Explicitly Tracking Entities

Article • 07/15/2021

Each DbContext instance tracks changes made to entities. These tracked entities in turn drive the changes to the database when SaveChanges is called.

Entity Framework Core (EF Core) change tracking works best when the same DbContext instance is used to both query for entities and update them by calling SaveChanges. This is because EF Core automatically tracks the state of queried entities and then detects any changes made to these entities when SaveChanges is called. This approach is covered in Change Tracking in EF Core.

> **Tip**
>
> This document assumes that entity states and the basics of EF Core change tracking are understood. See **Change Tracking in EF Core** for more information on these topics.

> **Tip**
>
> You can run and debug into all the code in this document by **downloading the sample code from GitHub** .

> **Tip**
>
> For simplicity, this document uses and references synchronous methods such as **SaveChanges** rather than their async equivalents such as **SaveChangesAsync**. Calling and awaiting the async method can be substituted unless otherwise noted.

Introduction

Entities can be explicitly "attached" to a DbContext such that the context then tracks those entities. This is primarily useful when:

1. Creating new entities that will be inserted into the database.

2. Re-attaching disconnected entities that were previously queried by a *different* DbContext instance.

The first of these will be needed by most applications, and is primarily handled by the DbContext.Add methods.

The second is only needed by applications that change entities or their relationships ***while the entities are not being tracked***. For example, a web application may send entities to the web client where the user makes changes and sends the entities back. These entities are referred to as "disconnected" since they were originally queried from a DbContext, but were then disconnected from that context when sent to the client.

The web application must now re-attach these entities so that they are again tracked and indicate the changes that have been made such that SaveChanges can make appropriate updates to the database. This is primarily handled by the DbContext.Attach and DbContext.Update methods.

> **Tip**
>
> Attaching entities to the *same DbContext instance* that they were queried from should not normally be needed. Do not routinely perform a no-tracking query and then attach the returned entities to the same context. This will be slower than using a tracking query, and may also result in issues such as missing shadow property values, making it harder to get right.

Generated versus explicit key values

By default, integer and GUID key properties are configured to use automatically generated key values. This has a **major advantage for change tracking: an unset key value indicates that the entity is "new"**. By "new", we mean that it has not yet been inserted into the database.

Two models are used in the following sections. The first is configured to **not** use generated key values:

C#

```
public class Blog
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
```

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

    public string Name { get; set; }

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

public class Post
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int Id { get; set; }

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

    public int? BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Non-generated (i.e. explicitly set) key values are shown first in each example because everything is very explicit and easy to follow. This is then followed by an example where generated key values are used:

C#

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

    public IList<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 int? BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Notice that the key properties in this model need no additional configuration here since using generated key values is the default for simple integer keys.

Inserting new entities

Explicit key values

An entity must be tracked in the `Added` state to be inserted by SaveChanges. Entities are typically put in the Added state by calling one of DbContext.Add, DbContext.AddRange, DbContext.AddAsync, DbContext.AddRangeAsync, or the equivalent methods on DbSet<TEntity>.

> **Tip**
>
> These methods all work in the same way in the context of change tracking. See **Additional Change Tracking Features** for more information.

For example, to start tracking a new blog:

C#

```
context.Add(
    new Blog { Id = 1, Name = ".NET Blog", });
```

Inspecting the change tracker debug view following this call shows that the context is tracking the new entity in the `Added` state:

Output

```
Blog {Id: 1} Added
  Id: 1 PK
  Name: '.NET Blog'
  Posts: []
```

However, the Add methods don't just work on an individual entity. They actually start tracking an *entire graph of related entities*, putting them all to the `Added` state. For example, to insert a new blog and associated new posts:

C#

```
context.Add(
    new Blog
    {
```

```
        Id = 1,
        Name = ".NET Blog",
        Posts =
        {
            new Post
            {
                Id = 1,
                Title = "Announcing the Release of EF Core 5.0",
                Content = "Announcing the release of EF Core 5.0, a full fea-
tured cross-platform..."
            },
            new Post
            {
                Id = 2,
                Title = "Announcing F# 5",
                Content = "F# 5 is the latest version of F#, the functional
programming language..."
            }
        }
    });
```

The context is now tracking all these entities as `Added`:

Output

```
Blog {Id: 1} Added
  Id: 1 PK
  Name: '.NET Blog'
  Posts: [{Id: 1}, {Id: 2}]
Post {Id: 1} Added
  Id: 1 PK
  BlogId: 1 FK
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: {Id: 1}
Post {Id: 2} Added
  Id: 2 PK
  BlogId: 1 FK
  Content: 'F# 5 is the latest version of F#, the functional programming...'
  Title: 'Announcing F# 5'
  Blog: {Id: 1}
```

Notice that explicit values have been set for the `Id` key properties in the examples above. This is because the model here has been configured to use explicitly set key values, rather than automatically generated key values. When not using generated keys, the key properties must be explicitly set *before* calling `Add`. These key values are then inserted when SaveChanges is called. For example, when using SQLite:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p0='1' (DbType = String), @p1='.NET
Blog' (Size = 9)], CommandType='Text', CommandTimeout='30']
INSERT INTO "Blogs" ("Id", "Name")
VALUES (@p0, @p1);

-- Executed DbCommand (0ms) [Parameters=[@p2='1' (DbType = String), @p3='1'
(DbType = String), @p4='Announcing the release of EF Core 5.0, a full featured
cross-platform...' (Size = 72), @p5='Announcing the Release of EF Core 5.0'
(Size = 37)], CommandType='Text', CommandTimeout='30']
INSERT INTO "Posts" ("Id", "BlogId", "Content", "Title")
VALUES (@p2, @p3, @p4, @p5);

-- Executed DbCommand (0ms) [Parameters=[@p0='2' (DbType = String), @p1='1'
(DbType = String), @p2='F# 5 is the latest version of F#, the functional pro-
gramming language...' (Size = 72), @p3='Announcing F# 5' (Size = 15)],
CommandType='Text', CommandTimeout='30']
INSERT INTO "Posts" ("Id", "BlogId", "Content", "Title")
VALUES (@p0, @p1, @p2, @p3);
```

All of these entities are tracked in the `Unchanged` state after SaveChanges completes, since these entities now exist in the database:

Output

```
Blog {Id: 1} Unchanged
  Id: 1 PK
  Name: '.NET Blog'
  Posts: [{Id: 1}, {Id: 2}]
Post {Id: 1} Unchanged
  Id: 1 PK
  BlogId: 1 FK
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: {Id: 1}
Post {Id: 2} Unchanged
  Id: 2 PK
  BlogId: 1 FK
  Content: 'F# 5 is the latest version of F#, the functional programming...'
  Title: 'Announcing F# 5'
  Blog: {Id: 1}
```

Generated key values

As mentioned above, integer and GUID key properties are configured to use automatically generated key values by default. This means that the application *must not set any key value explicitly*. For example, to insert a new blog and posts all with generated key values:

C#

```
context.Add(
    new Blog
    {
        Name = ".NET Blog",
        Posts =
        {
            new Post
            {
                Title = "Announcing the Release of EF Core 5.0",
                Content = "Announcing the release of EF Core 5.0, a full fea-
tured cross-platform..."
            },
            new Post
            {
                Title = "Announcing F# 5",
                Content = "F# 5 is the latest version of F#, the functional
programming language..."
            }
        }
    });
```

As with explicit key values, the context is now tracking all these entities as `Added`:

Output

```
Blog {Id: -2147482644} Added
  Id: -2147482644 PK Temporary
  Name: '.NET Blog'
  Posts: [{Id: -2147482637}, {Id: -2147482636}]
Post {Id: -2147482637} Added
  Id: -2147482637 PK Temporary
  BlogId: -2147482644 FK Temporary
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: {Id: -2147482644}
Post {Id: -2147482636} Added
  Id: -2147482636 PK Temporary
  BlogId: -2147482644 FK Temporary
  Content: 'F# 5 is the latest version of F#, the functional programming...'
  Title: 'Announcing F# 5'
  Blog: {Id: -2147482644}
```

Notice in this case that temporary key values have been generated for each entity. These values are used by EF Core until SaveChanges is called, at which point real key values are read back from the database. For example, when using SQLite:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p0='.NET Blog' (Size = 9)],
CommandType='Text', CommandTimeout='30']
INSERT INTO "Blogs" ("Name")
VALUES (@p0);
SELECT "Id"
FROM "Blogs"
WHERE changes() = 1 AND "rowid" = last_insert_rowid();

-- Executed DbCommand (0ms) [Parameters=[@p1='1' (DbType = String),
@p2='Announcing the release of EF Core 5.0, a full featured cross-platform...'
(Size = 72), @p3='Announcing the Release of EF Core 5.0' (Size = 37)],
CommandType='Text', CommandTimeout='30']
INSERT INTO "Posts" ("BlogId", "Content", "Title")
VALUES (@p1, @p2, @p3);
SELECT "Id"
FROM "Posts"
WHERE changes() = 1 AND "rowid" = last_insert_rowid();

-- Executed DbCommand (0ms) [Parameters=[@p0='1' (DbType = String), @p1='F# 5
is the latest version of F#, the functional programming language...' (Size =
72), @p2='Announcing F# 5' (Size = 15)], CommandType='Text',
CommandTimeout='30']
INSERT INTO "Posts" ("BlogId", "Content", "Title")
VALUES (@p0, @p1, @p2);
SELECT "Id"
FROM "Posts"
WHERE changes() = 1 AND "rowid" = last_insert_rowid();
```

After SaveChanges completes, all of the entities have been updated with their real key values and are tracked in the `Unchanged` state since they now match the state in the database:

Output

```
Blog {Id: 1} Unchanged
  Id: 1 PK
  Name: '.NET Blog'
  Posts: [{Id: 1}, {Id: 2}]
Post {Id: 1} Unchanged
  Id: 1 PK
  BlogId: 1 FK
```

```
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: {Id: 1}
Post {Id: 2} Unchanged
  Id: 2 PK
  BlogId: 1 FK
  Content: 'F# 5 is the latest version of F#, the functional programming...'
  Title: 'Announcing F# 5'
  Blog: {Id: 1}
```

This is exactly the same end-state as the previous example that used explicit key values.

> **Tip**
>
> An explicit key value can still be set even when using generated key values. EF Core will then attempt to insert using this key value. Some database configurations, including SQL Server with Identity columns, do not support such inserts and will throw (see these docs for a workaround).

Attaching existing entities

Explicit key values

Entities returned from queries are tracked in the `Unchanged` state. The `Unchanged` state means that the entity has not been modified since it was queried. A disconnected entity, perhaps returned from a web client in an HTTP request, can be put into this state using either DbContext.Attach, DbContext.AttachRange, or the equivalent methods on DbSet<TEntity>. For example, to start tracking an existing blog:

C#

```
context.Attach(
    new Blog { Id = 1, Name = ".NET Blog", });
```

> **Note**

> The examples here are creating entities explicitly with `new` for simplicity. Normally the entity instances will have come from another source, such as being deserialized from a client, or being created from data in an HTTP Post.

Inspecting the change tracker debug view following this call shows that the entity is tracked in the `Unchanged` state:

Output

```
Blog {Id: 1} Unchanged
  Id: 1 PK
  Name: '.NET Blog'
  Posts: []
```

Just like `Add`, `Attach` actually sets an entire graph of connected entities to the `Unchanged` state. For example, to attach an existing blog and associated existing posts:

C#

```
context.Attach(
    new Blog
    {
        Id = 1,
        Name = ".NET Blog",
        Posts =
        {
            new Post
            {
                Id = 1,
                Title = "Announcing the Release of EF Core 5.0",
                Content = "Announcing the release of EF Core 5.0, a full fea-
tured cross-platform..."
            },
            new Post
            {
                Id = 2,
                Title = "Announcing F# 5",
                Content = "F# 5 is the latest version of F#, the functional
programming language..."
            }
        }
    });
```

The context is now tracking all these entities as `Unchanged`:

Output

```
Blog {Id: 1} Unchanged
  Id: 1 PK
  Name: '.NET Blog'
  Posts: [{Id: 1}, {Id: 2}]
Post {Id: 1} Unchanged
  Id: 1 PK
  BlogId: 1 FK
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: {Id: 1}
Post {Id: 2} Unchanged
  Id: 2 PK
  BlogId: 1 FK
  Content: 'F# 5 is the latest version of F#, the functional programming...'
  Title: 'Announcing F# 5'
  Blog: {Id: 1}
```

Calling SaveChanges at this point will have no effect. All the entities are marked as `Unchanged`, so there is nothing to update in the database.

Generated key values

As mentioned above, integer and GUID key properties are configured to use automatically generated key values by default. This has a major advantage when working with disconnected entities: an unset key value indicates that the entity has not yet been inserted into the database. This allows the change tracker to automatically detect new entities and put them in the `Added` state. For example, consider attaching this graph of a blog and posts:

C#

```
context.Attach(
    new Blog
    {
        Id = 1,
        Name = ".NET Blog",
        Posts =
        {
            new Post
            {
                Id = 1,
                Title = "Announcing the Release of EF Core 5.0",
                Content = "Announcing the release of EF Core 5.0, a full fea-
```

```
tured cross-platform..."
            },
            new Post
            {
                Id = 2,
                Title = "Announcing F# 5",
                Content = "F# 5 is the latest version of F#, the functional
programming language..."
            },
            new Post
            {
                Title = "Announcing .NET 5.0",
                Content = ".NET 5.0 includes many enhancements, including sin-
gle file applications, more..."
            },
        }
    });
```

The blog has a key value of 1, indicating that it already exists in the database. Two of the posts also have key values set, but the third does not. EF Core will see this key value as 0, the CLR default for an integer. This results in EF Core marking the new entity as `Added` instead of `Unchanged`:

Output

```
Blog {Id: 1} Unchanged
  Id: 1 PK
  Name: '.NET Blog'
  Posts: [{Id: 1}, {Id: 2}, {Id: -2147482636}]
Post {Id: -2147482636} Added
  Id: -2147482636 PK Temporary
  BlogId: 1 FK
  Content: '.NET 5.0 includes many enhancements, including single file a...'
  Title: 'Announcing .NET 5.0'
  Blog: {Id: 1}
Post {Id: 1} Unchanged
  Id: 1 PK
  BlogId: 1 FK
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: {Id: 1}
Post {Id: 2} Unchanged
  Id: 2 PK
  BlogId: 1 FK
  Content: 'F# 5 is the latest version of F#, the functional programming...'
```

Calling SaveChanges at this point does nothing with the `Unchanged` entities, but inserts the new entity into the database. For example, when using SQLite:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p0='1' (DbType = String), @p1='.NET
5.0 includes many enhancements, including single file applications, more...'
(Size = 80), @p2='Announcing .NET 5.0' (Size = 19)], CommandType='Text',
CommandTimeout='30']
INSERT INTO "Posts" ("BlogId", "Content", "Title")
VALUES (@p0, @p1, @p2);
SELECT "Id"
FROM "Posts"
WHERE changes() = 1 AND "rowid" = last_insert_rowid();
```

The important point to notice here is that, with generated key values, EF Core is able to **automatically distinguish new from existing entities in a disconnected graph**. In a nutshell, when using generated keys, EF Core will always insert an entity when that entity has no key value set.

Updating existing entities

Explicit key values

DbContext.Update, DbContext.UpdateRange, and the equivalent methods on DbSet<TEntity> behave exactly as the `Attach` methods described above, except that entities are put into the `Modified` instead of the `Unchanged` state. For example, to start tracking an existing blog as `Modified`:

C#

```
context.Update(
    new Blog { Id = 1, Name = ".NET Blog", });
```

Inspecting the change tracker debug view following this call shows that the context is tracking this entity in the `Modified` state:

Output

```
Blog {Id: 1} Modified
  Id: 1 PK
```

```
  Name: '.NET Blog' Modified
  Posts: []
```

Just like with `Add` and `Attach`, `Update` actually marks an *entire graph* of related entities as `Modified`. For example, to attach an existing blog and associated existing posts as `Modified`:

C#

```
context.Update(
    new Blog
    {
        Id = 1,
        Name = ".NET Blog",
        Posts =
        {
            new Post
            {
                Id = 1,
                Title = "Announcing the Release of EF Core 5.0",
                Content = "Announcing the release of EF Core 5.0, a full fea-
tured cross-platform..."
            },
            new Post
            {
                Id = 2,
                Title = "Announcing F# 5",
                Content = "F# 5 is the latest version of F#, the functional
programming language..."
            }
        }
    });
```

The context is now tracking all these entities as `Modified`:

Output

```
Blog {Id: 1} Modified
  Id: 1 PK
  Name: '.NET Blog' Modified
  Posts: [{Id: 1}, {Id: 2}]
Post {Id: 1} Modified
  Id: 1 PK
  BlogId: 1 FK Modified Originally <null>
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
Modified
  Title: 'Announcing the Release of EF Core 5.0' Modified
```

```
  Blog: {Id: 1}
Post {Id: 2} Modified
  Id: 2 PK
  BlogId: 1 FK Modified Originally <null>
  Content: 'F# 5 is the latest version of F#, the functional programming...'
Modified
  Title: 'Announcing F# 5' Modified
  Blog: {Id: 1}
```

Calling SaveChanges at this point will cause updates to be sent to the database for all these entities. For example, when using SQLite:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p1='1' (DbType = String), @p0='.NET
Blog' (Size = 9)], CommandType='Text', CommandTimeout='30']
UPDATE "Blogs" SET "Name" = @p0
WHERE "Id" = @p1;
SELECT changes();

-- Executed DbCommand (0ms) [Parameters=[@p3='1' (DbType = String), @p0='1'
(DbType = String), @p1='Announcing the release of EF Core 5.0, a full featured
cross-platform...' (Size = 72), @p2='Announcing the Release of EF Core 5.0'
(Size = 37)], CommandType='Text', CommandTimeout='30']
UPDATE "Posts" SET "BlogId" = @p0, "Content" = @p1, "Title" = @p2
WHERE "Id" = @p3;
SELECT changes();

-- Executed DbCommand (0ms) [Parameters=[@p3='2' (DbType = String), @p0='1'
(DbType = String), @p1='F# 5 is the latest version of F#, the functional pro-
gramming language...' (Size = 72), @p2='Announcing F# 5' (Size = 15)],
CommandType='Text', CommandTimeout='30']
UPDATE "Posts" SET "BlogId" = @p0, "Content" = @p1, "Title" = @p2
WHERE "Id" = @p3;
SELECT changes();
```

Generated key values

As with `Attach`, generated key values have the same major benefit for `Update`: an unset key value indicates that the entity is new and has not yet been inserted into the database. As with `Attach`, this allows the DbContext to automatically detect new entities and put them in the `Added` state. For example, consider calling `Update` with this graph of a blog and posts:

C#

```
context.Update(
    new Blog
    {
        Id = 1,
        Name = ".NET Blog",
        Posts =
        {
            new Post
            {
                Id = 1,
                Title = "Announcing the Release of EF Core 5.0",
                Content = "Announcing the release of EF Core 5.0, a full fea-
tured cross-platform..."
            },
            new Post
            {
                Id = 2,
                Title = "Announcing F# 5",
                Content = "F# 5 is the latest version of F#, the functional
programming language..."
            },
            new Post
            {
                Title = "Announcing .NET 5.0",
                Content = ".NET 5.0 includes many enhancements, including sin-
gle file applications, more..."
            },
        }
    });
```

As with the `Attach` example, the post with no key value is detected as new and set to the `Added` state. The other entities are marked as `Modified`:

Output

```
Blog {Id: 1} Modified
  Id: 1 PK
  Name: '.NET Blog' Modified
  Posts: [{Id: 1}, {Id: 2}, {Id: -2147482633}]
Post {Id: -2147482633} Added
  Id: -2147482633 PK Temporary
  BlogId: 1 FK
  Content: '.NET 5.0 includes many enhancements, including single file a...'
  Title: 'Announcing .NET 5.0'
  Blog: {Id: 1}
Post {Id: 1} Modified
  Id: 1 PK
  BlogId: 1 FK Modified Originally <null>
```

```
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
Modified
  Title: 'Announcing the Release of EF Core 5.0' Modified
  Blog: {Id: 1}
Post {Id: 2} Modified
  Id: 2 PK
  BlogId: 1 FK Modified Originally <null>
  Content: 'F# 5 is the latest version of F#, the functional programming...'
Modified
  Title: 'Announcing F# 5' Modified
  Blog: {Id: 1}
```

Calling `SaveChanges` at this point will cause updates to be sent to the database for all the existing entities, while the new entity is inserted. For example, when using SQLite:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p1='1' (DbType = String), @p0='.NET
Blog' (Size = 9)], CommandType='Text', CommandTimeout='30']
UPDATE "Blogs" SET "Name" = @p0
WHERE "Id" = @p1;
SELECT changes();

-- Executed DbCommand (0ms) [Parameters=[@p3='1' (DbType = String), @p0='1'
(DbType = String), @p1='Announcing the release of EF Core 5.0, a full featured
cross-platform...' (Size = 72), @p2='Announcing the Release of EF Core 5.0'
(Size = 37)], CommandType='Text', CommandTimeout='30']
UPDATE "Posts" SET "BlogId" = @p0, "Content" = @p1, "Title" = @p2
WHERE "Id" = @p3;
SELECT changes();

-- Executed DbCommand (0ms) [Parameters=[@p3='2' (DbType = String), @p0='1'
(DbType = String), @p1='F# 5 is the latest version of F#, the functional pro-
gramming language...' (Size = 72), @p2='Announcing F# 5' (Size = 15)],
CommandType='Text', CommandTimeout='30']
UPDATE "Posts" SET "BlogId" = @p0, "Content" = @p1, "Title" = @p2
WHERE "Id" = @p3;
SELECT changes();

-- Executed DbCommand (0ms) [Parameters=[@p0='1' (DbType = String), @p1='.NET
5.0 includes many enhancements, including single file applications, more...'
(Size = 80), @p2='Announcing .NET 5.0' (Size = 19)], CommandType='Text',
CommandTimeout='30']
INSERT INTO "Posts" ("BlogId", "Content", "Title")
VALUES (@p0, @p1, @p2);
SELECT "Id"
FROM "Posts"
WHERE changes() = 1 AND "rowid" = last_insert_rowid();
```

This is a very easy way to generate updates and inserts from a disconnected graph. However, it results in updates or inserts being sent to the database for every property of every tracked entity, even when some property values may not have been changed. Don't be too scared by this; for many applications with small graphs, this can be an easy and pragmatic way of generating updates. That being said, other more complex patterns can sometimes result in more efficient updates, as described in Identity Resolution in EF Core.

Deleting existing entities

For an entity to be deleted by SaveChanges it must be tracked in the `Deleted` state. Entities are typically put in the `Deleted` state by calling one of DbContext.Remove, DbContext.RemoveRange, or the equivalent methods on DbSet<TEntity>. For example, to mark an existing post as `Deleted`:

C#

```
context.Remove(
    new Post { Id = 2 });
```

Inspecting the change tracker debug view following this call shows that the context is tracking the entity in the `Deleted` state:

Output

```
Post {Id: 2} Deleted
  Id: 2 PK
  BlogId: <null> FK
  Content: <null>
  Title: <null>
  Blog: <null>
```

This entity will be deleted when SaveChanges is called. For example, when using SQLite:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p0='2' (DbType = String)],
CommandType='Text', CommandTimeout='30']
DELETE FROM "Posts"
WHERE "Id" = @p0;
SELECT changes();
```

After SaveChanges completes, the deleted entity is detached from the DbContext since it no longer exists in the database. The debug view is therefore empty because no entities are being tracked.

Deleting dependent/child entities

Deleting dependent/child entities from a graph is more straightforward than deleting principal/parent entities. See the next section and Changing Foreign Keys and Navigations for more information.

It is unusual to call `Remove` on an entity created with `new`. Further, unlike `Add`, `Attach` and `Update`, it is uncommon to call `Remove` on an entity that isn't already tracked in the `Unchanged` or `Modified` state. Instead it is typical to track a single entity or graph of related entities, and then call `Remove` on the entities that should be deleted. This graph of tracked entities is typically created by either:

1. Running a query for the entities
2. Using the `Attach` or `Update` methods on a graph of disconnected entities, as described in the preceding sections.

For example, the code in the previous section is more likely to obtain a post from a client and then do something like this:

C#

```
context.Attach(post);
context.Remove(post);
```

This behaves exactly the same way as the previous example, since calling `Remove` on an un-tracked entity causes it to first be attached and then marked as `Deleted`.

In more realistic examples, a graph of entities is first attached, and then some of those entities are marked as deleted. For example:

C#

```
// Attach a blog and associated posts
context.Attach(blog);
```

```
// Mark one post as Deleted
context.Remove(blog.Posts[1]);
```

All entities are marked as `Unchanged`, except the one on which `Remove` was called:

Output

```
Blog {Id: 1} Unchanged
  Id: 1 PK
  Name: '.NET Blog'
  Posts: [{Id: 1}, {Id: 2}]
Post {Id: 1} Unchanged
  Id: 1 PK
  BlogId: 1 FK
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: {Id: 1}
Post {Id: 2} Deleted
  Id: 2 PK
  BlogId: 1 FK
  Content: 'F# 5 is the latest version of F#, the functional programming...'
  Title: 'Announcing F# 5'
  Blog: {Id: 1}
```

This entity will be deleted when SaveChanges is called. For example, when using SQLite:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p0='2' (DbType = String)],
CommandType='Text', CommandTimeout='30']
DELETE FROM "Posts"
WHERE "Id" = @p0;
SELECT changes();
```

After SaveChanges completes, the deleted entity is detached from the DbContext since it no longer exists in the database. Other entities remain in the `Unchanged` state:

Output

```
Blog {Id: 1} Unchanged
  Id: 1 PK
  Name: '.NET Blog'
  Posts: [{Id: 1}]
Post {Id: 1} Unchanged
  Id: 1 PK
  BlogId: 1 FK
```

```
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: {Id: 1}
```

Deleting principal/parent entities

Each relationship that connects two entity types has a principal or parent end, and a dependent or child end. The dependent/child entity is the one with the foreign key property. In a one-to-many relationship, the principal/parent is on the "one" side, and the dependent/child is on the "many" side. See Relationships for more information.

In the preceding examples we were deleting a post, which is a dependent/child entity in the blog-posts one-to-many relationship. This is relatively straightforward since removal of a dependent/child entity does not have any impact on other entities. On the other hand, deleting a principal/parent entity must also impact any dependent/child entities. Not doing so would leave a foreign key value referencing a primary key value that no longer exists. This is an invalid model state and results in a referential constraint error in most databases.

This invalid model state can be handled in two ways:

1. Setting FK values to null. This indicates that the dependents/children are no longer related to any principal/parent. This is the default for optional relationships where the foreign key must be nullable. Setting the FK to null is not valid for required relationships, where the foreign key is typically non-nullable.
2. Deleting the dependents/children. This is the default for required relationships, and is also valid for optional relationships.

See Changing Foreign Keys and Navigations for detailed information on change tracking and relationships.

Optional relationships

The `Post.BlogId` foreign key property is nullable in the model we have been using. This means the relationship is optional, and hence the default behavior of EF Core is to set `BlogId` foreign key properties to null when the blog is deleted. For example:

C#

```
// Attach a blog and associated posts
context.Attach(blog);
```

```
// Mark the blog as deleted
context.Remove(blog);
```

Inspecting the change tracker debug view following the call to `Remove` shows that, as expected, the blog is now marked as `Deleted`:

Output

```
Blog {Id: 1} Deleted
  Id: 1 PK
  Name: '.NET Blog'
  Posts: [{Id: 1}, {Id: 2}]
Post {Id: 1} Modified
  Id: 1 PK
  BlogId: <null> FK Modified Originally 1
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: <null>
Post {Id: 2} Modified
  Id: 2 PK
  BlogId: <null> FK Modified Originally 1
  Content: 'F# 5 is the latest version of F#, the functional programming...'
  Title: 'Announcing F# 5'
  Blog: <null>
```

More interestingly, all the related posts are now marked as `Modified`. This is because the foreign key property in each entity has been set to null. Calling SaveChanges updates the foreign key value for each post to null in the database, before then deleting the blog:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p1='1' (DbType = String), @p0=NULL],
CommandType='Text', CommandTimeout='30']
UPDATE "Posts" SET "BlogId" = @p0
WHERE "Id" = @p1;
SELECT changes();

-- Executed DbCommand (0ms) [Parameters=[@p1='2' (DbType = String), @p0=NULL],
CommandType='Text', CommandTimeout='30']
UPDATE "Posts" SET "BlogId" = @p0
WHERE "Id" = @p1;
SELECT changes();

-- Executed DbCommand (0ms) [Parameters=[@p2='1' (DbType = String)],
CommandType='Text', CommandTimeout='30']
DELETE FROM "Blogs"
```

```
WHERE "Id" = @p2;
SELECT changes();
```

After SaveChanges completes, the deleted entity is detached from the DbContext since it no longer exists in the database. Other entities are now marked as `Unchanged` with null foreign key values, which matches the state of the database:

Output

```
Post {Id: 1} Unchanged
  Id: 1 PK
  BlogId: <null> FK
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: <null>
Post {Id: 2} Unchanged
  Id: 2 PK
  BlogId: <null> FK
  Content: 'F# 5 is the latest version of F#, the functional programming...'
  Title: 'Announcing F# 5'
  Blog: <null>
```

Required relationships

If the `Post.BlogId` foreign key property is non-nullable, then the relationship between blogs and posts becomes "required". In this situation, EF Core will, by default, delete dependent/child entities when the principal/parent is deleted. For example, deleting a blog with related posts as in the previous example:

C#

```
// Attach a blog and associated posts
context.Attach(blog);

// Mark the blog as deleted
context.Remove(blog);
```

Inspecting the change tracker debug view following the call to `Remove` shows that, as expected, the blog is again marked as `Deleted`:

Output

```
Blog {Id: 1} Deleted
  Id: 1 PK
  Name: '.NET Blog'
  Posts: [{Id: 1}, {Id: 2}]
Post {Id: 1} Deleted
  Id: 1 PK
  BlogId: 1 FK
  Content: 'Announcing the release of EF Core 5.0, a full featured cross...'
  Title: 'Announcing the Release of EF Core 5.0'
  Blog: {Id: 1}
Post {Id: 2} Deleted
  Id: 2 PK
  BlogId: 1 FK
  Content: 'F# 5 is the latest version of F#, the functional programming...'
  Title: 'Announcing F# 5'
  Blog: {Id: 1}
```

More interestingly in this case is that all related posts have also been marked as `Deleted`. Calling SaveChanges causes the blog and all related posts to be deleted from the database:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p0='1' (DbType = String)],
CommandType='Text', CommandTimeout='30']
DELETE FROM "Posts"
WHERE "Id" = @p0;
SELECT changes();

-- Executed DbCommand (0ms) [Parameters=[@p0='2' (DbType = String)],
CommandType='Text', CommandTimeout='30']
DELETE FROM "Posts"
WHERE "Id" = @p0;
SELECT changes();

-- Executed DbCommand (0ms) [Parameters=[@p1='1' (DbType = String)],
CommandType='Text', CommandTimeout='30']
DELETE FROM "Blogs"
WHERE "Id" = @p1;
```

After SaveChanges completes, all the deleted entities are detached from the DbContext since they no longer exist in the database. Output from the debug view is therefore empty.

> **Note**

This document only scratches the surface on working with relationships in EF Core. See Relationships for more information on modeling relationships, and Changing Foreign Keys and Navigations for more information on updating/deleting dependent/child entities when calling SaveChanges.

Custom tracking with TrackGraph

ChangeTracker.TrackGraph works like `Add`, `Attach` and `Update` except that it generates a callback for every entity instance before tracking it. This allows custom logic to be used when determining how to track individual entities in a graph.

For example, consider the rule EF Core uses when tracking entities with generated key values: if the key value is zero, then the entity is new and should be inserted. Let's extend this rule to say if the key value is negative, then the entity should be deleted. This allows us to change the primary key values in entities of a disconnected graph to mark deleted entities:

C#

```
blog.Posts.Add(
    new Post
    {
        Title = "Announcing .NET 5.0",
        Content = ".NET 5.0 includes many enhancements, including single file
applications, more..."
    }
);

var toDelete = blog.Posts.Single(e => e.Title == "Announcing F# 5");
toDelete.Id = -toDelete.Id;
```

This disconnected graph can then be tracked using TrackGraph:

C#

```
public static void UpdateBlog(Blog blog)
{
    using var context = new BlogsContext();

    context.ChangeTracker.TrackGraph(
        blog, node =>
        {
            var propertyEntry = node.Entry.Property("Id");
```

```
            var keyValue = (int)propertyEntry.CurrentValue;

            if (keyValue == 0)
            {
                node.Entry.State = EntityState.Added;
            }
            else if (keyValue < 0)
            {
                propertyEntry.CurrentValue = -keyValue;
                node.Entry.State = EntityState.Deleted;
            }
            else
            {
                node.Entry.State = EntityState.Modified;
            }

            Console.WriteLine($"Tracking {node.Entry.Metadata.DisplayName()}
with key value {keyValue} as {node.Entry.State}");
        });

    context.SaveChanges();
}
```

For each entity in the graph, the code above checks the primary key value *before tracking the entity*. For unset (zero) key values, the code does what EF Core would normally do. That is, if the key is not set, then the entity is marked as `Added`. If the key is set and the value is non-negative, then the entity is marked as `Modified`. However, if a negative key value is found, then its real, non-negative value is restored and the entity is tracked as `Deleted`.

The output from running this code is:

Output

```
Tracking Blog with key value 1 as Modified
Tracking Post with key value 1 as Modified
Tracking Post with key value -2 as Deleted
Tracking Post with key value 0 as Added
```

> **Note**
>
> For simplicity, this code assumes each entity has an integer primary key property called `Id`. This could be codified into an abstract base class or interface. Alternately,

the primary key property or properties could be obtained from the **IEntityType** metadata such that this code would work with any type of entity.

TrackGraph has two overloads. In the simple overload used above, EF Core determines when to stop traversing the graph. Specifically, it stops visiting new related entities from a given entity when that entity is either already tracked, or when the callback does not start tracking the entity.

The advanced overload, ChangeTracker.TrackGraph<TState>(Object, TState, Func<EntityEntryGraphNode<TState>,Boolean>), has a callback that returns a bool. If the callback returns false, then graph traversal stops, otherwise it continues. Care must be taken to avoid infinite loops when using this overload.

The advanced overload also allows state to be supplied to TrackGraph and this state is then passed to each callback.