# Loading Related Data

10/27/2016 • 9 minutes to read • +8

**In this article**

Eager loading
Explicit loading
Lazy loading
Related data and serialization

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 related data.

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

**Tip**

You can view this article's sample on GitHub.

## Eager loading

You can use the `Include` method to specify related data to be included in query results. In the following example, the blogs that are returned in the results will have their `Posts` property populated with the related posts.

C# Copy

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

**Tip**

Entity Framework Core will automatically fix-up navigation properties to any other entities that were previously loaded into the context instance. So even if you don't explicitly include the data for a navigation property, the property may still be populated if some or all of the related entities were previously loaded.

You can include related data from multiple relationships in a single query.

C#

Copy

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

## Including multiple levels

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.

C#

Copy

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

ⓘ Note

Current versions of Visual Studio offer incorrect code completion options and can cause correct expressions to be flagged with syntax errors when using the `ThenInclude` method after a collection navigation property. This is a symptom of an IntelliSense bug tracked at https://github.com/dotnet/roslyn/issues/8237. It is safe to ignore these spurious syntax errors as long as the code is correct and can be compiled successfully.

You can chain multiple calls to `ThenInclude` to continue including further levels of related data.

C# Copy

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

You can combine all of this to include related data from multiple levels and multiple roots in the same query.

C# Copy

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
                .ThenInclude(author => author.Photo)
        .Include(blog => blog.Owner)
            .ThenInclude(owner => owner.Photo)
        .ToList();
}
```

You may want to include multiple related entities for one of the entities that is being included. For example, when querying `Blogs`, you include `Posts` and then want to include both the `Author` and `Tags` of the `Posts`. To do this, you need to specify each include path starting at the root. For example, `Blog -> Posts -> Author` and `Blog -> Posts -> Tags`. This does not mean you will get redundant joins; in most cases, EF will consolidate the joins when generating SQL.

C# Copy

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

## Include on derived types

You can include related data from navigations defined only on a derived type using `Include` and `ThenInclude`.

Given the following model:

C# Copy

```csharp
public class SchoolContext : DbContext
{
    public DbSet<Person> People { get; set; }
    public DbSet<School> Schools { get; set; }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<School>().HasMany(s =>
s.Students).WithOne(s => s.School);
    }
}

public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Student : Person
{
    public School School { get; set; }
}

public class School
{
    public int Id { get; set; }
    public string Name { get; set; }

    public List<Student> Students { get; set; }
}
```

Contents of `School` navigation of all People who are Students can be eagerly loaded using a number of patterns:

- using cast

  C# Copy

```
context.People.Include(person =>
((Student)person).School).ToList()
```

- using `as` operator

C# Copy

```
context.People.Include(person => (person as
Student).School).ToList()
```

- using overload of `Include` that takes parameter of type `string`

C# Copy

```
context.People.Include("School").ToList()
```

## Ignored includes

If you change the query so that it no longer returns instances of the entity type that the query began with, then the include operators are ignored.

In the following example, the include operators are based on the `Blog`, but then the `Select` operator is used to change the query to return an anonymous type. In this case, the include operators have no effect.

C# Copy

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

By default, EF Core will log a warning when include operators are ignored. See Logging for more information on viewing logging output. You can change the behavior when an include operator is ignored to either throw or do nothing. This is done when setting up the options for your context - typically in `DbContext.OnConfiguring`, or in `Startup.cs` if you are using ASP.NET Core.

C# Copy

```csharp
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFQuerying;Trusted_Connection=True;Co
nnectRetryCount=0")
        .ConfigureWarnings(warnings =>
warnings.Throw(CoreEventId.IncludeIgnoredWarning));
}
```

## Explicit loading

**Note**

This feature was introduced in EF Core 1.1.

You can explicitly load a navigation property via the `DbContext.Entry(...)` API.

C# Copy

```csharp
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    context.Entry(blog)
        .Collection(b => b.Posts)
        .Load();

    context.Entry(blog)
        .Reference(b => b.Owner)
        .Load();
}
```

You can also explicitly load a navigation property by executing a separate query that returns the related entities. If change tracking is enabled, then when loading an entity, EF Core will automatically set the navigation properties of the newly-loaded entitiy to refer to any entities already loaded, and set the navigation properties of the already-loaded entities to refer to the newly-loaded entity.

### Querying related entities

You can also get a LINQ query that represents the contents of a navigation property.

This allows you to do things such as running an aggregate operator over the related entities without loading them into memory.

C# Copy

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    var postCount = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Count();
}
```

You can also filter which related entities are loaded into memory.

C# Copy

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    var goodPosts = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Rating > 3)
        .ToList();
}
```

## Lazy loading

**Note**

This feature was introduced in EF Core 2.1.

The simplest way to use lazy-loading is by installing the Microsoft.EntityFrameworkCore.Proxies package and enabling it with a call to `UseLazyLoadingProxies`. For example:

C# Copy

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

Or when using AddDbContext:

C# Copy

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

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. For example, in the following entities, the `Post.Blog` and `Blog.Posts` navigation properties will be lazy-loaded.

C# Copy

```csharp
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; }

    public virtual ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public virtual Blog Blog { get; set; }
}
```

## Lazy loading without proxies

Lazy-loading proxies work by injecting the `ILazyLoader` service into an entity, as described in Entity Type Constructors. For example:

C# Copy

```csharp
public class Blog
{
    private ICollection<Post> _posts;

    public Blog()
    {
    }

    private Blog(ILazyLoader lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Post> Posts
    {
        get => LazyLoader.Load(this, ref _posts);
        set => _posts = value;
    }
}

public class Post
{
    private Blog _blog;

    public Post()
    {
    }

    private Post(ILazyLoader lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog
    {
        get => LazyLoader.Load(this, ref _blog);
        set => _blog = value;
    }
}
```

This doesn't require entity types to be inherited from or navigation properties to be virtual, and allows entity instances created with `new` to lazy-load once attached to a context. However, it requires a reference to the `ILazyLoader` service, which is defined in the Microsoft.EntityFrameworkCore.Abstractions package. This package contains a minimal set of types so that there is very little impact in depending on it. However, to completely avoid depending on any EF Core packages in the entity types, it is possible to inject the `ILazyLoader.Load` method as a delegate. For example:

C#

Copy

```csharp
public class Blog
{
    private ICollection<Post> _posts;

    public Blog()
    {
    }

    private Blog(Action<object, string> lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private Action<object, string> LazyLoader { get; set; }

    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Post> Posts
    {
        get => LazyLoader.Load(this, ref _posts);
        set => _posts = value;
    }
}

public class Post
{
    private Blog _blog;

    public Post()
    {
    }

    private Post(Action<object, string> lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private Action<object, string> LazyLoader { get; set; }

    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog
    {
        get => LazyLoader.Load(this, ref _blog);
        set => _blog = value;
    }
}
```

The code above uses a `Load` extension method to make using the delegate a bit cleaner:

C#

Copy

```csharp
public static class PocoLoadingExtensions
{
    public static TRelated Load<TRelated>(
        this Action<object, string> loader,
        object entity,
        ref TRelated navigationField,
        [CallerMemberName] string navigationName = null)
        where TRelated : class
    {
        loader?.Invoke(entity, navigationName);

        return navigationField;
    }
}
```

**Note**

The constructor parameter for the lazy-loading delegate must be called "lazyLoader". Configuration to use a different name than this is planned for a future release.

## Related data and serialization

Because EF Core will automatically fix-up navigation properties, you can end up with cycles in your object graph. For example, loading a blog and its related posts will result in a blog object that references a collection of posts. Each of those posts will have a reference back to the blog.

Some serialization frameworks do not allow such cycles. For example, Json.NET will throw the following exception if a cycle is encountered.

> Newtonsoft.Json.JsonSerializationException: Self referencing loop detected for property 'Blog' with type 'MyApplication.Models.Blog'.

If you are using ASP.NET Core, you can configure Json.NET to ignore cycles that it finds in the object graph. This is done in the `ConfigureServices(...)` method in `Startup.cs`.

C#                                                                                    Copy

```csharp
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddMvc()
        .AddJsonOptions(
            options =>
options.SerializerSettings.ReferenceLoopHandling =
Newtonsoft.Json.ReferenceLoopHandling.Ignore
        );

    ...
}
```

Another alternative is to decorate one of the navigation properties with the `[JsonIgnore]` attribute, which instructs Json.NET to not traverse that navigation property while serializing.

**Is this page helpful?**

Yes No