# Create Micro ORM EF Dapper

how to create own micro orm using .net core EF and Dapper

Creating your own **micro ORM** using **.NET Core** by combining **Entity Framework Core (EF Core)** and **Dapper** is a great learning exercise and practical approach to leverage both technologies' strengths:

- **EF Core**: Great for complex object tracking, migrations, and change tracking.
- **Dapper**: Super fast, lightweight, simple to execute raw SQL and map to POCOs.

---

## What is a Micro ORM?

A micro ORM usually provides:

- Simple CRUD operations
- Minimal or no change tracking
- Fast and lightweight query execution
- Minimal abstraction on top of the database

---

## High-Level Steps to Build Your Micro ORM

---

### 1. Setup the Project

- Create a **.NET Core** class library project.
- Add EF Core packages and Dapper via NuGet:

```bash
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer  # or your DB provider
dotnet add package Dapper
```

---

### 2. Create a Base Repository Interface

Define common CRUD methods:

```csharp
public interface IRepository<T> where T : class
{
    Task<T> GetByIdAsync(object id);
    Task<IEnumerable<T>> GetAllAsync();
    Task<int> AddAsync(T entity);
    Task<int> UpdateAsync(T entity);
    Task<int> DeleteAsync(object id);
}
```

---

### 3. Implement Repository with EF Core for Tracking

For operations requiring entity tracking (Add, Update, Delete):

```csharp
public class EfRepository<T> : IRepository<T> where T : class
{
    protected readonly DbContext _context;
    protected readonly DbSet<T> _dbSet;

    public EfRepository(DbContext context)
    {
        _context = context;
        _dbSet = _context.Set<T>();
    }

    public async Task<T> GetByIdAsync(object id)
    {
        return await _dbSet.FindAsync(id);
    }

    public async Task<IEnumerable<T>> GetAllAsync()
    {
        return await _dbSet.ToListAsync();
    }

    public async Task<int> AddAsync(T entity)
    {
        await _dbSet.AddAsync(entity);
        return await _context.SaveChangesAsync();
    }

    public async Task<int> UpdateAsync(T entity)
    {
        _dbSet.Update(entity);
        return await _context.SaveChangesAsync();
    }

    public async Task<int> DeleteAsync(object id)
    {
        var entity = await _dbSet.FindAsync(id);
        if (entity == null) return 0;
        _dbSet.Remove(entity);
        return await _context.SaveChangesAsync();
    }
}
```

---

### 4. Add a Dapper Query Layer for Fast Reads

Create a query class to run raw SQL with Dapper for fast, lightweight reads:

```csharp
using System.Data;
using Dapper;

public class DapperQuery<T> where T : class
{
    private readonly IDbConnection _connection;
    public DapperQuery(IDbConnection connection)
    {
        _connection = connection;
    }

    public async Task<IEnumerable<T>> QueryAsync(string sql, object parameters = null)
    {
        return await _connection.QueryAsync<T>(sql, parameters);
    }

    public async Task<T> QueryFirstOrDefaultAsync(string sql, object parameters = null)
    {
        return await _connection.QueryFirstOrDefaultAsync<T>(sql, parameters);
    }
}
```

---

### 5. Combine EF and Dapper in a Unit of Work or Service

Your application code can choose EF Core for writes and Dapper for reads:

```
csharp

public class MyService<T> where T : class
{
    private readonly EfRepository<T> _efRepo;
    private readonly DapperQuery<T> _dapperQuery;

    public MyService(EfRepository<T> efRepo, DapperQuery<T> dapperQuery)
    {
        _efRepo = efRepo;
        _dapperQuery = dapperQuery;
    }

    public Task<T> GetByIdAsync(object id) => _efRepo.GetByIdAsync(id);

    public Task<IEnumerable<T>> GetAllFastAsync()
    {
        var sql = $"SELECT * FROM {typeof(T).Name}s";  // Simple example; improve for safety
        return _dapperQuery.QueryAsync(sql);
    }

    public Task<int> AddAsync(T entity) => _efRepo.AddAsync(entity);

    public Task<int> UpdateAsync(T entity) => _efRepo.UpdateAsync(entity);

    public Task<int> DeleteAsync(object id) => _efRepo.DeleteAsync(id);
}
```

---

### 6. Additional Tips

- Use **reflection or attributes** to dynamically get table names or primary keys.
- Implement a method to build SQL dynamically but carefully to avoid SQL injection.
- Consider caching or compiled queries in Dapper for performance.
- Wrap transactions if you want atomicity across Dapper and EF Core.
- Use Dependency Injection to manage DbContext and IDbConnection lifecycle.

---

## Summary

| Feature | EF Core | Dapper | Your Micro ORM |
|---|---|---|---|
| Tracking | Yes | No | EF Core |
| Migrations | Yes | No | EF Core |
| Raw SQL Execution | Yes | Yes | Dapper |
| Query Speed | Moderate | Very Fast | Dapper for Read |
| Change Tracking & Updates | Yes | No | EF Core |
| Lightweight Abstraction | Moderate | Minimal | Mix of both |

---

If you want, I can help you build a small example project step-by-step or provide code templates for specific parts. Would you like that?

yes

Awesome! Let's build a simple **micro ORM** step-by-step using .NET Core, EF Core, and Dapper.

---

## Step 1: Create the Project

Open a terminal and create a solution and a class library:

```bash
mkdir MicroOrmDemo
cd MicroOrmDemo
dotnet new sln

dotnet new classlib -n MicroOrmLib
dotnet sln add MicroOrmLib/MicroOrmLib.csproj
```

Create a console app to test:

```bash
dotnet new console -n MicroOrmDemoApp
dotnet sln add MicroOrmDemoApp/MicroOrmDemoApp.csproj

dotnet add MicroOrmDemoApp reference MicroOrmLib/MicroOrmLib.csproj
```

---

## Step 2: Add EF Core & Dapper Packages

Add to both projects (or at least MicroOrmLib):

```bash
cd MicroOrmLib
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Dapper
cd ..
```

## Step 3: Setup Your EF Core DbContext & Entity

In **MicroOrmLib**, add a folder `Data` and create:

### AppDbContext.cs

```csharp
using Microsoft.EntityFrameworkCore;

namespace MicroOrmLib.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

        public DbSet<Person> People { get; set; }
    }
}
```

### Person.cs (entity)

```csharp
namespace MicroOrmLib.Data
{
    public class Person
    {
        public int Id { get; set; }
        public string Name { get; set; } = string.Empty;
        public int Age { get; set; }
    }
}
```

---

## Step 4: Create IRepository interface

### IRepository.cs

```csharp
using System.Collections.Generic;
using System.Threading.Tasks;

namespace MicroOrmLib.Data
{
    public interface IRepository<T> where T : class
    {
        Task<T?> GetByIdAsync(object id);
        Task<IEnumerable<T>> GetAllAsync();
        Task<int> AddAsync(T entity);
        Task<int> UpdateAsync(T entity);
        Task<int> DeleteAsync(object id);
    }
}
```

---

## Step 5: Implement EfRepository

### EfRepository.cs

```csharp
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace MicroOrmLib.Data
{
    public class EfRepository<T> : IRepository<T> where T : class
    {
        protected readonly AppDbContext _context;
        protected readonly DbSet<T> _dbSet;

        public EfRepository(AppDbContext context)
        {
            _context = context;
            _dbSet = _context.Set<T>();
        }

        public async Task<T?> GetByIdAsync(object id)
        {
            return await _dbSet.FindAsync(id);
        }

        public async Task<IEnumerable<T>> GetAllAsync()
        {
            return await _dbSet.ToListAsync();
        }

        public async Task<int> AddAsync(T entity)
        {
            await _dbSet.AddAsync(entity);
            return await _context.SaveChangesAsync();
        }

        public async Task<int> UpdateAsync(T entity)
        {
            _dbSet.Update(entity);
            return await _context.SaveChangesAsync();
        }

        public async Task<int> DeleteAsync(object id)
        {
            var entity = await _dbSet.FindAsync(id);
            if (entity == null) return 0;

            _dbSet.Remove(entity);
            return await _context.SaveChangesAsync();
        }
    }
}
```

---

## Step 6: Add Dapper Query Layer

### DapperQuery.cs

```csharp
using System.Collections.Generic;
using System.Data;
using System.Threading.Tasks;
using Dapper;

namespace MicroOrmLib.Data
{
    public class DapperQuery<T> where T : class
    {
        private readonly IDbConnection _connection;
        public DapperQuery(IDbConnection connection)
        {
            _connection = connection;
        }

        public async Task<IEnumerable<T>> QueryAsync(string sql, object? parameters = null)
        {
            return await _connection.QueryAsync<T>(sql, parameters);
        }

        public async Task<T?> QueryFirstOrDefaultAsync(string sql, object? parameters = null)
        {
            return await _connection.QueryFirstOrDefaultAsync<T>(sql, parameters);
        }
    }
}
```

---

## Step 7: Create a Service that Combines EF and Dapper

### PersonService.cs

```
csharp


using System.Collections.Generic;
using System.Data;
using System.Threading.Tasks;

namespace MicroOrmLib.Data
{
    public class PersonService
    {
        private readonly EfRepository<Person> _efRepository;
        private readonly DapperQuery<Person> _dapperQuery;

        public PersonService(EfRepository<Person> efRepository, DapperQuery<Person> dapperQuery)
        {
            _efRepository = efRepository;
            _dapperQuery = dapperQuery;
        }

        public Task<Person?> GetByIdAsync(int id)
        {
            return _efRepository.GetByIdAsync(id);
        }

        public Task<IEnumerable<Person>> GetAllFastAsync()
        {
            string sql = "SELECT * FROM People";
            return _dapperQuery.QueryAsync(sql);
        }

        public Task<int> AddAsync(Person person)
        {
            return _efRepository.AddAsync(person);
        }

        public Task<int> UpdateAsync(Person person)
        {
            return _efRepository.UpdateAsync(person);
        }

        public Task<int> DeleteAsync(int id)
        {
            return _efRepository.DeleteAsync(id);
        }
    }
}
```

---

## Step 8: Setup and Test in Console App

Edit **Program.cs** in **`MicroOrmDemoApp`**:

```
csharp

using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using MicroOrmLib.Data;
using System.Data;

var connectionString = "Server=
(localdb)\\mssqllocaldb;Database=MicroOrmDemoDb;Trusted_Connection=True;";

var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>();
optionsBuilder.UseSqlServer(connectionString);

using var context = new AppDbContext(optionsBuilder.Options);

// Make sure database is created
context.Database.EnsureCreated();

// Create Dapper connection
using IDbConnection dbConnection = new SqlConnection(connectionString);
dbConnection.Open();

// Create repositories and service
var efRepo = new EfRepository<Person>(context);
var dapperQuery = new DapperQuery<Person>(dbConnection);
var personService = new PersonService(efRepo, dapperQuery);

// Add a person
await personService.AddAsync(new Person { Name = "Alice", Age = 30 });

// Get all people fast with Dapper
var people = await personService.GetAllFastAsync();

foreach (var person in people)
{
    Console.WriteLine($"Person: {person.Id} - {person.Name} - {person.Age}");
}
```

---

## Step 9: Run Your App

```
bash

cd MicroOrmDemoApp
dotnet run
```

You should see your inserted person printed out, showing EF Core is handling the insert and Dapper is doing a fast read.

---

## Summary:

- **EF Core** handles the write operations and change tracking.
- **Dapper** does the fast raw SQL reads.
- You wrap them inside a service class to provide a clean API.

---

If you want, I can help you improve this with features like dynamic SQL generation, transactions, or integration with dependency injection and configuration. Would you like to do that next?

yes

Perfect! Let’s level up your micro ORM with **Dependency Injection (DI)**, **configuration**, and **transaction support** — all very important for real-world apps.

---

## Step 10: Add Dependency Injection & Configuration

---

### 1. Add `Microsoft.Extensions.DependencyInjection` and `Microsoft.Extensions.Configuration`

Add these NuGet packages to both projects (especially your app project):

```bash
dotnet add MicroOrmDemoApp package Microsoft.Extensions.DependencyInjection
dotnet add MicroOrmDemoApp package Microsoft.Extensions.Configuration
dotnet add MicroOrmDemoApp package Microsoft.Extensions.Configuration.Json
dotnet add MicroOrmDemoApp package Microsoft.Extensions.Hosting
dotnet add MicroOrmDemoApp package Microsoft.Extensions.Logging.Console
```

---

### 2. Setup Configuration and DI in your Console App

Replace your **Program.cs** with a generic host setup to wire everything up nicely:

```csharp
using System.Data;
using System.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using MicroOrmLib.Data;

var host = Host.CreateDefaultBuilder(args)
    .ConfigureAppConfiguration((context, config) =>
    {
        config.AddJsonFile("appsettings.json", optional: false);
    })
    .ConfigureServices((context, services) =>
    {
        var connectionString = context.Configuration.GetConnectionString("DefaultConnection");

        // Register DbContext with connection string
        services.AddDbContext<AppDbContext>(options =>
            options.UseSqlServer(connectionString));

        // Register IDbConnection as transient, so new connection per scope
        services.AddTransient<IDbConnection>(sp => new SqlConnection(connectionString));

        // Register repositories and services
        services.AddScoped(typeof(IRepository<>), typeof(EfRepository<>));
        services.AddScoped(typeof(DapperQuery<>));
        services.AddScoped<PersonService>();

        // Add logging
        services.AddLogging(config => config.AddConsole());
    })
    .Build();

// Run a scope to test service
using var scope = host.Services.CreateScope();

var personService = scope.ServiceProvider.GetRequiredService<PersonService>();
var logger = scope.ServiceProvider.GetRequiredService<ILogger<Program>>();

// Ensure database is created
var context = scope.ServiceProvider.GetRequiredService<AppDbContext>();
context.Database.EnsureCreated();

// Add and retrieve people
await personService.AddAsync(new Person { Name = "Bob", Age = 28 });

var people = await personService.GetAllFastAsync();

foreach (var person in people)
{
    logger.LogInformation("Person {Id} - {Name} - {Age}", person.Id, person.Name, person.Age);
}

await host.RunAsync();
```

---

### 3. Create appsettings.json in your console project:

```json
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MicroOrmDemoDb;Trusted_Connection=True;"
  }
}
```

---

## Step 11: Add Transaction Support

We want to execute EF and Dapper queries inside the same transaction if needed.

---

### 1. Add Unit of Work pattern

Create **IUnitOfWork.cs**:

```csharp
using System.Threading.Tasks;

namespace MicroOrmLib.Data
{
    public interface IUnitOfWork
    {
        Task BeginTransactionAsync();
        Task CommitAsync();
        Task RollbackAsync();
    }
}
```

---

## 2. Implement UnitOfWork with EF Core DbContext transaction and share connection for Dapper

### UnitOfWork.cs

```
csharp

using System;
using System.Data;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore.Storage;

namespace MicroOrmLib.Data
{
    public class UnitOfWork : IUnitOfWork, IDisposable
    {
        private readonly AppDbContext _context;
        private IDbTransaction? _dbTransaction;

        public UnitOfWork(AppDbContext context)
        {
            _context = context;
        }

        public async Task BeginTransactionAsync()
        {
            if (_dbTransaction != null)
                throw new InvalidOperationException("Transaction already started.");

            await _context.Database.OpenConnectionAsync();

            _dbTransaction = await _context.Database.BeginTransactionAsync();
        }

        public async Task CommitAsync()
        {
            if (_dbTransaction == null)
                throw new InvalidOperationException("No active transaction.");

            await _context.SaveChangesAsync();
            await _dbTransaction.CommitAsync();
            await _context.Database.CloseConnectionAsync();

            _dbTransaction.Dispose();
            _dbTransaction = null;
        }

        public async Task RollbackAsync()
        {
            if (_dbTransaction == null)
                throw new InvalidOperationException("No active transaction.");

            await _dbTransaction.RollbackAsync();
            await _context.Database.CloseConnectionAsync();

            _dbTransaction.Dispose();
            _dbTransaction = null;
        }

        public void Dispose()
        {
            _dbTransaction?.Dispose();
            _context.Dispose();
        }
    }
}
```

---

## 3. Inject `IUnitOfWork` into your service and update methods

Modify **PersonService.cs** to use the `UnitOfWork` for transactional operations:

```
csharp

using System.Collections.Generic;
using System.Threading.Tasks;

namespace MicroOrmLib.Data
{
    public class PersonService
    {
        private readonly EfRepository<Person> _efRepository;
        private readonly DapperQuery<Person> _dapperQuery;
        private readonly IUnitOfWork _unitOfWork;

        public PersonService(EfRepository<Person> efRepository, DapperQuery<Person> dapperQuery,
IUnitOfWork unitOfWork)
        {
            _efRepository = efRepository;
            _dapperQuery = dapperQuery;
            _unitOfWork = unitOfWork;
        }

        public Task<Person?> GetByIdAsync(int id)
            => _efRepository.GetByIdAsync(id);

        public Task<IEnumerable<Person>> GetAllFastAsync()
        {
            var sql = "SELECT * FROM People";
            return _dapperQuery.QueryAsync(sql);
        }

        public async Task<int> AddAsync(Person person)
        {
            await _unitOfWork.BeginTransactionAsync();
            try
            {
                var result = await _efRepository.AddAsync(person);
                await _unitOfWork.CommitAsync();
                return result;
            }
            catch
            {
                await _unitOfWork.RollbackAsync();
                throw;
            }
        }

        // similarly update UpdateAsync and DeleteAsync for transactions if needed
    }
}
```

---

## 4. Register UnitOfWork in DI

In your console app DI setup:

```
csharp

services.AddScoped<IUnitOfWork, UnitOfWork>();
```

## Step 12: Final Notes and Next Steps

- You now have DI and configuration to cleanly wire dependencies.
- Your micro ORM supports transactions over EF and Dapper.
- You can expand `UnitOfWork` to expose the `IDbConnection` and `IDbTransaction` to Dapper, so queries share the transaction — an advanced next step.

---

If you want, I can help you:

- Extend your micro ORM with **dynamic SQL builders** to avoid raw SQL strings
- Add **async batch operations**
- Handle **soft deletes** or audit fields automatically
- Or even build a **code-first migrations integration** wrapper

What do you want to focus on next?