Université Cheikh Anta Diop
École Supérieure Polytechnique (ESP)
Département Génie Informatique
M2GLSI SOIR
2ème année

Année Universitaire 2023/2024
Enseignement .NET
Travaux Pratiques ADO .NET
Chargé de Cours E. H. Ousmane Diallo
22/11/2024

Utiliser une base de données avec une API minimale, Entity Framework Core et ASP.NET Core

Découvrir comment utiliser une base de données avec une API *minimale*.

Objectifs d'apprentissage

Dans ce module, vous allez :

- Découvrez comment ajouter Entity Framework Core à une application API minimale.
- Persistance des données dans un magasin de données en mémoire.
- Persistance des données dans une base de données SQLite.
- Test API

Prérequis

- .NET6
- Notions de base sur ce qu'est une API

Introduction

Lorsque vous générez une application web qui traite des données, vous souhaitez probablement stocker ces données dans une base de données. Heureusement, les API minimales générées sur ASP.NET Core peuvent être facilement intégrées à un grand nombre de bases de données à l'aide d'Entity Framework (EF) Core.

Scénario : Créer un prototype

Vous êtes développeur dans une équipe. Vous avez généré une API qui gère les opérations de création, lecture, mise à jour et suppression (CRUD) sur une table de données. Vous envisagez de générer une application frontale qui utilise cette API. Vous souhaitez stocker les données dans une base de données pour pouvoir utiliser les données dans votre application frontale.

Ce que vous allez apprendre ?

Vous allez apprendre à utiliser EF Core pour conserver vos données, d'abord dans une base de données en mémoire, puis dans SQLite. Vous allez également apprendre à utiliser EF Core pour interroger la base de données.

Quel est l'objectif principal ?

Ajouter la prise en charge de base de données à une application API minimale.

Qu'est-ce qu'Entity Framework Core ?

La plupart des applications web non triviales devront exécuter en toute fiabilité des opérations de création, lecture, mise à jour et suppression (CRUD) sur les données. Elles devront également conserver ces modifications entre les redémarrages de l'application. Bien qu'il existe différentes options de persistance des données dans les applications .NET, Entity Framework (EF) Core est une solution conviviale et adaptée à de nombreuses applications .NET.

Comprendre EF Core

EF Core est une technologie d'accès aux données légère, extensible, open source et multiplateforme pour les applications .NET.

EF Core peut servir de mappeur objet-relationnel qui :

- Permet aux développeurs .NET de travailler avec une base de données à l'aide d'objets .NET.
- Élimine la nécessité d'une grande partie du code d'accès aux données qui doit généralement être écrit.

EF Core prend en charge un grand nombre de bases de données populaires, notamment SQLite, MySQL, PostgreSQL, Oracle et Microsoft SQL Server.

Le modèle

Avec Entity Framework Core, l'accès aux données est effectué à l'aide d'un modèle. Un modèle est constitué de classes d'entité et d'un objet *Contexte* qui représente une session avec la base de données. L'objet *Contexte* permet l'interrogation et l'enregistrement des données.

La classe d'entité

Dans ce scénario, vous implémentez une API de gestion des pizzas, vous allez donc utiliser une classe d'entité `Pizza`. Les pizzas de votre magasin auront un nom et une description. Elles auront également besoin d'un ID pour permettre à l'API et à la base de données de les identifier. La classe d'entité `Pizza` que vous utiliserez dans votre application identifie les pizzas :

```
namespace PizzaStore.Models
{
  public class Pizza
  {
      public int Id { get; set; }
      public string? Nom { get; set; }
```

```
        public string? Description { get; set; }
    }
}
```

La classe *Contexte*

Cette application ne possède qu'une seule classe d'entité, mais la plupart des applications en auront plusieurs. La classe *Contexte* est responsable de l'interrogation et de l'enregistrement des données dans vos classes d'entité, ainsi que de la création et de la gestion de la connexion de base de données.

Effectuer des opérations CRUD avec EF Core

Après configuration d'EF Core, vous pouvez l'utiliser pour effectuer des opérations CRUD sur vos classes d'entité. Ensuite, vous pouvez développer sur des classes C#, en déléguant les opérations de base de données à la classe *Contexte*. Les fournisseurs de base de données le traduisent à leur tour en langage de requête spécifique à la base de données. SQL pour une base de données relationnelle en est un exemple. Les requêtes sont toujours exécutées sur la base de données, même si les entités retournées dans le résultat existent déjà dans le contexte.

Interroger des données

L'objet *Contexte* expose une classe *collection* pour chaque type d'entité. Dans l'exemple précédent, la classe *Contexte* expose une collection d'objets `Pizza` en tant que `Pizzas`. Étant donné que nous disposons d'une instance de la classe *Contexte*, vous pouvez interroger la base de données pour toutes les pizzas :

```
var pizzas = await db.Pizzas.ToListAsync();
```

Insertion des données

Vous pouvez utiliser le même objet *contexte* pour insérer une nouvelle pizza :

```
await db.pizzas.AddAsync(
    new Pizza { ID = 1, Nom = "Pepperoni", Description = "The classic pepperoni pizza" });
```

Suppression de données

Les opérations de suppression sont simples. Elles n'ont besoin que d'un ID de l'élément à supprimer :

```
var pizza = await db.pizzas.FindAsync(id);
if (pizza is null)
```

```
{
    //Handle error
}
db.pizzas.Remove(pizza);
```

Mettre à jour des données

De même, vous pouvez mettre à jour une pizza existante :

```
int id = 1;
var updatepizza = new Pizza { Nom = "Ananas", Description = "Ummmm?" })
var pizza = await db.pizzas.FindAsync(id);
if (pizza is null)
{
    //Handle error
}
pizza.Item = updatepizza.Item;
pizza.IsComplete = updatepizza.IsComplete;
await db.SaveChangesAsync();
```

Utiliser la base de données en mémoire d'EF Core

EF Core comprend un fournisseur de base de données en mémoire qui peut être utilisé pour tester l'application. Le fournisseur de base de données en mémoire est utile pour le test et le développement, mais il ne doit pas être utilisé en production. Vous allez utiliser le fournisseur de base de données en mémoire pour créer une base de données et effectuer des opérations CRUD sur celle-ci.

Exercice : Ajouter EF Core à l'API minimale

Ce module utilise l'interface CLI .NET et Visual Studio Code pour le développement local. Vous pouvez appliquer les concepts avec Visual Studio (Windows) ou Visual Studio pour Mac (macOS), ou poursuivre le développement avec Visual Studio Code (Windows, Linux et macOS).

Ce module utilise le SDK .NET 6.0. Vérifiez que .NET 6.0 est installé en exécutant la commande suivante dans votre terminal par défaut :

CLI .NETCopier

```
dotnet --list-sdks
```

Une sortie similaire à ce qui suit s'affiche :

```
3.1.100 [C:\program files\dotnet\sdk]
```

```
5.0.100 [C:\program files\dotnet\sdk]
6.0.100 [C:\program files\dotnet\sdk]
```

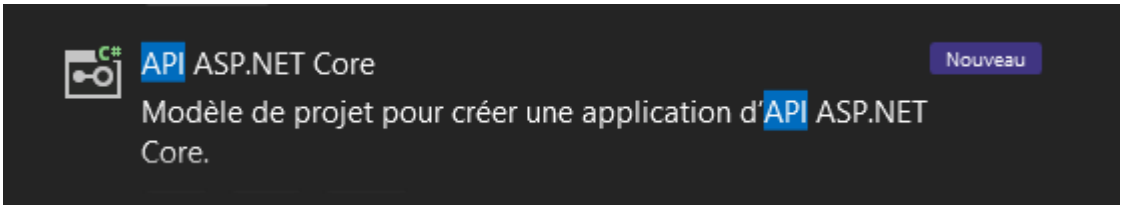
Vérifiez que la liste comporte une version commençant par 6. S'il n'y en a pas ou que la commande est introuvable, installez la dernière version du kit SDK .NET 6.0.

Configuration du projet

Tout d'abord, vous devez créer un projet. Vous avez installé .NET 6 et vous êtes prêt à travailler. Dans cette leçon, vous allez ajouter la persistance des données à une API de gestion des pizzas.

1. Dans un terminal, créez une API web en exécutant `dotnet new` :

   ```
   dotnet new web -o PizzaStore -f net6.0
   ```

   Vous devriez voir le répertoire *PizzaStore*.

2. Accédez au répertoire *PizzaStore* en entrant la commande suivante :

   ```
   cd PizzaStore
   ```

3. Installez le package Swashbuckle :

   CLI .NETCopier

   ```
   dotnet add package Swashbuckle.AspNetCore --version 6.2.3
   ```

4. Ouvrez le projet dans *Visual Studio* / *Visual Studio Code*.
5. Créez un fichier nommé *Pizza.cs* à la racine du projet et donnez-lui le contenu suivant :

   ```
   namespace PizzaStore.Models
   {
       public class PizzaEhod
       {
             public int IdEhod { get; set; }
             public string? NomEhod { get; set; }
             public string? DescriptionEhod { get; set; }
       }
   }
   ```

La classe `Pizza` précédente est un objet simple qui représente une pizza. Ce code est votre modèle de données. Plus tard, vous utiliserez Entity Framework (EF) Core pour mapper ce modèle de données à une table de base de données.

6. Ouvrez *Program.cs* et ajoutez le code en surbrillance :

```
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
     c.SwaggerDoc("v1", new OpenApiInfo {
         Title = " API PizzaStore ",
         Description = " Faire les pizzas que vous aimez ",
         Version = "v1" });
});

var app = builder.Build();
app.UseSwagger();
app.UseSwaggerUI(c =>
{
   c.SwaggerEndpoint("/swagger/v1/swagger.json", "PizzaStore API V1");
});

app.MapGet("/", () => "Bonjour Sénégal!");

app.Run();
```

Vous pouvez recevoir une invite de Visual Studio Code pour ajouter des ressources afin de déboguer le projet. Cliquez sur `Yes` dans la boîte de dialogue.

Ajouter EF Core au projet

Pour stocker les éléments dans la liste des tâches, installez le package `EntityFrameworkCore.InMemory`.

1. Appuyez sur **Ctrl+`** pour ouvrir un terminal dans Visual Studio Code. Dans le nouveau terminal, entrez le code suivant pour ajouter le package en mémoire EF Core :

```
dotnet add package Microsoft.EntityFrameworkCore.InMemory --version 6.0
```

2. Ajoutez `using Microsoft.EntityFrameworkCore;` au début de vos fichiers *Program.cs* et *Pizza.cs*.

Maintenant que vous avez ajouté EF Core au projet, vous pouvez associer votre code aux données que vous souhaitez enregistrer et interroger. Pour cette étape, vous créez une classe `PizzaDb`. La classe `PizzaDb` effectuera les tâches suivantes :

- Expose la propriété *Pizzas* à partir de la liste de `Pizzas` dans la base de données.
- Utilise `UseInMemoryDatabase` pour associer le stockage de base de données en mémoire. Les données sont stockées ici tant que l'application s'exécute.

3. Pour configurer la base de données en mémoire, ajoutez le code suivant en bas du fichier *Pizza.cs* (au-dessus du `}` final). Vous aurez deux définitions de classe dans l'espace de noms `PizzaStore.Models`.

```
class PizzaDb : DbContext
{
    public PizzaDb(DbContextOptions options) : base(options) { }
    public DbSet<Pizza> Pizzas { get; set; } = null!;
}
```

`DbContext` représente une connexion ou session utilisée pour interroger et enregistrer des instances d'entités dans une base de données.

4. Ajoutez `using PizzaStore.Models;` en haut du fichier *Program.cs*.
5. Dans *Program.cs*, avant d'appeler `AddSwaggerGen`, ajoutez le code suivant :

```
builder.Services.AddDbContext<PizzaDb>(options =>
options.UseInMemoryDatabase("items"));
```

Retourner une liste d'éléments

- Pour lire dans une liste d'éléments de la liste de pizzas, ajoutez le code suivant au-dessus de l'appel à `app.Run();` pour ajouter un itinéraire « /pizzas » :

```
app.MapGet("/pizzas", async (PizzaDb db) => await db.Pizzas.ToListAsync());
```

Exécution de l'application

1. Vérifiez que vous avez enregistré tous vos changements. Exécutez l'application. Cette action génère l'application et l'héberge sur un port compris entre 5000 et 5300. Un port HTTPS est sélectionné dans la plage comprise entre 7000 et 7300.

Notes

Si vous souhaitez remplacer le comportement de sélection aléatoire des ports, vous pouvez définir les ports à utiliser dans *launchSettings.json*.

```
dotnet run
```

Voici à quoi peut ressembler la sortie dans le terminal :

SortieCopier

```
Building...
 info: Microsoft.Hosting.Lifetime[14]
       Now listening on: https://localhost:7200
 info: Microsoft.Hosting.Lifetime[14]
       Now listening on: http://localhost:5100
 info: Microsoft.Hosting.Lifetime[0]
       Application started. Press Ctrl+C to shut down.
 info: Microsoft.Hosting.Lifetime[0]
       Hosting environment: Development
 info: Microsoft.Hosting.Lifetime[0]
       Content root path: /<path>/PizzaStore
```

2. Dans votre navigateur, accédez à *https://localhost:{PORT}/swagger*. Sélectionnez le bouton `GET /pizzas`, suivi de **Essayer** et **Exécuter**. Vous verrez que la liste est vide sous `Response body`.
3. Dans le premier terminal, appuyez sur **Ctrl + C** pour arrêter l'exécution du programme.

Créer des éléments

Ajoutons le code aux nouveaux éléments `POST` de la liste des pizzas. Dans *Program.cs*, ajoutez le code suivant sous le `app.MapGet` créé précédemment.

C#Copier

```
app.MapPost("/pizza", async (PizzaDb db, Pizza pizza) =>
{
    await db.Pizzas.AddAsync(pizza);
    await db.SaveChangesAsync();
    return Results.Created($"/pizza/{pizza.Id}", pizza);
});
```

Tester l'API

Vérifiez que vous avez enregistré toutes vos modifications et réexécutez l'application. Revenez à l'interface utilisateur Swagger, vous devriez maintenant voir `POST/pizza`. Pour ajouter de nouveaux éléments à la liste de pizzas :

1. Sélectionnez **POST /pizza**.
2. Sélectionnez **Essayer**.
3. Remplacez le corps de la requête par ce qui suit :

   JSONCopier

   ```
   {
       "name": "Pepperoni",
       "description": "A classic pepperoni pizza"
   }
   ```

4. Sélectionnez **Exécuter**.

Pour lire les éléments de la liste :

1. Sélectionnez **GET /pizzas**.
2. Sélectionnez **Essayer**.
3. Sélectionnez **Exécuter**.

   `Response body` inclura les éléments qui viennent d'être ajoutés.

   JSONCopier

   ```
   [
     {
       "id": 1,
       "name": "Pepperoni",
       "description": "A classic pepperoni pizza"
     }
   ]
   ```

4. Appuyez sur **Ctrl+C** dans le terminal pour arrêter l'exécution de l'application. Pour le reste de cet exercice, arrêtez et redémarrez l'application comme vous le souhaitez pour tester vos modifications. Veillez à enregistrer toutes vos modifications avant de `dotnet run`!

Obtenir un seul élément

Pour obtenir (GET) un élément par `id`, ajoutez ce code sous l'itinéraire `app.MapPost` que vous avez créé précédemment.

```
app.MapGet("/pizza/{id}", async (PizzaDb db, int id) => await db.Pizzas.FindAsync(id));
```

Tester GET par ID

Pour vérifier cela, vous pouvez accéder à *https://localhost:{PORT}/pizza/1* ou utiliser l'interface utilisateur Swagger. Étant donné que vous utilisez une base de données en mémoire, la pizza que vous avez créée précédemment ne sera pas répertoriée si vous avez redémarré l'application. Vous devrez donc l'entrer à nouveau.

Mettre à jour un élément

Pour mettre à jour un élément existant, ajoutez ce code sous l'itinéraire `GET /pizza/{id}` que vous avez créé :

C#Copier

```
app.MapPut("/pizza/{id}", async (PizzaDb db, Pizza updatepizza, int id) =>
{
      var pizza = await db.Pizzas.FindAsync(id);
      if (pizza is null) return Results.NotFound();
      pizza.Nom = updatepizza.Nom;
      pizza.Description = updatepizza.Description;
      await db.SaveChangesAsync();
      return Results.NoContent();
});
```

Tester PUT

1. Sélectionnez **PUT /pizza/{id}** dans l'interface utilisateur Swagger.
2. Sélectionnez **Essayer**.
3. Dans la zone de texte **id**, entrez **1**.
4. Enfin, mettez à jour `Request body`. Collez le code JSON suivant et remplacez `name` par `Ananas`.

   JSONCopier

   ```
   {
      "id": 1,
      "name": "Ananas"
   }
   ```

5. Sélectionnez **Exécuter**.

Pour tester le code, faites défiler vers `GET /pizza/{id}`. La pizza a maintenant le nom `Ananas`.

Supprimer un élément

Pour supprimer un élément existant, ajoutez ce code sous `PUT /pizza/{id}` que vous avez créé précédemment :

C#Copier

```
app.MapDelete("/pizza/{id}", async (PizzaDb db, int id) =>
{
   var pizza = await db.Pizzas.FindAsync(id);
   if (pizza is null)
   {
      return Results.NotFound();
   }
   db.Pizzas.Remove(pizza);
   await db.SaveChangesAsync();
   return Results.Ok();
});
```

Université Cheikh Anta Diop
École Supérieure Polytechnique (ESP)
Département Génie Informatique
M2GLSI SOIR
2ème année

Année Universitaire 2023/2024
Enseignement .NET
Travaux Pratiques ADO .NET
Chargé de Cours E. H. Ousmane Diallo
22/11/2024

Tests de DELETE

À présent, essayez de supprimer un élément à l'aide de l'interface Swagger.

Dans cette leçon, vous avez ajouté EF Core à une application API minimale existante et utilisé une base de données en mémoire pour stocker les données. Vous allez ensuite apprendre à utiliser une base de données réelle pour stocker les données afin qu'elles soient conservées entre les arrêts de l'application.

Utiliser le fournisseur de base de données SQLite avec EF Core

Effectué 100 XP

- 4 minutes

Dans l'unité précédente, vous avez appris à rendre les données persistantes dans une base de données en mémoire. La persistance des données dans une base de données en mémoire est utile lors du développement. Toutefois, étant donné que toutes les données sont perdues lors du redémarrage de l'application, ceci n'est pas tout à fait adapté à la production. En production, vous devez conserver les données dans une base de données comme SQL Server, MySQL, PostgreSQL ou SQLite.

Fournisseurs de base de données - Accès abstrait à la base de données à partir du code d'application

L'un des avantages de l'accès aux bases de données par le biais d'une couche d'abstraction comme Entity Framework (EF) Core est que cela dissocie votre application du fournisseur de base de données. Vous pouvez modifier le fournisseur de base de données sans réécrire votre code d'accès à la base de données. Ne vous attendez pas à pouvoir basculer entre fournisseurs de base de données sans effet sur le code de votre application, mais les modifications sont réduites et localisées.

L'un des avantages liés à l'utilisation d'EF Core est que vous pouvez réutiliser votre code, votre expérience et vos bibliothèques d'accès aux données pour travailler avec n'importe quel autre fournisseur de base de données EF Core.

Pour ce tutoriel, vous utiliserez une base de données SQLite, mais vous pouvez également en utiliser une qui vous convient mieux. EF Core prend actuellement en charge plus de 20 fournisseurs de bases de données. Les fournisseurs sont répertoriés dans la documentation.

Procédure d'ajout d'un nouveau fournisseur de base de données

En général, vous utiliserez les étapes suivantes pour implémenter un nouveau fournisseur de base de données :

1. Ajoutez un ou plusieurs packages NuGet à votre projet pour inclure le fournisseur de base de données.
2. Configurez la connexion à la base de données.
3. Configurez le fournisseur de base de données dans les services ASP.NET Core.
4. Effectuer des migrations de base de données.

Dans la leçon suivante, vous allez parcourir les étapes pour ajouter le fournisseur de base de données SQLite. Des étapes similaires s'appliquent aux autres fournisseurs de bases de données.

Unité suivante: Exercice : Utiliser le fournisseur de base de données SQLite avec EF Core

Exercice : Utiliser le fournisseur de base de données SQLite avec EF Core

Effectué 100 XP

- 10 minutes

Jusqu'à ce stade, vous avez enregistré vos données dans une base de données en mémoire. Cette base de données est facile à configurer et à utiliser pendant le développement de votre application, mais les données ne sont pas persistantes. Par conséquent, les données seront perdues lors du redémarrage de l'application. Avant de déployer votre application, vous devez rendre les données persistantes dans une base de données.

Dans cet exercice, vous mettrez à niveau votre application pour utiliser une base de données relationnelle pour stocker vos données. Vous utiliserez SQLite pour stocker vos données.

Configurer la base de données SQLite

Complétez les sections suivantes pour configurer la base de données SQLite.

Installez les outils et packages suivants

Dans le terminal, installez les packages suivants :

1. Fournisseur de base de données SQLite pour EF Core : Peut accéder à de nombreuses bases de données différentes par le biais de bibliothèques complémentaires appelées fournisseurs de base de données. Le package suivant est le fournisseur de base de données SQLite pour Entity Framework (EF) Core.

   CLI .NETCopier

   ```
   dotnet add package Microsoft.EntityFrameworkCore.Sqlite --version 6.0
   ```

2. Outils EF Core : des outils pour permettre à EF Core d'effectuer des tâches de développement au moment du design. Par exemple, ils créent des migrations, appliquent des migrations et génèrent du code pour un modèle basé sur une base de données existante.

   CLI .NETCopier

   ```
   dotnet tool install --global dotnet-ef
   ```

3. Microsoft.EntityFrameworkCore.Design : contient toute la logique au moment du design pour EF Core afin de créer votre base de données.

   CLI .NETCopier

   ```
   dotnet add package Microsoft.EntityFrameworkCore.Design --version 6.0
   ```

Activer la création de bases de données

Pour activer la création de bases de données, vous devez définir la chaîne de connexion de base de données. Migrez alors votre modèle de données vers une base de données SQLite.

1. Dans *Program.cs*, sous `var builder = WebApplication.CreateBuilder(args);`, ajoutez une chaîne de connexion.

   C#Copier

   ```
   var connectionString = builder.Configuration.GetConnectionString("Pizzas") ?? "Data Source=Pizzas.db";
   ```

   Ce code vérifie le fournisseur de configuration pour une chaîne de connexion nommée *Pizzas*. S'il n'en trouve pas, il utilisera `Data Source=Pizzas.db` comme chaîne de connexion. SQLite la mappe dans un fichier.

2. Dans la partie CRUD de ce didacticiel, vous avez utilisé une base de données en mémoire. Vous allez maintenant remplacer la base de données en mémoire par une base de données persistante.

   Remplacez votre implémentation de base de données en mémoire actuelle `builder.Services.AddDbContext<PizzaDb>(options => options.UseInMemoryDatabase("items"));` dans vos services de génération par la version SQLite ici :

   C#Copier

   ```
   builder.Services.AddSqlite<PizzaDb>(connectionString);
   ```

3. Avec l'outil de migration EF Core, vous pouvez maintenant générer votre première migration, `InitialCreate`. Enregistrez vos modifications et exécutez la commande suivante dans le terminal :

   ConsoleCopier

   ```
   dotnet ef migrations add InitialCreate
   ```

   EF Core crée un dossier nommé *Migrations* dans votre répertoire de projet qui contient deux fichiers avec le code qui représente les migrations de base de données.

4. Maintenant que vous avez terminé la migration, vous pouvez l'utiliser pour créer votre base de données et votre schéma.

   Dans une fenêtre de terminal, exécutez la commande `database update` suivante pour appliquer des migrations à une base de données :

   ConsoleCopier

   ```
   dotnet ef database update
   ```

   Vous devriez voir un fichier *Pizzas.db* nouvellement créé dans le répertoire de votre projet.

Exécuter et tester l'application

Maintenant que vous disposez d'une base de données de support, vos modifications sont conservées.

Testez votre application comme avant l'utilisation de `dotnet run` et l'interface utilisateur Swagger. Arrêtez l'application à l'aide de la commande **Ctrl+C**. Ensuite, réexécutez-la et vérifiez que vos modifications sont conservées dans *Pizzas.db*.

Université Cheikh Anta Diop **Année Universitaire 2023/2024**

École Supérieure Polytechnique (ESP) **Enseignement .NET**

Département Génie Informatique **Travaux Pratiques ADO .NET**

M2GLSI SOIR **Chargé de Cours E. H. Ousmane Diallo**

2ème année **22/11/2024**

Félicitations ! Vous avez câblé une base de données à votre API minimale !