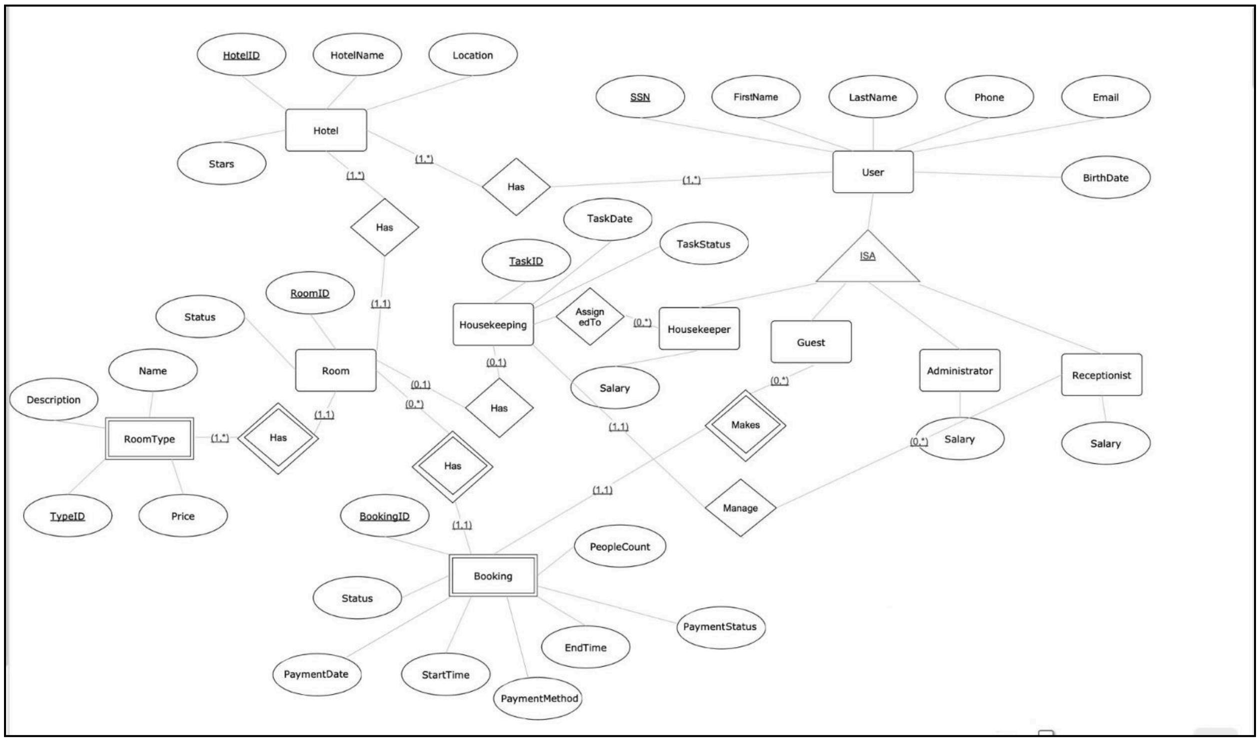
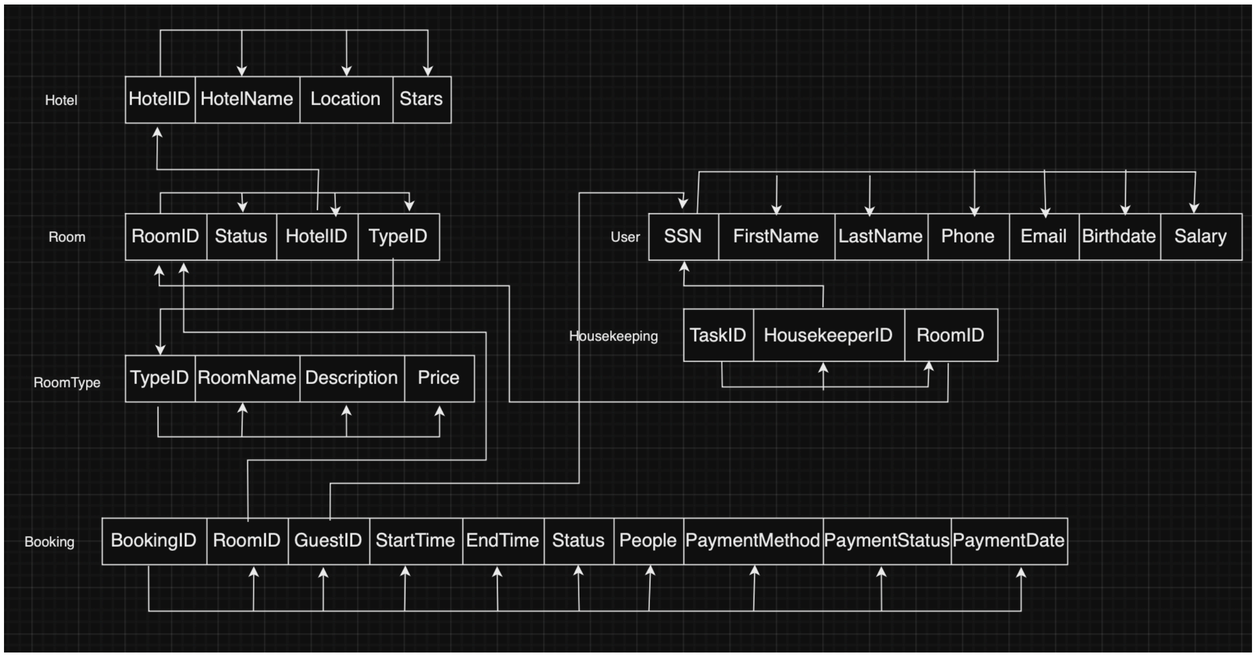
HOTEL MANAGEMENT SYSTEM

Ulaş Baran

ER Diagram:



Functional Dependencies:



Description of our design decisions and why we made them:

The database schema was designed to manage the operations of a hotel system, addressing various entities such as hotels, users, rooms, room types, bookings, and housekeepings. The goal was to create a flexible and normalized database structure that minimizes redundancy and ensures data integrity. Below are the descriptions of each key design decision and the rationale behind it:

1. Hotel Management

Entities:

- **Hotel:** Contains attributes like HotelID, HotelName, Location, and Stars.

Design Decision:

- Each hotel is uniquely identified by HotelID.

- Attributes like Location and Stars are included to differentiate between hotels and serve to various customer preferences.

2. Room Management

Entities:

- **Room:** Contains attributes like RoomID, Status, and a foreign key (TypeID) linking it to the RoomType table.
- **RoomType:** Includes TypeID, Name, Description, and Price.

Design Decision:

- Status in the Room table indicates whether the room is available or occupied.

Why: This structure minimizes redundancy and allows flexible management of room categories and their statuses.

3. User Management

Entities:

- **User:** Contains attributes like SSN, FirstName, LastName, Phone, Email, and BirthDate.
- **Roles (ISA Hierarchy):** Includes Guest, Administrator, Receptionist, and Housekeeper linked to User.

Design Decision:

- The ISA relationship was implemented to distinguish between user roles while sharing common attributes in the User table.
- We managed the Salary attribute by setting it as NULL for users who are not employees (e.g., guests). For employees (housekeepers, receptionists, administrators), the Salary is stored in the User table, ensuring that only relevant users have a salary value.
- **Scalability:** New roles can be added by extending the `ENUM` values or modifying the schema, ensuring adaptability for future system requirements.

- **Efficiency:** The use of `NULL` for irrelevant attributes ensures that no unnecessary data occupies storage space.

Why: This approach avoids duplicating user information and ensures each role's unique attributes are properly stored. It also reduces query complexity.

4. Booking Management

Entities:

- **Booking:** Includes BookingID, StartTime, EndTime, Status, PeopleCount, and foreign keys to Room and User.

Design Decision:

- A Booking table manages reservations, linking guests to specific rooms.
- Additional attributes like PaymentStatus, PaymentMethod, and PaymentDate were added to support financial tracking.
- A foreign key to Room ensures room availability is validated before booking.

Why: This design allows for detailed tracking of room usage and booking details, ensuring no conflicts occur for overbooking.

5. Housekeeping Management

Entities:

- **Housekeeping:** Includes TaskID, AssignedTo (links to Housekeeper), TaskDate, and TaskStatus.

Design Decision:

- A separate table manages housekeeping tasks, linking them to rooms and specific housekeepers.
- TaskStatus was added to monitor progress (e.g., Pending, Completed).

Why: This structure ensures efficient tracking of cleaning or maintenance activities, improving the hotel's operational workflow.

6. Payment Information

Entities:

- Payment-related attributes like PaymentStatus, PaymentMethod, and PaymentDate are part of the Booking table.
- At first, payment was considered as an entity with a primary key as the payment ID. However, since the booking ID directly identifies attributes like payment date, status, and method, having a separate payment entity would be redundant.
- Before the payment is done, payment method and payment date are set to null and payment status is set to pending.

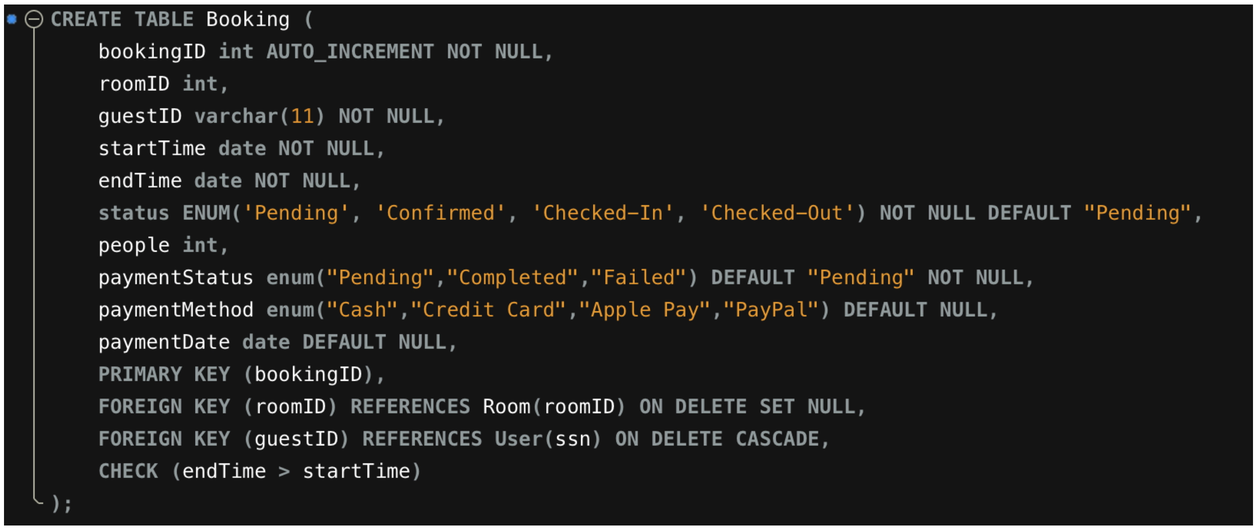
Design Decision:

- Instead of creating a separate Payment table, payment details are directly integrated into Booking.

Why: Payments are inherently tied to bookings, making it logical to store them together for simplicity and to reduce query complexity.

7. SQL Queries Explanation:

Example Create Table Statement:

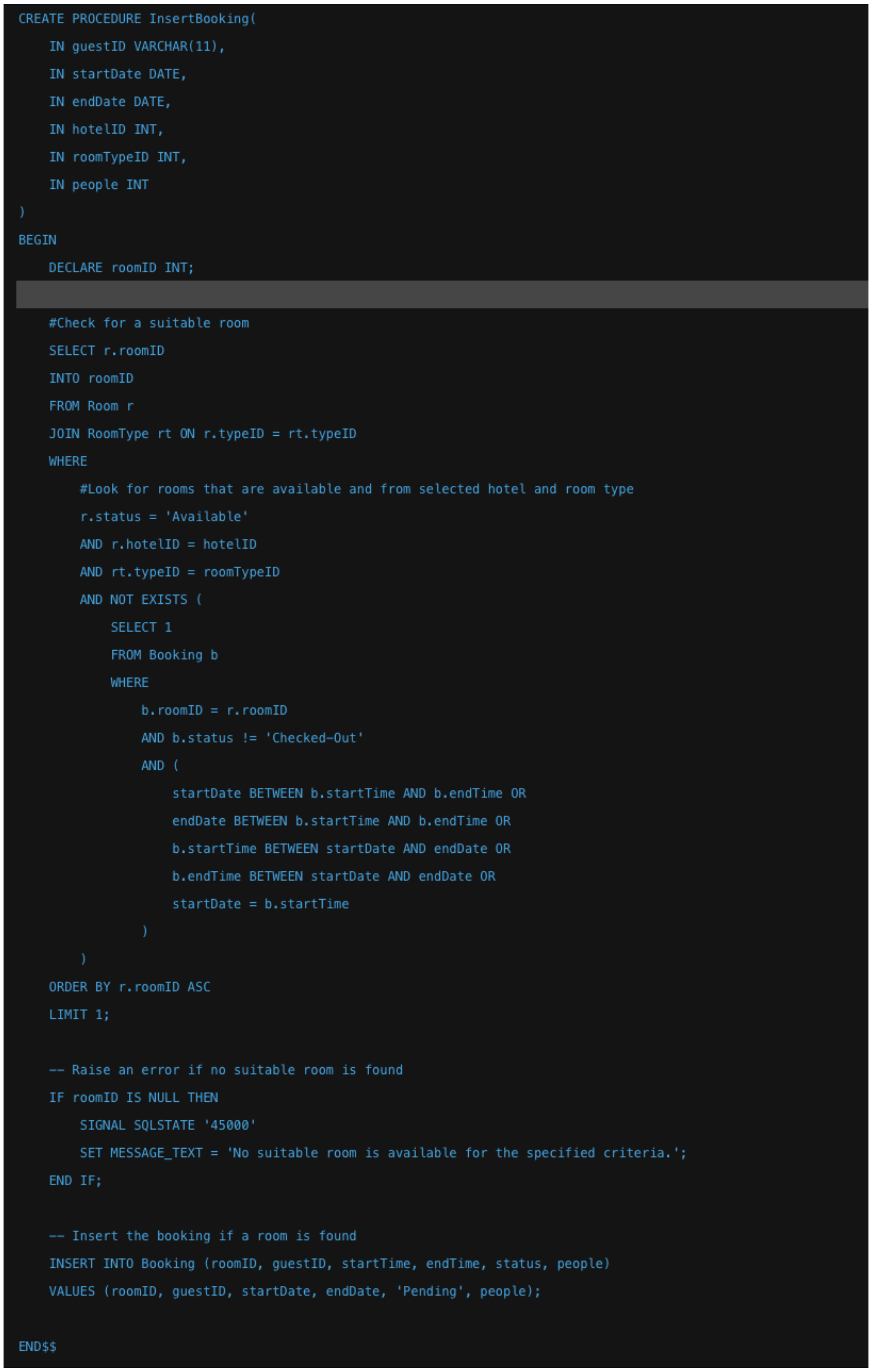


```
CREATE TABLE Booking (
    bookingID int AUTO_INCREMENT NOT NULL,
    roomID int,
    guestID varchar(11) NOT NULL,
    startTime date NOT NULL,
    endTime date NOT NULL,
    status ENUM('Pending', 'Confirmed', 'Checked-In', 'Checked-Out') NOT NULL DEFAULT "Pending",
    people int,
    paymentStatus enum("Pending","Completed","Failed") DEFAULT "Pending" NOT NULL,
    paymentMethod enum("Cash","Credit Card","Apple Pay","PayPal") DEFAULT NULL,
    paymentDate date DEFAULT NULL,
    PRIMARY KEY (bookingID),
    FOREIGN KEY (roomID) REFERENCES Room(roomID) ON DELETE SET NULL,
    FOREIGN KEY (guestID) REFERENCES User(ssn) ON DELETE CASCADE,
    CHECK (endTime > startTime)
);
```

- In the Booking table, bookingID is an integer with auto incrementation. This ensures that when inserting a booking, the user doesn’t need to include a bookingID, instead the system allocates one for them. This enhances the user experience.
- Status is defaulted as “Pending” since any booking that has been made needs to be approved by the receptionist to be “Confirmed”.
- While inserting a booking, payment details don't need to be given. They are set to NULL as default. When the payment is done the relevant attributes are updated.
- Foreign key to the room table has deletion integrity ON DELETE SET NULL, this means when the room of a booking is deleted, roomID in the booking table is set to null. This is a choice we made because if a certain room is deleted, it doesn’t mean that booking should be deleted. Instead guests may be noticed and ask to change the room of booking.
- Foreign key to the user table has deletion integrity ON DELETE CASCADE, this means that when a guest account is deleted, the bookings of that guest are also deleted. We made that choice because if there is no more such guest, then there is no need to store bookings of that guest.
- CHECK (endTime > startTime) statement checks if the end time of booking is after the start time of booking. This constraint is added to enforce logic.

Example Insert Statement:

- (The query image is given below.)
- Create procedure is used for creating a function in sql to insert booking. This will help a lot when we need to insert multiple bookings since it is more clear and easy to understand.
- In the procedure a local roomId is instantiated. This will be the room to create the booking with. Then in the where clause rooms with given hotel and room type are checked. Checking is done by comparing the start and end dates of the bookings already in the system and the booking we want to do. If there is a room available to our desires insert statement is executed, if not an error raises informing the user.


```
CREATE PROCEDURE InsertBooking(
    IN guestID VARCHAR(11),
    IN startDate DATE,
    IN endDate DATE,
    IN hotelID INT,
    IN roomTypeID INT,
    IN people INT
)
BEGIN
    DECLARE roomID INT;

    #Check for a suitable room
    SELECT r.roomID
    INTO roomID
    FROM Room r
    JOIN RoomType rt ON r.typeID = rt.typeID
    WHERE
        #Look for rooms that are available and from selected hotel and room type
        r.status = 'Available'
        AND r.hotelID = hotelID
        AND rt.typeID = roomTypeID
        AND NOT EXISTS (
            SELECT 1
            FROM Booking b
            WHERE
                b.roomID = r.roomID
                AND b.status != 'Checked-Out'
                AND (
                    startDate BETWEEN b.startTime AND b.endTime OR
                    endDate BETWEEN b.startTime AND b.endTime OR
                    b.startTime BETWEEN startDate AND endDate OR
                    b.endTime BETWEEN startDate AND endDate OR
                    startDate = b.startTime
                )
        )
    ORDER BY r.roomID ASC
    LIMIT 1;

    -- Raise an error if no suitable room is found
    IF roomID IS NULL THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'No suitable room is available for the specified criteria.';
    END IF;

    -- Insert the booking if a room is found
    INSERT INTO Booking (roomID, guestID, startTime, endTime, status, people)
    VALUES (roomID, guestID, startDate, endDate, 'Pending', people);

END$$
```

Example Select Statement:

```
SELECT r.roomID, r.status, rt.roomName, rt.price, h.hotelName, h.location
FROM Room r
JOIN RoomType rt ON r.typeID = rt.typeID
JOIN Hotel h ON r.hotelID = h.hotelID
WHERE r.status = 'Available'
  AND r.hotelID = 1            -- Replace with the guest's selected hotelID
  AND rt.typeID = 1            -- Replace with the guest's selected room type ID
ORDER BY r.roomID ASC;
```

- This is the query for viewing available rooms. It provides the rooms with their id, status, name, price, hotel name, location.

Example Delete Statement:

```
DELIMITER $$

CREATE PROCEDURE DeleteRoom(IN user_ssn VARCHAR(11), IN room_id INT)
BEGIN
    -- Check if the user is an Administrator
    DECLARE user_role ENUM('Guest', 'Receptionist', 'Administrator', 'Housekeeping');

    -- Get the role of the user
    SELECT role INTO user_role
    FROM User
    WHERE ssn = user_ssn;

    -- If the user is an Administrator, delete the room
    IF user_role = 'Administrator' THEN
        DELETE FROM Room
        WHERE roomID = room_id;
    ELSE
        -- Raise an error if the user is not an Administrator
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Only an administrator can delete a room.';
    END IF;
END$$

DELIMITER ;
```

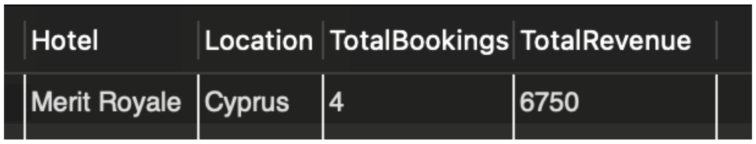
- Create procedure is used for creating a delete room function that only allows admins to use.

- Local user_role variable created and made equal to role of given ssn. Then if clause checks if the role is “Administrator” and if true executes the deletion, and if not raises an error.

Other Examples:

```
SELECT
    H.hotelName AS Hotel,
    H.location AS Location,
    COUNT(B.bookingID) AS TotalBookings,
    SUM(RT.price * DATEDIFF(B.endTime, B.startTime)) AS TotalRevenue
FROM
    Booking B
JOIN
    Room R ON B.roomID = R.roomID
JOIN
    RoomType RT ON R.typeID = RT.typeID
JOIN
    Hotel H ON R.hotelID = H.hotelID
WHERE
    B.status IN ('Confirmed', 'Checked-In', 'Checked-Out') -- Only count revenue-generating bookings
    AND B.paymentStatus = 'Completed' -- Only include bookings with completed payments
GROUP BY
    H.hotelID, H.hotelName, H.location
ORDER BY
    TotalRevenue DESC;
```

- This is the query for finding the most booked room type.
- Booking, room, room type and hotel tables are joined.
- SUM statement takes the difference between the end date and start date of booking and multiplies that with room price and sums all the bookings of a room type.
- Where clause checks if the calculated bookings are not “Pending” because those haven’t brang any income yet.
- Below is an example output.



Hotel	Location	TotalBookings	TotalRevenue
Merit Royale	Cyprus	4	6750

8. General Design Considerations

Normalization:

- The schema was normalized to 3NF to minimize redundancy. For example:
    - Room types are stored in a separate table Room Type to avoid duplicating room details.
    - User roles are split into child tables using the ISA hierarchy.

Scalability:

- Dynamic Role Management: The role attribute in the User table, implemented using an ENUM data type, ensures that new roles can be easily added or managed with minimal schema changes.
- Flexible Room Categories: The separation of RoomType as a distinct entity ensures that new categories of rooms can be added without modifying the existing structure, allowing scalability.
- ISA Hierarchy: The use of ISA relationships for user roles allows for the seamless addition of new employee or user types while maintaining shared attributes in a single User table.
- Normalization: Normalizing the schema to 3NF reduces redundancy and simplifies modifications, making it easier to scale the database as new features or data categories are introduced.

Developer Friendly:

- The design includes intuitive relationships and naming conventions, making it easier for developers and administrators to work with.

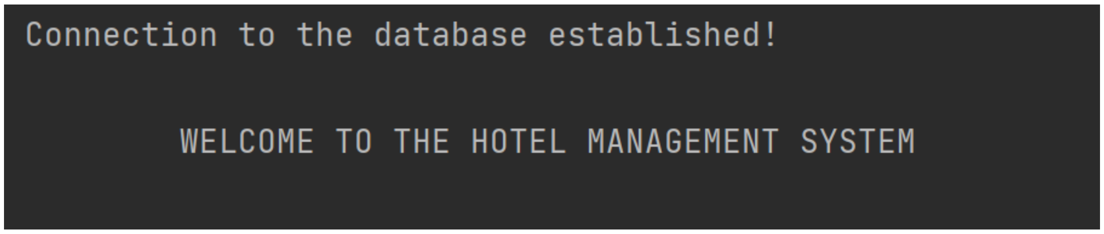
Conclusion

This database schema was designed to handle complex hotel operations efficiently while remaining easy to scale and maintain. By addressing different aspects like room management, user roles, housekeeping, and payment, the structure supports the seamless integration of all hotel services, ensuring data consistency and a smooth user experience.

Part 2 of the Project (Java):

1. Overview of the TerminalMenu Class

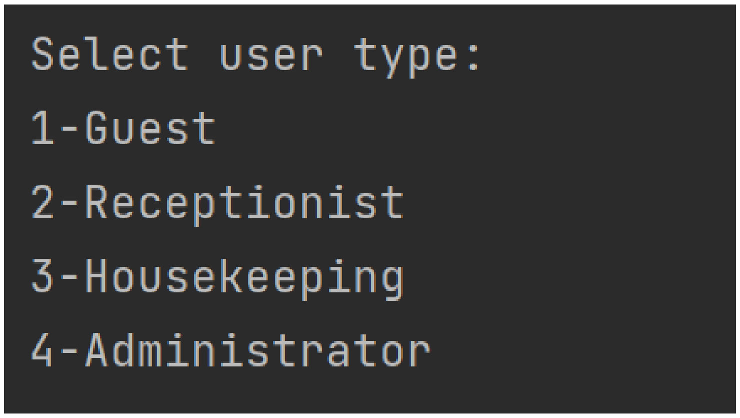
- The TerminalMenu class is responsible for handling user interactions in the console. It provides menus for various roles such as Guest, Receptionist, Housekeeping, and Administrator. Each role has access to different functionalities, and the user is guided through the available options via this class.



```
Connection to the database established!

        WELCOME TO THE HOTEL MANAGEMENT SYSTEM
```

Key Features of TerminalMenu:

- Dynamic Menu: It uses a loop to keep displaying the menu until the user chooses to exit, making it easy to navigate through different options without restarting the application.
- Hotel Selection: Receptionists, administrators, and housekeepers can select the hotel they are working at, ensuring that their data and tasks are confined to their designated hotel.
- Role-based Menus: The class first asks the user to select a role (e.g., Guest, Receptionist, Administrator), then shows the respective menu based on the role chosen.



```
Select user type:
1-Guest
2-Receptionist
3-Housekeeping
4-Administrator
```

Error Handling

User input is validated at each step:

- Invalid Number Inputs: The program uses try-catch blocks to handle invalid input formats and prompts users to retry.
- Dynamic Feedback: After each action, users are given the choice to either return to the menu or exit the program.

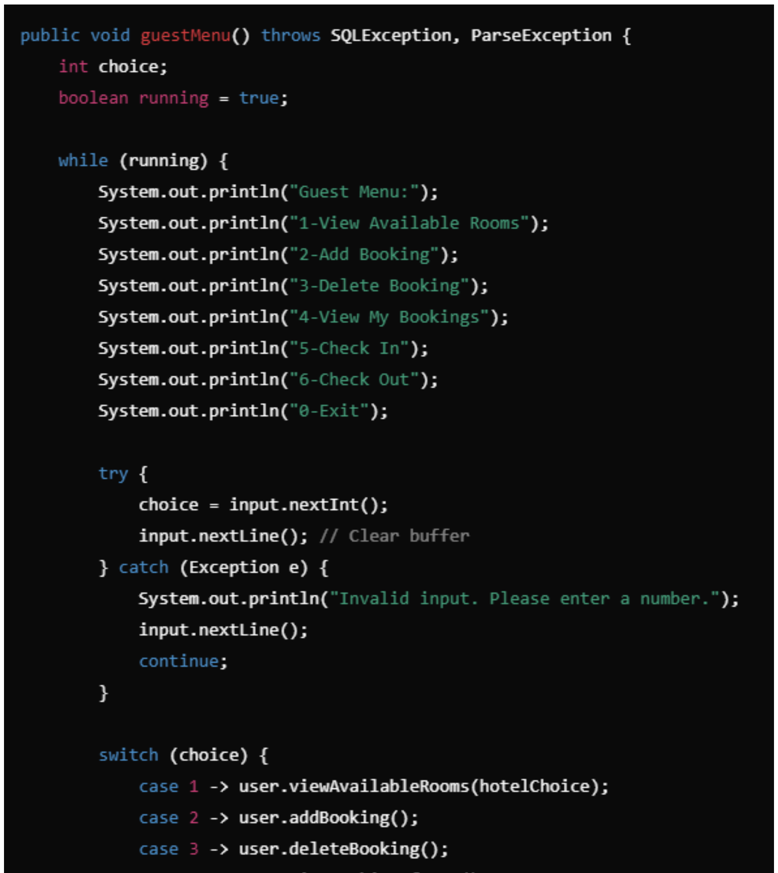
Detailed Explanation of Menus:

1. Hotel Selection Menu:This menu displays available hotels from the database and allows users to select the one they work at or are visiting. Once selected, it sets the hotelChoice variable to scope future actions.

2. Guest Menu

Guests interact with the system through the guestMenu. They can:

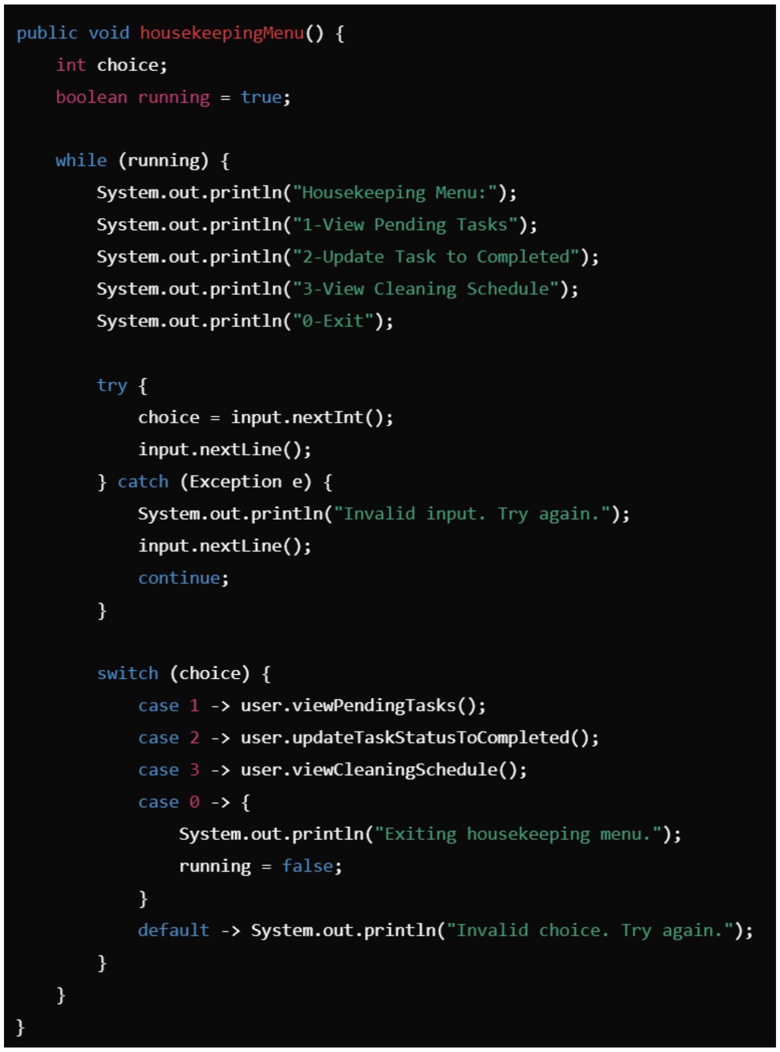
- View available rooms.
- Manage their bookings.
- Perform check-in and check-out operations.



```java
public void guestMenu() throws SQLException, ParseException {
    int choice;
    boolean running = true;

    while (running) {
        System.out.println("Guest Menu:");
        System.out.println("1-View Available Rooms");
        System.out.println("2-Add Booking");
        System.out.println("3-Delete Booking");
        System.out.println("4-View My Bookings");
        System.out.println("5-Check In");
        System.out.println("6-Check Out");
        System.out.println("0-Exit");

        try {
            choice = input.nextInt();
            input.nextLine(); // Clear buffer
        } catch (Exception e) {
            System.out.println("Invalid input. Please enter a number.");
            input.nextLine();
            continue;
        }

        switch (choice) {
            case 1 -> user.viewAvailableRooms(hotelChoice);
            case 2 -> user.addBooking();
            case 3 -> user.deleteBooking();
```

3. Housekeeping Menu:

Housekeepers view and manage their assigned tasks, focusing on cleaning schedules.



```java
public void housekeepingMenu() {
    int choice;
    boolean running = true;

    while (running) {
        System.out.println("Housekeeping Menu:");
        System.out.println("1-View Pending Tasks");
        System.out.println("2-Update Task to Completed");
        System.out.println("3-View Cleaning Schedule");
        System.out.println("0-Exit");

        try {
            choice = input.nextInt();
            input.nextLine();
        } catch (Exception e) {
            System.out.println("Invalid input. Try again.");
            input.nextLine();
            continue;
        }

        switch (choice) {
            case 1 -> user.viewPendingTasks();
            case 2 -> user.updateTaskStatusToCompleted();
            case 3 -> user.viewCleaningSchedule();
            case 0 -> {
                System.out.println("Exiting housekeeping menu.");
                running = false;
            }
            default -> System.out.println("Invalid choice. Try again.");
        }
    }
}
```

Basically, the Menus (e.g., GuestMenu, ReceptionistMenu, HousekeepingMenu, AdministratorMenu) in the TerminalMenu class work in a similar way.

Each menu:

1. Displays role-specific options: Depending on the user type, the system shows options like managing bookings, viewing tasks, or generating reports.
2. Handles user input: The menus prompt the user to make a choice, validate the input, and prevent invalid actions.
3. Delegates operations: Each menu calls the corresponding methods in the UserDAO class to interact with the database and execute tasks.
4. Supports navigation: Users can return to the menu or exit after completing an action, ensuring smooth and error-free interaction.

UserDAO Class

The UserDAO class serves as the Data Access Object (DAO), facilitating database interactions for user-related and hotel management operations. It contains various methods to manage bookings, rooms, payments, and users in the system.

Constructors:

With All Attributes: Initializes a UserDAO instance with details like ssn, firstName, and role for user account creation. Ensures complete encapsulation of user data.

With Database Lookup: Retrieves user details from the database using an SSN. Throws UserNotFoundException if no match is found, ensuring robust error handling, making it ideal for login operations.

Methods:

- addBooking()

- Purpose: Inserts a new booking into the Booking table.

  Steps:

-Displays room types available for booking.

-Takes inputs such as roomTypeID, startDate, endDate, and people.

-Calls a stored procedure InsertBooking to handle booking conflicts and availability checks.

viewBookingOfUser()

- Purpose: Displays all bookings associated with the logged-in user's SSN.

Steps:

-Executes a query to fetch relevant bookings.

-Formats and displays booking details, including room ID, start/end times, and status.

- deleteBooking()

- Purpose: Deletes a booking specified by the user.

Steps:

-Prompts the user to select a booking ID.

-Validates and deletes the booking from the database.

- Design Consideration: The use of foreign key constraints ensures cascading effects for related records.

- modifyBooking()

- Purpose: Modifies an existing booking.

Options for Modification:

-Change room assignment.

-Change guest SSN.

-Update booking start/end times.

Steps: Executes parameterized SQL UPDATE statements based on the user's choice.

confirmBooking()

- Purpose: Updates the status of a booking to "Confirmed."

Steps: Updates the status column in the Booking table for the given booking ID.

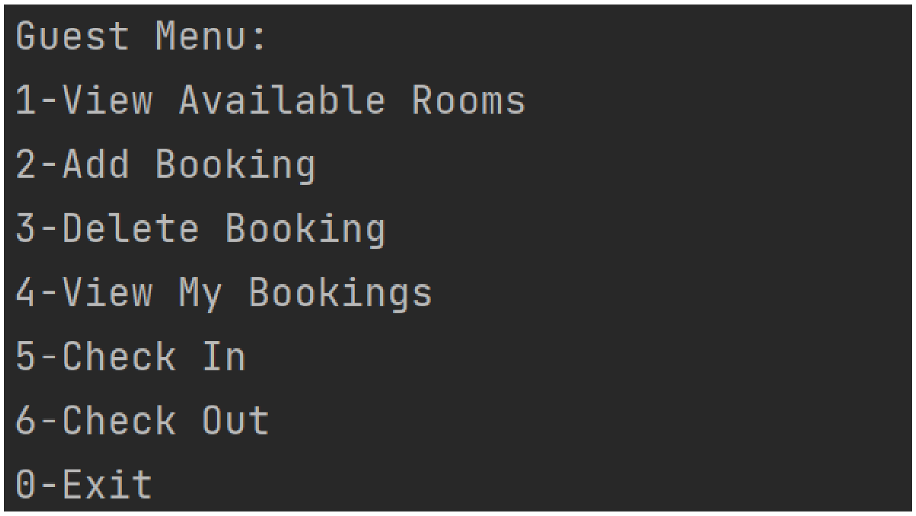
- viewBookingOfHotel()

- Purpose: Displays all bookings for a specific hotel.

Steps:

-Filters bookings by hotelID.

-Displays comprehensive booking details, including guest ID and room ID.



```
Guest Menu:
1-View Available Rooms
2-Add Booking
3-Delete Booking
4-View My Bookings
5-Check In
6-Check Out
0-Exit
```

-viewAvailableRooms(int hotelID)

- Purpose: Displays all rooms marked as "Available" for the specified hotel.

Steps:

-Fetches rooms based on hotelID and status = 'Available'.

-Displays room details like roomID, price, and location.

- viewRoomTypeByHotel()

- Purpose: Lists all room types available in a given hotel.

Steps:

-Fetches distinct room types for the hotelID.

-Displays type ID, room name, description, and price.

- processPayment()

- Purpose: Updates payment details for a specified booking.

Steps:

-Takes the bookingID and payment method as input.

-Marks the payment as "Completed" and updates the payment method and date.

- Flexibility: Allows multiple payment methods, including Cash, Credit Card, PayPal, etc.

-assignHousekeepingTask()

- Purpose: Assigns a housekeeping task to a specific housekeeper.

Steps:

-Takes inputs for the housekeeper's SSN, room ID, and task date.

-Parses the date and inserts a task into the HouseKeeping table.

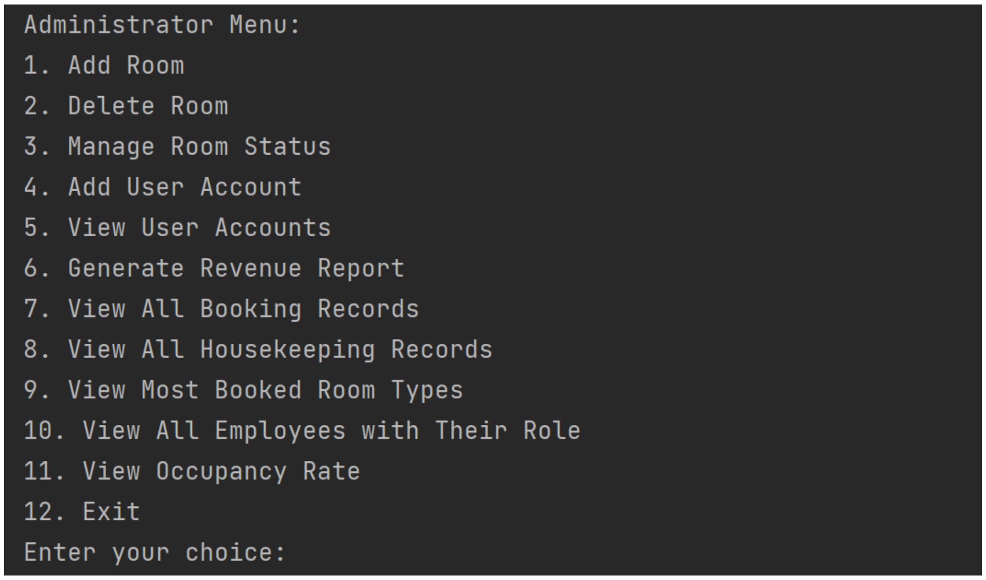
-viewAllHousekeepers()

- Purpose: Displays all housekeepers and their availability for the day.

Steps:

-Fetches data from the User and HouseKeeping tables.

- -Identifies availability based on whether a housekeeper is assigned tasks for the current day.
- Output: Displays housekeepers' details, including their status (Available/Unavailable).
- Why It’s Important: Provides a clear view of housekeeper resources for better task management.



```
Administrator Menu:
1. Add Room
2. Delete Room
3. Manage Room Status
4. Add User Account
5. View User Accounts
6. Generate Revenue Report
7. View All Booking Records
8. View All Housekeeping Records
9. View Most Booked Room Types
10. View All Employees with Their Role
11. View Occupancy Rate
12. Exit
Enter your choice:
```

addRoom();

- Purpose: Adds a new room to the hotel.,

Steps:

-Displays room types available in the hotel.

-Inserts a new room into the Room table with the status set to "Available."

deleteRoom()

- Purpose: Deletes a room from the hotel after validation.

  Steps:

  -Checks for active bookings tied to the room.

  -Deletes the room if no active bookings exist.

- Why It's Important: Prevents accidental deletion of rooms with ongoing reservations, maintaining data integrity.

manageRoomStatus()

- Purpose: Updates the status of a room (e.g., "Available" or "Occupied").

  Steps:

  -Takes room ID and new status as inputs.

  -Updates the Room table based on the given room ID.

- Why It's Important: Enables real-time management of room availability.

generateRevenueReport()

- Purpose: Generates a revenue report for the hotel.
  Key Steps:

  Aggregates booking data to calculate total revenue and bookings.

  Displays the report for the selected hotel.
  Why It's Important: Helps administrators track financial performance.

addUserAccount()

- Purpose: Creates a new user account in the system.
  Key Steps:

  Collects user details, including role and salary (if applicable).

  Inserts the user into the User table.
  Why It’s Important: Allows for the seamless addition of staff and guests to the system.

viewUserAccounts()

- Purpose: Displays all user accounts associated with a specific hotel.
  Key Steps:

  Retrieves and displays data from the User table.
  Why It’s Important: Provides administrators with an overview of staff and guests.

viewAllBookings()

- Purpose: Displays all booking records for a specific hotel.
  Key Steps:

  Fetches and displays data from the Booking table.

  Organized output includes booking ID, room ID, dates, status, and guest details.
  Why It’s Important: Provides transparency in booking management.

viewAllHousekeepingRecords()

- Purpose: Displays housekeeping task records for a hotel.
  Key Steps:

  Retrieves data from the HouseKeeping table.

  Shows task IDs, assigned housekeepers, task dates, and statuses.
  Why It’s Important: Facilitates tracking of housekeeping operations.

viewMostBookedRoomTypes()

- Purpose: Identifies the most booked room types for a hotel.
  Key Steps:

  Aggregates booking data by room type.

  Displays room type names and total bookings in descending order.
  Why It's Important: Helps administrators identify popular room types.

viewAllEmployeesWithRole()

- Purpose: Lists all employees and their roles in a hotel.
  Key Steps:

  Retrieves staff data from the User table.

  Displays SSNs, names, roles, and contact details.
  Why It's Important: Provides an overview of the workforce for effective management.

viewOccupancyRate() - **extra method**

- Purpose: Calculates and displays the occupancy rate for a hotel.
  Key Steps:

  Compares the number of occupied rooms to the total rooms.

  Outputs the percentage of occupied rooms.
  Why It's Important: Gives insights into room utilization and operational efficiency.

viewPendingTasks() and viewCompletedTasks()

- Purpose: Displays housekeeping tasks for a housekeeper based on their status.
  Key Steps:

  Fetches tasks from the HouseKeeping table based on the housekeeper's SSN.

Outputs pending or completed tasks in an organized format.
Why It’s Important: Helps housekeepers track their tasks effectively.

updateTaskStatusToCompleted()

- Purpose: Marks a housekeeping task as completed.
  Key Steps:

  Prompts the user to enter a task ID.

  Updates the status of the task in the HouseKeeping table to "Completed," but only if:

  The task belongs to the current housekeeper.

  The task is currently marked as "Pending."

  Feedback is provided based on whether the update was successful or not.
  Error Handling: Ensures proper SQL exception handling. Why It’s Important: Streamlines task management by allowing housekeepers to update their progress.

viewCleaningSchedule()

- Purpose: Displays the cleaning schedule for a housekeeper.
  Key Steps:

  Retrieves tasks assigned to the current housekeeper, sorted by date.

  Displays task details such as task ID, room ID, task date, and status.

  If no tasks are assigned, notifies the housekeeper.
  Why It’s Important: Provides housekeepers with a clear overview of their responsibilities.

checkIn()

- Purpose: Facilitates guest check-in.
  Key Steps:

Prompts the guest for their booking ID.

Verifies:

-The booking exists.

-The booking is confirmed.

-The guest hasn't already checked in.

Updates the status of the booking to "Checked-In."
Error Handling: Ensures guests cannot check-in without a confirmed booking or if they're already checked-in.

Why It's Important: Automates the check-in process, ensuring a smooth start to the guest's stay.

checkOut()

- Purpose: Facilitates guest check-out.
  Key Steps:

  Prompts the guest for their booking ID.

  Verifies: -Payment for the booking is completed.

  -The guest has checked in but hasn't already checked out.

  Updates the status of the booking to "Checked-Out."
  Error Handling: Prevents check-outs if:

  -Payment is pending.

  -The guest hasn't checked in yet.

  Why It's Important: Ensures a seamless check-out process while maintaining financial accountability.

By integrating these functionalities, the UserDAO class supports both back-end task handling and front-end user interactions. Now continuing with the Main Class.

Main Class

The Main class serves as the entry point for the hotel management system. Its primary purpose is to establish a database connection and initiate the interactive terminal menu. Here's an overview of its functionality:

General Purpose

- **Database Connection**: The Main class uses the DBConnection.getConnection() method to establish a connection to the system's database. It ensures that the system is ready to execute queries and retrieve or update data.
- **Menu Initialization**: It creates an instance of the TerminalMenu class, passing the database connection to allow seamless interaction between the terminal interface and the database.

Interaction with Other Classes

- **DBConnection**: Provides the database connection required for all subsequent operations.
- **TerminalMenu:** Once the database connection is established, the Main class invokes the hotelSelectionMenu() method of the TerminalMenu class. This starts the user interaction process, allowing users to choose a hotel and role (Guest, Receptionist, Housekeeping, Administrator).

Error Handling

The class includes mechanisms to handle common errors:

- **Database Connection Issues:** It gracefully handles errors like failed connections or incorrect database credentials using SQLException.

- **Input Errors:** Any issues related to parsing inputs (like date formats) are managed with a ParseException.

The Main class plays a foundational role in setting up the environment for the system. It ensures a robust start by:

1. Connecting to the database.
2. Delegating user interactions to the TerminalMenu. This streamlined design separates concerns and keeps the class focused on initialization and system setup, ensuring clarity and maintainability.

DBConnection Class:

The DBConnection class is responsible for managing the connection to the MySQL database used by the hotel management system.

General Purpose

- Database Connection Setup: This class contains the necessary credentials (URL, USER, and PASSWORD) for connecting to the Hotel_Management database.
- Connection Retrieval: The getConnection() method utilizes DriverManager.getConnection() to establish a connection to the database and returns it for use in other parts of the application.

Interaction with Other Classes

- Integration: The DBConnection class is used by the Main class to initiate the connection to the database before any menu operations or queries are executed in the system.

The DBConnection class centralizes the database connection logic, making it reusable and easily configurable, ensuring smooth interaction with the MySQL database throughout the system.

Conclusion

The hotel management system successfully streamlines the management of hotel operations through a user-friendly interface and a well-organized database structure. It accommodates different user roles such as guests, receptionists, housekeepers, and administrators, each with tailored functionalities. The system’s design ensures efficient handling of tasks such as room bookings, housekeeping assignments, and user management while maintaining data integrity and scalability. Overall, the project offers a solid solution for hotel management, emphasizing ease of use, flexibility, and reliable performance.