



# Utility Billing System – LINQ Usage Overview

---

## 1. Meter Reading Management (Billing Officer)

### Feature area:

- **Module:** Meter Readings
- **Role:** Billing Officer
- **Main service:** `MeterReadingService`

### 1.1. Find Connections That Need Readings for Current Billing Period

File: `Services/Implementations/MeterReadingService.cs`

Method: `GetConnectionsNeedingReadingsAsync`

### Purpose:

- Get all active connections whose consumers and tariffs are valid.
- Check if a meter reading already exists for the current period.
- If not, compute the previous reading and return them as "pending" for data entry.

### Representative LINQ:

```
// Get all active connections with active (non-deleted) users and their utility types and billing cycles
var allConnections = await _context.Connections
    .Where(c => c.Status == "Active" && c.User.Status == "Active")
    .Include(c => c.User)
    .Include(c => c.UtilityType)
    .ThenInclude(ut => ut!.BillingCycle)
```

```

.Include(c => c.Tariff)
.ToListAsync();

// For each connection, check if a reading exists in the current billing period
var existingReading = await _context.MeterReadings
    .Where(mr => mr.ConnectionId == connection.Id &&
        mr.BillingCycleId == billingCycle.Id &&
        mr.ReadingDate >= periodStart &&
        mr.ReadingDate <= periodEnd)
    .FirstOrDefaultAsync();

// Get last billed reading to compute previous reading
var previousReading = await _context.MeterReadings
    .Where(mr => mr.ConnectionId == connection.Id && mr.Status == "Billed")
    .OrderByDescending(mr => mr.ReadingDate)
    .Select(mr => mr.CurrentReading)
    .FirstOrDefaultAsync();

```

#### Used by:

- **Frontend component:** "Connections needing readings" grid on the **Meter Reading Entry** page for Billing Officers.

## 1.2. Paginated & Filtered Meter Reading History

File: `MeterReadingService.cs`

Method: `GetMeterReadingHistoryAsync`

#### Purpose:

- Supports filtering by date range, utility type, consumer name, and status.
- Returns paged results sorted with "ReadyForBilling" on top.

#### Representative LINQ:

```

var query = _context.MeterReadings
    .Include(mr => mr.Connection)
    .ThenInclude(c => c.User)
    .Include(mr => mr.Connection)

```

```

        .ThenInclude(c => c.UtilityType)
        .Include(mr => mr.Tariff)
        .AsQueryable();

if (startDate.HasValue)
    query = query.Where(mr => mr.ReadingDate >= startDate.Value);

if (endDate.HasValue)
    query = query.Where(mr => mr.ReadingDate <= endDate.Value);

if (!string.IsNullOrEmpty(utilityTypeId))
    query = query.Where(mr => mr.Connection.UtilityTypeId == utilityTypeId);

if (!string.IsNullOrEmpty(consumerName))
    query = query.Where(mr => mr.Connection.User.FullName.Contains(consumerName));

if (!string.IsNullOrEmpty(status))
    query = query.Where(mr => mr.Status == status);

var readings = await query
    .OrderBy(mr => mr.Status == "ReadyForBilling" ? 0 : 1)
    .ThenByDescending(mr => mr.ReadingDate)
    .Skip((page - 1) * pageSize)
    .Take(pageSize)
    .ToListAsync();

```

#### Used by:

- **Frontend component: Meter Reading History** page (Billing Officer) with filters and pagination.

## 2. Billing & Invoice Generation (Billing Officer)

#### Feature area:

- **Module:** Billing

- **Role:** Billing Officer
- **Main service:** `BillService`

## 2.1. Identify Readings Ready for Billing (Pending Bills List)

File: `BillService.cs`

Method: `GetPendingBillsAsync`

### Purpose:

- Gather all meter readings with status "ReadyForBilling".
- Ensure billing cycle is active and in the correct period.
- Ensure a bill does not already exist for that connection and billing period.

### Representative LINQ:

```
var readyReadings = await _context.MeterReadings
    .Where(mr => mr.Status == "ReadyForBilling")
    .Include(mr => mr.Connection)
    .ThenInclude(c => c.User)
    .Include(mr => mr.Connection)
    .ThenInclude(c => c.UtilityType)
    .Include(mr => mr.Tariff)
    .Include(mr => mr.BillingCycle)
    .ToListAsync();

// Check for existing bill for the same connection and billing period
var existingBill = await _context.Bills
    .Where(b => b.ConnectionId == reading.ConnectionId &&
        b.BillingPeriod == billingPeriod)
    .FirstOrDefaultAsync();
```

### Used by:

- **Frontend component:** **Pending Bills** table, from where Billing Officer can generate invoices.

## 2.2. Retrieve Bills for a Connection / Consumer

File: `BillService.cs`

Methods:

- `GetBillsByConnectionAsync`
- `GetBillsForUserAsync`
- `GetBillForUserByIdAsync`

**Purpose:**

- Load all bills for a connection or specific consumer.
- Keep status (Generated / Due / Overdue) consistent using in-memory update based on current date.
- Sort by most recent generation date.

**Representative LINQ:**

```
var bills = await _context.Bills
    .Where(b => b.ConnectionId == connectionId)
    .Include(b => b.Connection)
    .ThenInclude(c => c.User)
    .Include(b => b.Connection)
    .ThenInclude(c => c.UtilityType)
    .ThenInclude(u => u!.BillingCycle)
    .OrderByDescending(b => b.GenerationDate)
    .ToListAsync();
```

```
// For consumer-specific listing
var userBills = await _context.Bills
    .Include(b => b.Connection)
    .ThenInclude(c => c.User)
    .Include(b => b.Connection)
    .ThenInclude(c => c.UtilityType)
    .ThenInclude(u => u!.BillingCycle)
    .Where(b => b.Connection.UserId == userId)
```

```
.OrderByDescending(b => b.GenerationDate)
.ToListAsync();
```

**Used by:**

- **Frontend components:**
  - Bills list under **Bills & Invoices** for Billing Officer.
  - Consumer's own **Bills** page.

### 2.3. Count Due/Overdue Bills for Consumer (Dashboard Badge)

File: `BillService.cs`

Method: `GetDueBillsCountForUserAsync`

**Purpose:**

- Count how many bills for a consumer are currently Due or Overdue.

**Representative LINQ:**

```
var bills = await _context.Bills
    .Include(b => b.Connection)
    .ThenInclude(c => c.UtilityType)
    .ThenInclude(u => u!.BillingCycle)
    .Where(b => b.Connection.UserId == userId && b.Status != "Paid")
    .ToListAsync();

var dueCount = bills.Count(b => b.Status == "Due" || b.Status == "Overdue");
```

**Used by:**

- **Frontend component:** Consumer **Dashboard** card showing "Due Bills" count.

---

## 3. Payment Tracking (Account Officer & Consumer)

**Feature area:**

- **Module:** Payments
- **Roles:** Consumer, Account Officer

- **Main service:** `PaymentService`

### 3.1. Payment History with Filters (by date & utility type)

File: `PaymentService.cs`

Method: `GetPaymentHistoryForUserAsync`

#### **Purpose:**

- Allow a consumer to filter payment history by date range and utility type.

#### **Representative LINQ:**

```
var query = _context.Payments
    .Include(p => p.Bill)
    .ThenInclude(b => b.Connection)
    .ThenInclude(c => c.UtilityType)
    .Include(p => p.Bill)
    .ThenInclude(b => b.Connection)
    .ThenInclude(c => c.User)
    .Where(p => p.Bill.Connection.UserId == userId)
    .AsQueryable();

if (startDate.HasValue)
    query = query.Where(p => p.PaymentDate >= startDate.Value);

if (endDate.HasValue)
    query = query.Where(p => p.PaymentDate <= endDate.Value);

if (!string.IsNullOrEmpty(utilityTypeId))
    query = query.Where(p => p.Bill.Connection.UtilityTypeId == utilityTypeId);

var payments = await query
    .OrderByDescending(p => p.PaymentDate)
    .ToListAsync();
```

#### **Used by:**

- **Frontend component: Payment History** page for consumers with date and utility filters.

### 3.2. Compute Outstanding Balance for Consumer

File: `PaymentService.cs`

Method: `GetOutstandingBalanceForUserAsync`

#### Purpose:

- Sum the **TotalAmount** of all unpaid bills (Generated, Due, Overdue) for a given user.

#### Representative LINQ:

```
var outstanding = await _context.Bills
    .Include(b => b.Connection)
    .ThenInclude(c => c.UtilityType)
    .ThenInclude(u => u!.BillingCycle)
    .Include(b => b.Connection)
    .ThenInclude(c => c.Tariff)
    .Where(b => b.Connection.UserId == userId && b.Status != "Paid")
    .ToListAsync();

var totalOutstanding = outstanding.Sum(b => b.TotalAmount);
```

#### Used by:

- **Frontend component: Consumer Dashboard** and/or Account Officer summaries showing outstanding dues per consumer.

### 3.3. Monthly Spending (Bills Generated in a Month)

File: `PaymentService.cs`

Method: `GetMonthlySpendingForUserAsync`

#### Purpose:

- Sum all bill amounts generated for a user in a given month (paid + unpaid).

#### Representative LINQ:

```

var monthStart = new DateTime(monthDateUtc.Year, monthDateUtc.Month, 1,
0, 0, 0, DateTimeKind.Utc);
var monthEnd = monthStart.AddMonths(1);

var bills = await _context.Bills
    .Include(b => b.Connection)
    .ThenInclude(c => c.UtilityType)
    .ThenInclude(u => u!.BillingCycle)
    .Include(b => b.Connection)
    .ThenInclude(c => c.Tariff)
    .Where(b => b.Connection.UserId == userId &&
        b.GenerationDate >= monthStart &&
        b.GenerationDate < monthEnd)
    .ToListAsync();

var monthlySpending = bills.Sum(b => b.TotalAmount);

```

#### Used by:

- **Frontend components:** Consumer **Dashboard** ("This month's spending"), Account Officer analysis.

### 3.4. Monthly Consumption Trend per Consumer

File: `PaymentService.cs`

Method: `GetMonthlyConsumptionForUserAsync`

#### Purpose:

- Group meter readings by year/month and sum consumption to build a monthly consumption trend.

#### Representative LINQ:

```

var readings = await _context.MeterReadings
    .Include(mr => mr.Connection)
    .Where(mr => mr.Connection.UserId == userId)
    .ToListAsync();

```

```

var grouped = readings
    .GroupBy(mr => new { mr.ReadingDate.Year, mr.ReadingDate.Month })
    .OrderBy(g => g.Key.Year)
    .ThenBy(g => g.Key.Month)
    .Select(g => new
    {
        MonthLabel = $"{g.Key.Year}-{g.Key.Month:00}",
        TotalConsumption = g.Sum(x => x.Consumption)
    });

```

#### Used by:

- **Frontend components:**
  - Consumer **Dashboard** consumption chart.
  - Consumer self-service **Reports** or usage graphs.

## 4. Reports & Dashboards

### Feature areas:

- **Modules:** Reports, Dashboards
- **Roles:** Admin, Account Officer, Consumer
- **Main services:** `ReportService` , `AccountOfficerService` , `DashboardMetricsHelper`

### 4.1. Admin Summary of Active Users & Pending Requests

File: `ReportService.cs`

Method: `GetReportSummaryAsync`

#### Purpose:

- Count active Billing Officers, Account Officers, Consumers, and pending utility requests.

#### Representative LINQ (role-based counts):

```

var activeBillingOfficers = await _context.Users
    .Join(_context.UserRoles,
        u => u.Id,
        ur => ur.UserId,
        (u, ur) => new { u, ur.RoleId })
    .Join(_context.Roles,
        x => x.RoleId,
        r => r.Id,
        (x, r) => new { x.u, r.Name })
    .Where(x => x.Name == "Billing Officer" && x.u.Status == "Active")
    .CountAsync();

var pendingUtilityRequests = await _context.UtilityRequests
    .Where(ur => ur.Status == "Pending")
    .CountAsync();

```

#### Used by:

- **Frontend component: Admin Dashboard** tiles (active officers, total consumers, pending requests).

## 4.2. Overdue Bills Listing

File: `ReportService.cs`

Method: `GetOverdueBillsAsync`

#### Purpose:

- Load all unpaid bills, update their status in memory, then project only those that are Overdue.

#### Representative LINQ (projection and ordering):

```

var unpaidBills = await _context.Bills
    .Where(b => b.Status != "Paid")
    .Include(b => b.Connection)
    .ThenInclude(c => c.User)
    .Include(b => b.Connection)

```

```

        .ThenInclude(c => c.UtilityType)
        .ThenInclude(u => u!.BillingCycle)
        .Include(b => b.Connection)
        .ThenInclude(c => c.Tariff)
        .ToListAsync();

// After status recalculation
return unpaidBills
    .Where(b => b.Status == "Overdue")
    .Select(b => new OverdueBillDto
    {
        BillId = b.Id,
        ConsumerName = b.Connection.User.FullName,
        UtilityName = b.Connection.UtilityType.Name,
        Amount = b.TotalAmount,
        DueDate = b.DueDate
    })
    .OrderBy(o => o.DueDate)
    .ToList();

```

#### Used by:

- **Frontend component: Overdue Bills Report** (Account Officer / Admin).

### 4.3. Consumption by Utility Type (Aggregation)

File: `ReportService.cs`

Method: `GetConsumptionByUtilityAsync`

#### Purpose:

- Join bills with connections and utility types.
- Group by utility name and sum consumption.

#### Representative LINQ:

```

var data = await _context.Bills
    .Join(_context.Connections,
        b => b.ConnectionId,

```

```

        c => c.Id,
        (b, c) => new { b.Consumption, c.UtilityTypeId })
.Join(_context.UtilityTypes,
    x => x.UtilityTypeId,
    ut => ut.Id,
    (x, ut) => new { x.Consumption, ut.Name })
.GroupBy(x => x.Name)
.Select(g => new ConsumptionDataDto
{
    UtilityName = g.Key,
    Consumption = g.Sum(x => x.Consumption)
})
.OrderByDescending(c => c.Consumption)
.ToListAsync();

```

#### Used by:

- **Frontend component: Reports** → "Consumption by Utility" chart or table.

#### 4.4. Monthly Revenue Report (Aggregation)

Files & methods:

- `ReportService.GetMonthlyRevenueAsync()`
- `AccountOfficerService.GetMonthlyRevenueByBillingDateAsync(...)`

#### Purpose:

- Group completed payments by year/month and sum payment amounts to compute monthly revenue.

#### Representative LINQ:

```

var monthlyRevenueData = await _context.Payments
    .Where(p => p.Status == "Completed")
    .GroupBy(p => new { p.PaymentDate.Year, p.PaymentDate.Month })
    .Select(g => new
    {
        Year = g.Key.Year,

```

```

        Month = g.Key.Month,
        Revenue = g.Sum(p => p.Amount)
    })
    .OrderBy(x => x.Year)
    .ThenBy(x => x.Month)
    .ToListAsync();

```

#### Used by:

- **Frontend components:**
  - Admin / Account Officer **Monthly Revenue** chart.
  - Reports → "Revenue per Month".

### 4.5. Average Consumption per Utility Type

File: `ReportService.cs`

Method: `GetAverageConsumptionAsync`

#### Purpose:

- Use bills joined with connections and utility types.
- Filter out zero-consumption bills, then compute **Average** per utility.

#### Representative LINQ:

```

var results = await _context.Bills
    .Where(b => b.Consumption > 0)
    .Join(_context.Connections,
        b => b.ConnectionId,
        c => c.Id,
        (b, c) => new { b.Consumption, c.UtilityTypeId })
    .Join(_context.UtilityTypes,
        x => x.UtilityTypeId,
        ut => ut.Id,
        (x, ut) => new { x.Consumption, ut.Name })
    .GroupBy(x => x.Name)
    .Select(g => new
    {

```

```

        UtilityName = g.Key,
        AverageConsumption = g.Average(x ⇒ x.Consumption)
    })
    .OrderByDescending(c ⇒ c.AverageConsumption)
    .ToListAsync();

```

**Used by:**

- **Frontend component:** Analytics section in **Reports** showing average consumption per utility type.

#### 4.6. Connections per Utility Type

File: `ReportService.cs`

Method: `GetConnectionsByUtilityAsync`

**Purpose:**

- Count how many connections exist per utility type.

**Representative LINQ:**

```

var data = await _context.Connections
    .Join(_context.UtilityTypes,
        c ⇒ c.UtilityTypeId,
        ut ⇒ ut.Id,
        (c, ut) ⇒ new { ut.Name })
    .GroupBy(x ⇒ x.Name)
    .Select(g ⇒ new ConnectionsByUtilityDto
    {
        UtilityName = g.Key,
        ConnectionCount = g.Count()
    })
    .OrderByDescending(c ⇒ c.ConnectionCount)
    .ToListAsync();

```

**Used by:**

- **Frontend component:** **Admin / Reports** → "Connections by Utility" pie chart / table.

## 4.7. Consumer Self-Service Consumption Trend & Cost Estimate

File: `ReportService.cs`

Method: `GetMyConsumptionAsync`

### Purpose:

- For the logged-in consumer, group readings by month, sum units, and optionally estimate cost using tariff info.

### Representative LINQ (grouping per month):

```
var readings = await query.ToListAsync();

// Combined trend across utilities
var grouped = readings
    .GroupBy(mr => new { mr.ReadingDate.Year, mr.ReadingDate.Month })
    .OrderBy(g => g.Key.Year)
    .ThenBy(g => g.Key.Month)
    .Select(g => new
    {
        Month = $"{g.Key.Year}-{g.Key.Month:00}",
        TotalConsumption = g.Sum(x => x.Consumption)
    })
    .ToList();
```

### Used by:

- **Frontend component:** Consumer **My Consumption** page / chart.

## 4.8. Account Officer Dashboard Metrics (Aggregations)

Files:

- `DashboardMetricsHelper.cs`
- `AccountOfficerService.cs`

### Key aggregations:

```

// Total revenue = sum of paid bills
public static async Task<decimal> CalculateTotalRevenueAsync(AppDbContext context)
{
    return await context.Bills
        .Where(b => b.Status == "Paid")
        .SumAsync(b => b.TotalAmount);
}

// Total outstanding dues (sum of all unpaid bill totals)
public static async Task<decimal> CalculateOutstandingDuesAsync(AppDbContext context)
{
    var unpaidBills = await context.Bills
        .Include(b => b.Connection)
        .ThenInclude(c => c.UtilityType)
        .ThenInclude(u => u!.BillingCycle)
        .Include(b => b.Connection)
        .ThenInclude(c => c.Tariff)
        .Where(b => b.Status != "Paid")
        .ToListAsync();

    return unpaidBills.Sum(b => b.TotalAmount);
}

// Total consumption across all bills
public static async Task<decimal> CalculateTotalConsumptionAsync(AppDbContext context)
{
    return await context.Bills
        .SumAsync(b => b.Consumption);
}

```

**Used by:**

- **Frontend component: Account Officer Dashboard** cards (Total Revenue, Outstanding Dues, Total Consumption, Unpaid Bills Count).

#### 4.9. Outstanding by Utility & Outstanding Bills Paging

File: `AccountOfficerService.cs`

Methods:

- `GetOutstandingByUtilityAsync`
- `GetOutstandingBillsAsync`

#### Representative LINQ:

```
var unpaidBills = await _context.Bills
    .Include(b => b.Connection)
    .ThenInclude(c => c.UtilityType)
    .ThenInclude(u => u!.BillingCycle)
    .Include(b => b.Connection)
    .ThenInclude(c => c.Tariff)
    .Where(b => b.Status != "Paid")
    .ToListAsync();

var outstandingByUtility = unpaidBills
    .Where(b => b.Status == "Due" || b.Status == "Overdue")
    .GroupBy(b => b.Connection.UtilityType.Name)
    .Select(g => new OutstandingByUtilityDto
    {
        UtilityName = g.Key,
        OutstandingAmount = g.Sum(b => b.TotalAmount)
    })
    .OrderByDescending(x => x.OutstandingAmount)
    .ToList();
```

#### Used by:

- **Frontend components:**
  - **Outstanding by Utility** chart for Account Officer.

- Paged **Outstanding Bills** table with status filters.

#### 4.10. Consumer-wise Billing Summary

File: `AccountOfficerService.cs`

Method: `GetConsumerBillingSummaryAsync`

##### Purpose:

- Group bills by consumer and calculate total billed, total paid, outstanding, and overdue count.

##### Representative LINQ:

```
var summaries = bills
    .GroupBy(b => new { b.Connection.UserId, b.Connection.User.FullName })
    .Select(g => new ConsumerBillingSummaryDto
    {
        ConsumerId = g.Key.UserId,
        ConsumerName = g.Key.FullName,
        TotalBilled = g.Sum(b => b.TotalAmount),
        TotalPaid = g.Where(b => b.Status == "Paid").Sum(b => b.TotalAmount),
        OutstandingBalance = g.Where(b => b.Status != "Paid").Sum(b => b.TotalA
mount),
        OverdueCount = g.Count(b => b.Status == "Overdue")
    });
```

##### Used by:

- **Frontend component: Consumer-wise Billing Summary** report for Account Officer.

## 5. Admin, Master Data & User Management

These features mostly use simple LINQ for lookups, ordering, and validation (Any/FirstOrDefault).

### 5.1. Tariffs Management ( `TariffService` )

Examples:

```
// List tariffs ordered by name
var tariffs = await _context.Tariffs
    .OrderBy(t => t.Name.ToLower())
    .ToListAsync();

// Validate that utility type exists
var utilityTypeExists = await _context.UtilityTypes
    .AnyAsync(u => u.Id == dto.UtilityTypeId);

// Check active connections when updating tariff
var activeConnections = tariff.Connections
    .Where(c => c.Status == "Active")
    .ToList();
```

#### Used by:

- **Frontend components: Tariff Management** pages in the Admin panel.

## 5.2. Utility Types Management ( `UtilityTypeService` )

Examples:

```
// List all utility types ordered by name
var utilityTypes = await _context.UtilityTypes
    .OrderBy(u => u.Name.ToLower())
    .ToListAsync();

// Find utility types for which a user has connections
var utilityTypeIds = await _context.Connections
    .Where(c => c.UserId == userId)
    .Select(c => c.UtilityTypeId)
    .Distinct()
    .ToListAsync();

var enabledTypesForUser = await _context.UtilityTypes
```

```
.Where(u => utilityTypeIds.Contains(u.Id) && u.Status == "Enabled")
.ToListAsync();
```

**Used by:**

- **Frontend components: Utility Types** admin pages, and consumer filter dropdowns.

### 5.3. Connections Management ( **ConnectionService** )

**Examples:**

```
// Get all connections with related entities
var connections = await _context.Connections
    .Include(c => c.User)
    .Include(c => c.UtilityType)
    .Include(c => c.Tariff)
    .ToListAsync();

// Connections for a user
var userConnections = await _context.Connections
    .Include(c => c.User)
    .Include(c => c.UtilityType)
    .Include(c => c.Tariff)
    .Where(c => c.UserId == userId)
    .ToListAsync();

// Existence / uniqueness checks
if (await _context.Connections.AnyAsync(c => c.MeterNumber == dto.MeterNumber))
    throw new InvalidOperationException("Meter number already exists");
```

**Used by:**

- **Frontend component: Connection Management** (Admin) and **My Connections** (Consumer).

### 5.4. Billing Cycle Management ( **BillingCycleService** )

Examples:

```
// Load billing cycles
var cycles = await _context.BillingCycles.ToListAsync();

// Check if cycle is assigned to utility types
var cycle = await _context.BillingCycles
    .Include(bc => bc.UtilityTypes)
    .FirstOrDefaultAsync(bc => bc.Id == id);

// Check for meter readings in current/previous periods
var hasActiveReadings = await _context.MeterReadings
    .AnyAsync(mr => mr.BillingCycleId == id &&
        mr.Status == "ReadyForBilling" &&
        ((mr.ReadingDate >= currentPeriodStart && mr.ReadingDate <= currentPeriodEnd) ||
            (mr.ReadingDate >= previousPeriodStart && mr.ReadingDate <= previousPeriodEnd)));
```

**Used by:**

- **Frontend component: Billing Cycle Management** page in Admin panel.

## 5.5. User Management & Soft Delete ( **UserService** )

Examples:

```
// List non-deleted users
var userList = await _userManager.Users
    .Where(u => u.Status != "Deleted")
    .ToListAsync();

// Deactivate all active connections when user is set to Inactive
var activeConnections = await _context.Connections
    .Where(c => c.UserId == id && c.Status == "Active")
    .ToListAsync();
```

```
// Prevent deleting if unpaid bills exist
var hasUnpaidBills = await _context.Bills
    .Include(b => b.Connection)
    .AnyAsync(b => b.Connection.UserId == id && b.Status != "Paid");
```

**Used by:**

- **Frontend component: User Management** grid in Admin panel.

## 5.6. Utility Requests ( `UtilityRequestService` )

**Examples:**

```
// All requests
var requests = await _context.UtilityRequests.ToListAsync();

// Requests for a specific user
var userRequests = await _context.UtilityRequests
    .Where(ur => ur.UserId == userId)
    .ToListAsync();
```

**Used by:**

- **Frontend components:**
  - **Request Management** page for Admin.
  - Consumer's **My Requests** page.

---

## 6. Seed Data & Demo Scenarios (DataSeeder)

File: `Data/DataSeeder.cs`

**Purpose:**

- Populate roles, users, billing cycles, utility types, tariffs, connections, historical meter readings, bills, and payments for demo/testing.

**Representative LINQ usages:**

```

// Check if any data exists before seeding
if (!await _context.UtilityTypes.AnyAsync()) { ... }
if (!await _context.BillingCycles.AnyAsync()) { ... }

// Load connections with related entities for demo readings
var connections = await _context.Connections
    .Include(c => c.UtilityType)
    .ThenInclude(ut => ut!.BillingCycle)
    .Include(c => c.Tariff)
    .Where(c => c.UserId == consumerUser.Id && c.Status == "Active")
    .ToListAsync();

// Find previous billed reading to chain cumulative meter readings
var previousReading = await _context.MeterReadings
    .Where(mr => mr.ConnectionId == connection.Id && mr.Status == "Billed")
    .OrderByDescending(mr => mr.ReadingDate)
    .Select(mr => mr.CurrentReading)
    .FirstOrDefaultAsync();

// Ensure one bill per connection per billing period
var existingBill = await _context.Bills
    .Where(b => b.ConnectionId == connection.Id && b.BillingPeriod == billingP
eriod)
    .FirstOrDefaultAsync();

```