Loading Related Entities

10/23/2016 • 5 minutes to read • +1

In this article

Entity Framework supports three ways to load related data - eager loading, lazy loading and explicit loading. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

Eagerly Loading

Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query. Eager loading is achieved by use of the Include method. For example, the queries below will load blogs and all the posts related to each blog.

C# Copy

```csharp
using (var context = new BloggingContext())
{
    // Load all blogs and related posts.
    var blogs1 = context.Blogs
                        .Include(b => b.Posts)
                        .ToList();

    // Load one blog and its related posts.
    var blog1 = context.Blogs
```

```
                      .Where(b => b.Name == "ADO.NET Blog")
                      .Include(b => b.Posts)
                      .FirstOrDefault();

    // Load all blogs and related posts
    // using a string to specify the relationship.
    var blogs2 = context.Blogs
                       .Include("Posts")
                       .ToList();

    // Load one blog and its related posts
    // using a string to specify the relationship.
    var blog2 = context.Blogs
                      .Where(b => b.Name == "ADO.NET Blog")
                      .Include("Posts")
                      .FirstOrDefault();
}
```

Note

Include is an extension method in the System.Data.Entity namespace so make sure you are using that namespace.

Eagerly loading multiple levels

It is also possible to eagerly load multiple levels of related entities. The queries below show examples of how to do this for both collection and reference navigation properties.

C# Copy

```
using (var context = new BloggingContext())
{
    // Load all blogs, all related posts, and all related comments.
    var blogs1 = context.Blogs
                       .Include(b => b.Posts.Select(p => p.Comments))
```

```
                        .ToList();

    // Load all users, their related profiles, and related avatar.
    var users1 = context.Users
                        .Include(u => u.Profile.Avatar)
                        .ToList();

    // Load all blogs, all related posts, and all related comments
    // using a string to specify the relationships.
    var blogs2 = context.Blogs
                        .Include("Posts.Comments")
                        .ToList();

    // Load all users, their related profiles, and related avatar
    // using a string to specify the relationships.
    var users2 = context.Users
                        .Include("Profile.Avatar")
                        .ToList();
}
```

Note

It is not currently possible to filter which related entities are loaded. Include will always bring in all related entities.

Lazy Loading

Lazy loading is the process whereby an entity or collection of entities is automatically loaded from the database the first time that a property referring to the entity/entities is accessed. When using POCO entity types, lazy loading is achieved by creating instances of derived proxy types and then overriding virtual properties to add the loading hook. For example, when using the Blog entity class defined below, the related Posts will be loaded the first time the Posts navigation property is accessed:

C# Copy

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public string Tags { get; set; }

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

Turn lazy loading off for serialization

Lazy loading and serialization don’t mix well, and if you aren’t careful you can end up querying for your entire database just because lazy loading is enabled. Most serializers work by accessing each property on an instance of a type. Property access triggers lazy loading, so more entities get serialized. On those entities properties are accessed, and even more entities are loaded. It’s a good practice to turn lazy loading off before you serialize an entity. The following sections show how to do this.

Turning off lazy loading for specific navigation properties

Lazy loading of the Posts collection can be turned off by making the Posts property non-virtual:

C# Copy

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public string Tags { get; set; }

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

Loading of the Posts collection can still be achieved using eager loading (see *Eagerly Loading* above) or the Load method (see *Explicitly Loading* below).

Turn off lazy loading for all entities

Lazy loading can be turned off for all entities in the context by setting a flag on the Configuration property. For example:

C#

Copy

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

Loading of related entities can still be achieved using eager loading (see *Eagerly Loading* above) or the Load method (see *Explicitly Loading* below).

Explicitly Loading

Even with lazy loading disabled it is still possible to lazily load related entities, but it must be done with an explicit call. To do so you use the Load method on the related entity's entry. For example:

C#

Copy

```
using (var context = new BloggingContext())
{
    var post = context.Posts.Find(2);

    // Load the blog related to a given post.
    context.Entry(post).Reference(p => p.Blog).Load();
    // Load the blog related to a given post using a string.
    context.Entry(post).Reference("Blog").Load();

    var blog = context.Blogs.Find(1);

    // Load the posts related to a given blog.
    context.Entry(blog).Collection(p => p.Posts).Load();

    // Load the posts related to a given blog
    // using a string to specify the relationship.
    context.Entry(blog).Collection("Posts").Load();
}
```

Note

The Reference method should be used when an entity has a navigation property to another single entity. On the other hand, the Collection method should be used when an entity has a navigation property to a collection of other entities.

Applying filters when explicitly loading related entities

The Query method provides access to the underlying query that Entity Framework will use when loading related entities. You can then use LINQ to apply filters to the query before executing it with a call to a LINQ extension method such as ToList, Load, etc. The Query method can be used with both reference and collection navigation properties but is most useful for collections where it can be used to load only part of the collection. For example:

C# Copy

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Load the posts with the 'entity-framework' tag related to a given blog.
    context.Entry(blog)
           .Collection(b => b.Posts)
           .Query()
           .Where(p => p.Tags.Contains("entity-framework"))
           .Load();

    // Load the posts with the 'entity-framework' tag related to a given blog
    // using a string to specify the relationship.
    context.Entry(blog)
           .Collection("Posts")
           .Query()
           .Where(p => p.Tags.Contains("entity-framework"))
           .Load();
}
```

When using the Query method it is usually best to turn off lazy loading for the navigation property. This is because otherwise the entire collection may get loaded automatically by the lazy loading mechanism either before or after the filtered query has been executed.

Note

While the relationship can be specified as a string instead of a lambda expression, the returned IQueryable is not generic when a string is used and so the Cast method is usually needed before anything useful can be done with it.

Using Query to count related entities without loading them

Sometimes it is useful to know how many entities are related to another entity in the database without actually incurring the cost of loading all those entities. The Query method with the LINQ Count method can be used to do this. For example:

C# Copy

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Count how many posts the blog has.
    var postCount = context.Entry(blog)
                           .Collection(b => b.Posts)
                           .Query()
                           .Count();
}
```

Is this page helpful?

Yes No