Querying With SQL and LINQ

.NET CORE

Entity Framework (EF) Core *allows you to use the navigation properties in your Entity Models to load entities.*

Loading Data from the DB

Entity Framework Core allows you to use the navigation properties in your *model* to load related entities.

There are three common *O/RM* patterns used to load data.

- *Eager loading* - data is loaded from the database as part of the initial query.

- *Explicit loading* - data is explicitly loaded from the database at a later time.

- *Lazy loading* - data is loaded from the database when the navigation property is accessed. By default, LINQ to SQL load related entities using Lazy Loading.

**We'll focus on Eager and Lazy Loading only**

Lazy and Eager Loading Explained

Lazy Loading - The default behavior of *Entity Framework Core*. An entity is loaded only when it is accessed for the first time. This delays the loading of the related data until you ask for it.

Ex. When the departments are loaded, the related entities (Employee) are not loaded.

Eager Loading – All entities are loaded at once as a part of the initial query. A large amount of data may be loaded at once.
Ex. Load the first found employee with DepartmentId == ID along with the employee details.

```
var dept = dbContext.Employee.Where(a =>
a.UserId == userId).FirstOrDefault();
```
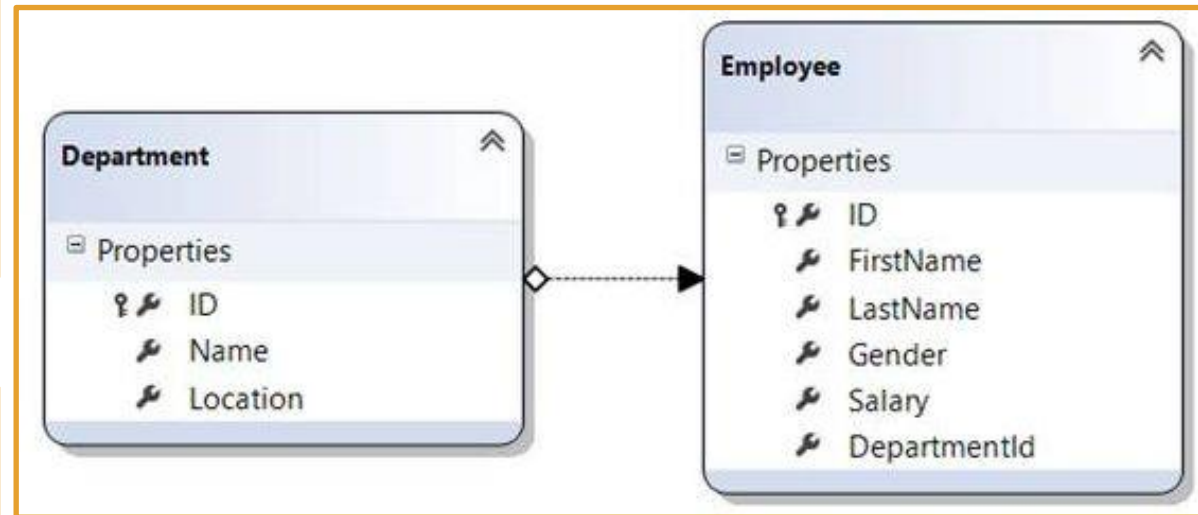


```
var dept = dbContext.Department.Include(a =>
a.Employee).Where(a => a.DepartmentId ==
ID);
```

Eager Loading - Example

Use the **.Include()** method to specify related data from one or multiple relationships to be included in a single query result (Eager Loading). Here, the blogs that are returned in the results will have their Posts property populated with the related posts automatically.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ToList();
}
```

You can drill down through relationships to include multiple levels of related data using the .ThenInclude() method. The following example loads all blogs, their related posts, and the author of each post.

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
            .ThenInclude(author => author.Photo)
    .ToList();
```

Tracking Queries

By default, queries that return entity types are *tracked*. Tracking behavior controls whether *Entity Framework Core* will keep information about an entity instance in its "change tracker".

If an entity is *tracked*, any changes detected in the entity will be persisted to the database during *.SaveChanges()*.

```
var blog = context.Blogs.SingleOrDefault(b => b.BlogId == 1);
blog.Rating = 5;
context.SaveChanges();
```

No-Tracking Queries

If you don't need to update the entities (a read-only scenario) retrieved from the database, then a '*no-tracking'* query should be used. They are quicker to execute because there's no need to set up the *change tracking* information.

```
var blogs = context.Blogs
        .AsNoTracking()
        .ToList();
```

You can also change the default tracking behavior at the context instance level

```
context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

var blogs = context.Blogs.ToList();
```

Raw SQL Queries

https://docs.microsoft.com/en-us/ef/core/querying/raw-sql

Entity Framework Core allows you to use *raw SQL queries*. *Raw SQL queries* are useful if the query you want can't be expressed using *LINQ* or if using a *LINQ* query is resulting in an inefficient SQL query.

Use the .FromSqlRaw() extension method to begin a *LINQ* query based on a *raw SQL query*. .FromSqlRaw() can only be used on query roots, that is directly on the DbSet<>.

Raw SQL queries can be used to execute a stored procedure.

```
var blogs = context.Blogs
    .FromSqlRaw("SELECT * FROM dbo.Blogs")
    .ToList();
```

```
var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogs")
    .ToList();
```

Raw SQL Queries

You can compose on top of the initial *raw SQL query* using *LINQ* operators. Composing with *LINQ* requires your *raw SQL query* to be composable since *EF Core* will treat the supplied SQL as a subquery. SQL queries that can be composed on begin with the *SELECT* keyword.

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSqlInterpolated($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Where(b => b.Rating > 3)
    .OrderByDescending(b => b.Rating)
    .ToList();
```