Add icon Add cover

EF Core

Here's Microsoft's documentation for this, it's pretty good.

Note: some of this is my opinion (Paul S. Hazelton), but some of it is recommended guidelines from Microsoft docs, which I will try to link where applicable.

Creating Entities

Constructors and Initialization

It looks like Ef core requires entities to have a constructor. Even if this ever changes, I prefer having constructors while using object initializers in addition for the sake of clarity anyway. For these notes we will assume that EF requires a constructor to be present.

I propose 3 ways to define an entity in terms of initialization.

Method 1: Database Constructor and Program Constructor

The entity will have 2 (or more) constructors: an internal constructor (optionally with parameters for each required property, with matching names) intended only for use by EF when loading the object from the database, and one or more public constructors for use by the program when actually creating new entities. While this method is a bit verbose, it is my preference.

C#

```
[Table(nameof(MyEntity))]
[PrimaryKey(nameof(Id))]
public class MyEntity
{
    public Guid Id { get; set; }

    [Required]
    public string Name { get; set; }

    [Required]
    public string Details { get; set; }

    // Navigation property
    public List<ChildEntity> Children { get; private set; }

    // Record that will exist as extra columns on the same table
    public CreatedInformation CreatedInformation { get; private set; }

    // EF Constructor
    // Could omit the parameters and just set all properties to `default!`.
    internal MyEntity(Guid id, string name, string details)
    {
        Id = id;
        Name = name;
        Details = details;

        // EF core can't set navigation properties, so we have to assign default values
        // EF core will set these values automatically after the constructor is called.
        Children = [];
        CreatedInformation = default!;
    }

    // API Post Constructor
    public MyEntity(string name, string details, List<ChildEntity> children, string userId)
    {
        Id = Guid.NewGuid();
        Name = name;
        Details = details;
        Children = children;
        CreatedInformation = new(userId);
    }
}
```

	Pros	Cons
1	✅ Clarity: it is very clear what properties are expected to be not null, and which aren't.	❌ Verbose: EF Core cannot set navigation properties using a constructor, so they have to be initialized with `default!` . And all properties must be initialized in the constructor.
2	✅ Safe: if a new property is added or changed, warnings will be generated if we forget to initialize them.	❌ Verbose: all required properties must essentially be initialized twice.
3	✅ More control: it allows for clear initialization logic even in the database constructor.	ℹ️ Internal Constructor: If the constructor is private, there's an info warning about an unused private member. You can get around this by marking it as internal, which is not idea since it should only be used by EF Core. But I think making it internal is the best middle ground.

Method 2: Suppress Warning

This method has the least boilerplate and is probably the simplest, but at the cost of ugly `#pragma` code. It's my second favorite method, and I think it is the most palatable option for most people on the team.

C#

```csharp
[Table(nameof(MyEntity))]
[PrimaryKey(nameof(Id))]
public class MyEntity
{
    public Guid Id { get; set; }

    [Required]
    public string Name { get; set; }

    [Required]
    public string Details { get; set; }

    // Navigation property
    public List<ChildEntity> Children { get; set; }

    // Record that will exist as extra columns on the same table
    public CreatedInformation CreatedInformation { get; set; }

    // EF Constructor
#pragma warning disable CS8618 // Non-nullable field must contain a non-null value when exiting constructor. Consider adding the 'required' modifier or declaring as nullable.
    private MyEntity() { }
#pragma warning restore CS8618 // Non-nullable field must contain a non-null value when exiting constructor. Consider adding the 'required' modifier or declaring as nullable.

    // API Post Constructor
    public MyEntity(string name, string details, List<ChildEntity> children, string userId)
    {
        Id = Guid.NewGuid();
        Name = name;
        Details = details;
        Children = children;
        CreatedInformation = new(userId);
    }
}
```

	Column 1	Column 2
1	✅ Clarity: less boilerplate, and the only constructors with any meaning are the only ones with any content.	❌ Ugly: pragma warning disable is ugly.
2	✅ Low risk. if a new property is added or changed, warnings will be generated if we forget to initialize them.	

Method 3: Required Members with Optional `SetsRequiredMembers` Attribute

This time the EF Core constructor is parameterless and empty, and all the required properties have the `required` modifier. This actually works just fine if you don't intend on using a constructor and want to only use the object-initializer syntax. I dislike this method because it opens us up to a pit of failure (lmao) which kind of defeats the purpose of the null reference analyzer.

C#

```
[Table(nameof(MyEntity))]
[PrimaryKey(nameof(Id))]
public class MyEntity
{
    public Guid Id { get; set; }

    [Required]
    public required string Name { get; set; }

    [Required]
    public required string Details { get; set; }

    // Navigation property
    public required List<ChildEntity> Children { get; set; }

    // Record that will exist as extra columns on the same table
    public required CreatedInformation CreatedInformation { get; set; }

    // EF Constructor
    internal MyEntity() { }

    // API Post Constructor
    // This attribute is required so consumers of MyEntity can just call the constructor without manually initializing all the required properties.
    // Or omit this constructor entirely if you want to just use object-initializer syntax.
    [SetsRequiredMembers]
    public MyEntity(string name, string details, List<ChildEntity> children, string userId)
    {
        Id = Guid.NewGuid();
        Name = name;
        Details = details;
        Children = children;
        CreatedInformation = new(userId);
    }
}
```

	Pros	Cons
1	✅ No boilerplate: Properties only have to be initialized in the "real" constructor	❌❌ Unsafe: If we add or change a property, the compiler will not warn us if we forget to initialize it. There's a big warning on the Microsoft page about this. I recommend commenting out the attribute while working with the entity itself to get the warning back temporarily.

Navigation Properties

There's a section about this on the Microsoft docs that discusses this in more detail.

In my opinion, required navigations should always have the id (the foreign key) as not nullable and have the navigation object as nullable. Since you can query for the entity without including the navigation, it's expected that the navigation may be null even if the FK is required.

```csharp
public Guid ModuleId { get; set; }
public Module? Module { get; set; }
```

Queries

Includes

The expressions for includes or then-includes that target nullable properties are used by EF Core to identify properties and thus will not produce null reference exceptions. So just use the null forgiveness operator " ! " in these situations.

Example

```csharp
var basicContents = await Context.Set<BasicContent>()
  .Include(c => c.ValueSet!.ValueSetVersionItems)
```