LINQ USAGE IN HEALTH INSURANCE MANAGEMENT SYSTEM

1. In ClaimService Uses LINQ with eager loading (`Include`) and predicate-based filtering (`FirstOrDefaultAsync`) to efficiently retrieve claim details while avoiding N+1 queries.

```
3 references
public async Task<ClaimDto?> GetClaimByIdAsync(int claimId)
{
    var claim = await _context.Claims
        .Include(c => c.User)
        .Include(c => c.Policy)
        .Include(c => c.Hospital)
        .Include(c => c.Reviewer)
        .AsNoTracking()
        .FirstOrDefaultAsync(c => c.ClaimId == claimId);

    if (claim == null) return null;

    var claimDtos = await MapClaimsToDtosAsync(new[] { claim });
    return claimDtos.FirstOrDefault();
}
```

2.In ClaimService pending review query, LINQ constructs a precise business logic filter: Where with complex conditions: Where retrieves only claims that are "InReview" with medical notes OR "Approved" status, ensuring claims officers see exactly the claims requiring their attention without loading irrelevant data from the database.

```
2 references
public async Task<IEnumerable<ClaimDto>> GetClaimsPendingReviewAsync()
{
    var claims = await _context.Claims
        .Include(c => c.User)
        .Include(c => c.Policy)
        .Include(c => c.Hospital)
        .Include(c => c.Reviewer)
        .Where(c => (c.Status == ClaimStatus.InReview && !string.IsNullOrEmpty(c.MedicalNotes)) ||
                    c.Status == ClaimStatus.Approved)
        .AsNoTracking()
        .ToListAsync();
    return await MapClaimsToDtosAsync(claims);
}
```

3.In ClaimService number generation, LINQ is used for count-based and existence-check operations during claim number generation. `CountAsync` with a predicate filters claims by submission year to generate sequential claim numbers, while `AnyAsync` verifies uniqueness to prevent duplicates. A fallback timestamp-based strategy ensures reliability under high

concurrency.

```
do
{
    var count = await _claimRepository.CountAsync(c => c.SubmittedAt.Year == year);
    claimNumber = $"CLM-{year}-{(count + 1 + attempts):D3}";
    attempts++;

    if (attempts > maxAttempts)
    {
        // Fallback to timestamp-based generation
        var timestamp = DateTimeOffset.UtcNow.ToUnixTimeSeconds();
        claimNumber = $"CLM-{year}-{timestamp.ToString().Substring(timestamp.ToString().Length - 6)}";
        break;
    }
}
while (await _context.Claims.AnyAsync(c => c.ClaimNumber == claimNumber));

return claimNumber;
```

5.In ClaimService mapping operations,LINQ is used during the final transformation stage to sort mapped `ClaimDto` objects using `OrderByDescending`. After manually mapping entity data to DTOs, LINQ ensures consistent ordering of results by claim number, enabling the most recent claims to appear first in the user interface.

```
    foreach (var claim in claims)
    {
        var claimDto = _mapper.Map<ClaimDto>(claim);
        claimDto.PolicyNumber = claim.Policy?.PolicyNumber ?? "Unknown";
        claimDto.UserId = claim.UserId;
        claimDto.UserName = claim.User != null ? $"{claim.User.FirstName} {claim.User.LastName}" : "Unknown";
        claimDto.HospitalName = claim.Hospital?.HospitalName ?? "Unknown";
        claimDto.MedicalNotes = claim.MedicalNotes;
        claimDto.ReviewerName = claim.Reviewer != null ? $"{claim.Reviewer.FirstName} {claim.Reviewer.LastName}" : null;

        claimDtos.Add(claimDto);
    }

    return claimDtos.OrderByDescending(c => c.ClaimNumber);
}
```

6.In Dashboard statistics calculation, LINQ serves as real-time aggregation:
CountAsync & Where: CountAsync with Where conditions calculates unread notification counts and claim status statistics directly in the database, while frontend filtering uses Where to count active policies and claims by status for role-specific dashboard metrics.

```
public async Task<ActionResult<int>> GetUnreadCount()
{
    try
    {
        var userId = int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value ??
            throw new UnauthorizedAccessException("User ID not found"));

        var count = await _context.NotificationHistories
            .CountAsync(n => n.UserId == userId && !n.IsRead);

        return Ok(count);
    }
```

7.In NotificationsController LINQ performs efficient user-specific queries:
Where & OrderByDescending & Take: Where filters notifications by userId to ensure data privacy, OrderByDescending sorts by creation date for chronological display, and Take limits results to the 50 most recent notifications, minimizing memory usage and improving performance.

```
0 references
public async Task<ActionResult<IEnumerable<object>>> GetMyNotifications()
{
    try
    {
        var userId = int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value ??
            throw new UnauthorizedAccessException("User ID not found"));

        var notifications = await _context.NotificationHistories
            .Where(n => n.UserId == userId)
            .OrderByDescending(n => n.CreatedAt)
            .Take(50)
            .Select(n => new
            {
                n.NotificationId,
                n.Type,
                n.Title,
                n.Message,
                n.IsRead,
                n.CreatedAt,
                n.PolicyId,
                n.ClaimId
            })
            .ToListAsync();

        return Ok(notifications);
    }
```