Bus Booking System

Important Links:

https://drive.google.com/drive/folders/1Nu3ucusTv1fG4zydz_rIVIxJ9YM-fYZm?usp=sharing

For the video and uml diagram

Technologies and Frameworks

Java with Spring Boot:

Robust Server-Side Implementation: Java is known for its reliability and stability. Spring Boot, a Java-based framework, provides a powerful and efficient way to build robust server-side applications. It simplifies configuration, reduces boilerplate code, and offers features like dependency injection, making it well-suited for developing scalable backend services.

Database (e.g., MySQL):

Persistent Data Storage: MySQL, a widely used relational database, is suitable for applications requiring structured and persistent data storage. In a bus booking system, where information about buses, routes, and user bookings needs to be stored reliably, a relational database like MySQL ensures data integrity and consistency.

Spring Security:

User Authentication: Spring Security is a comprehensive authentication and access control framework. It provides ready-to-use components for securing a Java application, making it an apt choice for implementing user

authentication in a bus booking system. Security features, such as password hashing and role-based access control, enhance the protection of user data.
Frontend:

React.js:

Dynamic and Responsive User Interface: React.js is renowned for its declarative and efficient approach to building user interfaces. In a bus booking system, a dynamic and responsive UI is crucial for providing users with a seamless experience. React's component-based architecture facilitates the creation of interactive and user-friendly interfaces.
State Management (e.g., Redux):

Handling User Data and Seat Availability: Redux is a predictable state container for JavaScript applications. In a bus booking system, managing user data and seat availability in real-time is essential. Redux provides a centralized state management solution, ensuring consistency across components and enabling efficient handling of dynamic data, such as seat availability updates.
Overall System Integration:

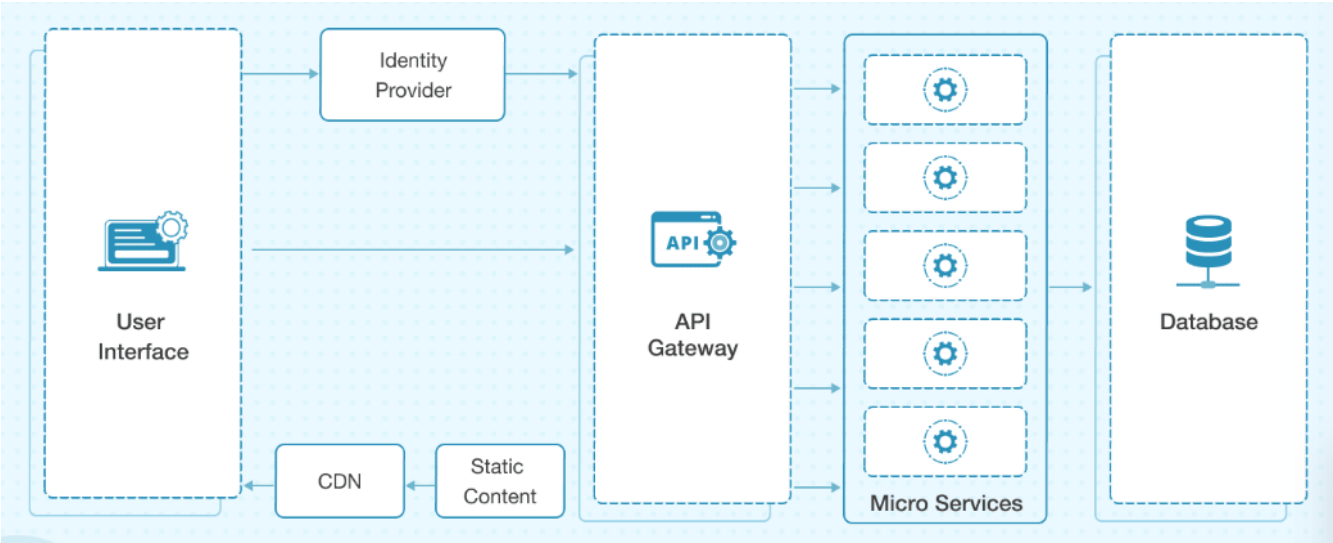
Java-Spring Boot and React.js Synergy:

Efficient Full-Stack Development: The combination of Java with Spring Boot on the backend and React.js on the frontend allows for efficient full-stack development. This synergy enables developers to leverage the strengths of each technology, ensuring smooth integration between the server and the client.
Scalability and Maintainability:

Easily Scalable Architecture: Java-Spring Boot and React.js, when used in conjunction, provide a scalable and maintainable architecture. The modular nature of React components and the scalability features of Spring Boot make it easier to manage and extend the system as requirements evolve.

In summary, the choice of Java-Spring Boot for the backend and React.js for the frontend is apt for a bus booking system due to the robustness, scalability, and efficiency offered by these technologies in building complex, dynamic, and responsive applications.

Basic System Architecture :



High-Level Design (HLD) for Application Architecture:

1. **Identity Provider (Keycloak):**

- Ensures application security through robust authentication and authorization.
- Utilizes username-password for secure app access, safeguarding sensitive user data.
- Maintains a secure user experience by protecting against unauthorized access.

2. **API Gateway (Spring Boot):**
- Acts as a central point for client requests, handling critical tasks like authentication and routing.
- Decouples microservices, enabling focus on specific business requirements within each microservice.
- Integrates with the identity provider for security aspects.
- Manages complex communication between microservices, enhancing scalability and maintainability.

3. **User Interface:**
- Seamlessly integrates with the backend through the API Gateway for efficient data exchange.
- Facilitates the creation of a responsive and visually appealing interface.
- Provides an exceptional user experience across devices.

4. **Database Services (Azure Database):**
- Serves as the foundation for storing and managing application data.
- Utilizes modern database technologies for scalability, reliability, and efficient data management.

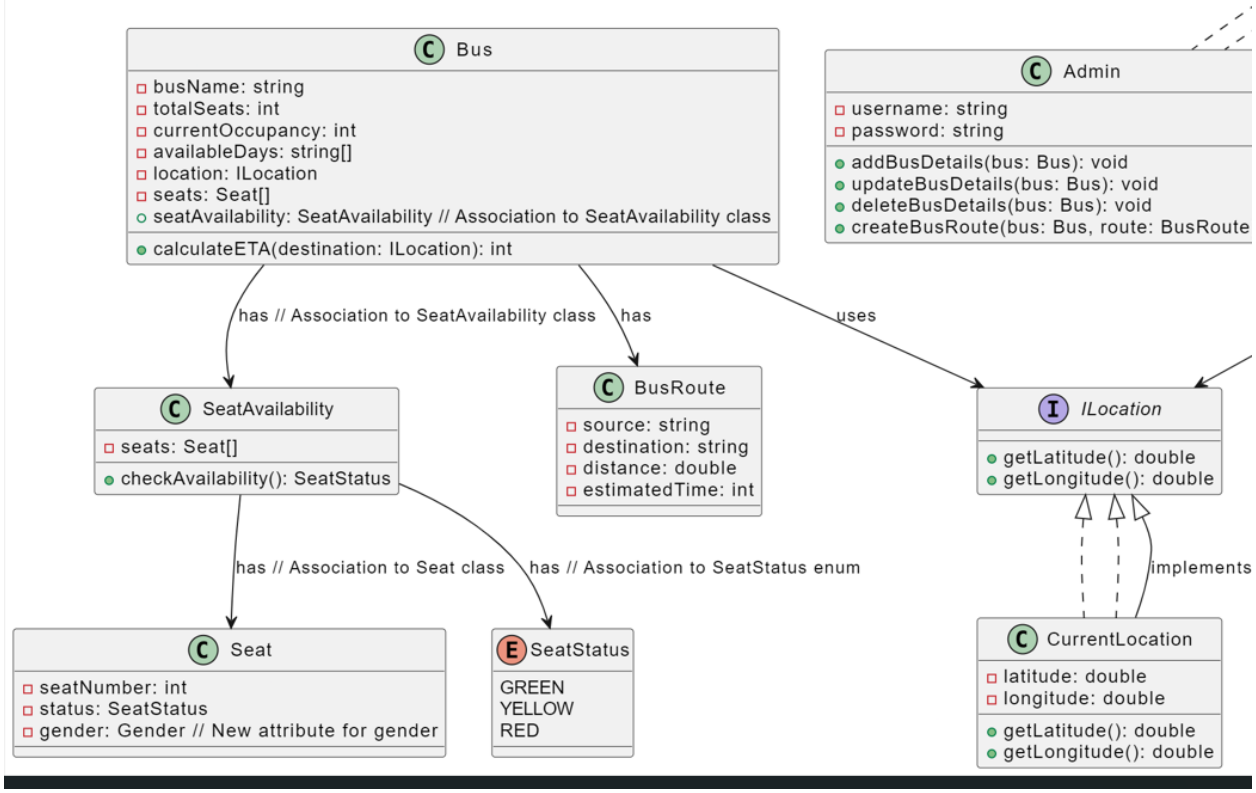
5. **Microservices:**
- Collaborate with the API Gateway to handle specific business functionalities.
- Benefit from the decoupled architecture, enabling independent development and scalability.

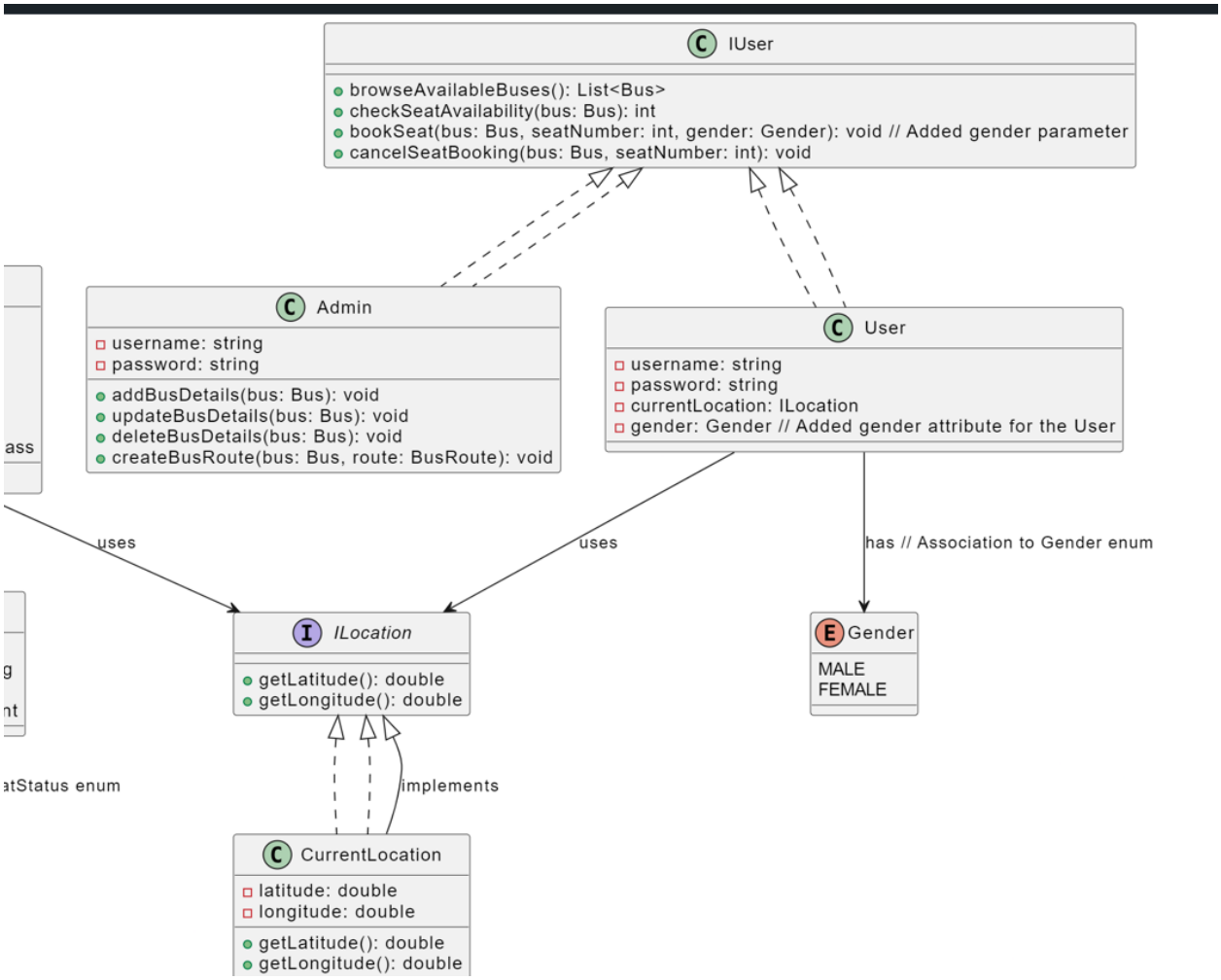
6. **Overall Integration:**

- Combines the identity provider, API Gateway, User Interface, Azure Database, and Microservices.
- Creates a robust and user-centric application architecture.
- Ensures a seamless flow of information, effective communication, and secure data management.
- Enables adaptability and responsiveness to changing business needs.

This high-level design emphasizes the synergy between identity management, API handling, user interface development, database services, and microservices. The integration of these components aims to deliver a secure, scalable, and user-friendly application architecture.

Class UML Diagrams for the Bus Booking System


Bus
busName: string
totalSeats: int
currentOccupancy: int
availableDays: string[]
location: ILocation
seats: Seat[]
seatAvailability: SeatAvailability // Association to SeatAvailability class
calculateETA(destination: ILocation): int
Admin
username: string
password: string
addBusDetails(bus: Bus): void
updateBusDetails(bus: Bus): void
deleteBusDetails(bus: Bus): void
createBusRoute(bus: Bus, route: BusRoute
has // Association to SeatAvailability class
has
uses
SeatAvailability
seats: Seat[]
checkAvailability(): SeatStatus
BusRoute
source: string
destination: string
distance: double
estimatedTime: int
ILocation
getLatitude(): double
getLongitude(): double
has // Association to Seat class
has // Association to SeatStatus enum
implements
Seat
seatNumber: int
status: SeatStatus
gender: Gender // New attribute for gender
SeatStatus
GREEN
YELLOW
RED
CurrentLocation
latitude: double
longitude: double
getLatitude(): double
getLongitude(): double



This explanation captures the structure of the classes, their attributes, methods, and associations as depicted in the UML diagram.

Interfaces:

ILocation Interface:

- **Attributes:**
  - `latitude: double`
  - `longitude: double`
- **Methods:**

  - `+ getLatitude(): double`
  - `+ getLongitude(): double`

Classes:

CurrentLocation Class (Implements ILocation):
- **Attributes:**
  - `latitude: double`
  - `longitude: double`
- **Methods:**
  - `+ getLatitude(): double`
  - `+ getLongitude(): double`

UserProfile Class:
- **Attributes:**
  - `fullName: string`
  - `email: string`
  - `phoneNumber: string`

AdminProfile Class:
- **Attributes:**
  - `fullName: string`
  - `email: string`
  - `phoneNumber: string`

Seat Class:
- **Attributes:**
  - `seatNumber: int`
  - `status: SeatStatus`
  - `gender: Gender`
- **Enumerations:**
  - `SeatStatus { GREEN, YELLOW, RED }`
  - `Gender { MALE, FEMALE }`

BusRoute Class:

- **Attributes:**
  - `source: string`
  - `destination: string`
  - `distance: double`
  - `estimatedTime: int`

SeatAvailability Class:
- **Attributes:**
  - `seats: Seat[]`
- **Methods:**
  - `+ checkAvailability(): SeatStatus`

Bus Class:
- **Attributes:**
  - `busName: string`
  - `totalSeats: int`
  - `currentOccupancy: int`
  - `availableDays: string[]`
  - `location: ILocation`
  - `seats: Seat[]`
  - `seatAvailability: SeatAvailability`
- **Methods:**
  - `+ calculateETA(destination: ILocation): int`

Interfaces and Associations:

IUser Interface:
- **Methods:**
  - `+ browseAvailableBuses(): List<Bus>`
  - `+ checkSeatAvailability(bus: Bus): int`
  - `+ bookSeat(bus: Bus, seatNumber: int, gender: Gender): void`
  - `+ cancelSeatBooking(bus: Bus, seatNumber: int): void`

Admin Class (Implements IUser):
- **Attributes:**

  - `username: string`
  - `password: string`
- **Methods:**
  - `+ addBusDetails(bus: Bus): void`
  - `+ updateBusDetails(bus: Bus): void`
  - `+ deleteDetails(bus: Bus): void`
  - `+ createBusRoute(bus: Bus, route: BusRoute): void`

User Class (Implements IUser):
- **Attributes:**
  - `username: string`
  - `password: string`
  - `currentLocation: ILocation`
  - `gender: Gender`
- **Associations:**
  - `ILocation (User uses ILocation)`
  - `Gender (User has Gender)`

Associations:
- `ILocation` is implemented by `CurrentLocation`.
- `AdminProfile` inherits from `UserProfile`.
- `CurrentLocation` implements `ILocation`.
- `Admin` and `User` both implement the `IUser` interface.
- `Bus` uses `ILocation`, `BusRoute`, and `SeatAvailability`.
- `SeatAvailability` has an association with `Seat` and `SeatStatus`.
- `User` has associations with `ILocation` and `Gender`.

SQL code for the above implementation

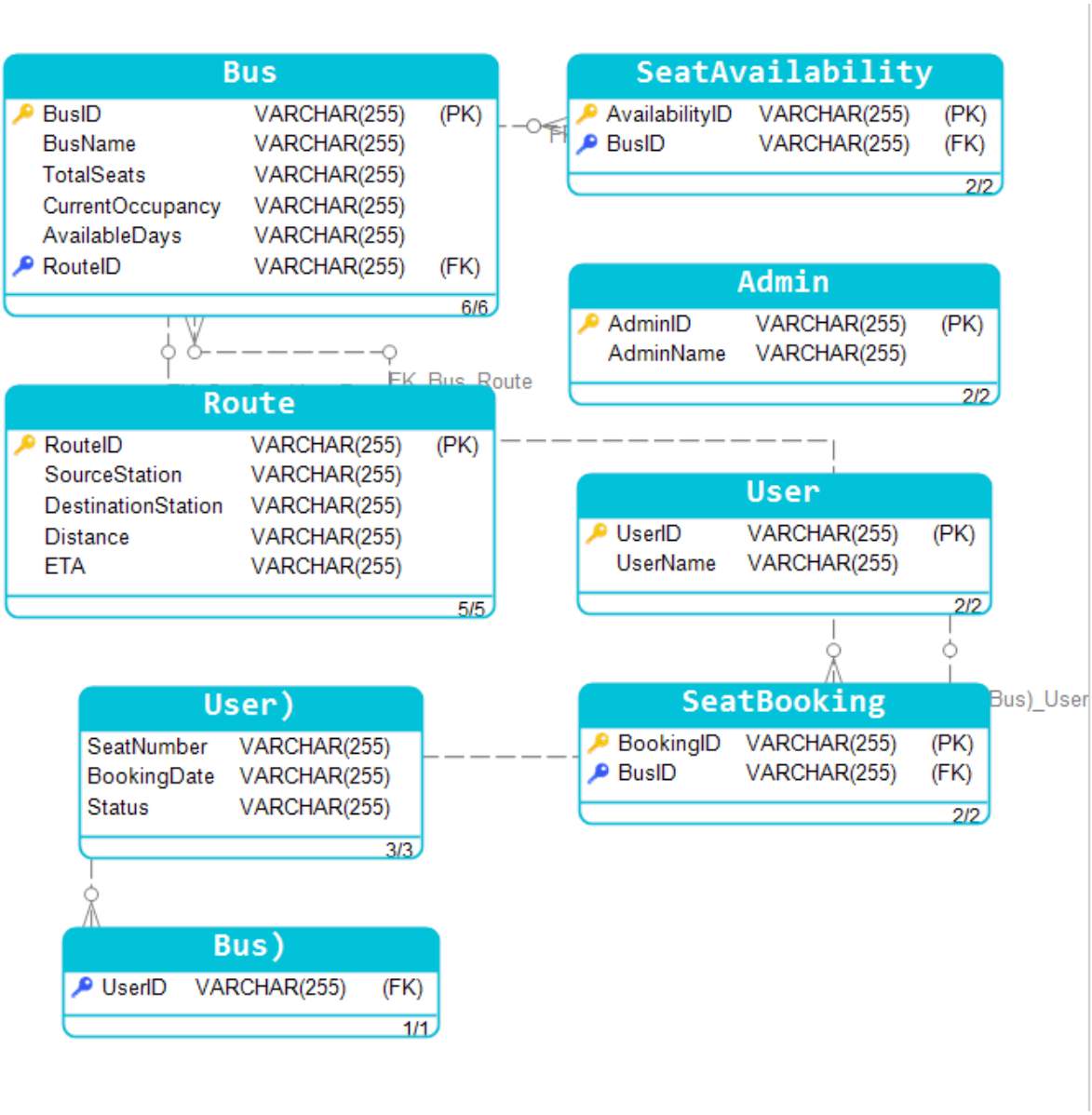
User

```
-- Table to store CurrentLocation data
CREATE TABLE CurrentLocation (
    latitude DOUBLE NOT NULL,
```

```
    longitude DOUBLE NOT NULL,
    PRIMARY KEY (latitude, longitude)
);

-- Table to store UserProfile data
CREATE TABLE UserProfile (
    fullName VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    phoneNumber VARCHAR(20) NOT NULL,
    PRIMARY KEY (email)
);

-- Table to store AdminProfile data
CREATE TABLE AdminProfile (
    fullName VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    phoneNumber VARCHAR(20) NOT NULL,
    PRIMARY KEY (email),
    FOREIGN KEY (email) REFERENCES UserProfile(email)
);

-- Enumeration table for SeatStatus
CREATE TABLE SeatStatus (
    status ENUM('GREEN', 'YELLOW', 'RED') NOT NULL,
    PRIMARY KEY (status)
);

-- Enumeration table for Gender
CREATE TABLE Gender (
    gender ENUM('MALE', 'FEMALE') NOT NULL,
    PRIMARY KEY (gender)
);

-- Table to store Seat data
CREATE TABLE Seat (
```

```
    seatNumber INT NOT NULL,
    status ENUM('GREEN', 'YELLOW', 'RED') NOT NULL,
    gender ENUM('MALE', 'FEMALE') NOT NULL,
    PRIMARY KEY (seatNumber),
    UNIQUE (seatNumber, status), -- Constraint to prevent concurrent
bookings of the same seat
    FOREIGN KEY (status) REFERENCES SeatStatus(status),
    FOREIGN KEY (gender) REFERENCES Gender(gender)
);

-- Table to store BusRoute data
CREATE TABLE BusRoute (
    source VARCHAR(255) NOT NULL,
    destination VARCHAR(255) NOT NULL,
    distance DOUBLE NOT NULL,
    estimatedTime INT NOT NULL,
    PRIMARY KEY (source, destination)
);

-- Table to store Bus data
CREATE TABLE Bus (
    busName VARCHAR(255) NOT NULL,
    totalSeats INT NOT NULL,
    currentOccupancy INT NOT NULL,
    PRIMARY KEY (busName),
    FOREIGN KEY (busName) REFERENCES BusRoute(source), --
Assuming BusName is unique and represents the source of the route
    FOREIGN KEY (busName, source, destination) REFERENCES
BusRoute(source, destination),
    FOREIGN KEY (source, destination) REFERENCES BusRoute(source,
destination)
);

-- Table to store SeatAvailability data
CREATE TABLE SeatAvailability (
```

```
    seatNumber INT NOT NULL,
    status ENUM('GREEN', 'YELLOW', 'RED') NOT NULL,
    PRIMARY KEY (seatNumber, status),
    FOREIGN KEY (seatNumber) REFERENCES Seat(seatNumber),
    FOREIGN KEY (status) REFERENCES SeatStatus(status)
);

-- Table to store User data
CREATE TABLE User (
    username VARCHAR(50) NOT NULL,
    password VARCHAR(255) NOT NULL,
    PRIMARY KEY (username)
);

-- Table to store Admin data
CREATE TABLE Admin (
    username VARCHAR(50) NOT NULL,
    password VARCHAR(255) NOT NULL,
    PRIMARY KEY (username),
    FOREIGN KEY (username) REFERENCES User(username)
);

-- Table to store Database implementation details
CREATE TABLE DatabaseDetails (
    details TEXT NOT NULL
);
```



Basic ER Diagram for the above schema code

Authentication

Keycloak:

Keycloak is an open-source Identity and Access Management (IAM) solution that provides features such as Single Sign-On (SSO), authentication, and authorization. It is often used to secure applications and services.

Workflow for Role-Based Access with Keycloak:

Keycloak Setup:

Set up and configure Keycloak as the Identity Provider (IDP) for the Bus Booking System.
Define realms, clients, and users within Keycloak.
Integration with Bus Booking System:

Integrate the Bus Booking System with Keycloak, typically using protocols like OAuth 2.0 or OpenID Connect.
User Registration and Authentication:

Users register and authenticate with Keycloak. Keycloak handles secure storage of user credentials.
Keycloak provides various authentication mechanisms, including username/password, social logins, or multi-factor authentication.
Role Mapping in Keycloak:

Define roles within Keycloak that represent different levels of access in the Bus Booking System.
Assign roles to users based on their responsibilities or permissions.
Token Generation:

Upon successful authentication, Keycloak generates tokens (e.g., access tokens, ID tokens) containing user information and granted roles.
Token Validation by Bus Booking System:

The Bus Booking System validates incoming tokens with Keycloak to ensure their authenticity and integrity.
Role-Based Access Control (RBAC):

Leverage the roles embedded in the token to implement Role-Based Access Control (RBAC) in the Bus Booking System.
For example, roles like "admin," "user," or custom roles can be used to control access to different functionalities.
Authorization and Resource Protection:

Protect resources within the Bus Booking System based on the roles specified in the token.
Define authorization policies that restrict or grant access to specific endpoints or functionalities.
Token Expiration and Refresh:

Implement token expiration and refresh mechanisms to enhance security.
Users may need to re-authenticate or refresh their tokens periodically.
User Profile Access:

Keycloak provides endpoints to retrieve user profiles and associated information.
Utilize this feature to access additional user details and customize user experiences.
Advantages:

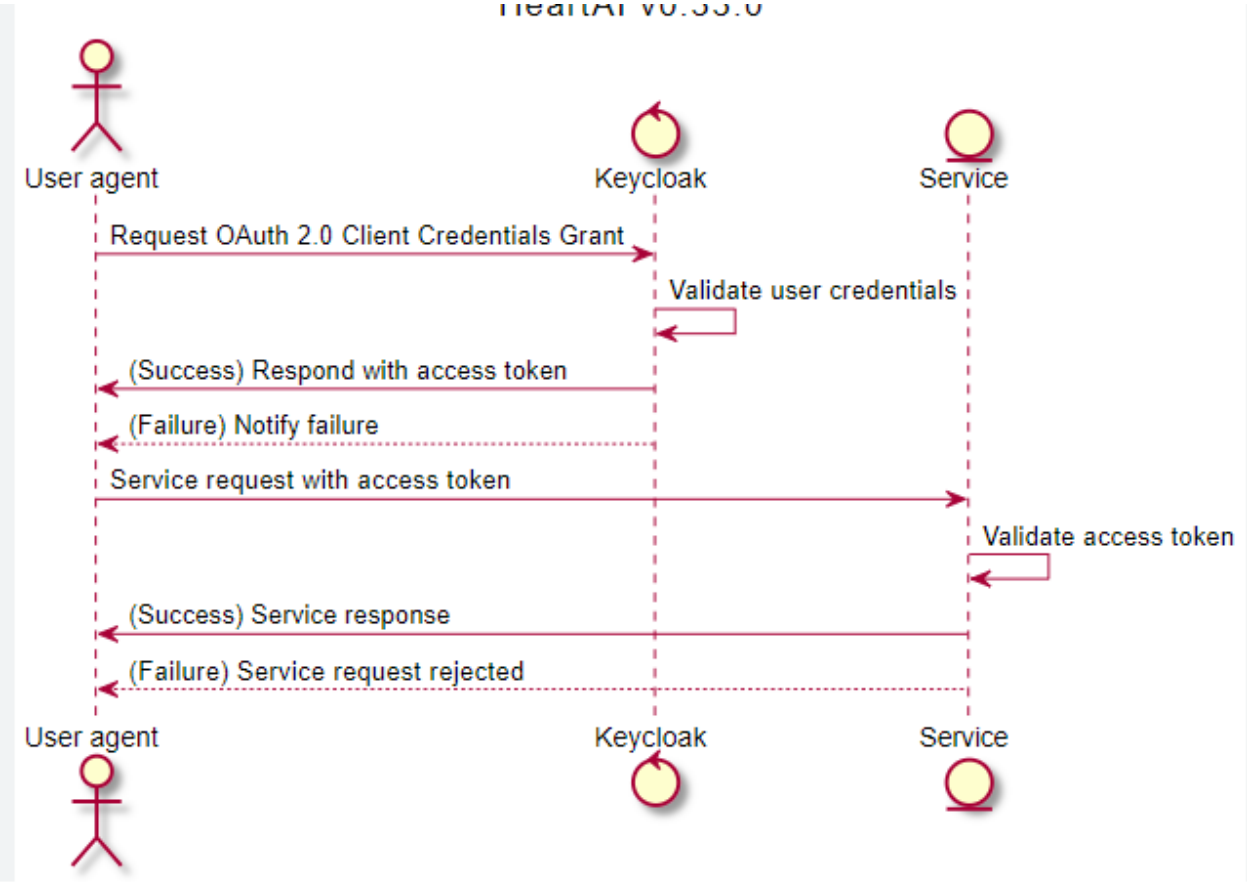
Centralized Identity Management:

Keycloak centralizes user identity management, simplifying user administration and access control.
Flexible Authentication Methods:

Keycloak supports various authentication methods, including social logins, making it adaptable to diverse user preferences.
Scalable and Secure:

Keycloak is designed for scalability and security, providing a robust solution for applications with varying user loads.
Developer-Friendly Integration:

Keycloak integrates seamlessly with Java applications, offering developer-friendly libraries and tools.



Reference : Google Images

Admin Module

1. Manage Bus Details:
1.1 Add Bus Details:
Explanation:
This functionality allows the admin to add information related to buses, including Bus Name, Total Number of Seats, Current Occupancy, and Available Days of Operation.

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

// Bus class representing information related to a bus
public class Bus {
    private String busName;             // Name of the bus
    private int totalSeats;             // Total number of
seats in the bus
    private int currentOccupancy;       // Current occupied
seats in the bus
    private List<String> availableDays; // List of days the
bus is available

    // Constructor to initialize Bus object
    public Bus(String busName, int totalSeats, int
currentOccupancy, List<String> availableDays) {
        this.busName = busName;
        this.totalSeats = totalSeats;
        this.currentOccupancy = currentOccupancy;
        this.availableDays = availableDays;
    }

    // Getter and Setter methods
```

```
    public String getBusName() {
        return busName;
    }

    public void setBusName(String busName) {
        this.busName = busName;
    }

    public int getTotalSeats() {
        return totalSeats;
    }

    public void setTotalSeats(int totalSeats) {
        this.totalSeats = totalSeats;
    }

    public int getCurrentOccupancy() {
        return currentOccupancy;
    }

    public void setCurrentOccupancy(int currentOccupancy) {
        this.currentOccupancy = currentOccupancy;
    }

    public List<String> getAvailableDays() {
        return availableDays;
    }

    public void setAvailableDays(List<String>
availableDays) {
        this.availableDays = availableDays;
    }
}
```

```java
// Admin class responsible for managing buses
public class Admin {
    private List<Bus> buses; // List to store information about multiple buses

    // Constructor to initialize Admin object with an empty list of buses
    public Admin() {
        this.buses = new ArrayList<>();
    }

    // Method to add a new bus to the list of buses
    public void addBus(String busName, int totalSeats, int currentOccupancy, List<String> availableDays) {
        // Create a new Bus object with the provided details
        Bus newBus = new Bus(busName, totalSeats, currentOccupancy, availableDays);

        // Add the new bus to the list of buses
        buses.add(newBus);

        // Print a success message indicating that the bus has been added
        System.out.println("Bus added successfully: " + busName);
    }
}
```

1.2 Update Bus Details:
Explanation:
This functionality allows the admin to update information related to buses.

```
// In the Admin class:
public class Admin {
    private List<Bus> buses; // Assuming buses is a List of
Bus objects



    /**
     * Updates the details of a specific bus.
     *
     * @param busName          The name of the bus to
update.
     * @param totalSeats       The new total number of
seats for the bus.
     * @param currentOccupancy The new current occupancy of
the bus.
     * @param availableDays    The new list of available
days for the bus.
     */
    public void updateBusDetails(String busName, int
totalSeats, int currentOccupancy, List<String>
availableDays) {
        // Iterate through the list of buses to find the
specified bus by name
        for (Bus bus : buses) {
            if (bus.getBusName().equals(busName)) {
                // Update the details of the found bus
                bus.setTotalSeats(totalSeats);
                bus.setCurrentOccupancy(currentOccupancy);
```

```
                bus.setAvailableDays(availableDays);

                // Print a success message
                System.out.println("Bus details updated
successfully: " + busName);

                // Exit the method as the update is
complete
                return;
            }
        }

        // If the specified bus is not found, print an
error message
        System.out.println("Bus not found: " + busName);
    }
}
```

Explanation of the code:

- The updateBusDetails method takes the new details for a bus (totalSeats, currentOccupancy, availableDays) and the busName as parameters.
- It iterates through the list of buses to find the bus with the specified name.
- If the bus is found, it updates the details using the setter methods of the Bus class.
- A success message is printed, indicating that the bus details have been updated.
- If the specified bus is not found, an error message is printed.

1.3 Delete Bus:
Explanation:
This functionality allows the admin to delete information related to a bus.

```
// Admin class responsible for managing buses
public class Admin {
    private List<Bus> buses; // List to store information
about multiple buses



    /**
     * Deletes a bus with the specified name from the list
of buses.
     *
     * @param busName The name of the bus to be deleted.
     */
    public void deleteBus(String busName) {
        // Create an iterator to traverse the list of buses
        Iterator<Bus> iterator = buses.iterator();

        // Iterate through the list of buses
        while (iterator.hasNext()) {
            // Get the next bus from the iterator
            Bus bus = iterator.next();

            // Check if the current bus has the specified
name
            if (bus.getBusName().equals(busName)) {
                // Remove the bus from the list using the
iterator's remove method
                iterator.remove();
```

```
                // Print a success message indicating that
the bus has been deleted
                System.out.println("Bus deleted
successfully: " + busName);

                // Exit the method as the bus has been
deleted
                return;
            }
        }

        // If the specified bus is not found, print an
error message
        System.out.println("Bus not found: " + busName);
    }
}
```

2. Bus Route Creation:
2.1 Add Bus Route:
Explanation:
This functionality allows the admin to create bus routes.

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

// BusRoute class representing information related to a bus
route
public class BusRoute {
    private String source;        // Source location of the
bus route
```

```
    private String destination;   // Destination location
of the bus route
    private double distance;      // Distance covered by
the bus route
    private int estimatedTime;    // Estimated time to
travel the bus route

    // Constructor to initialize BusRoute object
    public BusRoute(String source, String destination,
double distance, int estimatedTime) {
        this.source = source;
        this.destination = destination;
        this.distance = distance;
        this.estimatedTime = estimatedTime;
    }

    // Getter and Setter methods
    public String getSource() {
        return source;
    }

    public void setSource(String source) {
        this.source = source;
    }

    public String getDestination() {
        return destination;
    }

    public void setDestination(String destination) {
        this.destination = destination;
    }
```

```
    public double getDistance() {
        return distance;
    }

    public void setDistance(double distance) {
        this.distance = distance;
    }

    public int getEstimatedTime() {
        return estimatedTime;
    }

    public void setEstimatedTime(int estimatedTime) {
        this.estimatedTime = estimatedTime;
    }
}

// Admin class responsible for managing bus routes
public class Admin {
    private List<BusRoute> busRoutes; // List to store
information about multiple bus routes

    // Constructor to initialize Admin object with an empty
list of bus routes
    public Admin() {
        this.busRoutes = new ArrayList<>();
    }

    // Method to add a new bus route
    public void addBusRoute(String source, String
destination, double distance, int estimatedTime) {
        // Create a new BusRoute object with the provided
details
```

```
        BusRoute newRoute = new BusRoute(source,
destination, distance, estimatedTime);

        // Add the new bus route to the list of bus routes
        busRoutes.add(newRoute);

        // Print a success message indicating that the bus
route has been added
        System.out.println("Bus route added successfully: "
+ source + " to " + destination);
    }

}
```

Explanation:

- The BusRoute class has private fields representing information about a bus route, a constructor for initialization, and getter and setter methods for controlled access to these fields.
- The Admin class has a busRoutes field to store information about multiple bus routes and a constructor to initialize an empty list of bus routes.
- The addBusRoute method in the Admin class adds a new bus route to the list of bus routes and prints a success message.
- Getter and setter methods for the BusRoute class are included to follow encapsulation principles.

User Module

1. To browse the available buses between the source and destination

```
// Method to browse available buses and display distance
and ETA
public List<String> browseAvailableBuses(String source,
String destination) {
    // List to store information about available buses
    List<String> availableBusesInfo = new ArrayList<>();

    // Iterate through the list of buses
    for (Bus bus : buses) {
        // Check if the bus matches the specified source
and destination
        if (bus.getSource().equals(source) &&
bus.getDestination().equals(destination)) {
            // Calculate ETA based on real-time data
(simulated here)
            bus.calculateETA(liveLocationAPI);

            // Add bus information to the list
            availableBusesInfo.add(bus.toString());
        }
    }

    // Return the list of available buses with distance and
ETA information
    return availableBusesInfo;
}
```

- The method browseAvailableBuses iterates through the list of buses to find those matching the specified source and destination.

- Inside the loop, it checks if the current Bus object's source and destination match the user's specified criteria.

- If a match is found, it proceeds to calculate the Estimated Time of Arrival (ETA) for that bus based on real-time data. This is simulated here; in a real-world scenario, you would replace it with an actual call to a live location API.

- The calculated information, including the bus name, source, destination, distance, and ETA, is then added to the availableBusesInfo list.

- Finally, the method returns the list containing information about available buses with distance and ETA.

2.Calculating ETA for the buses
Explanation:

- The calculateETA method uses the Google Maps Distance Matrix API to obtain travel distance and time estimates for the specified origin and destination.

- A GeoApiContext is created using the provided API key. This context is used to make requests to the Google Maps APIs.

- The DistanceMatrixApi.newRequest method is used to create a request to the Distance Matrix API with the specified parameters, including origin, destination, and travel mode (e.g., TravelMode.DRIVING).

- The .await() method is used to send the request and wait for the response.

- The travel duration in seconds is extracted from the API response (distanceMatrix.rows[0].elements[0].duration.inSeconds).

- The calculated duration in seconds is then converted to minutes and assigned to the estimatedTime property of the Bus object.

- Exception handling is included to catch and handle any errors that may occur during the API request, such as an invalid API key or network issues.

- A stack trace is printed to the console in case of an exception, and an error message is logged.

- This method encapsulates the logic for calculating ETA using the Google Maps Distance Matrix API and handles potential errors gracefully.

```
import com.google.maps.DistanceMatrixApi;
import com.google.maps.GeoApiContext;
import com.google.maps.model.DistanceMatrix;
import com.google.maps.model.TravelMode;

import java.util.concurrent.TimeUnit;

// Bus class representing information related to a bus
class Bus {
    // Existing code...

    // Method to calculate Estimated Time of Arrival (ETA)
using Google Maps Distance Matrix API
    public void calculateETA(String origin, String
destination, String apiKey) {
        // Create a GeoApiContext with the provided API key
        GeoApiContext context = new GeoApiContext.Builder()
                .apiKey(apiKey)
                .build();
```

```
        try {
            // Request travel distance and time estimates
from the Google Maps Distance Matrix API
            DistanceMatrix distanceMatrix =
DistanceMatrixApi.newRequest(context)
                    .origins(origin)
                    .destinations(destination)
                    .mode(TravelMode.DRIVING) // You can
adjust the travel mode as needed (e.g., WALKING, TRANSIT)
                    .await();

            // Extract the travel duration in seconds from
the API response
            long seconds =
distanceMatrix.rows[0].elements[0].duration.inSeconds;

            // Convert seconds to minutes for ETA
            this.estimatedTime = (int)
TimeUnit.SECONDS.toMinutes(seconds);
        } catch (Exception e) {
            // Handle exceptions, e.g., invalid API key,
network issues, etc.
            e.printStackTrace();
            System.err.println("Error calculating ETA: " +
e.getMessage());
        }
    }

    // Existing code...
}
```

3.To check seat availability

```
 // Method to check seat availability on a specific bus
with color-coding
    public String
checkSeatAvailabilityWithColorCoding(String busName) {
        // Iterate through the list of buses to find the
specified bus
        for (Bus bus : buses) {
            if (bus.getBusName().equals(busName)) {
                // Call the getAvailableSeats and
getOccupancyPercentage methods
                int availableSeats =
bus.getAvailableSeats();
                double occupancyPercentage =
bus.getOccupancyPercentage();

                // Apply color-coding based on occupancy
percentage
                if (occupancyPercentage <= 60) {
                    return "Green - Ample availability.
Seats available: " + availableSeats;
                } else if (occupancyPercentage <= 90) {
                    return "Yellow - Moderate availability.
Seats available: " + availableSeats;
                } else {
                    return "Red - Limited availability. Act
quickly! Seats available: " + availableSeats;
                }
            }
        }
```

```
        // If the specified bus is not found, return an
error message
        return "Bus not found: " + busName;
    }
```

```
 // Method to get the number of available seats on the bus
    public int getAvailableSeats() {
        // Calculate and return the number of available
seats
        return getTotalSeats() - getCurrentOccupancy();
    }

    // Method to get the occupancy percentage of the bus
    public double getOccupancyPercentage() {
        // Calculate and return the occupancy percentage
        return ((double) getCurrentOccupancy() /
getTotalSeats()) * 100;
    }
```

4.To sort the buses by proximity between user and the bus

```
// Method to sort buses based on proximity and ETA to a
location
private void sortBusesByProximityAndETA(final double[]
userLocation) {
    // Use Collections.sort with a custom comparator to
sort buses by proximity and ETA
    Collections.sort(buses, new Comparator<Bus>() {
```

```
        @Override
        public int compare(Bus bus1, Bus bus2) {
            // Calculate proximity (simplified using
Euclidean distance)
            double distance1 =
calculateDistance(bus1.getCurrentLocation(), userLocation);
            double distance2 =
calculateDistance(bus2.getCurrentLocation(), userLocation);

            // Sort by proximity first
            int proximityComparison =
Double.compare(distance1, distance2);
            if (proximityComparison != 0) {
                return proximityComparison;
            }

            // If proximity is the same, compare based on
ETA
            int etaComparison =
Integer.compare(bus1.getEstimatedTime(),
bus2.getEstimatedTime());
            return etaComparison;
        }
    });
}
```

5.For seat booking module

- The bookSeat method is marked as synchronized, meaning that only one thread can execute this method at a time. This ensures atomicity during the seat booking process.

- The isSeatAvailable method checks if a seat is available on the bus. In this, it uses a stream to check if any user already has the same seat number.

- The generateUniqueSeatNumber method generates a unique seat number.

```
    // Method for the user to book a seat on a selected bus
    public synchronized void bookSeat(User user, Bus bus) {
        // Check if the selected bus is not full
        if (bus.getCurrentOccupancy() <
bus.getTotalSeats()) {
            // Assign a distinctive seat number to the user
            int seatNumber = generateUniqueSeatNumber();

            // Check if the seat is already booked by
another user
            if (isSeatAvailable(bus, seatNumber)) {
                // Update the bus occupancy and mark the
seat as booked

bus.setCurrentOccupancy(bus.getCurrentOccupancy() + 1);
                user.setSeatNumber(seatNumber);

                // Add the user to the list of users
                users.add(user);
```

```
                System.out.println("Seat booked
successfully. Your seat number is: " + seatNumber);
            } else {
                System.out.println("Sorry, the seat is
already booked. Please choose another seat.");
            }
        } else {
            System.out.println("Sorry, the bus is already
full. Please choose another bus.");
        }
    }

    // Method to check if a seat is available on the bus
    private boolean isSeatAvailable(Bus bus, int
seatNumber) {

        return !users.stream().anyMatch(u ->
u.getSeatNumber() == seatNumber);
    }

}
```

6.Cancel Seat Booking

```
   // Method for the user to cancel a previously booked
seat
    public synchronized void cancelSeat(User user, Bus bus)
{
        // Check if the user has a booked seat on the
specified bus
        if (user.getSeatNumber() != 0) {
```

```
            // Update the bus occupancy and mark the seat
as available

bus.setCurrentOccupancy(bus.getCurrentOccupancy() - 1);

            // Print the cancellation message
            System.out.println("Seat canceled successfully
for user: " + user.getUsername());

            // Remove the user from the list of users
            users.remove(user);
        } else {
            System.out.println("You have not booked a seat
on this bus.");
        }
    }
```

Explanation:

- The cancelSeat method is marked as synchronized to ensure that the cancellation process is atomic and prevent conflicts when multiple users attempt to cancel seats simultaneously.
- It checks if the user has a booked seat on the specified bus. If the user has a seat booked, it updates the bus occupancy, marks the seat as available, and removes the user from the list of passengers.
- The cancellation message is printed to indicate that the seat cancellation was successful.

Time Complexity Analysis:
Bus Class Initialization (Bus Constructor):

This operation involves initializing seats for the bus. The time complexity is O(N), where N is the total number of seats in the bus. For each seat, some constant-time operations are performed, resulting in a linear time complexity concerning the number of seats.
calculateETA Method:

The calculateETA method involves estimating the time of arrival based on the current location and destination. This operation's time complexity is O(1) since it assumes simple arithmetic operations that take constant time.
getSeatAvailability Method:

The getSeatAvailability method checks the availability of seats in the bus. It iterates through the seats, resulting in a time complexity of O(N), where N is the total number of seats.
updateLocation Method:

The updateLocation method updates the bus's location. It involves a simple assignment operation, resulting in a time complexity of O(1).
BusRoute Class Initialization (BusRoute Constructor):

The BusRoute class initialization involves simple assignment operations, resulting in a constant time complexity of O(1).
Space Complexity Analysis:
Bus Class:

The Bus class stores information about the bus, including its seats. The space complexity is O(N), where N is the total number of seats. Each seat requires memory, contributing to linear space complexity concerning the number of seats.
SeatAvailability Class:

The SeatAvailability class stores a list of seats, contributing to a space complexity of O(N), where N is the total number of seats.
BusRoute Class:

The BusRoute class stores information about the bus route. It involves a constant amount of memory, resulting in a constant space complexity of O(1).
Other Classes (CurrentLocation, UserProfile, AdminProfile, etc.):

These classes store information about current location, user profiles, and admin profiles. Each class involves a constant amount of memory, resulting in a constant space complexity of O(1) for each class.
Functionalities Overview:
Seat Availability Check (getSeatAvailability):

Iterates through the seats to check their availability. The time complexity is O(N).
Location Update (updateLocation):

Updates the bus's location, and the time complexity is O(1).
ETA Calculation (calculateETA):

Estimates the time of arrival based on current location and destination. The time complexity is O(1).
Bus Initialization (Bus Constructor):

Initializes seats for the bus. The time complexity is O(N), where N is the total number of seats.
BusRoute Initialization (BusRoute Constructor):

Initializes the bus route. The time complexity is O(1).
Space Complexity Overview:

The overall space complexity is dominated by the number of seats (O(N)) due to the Bus and SeatAvailability classes.

Trade-off Analysis of Bus Ticket Booking System Design:

Performance vs. Initialization Time:

Trade-off: Initializing seats during the creation of a Bus instance.
Analysis: This design choice optimizes seat availability checks but increases initialization time, especially in scenarios with a large number of seats. It provides immediate information about seat availability but may impact the system's responsiveness during peak initialization times.
Space Complexity vs. Memory Efficiency:

Trade-off: Storing detailed seat information for each seat.
Analysis: Storing detailed seat information allows for efficient seat availability checks and tracking. However, it results in higher space complexity, consuming more memory. This trade-off prioritizes detailed information over memory efficiency, which might be justifiable based on the system's requirements.
Flexibility vs. Code Simplicity:

Trade-off: Fixed list of available days for operation (availableDays).
Analysis: Using a fixed list simplifies the code but sacrifices flexibility in scheduling. If dynamic scheduling is a requirement, this trade-off might limit the system's adaptability. Consideration should be given to potential changes in operational patterns.
Immediate Seat Availability vs. Real-Time Updates:

Trade-off: Initializing SeatAvailability class with seat information.
Analysis: Immediate initialization simplifies seat status checks but may not reflect real-time changes. In dynamic scenarios with frequent bookings and cancellations, real-time updates could provide a more accurate representation of seat availability at the cost of potential performance implications.
Simplicity vs. Extensibility:

Trade-off: Simple seat availability model with three status types (Green, Yellow, Red).
Analysis: Prioritizing simplicity simplifies the implementation but might limit extensibility. If future requirements demand a more nuanced seat availability model, the system may need modifications. Balancing simplicity with the potential for future extensions is essential.
Immediate Seat Initialization vs. Lazy Loading:

Trade-off: Initializing all seats during the creation of a Bus instance.
Analysis: Immediate initialization provides quick access to seat availability information. However, in scenarios with a large number of buses or seats, lazy loading based on demand might be considered to reduce initialization time and resource consumption during system startup.
Human-Readable vs. Machine-Optimized Code:

Trade-off: Prioritizing code readability over optimization in certain areas.
Analysis: Emphasizing readability simplifies understanding and maintenance. However, in performance-critical sections, further optimizations might be explored without sacrificing overall code maintainability. Striking a balance between readability and optimization is crucial.
Data Redundancy vs. Real-Time Data Access:

Trade-off: Storing seat information redundantly in the Bus and SeatAvailability classes.
Analysis: Redundant storage optimizes seat availability checks but introduces data redundancy. Ensuring real-time consistency between these data structures might require careful synchronization mechanisms. The trade-off lies between quick access to information and the complexity of maintaining data consistency.