10 Secret Features in EF Core that Senior devs use

– that Middles and Juniors don't know.

Anton Martyniuk

1. Shadow Properties

EF Core lets you define properties that exist only in the database, not in your classes. Perfect for storing extra data like audit information without cluttering your entities.
Keeps your domain models clean and focused. Easy to use with Fluent API configuration.

```
public class ShadowPropertyInterceptor : SaveChangesInterceptor
{
    public override InterceptionResult<int> SavingChanges(
        DbContextEventData eventData, InterceptionResult<int> result)
    {
        var ctx = eventData.Context!;
        var now = DateTime.UtcNow;

        foreach (var entry in ctx.ChangeTracker.Entries())
        {
            if (entry.State == EntityState.Added)
            {
                entry.Property("created_at_utc").CurrentValue = now;
            }
            if (entry.State is EntityState.Added or EntityState.Modified)
            {
                entry.Property("updated_at_utc").CurrentValue = now;
            }
        }
        return base.SavingChanges(eventData, result);
    }
}
```

Anton Martyniuk

antondevtips.com

2. Query Tags

Easily add comments or tags to your SQL queries generated by EF Core. Great for tracking, debugging, and performance tuning in complex systems.
Helps quickly identify slow queries by adding context directly into your SQL logs. Simple yet powerful for troubleshooting database performance issues.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>(o ⇒
    o.UseSqlite("Data Source=app.db"));

var app = builder.Build();

app.MapGet("/blogs", async (AppDbContext db) ⇒
{
    // Add a query tag to help identify this query in SQL logs
    var blogs = await db.Blogs
        .TagWith("GET /blogs - HomePage Query")
        .ToListAsync();

    return Results.Ok(blogs);
    // Useful for tracking down slow queries in diagnostics tools
});

app.Run();
```

3. Compiled Queries

Precompile frequently used queries to improve query performance. Reduces the overhead of query parsing and planning each time it's executed.
Essential for high-performance applications where milliseconds matter. Enhances response time and efficiency of your database interactions.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>(o ⇒
    o.UseSqlite("Data Source=app.db"));

var app = builder.Build();

// Precompile the query and reuse it for better performance
static readonly Func<AppDbContext, string, Task<Blog?>> FindBlogByTitle =
    EF.CompileAsyncQuery((AppDbContext db, string title) ⇒
        db.Blogs.FirstOrDefault(b ⇒ b.Title == title));

app.MapGet("/blog/{title}", async (AppDbContext db, string title) ⇒
{
    // Fast, precompiled LINQ query
    var blog = await FindBlogByTitle(db, title);
    return blog is null ? Results.NotFound() : Results.Ok(blog);
});

app.Run();
```

Anton Martyniuk

4. DbContext Pooling

Reuse instances of your DbContext instead of creating new ones each time. Improves application performance and reduces memory usage significantly.
Ideal for high-load applications where context creation overhead is critical. Makes your app more efficient and scalable.

```
var builder = WebApplication.CreateBuilder(args);

// Enable DbContext pooling for performance (reuses instances)
builder.Services.AddDbContextPool<AppDbContext>(o =>
    o.UseSqlite("Data Source=app.db"));

var app = builder.Build();

app.MapGet("/blogs", async (AppDbContext db) =>
{
    // Pooled context reduces object allocations and improves perf
    var blogs = await db.Blogs.ToListAsync();
    return Results.Ok(blogs);
});

app.Run();
```

Anton Martyniuk

antondevtips.com

5. Value Converters

Automatically convert property values between your database and your .NET application. Useful for handling custom data types, enums, and formatting effortlessly.
Simplifies data transformations without writing manual conversions. Enhances readability and maintainability of your entity configurations.

```
public class Shipment
{
    public Guid Id { get; set; }
    public ShipmentNumber Number { get; set; }
    public CarrierName Carrier { get; set; }
    public ShipmentStatus Status { get; set; }
}

public class ShipmentConfiguration : IEntityTypeConfiguration<Shipment>
{
    public void Configure(EntityTypeBuilder<Shipment> builder)
    {
        builder.HasKey(x => x.Id);
        builder.HasIndex(x => x.Number);

        builder.Property(x => x.Carrier)
            .HasConversion(
                carrier => carrier.Value,
                value => new CarrierName(value)
            )
            .IsRequired();

        builder.Property(x => x.Status)
            .HasConversion<string>()
            .IsRequired();
    }
}
```

Anton Martyniuk

antondevtips.com

6. Temporal Tables

Automatically track history and changes to your data directly in the database. Great for auditing, rollback, and historical data analysis.
Eliminates manual audit tables or additional logic. Simplifies complex tasks related to data changes and audit trails.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services
.AddDbContext<AppDbContext>(o ⇒
    o.UseSqlServer("Data Source=.;Initial Catalog=app;Integrated Security=True"));

class AppDbContext(DbContextOptions<AppDbContext> opts) : DbContext(opts)
{
    public DbSet<Blog> Blogs ⇒ Set<Blog>();

    protected override void OnModelCreating(ModelBuilder b)
    {
        // Enable SQL Server Temporal Tables for Blog entity
        b.Entity<Blog>().ToTable("Blogs", b ⇒ b.IsTemporal());
    }
}
```

Anton Martyniuk

antondevtips.com

7. Database Seeding

Easily populate your database with new UseSeeding, UseAsyncSeeding methods in EF Core 9.
Great for automated testing and initial setup of your application environment. Simplifies database setup and ensures consistent data states.

```
builder.Services.AddDbContext<ApplicationDbContext>((provider, options) ⇒
{
    options
        .UseNpgsql(connectionString)
        .UseSeeding(dbContext, _, cancellationToken) ⇒
        {
            // The same seeding implementation here
        }
        .UseAsyncSeeding(async (dbContext, _, cancellationToken) ⇒
        {
            var authors = GetAuthors(3);
            var books = GetBooks(20, authors);

            await dbContext.Set<Author>().AddRangeAsync(authors);
            await dbContext.Set<Book>().AddRangeAsync(books);

            await dbContext.SaveChangesAsync();
        });
});
```

Anton Martyniuk

8. Split Queries for Includes

Improve query performance by splitting complex queries into multiple simpler queries. Prevents slow queries caused by complex JOIN operations.
Significantly reduces the number of database rows returned, improving app performance. Ideal for handling large datasets and complex relationships efficiently.

```
record Blog(int Id, string Title, List<Post> Posts);
record Post(int Id, string Content);

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>(o ⇒
    o.UseSqlite("Data Source=app.db"));

var app = builder.Build();

app.MapGet("/blogs-with-posts", async (AppDbContext db) ⇒
{
    // Use AsSplitQuery to avoid cartesian explosion with Includes
    var blogs = await db.Blogs
        .Include(b ⇒ b.Posts)
        .AsSplitQuery()
        .ToListAsync();
    return Results.Ok(blogs);
});

app.Run();
```

Anton Martyniuk

9. Raw SQL Queries

Execute plain SQL queries directly from your EF Core context.
Ideal when you need performance-optimized queries or database-specific features.
Provides greater control and flexibility for advanced scenarios.
Useful for optimizing complex database operations.

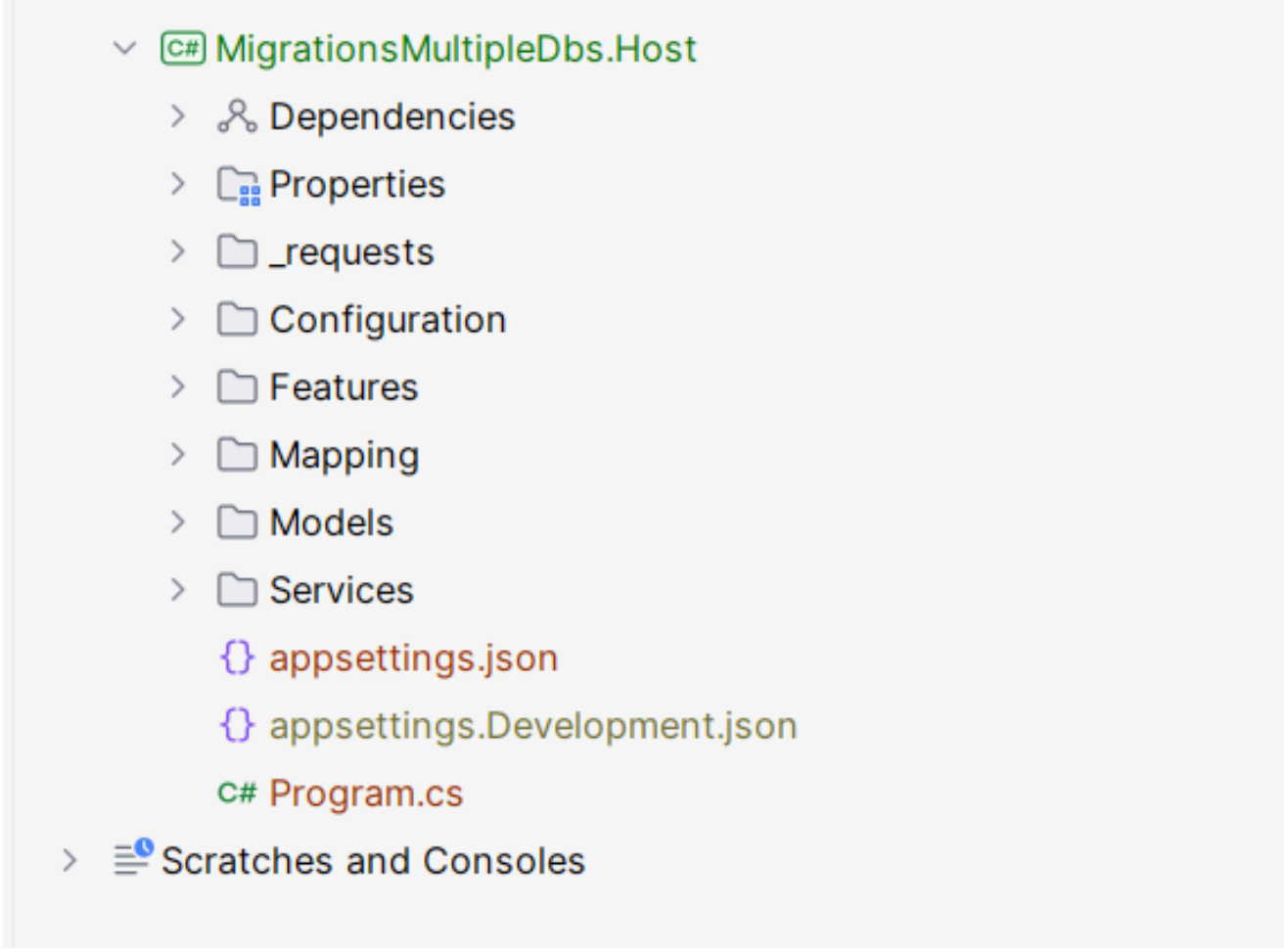
```
record Blog(int Id, string Title, List<Post> Posts);
record Post(int Id, string Content);

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>(o ⇒
    o.UseSqlite("Data Source=app.db"));

var app = builder.Build();

app.MapGet("/blogs-search", async (AppDbContext db, string keyword) ⇒
{
    // Directly execute raw SQL query (with parameterization!)
    var blogs = await db.Blogs
        .FromSql(
        $"SELECT * FROM Blogs WHERE Title LIKE {'%' + keyword + '%'}")
        .ToListAsync();

    return Results.Ok(blogs);
});

app.Run();
```

Anton Martyniuk

antondevtips.com

10. Create Database Migrations for Multiple Databases

Easily manage migrations and schema updates across different database systems. Streamlines development and testing across diverse environments.
Ideal for applications supporting multiple database providers like SQL Server, PostgreSQL, and SQLite. Reduces complexity and overhead managing multiple databases.



Next Steps

Hello there!
I'm Anton Martyniuk — a Microsoft MVP and Senior Tech Lead.

I have over 10 years of hands-on experience in .NET development and architecture. I've dedicated my career to empowering developers to excel in building robust, scalable systems.

Join my newsletter readers and let's build the future of .NET together!

Follow me on LinkedIn

01 I share amazing .NET and Software Development tips every day

Repost to your network

02 Share the knowledge with your network

Subscribe to my free newsletter

03 5 minutes every week to improve your .NET skills and learn how to craft better software

Anton Martyniuk

antondevtips.com