MoveInSync SDE Case Study

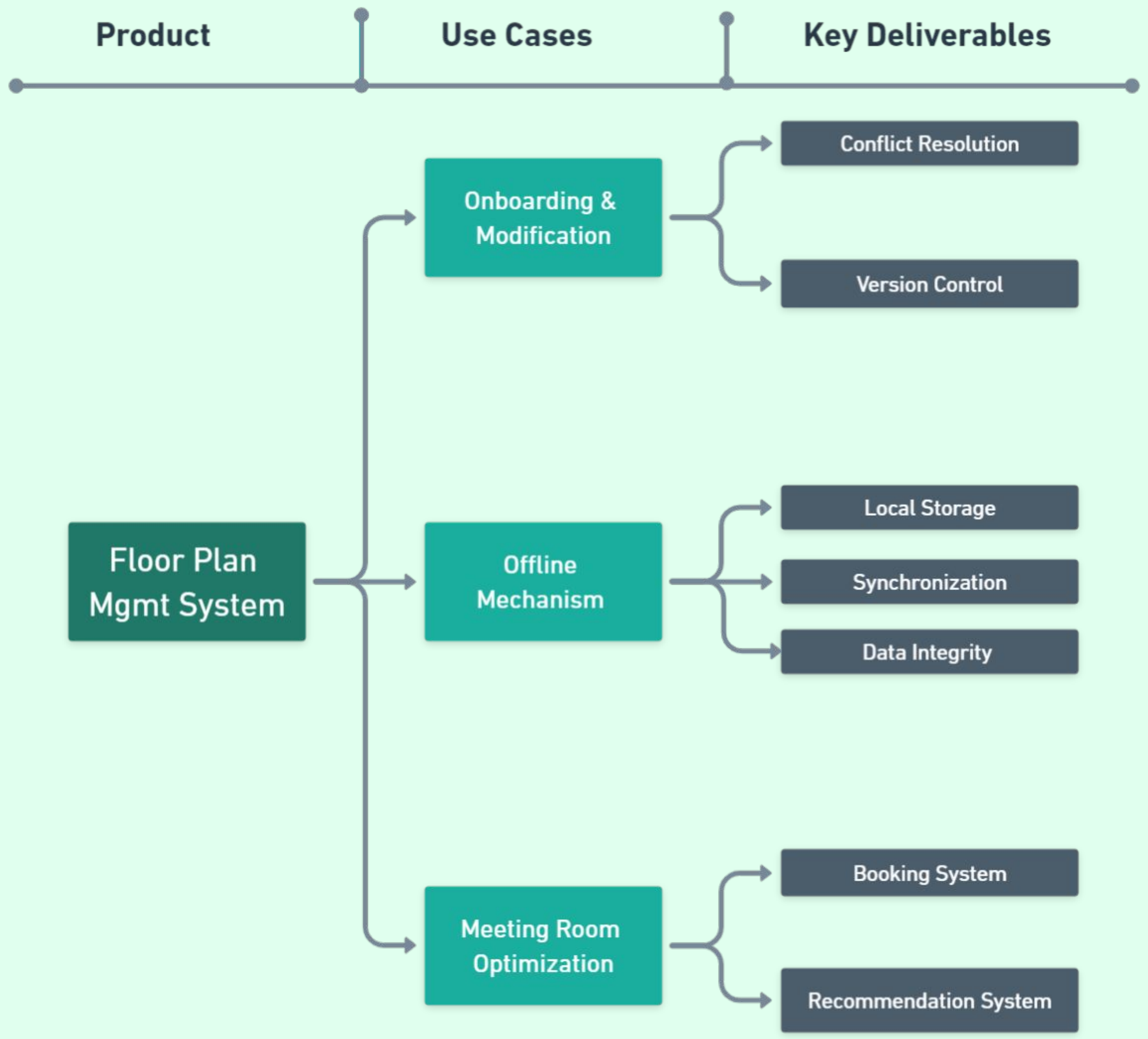
Floor Plan Management System

Akshat Awasthi
2020A8PS1790P

Introduction

The Floor Plan Management System presented herein is a **robust** solution catering to the diverse needs of administrators and users alike. The system seamlessly manages floor plans, meeting room bookings, and addresses challenges associated with conflicts during concurrent updates and offline scenarios.

The Intelligent floor plan management system offers various functionalities for the Admin and User.

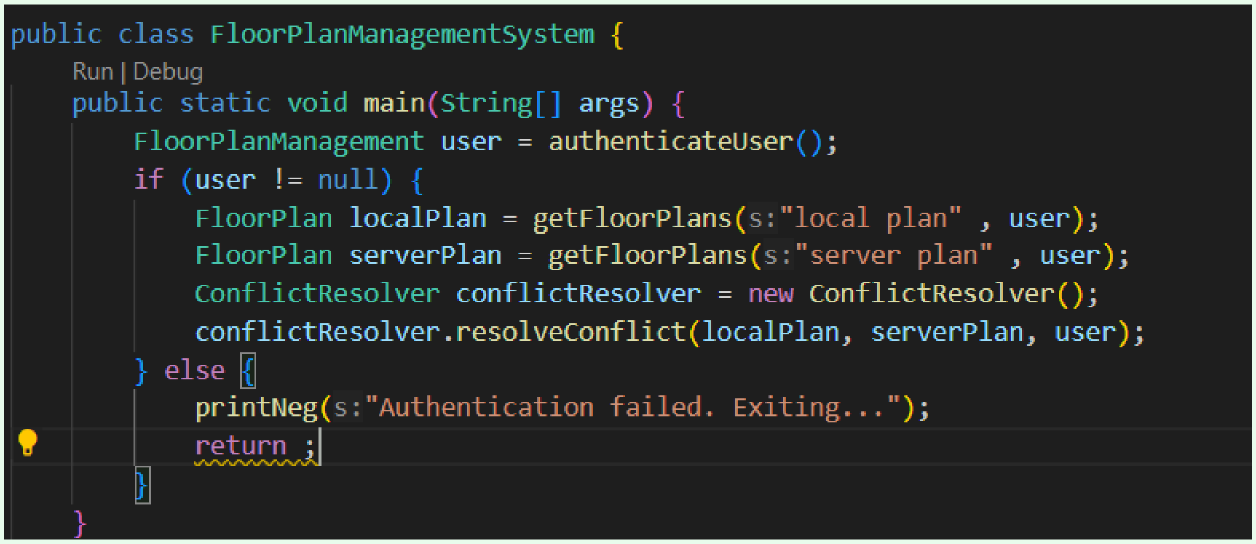


Task 1 - Admin’s Floor Plan Management

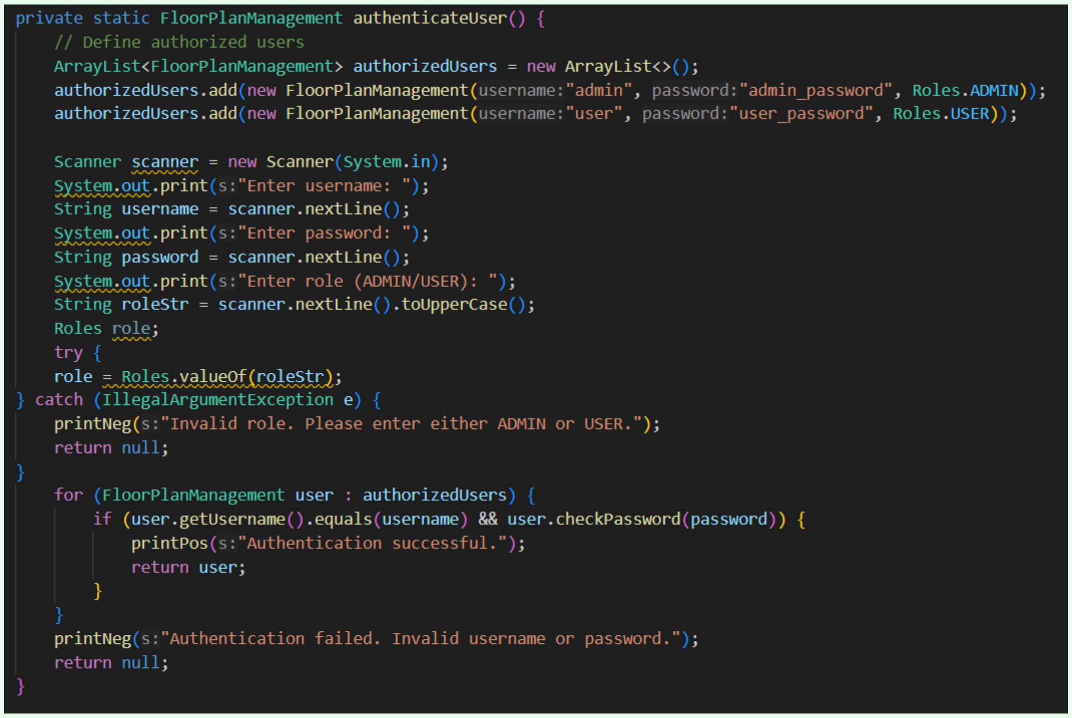
Develop features for administrators to upload floor plans, addressing potential conflicts during simultaneous updates.

The FloorPlanManagementSystem Class

- Wrote main function in FloorPlanManagementSystem.java
- User the system ask the user for their name , password and role for the user authentication

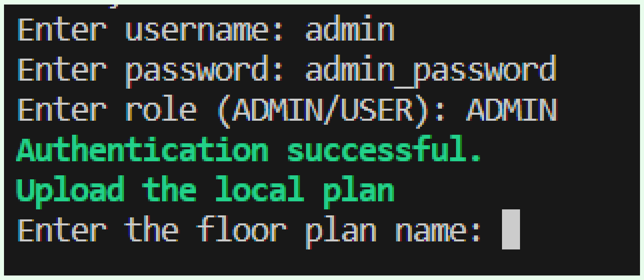


```java
public class FloorPlanManagementSystem {
    Run | Debug
    public static void main(String[] args) {
        FloorPlanManagement user = authenticateUser();
        if (user != null) {
            FloorPlan localPlan = getFloorPlans(s:"local plan" , user);
            FloorPlan serverPlan = getFloorPlans(s:"server plan" , user);
            ConflictResolver conflictResolver = new ConflictResolver();
            conflictResolver.resolveConflict(localPlan, serverPlan, user);
        } else {
            printNeg(s:"Authentication failed. Exiting...");
            return ;
        }
    }
```

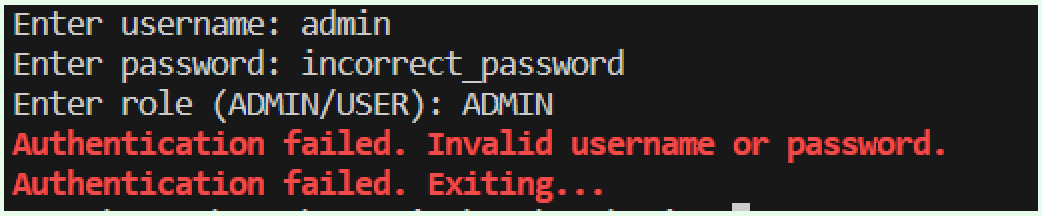


```java
private static FloorPlanManagement authenticateUser() {
    // Define authorized users
    ArrayList<FloorPlanManagement> authorizedUsers = new ArrayList<>();
    authorizedUsers.add(new FloorPlanManagement(username:"admin", password:"admin_password", Roles.ADMIN));
    authorizedUsers.add(new FloorPlanManagement(username:"user", password:"user_password", Roles.USER));

    Scanner scanner = new Scanner(System.in);
    System.out.print(s:"Enter username: ");
    String username = scanner.nextLine();
    System.out.print(s:"Enter password: ");
    String password = scanner.nextLine();
    System.out.print(s:"Enter role (ADMIN/USER): ");
    String roleStr = scanner.nextLine().toUpperCase();
    Roles role;
    try {
    role = Roles.valueOf(roleStr);
} catch (IllegalArgumentException e) {
    printNeg(s:"Invalid role. Please enter either ADMIN or USER.");
    return null;
}
    for (FloorPlanManagement user : authorizedUsers) {
        if (user.getUsername().equals(username) && user.checkPassword(password)) {
            printPos(s:"Authentication successful.");
            return user;
        }
    }
    printNeg(s:"Authentication failed. Invalid username or password.");
    return null;
}
```

- After adding the username , password and role the user is allowed to move forward if the user is authenticated otherwise we exit the program.



```
Enter username: admin
Enter password: admin_password
Enter role (ADMIN/USER): ADMIN
Authentication successful.
Upload the local plan
Enter the floor plan name: 
```

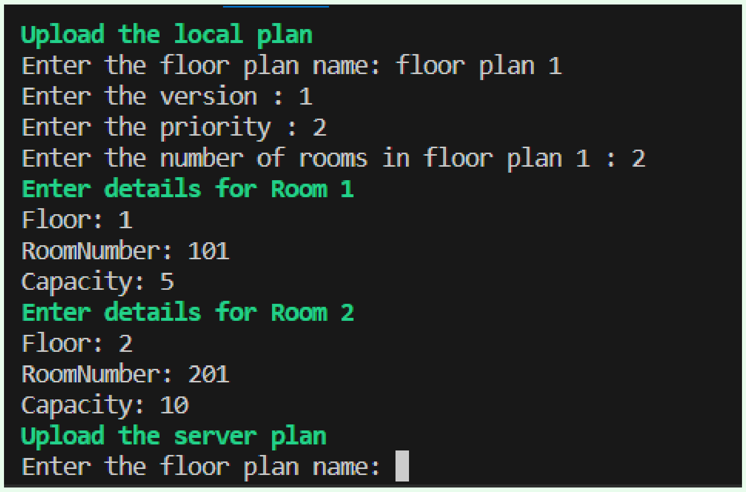


```
Enter username: admin
Enter password: incorrect_password
Enter role (ADMIN/USER): ADMIN
Authentication failed. Invalid username or password.
Authentication failed. Exiting...
```

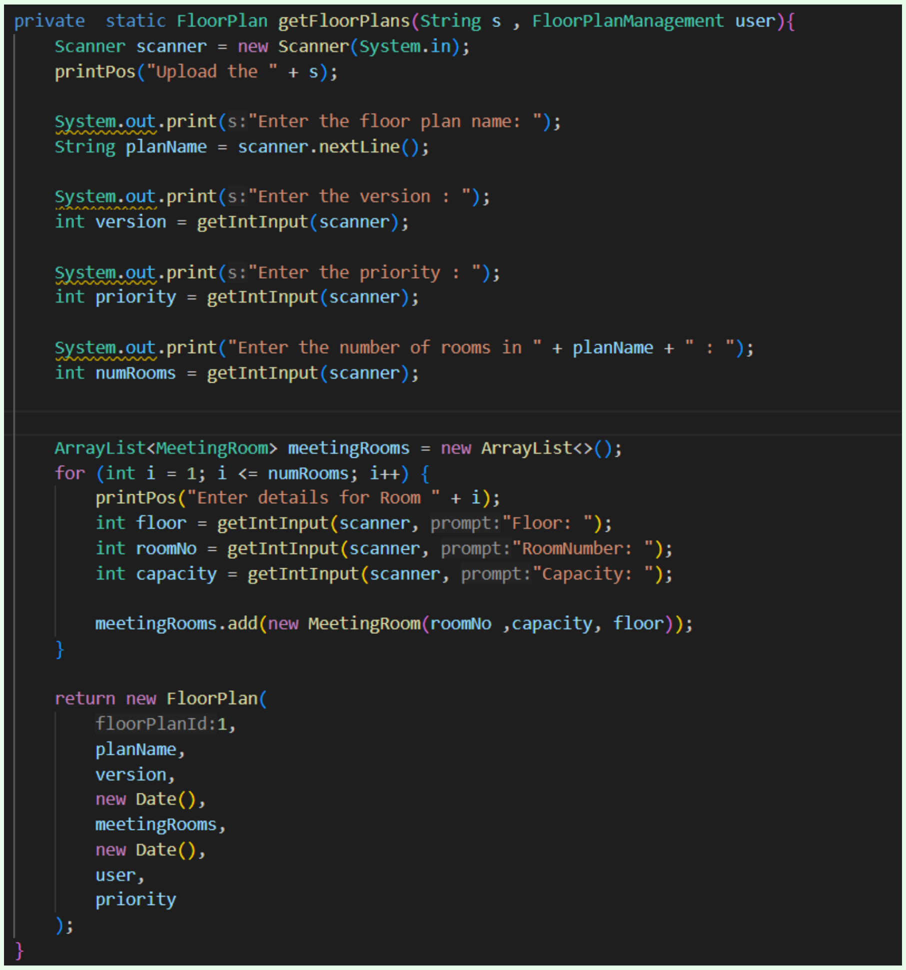
- Once user authorization is completed , the user is asked to upload the local and the server floor plan.

The user is asked to upload ,

- Name of the floor plan
- Version of the plan
- Priority of the plan
- Number of rooms in the floor plan
- Floor , RoomNo and capacity for each room



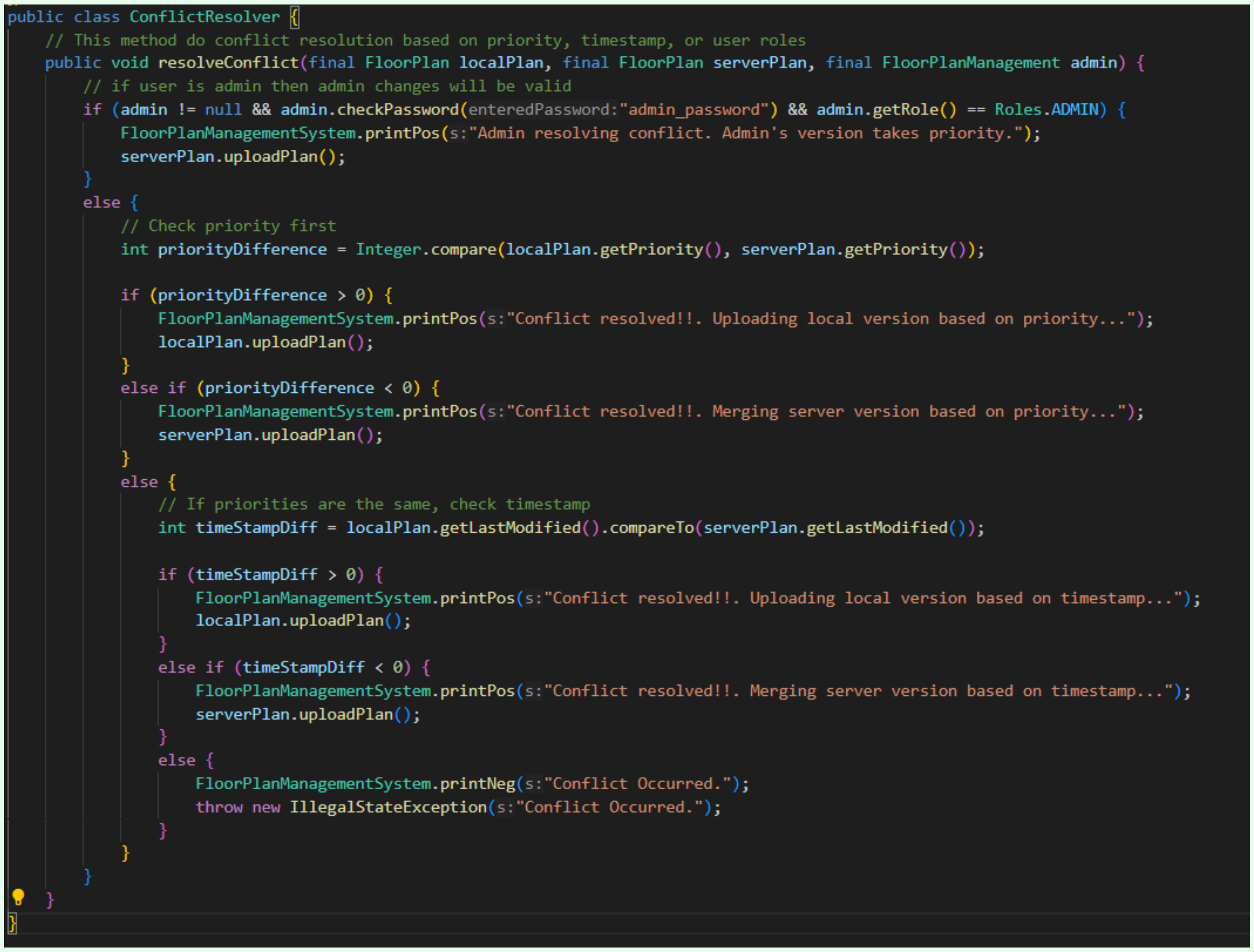
```
Upload the local plan
Enter the floor plan name: floor plan 1
Enter the version : 1
Enter the priority : 2
Enter the number of rooms in floor plan 1 : 2
Enter details for Room 1
Floor: 1
RoomNumber: 101
Capacity: 5
Enter details for Room 2
Floor: 2
RoomNumber: 201
Capacity: 10
Upload the server plan
Enter the floor plan name: 
```



```java
private  static FloorPlan getFloorPlans(String s , FloorPlanManagement user){
    Scanner scanner = new Scanner(System.in);
    printPos("Upload the " + s);

    System.out.print(s:"Enter the floor plan name: ");
    String planName = scanner.nextLine();

    System.out.print(s:"Enter the version : ");
    int version = getIntInput(scanner);

    System.out.print(s:"Enter the priority : ");
    int priority = getIntInput(scanner);

    System.out.print("Enter the number of rooms in " + planName + " : ");
    int numRooms = getIntInput(scanner);


    ArrayList<MeetingRoom> meetingRooms = new ArrayList<>();
    for (int i = 1; i <= numRooms; i++) {
        printPos("Enter details for Room " + i);
        int floor = getIntInput(scanner, prompt:"Floor: ");
        int roomNo = getIntInput(scanner, prompt:"RoomNumber: ");
        int capacity = getIntInput(scanner, prompt:"Capacity: ");

        meetingRooms.add(new MeetingRoom(roomNo ,capacity, floor));
    }

    return new FloorPlan(
        floorPlanId:1,
        planName,
        version,
        new Date(),
        meetingRooms,
        new Date(),
        user,
        priority
    );
}
```

- Once User has uploaded the local and the server plan, we call the *resolveConflict()* method in the Conflict resolver class.

The Conflict Resolver Class



```java
public class ConflictResolver {
    // This method do conflict resolution based on priority, timestamp, or user roles
    public void resolveConflict(final FloorPlan localPlan, final FloorPlan serverPlan, final FloorPlanManagement admin) {
        // if user is admin then admin changes will be valid
        if (admin != null && admin.checkPassword(enteredPassword:"admin_password") && admin.getRole() == Roles.ADMIN) {
            FloorPlanManagementSystem.printPos(s:"Admin resolving conflict. Admin's version takes priority.");
            serverPlan.uploadPlan();
        }
        else {
            // Check priority first
            int priorityDifference = Integer.compare(localPlan.getPriority(), serverPlan.getPriority());

            if (priorityDifference > 0) {
                FloorPlanManagementSystem.printPos(s:"Conflict resolved!!. Uploading local version based on priority...");
                localPlan.uploadPlan();
            }
            else if (priorityDifference < 0) {
                FloorPlanManagementSystem.printPos(s:"Conflict resolved!!. Merging server version based on priority...");
                serverPlan.uploadPlan();
            }
            else {
                // If priorities are the same, check timestamp
                int timeStampDiff = localPlan.getLastModified().compareTo(serverPlan.getLastModified());

                if (timeStampDiff > 0) {
                    FloorPlanManagementSystem.printPos(s:"Conflict resolved!!. Uploading local version based on timestamp...");
                    localPlan.uploadPlan();
                }
                else if (timeStampDiff < 0) {
                    FloorPlanManagementSystem.printPos(s:"Conflict resolved!!. Merging server version based on timestamp...");
                    serverPlan.uploadPlan();
                }
                else {
                    FloorPlanManagementSystem.printNeg(s:"Conflict Occurred.");
                    throw new IllegalStateException(s:"Conflict Occurred.");
                }
            }
        }
    }
}
```

We resolve the conflict this way ,

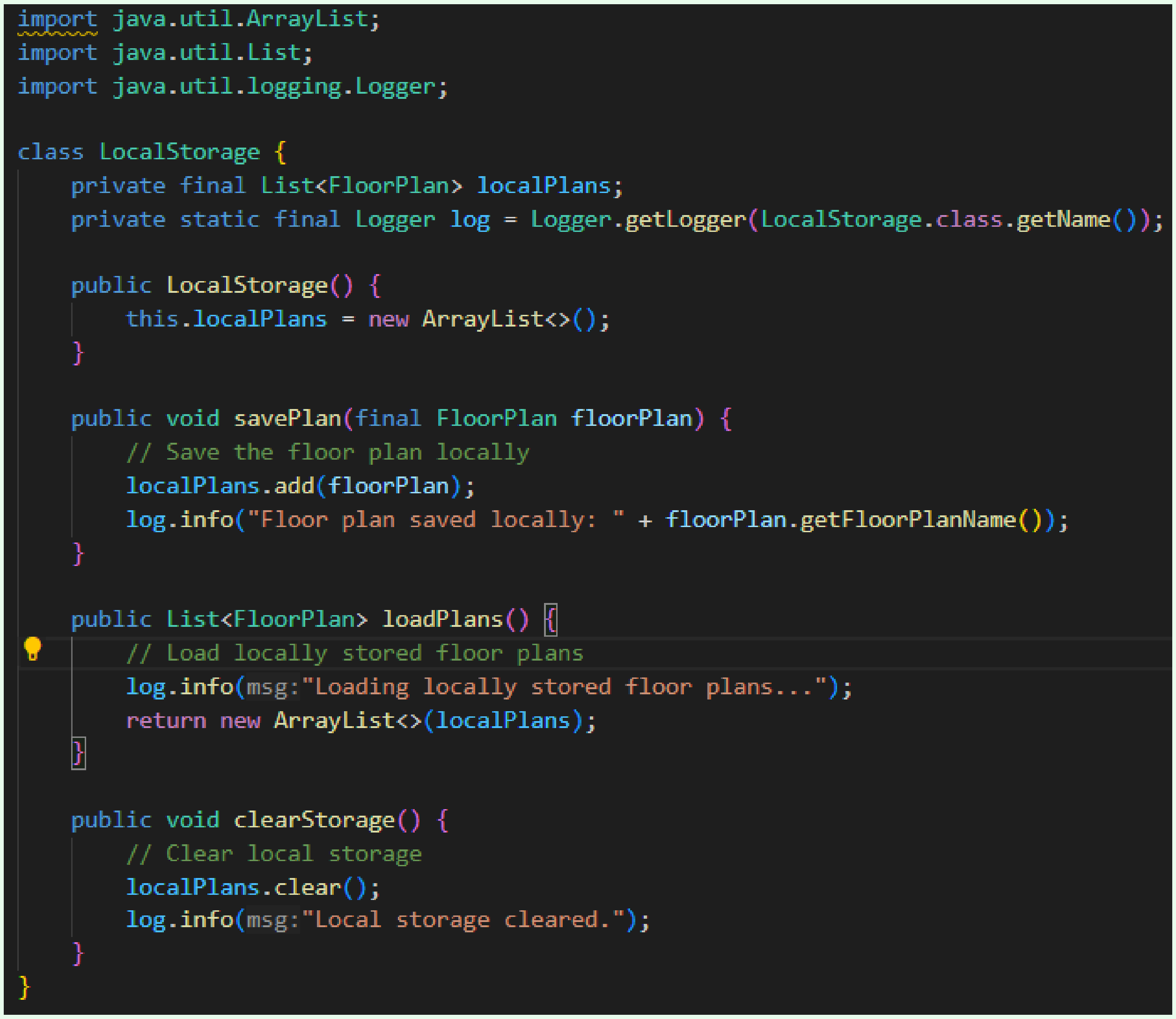
- If the admin has uploaded the server Plan then the admin version takes priority and server plan is uploaded.
- Then we judge on the basis of priority , the version with the higher priority is uploaded.
- If both the plans have the same priority then we judge on the basis of lastModifiedTimestamp, the one with the higher timestamp gets uploaded .
- If we are not able to find any difference between two plans then we throw the java IllegalStateException

Task 2 - Offline Mechanism for Admins

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

The LocalStorage Class

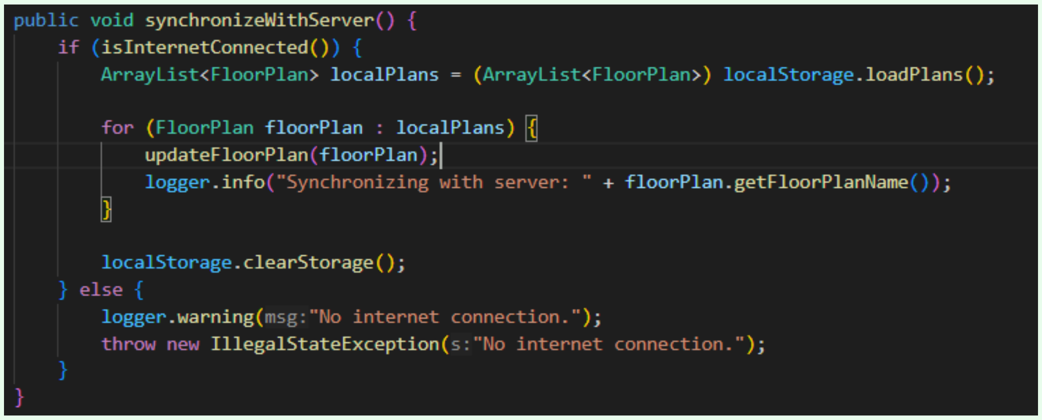
- The local storage class is an offline database created for the admin.
- It allows the admin to save a plan locally
- Allows admin to load the locally stored saved plans
- Allows admin to clean the local storage whenever needed.



```java
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Logger;

class LocalStorage {
    private final List<FloorPlan> localPlans;
    private static final Logger log = Logger.getLogger(LocalStorage.class.getName());

    public LocalStorage() {
        this.localPlans = new ArrayList<>();
    }

    public void savePlan(final FloorPlan floorPlan) {
        // Save the floor plan locally
        localPlans.add(floorPlan);
        log.info("Floor plan saved locally: " + floorPlan.getFloorPlanName());
    }

    public List<FloorPlan> loadPlans() {
        // Load locally stored floor plans
        log.info(msg:"Loading locally stored floor plans...");
        return new ArrayList<>(localPlans);
    }

    public void clearStorage() {
        // Clear local storage
        localPlans.clear();
        log.info(msg:"Local storage cleared.");
    }
}
```

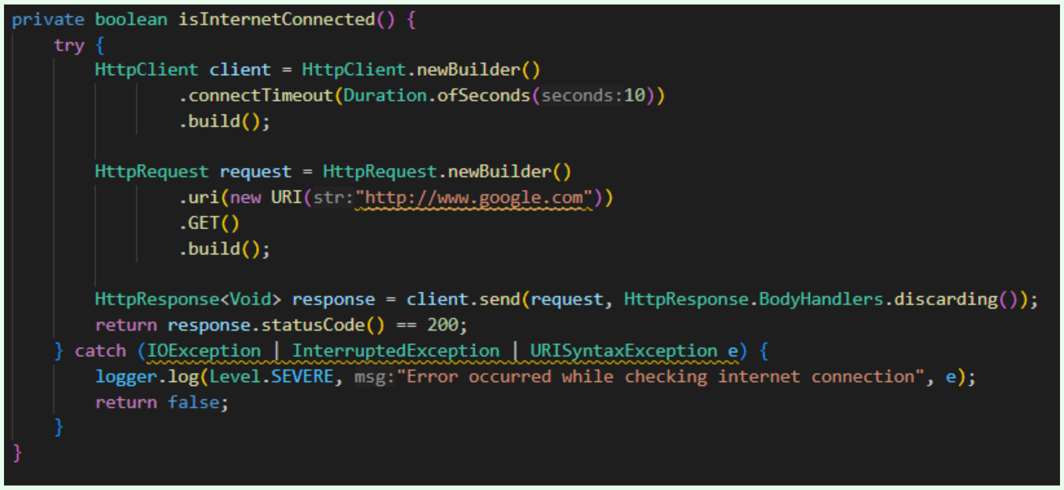
The ServerSyncer Class

- This class is responsible for synchronizing floor plans with a server in presence of internet connection with appropriate error handling.
- The method *synchronizeWithServer()*
    - Check if there is an internet connection.
    - If there is a connection, it loads local floor plans using the localStorage object.
    - Iterates through the local floor plans and calls the updateFloorPlan() method for each one.
    - Clears the local storage after synchronization.



```java
public void synchronizeWithServer() {
    if (isInternetConnected()) {
        ArrayList<FloorPlan> localPlans = (ArrayList<FloorPlan>) localStorage.loadPlans();

        for (FloorPlan floorPlan : localPlans) {
            updateFloorPlan(floorPlan);
            logger.info("Synchronizing with server: " + floorPlan.getFloorPlanName());
        }

        localStorage.clearStorage();
    } else {
        logger.warning(msg:"No internet connection.");
        throw new IllegalStateException(s:"No internet connection.");
    }
}
```

- The Method *isInternetConnected()*
    - Attempts to connect to http://www.google.com with a timeout of 10 seconds to check for internet connectivity.
    - Returns true if the HTTP response status code is 200 (indicating a successful connection), otherwise returns false.



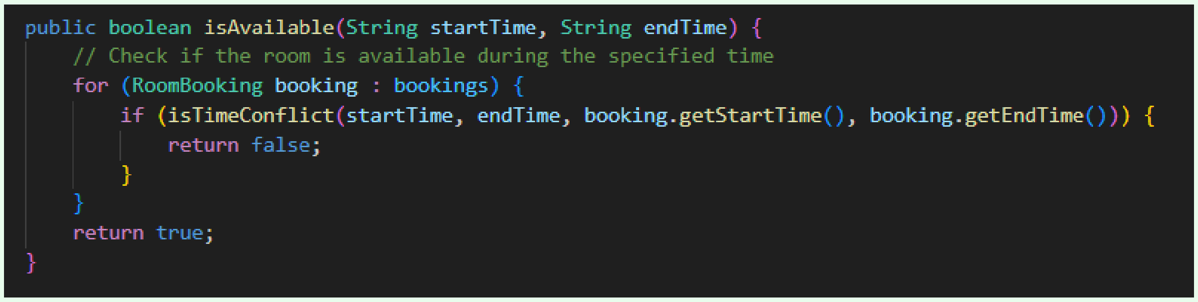
```java
private boolean isInternetConnected() {
    try {
        HttpClient client = HttpClient.newBuilder()
                .connectTimeout(Duration.ofSeconds(seconds:10))
                .build();

        HttpRequest request = HttpRequest.newBuilder()
                .uri(new URI(str:"http://www.google.com"))
                .GET()
                .build();

        HttpResponse<Void> response = client.send(request, HttpResponse.BodyHandlers.discarding());
        return response.statusCode() == 200;
    } catch (IOException | InterruptedException | URISyntaxException e) {
        logger.log(Level.SEVERE, msg:"Error occurred while checking internet connection", e);
        return false;
    }
}
```

TASK 3 - Meeting Room Optimization

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

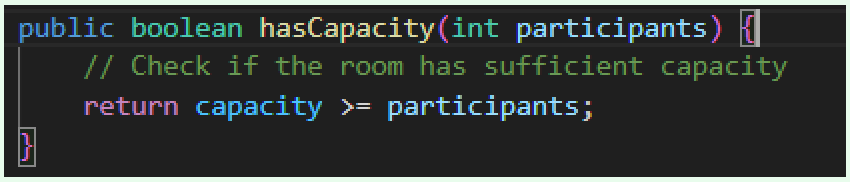
The Meeting Room Class

- The meeting room contains the *isAvailable()* method which checks if the room is available during the specified time or not.



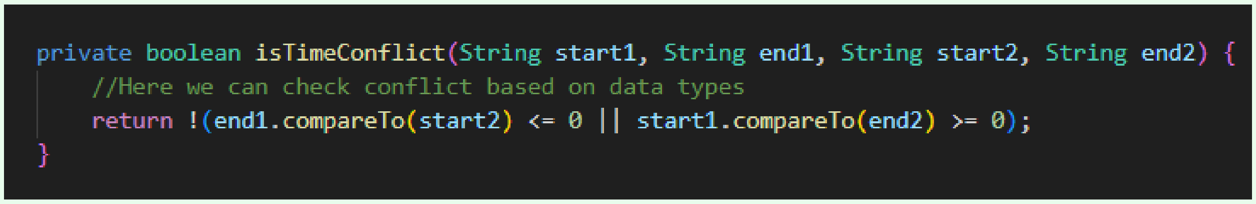
```java
public boolean isAvailable(String startTime, String endTime) {
    // Check if the room is available during the specified time
    for (RoomBooking booking : bookings) {
        if (isTimeConflict(startTime, endTime, booking.getStartTime(), booking.getEndTime())) {
            return false;
        }
    }
    return true;
}
```

- The *hasCapacity()* method checks if the room has sufficient capacity to include the number of participants.



```java
public boolean hasCapacity(int participants) {
    // Check if the room has sufficient capacity
    return capacity >= participants;
}
```

- The is*TimeConflict()* method checks for any potential conflict in the meeting time by comparing the start and end time of the meetings to be held in the room.



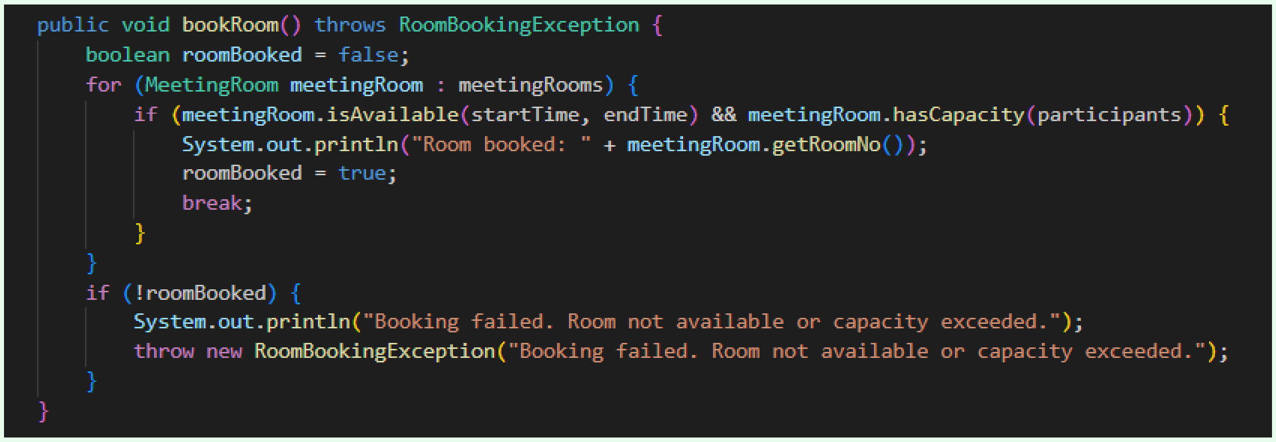
```java
private boolean isTimeConflict(String start1, String end1, String start2, String end2) {
    //Here we can check conflict based on data types
    return !(end1.compareTo(start2) <= 0 || start1.compareTo(end2) >= 0);
}
```

The Room Booking Class

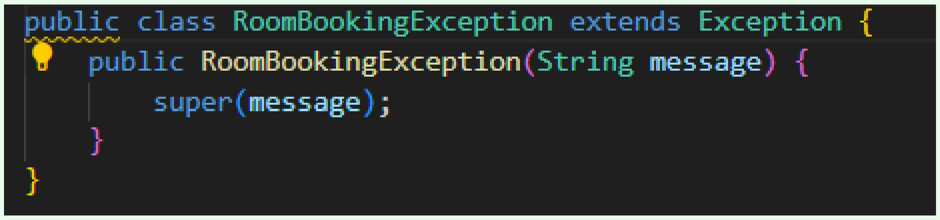
This class encapsulates the functionality related to booking meeting rooms, including checking availability and capacity constraints. It follows the principle of encapsulation by hiding the internal details of room booking and providing methods to interact with its data in a controlled manner.

bookRoom Method():

- This method attempts to book a room for the specified time period and number of participants.
- It iterates over the list of meeting rooms to find an available room that can accommodate the required number of participants within the specified time frame.
- If a suitable room is found, it prints a confirmation message and sets the roomBooked flag to true.
- If no suitable room is found, it prints a failure message and throws a *RoomBookingException*.



```java
public void bookRoom() throws RoomBookingException {
    boolean roomBooked = false;
    for (MeetingRoom meetingRoom : meetingRooms) {
        if (meetingRoom.isAvailable(startTime, endTime) && meetingRoom.hasCapacity(participants)) {
            System.out.println("Room booked: " + meetingRoom.getRoomNo());
            roomBooked = true;
            break;
        }
    }
    if (!roomBooked) {
        System.out.println("Booking failed. Room not available or capacity exceeded.");
        throw new RoomBookingException("Booking failed. Room not available or capacity exceeded.");
    }
}
```



```java
public class RoomBookingException extends Exception {
    public RoomBookingException(String message) {
        super(message);
    }
}
```