CASE STUDY

INTELLIGENT FLOOR PLAN MANAGEMENT SYSTEM FOR A SEAMLESS WORKSPACE EXPERIENCE

Admin's Floor Plan Management (Onboarding/Changing the floor plans):

Objective: Develop features for administrators to upload floor plans, addressing potential conflicts during simultaneous updates.

Tasks:

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

Offline Mechanism for Admins (For the UI person):

Objective: Implement an offline mechanism for admins to update the floor plan in internet connectivity loss or server downtime scenarios.

Tasks:

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

Meeting Room Optimization (Meeting Room Suggestions and Booking):

Objective: Enhance the system to optimise bookings, suggesting the best meeting room based on capacity and availability.

Tasks:

- 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.

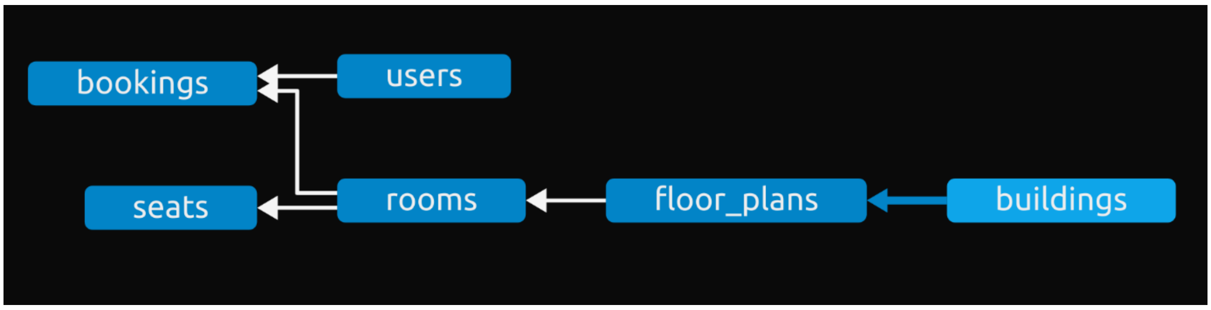
- Ensure dynamic updates to meeting room suggestions as bookings occur and capacities change.
- Show the preferred meeting room based on the last booking weightage.

IMPLEMENTATION - FMS (FLOOR MANAGEMENT SYSTEM)

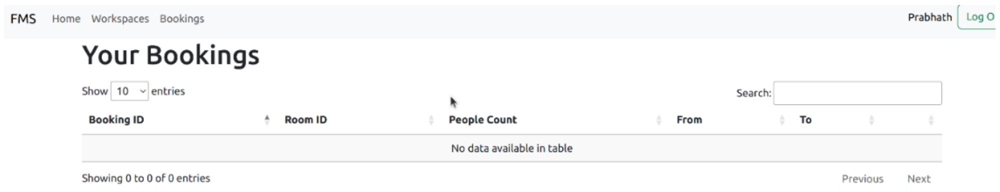
The code is part of a case study for an intelligent floor plan management system. The code is related to the features implemented in the system, the tools used, and the steps required to run the web app.

Implementation - FMS (Floor Management System): This section contains the implementation details, including the features implemented, the tools used, and a screenshot of the workspace page. The key features are:

- A database schema is designed with the following considerations:
  - They are designed such that they are easily modifiable and extendable.
  - CASCADING such that independent child entries are deleted along with parent entries; for example, floors and rooms are removed along with the building.
  - Implemented pre-checks before uploading child entries so that blocking occurs when no parent entry exists; for example, there can't be a floor without a building.
  - I created an interface for each creation and destruction of entities.



- It is a neat, impressive, and secure website with authentication.
- Use the "Your Activity" page to cancel the booking previously made.



Your Bookings

Show 10 entries

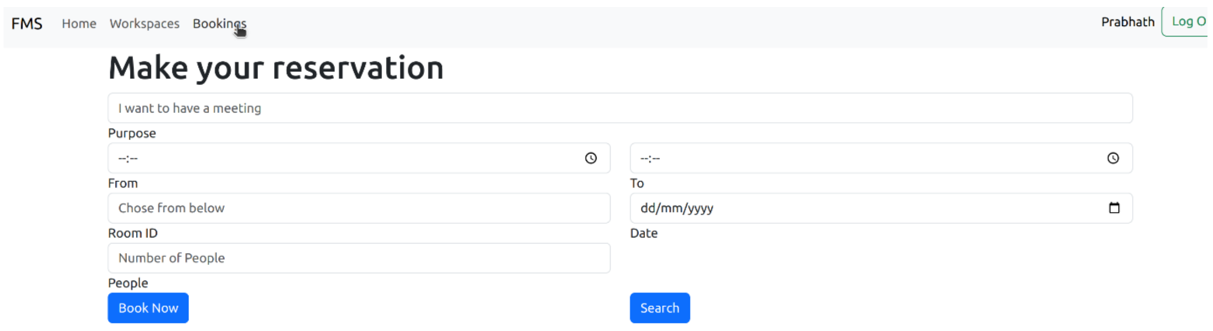
Search:

Booking ID	Room ID	People Count	From	To
No data available in table				

Showing 0 to 0 of 0 entries

Previous Next

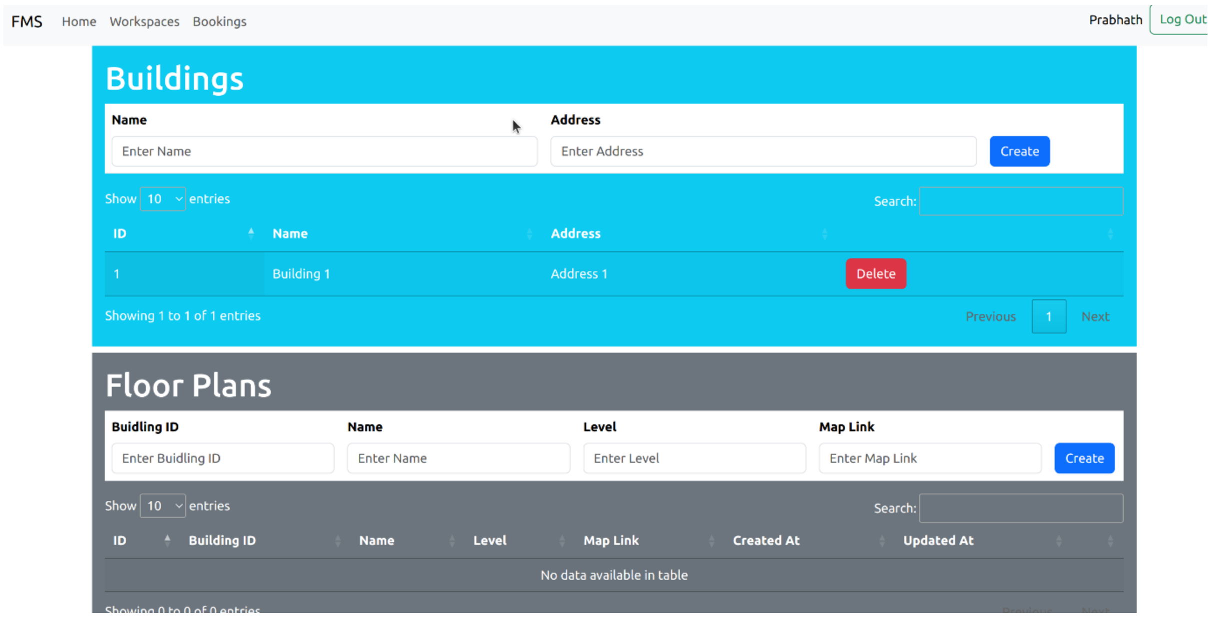
- A "Reservation" page to book the rooms.



Make your reservation

Purpose: I want to have a meeting	
From: --:--	To: --:--
Room ID: Chose from below	Date: dd/mm/yyyy
People: Number of People	

- Running the database on Celery so requests can be populated over Flask will happen whenever the server is up.
- An interactive workspace page to add and delete the floor plans, buildings, rooms, and seats.



Buildings

Name	Address
Enter Name	Enter Address

Show 10 entries Search:

ID	Name	Address	
1	Building 1	Address 1	Delete

Showing 1 to 1 of 1 entries

Floor Plans

Buidling ID	Name	Level	Map Link
Enter Buidling ID	Enter Name	Enter Level	Enter Map Link

Show 10 entries Search:

ID	Building ID	Name	Level	Map Link	Created At	Updated At	
No data available in table							

Showing 0 to 0 of 0 entries

Tools Used:

- Flask
- Celery
- Python

Demo: Google Drive

ADDITIONAL FEATURES THAT I WANT TO ADD

- Making the workspace page more interactive by introducing graphical features
- Using AI/ML models to enhance the recommendation system.
- Researching more over UI to make a customer immersive space.

RUNNING THE WEBAPP - FMS

1. It's better if you create virtualenv to run the project.
2. Install the requirements from the requirements.txt file

```
$ pip install -r requirements.txt
```

3. Install the Redis server and start it (for celery)

```
$ sudo apt install redis-server
$ redis-server
```

4. start the celery_app first

```
$ celery -A make_celery worker --log level INFO
```

5. Now start the task_app in a separate terminal

```
$ flask -A task_app run --debug
```

CODE DETAILS:

1. **Make Celery:**

- The code is a function definition in Python. The function creates a Celery application instance, an asynchronous task queue based on Python.
- The function takes two arguments: flask_app, a Flask application instance, and return, the Celery application instance.
- The function first creates a Flask application instance by calling the create_app function. Then, it retrieves the Celery application instance from the Flask application instance's extensions dictionary using the "celery" key.
- The code is a helper function used to create the Celery application instance in a Flask application.

2. Task_app- __init__.py:

- The code is part of a Python file that creates a Flask application with Celery and Flask-SQLAlchemy as dependencies. It also sets up authentication using Flask-Login.
- The code starts by importing the necessary modules: Flask, SQLAlchemy, path, and Celery. Then, it defines a constant DB_NAME that represents the name of the database file.
- The create_app function creates a new Flask application and sets the configuration using the app. config dictionary. The setup includes the SECRET_KEY, SQLALCHEMY_DATABASE_URI, and CELERY configuration.
- The celery_init_app function initialises Celery with the given Flask application. It creates a custom Celery task that can run in the Flask application context.
- The code then imports the views and auth modules containing the Flask application's blueprints. It also imports the User model and initialises the database using the db.create_all function.
- The code initialises the LoginManager and sets the login_view to the auth. Login route. It also sets the user_loader function to load_user, which returns the User object based on the user ID.
- Finally, the code returns the Flask application.

3. Auth.py:

- The code is part of a Python script that defines the routes for an authentication blueprint in a Flask application. The code is part of a more significant function responsible for managing all routes related to user authentication, including login, logout, and sign-up.
- The code defines the routes using the Flask 'route' decorator. The decorator takes two arguments: the URL pattern and a list of HTTP methods. In this case, the route is defined for the '/login' URL and accepts both 'GET' and 'POST' requests.
- Within the function, the request data is retrieved using the 'request. Form' method, which returns a dictionary of form data. The data is then validated using a series of if statements that check for valid input. If the input is valid, the user is authenticated using the 'login_user' function from the 'flask_login' module.

- Once authenticated, the user is redirected to the homepage. If the user attempts to access a restricted page without being authenticated, they will be redirected to the login page.
- The 'sign_up' route is similar but includes additional validation steps to ensure the user provides valid input. If the input is correct, a new user is created using the 'User' model, and the 'login_user' function is used to authenticate the user.
- The 'logout' route logs out the user by calling the 'logout_user' function.
- Overall, this code validates and authenticates user input using the Flask-Login module.

4. Models.py:

- The code defines the database schema for a fictitious event space management system. It consists of several classes and relationships between them.
- The classes are:
  - FloorPlan: A class representing a floor plan in the event space.
  - Room: A class that represents a room in the event space.
  - Seat: A class that represents a seat in a room.
  - Building: A class that represents a building in the event space.
  - User: A class that represents a user in the system.
  - Booking: A class that represents a booking in the system.
- The relationships are:
  - FloorPlan.rooms: A relationship that connects FloorPlan to Room.
  - Room.seats: A relationship that connects Room to Seat.
  - Room.bookings: A relationship that connects Room to Booking.
  - Seat.room: A relationship that connects Seat to Room.
  - Booking.room: A relationship that connects Booking to Room.
  - Booking.user: A relationship that connects Booking to User.
- The FloorPlan class has five attributes: id, building_id, name, level, image_file, created_at, and updated_at. The id attribute is the primary key of the FloorPlan, building_id is the foreign key to the Building that the FloorPlan belongs to, name is the name of the FloorPlan, level is the level of the FloorPlan, image_file is the file name of the image of the FloorPlan, created_at is the datetime when the FloorPlan was created, and updated_at is the datetime when the FloorPlan was last updated. The FloorPlan class also has a relationship called rooms, which connects FloorPlan to Room.
- The Room class has six attributes: id, floor_plan_id, name, type, capacity, equipment, and created_at and updated_at. The id attribute is the primary

key of the Room, floor_plan_id is the foreign key to the FloorPlan that the Room belongs to, name is the name of the Room, type is the type of the Room (e.g., lecture hall, workshop room), capacity is the capacity of the Room, equipment is the equipment available in the Room, and created_at and updated_at are the datetime when the Room was created and last updated. The Room class also has two relationships: seats and bookings. The seats relationship connects Room to Seat, and the bookings relationship connects Room to Booking.

- The Seat class has three attributes: id, room_id, and label. The id attribute is the primary key of the Seat, room_id is the foreign key to the Room that the Seat belongs to, and the label is the Seat's label.
- The Building class has three attributes: id, name, and address. The id attribute is the primary key of the Building, the name is the name of the Building, and the address is the address of the Building. The Building class also has a relationship called floor_plans, which connects Building to FloorPlan.
- The User class has five attributes: id, email, role, password, and first_name. The id attribute is the user's primary key, email is the user's email, role is the user's role (e.g., admin, regular user), password is the user's password, and first_name is the user's first name. The User class inherits from the UserMixin class from Flask-Login, providing the necessary authentication methods.
- The Booking class has seven attributes: id, room_id, user_id, people_count, start_time, end_time, and purpose. The id attribute is the primary key of the Booking, room_id is the foreign key to the Room that the Booking is for, user_id is the foreign key to the User that made the Booking, people_count is the number of people in the Booking, start_time is the start time of the Booking, end_time is the end time of the Booking, and purpose is the purpose of the Booking. The Booking class also has a status attribute, which indicates the Booking status (e.g., open, cancelled).
- The code then defines several unique constraints to ensure the integrity of the data in the database. For example, there is a unique constraint on the (name, address) fields in the building class to ensure each Building has a unique name and address.

The code defines a comprehensive database schema for a fictitious event space management system consisting of several classes and relationships between them.

5. databaseControl.py:

- The code is part of a Database model module that holds database functions for a building management system. The functions allow the creation, modification, and deletion of the database's buildings, floors, rooms, and seats.
- The code is well-structured and follows a clear structure. The module docstring supplies an overview of the module and its purpose. In contrast, the function docstrings offer detailed information about each function, including its purpose, inputs, outputs, and other relevant information.
- The code is well-documented with explicit comments and docstrings that explain its purpose and functionality. This makes the code easy to understand and support and helps ensure the functions are used correctly.
- Overall, the code is a high-quality example of database functions that can be used as a reference for developing similar code in other projects.

6. Views.py:

The code is part of the blueprint for the tasks page in the software application, where the requests from the HTML are captured. The tasks page allows users to interact with asynchronous tasks running in the background, check their status, and cancel them if necessary. The code is part of the function that returns the result of an asynchronous task. The function takes an ID as a parameter and returns a dictionary with ready, successful, and value keys. The ready key shows whether the task is prepared, the successful key means whether the task was successful, and the value key contains the task result.

The code is well-structured and follows best practices in software development and security. It is well-documented using Python docstrings and follows the MVC architectural pattern.

7. Task.py:

The code is part of a Python script implementing a Room booking system. The script holds three functions: cancel, search, and book.

The cancel function is used to cancel a booking based on the booking ID. It uses the SQLalchemy library to query the database for the booking with the specified ID, and if the booking is found, it updates the status to "closed" and commits the changes to the database. If the booking is not found, the function prints an error message.

The search function is used to search for available rooms for a booking based on the specified criteria. It uses the SQLalchemy library to query the database for rooms with at least the requested seats available and not booked during the desired period. The function returns a list of available rooms, where each room is represented as a list of its attributes, including a button to choose the room.

The book function is used to book a room based on the specified criteria. If a booking with the same criteria already exists, it will be updated with the latest information. It uses the SQLalchemy library to query the database for existing bookings with the same criteria, and if a booking is found, it updates the status to "open" and the number of people to booking and commits the changes to the database. If no booking is located, a new booking is created. The function also manages cases with duplicate bookings and updates the status accordingly.