CASE STUDY

INTELLIGENT FLOOR PLAN MANAGEMENT SYSTEM FOR A SEAMLESS WORKSPACE EXPERIENCE

Report by:
Tanwish Yesankar
210001072

ADMIN'S FLOOR PLAN MANAGEMENT (ONBOARDING/MODIFYING THE FLOOR PLANS):

- **Develop a conflict resolution mechanism for simultaneous seat/room information uploads.**
- **Implement a version control system to track changes and merge floor plans seamlessly.**
- **Resolve conflicts intelligently, considering factors such as priority, timestamp, or user roles.**

.**Operational Transformation (OT)** is a technique developed for managing concurrent updates in distributed systems, originally designed for collaborative editing. It ensures **consistency** and **coherence** in shared data despite simultaneous modifications by intelligently addressing conflicts and transforming operations based on concurrent changes.

To implement conflict resolution for simultaneous seat and room uploads using Operational Transformation (OT), define operations, embed metadata with timestamps, and develop intelligent transformation functions. Detect conflicts using timestamps and unique identifiers, applying resolution strategies based on priorities and user roles. Admins use a user-friendly interface for manual conflict resolution, and thorough testing ensures reliability. Adaptive adjustments based on historical conflict resolutions enable continuous improvement, ensuring seamless management of concurrent updates in the seat and room information system.

This is simple version control system in Python for managing floor plans. This basic setup allows seamless **tracking of modifications**, supports **version rollback**, and provides a clear **history of updates.**Given below is the demo implementation code for esatblishing the version control mechanism.There are many alternatives for the version control such as a database that can store the changes made.

```python
from datetime import datetime

class FloorPlan:
    def __init__(self):
        self.current_version, self.version_history = 1,
        {1: {'data': {}, 'timestamp': 'initial'}}

    def update_floor_plan(self, user_id, update_op):
        self.current_version += 1
        self.version_history[self.current_version] = {'data': self.get_curr_data(update_op),
        'timestamp': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
        'user_id': user_id}

    def get_curr_data(self, update_op):
        curr_data = self.version_history[self.current_version]['data'].copy()
        curr_data.update(update_op)
        return curr_data

    def rollback_to_version(self, version):
        self.current_version = version if version in self.version_history else self.current_version

    def display_version_history(self):
        for v, d in self.version_history.items():
            print(f"Version {v} - User: {d['user_id']}, Timestamp: {d['timestamp']}")

# Example usage:
floor_plan = FloorPlan()
floor_plan.update_floor_plan(user_id=1, update_op={'add_seat': 'A1'})
floor_plan.update_floor_plan(user_id=2, update_op={'add_room': 'Conference Room'})
floor_plan.display_version_history()
floor_plan.rollback_to_version(version=1)
print("\nAfter rollback:")
floor_plan.display_version_history()
```

There is also a different method where we can use Merkle Tree for performing the version control.

.Concurrent Updates Harmonization:
Operational Transformation (OT) enables smooth collaboration among administrators by intelligently handling simultaneous updates to the floor plan.

Intelligent Conflict Resolution:
OT ensures conflict resolution is nuanced and considers factors such as priority, timestamps, and user roles, preserving the intentions of administrators during simultaneous edits.

User-Empowered Collaboration:
Administrators are empowered to efficiently collaborate in real-time, with OT dynamically adapting to individual roles and priorities, fostering a coherent and consistent floor plan management system.

The underlying principle of Operational Transformation (OT) is to maintain causal consistency during collaborative editing. It achieves this by preserving the causal order of operations, transforming conflicting operations based on their semantic intent, and ensuring that the final state reflects the intentions of all users involved. This principle allows OT to handle concurrent updates, resolve conflicts intelligently, and support real-time collaboration in distributed systems.

OFFLINE MECHANISM FOR ADMINS (FOR THE UI PERSON):

- **Develop a local storage mechanism for admins to make changes offline.**
- **Implement synchronization to update the server when the internet or server connection is re-established.**
- **Ensure data integrity and consistency during offline and online transitions.**

- **Store Data Locally:**

For Websites: A tool called IndexedDB to save important floor plan details on the user's computer. It can be considered as a private storage space in their web browser.
For Mobile Apps: For phone-based apps, we can use SQLite, a system that helps organize and keep track of your floor plan data on the user's device.

- **Offline Editing & Sync:**

For offline editing, JavaScript captures user changes, saved locally with IndexedDB (web) or SQLite (mobile). Synchronization, triggered by periodic connectivity checks, uses Fetch API (web) or Alamofire (mobile) to send changes to the main server. Conflict resolution ensures consistency between local and server data.

MEETING ROOM OPTIMIZATION (MEETING ROOM SUGGESTIONS AND BOOKING):

- **Develop a meeting room booking system considering the number of participants and other requirements.**
- **Implement a recommendation system to suggest meeting rooms based on capacity and proximity.**
- **Ensure dynamic updates to meeting room suggestions as bookings occur and capacities change.**
- **Show the preferred meeting room based on the last booking weightage.**

A **Database** must be created to store essential details about meeting rooms, such as their capacity, facilities, and availability schedules.A user-friendly interface will be designed, allowing users to easily book meetings while considering factors like the number of participants and specific requirements.**Validations** in the form of filter would ensure if room meets the desired criteria.

A scoring system will take into account various factors such as meeting room attributes, historical data, and user feedback. The relevance of each meeting room will be quantified based on the scoring system.

Push notifications or **web sockets** will be employed to provide instant updates when bookings occur or when meeting room capacities change. Additionally, the recommendation algorithm will be regularly updated to account for changing patterns and user preferences.

The system will prioritize showing the preferred meeting room based on the last booking weightage. This will involve **assigning weightage** to meeting rooms based on factors such as the frequency of past bookings, user rating or specific preferences . The system will dynamically adjust this weightage , ensuring that the preferred meeting room is prominently displayed at the top of the recommendation list.

ADDITIONAL FEATURES

.Authentication
Google Auth and manual login support can be provided.

Authorization
JWT(JSON Web Token) can be used to provide role-based access and authorization.

Architecture
Microservice architecture would be suitable for this implementation.

Scalability
Multiple containers of the same code can be deployed over distributed servers to tackle the load balancing .

OOPs(Object oriented Programming)
The version control demo was showcased in Python , some others languages such as C++ and Java can also be used for the backend.

Backup and Recovery
AWS storage , mongo db or some sql databases can be used for backup and recovery as per the needs.