Backend LINQ Usage Report

This document outlines all LINQ (Language Integrated Query) usages in the Backend repositories of the **Smart Ticket System**. The queries are primarily used for data retrieval, filtering, eager loading, and projections using Entity Framework Core.

Summary of Usage

- **Entity Framework Core**: Used extensively with `ToListAsync`, `FirstOrDefaultAsync`, `FindAsync`, and `AnyAsync`.
- **Filtering**: `.Where()` is the primary method for filtering records based on IDs, roles, status, and relationships.
- **Projections**: `.Select()` is used for projecting specific fields (e.g., user IDs).
- **Ordering**: `.OrderByDescending()` is used for sorting by date fields.
- **Pagination**: `.Skip()` and `.Take()` are used in notification queries.
- **Query Optimization**: `.AsNoTracking()` is used for read-only queries.

Detailed LINQ Queries by Repository

1. AgentRepository.cs

Location: `SmartTicketSystem.Infrastructure.Repositories/AgentRepository.cs`

Method / Context	LINQ Operators Used	Description
`GetByIdAsync`	`Include`, `ThenInclude`, `FirstOrDefaultAsync`	Fetches agent profile by ID with optional skills and categories.
`AddProfileAsync`	`AddAsync`	Adds a new agent profile to the context.
`AddSkillAsync`	`AddAsync`	Adds a new agent category skill to the context.
`GetByUserIdAsync`	`Include`, `ThenInclude`, `FirstOrDefaultAsync`	Fetches agent profile by user ID with optional eager loading.
`GetAllAsync`	`Include`, `ThenInclude`, `ToListAsync`	Retrieves all agent profiles with optional skills.

`CategoryExistsAsync`	`AnyAsync`	Checks if a category exists by ID.
`GetCategoriesByIdsAsync`	`Where`, `Contains`, `ToListAsync`	Fetches multiple categories by their IDs.

Code Snippet (`GetByIdAsync`):

```
IQueryable<AgentProfile> query = _context.AgentProfiles;

if (includeSkills)
{
    query = query
        .Include(a => a.Skills)
        .ThenInclude(s => s.Category);
}
return await query.FirstOrDefaultAsync(a => a.Id == agentProfileId);
```

Eager Loading (using `.Include()`) is a double-edged sword. While it prevents the "N+1 Query Problem," it can significantly increase the "Data Payload" and "Query Execution Time". By using the `bool includeSkills = false` pattern in Repository ,I am implementing Conditional Eager Loading.

```
    Task<AgentProfile?> GetByIdAsync(Guid agentProfileId, bool includeSkills = false);
    Task<AgentProfile?> GetByUserIdAsync(Guid userId, bool includeSkills = false);
    Task<IReadOnlyList<AgentProfile>> GetAllAsync(bool includeSkills = false);
```

---

2. AuthRepository.cs

Location: `SmartTicketSystem.Infrastructure.Repositories/AuthRepository.cs`

Method / Context	LINQ Operators Used	Description
`GetByEmailWithRole`	`Include`, `ThenInclude`, `FirstOrDefaultAsync`	Fetches user by email with role and profile information.
`GetByEmail`	`Include`, `FirstOrDefaultAsync`	Retrieves user by email with profile.

`GetByIdAsync`	`FindAsync`	Finds user by primary key (optimized lookup).
`AddAgentProfile`	`AddAsync`	Adds agent profile to context.
`AddUser`	`AddAsync`	Adds new user to context.
`GetUserIdsByRoles`	`Where`, `Contains`, `Select`, `ToListAsync`	Fetches user IDs that belong to specified roles.

Code Snippet (`GetUserIdsByRoles`):

```
return await _context.Users
    .Where(u => u.UserRole != null && roleIds.Contains(u.UserRole.RoleId))
    .Select(u => u.Id)
    .ToListAsync();
```

---

3. NotificationRepository.cs

Location:
`SmartTicketSystem.Infrastructure.Repositories/NotificationRepository.cs`

Method / Context	LINQ Operators Used	Description
`GetByIdAsync`	`FindAsync`	Retrieves notification by primary key.
`GetUnreadByUserIdAsync`	`Where`, `OrderByDescending`, `ToListAsync`	Fetches unread notifications for a user, sorted by creation date.
`GetByUserIdAsync`	`Where`, `OrderByDescending`, `Skip`, `Take`, `ToListAsync`	Retrieves paginated notifications for a user.
`AddAsync`	`AddAsync`	Adds notification to context.
`UpdateAsync`	`Update`	Marks notification entity as modified.

Code Snippet (`GetByUserIdAsync` - Pagination):

```
return await _context.Notifications
```

```
    .Where(n => n.UserId == userId)
    .OrderByDescending(n => n.CreatedAt)
    .Skip((pageNumber - 1) * pageSize)
    .Take(pageSize)
    .ToListAsync();
```

---

4. TicketCommentRepository.cs

Location:
`SmartTicketSystem.Infrastructure.Repositories/TicketCommentRepository.cs`

Method / Context	LINQ Operators Used	Description
`AddCommentAsync`	`AddAsync`	Adds new comment to context.
`GetCommentsByTicketAsync`	`Include`, `Where`, `OrderByDescending`, `ToListAsync`	Fetches all comments for a ticket with user info, ordered by comment ID descending.
`GetByIdAsync`	`FirstOrDefaultAsync`	Retrieves specific comment by ID.
`UpdateCommentAsync`	`Update`	Updates comment entity.

Code Snippet (`GetCommentsByTicketAsync`):

```
await _context.TicketComments
    .Include(c => c.User)
    .Where(c => c.TicketId == ticketId)
    .OrderByDescending(c => c.CommentId)
    .ToListAsync();
```

---

5. TicketHistoryRepository.cs

Location:
`SmartTicketSystem.Infrastructure.Repositories/TicketHistoryRepository.cs`

Method / Context	LINQ Operators Used	Description
`AddAsync`	`AddAsync`	Adds history record to context.
`GetByTicketIdAsync`	`Include`, `Where`, `OrderByDescending`, `ToListAsync`	Fetches ticket history with user information, sorted by change date.

Code Snippet (`GetByTicketIdAsync`):

```
await _context.TicketHistories
    .Include(h => h.User)
    .Where(h => h.TicketId == ticketId)
    .OrderByDescending(h => h.ChangedAt)
    .ToListAsync();
```

---

6. TicketPriorityRepository.cs

Location:
`SmartTicketSystem.Infrastructure.Repositories/TicketPriorityRepository.cs`

Method / Context	LINQ Operators Used	Description
`GetAllAsync`	`AsNoTracking`, `ToListAsync`	Retrieves all priorities without tracking (read-only optimization).
`GetPriorityByIdAsync`	`FirstOrDefaultAsync`	Fetches single priority by ID.
`UpdateAsync`	`Update`	Marks priority as modified.

Code Snippet (`GetAllAsync`):

```
await _context.TicketPriorities.AsNoTracking().ToListAsync();
```

---

7. TicketRepository.cs

Location: `SmartTicketSystem.Infrastructure.Repositories/TicketRepository.cs`

Method / Context	LINQ Operators Used	Description
`AddAsync`	`AddAsync`	Adds new ticket to context.
`GetByIdAsync`	`Include` (multiple), `FirstOrDefaultAsync`	Fetches ticket with all related entities (Category, Priority, Status, Owner, AssignedTo).
`GetByOwnerIdAsync`	`Include` (multiple), `Where`, `ToListAsync`	Retrieves all tickets owned by a specific user.
`GetByAssignedToAsync`	`Include` (multiple), `Where`, `ToListAsync`	Fetches tickets assigned to a specific agent.
`GetUnassignedAsync`	`Include` (multiple), `Where`, `ToListAsync`	Retrieves unassigned tickets (where AssignedToId is null).
`UpdateAsync`	`Update`	Marks ticket as modified.
`DeleteAsync`	`Remove`	Marks ticket for deletion.
`GetAllTicketsAsync`	`Include` (multiple), `ToListAsync`	Fetches all tickets with related entities.

Code Snippet (`GetByIdAsync`):

```
return await _context.Tickets
    .Include(t => t.Category)
    .Include(t => t.Priority)
    .Include(t => t.Status)
    .Include(t => t.Owner)
    .Include(t => t.AssignedTo)
    .FirstOrDefaultAsync(t => t.TicketId == ticketId);
```

---

8. UserProfileRepository.cs

Location:
`SmartTicketSystem.Infrastructure.Repositories/UserProfileRepository.cs`

Method / Context	LINQ Operators Used	Description
`GetByUserIdAsync`	`Include`, `FirstOrDefaultAsync`	Fetches user profile by user ID with user entity.
`AddAsync`	`AddAsync`	Adds new profile to context.
`UpdateAsync`	`Update`	Marks profile as modified.

Code Snippet (`GetByUserIdAsync`):

```
return await _context.UserProfiles
    .Include(p => p.User)
    .FirstOrDefaultAsync(p => p.UserId == userId);
```

---

9. UserRepository.cs

Location: `SmartTicketSystem.Infrastructure.Repositories/UserRepository.cs`

Method / Context	LINQ Operators Used	Description
`GetAllUsersAsync`	`Include`, `ThenInclude`, `ToListAsync`	Retrieves all users with their roles.
`GetByIdAsync`	`Include`, `FirstOrDefaultAsync`	Fetches user by ID with role information.
`GetSupportAgentsAsync`	`AsQueryable`, LINQ Query Syntax (`from`/`join`), `Where`, `Any`, `Include`, `ThenInclude`, `ToListAsync`	Complex query that fetches support agents with optional category filtering using query syntax joins and method syntax filtering.
`GetInactiveUsersAsync`	`Include`, `ThenInclude`, `Where`, `ToListAsync`	Retrieves inactive users with role information.

`UpdateAsync`	`Update`	Marks user as modified.

Code Snippet (`GetSupportAgentsAsync`):

```
var query = _context.Users.AsQueryable();

query = from user in query
        join userRole in _context.UserRoles on user.Id equals userRole.UserId
        join role in _context.Roles on userRole.RoleId equals role.RoleId
        where role.RoleName == "SupportAgent"
        select user;

if (categoryId.HasValue)
{
    query = query.Where(u => u.AgentProfiles.Any(ap =>
        ap.Skills.Any(s => s.CategoryId == categoryId.Value)
    ));
}

return await query
    .Include(u => u.AgentProfiles)
    .ThenInclude(ap => ap.Skills)
    .ToListAsync();
```

Key Patterns & Best Practices

1. **Repository Pattern**: All data access is encapsulated in repository classes, separating concerns from business logic.
2. **Async/Await**: All database operations use async methods for better scalability.
3. **Eager Loading Strategy**: Multiple `.Include()` calls are used to prevent N+1 query problems when related data is needed.
4. **Conditional Eager Loading**: Some methods (like `GetByIdAsync` in AgentRepository) use boolean flags to conditionally include related data.
5. **Read Optimization**: `.AsNoTracking()` is used in TicketPriorityRepository for read-only scenarios.
6. **Pagination Support**: NotificationRepository implements proper pagination with `.Skip()` and `.Take()`.
7. **Query Syntax vs Method Syntax**: UserRepository's `GetSupportAgentsAsync` demonstrates LINQ query syntax for complex joins, while most other queries use method syntax.

8. **Null Safety**: Nullable return types (?) are used appropriately for single-entity queries.
