Intelligent Floor Plan Management System for a Seamless Experience

By-

Sai Uma Jayanth Badisa

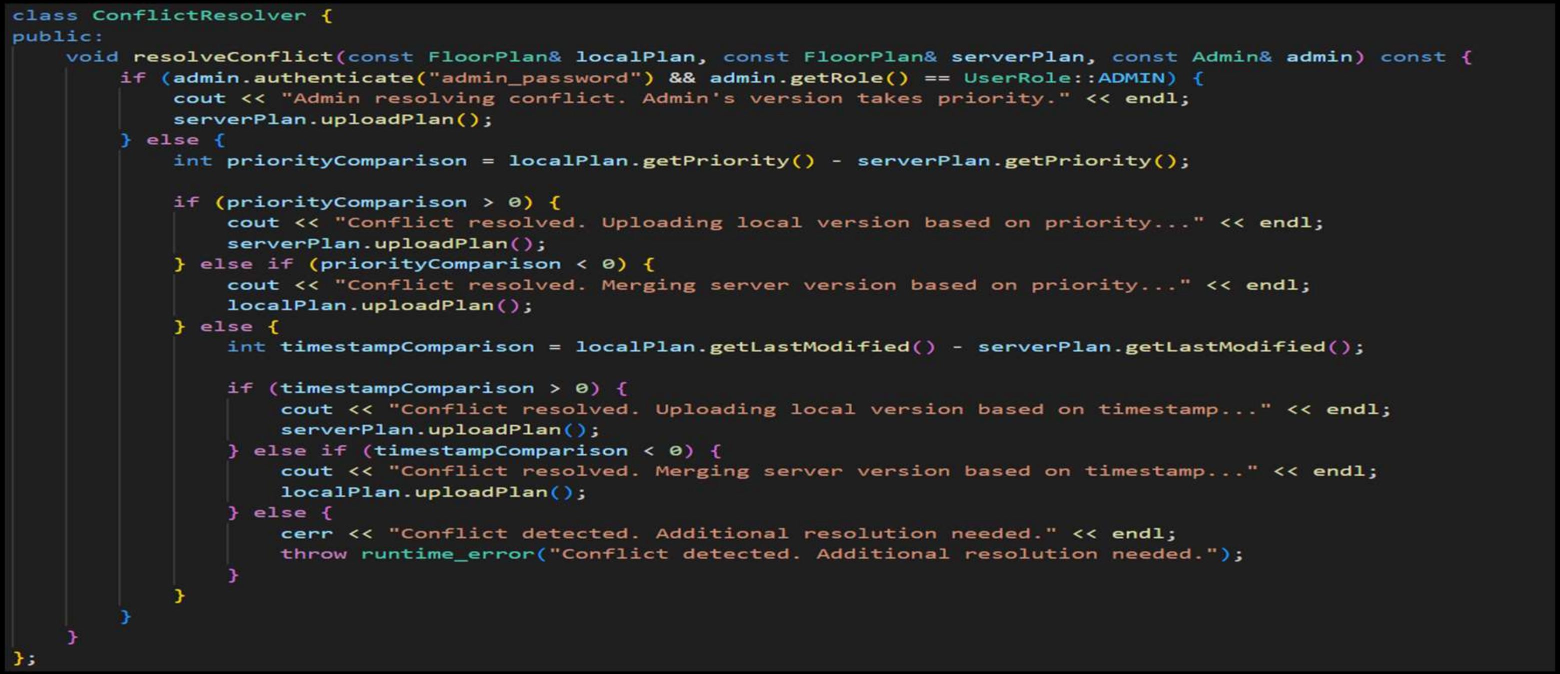
programming Language - C++

INTRODUCTION

- Our project focuses on three key objectives. Firstly, in **Admin's Floor Plan Management**, we aim to provide administrators with seamless floor plan uploads while intelligently resolving conflicts during simultaneous updates. Secondly, the **Offline Mechanism for Admins** ensures uninterrupted productivity by allowing administrators to make floor plan modifications offline and sync them when connectivity is restored. Lastly, under **Meeting Room Optimization**, we enhance the system's intelligence in suggesting and booking meeting rooms based on factors like capacity and availability. These objectives collectively contribute to an adaptive, collaborative, and efficient workspace experience.

OBJ-1: Admin’s Floor Plan Management

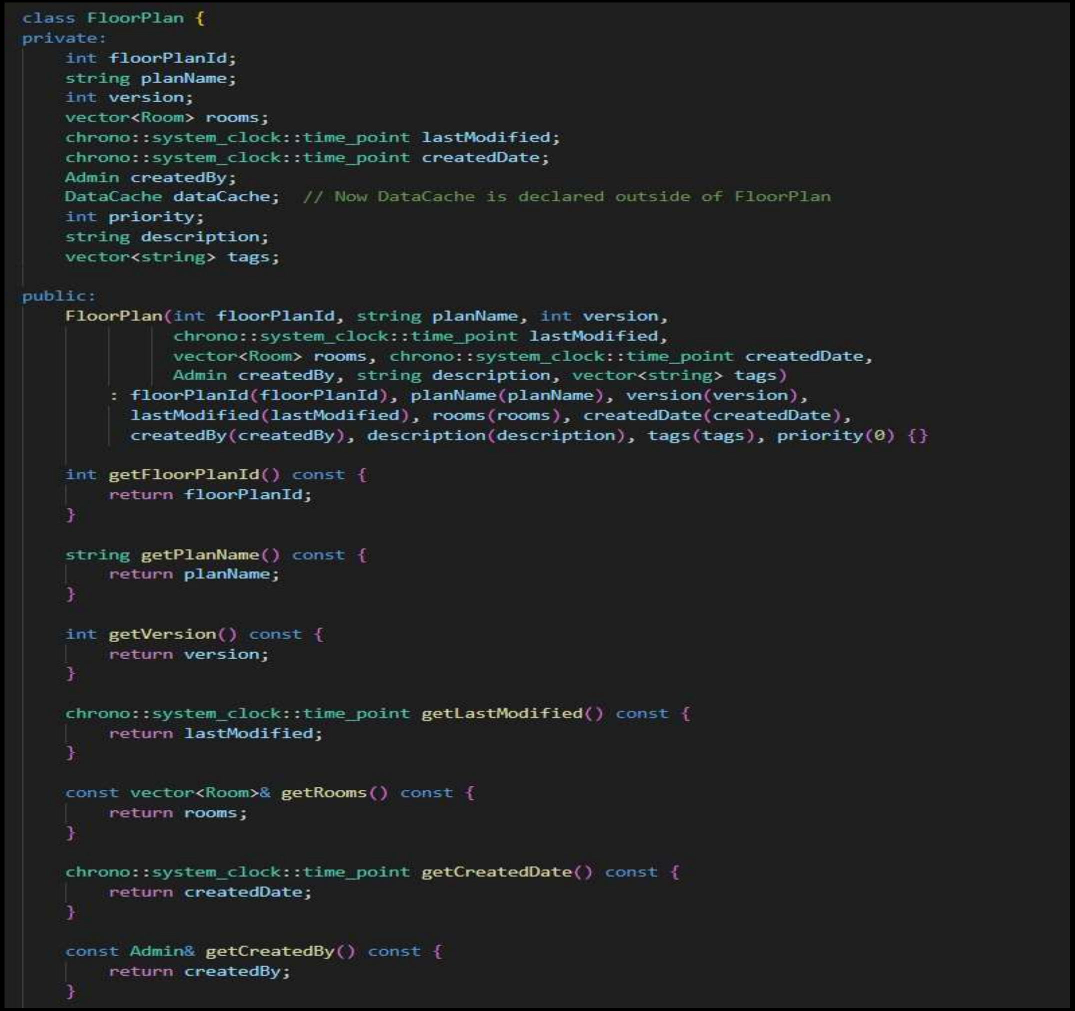
- Step-1: Conflict Resolution.



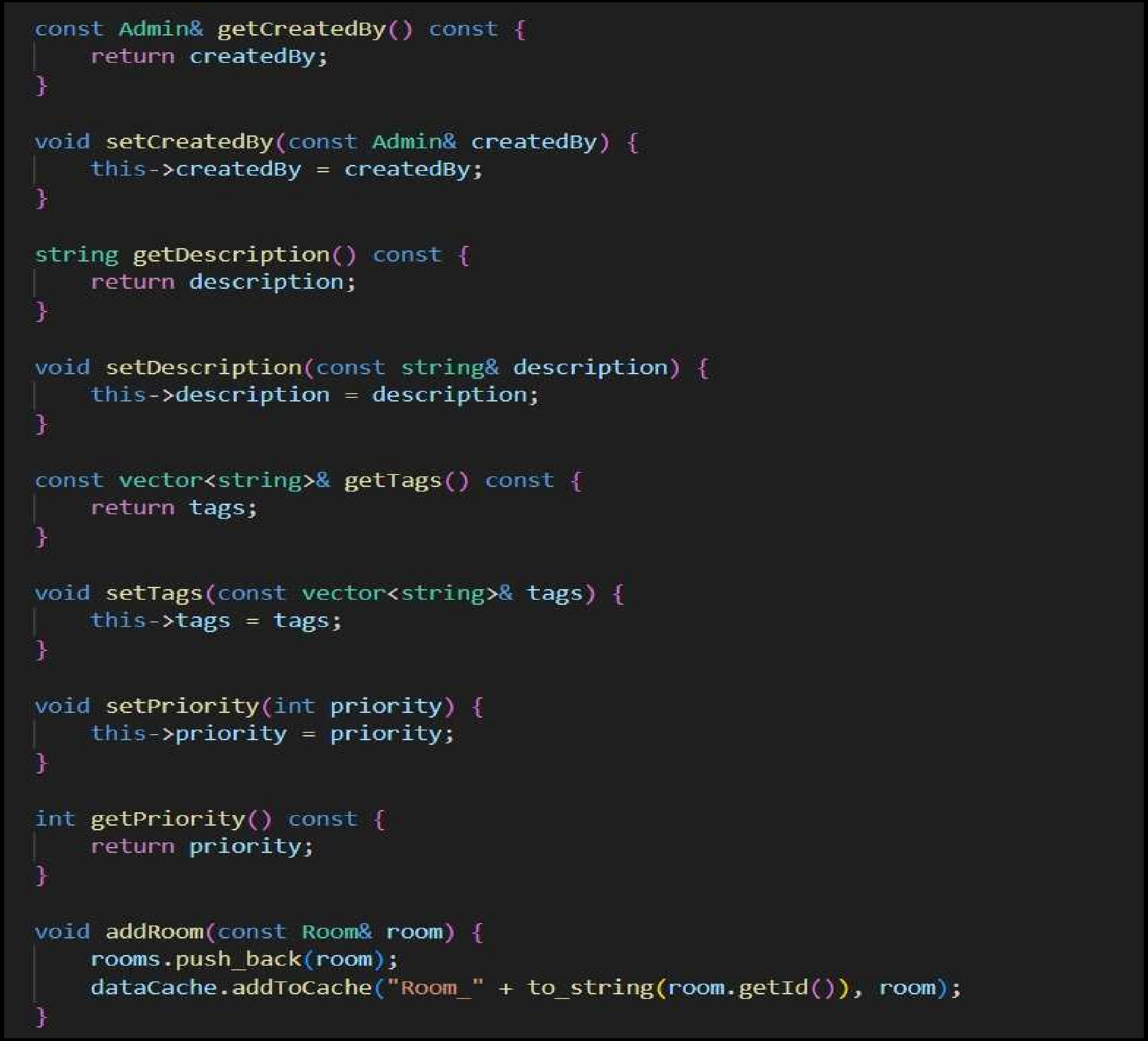
```
class ConflictResolver {
public:
    void resolveConflict(const FloorPlan& localPlan, const FloorPlan& serverPlan, const Admin& admin) const {
        if (admin.authenticate("admin_password") && admin.getRole() == UserRole::ADMIN) {
            cout << "Admin resolving conflict. Admin's version takes priority." << endl;
            serverPlan.uploadPlan();
        } else {
            int priorityComparison = localPlan.getPriority() - serverPlan.getPriority();

            if (priorityComparison > 0) {
                cout << "Conflict resolved. Uploading local version based on priority..." << endl;
                serverPlan.uploadPlan();
            } else if (priorityComparison < 0) {
                cout << "Conflict resolved. Merging server version based on priority..." << endl;
                localPlan.uploadPlan();
            } else {
                int timestampComparison = localPlan.getLastModified() - serverPlan.getLastModified();

                if (timestampComparison > 0) {
                    cout << "Conflict resolved. Uploading local version based on timestamp..." << endl;
                    serverPlan.uploadPlan();
                } else if (timestampComparison < 0) {
                    cout << "Conflict resolved. Merging server version based on timestamp..." << endl;
                    localPlan.uploadPlan();
                } else {
                    cerr << "Conflict detected. Additional resolution needed." << endl;
                    throw runtime_error("Conflict detected. Additional resolution needed.");
                }
            }
        }
    }
};
```

resolveConflict() method Prioritizes Updates intelligently based on factors such as User roles, Priority & TimeStamp. We can understand the Hierarchy of priorities from the above code.

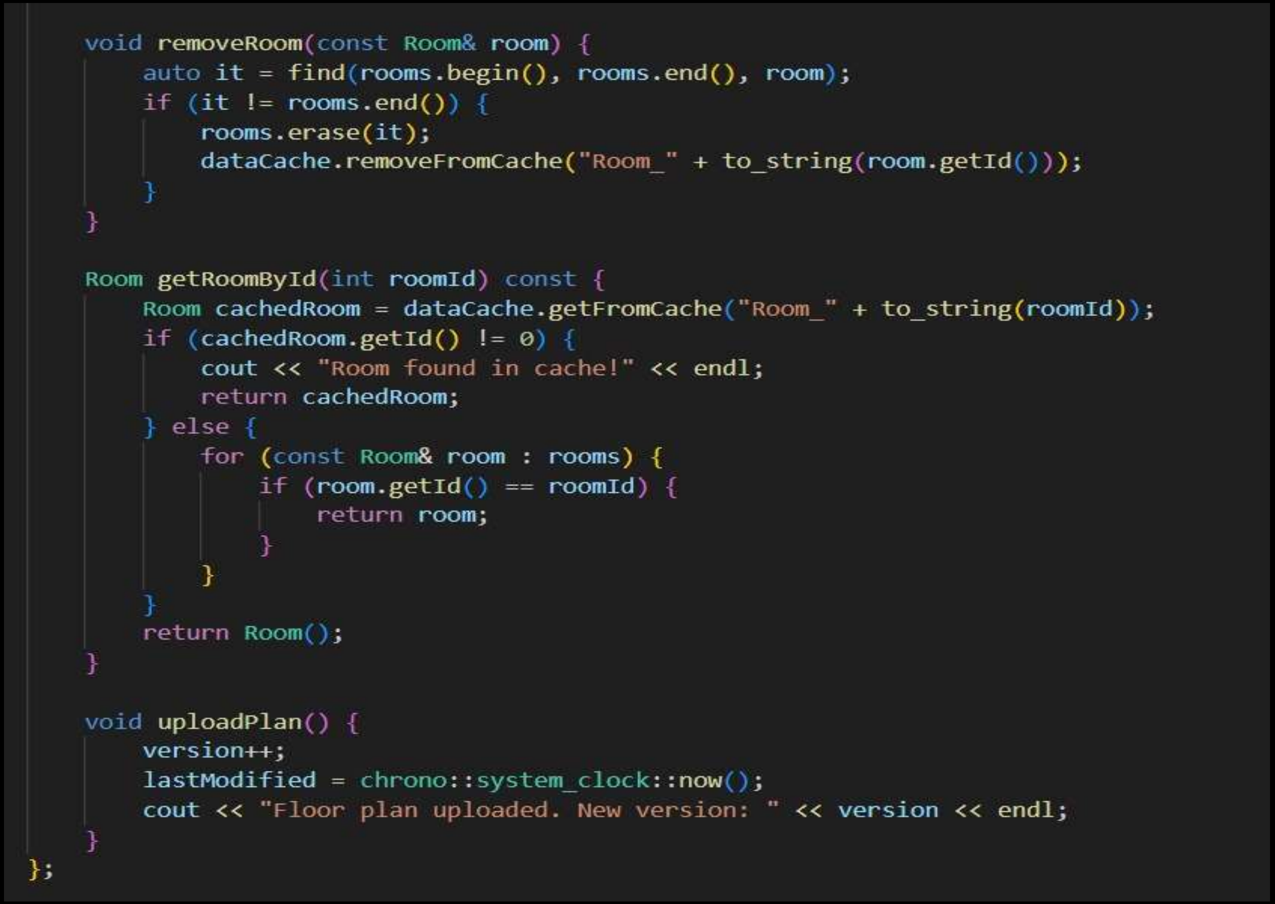
- Step-2: Version Control System.



```
class FloorPlan {
private:
    int floorPlanId;
    string planName;
    int version;
    vector<Room> rooms;
    chrono::system_clock::time_point lastModified;
    chrono::system_clock::time_point createdDate;
    Admin createdBy;
    DataCache dataCache;  // Now DataCache is declared outside of FloorPlan
    int priority;
    string description;
    vector<string> tags;

public:
    FloorPlan(int floorPlanId, string planName, int version,
              chrono::system_clock::time_point lastModified,
              vector<Room> rooms, chrono::system_clock::time_point createdDate,
              Admin createdBy, string description, vector<string> tags)
        : floorPlanId(floorPlanId), planName(planName), version(version),
          lastModified(lastModified), rooms(rooms), createdDate(createdDate),
          createdBy(createdBy), description(description), tags(tags), priority(0) {}

    int getFloorPlanId() const {
        return floorPlanId;
    }

    string getPlanName() const {
        return planName;
    }

    int getVersion() const {
        return version;
    }

    chrono::system_clock::time_point getLastModified() const {
        return lastModified;
    }

    const vector<Room>& getRooms() const {
        return rooms;
    }

    chrono::system_clock::time_point getCreatedDate() const {
        return createdDate;
    }

    const Admin& getCreatedBy() const {
        return createdBy;
    }
```



```cpp
const Admin& getCreatedBy() const {
    return createdBy;
}

void setCreatedBy(const Admin& createdBy) {
    this->createdBy = createdBy;
}

string getDescription() const {
    return description;
}

void setDescription(const string& description) {
    this->description = description;
}

const vector<string>& getTags() const {
    return tags;
}

void setTags(const vector<string>& tags) {
    this->tags = tags;
}

void setPriority(int priority) {
    this->priority = priority;
}

int getPriority() const {
    return priority;
}

void addRoom(const Room& room) {
    rooms.push_back(room);
    dataCache.addToCache("Room_" + to_string(room.getId()), room);
}
```



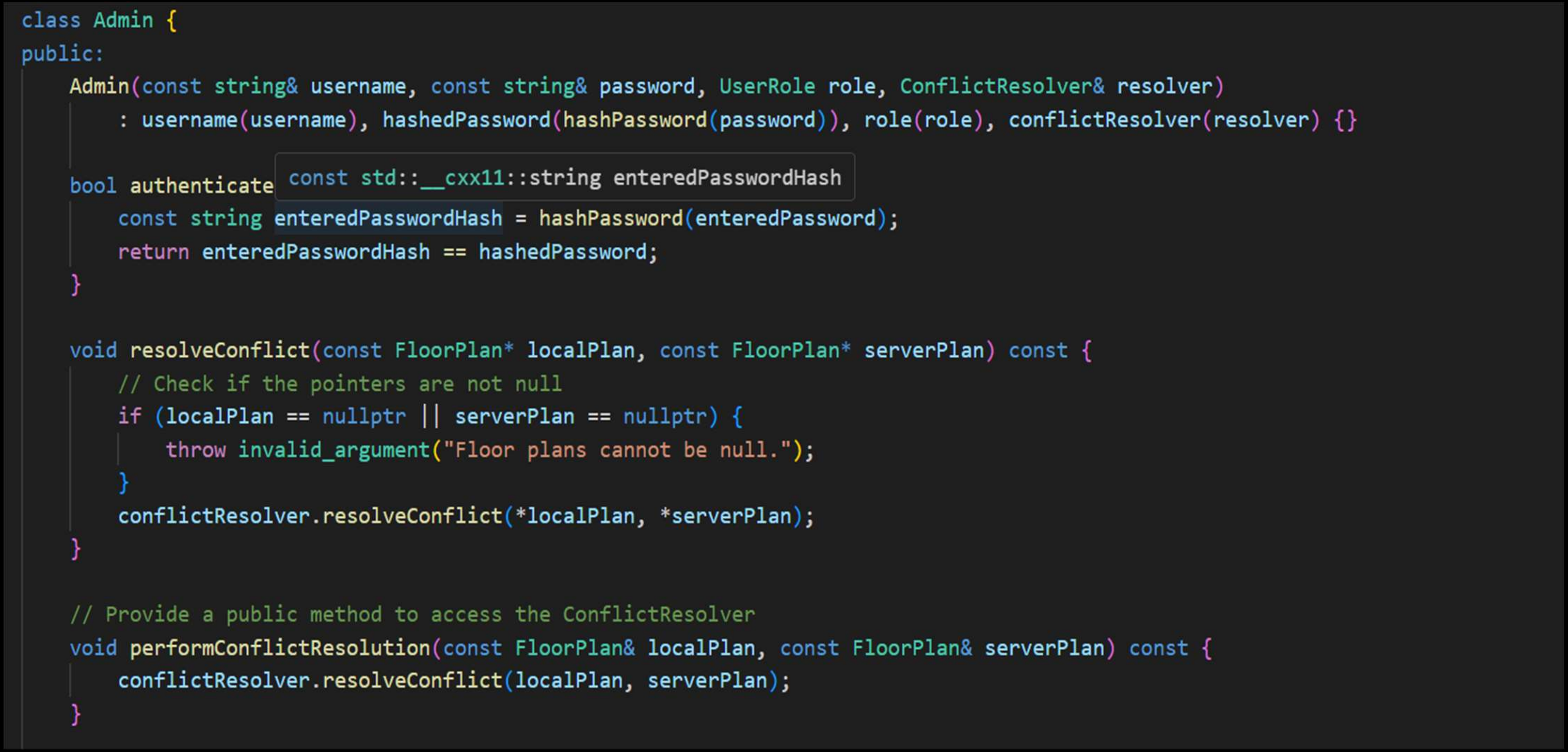
```
    void removeRoom(const Room& room) {
        auto it = find(rooms.begin(), rooms.end(), room);
        if (it != rooms.end()) {
            rooms.erase(it);
            dataCache.removeFromCache("Room_" + to_string(room.getId()));
        }
    }

    Room getRoomById(int roomId) const {
        Room cachedRoom = dataCache.getFromCache("Room_" + to_string(roomId));
        if (cachedRoom.getId() != 0) {
            cout << "Room found in cache!" << endl;
            return cachedRoom;
        } else {
            for (const Room& room : rooms) {
                if (room.getId() == roomId) {
                    return room;
                }
            }
        }
        return Room();
    }

    void uploadPlan() {
        version++;
        lastModified = chrono::system_clock::now();
        cout << "Floor plan uploaded. New version: " << version << endl;
    }
};
```

Class-Name: FloorPlan.

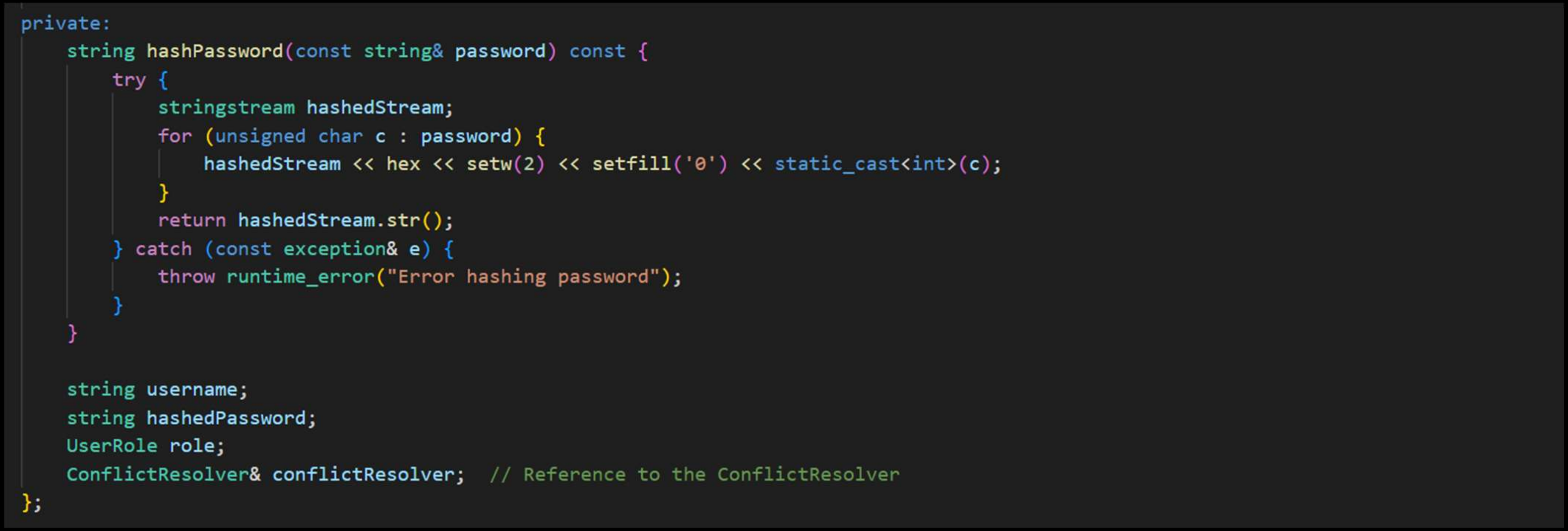
Attributes : 'Version' - Represents the iteration of changes.
'lastModified' - Keeps track of last modification timestamp.

Method : 'uploadPlan' - Increments the version and updates the last modification timestamp upon uploading the floor plan.

Step-3: Admin Roles



```
class Admin {
public:
    Admin(const string& username, const string& password, UserRole role, ConflictResolver& resolver)
        : username(username), hashedPassword(hashPassword(password)), role(role), conflictResolver(resolver) {}

    bool authenticate const std::__cxx11::string enteredPasswordHash
        const string enteredPasswordHash = hashPassword(enteredPassword);
        return enteredPasswordHash == hashedPassword;
    }

    void resolveConflict(const FloorPlan* localPlan, const FloorPlan* serverPlan) const {
        // Check if the pointers are not null
        if (localPlan == nullptr || serverPlan == nullptr) {
            throw invalid_argument("Floor plans cannot be null.");
        }
        conflictResolver.resolveConflict(*localPlan, *serverPlan);
    }

    // Provide a public method to access the ConflictResolver
    void performConflictResolution(const FloorPlan& localPlan, const FloorPlan& serverPlan) const {
        conflictResolver.resolveConflict(localPlan, serverPlan);
    }
```

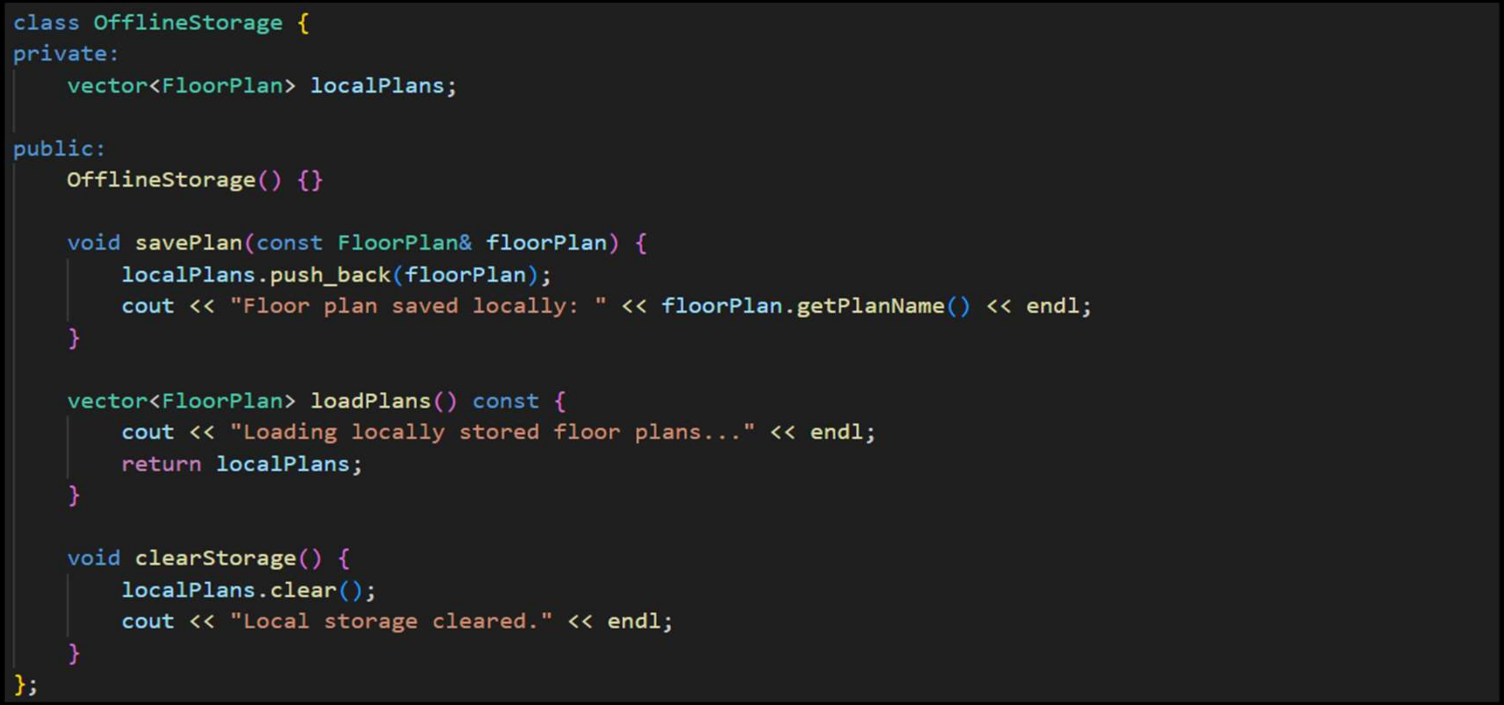


```cpp
private:
    string hashPassword(const string& password) const {
        try {
            stringstream hashedStream;
            for (unsigned char c : password) {
                hashedStream << hex << setw(2) << setfill('0') << static_cast<int>(c);
            }
            return hashedStream.str();
        } catch (const exception& e) {
            throw runtime_error("Error hashing password");
        }
    }

    string username;
    string hashedPassword;
    UserRole role;
    ConflictResolver& conflictResolver;  // Reference to the ConflictResolver
};
```

Class-Name: Admin
Attributes : ‘username’, ‘role’, ‘conflictResolver, ‘hashedPassword’.
Methods : ‘authenticate()’ – Verifies whether the password after the hashing matches the original password.
‘hashPassword()’ .
‘resolveConflict()’.

OBJ-2: Offline Mechanism for Admins.

Step-1: Local Storage System



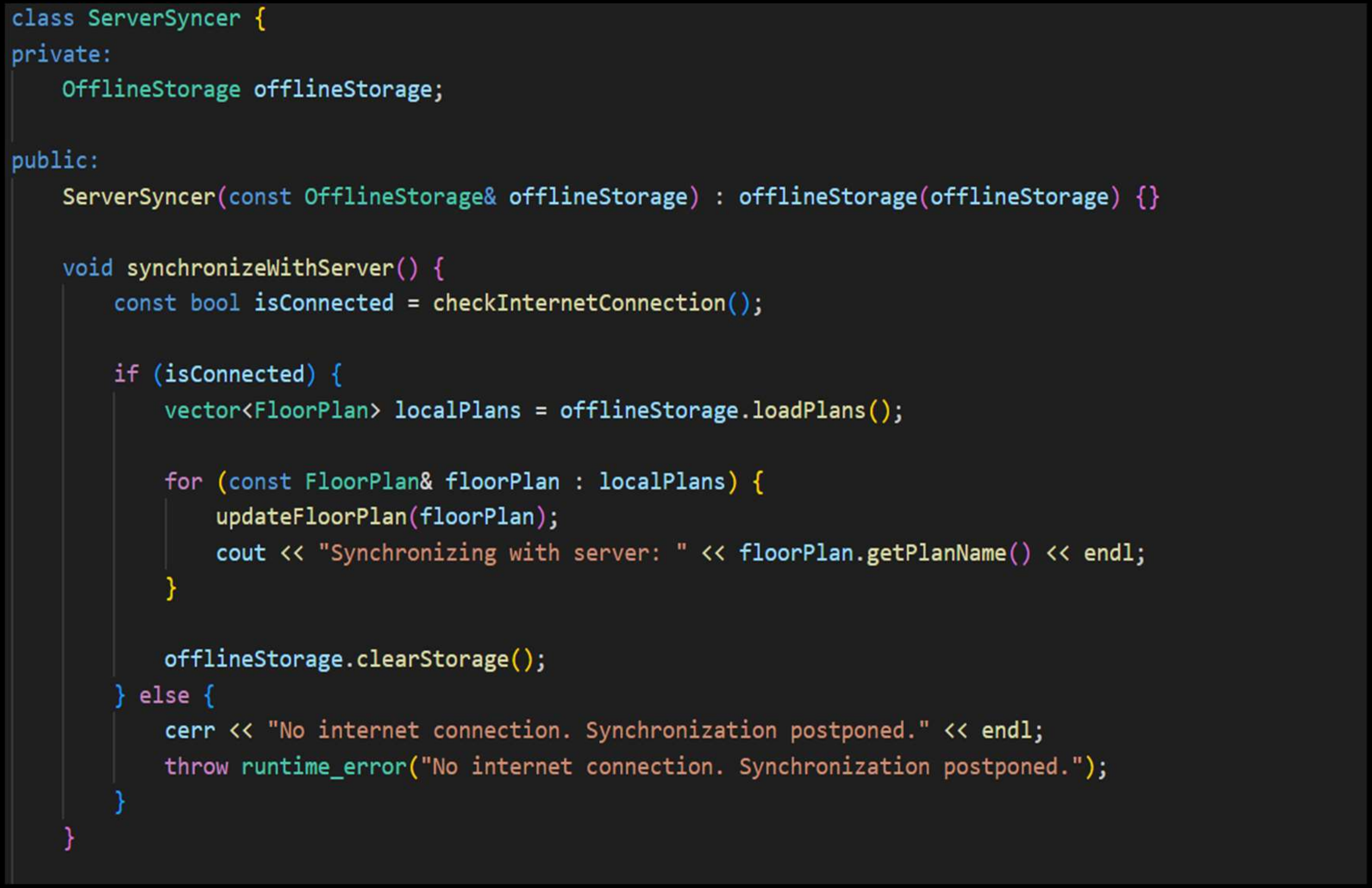
```cpp
class OfflineStorage {
private:
    vector<FloorPlan> localPlans;

public:
    OfflineStorage() {}

    void savePlan(const FloorPlan& floorPlan) {
        localPlans.push_back(floorPlan);
        cout << "Floor plan saved locally: " << floorPlan.getPlanName() << endl;
    }

    vector<FloorPlan> loadPlans() const {
        cout << "Loading locally stored floor plans..." << endl;
        return localPlans;
    }

    void clearStorage() {
        localPlans.clear();
        cout << "Local storage cleared." << endl;
    }
};
```

Class-Name: OfflineStorage

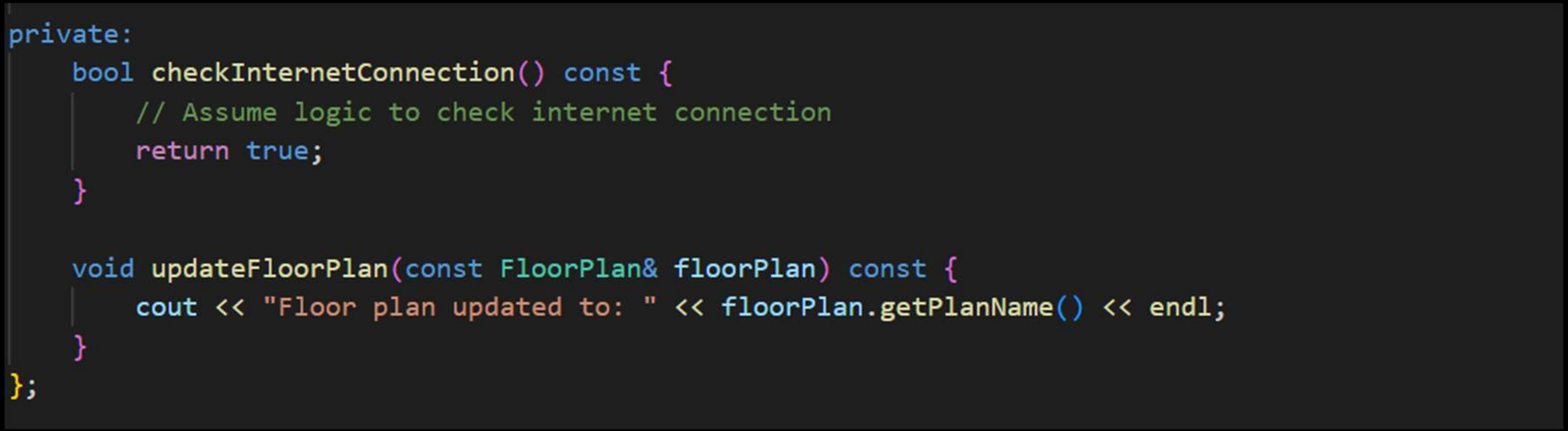
Upon going offline – ‘savePlan()’ method saves the floor plans locally.

- ‘loadPlans()’ method loads locally stored plans.
- ‘clearStorage()’ method clears local storage.

Step-2 : Server Synchronization



```cpp
class ServerSyncer {
private:
    OfflineStorage offlineStorage;

public:
    ServerSyncer(const OfflineStorage& offlineStorage) : offlineStorage(offlineStorage) {}

    void synchronizeWithServer() {
        const bool isConnected = checkInternetConnection();

        if (isConnected) {
            vector<FloorPlan> localPlans = offlineStorage.loadPlans();

            for (const FloorPlan& floorPlan : localPlans) {
                updateFloorPlan(floorPlan);
                cout << "Synchronizing with server: " << floorPlan.getPlanName() << endl;
            }

            offlineStorage.clearStorage();
        } else {
            cerr << "No internet connection. Synchronization postponed." << endl;
            throw runtime_error("No internet connection. Synchronization postponed.");
        }
    }
```



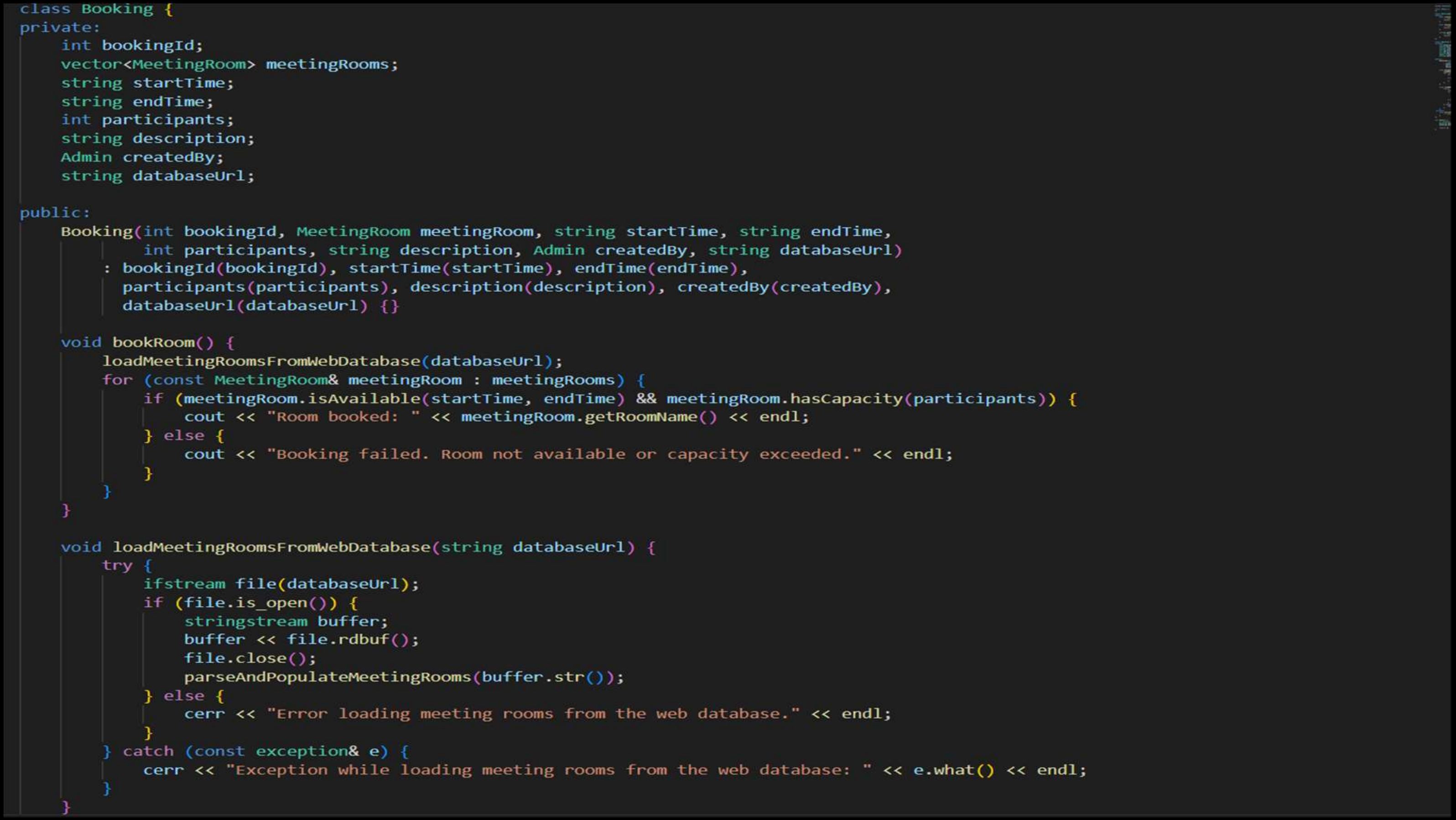
```cpp
private:
    bool checkInternetConnection() const {
        // Assume logic to check internet connection
        return true;
    }

    void updateFloorPlan(const FloorPlan& floorPlan) const {
        cout << "Floor plan updated to: " << floorPlan.getPlanName() << endl;
    }
};
```

Class-Name : 'ServerSyncer'

Methods : 'SynchronizeWithServer()' – On offline mode, checks the connection and loads plans to local storage and synchronizes with server after going online. Clears the local storage after successful synchronization.

OBJ-3: Meeting Room Optimization

Booking System & Meeting Room



```cpp
class Booking {
private:
    int bookingId;
    vector<MeetingRoom> meetingRooms;
    string startTime;
    string endTime;
    int participants;
    string description;
    Admin createdBy;
    string databaseUrl;

public:
    Booking(int bookingId, MeetingRoom meetingRoom, string startTime, string endTime,
            int participants, string description, Admin createdBy, string databaseUrl)
        : bookingId(bookingId), startTime(startTime), endTime(endTime),
          participants(participants), description(description), createdBy(createdBy),
          databaseUrl(databaseUrl) {}

    void bookRoom() {
        loadMeetingRoomsFromWebDatabase(databaseUrl);
        for (const MeetingRoom& meetingRoom : meetingRooms) {
            if (meetingRoom.isAvailable(startTime, endTime) && meetingRoom.hasCapacity(participants)) {
                cout << "Room booked: " << meetingRoom.getRoomName() << endl;
            } else {
                cout << "Booking failed. Room not available or capacity exceeded." << endl;
            }
        }
    }

    void loadMeetingRoomsFromWebDatabase(string databaseUrl) {
        try {
            ifstream file(databaseUrl);
            if (file.is_open()) {
                stringstream buffer;
                buffer << file.rdbuf();
                file.close();
                parseAndPopulateMeetingRooms(buffer.str());
            } else {
                cerr << "Error loading meeting rooms from the web database." << endl;
            }
        } catch (const exception& e) {
            cerr << "Exception while loading meeting rooms from the web database: " << e.what() << endl;
        }
    }
```

Class-Name: Booking
-It contains attributes ‘bookingId, ‘startTime’, ‘endTime’, ‘participants’, ‘description’, ‘createdBy’, ‘databaseUrl’.
- bookRoom() method iterates through the list meeting rooms and books the one that is optimally efficient.
- loadMeetingsFromWebDatabase() method creates HttpUrl connection and gets the meeting room information.

Class-Name: ‘MeetingRoom’
-It contains attributes ‘roomId’, ‘roomName’, ‘capacity’, ‘location’, ‘bookings’.
-isAvailable() checks room availability.
-hasCapacity() checks capacity.
-isTimeConflict() checks time conflict.
-addBooking() adds booking.
-getRoomNumber() gets room number.

KEY-POINTS:

- The above project implements a secure process to upload and modify plans.
- It implements optimal algorithms to handle conflict issues and merging problems.
- Handling System Failures is a key aspect, for example upon going offline while uploading or modifying plans, it stores the data in a local storage and synchronizes with the server upon going online.
- In all these functions C++ language (an OOPS language) is efficiently used taking encapsulation and extensibility as key aspects.
- Caching is used in modification of floor plans while considering the important variables in order to optimize merging function.
- Error Updates are issued on to the screen as a part of error/exception handling in cases like trying to access non existing meeting rooms.