Querying With SQL and LINQ

.NET

Entity Framework Core *allows you to use the navigation properties in your Entity models to load related entities. There are three common O/RM patterns used to load related data.*

[HTTPS://DOCS.MICROSOFT.COM/EN-US/EF/CORE/QUERYING/RELATED-DATA](https://docs.microsoft.com/en-us/ef/core/querying/related-data)

Loading Data from the DB

https://docs.microsoft.com/en-us/ef/core/querying/related-data

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.

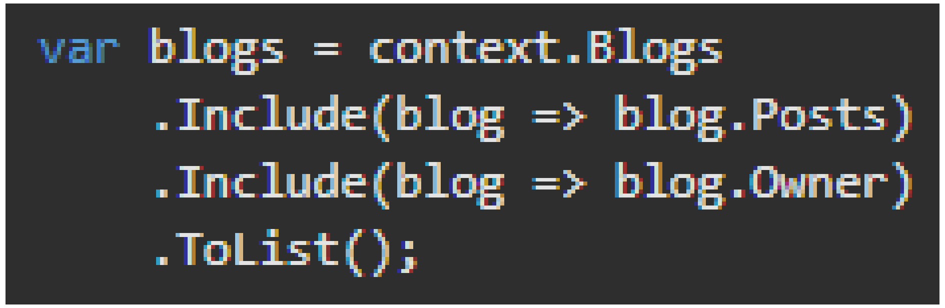
- *Eager loading* - data is loaded from the database as part of the initial query.
- *Explicit loading* - data is explicitly loaded from the database at a later time.
- *Lazy loading* - data is transparently loaded from the database when the navigation property is accessed.

We'll focus on Eager and Lazy Loading only

Eager Loading

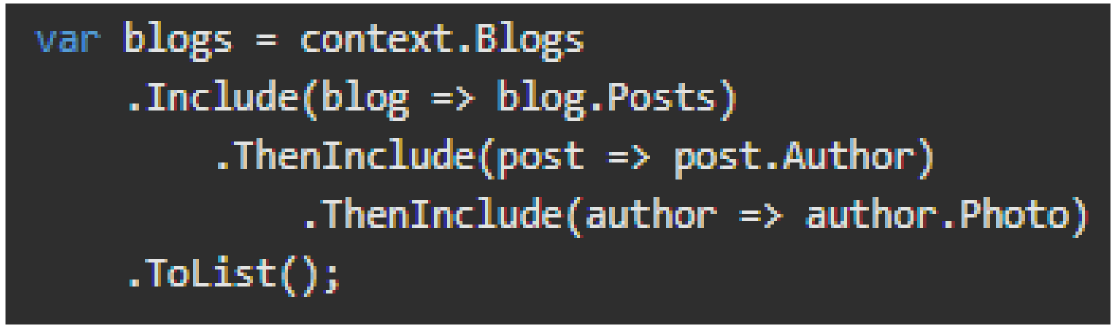
https://docs.microsoft.com/en-us/ef/core/querying/related-data#eager-loading

Use the *.Include()* method to specify related data from one or multiple relationships to be included in query results. In this example, the blogs that are returned in the results will have their Posts property populated with the related posts.



```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
    .Include(blog => blog.Owner)
    .ToList();
```

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.



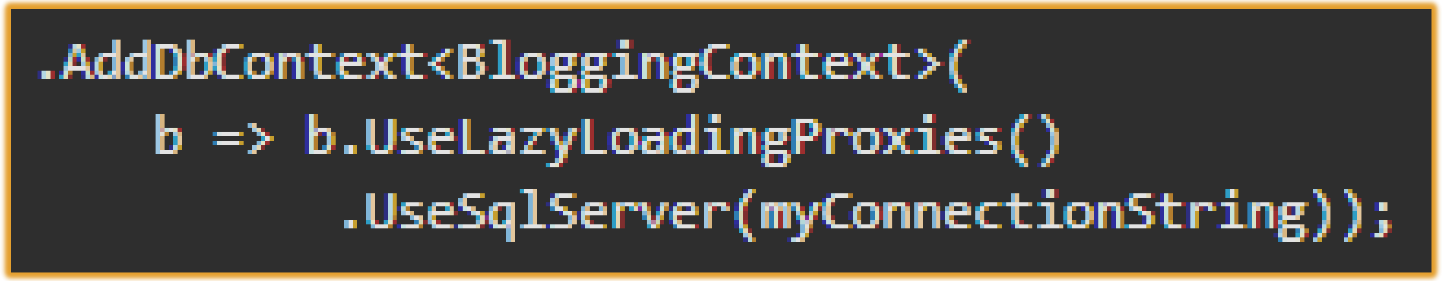
```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
            .ThenInclude(author => author.Photo)
    .ToList();
```

Lazy Loading

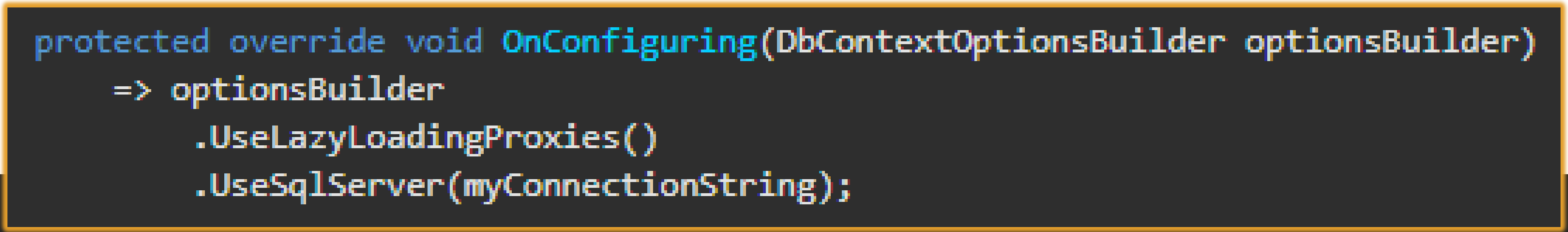
https://docs.microsoft.com/en-us/ef/core/querying/related-data#lazy-loading

The simplest way to use *lazy-loading* is by installing the *Microsoft.EntityFrameworkCore.Proxies* package and enabling it with a call to *UseLazyLoadingProxies…* or when using *AddDbContext*

EF Core will then *enable lazy* loading for any navigation property that can be overridden--that is, it must be virtual and on a class that can be inherited from.



```
.AddDbContext<BloggingContext>(
    b => b.UseLazyLoadingProxies()
          .UseSqlServer(myConnectionString));
```



```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);
```

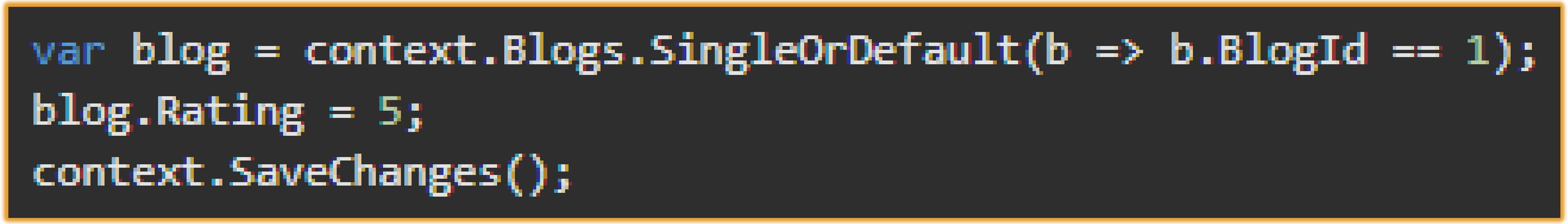
Tracking Queries

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

Tracking behavior controls if *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()*.

By default, queries that return entity types are ***tracking***. Which means you can make changes to those entity instances and have those changes persisted by *.SaveChanges()*.



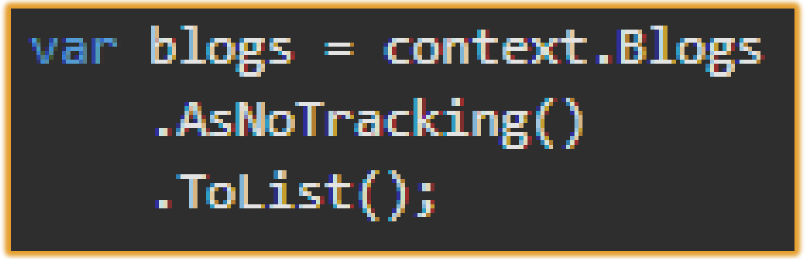
```
var blog = context.Blogs.SingleOrDefault(b => b.BlogId == 1);
blog.Rating = 5;
context.SaveChanges();
```

No-Tracking Queries

https://docs.microsoft.com/en-us/ef/core/querying/tracking#no-tracking-queries

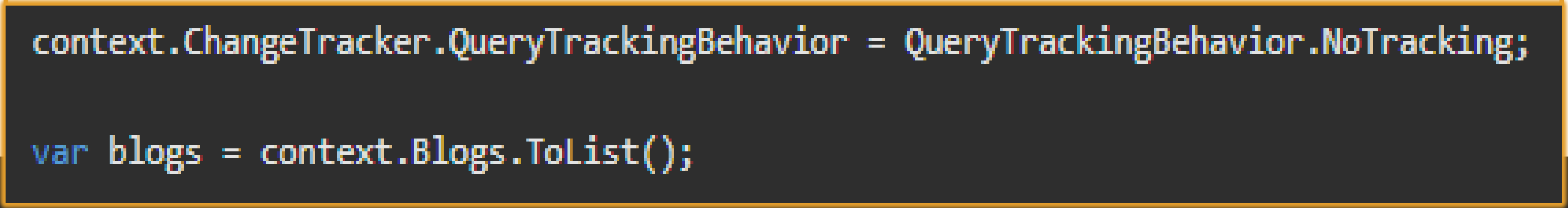
‘No tracking’ queries are useful when the results are used in a read-only scenario. They're quicker to execute because there's no need to set up the ***change tracking*** information.

If you don't need to update the entities retrieved from the database, then a ‘***no-tracking’*** query should be used. You can swap an individual query to be no-tracking.



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

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



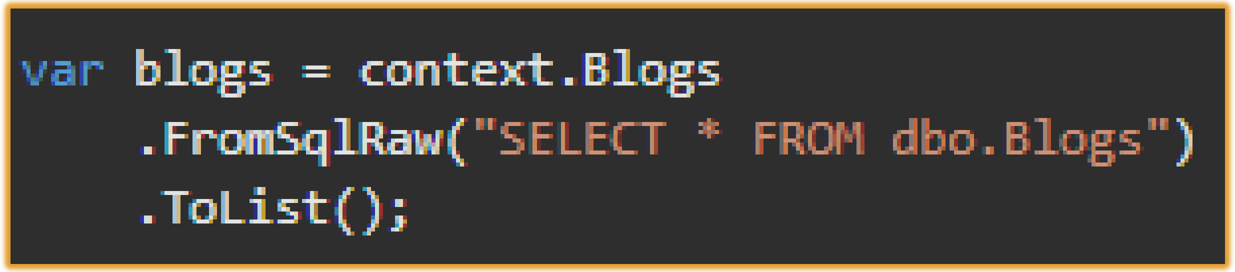
```
context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

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

Raw SQL Queries

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

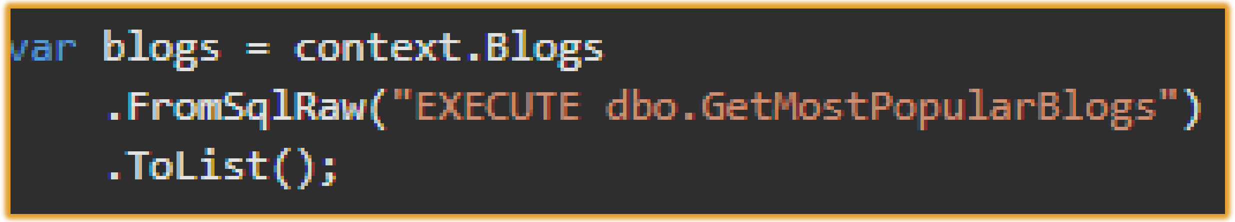
Entity Framework Core allows you to drop down to *raw SQL queries* when working with a relational database. *Raw SQL queries* are useful if the query you want can't be expressed using *LINQ*. *Raw SQL queries* are also used if using a *LINQ* query is resulting in an inefficient SQL query.

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<>*.



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

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

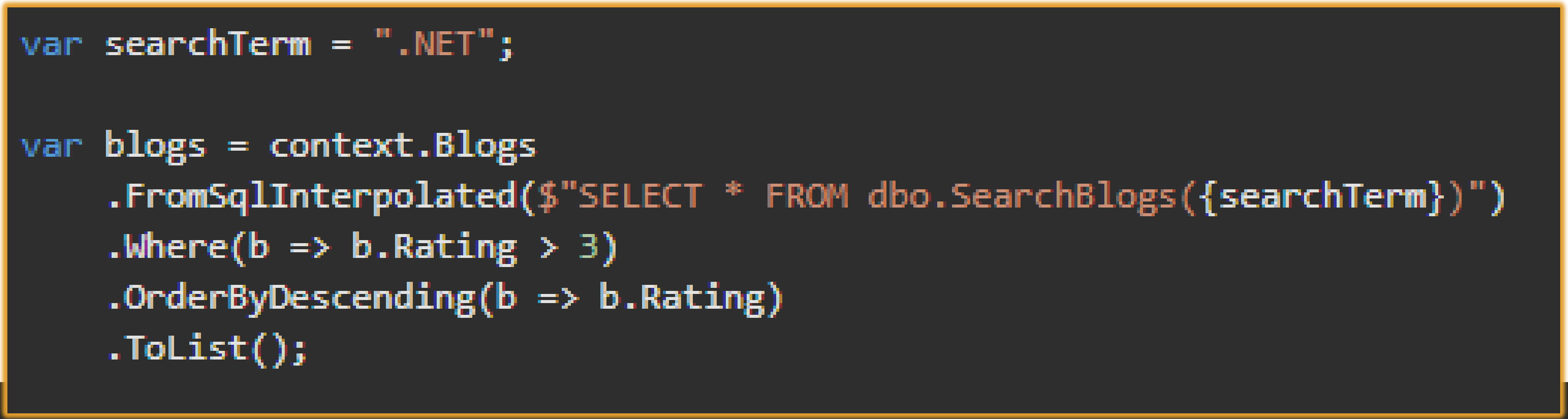


```
var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogs")
    .ToList();
```

Raw SQL Queries

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

You can compose on top of the initial ***raw SQL query*** using ***LINQ*** operators. ***EF Core*** will treat it as subquery and compose over it in the database. 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();
```