

Intelligent Floor Plan Management System for a Seamless Workspace Experience

Suyog Gupta

Introduction

For the case study we were asked to develop an Intelligent Floor Plan Management System with three key features

1. Update seat/room information without conflict resolution.
2. Mechanism for synchronization of online changes.
3. Optimized room booking.

The implementation that has been done has these functionalities-

- Validation of user via login page
- A validated user can either update a room information or book a room.
- We have handled concurrency in all these processes.
- We have used in-memory cache for faster API response.
- We have handled exceptions by various checks to avoid code-break.

Note - Instead of throwing exception we have returned strings and displayed on the screen.

- We have used JAVA, Spring, SpringBoot and Thymeleaf
-

ProjectApplication.java

```
package com.floorplanmanagment.Project;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ProjectApplication {

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(primarySource:ProjectApplication.class, args);
    }

}
```

This code defines the Spring Boot application with a main class (ProjectApplication) annotated with @SpringBootApplication. The main method initializes and starts the Spring Boot application using SpringApplication.run. This setup allows for the automatic configuration of the Spring context and other application components, making it easy to build and run Spring-based applications.

AppController.java

The code represents a Spring MVC controller for the application. Let's go through the functionalities of the code:

```
@Controller
public class AppController {

    @Autowired
    private FloorPlanService floorPlanService;
```

1. Login Page:

```
@RequestMapping("/")
public String loginPage(Model model)
{
    model.addAttribute(attributeName:"user", new User());
    return "login";
}
```

URL: `/`

Functionality: Prepares the model for the login page by adding a new `User` object. This method is responsible for rendering the login page at the Root.

2. Home Page:

```
@RequestMapping("/home")
public String firstHandler(Model model){

    return "home";
}
```

URL: `/home`

Functionality: Displays the home page. The home page comes up with option to select option whether to update information or book a room.

3. Room Page:

```
@RequestMapping("/room")
public String roomPage(Model model){
    model.addAttribute(attributeName:"booking", new Booking());
    return "room";
}
```

URL: `/room`

Functionality: Prepares the model for the room page by adding a new `Booking` object. This implies that it handles the functionality related to booking rooms, and this method handles the rendering of the room booking page.

4. Save Page:

```
@RequestMapping("/save")
public String savePage(Model model){
    model.addAttribute(attributeName:"room", new Room());
    return "save";
}
```

URL: `/save`

Functionality: Prepares the model for the save page by adding a new `Room` object. This suggests there is a functionality related to saving room

information, and this method handles the rendering of the page where users can save room details.

5. RESTful Endpoints for Room Data:

```
@GetMapping("/getAll")
@ResponseBody
public ArrayList<Room> getAll()
{
    return floorPlanService.getAll();
}

@GetMapping("/getById")
@ResponseBody
public String getById(int id)
{
    return floorPlanService.getRoomByRoomId(id).toString();
}
```

URL: `/getAll`

Functionality: Returns a list of all rooms in JSON format. This is a RESTful endpoint that provides information about all available rooms.

URL: `/getById`

Functionality: Returns room information for a specific room ID in JSON format. This is another RESTful endpoint that retrieves information about a specific room.

6. Save Room Information:

```
@PostMapping("/saveRoom")
public String saveRoomInfo(@ModelAttribute("room") Room room, BindingResult bindingResult, Model model)
{
    if (bindingResult.hasErrors()) {
        return "save";
    }
    String res = floorPlanService.savefloorPlan(room);
    if(res.equals("SUCCESS"))
    {
        model.addAttribute(attributeName:"message", attributeValue:"Information Updated Successfully");
    }
    else{
        model.addAttribute(attributeName:"message", res);
    }

    return "message";
}
```

URL: `/saveRoom`

Functionality: Handles the submission of a form to save room information. Validates the form input and uses the `floorPlanService` to save the floor plan. Displays a success or error message on the "message" page based on the result.

7. Book Room:

```
@PostMapping("/book")
public String bookRoom(@ModelAttribute("booking") Booking booking, BindingResult bindingResult, Model model)
{
    int roomId = floorPlanService.getRoomAllocation(booking);
    if(roomId==-1)
    {
        model.addAttribute(attributeName:"message", attributeValue:"No rooms available");
    }
    else
    {
        model.addAttribute(attributeName:"message", "Room No "+roomId+" Booked");
    }
    return "message";
}
```

URL: `/book`

Functionality: Handles the submission of a form to book a room. Uses the `floorPlanService` to check room availability and displays a success or error message on the "message" page based on the result.

8. User Login:

```
@PostMapping("/login")
public String loginUser(@ModelAttribute("user") User user, BindingResult bindingResult, Model model)
{
    String res = floorPlanService.validateUser(user.getId(), user.getPassword());
    if(res==Constants.INVALID_USER)
    {
        model.addAttribute(attributeName:"message", attributeValue:"User does not exist");
        return "message2";
    }
    else if(res==Constants.INVALID_PASSWORD)
    {
        model.addAttribute(attributeName:"message", attributeValue:"Passowrd Entered is incorrect for the user");
        return "message2";
    }
    else
    {
        model.addAttribute(attributeName:"name", res);
        return "home";
    }
}
```

URL: `/login`

Functionality: Handles the submission of a form for user login. Uses the `floorPlanService` to validate user credentials and displays a success or error message on the "message2" page based on the result.

In summary, the code provides functionalities for user authentication, viewing and booking rooms, saving room information, and exposing RESTful endpoints for room data retrieval. The actual business logic for these functionalities is implemented in the `floorPlanService` component.

FloorPlanService.java

This code defines a service class named `FloorPlanService` that provides various functionalities related to floor plan management. Let's go through the key functionalities:

```
@Component("floorPlanService")
public class FloorPlanService {

    @Autowired
    private FloorPlanRepository floorPlanRepository;

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private OfflineSyncService offlineSyncService;

    private Map<Integer, Room> roomsByroomId = Collections.synchronizedMap(new HashMap<>());
    private Map<Integer, User> usersById = Collections.synchronizedMap(new HashMap<>());
}
```

1. Dependencies:

`FloorPlanRepository`: This is the data access class responsible for interacting with the database to perform CRUD operations on floor plans.

`UserRepository`: Similar to `FloorPlanRepository`, it interacts with the database to handle user-related data.

`OfflineSyncService`: A service responsible for managing offline synchronization changes.

2. Caching:

```
@PostConstruct
public void cache()
{
    // synchronizeOfflineChanges();
    ArrayList<Room> allRooms = floorPlanRepository.getAllRooms();
    ArrayList<User> allUsers = userRepository.findAll();
    Map<Integer,Room> newMap = new HashMap<>();
    for (Room room : allRooms) {
        newMap.put(room.getId(), room);
    }
    Map<Integer,User> userMap = new HashMap<>();
    for (User user : allUsers) {
        userMap.put(user.getId(), user);
    }
    roomsByroomId = Collections.synchronizedMap(newMap);
    usersByuserId = Collections.synchronizedMap(userMap);
}
```

The class uses two synchronized maps (`roomsByroomId` and `usersByuserId`) to cache room and user data for faster access.

The `@PostConstruct` annotated method `cache()` is called after the bean has been constructed. It initializes the cache by loading data from the database into the synchronized maps.

3. Room and User Retrieval:

```
public ArrayList<Room> getAll()
{
    ArrayList<Room> result = floorPlanRepository.getAllRooms();
    return result;
}
```

- `getAll()`: Retrieves a list of all rooms from the database.

```
public Room getRoomById(int num)
{
    return floorPlanRepository.getById(num);
}

public Room getRoomByRoomId(int num)
{
    return roomsByroomId.get(num);
}
```

`getRoomById(int num)`: Retrieves a specific room by its ID from the database.

`getRoomByRoomId(int num)`: Retrieves a specific room from the cached map using its ID.

4. Save Floor Plan Information:

```
@Transactional
public String savefloorPlan(Room room)
{
    room.setTimestamp(new Timestamp(System.currentTimeMillis()));
    Room existingRoom = roomsByroomId.get(room.getId());
    if (existingRoom!=null && existingRoom.getTimestamp().after(room.getTimestamp())) {
        return "The floor plan has been updated more recently by another user";
    }
    floorPlanRepository.saveInfo(room);
    roomsByroomId.put(room.getId(),room);
    return "SUCCESS";
}
```

`savefloorPlan(Room room)`: Saves floor plan information to the database. It checks for updates from other users using timestamps to avoid conflicts. The updated room information is also stored in the cache.

To ensure data integrity the '@Transactional' helps us.

To avoid conflict it is checked that the timestamp associated with the latest update is not after the update that is going to be performed.

5. Offline Changes:

```
@Transactional
public void saveOfflineChange(Room room)
{
    Room existingRoom = roomsById.get(room.getId());
    if (existingRoom != null && existingRoom.getTimestamp().after(room.getTimestamp())) {
        return;
    }
    floorPlanRepository.saveInfo(room);
    roomsById.put(room.getId(), room);
}
```

`saveOfflineChange(Room room)`: Saves offline changes to the database and updates the cache. It performs a similar timestamp-based check to avoid conflicts.

6. Room Allocation:

```

@Transactional
public int getRoomAllocation(Booking booking)
{
    int num = booking.getNum();
    Room room = null;

    for (Integer Id : roomsByroomId.keySet()) {
        Room thisRoom = roomsByroomId.get(Id);
        if(thisRoom.getStatus()==Constants.AVAILABLE)
        {
            if(room != null)
            {
                if(thisRoom.getCapacity() < room.getCapacity() && thisRoom.getCapacity() >= num){
                    room = thisRoom;
                }
                else if(room.getCapacity() == thisRoom.getCapacity())
                {
                    if(thisRoom.getTimestamp().before(room.getTimestamp()))
                    {
                        room = thisRoom;
                    }
                }
            }
            else
            {
                if (thisRoom.getCapacity() >= num) {
                    room = thisRoom;
                }
            }
        }
    }

    if(room == null)
    {
        return -1;
    }
    else
    {
        updateCache(room);
        return room.getId();
    }
}

```

`getRoomAllocation(Booking booking)`: Allocates a room for booking based on certain criteria, such as room availability, capacity, and timestamp. It

updates the cache and returns the allocated room ID or -1 if no rooms are available.

It returns the most optimal solution by booking the room with least capacity (ofcourse more than the needed capacity), when there is the same capacity the room which was updated more time ago is booked, as it will reduce conflicts due to merge time intervals.

7. User Validation:

```
public String validateUser(int userId, String password)
{
    User user = usersById.get(userId);
    if(user==null)
    {
        return Constants.INVALID_USER;
    }
    else if(!password.equals(user.getPassword()))
    {
        return Constants.INVALID_PASSWORD;
    }
    else{
        return user.getName();
    }
}
```

`validateUser(int userId, String password)`: Validates a user based on the provided user ID and password. It returns a status constant (`INVALID_USER`, `INVALID_PASSWORD`, or the user's name).

8. Offline Changes Synchronization:

```
public void synchronizeOfflineChanges() {
    ArrayList<Room> offlineChanges = offlineSyncService.getAllOfflineChanges();

    for (Room room : offlineChanges) {
        saveOfflineChange(room);
    }
}
```

`synchronizeOfflineChanges()`: Retrieves offline changes from the `OfflineSyncService` and applies them to the database and cache.

In summary, this service class encapsulates the business logic for floor plan management, including room allocation, user validation, offline synchronization, and caching for improved performance. The class interacts with repositories to perform database operations and uses synchronized maps to cache frequently accessed data.

OfflineSyncService.java

This code defines a service class named `OfflineSyncService`, which is responsible for managing offline synchronization changes related to room data. Let's go through the key functionalities:

```
@Service
public class OfflineSyncService {

    @Autowired
    private OfflineChangeRepository offlineChangeRepository;

    public void saveOfflineChange(Room room) {
        offlineChangeRepository.saveInfo(room);
    }

    public ArrayList<Room> getAllOfflineChanges()
    {
        ArrayList<Room> res = offlineChangeRepository.findAll();
        offlineChangeRepository.deleteAll();
        return res;
    }
}
```

1. Dependencies:

`OfflineChangeRepository`: The data access class responsible for interacting with the database to perform CRUD operations on offline changes.

2. Save Offline Change:

`saveOfflineChange(Room room)`: Saves offline changes to the database using the `offlineChangeRepository`. It seems that this method is called when there are changes to room data that need to be synchronized offline.

3. Get All Offline Changes:

`getAllOfflineChanges()`: Retrieves all offline changes from the database using the `offlineChangeRepository`. After retrieval, it deletes all the offline

changes from the database. This method is likely called when synchronization is needed, and the changes need to be applied.

4. Note:

It's important to mention that deleting all offline changes after retrieval implies a one-time consumption of the changes. Once the changes are retrieved, they are removed from the repository. If a different strategy is needed, such as maintaining a history of changes, this behavior should be adjusted accordingly.

In summary, the `OfflineSyncService` is a service class that interacts with an `OfflineChangeRepository` to save and retrieve offline changes related to room data. This service can be used to manage synchronization between offline and online data, particularly useful in scenarios where the application needs to handle data changes made in an offline mode.

FloorPlanRepository.java

```
public interface FloorPlanRepository extends CrudRepository<Room,Integer>{

    @Modifying
    @Query("INSERT INTO Room (id, status, capacity, timestamp) VALUES (:#{#room.id}, :#{#room.status}, :#{#room.capacity}, :#{#room.timestamp})")
    void saveInfo(@Param("room") Room room);

    @Query(value = "SELECT * FROM room WHERE id = :id ORDER BY timestamp DESC LIMIT 1", nativeQuery = true)
    Room getById(@Param("id")int id);

    @Query(value = "WITH RankedRoom AS ( SELECT room.*, ROW_NUMBER() OVER (PARTITION BY id ORDER BY TIMESTAMP DESC) AS RowRank FROM room )SELECT id,STATUS,capacity,TIMESTAMP
    ArrayList<Room> getAllRooms();

}
```

This code defines a Spring Data JPA repository interface named `FloorPlanRepository`, which extends `CrudRepository` to perform CRUD operations on the `Room` entity. Let's go through the key functionalities:

1. Save Information Query:

`saveInfo(@Param("room") Room room)`: This method is annotated with `@Modifying` and `@Query`. It's used to insert room information into the database. The `:#{#room.id}`, `:#{#room.status}`, `:#{#room.capacity}`, and `:#{#room.timestamp}` are placeholders for the corresponding attributes of the `Room` entity. The `@Param` annotation is used to map the `room` parameter to these placeholders.

2. Get Room by ID Query:

`getById(@Param("id") int id)`: This method retrieves a room from the database based on the provided room ID. It uses a native SQL query with the

`@Query`` annotation. The ``ORDER BY timestamp DESC LIMIT 1`` ensures that the most recent entry for the specified room ID is retrieved.

3. Get All Rooms Query:

``getAllRooms()``: This method retrieves all rooms from the database. It uses a native SQL query with the `@Query`` annotation. The query employs a common table expression (CTE) named ``RankedRoom`` to assign row numbers to entries based on the timestamp in descending order. It then selects only the rows where the row number is 1, effectively giving the most recent entry for each room.

In summary, this repository interface provides methods for saving room information, retrieving a room by ID, and getting a list of all rooms. The queries use native SQL and are annotated with Spring Data JPA annotations to map method parameters and return values to the corresponding database entities.

Note - `OfflineChangeRepository.java` for offline changes is also built in a similar way

UserRepository.java

```
package com.floorplanmanagment.Project;
import java.util.ArrayList;

import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<User,Integer>{

    public ArrayList<User> findAll();
}
```

This code defines a Spring Data JPA repository interface named `UserRepository` that extends `CrudRepository`. It is used to perform CRUD operations on the `User` entity. Let's break down the key elements:

`public ArrayList<User> findAll();`: This method is used to retrieve all users from the database. It overrides the `findAll` method provided by the `CrudRepository`. The return type is an `ArrayList<User>`, indicating that it returns a list of all users.

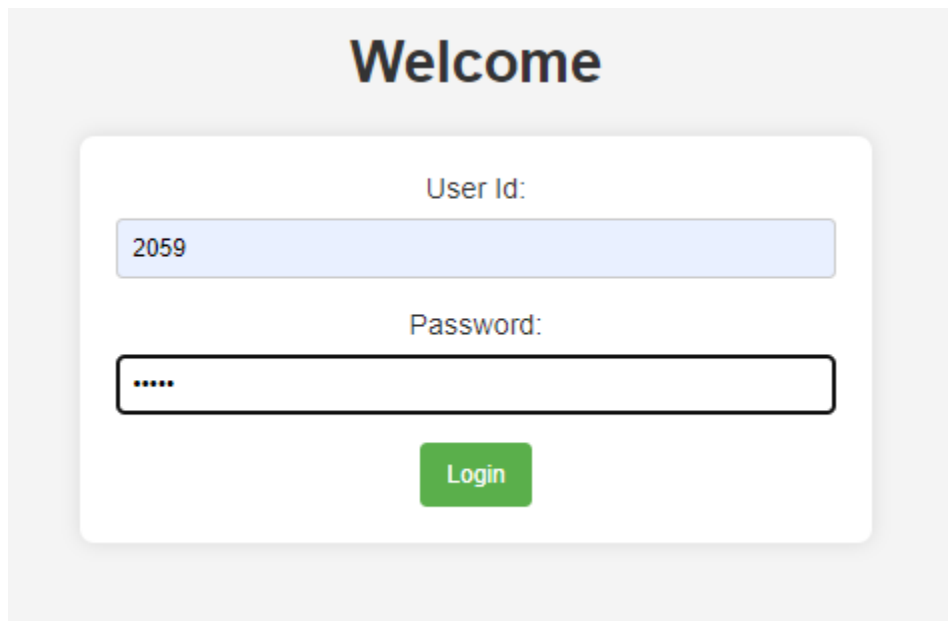
Room.java & User.java

These classes are the JPA entities representing information about a room and user respectively.

Frontend | UI

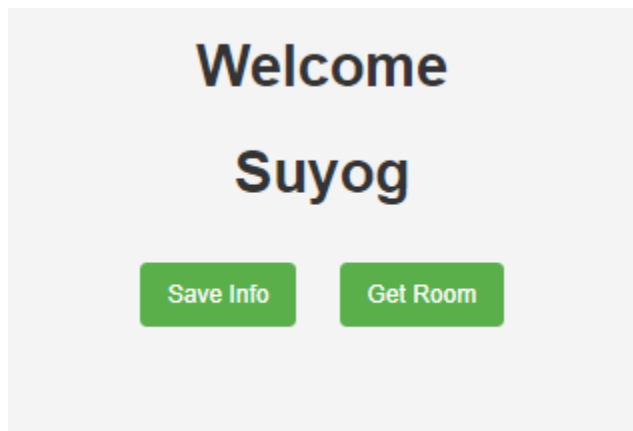
We have used the Thymeleaf template to achieve the functionalities of UI.

Login Page



The login page features a light gray background. At the top center, the word "Welcome" is displayed in a large, bold, black font. Below this, a white rounded rectangle contains the login form. Inside the form, the text "User Id:" is positioned above a light blue input field containing the value "2059". Below this, the text "Password:" is positioned above a white input field with black dots representing masked characters. At the bottom center of the white form is a green button with the text "Login" in white.

Home Page



The home page has a light gray background. At the top center, the word "Welcome" is displayed in a large, bold, black font. Below it, the name "Suyog" is displayed in a slightly smaller, bold, black font. At the bottom of the page, there are two green buttons with white text: "Save Info" on the left and "Get Room" on the right.

Save Room/Plan Page

Welcome

Floor Plan Form

Room No:

Status:

☒ Available ☐ Unavailable

Capacity:

Room Booking Page

Welcome

Capacity Needed:

10

Book Room

Room No 201 Booked

[Return to Home](#)

Database

<input type="checkbox"/>	id	status	capacity	timestamp
<input type="checkbox"/>	301	1	80	2023-12-07 07:16:25
<input type="checkbox"/>	301	1	80	2023-12-07 07:22:22
<input type="checkbox"/>	101	1	20	2023-12-07 07:22:48
<input type="checkbox"/>	102	1	20	2023-12-07 07:22:52
<input type="checkbox"/>	103	1	20	2023-12-07 07:22:55
<input type="checkbox"/>	104	1	20	2023-12-07 07:22:57
<input type="checkbox"/>	105	1	50	2023-12-07 07:23:05
<input type="checkbox"/>	106	1	50	2023-12-07 07:23:12
<input type="checkbox"/>	110	1	100	2023-12-07 07:23:21
<input type="checkbox"/>	201	1	10	2023-12-07 07:23:31
<input type="checkbox"/>	202	1	10	2023-12-07 07:23:34
<input type="checkbox"/>	203	1	18	2023-12-07 07:23:40
<input type="checkbox"/>	301	1	80	2023-12-07 07:23:52
<input type="checkbox"/>	302	1	80	2023-12-07 07:23:56
<input type="checkbox"/>	105	0	50	2023-12-07 07:24:26
<input type="checkbox"/>	302	1	80	2023-12-07 07:25:27
<input type="checkbox"/>	302	1	80	2023-12-07 07:25:42

This is the room table that contains the id,status, capacity, timestamp.The id is the room number and status = 1 means available and status = 0 means unavailable, a same room can have different entries at different timestamp, it allows to keep track of the history of updates of room.

Note-There is scope of normalization for this table but for simplicity, it has been kept like this.