…

Entity types with constructors

Article •

•

6 minutes to read •

11 contributors

Feedback

It's possible to define a constructor with parameters and have EF Core call this constructor when creating an instance of the entity. The constructor parameters can be bound to mapped properties, or to various kinds of services to facilitate behaviors like lazy-loading.

Note

Currently, all constructor binding is by convention. Configuration of specific constructors to use is planned for a future release.

Binding to mapped properties

Consider a typical Blog/Post model:

```
public class Blog
{
    public int Id { get; set; }

    public string Name { get; set; }
    public string Author { get; set; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public int Id { get; set; }

    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; set; }

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

When EF Core creates instances of these types, such as for the results of a query, it will first call the default parameterless constructor and then set each property to the value from the database. However, if EF Core finds a parameterized constructor with parameter names and types that match those of mapped properties, then it will instead call the parameterized constructor with values for those properties and will not set each property explicitly. For example:

```
public class Blog
{
    public Blog(int id, string name, string author)
    {
        Id = id;
        Name = name;
        Author = author;
    }

    public int Id { get; set; }

    public string Name { get; set; }
    public string Author { get; set; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public Post(int id, string title, DateTime postedOn)
    {
        Id = id;
        Title = title;
        PostedOn = postedOn;
    }

    public int Id { get; set; }

    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; set; }

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

Some things to note:

- Not all properties need to have constructor parameters. For example, the Post.Content property is not set by any constructor parameter, so EF Core will set it after calling the constructor in the normal way.
- The parameter types and names must match property types and names, except that properties can be Pascal-cased while the parameters are camel-cased.
- EF Core cannot set navigation properties (such as Blog or Posts above) using a constructor.
- The constructor can be public, private, or have any other accessibility. However, lazy-loading proxies require that the constructor is accessible from the inheriting proxy class. Usually this means making it either public or protected.

Read-only properties

Once properties are being set via the constructor it can make sense to make some of them read-only. EF Core supports this, but there are some things to look out for:

- Properties without setters are not mapped by convention. (Doing so tends to map properties that should not be mapped, such as computed properties.)
- Using automatically generated key values requires a key property that is read-write, since the key value needs to be set by the key generator when inserting new entities.

An easy way to avoid these things is to use private setters. For example:

```
public class Blog
{
    public Blog(int id, string name, string author)
    {
        Id = id;
        Name = name;
        Author = author;
    }

    public int Id { get; private set; }

    public string Name { get; private set; }
    public string Author { get; private set; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public Post(int id, string title, DateTime postedOn)
    {
        Id = id;
        Title = title;
        PostedOn = postedOn;
    }

    public int Id { get; private set; }

    public string Title { get; private set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; private set; }

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

EF Core sees a property with a private setter as read-write, which means that all properties are mapped as before and the key can still be store-generated.

An alternative to using private setters is to make properties really read-only and add more explicit mapping in OnModelCreating. Likewise, some properties can be removed completely and replaced with only fields. For example, consider these entity types:

```
public class Blog
{
    private int _id;

    public Blog(string name, string author)
    {
        Name = name;
        Author = author;
    }

    public string Name { get; }
    public string Author { get; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    private int _id;

    public Post(string title, DateTime postedOn)
    {
        Title = title;
        PostedOn = postedOn;
    }

    public string Title { get; }
    public string Content { get; set; }
    public DateTime PostedOn { get; }

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

And this configuration in OnModelCreating:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>(
        b =>
        {
            b.HasKey("_id");
            b.Property(e => e.Author);
            b.Property(e => e.Name);
        });

    modelBuilder.Entity<Post>(
        b =>
        {
            b.HasKey("_id");
            b.Property(e => e.Title);
            b.Property(e => e.PostedOn);
        });
}
```

Things to note:

- The key "property" is now a field. It is not a `readonly` field so that store-generated keys can be used.
- The other properties are read-only properties set only in the constructor.
- If the primary key value is only ever set by EF or read from the database, then there is no need to include it in the constructor. This leaves the key "property" as a simple field and makes it clear that it should not be set explicitly when creating new blogs or posts.

Note

This code will result in compiler warning '169' indicating that the field is never used. This can be ignored since in reality EF Core is using the field in an extralinguistic manner.

Injecting services

EF Core can also inject "services" into an entity type's constructor. For example, the following can be injected:

- `DbContext` - the current context instance, which can also be typed as your derived DbContext type
- `ILazyLoader` - the lazy-loading service--see the lazy-loading documentation for more details
- `Action<object, string>` - a lazy-loading delegate--see the lazy-loading documentation for more details
- `IEntityType` - the EF Core metadata associated with this entity type

Note

Currently, only services known by EF Core can be injected. Support for injecting application services is being considered for a future release.

For example, an injected DbContext can be used to selectively access the database to obtain information about related entities without loading them all. In the example below this is used to obtain the number of posts in a blog without loading the posts:

```
public class Blog
{
    public Blog()
    {
    }

    private Blog(BloggingContext context)
    {
        Context = context;
    }

    private BloggingContext Context { get; set; }

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

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

    public int PostsCount
        => Posts?.Count
           ?? Context?.Set<Post>().Count(p => Id == EF.Property<int?>(p, "BlogId"))
           ?? 0;
}

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

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

A few things to notice about this:

- The constructor is private, since it is only ever called by EF Core, and there is another public constructor for general use.
- The code using the injected service (that is, the context) is defensive against it being `null` to handle cases where EF Core is not creating the instance.
- Because service is stored in a read/write property it will be reset when the entity is attached to a new context instance.

Warning

Injecting the DbContext like this is often considered an anti-pattern since it couples your entity types directly to EF Core. Carefully consider all options before using service injection like this.

Feedback

Submit and view feedback for

This product **This page**

View all page feedback

Theme

