# Change Tracking in EF Core

Article • 01/12/2023

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

This document presents an overview of Entity Framework Core (EF Core) change tracking and how it relates to queries and updates.

> **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.

## How to track entities

Entity instances become tracked when they are:

- Returned from a query executed against the database
- Explicitly attached to the DbContext by `Add`, `Attach`, `Update`, or similar methods
- Detected as new entities connected to existing tracked entities

Entity instances are no longer tracked when:

- The DbContext is disposed
- The change tracker is cleared
- The entities are explicitly detached

DbContext is designed to represent a short-lived unit-of-work, as described in DbContext Initialization and Configuration. This means that disposing the DbContext is *the normal way* to stop tracking entities. In other words, the lifetime of a DbContext should be:

1. Create the DbContext instance
2. Track some entities
3. Make some changes to the entities
4. Call SaveChanges to update the database
5. Dispose the DbContext instance

> **Tip**
>
> It is not necessary to clear the change tracker or explicitly detach entity instances when taking this approach. However, if you do need to detach entities, then calling ChangeTracker.Clear is more efficient than detaching entities one-by-one.

# Entity states

Every entity is associated with a given EntityState:

- `Detached` entities are not being tracked by the DbContext.
- `Added` entities are new and have not yet been inserted into the database. This means they will be inserted when SaveChanges is called.
- `Unchanged` entities have *not* been changed since they were queried from the database. All entities returned from queries are initially in this state.
- `Modified` entities have been changed since they were queried from the database. This means they will be updated when SaveChanges is called.
- `Deleted` entities exist in the database, but are marked to be deleted when SaveChanges is called.

EF Core tracks changes at the property level. For example, if only a single property value is modified, then a database update will change only that value. However, properties can only be marked as modified when the entity itself is in the Modified state. (Or, from an alternate perspective, the Modified state means that at least one property value has been marked as modified.)

The following table summarizes the different states:

Expand table

| Entity state | Tracked by DbContext | Exists in database | Properties modified | Action on SaveChanges |
|---|---|---|---|---|
| `Detached` | No | - | - | - |
| `Added` | Yes | No | - | Insert |
| `Unchanged` | Yes | Yes | No | - |
| `Modified` | Yes | Yes | Yes | Update |
| `Deleted` | Yes | Yes | - | Delete |

> **Note**
>
> This text uses relational database terms for clarity. NoSQL databases typically support similar operations but possibly with different names. Consult your database provider documentation for more information.

# Tracking from queries

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 has several advantages over explicitly tracking entity instances:

- It is simple. Entity states rarely need to be manipulated explicitly--EF Core takes care of state changes.
- Updates are limited to only those values that have actually changed.
- The values of shadow properties are preserved and used as needed. This is especially relevant when foreign keys are stored in shadow state.
- The original values of properties are preserved automatically and used for efficient updates.

# Simple query and update

For example, consider a simple blog/posts model:

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; }
}
```

We can use this model to query for blogs and posts and then make some updates to the database:

C#

```
using var context = new BlogsContext();

var blog = context.Blogs.Include(e => e.Posts).First(e => e.Name == ".NET
Blog");

blog.Name = ".NET Blog (Updated!)";

foreach (var post in blog.Posts.Where(e => !e.Title.Contains("5.0")))
{
    post.Title = post.Title.Replace("5", "5.0");
}

context.SaveChanges();
```

Calling SaveChanges results in the following database updates, using SQLite as an example database:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p1='1' (DbType = String), @p0='.NET
Blog (Updated!)' (Size = 20)], CommandType='Text', CommandTimeout='30']
UPDATE "Blogs" SET "Name" = @p0
```

```
WHERE "Id" = @p1;
SELECT changes();

-- Executed DbCommand (0ms) [Parameters=[@p1='2' (DbType = String),
@p0='Announcing F# 5.0' (Size = 17)], CommandType='Text', CommandTimeout='30']
UPDATE "Posts" SET "Title" = @p0
WHERE "Id" = @p1;
SELECT changes();
```

The change tracker debug view is a great way visualize which entities are being tracked and what their states are. For example, inserting the following code into the sample above before calling SaveChanges:

C#

```
context.ChangeTracker.DetectChanges();
Console.WriteLine(context.ChangeTracker.DebugView.LongView);
```

Generates the following output:

Output

```
Blog {Id: 1} Modified
  Id: 1 PK
  Name: '.NET Blog (Updated!)' Modified Originally '.NET Blog'
  Posts: [{Id: 1}, {Id: 2}, {Id: 3}]
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} Modified
  Id: 2 PK
  BlogId: 1 FK
  Content: 'F# 5 is the latest version of F#, the functional programming...'
  Title: 'Announcing F# 5.0' Modified Originally 'Announcing F# 5'
  Blog: {Id: 1}
```

Notice specifically:

- The `Blog.Name` property is marked as modified (`Name: '.NET Blog (Updated!)' Modified Originally '.NET Blog'`), and this results in the blog being in the `Modified` state.

- The `Post.Title` property of post 2 is marked as modified (`Title: 'Announcing F# 5.0' Modified Originally 'Announcing F# 5'`), and this results in this post being in the `Modified` state.
- The other property values of post 2 have not changed and are therefore not marked as modified. This is why these values are not included in the database update.
- The other post was not modified in any way. This is why it is still in the `Unchanged` state and is not included in the database update.

# Query then insert, update, and delete

Updates like those in the previous example can be combined with inserts and deletes in the same unit-of-work. For example:

C#

```
using var context = new BlogsContext();

var blog = context.Blogs.Include(e => e.Posts).First(e => e.Name == ".NET
Blog");

// Modify property values
blog.Name = ".NET Blog (Updated!)";

// Insert a new Post
blog.Posts.Add(
    new Post
    {
        Title = "What’s next for System.Text.Json?", Content = ".NET 5.0 was
released recently and has come with many..."
    });

// Mark an existing Post as Deleted
var postToDelete = blog.Posts.Single(e => e.Title == "Announcing F# 5");
context.Remove(postToDelete);

context.ChangeTracker.DetectChanges();
Console.WriteLine(context.ChangeTracker.DebugView.LongView);

context.SaveChanges();
```

In this example:

- A blog and related posts are queried from the database and tracked
- The `Blog.Name` property is changed

- A new post is added to the collection of existing posts for the blog
- An existing post is marked for deletion by calling DbContext.Remove

Looking again at the change tracker debug view before calling SaveChanges shows how EF Core is tracking these changes:

Output

```
Blog {Id: 1} Modified
  Id: 1 PK
  Name: '.NET Blog (Updated!)' Modified Originally '.NET Blog'
  Posts: [{Id: 1}, {Id: 2}, {Id: 3}, {Id: -2147482638}]
Post {Id: -2147482638} Added
  Id: -2147482638 PK Temporary
  BlogId: 1 FK
  Content: '.NET 5.0 was released recently and has come with many...'
  Title: 'What's next for System.Text.Json?'
  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} 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}
```

Notice that:

- The blog is marked as `Modified`. This will generate a database update.
- Post 2 is marked as `Deleted`. This will generate a database delete.
- A new post with a temporary ID is associated with blog 1 and is marked as `Added`. This will generate a database insert.

This results in the following database commands (using SQLite) when SaveChanges is called:

SQL

```
-- Executed DbCommand (0ms) [Parameters=[@p1='1' (DbType = String), @p0='.NET
Blog (Updated!)' (Size = 20)], CommandType='Text', CommandTimeout='30']
UPDATE "Blogs" SET "Name" = @p0
```

```
WHERE "Id" = @p1;
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=[@p0='1' (DbType = String), @p1='.NET
5.0 was released recently and has come with many...' (Size = 56), @p2='What's
next for System.Text.Json?' (Size = 33)], 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();
```

See Explicitly Tracking Entities for more information on inserting and deleting entities. See Change Detection and Notifications for more information on how EF Core automatically detects changes like this.

**Tip**

Call **ChangeTracker.HasChanges()** to determine whether any changes have been made that will cause SaveChanges to make updates to the database. If HasChanges return false, then SaveChanges will be a no-op.

**Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

**.NET feedback**

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback