LINQ QUERIES IN HOTEL RESERVATION SYSTEM

```
var roomsQuery = _context.Rooms
    .Include(r => r.Hotel)
    .Include(r => r.Reservations)
    .Where(r => r.Status == RoomStatus.Available)
    .AsQueryable();
if (hotelId.HasValue)
{
    roomsQuery = roomsQuery.Where(r => r.HotelId == hotelId.Value);
}

//I'm excluding rooms with overlapping reservations i.e duplicate bookings
roomsQuery = roomsQuery.Where(r =>
    !r.Reservations.Any(res =>
        res.Status != ReservationStatus.Cancelled &&
        res.Status != ReservationStatus.CheckedOut &&
        checkIn < res.CheckOutDate &&
        checkOut > res.CheckInDate
    )
);
```

Reason:
Filters out rooms with overlapping reservations using a single efficient query via .where() and !Any() on date ranges.

```
public async Task<IEnumerable<RoomDto>> GetRecommendedRoomsAsync(string userId)
{
    // firstly i'm taking user's booking history i.e user reservations
    var history = await _context.Reservations
        .Include(r => r.Room)
        .Where(r => r.UserId == userId && (r.Status == ReservationStatus.CheckedOut || r.Status == ReservationStatus.Confirmed))
        .ToListAsync();

    if (!history.Any())
    {
        // if there are no reservations, i'll return some available rooms
        return await _context.Rooms
            .Include(r => r.Hotel)
            .Where(r => r.Status == RoomStatus.Available)
            .Take(5)
            .Select(r => new RoomDto
            {
                Id = r.Id,
                RoomNumber = r.RoomNumber,
                Type = r.Type,
                BasePrice = r.BasePrice,
                Capacity = r.Capacity,
                Status = r.Status,
                HotelId = r.HotelId,
                HotelName = r.Hotel.Name
            })
            .ToListAsync();
    }
```

Reason:
Analyzes user booking history using `.GroupBy()`, `.Sum()`, and `.OrderByDescending()` to identify preferences like room type and average spend.

```
// scoring available rooms
var availableRooms = await _context.Rooms
    .Include(r => r.Hotel)
    .Where(r => r.Status == RoomStatus.Available)
    .ToListAsync();

var recommendedRooms = availableRooms
    .Select(room => new
    {
        Room = room,
        Score = (room.Type == preferredType ? 10 : 0) +
                (Math.Abs(room.BasePrice - avgDailySpend) < 50 ? 5 :
                 Math.Abs(room.BasePrice - avgDailySpend) < 100 ? 2 : 0)
    })
    .OrderByDescending(x => x.Score)
    .Take(5)
    .Select(x => new RoomDto
    {
        Id = x.Room.Id,
        RoomNumber = x.Room.RoomNumber,
        Type = x.Room.Type,
        BasePrice = x.Room.BasePrice,
        Capacity = x.Room.Capacity,
        Status = x.Room.Status,
        HotelId = x.Room.HotelId,
        HotelName = x.Room.Hotel.Name
    })
```

Reason:
Calculates a custom match score per room using .Select() and ranks the best options with .OrderByDescending(score)

```
// Checking for double booking - not allowing duplicate reservations..
bool isRoomOccupied = await _context.Reservations
    .AnyAsync(r => r.RoomId == createReservationDto.RoomId
                && r.Status != ReservationStatus.Cancelled
                && r.Status != ReservationStatus.CheckedOut
                && createReservationDto.CheckInDate < r.CheckOutDate
                && createReservationDto.CheckOutDate > r.CheckInDate);

if (isRoomOccupied)
    throw new ArgumentException("Room is already booked for the selected dates.");
```

Reason:
.AnyAsync() is used to efficiently check for overlapping reservations before creating a new one, ensuring the same room isn't booked twice for the same dates.

```
2 references
public async Task<IEnumerable<ReservationDto>> GetAllReservationsAsync()
{
    return await _context.Reservations
        .Include(r => r.Room)
        .ThenInclude(r => r.Hotel)
        .Include(r => r.User)
        .Select(r => new ReservationDto
        {
            Id = r.Id,
            CheckInDate = r.CheckInDate,
            CheckOutDate = r.CheckOutDate,
            NumberOfGuests = r.NumberOfGuests,
            TotalAmount = r.TotalAmount,
            Status = r.Status,
            CreatedAt = r.CreatedAt,
            CheckedInAt = r.CheckedInAt,
            CheckedOutAt = r.CheckedOutAt,
            UserId = r.UserId,
            UserName = $"{r.User.FirstName} {r.User.LastName}",
            RoomId = r.RoomId,
            RoomNumber = r.Room.RoomNumber,
            HotelName = r.Room.Hotel.Name,
            HotelId = r.Room.HotelId
        })
```

Reason:
`.Select()` maps complex, related entities into a clean `ReservationDto`, exposing only required data and making it safe and easy for frontend consumption.

```
2 references
public async Task<IEnumerable<BillDto>> GetAllBillsAsync()
{
    return await _context.Bills
        .Include(b => b.Reservation)
        .ThenInclude(r => r.User)
        .Include(b => b.Reservation)
        .ThenInclude(r => r.Room)
        .Select(b => new BillDto
        {
            Id = b.Id,
            RoomCharges = b.RoomCharges,
            AdditionalCharges = b.AdditionalCharges,
            TaxAmount = b.TaxAmount,
            TotalAmount = b.TotalAmount,
            PaymentStatus = b.PaymentStatus,
            CreatedAt = b.CreatedAt,
            PaidAt = b.PaidAt,
            ReservationId = b.ReservationId,
            UserName = b.Reservation.User.UserName ?? "",
            RoomNumber = b.Reservation.Room.RoomNumber,
            HotelName = b.Reservation.Room.Hotel.Name,
            HotelId = b.Reservation.Room.HotelId
        })
```

Reason: For getting all the bills from the user reservations which are checked-out

```
6 references
public async Task<BillDto?> GetBillByIdAsync(int id)
{
    var bill = await _context.Bills
        .Include(b => b.Reservation)
        .ThenInclude(r => r.User)
        .Include(b => b.Reservation)
        .ThenInclude(r => r.Room)
        .ThenInclude(rm => rm.Hotel)
        .FirstOrDefaultAsync(b => b.Id == id);

    if (bill == null) return null;

    return new BillDto
    {
        Id = bill.Id,
        RoomCharges = bill.RoomCharges,
        AdditionalCharges = bill.AdditionalCharges,
        TaxAmount = bill.TaxAmount,
        TotalAmount = bill.TotalAmount,
        PaymentStatus = bill.PaymentStatus,
        CreatedAt = bill.CreatedAt,
        PaidAt = bill.PaidAt,
        ReservationId = bill.ReservationId,
        UserName = bill.Reservation.User.UserName ?? "",
```

Reason:
`.Where(b => b.Reservation.UserId == userId)` ensures only bills belonging to the logged-in user are fetched, maintaining data security.

```
2 references
public async Task<bool> UserOwnsBillAsync(string userId, int billId)
{
    return await _context.Bills
        .Include(b => b.Reservation)
        .AnyAsync(b => b.Id == billId && b.Reservation.UserId == userId);
}
```

Reason:
`.Include(b => b.Reservation)` loads related reservation data in a single query, avoiding the N+1 query problem and improving performance.

```
public async Task<ApiResponse<bool>> DeleteUserAsync(string userId)
{
    var user = await _userManager.FindByIdAsync(userId);
    if (user == null)
        return new ApiResponse<bool> { Success = false, Message = "User not found" };

    // user cannot be deleted if he has active reservations
    var hasActiveReservations = await _context.Reservations
        .AnyAsync(r => r.UserId == userId &&
                      (r.Status == ReservationStatus.Booked ||
                       r.Status == ReservationStatus.Confirmed ||
                       r.Status == ReservationStatus.CheckedIn));

    if (hasActiveReservations)
        return new ApiResponse<bool>
        {
            Success = false,
            Message = "Cannot delete user with active reservations"
        };
```

Reason:
Uses .AnyAsync() to check if an email already exists in the database before allowing registration.

```
2 references
public async Task<IEnumerable<SeasonalRateDto>> GetRatesByHotelAsync(int hotelId)
{
    return await _context.SeasonalRates
        .Where(s => s.HotelId == hotelId)
        .OrderBy(s => s.StartDate)
        .Select(s => new SeasonalRateDto
        {
            Id = s.Id,
            Name = s.Name,
            StartDate = s.StartDate,
            EndDate = s.EndDate,
            Multiplier = s.Multiplier,
            HotelId = s.HotelId
        })
        .ToListAsync();
}
```

Reason:
Used for fetching the existing seasonal prices for the particular Hotel (hotelId).