ASP.NET Core

Présenté par Tidiane diallo

→

Qu’est-ce qu’un DTO ?

Un DTO (Data Transfer Object) est une classe utilisée pour transférer des données entre différentes couches ou parties d’une application, notamment entre l’API et le client. Contrairement aux entités de la base de données, un DTO est conçu pour contenir uniquement les données nécessaires à une opération particulière.

Pourquoi utiliser des DTOs ?

- Séparation des responsabilités:
  - Cela empêche d'exposer des informations sensibles ou inutiles de votre modèle de données.
- Contrôle des données échangées:
  - Avec un DTO, vous pouvez contrôler exactement quelles propriétés sont envoyées ou reçues.
- Flexibilité
  - Vous pouvez transformer ou enrichir les données avant de les transmettre au client.
- Facilitation des validations
  - Les DTOs peuvent inclure des annotations de validation comme [Required], [MaxLength], etc., pour s'assurer que les données envoyées par le client respectent certaines règles.

Résumé

Les DTOs permettent de :

- Contrôler les données échangées entre les clients et l'API.
- Protéger votre modèle de base de données.
- Faciliter les validations et les transformations.
- Maintenir une flexibilité et une modularité dans votre code.

Mapping (mappage)

Qu’est-ce que le mapping ?

Le **mapping** consiste à convertir ou copier les données d’un objet vers un autre. Dans le contexte des applications ASP.NET Core, cela signifie généralement **mapper**

- Les données d’un DTO (Data Transfer Object) vers une entité.
- Ou les données d’une entité vers un DTO.

Le mapping est utilisé pour s’assurer que chaque couche de votre application (par exemple, la couche de persistance et la couche d’API) manipule uniquement les objets qui lui sont destinés.

Mapping automatique avec AutoMapper

AutoMapper est une bibliothèque populaire en .NET qui permet de mapper automatiquement les propriétés similaires entre deux objets.

Les données d'un DTO (Data Transfer Object) vers une entité. Ou les données d'une entité vers un DTO. Le mapping est utilisé pour s'assurer que chaque couche de votre application (par exemple, la couche de persistance et la couche d'API) manipule uniquement les objets qui lui sont destinés.

Étapes pour configurer et utiliser AutoMapper

Ajoutez AutoMapper et AutoMapper.Extensions.Microsoft.DependencyInjection à votre projet en utilisant la commande suivante dans le terminal :

1. Installer le package NuGet

```
dotnet add package AutoMapper
dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection
```

Étapes pour configurer et utiliser AutoMapper

2. Créer un profil de mappage

```
public class BookMapping : Profile
{
    public BookMapping()
    {
        CreateMap<Book, BookDto>();
        CreateMap<CreateBookDto, Book>();
    }
}
```

Étapes pour configurer et utiliser AutoMapper

3. Configurer AutoMapper dans le projet

```
using AutoMapper;
builder.Services.AddAutoMapper(typeof(Program));
```

Étapes pour configurer et utiliser AutoMapper

4. Utiliser AutoMapper dans vos endpoints

Injectez l’interface IMapper pour effectuer le mappage.

Exemple : Endpoint pour récupérer une liste de livres (Mapping de Book vers BookDto)

Introduction à Entity Framework Core (EF Core)

Entity Framework Core (EF Core) est un ORM (Object-Relational Mapper) open source pour .NET. Il simplifie les interactions entre une application .NET et une base de données en permettant de manipuler les données comme des objets C# sans écrire de requêtes SQL complexes.

En termes simples, EF Core fait le lien entre vos classes C# (entités) et votre base de données, en vous permettant d'effectuer des opérations CRUD (Create, Read, Update, Delete) avec des objets.

Pourquoi utiliser EF Core ?

- Code maintenable : Les entités et leurs relations sont définies dans le code, ce qui facilite les modifications et la compréhension du projet.
- Indépendance du fournisseur : EF Core prend en charge plusieurs bases de données (SQL Server, MySQL, PostgreSQL, SQLite, etc.).
- Migrations : Vous pouvez facilement créer, modifier ou synchroniser les schémas de base de données avec vos entités via les migrations.
- Support des relations complexes : EF Core gère les relations entre tables (un-à-un, un-à-plusieurs, plusieurs-à-plusieurs).
- nteropérabilité LINQ : Vous pouvez écrire des requêtes avec LINQ (Language-Integrated Query) pour interroger les données.

Comment fonctionne EF Core ?

EF Core repose sur trois concepts principaux :

1. **Entités** : Ce sont des classes C# qui représentent les tables de votre base de données.

Exemple: Une entité Book correspondra à une table Books.

```
namespace GestionBibilotheque.Api.Entities
{
    public class Book
    {
        public int Id { get; set; }
        public required string Title { get; set; }
        public required string Author { get; set; }
        public DateOnly DatePub{ get; set; }
    }
}
```

install

- dotnet add package Microsoft.EntityFrameworkCore.Sqlite --version 8.0.11
- dotnet tool install --global dotnet-ef
- dotnet add package Microsoft.EntityFrameworkCore.Design
- dotnet ef migrations add InitialCreate
- dotnet ef database update

Comment fonctionne EF Core ?

2. DbContext : Une classe spéciale qui gère la connexion à la base de données et le suivi des entités.

Exemple : BookDbContext permet à EF Core de gérer la table Books.

```
public class BookDbContext : DbContext
{
    public BookDbContext(DbContextOptions<BookDbContext> options) : base(options) { }

    public DbSet<Book> Books { get; set; }
}
// ou bien avec Primary constructor
public class BookDbContext(DbContextOptions<BookDbContext> options) : DbContext(options)
{
    public DbSet<Book>? Books { get; set; }

}
```

Ajouter

Comment fonctionne EF Core ?

3. Migrations : EF Core utilise un système de migrations pour gérer les modifications du schéma de la base de données.

dotnet ef migrations add InitialCreate dotnet ef database update

COnfigurer EF Core dans Program.cs

Ajouter dans Program.cs

```
// Ajouter EF Core Sqlite
//var connectionString = "Data  Souce = Bibliotheque.db";
var connectionString = builder.Configuration.GetConnectionString("Bibliotheque");
builder.Services.AddSqlite<BookDbContext>(connectionString);
var builder = WebApplication.CreateBuilder(args);



 // Ajouter EF Core avec SQL Server
builder.Services.AddDbContext<BookDbContext>(options ⇒
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();
```

Tidiane Diallo

Développeur Fullstack et Formateur

Entity Framework Core

1. Installation des packages nécessaires

Ajoutez les packages NuGet nécessaires dans votre projet pour travailler avec EF Core. Cela peut être fait via la console du gestionnaire de packages ou le fichier `.csproj`.

Exemple pour SQLite :

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="8.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="8.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="8.0.0" />
</ItemGroup>
```

2. Création d’un contexte de base de données

Créez une classe héritant de `DbContext` pour gérer vos entités.

Exemple :

```
using Microsoft.EntityFrameworkCore;

namespace MonProjet.Data;

public class BookDbContext : DbContext
{
    public BookDbContext(DbContextOptions<BookDbContext> options) : base(options) { }

    public DbSet<Book> Books { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Book>().HasData(
            new Book { Id = 1, Title = "Clean Code", Author = "Robert C. Martin", DatePub = new DateTime(2008, 8, 1) },
            new Book { Id = 2, Title = "The Pragmatic Programmer", Author = "Andrew Hunt", DatePub = new DateTime(1999,
        );
    }
}
```

3. Configuration de la chaîne de connexion

Ajoutez une chaîne de connexion dans `appsettings.json` pour spécifier la base de données.

Exemple :

```
{
  "ConnectionStrings": {
    "Bibliotheque": "Data Source=Bibliotheque.db"
  }
}
```

4. Configuration dans `Program.cs`

Configurez EF Core pour utiliser la chaîne de connexion.

Exemple :

```
var builder = WebApplication.CreateBuilder(args);
var connectionString = builder.Configuration.GetConnectionString("Bibliotheque");
builder.Services.AddSqlite<BookDbContext>(connectionString);
var app = builder.Build();
```

5. Création des migrations

Créez et appliquez les migrations avec les outils EF Core :

Commandes :

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

6. Utilisation dans l’application

Injectez `BookDbContext` dans vos contrôleurs ou endpoints pour effectuer des opérations CRUD.

Exemple :

```
app.MapGet("/books", async (BookDbContext dbContext) ⇒
{
    var books = await dbContext.Books.ToListAsync();
    return Results.Ok(books);
});
```

7. Migration automatique de la base de données

Ajoutez une méthode pour appliquer automatiquement les migrations lors du démarrage de l'application.

Exemple :

```
public static class DataExtensions
{
    public static async Task MigrateDbAsync(this WebApplication app)
    {
        using var scope = app.Services.CreateScope();
        var dbContext = scope.ServiceProvider.GetRequiredService<BookDbContext>();
        await dbContext.Database.MigrateAsync();
    }
}

// Appel dans `Program.cs` :
await app.MigrateDbAsync();
```

Avec ces étapes, votre application est configurée pour utiliser Entity Framework Core avec une base de données SQLite.