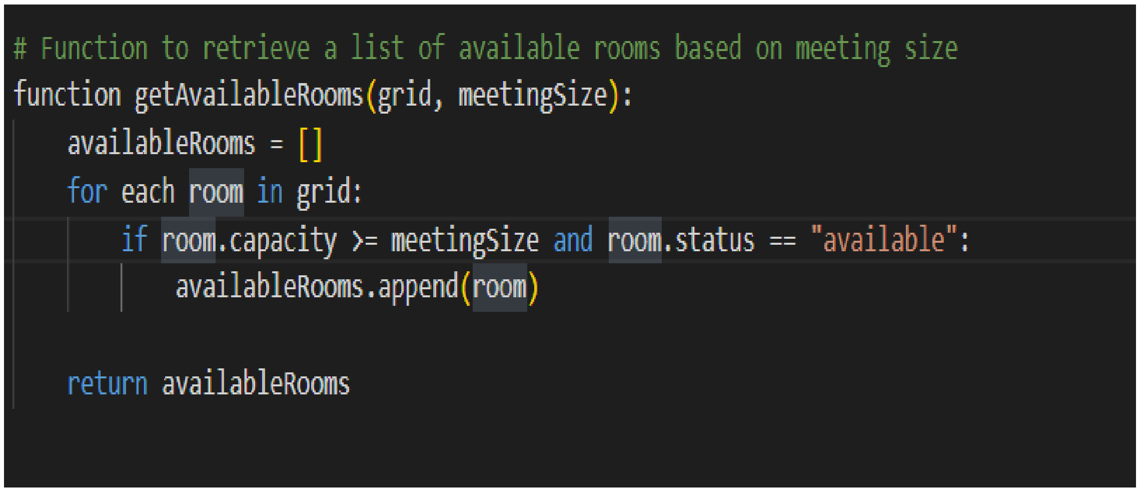
```
function createFloorPlanWithIDs(coordinates):
    # Find the dimensions of the floor plan
    maxX = findMaxCoordinate(coordinates, 'x')
    maxY = findMaxCoordinate(coordinates, 'y')
    # Initialize a 2D grid
    grid = initializeGrid(maxX, maxY)
    # Place rooms on the grid with unique IDs
    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:

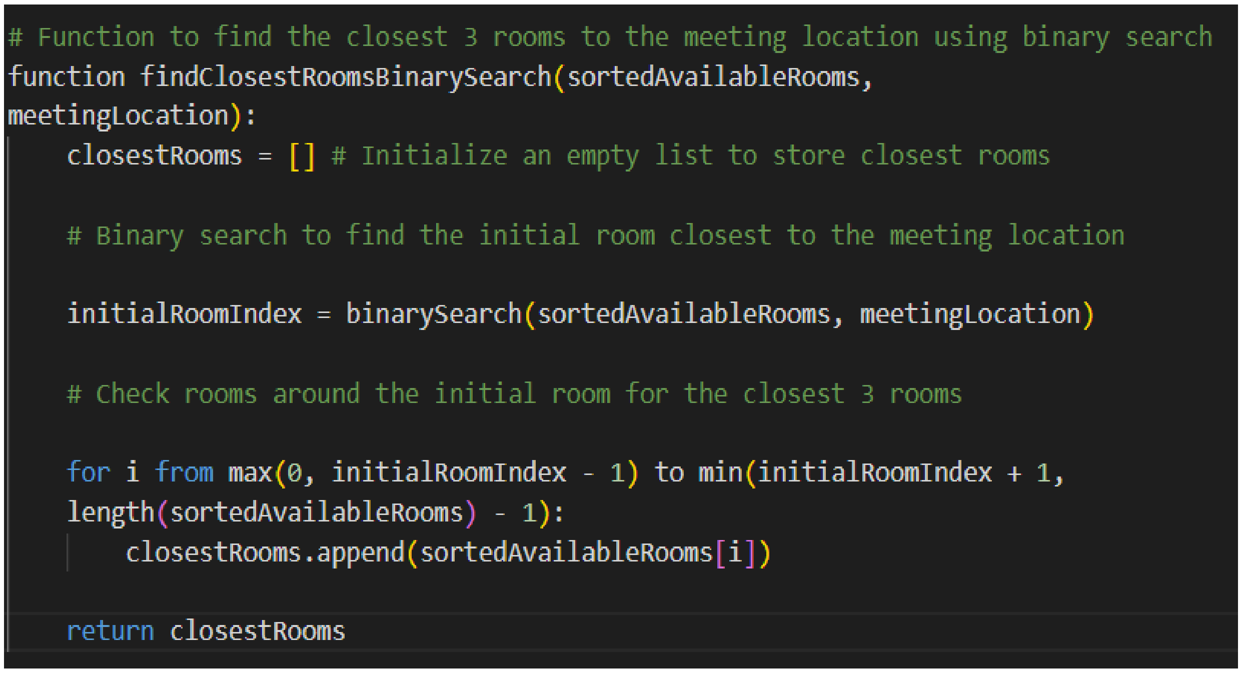
Individual users will have visibility only to the rooms authorized by the administrator, with other rooms appearing greyed out. A bookings object is maintained to store comprehensive booking information, encompassing User ID, Room ID, booking timestamp, meeting time range, and more.

Additionally, a room object is established to house details such as its ID, type, attendees, and location. When a user initiates a room booking, they are prompted to list the names of all invited members. This serves a dual purpose: determining the expected meeting attendance (and thus the requisite room size) and providing the system with information about the locations of the invitees' desks. Leveraging desk locations and attendee count, the system suggests an available meeting room situated equidistant from all invitees and of sufficient size (using a Best-fit algorithm). Further functionality can be implemented to automate email invitations and reminders to all invitees based on the data collected during the initial booking process.

The room suggestion algorithm operates as follows:
Firstly, identify the optimal room IDs for the meeting based on user specifications (e.g., PR30 for a room with a projector for a maximum of 30 people or WB5 for a room with a whiteboard for a maximum of 5 people). Next, iterate through the grid from the median team location to identify all available rooms. These available rooms can be visually marked in green, while incompatible or inaccessible rooms are marked in red or grey. Finally, employ binary search to identify the optimal meeting location and suggest the three closest rooms to that 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:

Utilizing the coordinates of conflict regions, the system can generate a Floor Plan layout that highlights these areas, employing a distinctive color, such as red, to denote conflict.

Conflict resolution encompasses multiple mechanisms, listed in decreasing order of priority:

1) Business Priority: Defined by the company, this parameter reflects the importance of tasks, prioritizing them over other considerations.

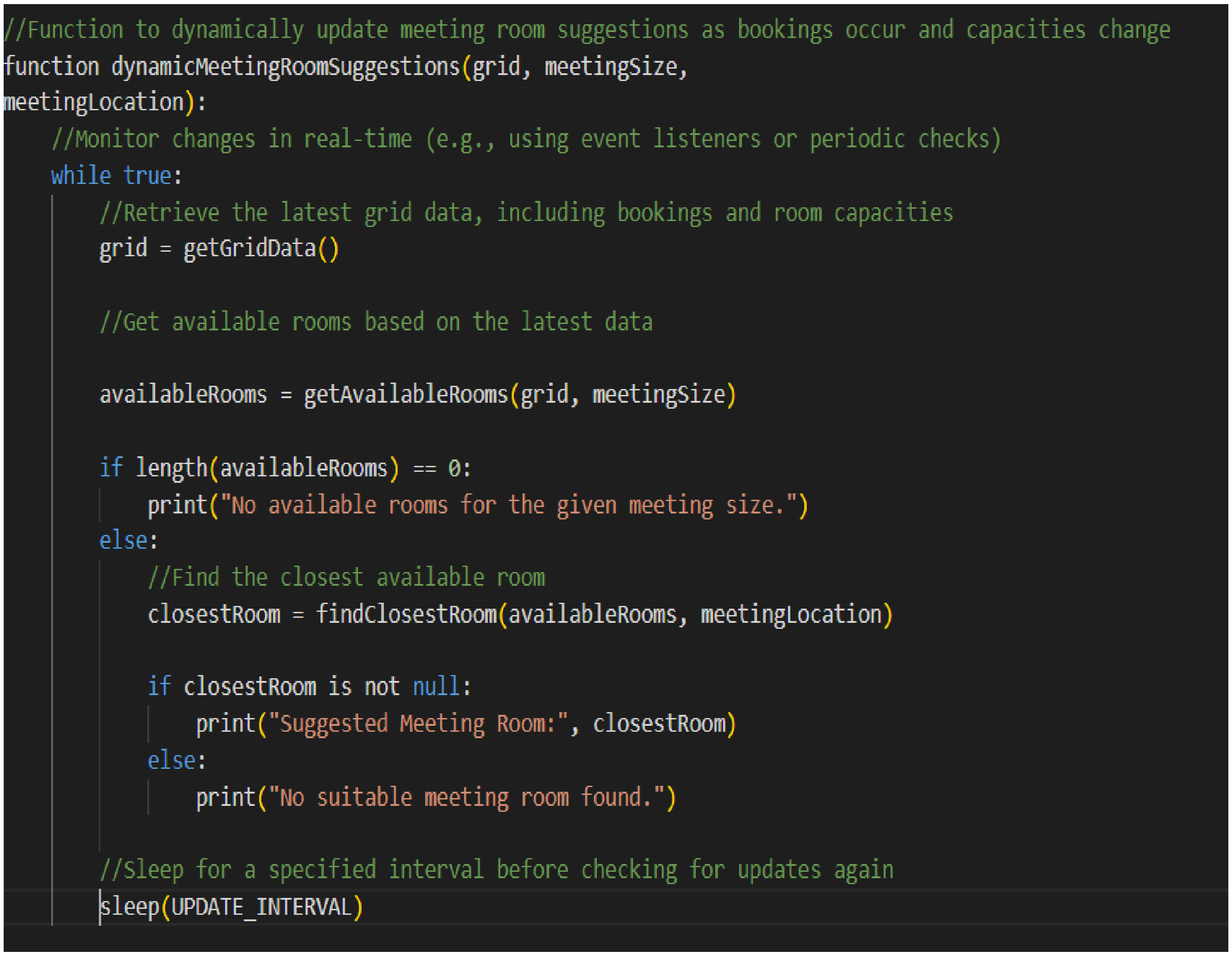
2) User Hierarchy: Users with higher authority possess the capability to override existing plans, allowing them to upload their own plans. Lower-priority tasks can then be shifted to a similar type of room or seat.

3) Timestamp: The time at which a user uploads a plan determines its priority; earlier plans take precedence over subsequent ones.

The resolution process can occur automatically or manually, depending on the organization's requirements. Lower-priority bookings will be notified of the changes and can modify their booking if needed.

Dynamic updates:

To incorporate dynamic updates, a system can be implemented to continually monitor and adjust the availability of meeting rooms in response to real-time data. This involves executing a continuous loop that periodically refreshes the grid data, ensuring it remains synchronized with the latest booking information and room capacities. However, this introduces a tradeoff that requires careful consideration: more frequent refreshes provide more accurate data to users but increase the system load, while less frequent updates may not reflect the very latest information but contribute to overall system efficiency.



```
//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:

The implementation of version control entails the tracking of changes made to booking data over time, the management of various versions, and the provision of mechanisms to revert to or merge specific versions. This can be achieved by storing comprehensive information pertaining to bookings, accompanied by timestamps or version numbers. When a particular version needs retrieval, the system can revert to all the bookings made up to that specific point in time, ensuring the restoration of the desired historical state.

Analytics

Analyzing metrics to evaluate the system's performance is very important to optimize 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

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

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

To determine the time usage of each room, the system can leverage the bookings class by iterating through all bookings within a specified time range and calculating the utilization value. Visual representation of this data provides a compelling way to showcase areas that are less frequently used and suggests improvements for future planning. This can be achieved through color-coding regions based on utilization percentages, such as red for 0%-50%, green for 90%-100%, and so forth. This visual representation offers an intuitive way to identify underutilized and highly utilized spaces within the floor plan.

Other metrics like user satisfaction with the meeting room/seats and user feedback can also be considered.

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