west monroe

Entity Framework Core Unchained

Maximizing the Performance from Your ORM

Dan Mallott

AUGUST 9, 2025

Thank You to Our Sponsors

AGENDA

01 Configuration – Default or Not?

02 Access Patterns – Optimizing Your CRUD

03 Foot Guns – Delayed and Otherwise

04 Database Design – First Time's the Charm

05 This is Not the Tool You're Looking For

06 Questions

Dan Mallott

@danielmallott.bsky.social
github.com/danielmallott
linkedin.com/in/danielmallott
dmallott@westmonroe.com

Advisory Lead in Technology & Experience at West Monroe

Developing software (and an occasional DBA) since 2011

Most experience is with Microsoft technologies, specifically .NET and SQL Server

Ice hockey referee (USA Hockey and DIU) – ask me about the offseason!

01

Configuration – Default or Not?

Or…why the defaults might let you down

What Are the Defaults?

```
builder.Services.AddDbContext<StackOverflowContext>(options ⇒ {
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));
});
```

- No Query Retry Enabled
- All Query Results are Tracked
- Related Entities Loaded Explicitly
- Limited Logging of Errors and Warnings

How Should We Configure Our Context?

Implement a Retry Policy

- Retry policies protect us against transient database issues (like connection blips in the cloud)
- The retry policy is highly configurable

Use the Appropriate Tracking Behavior

- Using `NoTracking` has a noticeable (positive) effect on read performance and memory usage
- Write-heavy applications will prefer `Tracking` (on by default)

Log All the Things (In Development)

- Enabling detailed error logging and parameter values are extremely valuable when in development mode
- Logging certain warnings produced by EF Core can help us avoid performance issues later

Avoid Lazy Loading Proxies

- Lazy loading can result in the dreaded “N+1” query pattern, with impacts on performance
- If lazy loading a particular entity will be beneficial, consider using the `ILazyLoader` service instead

A Suggested Configuration

```
builder.Services.AddDbContext<StackOverflowContext>(options ⇒
{
    options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection"), sqlServerOptionsAction ⇒
        {
            sqlServerOptionsAction.EnableRetryOnFailure(
                maxRetryCount: 4,
                maxRetryDelay:
                TimeSpan.FromSeconds(1),
                errorNumbersToAdd: []);
        });

    // Highly dependent on your use case, but generally a good idea to use NoTracking
    options.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);

    if (builder.Environment.IsDevelopment())
    {
        options.EnableDetailedErrors(); // Gets field-level error details
        options.EnableSensitiveDataLogging(); // Logs parameter values - don't use in production!
        options.ConfigureWarnings(warnings ⇒
        {
            warnings.Log([
                CoreEventId.FirstWithoutOrderByAndFilterWarning,
                CoreEventId.RowLimitingOperationWithoutOrderByWarning,
                CoreEventId.DistinctAfterOrderByWithoutRowLimitingOperatorWarning,
                CoreEventId.NavigationLazyLoading
            ]);
        });
    }
});
```

02

Access Patterns – Optimizing Your CRUD

Or…why does Entity Framework Core generate the SQL it generates?

Let’s Insert One Record

```
2025-02-24 15:48:32.626 -06:00 [INF] Request starting HTTP/1.1 POST http://localhost:5257/users - application/json 152
2025-02-24 15:48:32.627 -06:00 [INF] Executing endpoint 'CreateUsers'
2025-02-24 15:48:32.738 -06:00 [INF] Executed DbCommand (4ms) [Parameters=[@p0='New user' (Size = 4000), @p1=NULL (DbType = Int32), @p2='43' (Nullable = true), @p3='
2025-02-24T15:48:32.6619650-06:00' (DbType = DateTime), @p4='New User' (Nullable = false) (Size = 40), @p5='0', @p6=NULL (Size = 40), @p7='2025-02-24T15:48:32.664129
0-06:00' (DbType = DateTime), @p8=NULL (Size = 100), @p9='1', @p10='0', @p11='0', @p12=NULL (Size = 200)], CommandType='Text', CommandTimeout='30']
SET IMPLICIT_TRANSACTIONS OFF;
SET NOCOUNT ON;
INSERT INTO [Users] ([AboutMe], [AccountId], [Age], [CreationDate], [DisplayName], [DownVotes], [EmailHash], [LastAccessDate], [Location], [Reputation], [UpVotes], [
Views], [WebsiteUrl])
OUTPUT INSERTED.[Id]
VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8, @p9, @p10, @p11, @p12);
2025-02-24 15:48:32.747 -06:00 [INF] Setting HTTP status code 201.
2025-02-24 15:48:32.747 -06:00 [INF] Writing value of type 'List`1' as Json.
2025-02-24 15:48:32.748 -06:00 [INF] Executed endpoint 'CreateUsers'
2025-02-24 15:48:32.748 -06:00 [INF] HTTP POST /users responded 201 in 121.3888 ms
2025-02-24 15:48:32.748 -06:00 [INF] Request finished HTTP/1.1 POST http://localhost:5257/users - 201 null application/json; charset=utf-8 122.6385ms
```

Now Let’s Insert Multiple Records

```
2025-02-24 15:51:21.552 -06:00 [INF] Request starting HTTP/1.1 POST http://localhost:5257/users - application/json 456
2025-02-24 15:51:21.552 -06:00 [INF] Executing endpoint 'CreateUsers'
2025-02-24 15:51:21.569 -06:00 [INF] Executed DbCommand (6ms) [Parameters=[@p0='New user' (Size = 4000), @p1=NULL (DbType = Int32), @p2='43' (Nullable = true), @p3='
2025-02-24T15:51:21.5549390-06:00' (DbType = DateTime), @p4='New User' (Nullable = false) (Size = 40), @p5='0', @p6=NULL (Size = 40), @p7='2025-02-24T15:51:21.554956
0-06:00' (DbType = DateTime), @p8=NULL (Size = 100), @p9='1', @p10='0', @p11='0', @p12=NULL (Size = 200), @p13='New user' (Size = 4000), @p14=NULL (DbType = Int32),
@p15='43' (Nullable = true), @p16='2025-02-24T15:51:21.5553540-06:00' (DbType = DateTime), @p17='New User' (Nullable = false) (Size = 40), @p18='0', @p19=NULL (Size
= 40), @p20='2025-02-24T15:51:21.5553610-06:00' (DbType = DateTime), @p21=NULL (Size = 100), @p22='1', @p23='0', @p24='0', @p25=NULL (Size = 200), @p26='New user' (S
ize = 4000), @p27=NULL (DbType = Int32), @p28='43' (Nullable = true), @p29='2025-02-24T15:51:21.5554290-06:00' (DbType = DateTime), @p30='New User' (Nullable = false
) (Size = 40), @p31='0', @p32=NULL (Size = 40), @p33='2025-02-24T15:51:21.5554330-06:00' (DbType = DateTime), @p34=NULL (Size = 100), @p35='1', @p36='0', @p37='0', @
p38=NULL (Size = 200)], CommandType='Text', CommandTimeout='30']
SET IMPLICIT_TRANSACTIONS OFF;
SET NOCOUNT ON;
MERGE [Users] USING (
VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8, @p9, @p10, @p11, @p12, 0),
(@p13, @p14, @p15, @p16, @p17, @p18, @p19, @p20, @p21, @p22, @p23, @p24, @p25, 1),
(@p26, @p27, @p28, @p29, @p30, @p31, @p32, @p33, @p34, @p35, @p36, @p37, @p38, 2)) AS i ([AboutMe], [AccountId], [Age], [CreationDate], [DisplayName], [DownVotes], [
EmailHash], [LastAccessDate], [Location], [Reputation], [UpVotes], [Views], [WebsiteUrl], _Position) ON 1=0
WHEN NOT MATCHED THEN
INSERT ([AboutMe], [AccountId], [Age], [CreationDate], [DisplayName], [DownVotes], [EmailHash], [LastAccessDate], [Location], [Reputation], [UpVotes], [Views], [Webs
iteUrl])
VALUES (i.[AboutMe], i.[AccountId], i.[Age], i.[CreationDate], i.[DisplayName], i.[DownVotes], i.[EmailHash], i.[LastAccessDate], i.[Location], i.[Reputation], i.[Up
Votes], i.[Views], i.[WebsiteUrl])
OUTPUT INSERTED.[Id], i._Position;
2025-02-24 15:51:21.569 -06:00 [INF] Setting HTTP status code 201.
2025-02-24 15:51:21.569 -06:00 [INF] Writing value of type 'List`1' as Json.
2025-02-24 15:51:21.570 -06:00 [INF] Executed endpoint 'CreateUsers'
2025-02-24 15:51:21.570 -06:00 [INF] HTTP POST /users responded 201 in 17.4402 ms
2025-02-24 15:51:21.570 -06:00 [INF] Request finished HTTP/1.1 POST http://localhost:5257/users - 201 null application/json; charset=utf-8 17.8007ms
```

Why Is This Happening?

Single insert statements are suboptimal (and slow)

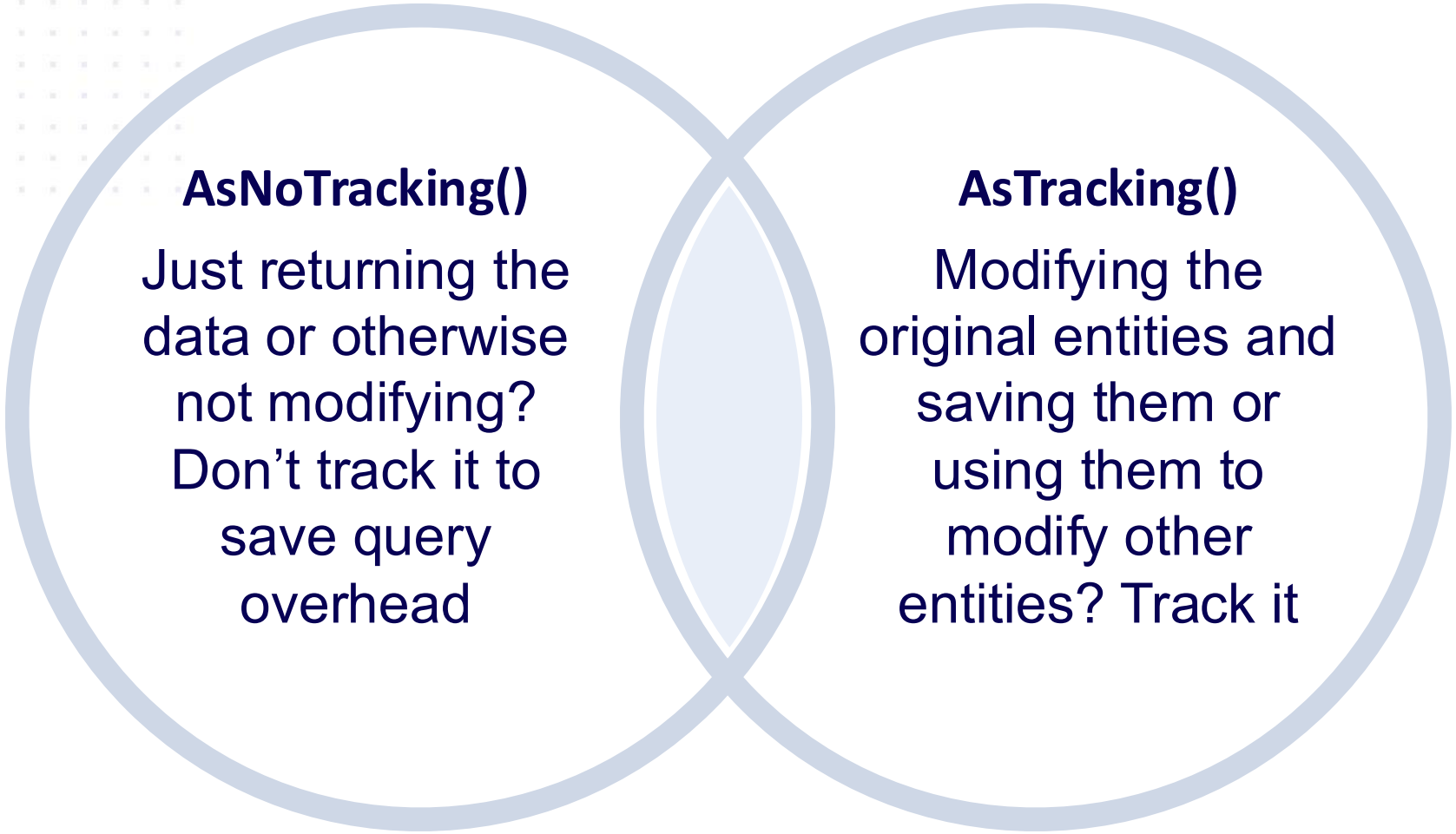
Inserting multiple records is more efficient

Use MERGE to get around some engine-specific issues

There are cases where you might want the “old” behavior. In this case, use the MaxBatchSize configuration to force single insert statements:

```
builder.Services.AddDbContext<StackOverflowContext>(options ⇒ {
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"),
    sqlServerOptionsAction ⇒ {
        sqlServerOptionsAction.MaxBatchSize(1);
    });
});
```

Reading Data – Will You Need to Change It?

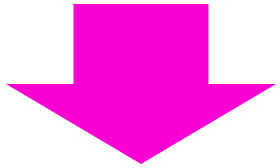


Another Read Optimization

COMPILED QUERIES

- There is overhead associated with both initial compilation of the query tree and finding a compiled query in the cache
- By compiling the query in advance, you can save up to 20% of the resources
- There are limitations on query shape (and the syntax is not necessarily straightforward)

```
public static readonly Func<StackOverflowContext, int, Task<User?>>
    1 reference
    GetUserById = EF.CompileAsyncQuery(
        (StackOverflowContext context, int id) ⇒
            context.Users.FirstOrDefault(u ⇒ u.Id == id));
```



```
app.MapGet("/users/compiled/{id}",
    async (StackOverflowContext context, int id) ⇒
{
    var user = await CompiledQueries.GetUserById(context, id);
    if (user is null)
    {
        return Results.NotFound();
    }
    return Results.Ok(user);
});
```

Updates and Deletes – No Real Options

Unlike INSERT, UPDATE and DELETE happens as single statements. You can change MaxBatchSize to limit how many are sent to the database at once

```
2025-02-24 17:05:17.380 -06:00 [INF] Request starting HTTP/1.1 DELETE http://localhost:5257/users - application/json 28
2025-02-24 17:05:17.382 -06:00 [INF] Executing endpoint 'DeleteUsers'
2025-02-24 17:05:17.425 -06:00 [INF] Executed DbCommand (24ms) [Parameters=[@__ids_0='[10251173,10251174,10251175]' (Size = 4000)], CommandType='Text', CommandTimeou
t='30']
SELECT [u].[Id], [u].[AboutMe], [u].[AccountId], [u].[Age], [u].[CreationDate], [u].[DisplayName], [u].[DownVotes], [u].[EmailHash], [u].[LastAccessDate], [u].[Locat
ion], [u].[Reputation], [u].[UpVotes], [u].[Views], [u].[WebsiteUrl]
FROM [Users] AS [u]
WHERE [u].[Id] IN (
    SELECT [i].[value]
    FROM OPENJSON(@__ids_0) WITH ([value] int '$') AS [i]
)
2025-02-24 17:05:17.510 -06:00 [INF] Executed DbCommand (10ms) [Parameters=[@p0='10251173', @p1='10251174', @p2='10251175'], CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
DELETE FROM [Users]
OUTPUT 1
WHERE [Id] = @p0;
DELETE FROM [Users]
OUTPUT 1
WHERE [Id] = @p1;
DELETE FROM [Users]
OUTPUT 1
WHERE [Id] = @p2;
2025-02-24 17:05:17.521 -06:00 [INF] Setting HTTP status code 204.
2025-02-24 17:05:17.521 -06:00 [INF] Executed endpoint 'DeleteUsers'
2025-02-24 17:05:17.521 -06:00 [INF] HTTP DELETE /users responded 204 in 139.5310 ms
2025-02-24 17:05:17.522 -06:00 [INF] Request finished HTTP/1.1 DELETE http://localhost:5257/users - 204 null null 141.6027ms
```

03

Foot Guns – Delayed or Otherwise

Or…How to negatively affect performance

Always Using Autogenerated Entities

The autogenerated entities are very useful; however, they include *all* the columns from the table, potentially resulting in too much data

Using a custom DTO plus Select() can result in a much smaller query (which will perform better at scale)

```
public record Post (int Id, string Body);
```

```
var post = await context.Posts
    .Where(p => p.Id == id)
    .Select(p =>
        new EntityFrameworkCoreUnchained.Data.CustomModels.Post
            (p.Id, p.Body))
    .FirstOrDefaultAsync();
```

```
2025-02-24 18:48:21.749 -06:00 [INF] Executed DbCommand (28ms)
SELECT TOP(1) [p].[Id], [p].[Body]
FROM [Posts] AS [p]
WHERE [p].[Id] = @__id_0
```

Not Using Async Methods

You should always prefer using –Async() methods where available

The system as a whole will perform better if threads are able to be shared/reused

```
var user = context.Users.Find(id);
```

```
var user = await context.Users.FindAsync(id);
```

Defeating EF Query Caching

Just like when writing T-SQL, parameterization matters for the internal query cache

Using variables instead of hard-coded values can reduce the number of queries EF has to cache, which improves performance

```
var firstPost = await context.Posts.FirstOrDefaultAsync(p ⇒ p.Title == "First Post");
var secondPost = await context.Posts.FirstOrDefaultAsync(p ⇒ p.Title == "Second Post");
```

```
var postTitle = "First Post";
var firstPost = await context.Posts.FirstOrDefaultAsync(p ⇒ p.Title == postTitle);
postTitle = "Second Post";
var secondPost = await context.Posts.FirstOrDefaultAsync(p ⇒ p.Title == postTitle);
```

Defeating Database Indexing

SQL Server (and other databases) improve performance through use of indexes but only if queries can take advantage of them

Check with your DBA team if you need more detail on how your specific database handles indexes

```
var howPost = await context.Posts.Where(p ⇒ p.Body.EndsWith("How")).FirstOrDefaultAsync();
```

```
var howPost = await context.Posts.Where(p ⇒ p.Body.StartsWith("How")).FirstOrDefaultAsync();
```

04

Database Design – First Time's the Charm

Or…how to create an impossible performance problem

General Guidance

Do it right the first time

- Changing the design becomes much more difficult once data is in the system

Design for your access patterns

- Think about how you're accessing the data – pure 3rd Normal Form may not always make sense

Use your database's tools/features

- Views, computed columns, column defaults are all your friend for reducing the work EF needs to do

Consider whether Table Per Type makes sense

- When mapping from the object design to the database, consider whether you should have a table per type or whether to split them up

Code First Best Practices

DATA TYPES

- Be sure to specify data types in code – otherwise, EF Core might choose for you
- In particular, string types can end up as NVARCHAR(MAX)

INDEXES

- No columns are indexed by default (database engine-dependent)
- Work with the DBA team to add the appropriate indexes

VIEWS

- Don't be afraid to use views – they can cut down on the amount of INCLUDE() and JOIN() you must write in your code
- They are especially valuable to abstract lookup tables away

MIGRATIONS

- Remember migrations run at startup – this might have some unfortunate side effects
- Consider running migrations as a separate deployment step

05

This Is Not the Tool You're Looking For

Or…there are better choices than EF Core for some things

Complex Data Operations

Lots of Joins

- EF Core does a pretty good job of translating LINQ into SQL
- If you have a lot (like, 10 or more) `JOIN()` or `INCLUDE()` clauses, that SQL is going to get ugly and may not perform well

ETL (or ELT)

- It's not an ETL tool – if you need to do ETL, use Azure Data Factory (or whatever your organization uses)
- If you must use C#, this is a really good case for using ADO.NET

Parameter Issues

- The database engine (particuarly SQL Server) is going to do what it wants with the queries EF Core sends it
- Sometimes, it will do the wrong thing, and that will be very difficult to solve outside of using `FromSql<>()` or one of its cousins

Providing Optimal Performance

Entity Framework Core's primary goal is reliable execution, so the SQL it generates will not be the most performant (especially against Cosmos DB)

EF Core provides performance on par with Dapper, but if you need control over the SQL used, consider using a lighter tool instead of FromSql<>()

06

Questions?

Thank You!

@danielmallott.bsky.social
github.com/danielmallott
linkedin.com/in/danielmallott
dmallott@westmonroe.com

west monroe