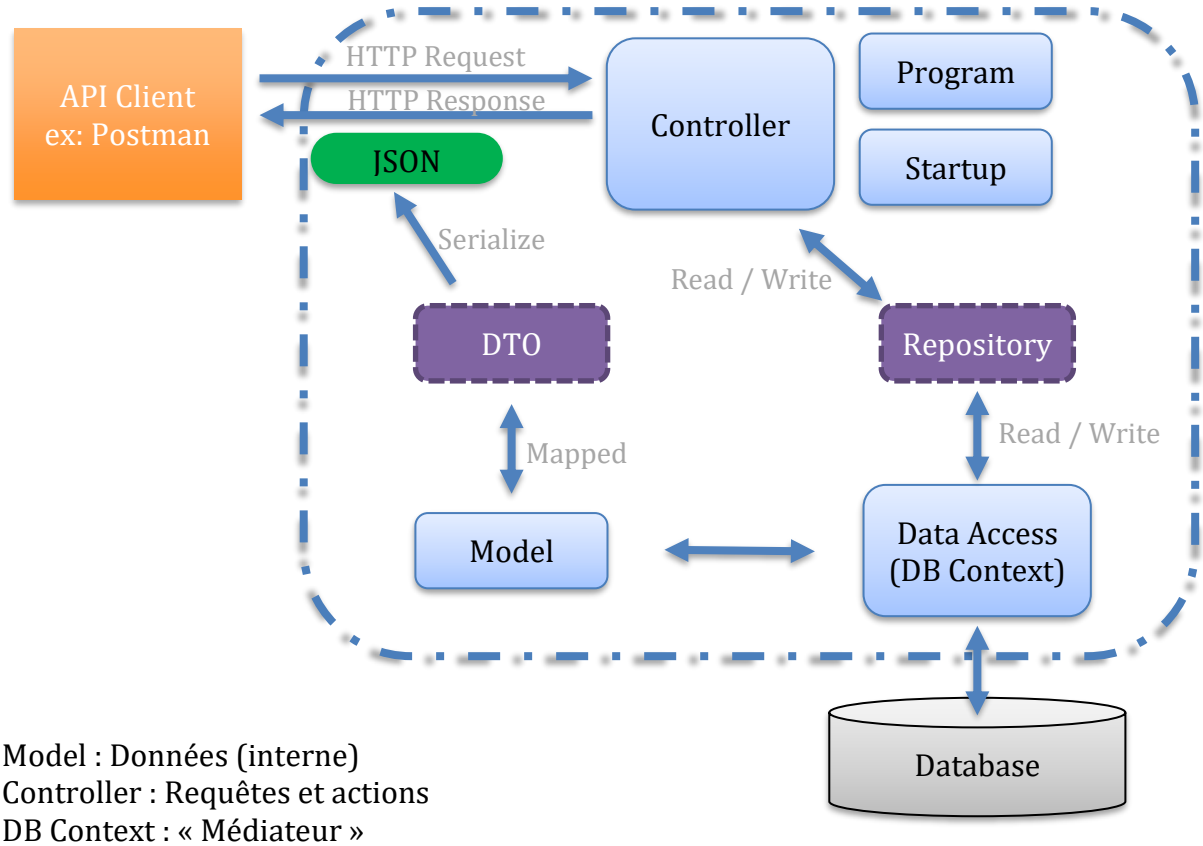
API .NET Core avec Entity Framework

I) Présentation

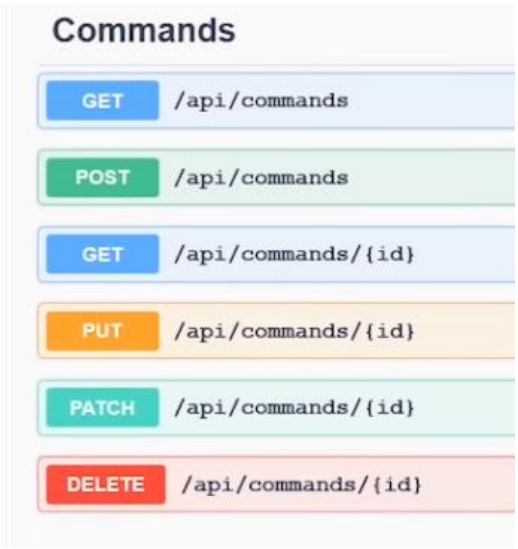
Nous allons créer une API qui permet de faire les opérations classiques sur une table.

Voici l'architecture visée



Model : Données (interne)
Controller : Requêtes et actions
DB Context : « Médiateur »
Repository : Persistance Ignorance
Data Transfert Object : Données externe

Et les commandes qui vont être mise en place



II) Quelques notions de base

II.1) Qu'est-ce qu'un DTO : Data Transfert Object

Un DTO est une classe légère qui ne contient que des propriétés, et nous pouvons obtenir et définir ces propriétés de classe.

DTO ne contient aucun comportement ni logique personnalisée.

II.2) Utilisation de DTO

DTO sert à transférer des données entre les couches.

II.3) Pourquoi faire un objet aussi simple ?

Pour créer un conteneur de type uniquement pour collecter des données sans aucune logique personnalisée et en raison de son poids léger, il peut facilement être transféré entre les couches.

II.4) Qu'est-ce qu'un POCO ?

POCO signifie Plain Old CLR Object, POCO est un objet métier qui contient des données, une validation et une logique personnalisée ou métier, mais il ne contient pas de logique de persistance, ce qui signifie une logique liée à la base de données, donc en raison de ce POCO sont ignorant persistant.

Cela signifie que la classe POCO ne contient que des propriétés, une validation et une logique métier, mais ne contient aucune logique de données

II.5) Ignorance de la persistance

L'ignorance de la persistance signifie ignorer la logique de la persistance ; cela signifie une logique liée à la base de données,

Dans Entity Framework, la classe POCO ne contient pas de logique liée à la base de données, comme l'enregistrement de données la base de données ou la récupération de données à partir de cette base.

II.6) En résumé

Lorsque vous travaillez sur des couches métier avec des objets métier, la classe POCO considérera qui contient les propriétés et la logique métier mais sans aucune logique de persistance, et la classe POCO ne dépend pas de la structure de la base de données. Et DTO n'est qu'une classe légère qui contient uniquement des propriétés et est utilisée pour transférer des données entre les couches ou entre les applications.

III) Mise en place de la base de données

Dans une base de données de votre choix, créer la table personnes à l'aide de ce script.

Vérifier que la table est bien créée et qu'elle contient des données.

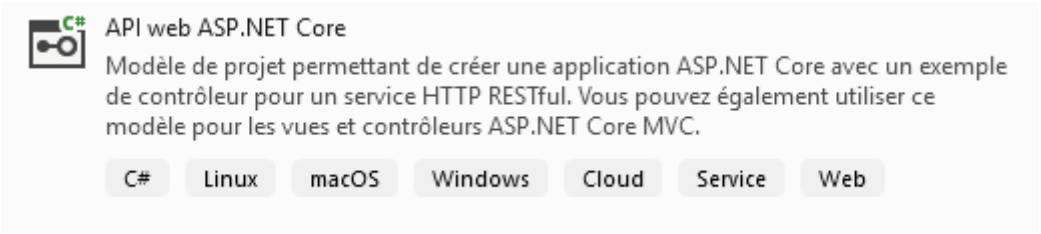
IV) Créer un nouveau projet C#

IV.1) La solution

Créer un dossier pour votre projet puis clic-droit Ouvrir avec Visual Studio

Fichier, Nouveau, Projet

Sélectionner le modèle



Donner un nom au projet, garder la version courante du Framework et cliquer sur créer

Supprimer les 2 fichiers qui font référence à WeatherForecast

Modifier le fichier launchSettings.json, mettre launchURL à api/Personnes

IV.2) Les dépendances nuget

Ajouter les dépendances au projet en cliquant sur Outils, Gestionnaire de package NuGet, Gérer les packages nugets pour la solution

```
MySql.EntityFrameworkCore
```

```
Microsoft.EntityFrameworkCore.Tools
```

```
Microsoft.EntityFrameworkCore
```

Vous pouvez voir les packages inclus, en double cliquant sur le nom du projet (fichier nomProjet.csproj)

IV.3) Le contexte

La classe contexte permet de créer la relation (session) avec la base de données. Elle étend la classe System.Data.Entity.DbContext.DbContext. La classe de contexte est utilisée pour interroger ou enregistrer des données dans la base de données. Il est également utilisé pour configurer les classes de domaine, les mappages liés à la base de données, les paramètres de suivi des modifications, la mise en cache, les transactions, etc.

- Créer un dossier Models dans le projet

- Ajouter le service DbContext au projet
    - Nous éviterons de mettre notre chaine de connexion directement dans la méthode de configuration de service. Nous allons créer une entrée dans le fichier json de configuration
    - L'entrée doit porter le nom `"ConnectionStrings"` obligatoirement avec un s. On peut configurer plusieurs bases de données pour différentes utilisations
    - Dans le fichier appsettings.json, ajouter l'entrée suivante.

```
"ConnectionStrings": {
    "Default": "server=localhost;user=root;database=EFAPI;port=3306;ssl mode=none" },
```

    - Dans le fichier Program.cs, nous allons ajouter le service DbContext

```
builder.Services.AddDbContext<APIContext>(options =>
options.UseMySQL(builder.Configuration.GetConnectionString("Default")));
```

IV.4) Utiliser EntityFramework Scaffold

Import des tables

- Lancer la commande suivante dans le package manager

```
scaffold-DbContext -Connection name=default -Provider MySql.EntityFrameworkCore -
OutputDir Models/Data -Context APIContext -ContextDir Models
```

- Les entités correspondantes aux tables de la base de données ont été créées.

V) Créer le service pour la persistance

- Créer un dossier Services dans le dossier Models
- Créer une classe PersonnesServices
- Créer un attribut privé qui contient le contexte amené par le constructeur

```
public class PersonnesServices
{
    private readonly MyDbContext _context ;
    public PersonnesServices(APIContext context)
    {
        _context = context;

    }
```

- Créer les méthodes de CRUD

```
        public void AddPersonnes(Personne p)
        {
            if (p == null)
            {
                throw new ArgumentNullException(nameof(p));
            }
            _context.Personnes.Add(p);
            _context.SaveChanges();
        }

        public void DeletePersonne(Personne p)
        {
            //si l'objet personne est null, on renvoi une exception
```

```
            if (p == null)
            {
                throw new ArgumentNullException(nameof(p));
            }
            // on met à jour le context
            _context.Personnes.Remove(p);
            _context.SaveChanges();
        }

        public IEnumerable<Personne> GetAllPersonnes()
        {
            return _context.Personnes.ToList();
        }

        public Personne GetPersonneById(int id)
        {
            return _context.Personnes.FirstOrDefault(p => p.IdPersonne == id);
        }

        public void UpdatePersonne(Personne p)
        {
            //nothing
            //on va mettre à jour le context dans le controller par mapping et passer
les modifs à la base

        }
```

- Ajouter le service au program

```
builder.Services.AddTransient<PersonnesService>();
```

VI) Créer le DTO

- Dans le dossier Models, créer un dossier Dtos
- Créer une classe PersonnesDTO

Elle contient les mêmes propriétés que Personne dans Data, sauf l'ID

```
namespace PersonnesAPI.Data.Dtos
{
    public class PersonnesDTO
    {
        public string Nom { get; set; }
        public string Prenom { get; set; }
    }
}
```

Cette classe servira à envoyer les données à l'extérieur sans mentionner l'id qui reste de la logique interne.

Dans des projets plus complexes, nous mettrons en place plusieurs niveaux de DTO, par exemple PersonneOutDTO et PersonneInDTO, pour séparer les attributs nécessaires en entrée de ceux utile en sortie.

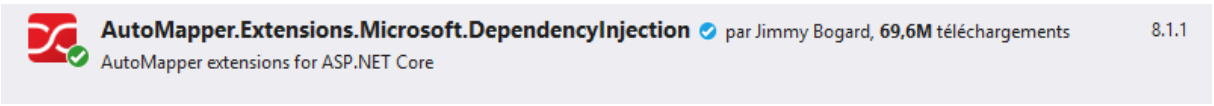
Les DTOs peuvent être suffixés VM (View Model), un fichier peut contenir plusieurs classes.

Lorsque nous renverrons des données, nous renverrons un type PersonnesDto plutôt qu'un type Personne.

Pour faciliter le passage d'un objet POCO à un objet DTO, nous utiliserons le mapping.

VII) Créer le mapping avec automapper

- Installer le NuGet AutoMapper



- Créer un dossier Profiles dans le dossier Models
- Créer une classe PersonnesProfile, elle étend Profile
- Créer le constructeur avec un mapping dans chaque sens

```
    public class PersonnesProfile : Profile
    {
        public PersonnesProfile()
        {
            CreateMap<Personne, PersonnesDTO>();
            CreateMap<PersonnesDTO, Personne>();
        }

    }
```

- Ajouter le service au program (un seul pour tous les mapping de toutes les classes)

```
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
```

VIII) Créer le controller

VIII.1) La structure

- Dans le dossier Controller, créer une classe PersonnesController qui étend ControllerBase
- Ajouter un attribut privé qui contient le service amené par le constructeur
- Ajouter un attribut privé qui contient le mapper amené par le constructeur
- Déclarer la classe comme répondant à l'APIController

```
namespace PersonnesAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class PersonnesController:ControllerBase
    {
        private readonly PersonnesService _service;
        private readonly IMapper _mapper;

        public PersonnesController(PersonnesService service, IMapper mapper)
        {
            _service = service;
            _mapper = mapper;
        }
    }

}
```

[Route("api/[controller]")] : définit la route qui mènera à cette classe. [controller] signifie le nom de la classe sans le suffixe Controller ici Personnes. Ainsi l'appel à cette classe se fera via l'url GET api/personnes

VIII.2) Les différents appels

➡ La méthode GET sans paramètre correspond à GetAll

```
        //GET api/personnes
        [HttpGet]
        public ActionResult<IEnumerable<PersonnesDTO>> getAllPersonnes()
        {
            var listePersonnes = _service.GetAllPersonnes();
            return Ok(_mapper.Map<IEnumerable<PersonnesDTO>>(listePersonnes));

        }
```

[HttpGet] : Précise le chemin qui amènera à cette méthode. Ici un get sur la classe c'est-à-dire GET api/personnes

Cette méthode renvoi toutes les occurrences de la base de données.

Testons cet exemple :

- Lancer le serveur IIS Express (dans le lacement de programme habituelle, sélectionner IIS Express)
- Autoriser l'affichage de la page

➡ La méthode GET avec un paramètre correspond à FindById

```
        //GET api/personnes/{id}
        [HttpGet("{id}", Name = "GetPersonneById")]
        public ActionResult<PersonnesDTO> GetPersonneById(int id)
        {
```

```
            var commandItem = _service.GetPersonneById(id);
            if (commandItem != null)
            {
                return Ok(_mapper.Map<PersonnesDTO>(commandItem));
            }
            return NotFound();

        }
```

[HttpGet("{id}", Name = "GetPersonneById")] : Précise le chemin qui amènera à cette méthode avec l'argument id.

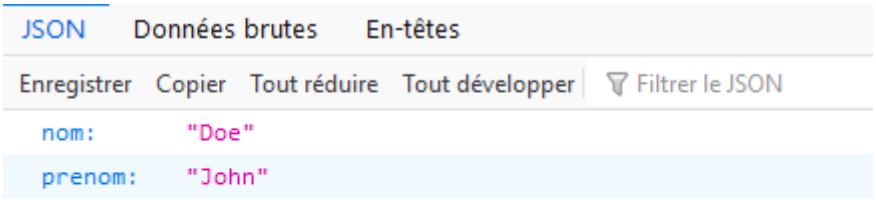
Ici un get sur la classe c'est-à-dire GET api/personnes/1

Le name permettra par la suite d'appeler cette méthode en interne

Cette méthode renvoi l'occurrence de la base de données correspond à l'id

Testons cet exemple :

- Lancer le serveur IIS Express
- Ajouter /1 à l'URL



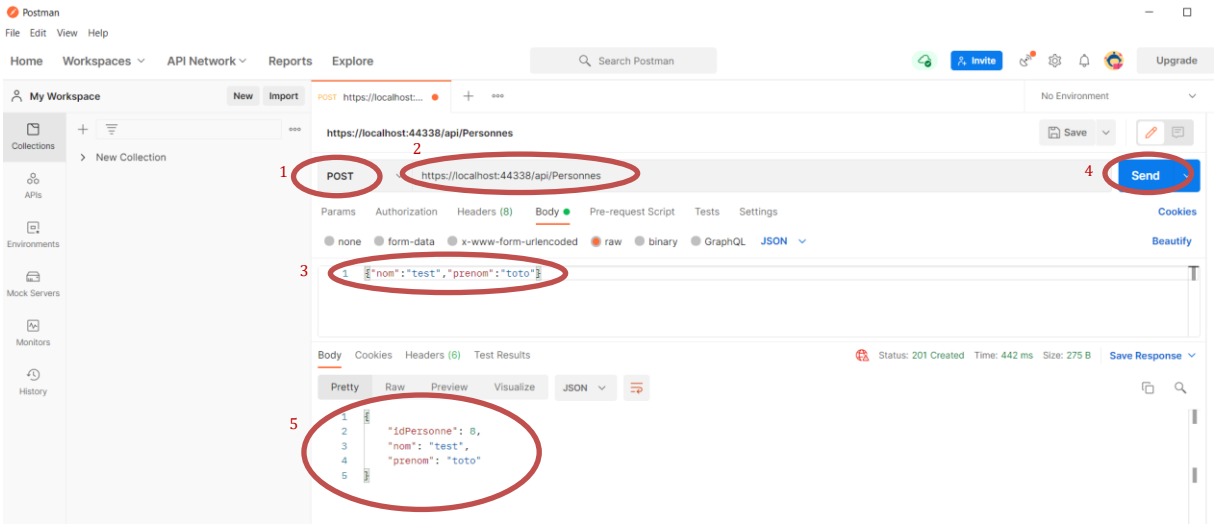
➡ La méthode POST avec un paramètre caché correspond à Add

```
//POST api/personnes
        [HttpPost]
        public ActionResult<PersonnesDTO> CreatePersonne(Personne personne)
        {
              //on ajoute l'objet à la base de données
            _service.AddPersonnes(personne);
            //on retourne le chemin de findById avec l'objet créé
            return CreatedAtRoute(nameof(GetPersonneById), new { Id =
personne.IdPersonne }, personne);
        }
```

Cette méthode insère l'objet passé en paramètre dans la base de données et renvoi vers la page qui présente ce nouvel enregistrement.

Testons cet exemple :

- Lancer le serveur IIS Express
- Nous ne pouvons plus tester directement dans le navigateur. Nous allons utiliser Postman (Lire la doc correspondante pour connaître les manipulations)



➡ La méthode PUT avec un id dans l'url et un paramètre caché correspond à Update

```
        //PUT api/personnes/{id}
        [HttpPut("{id}")]
        public ActionResult UpdatePersonne(int id, PersonnesDTO personne)
        {
            var personneFromRepo = _service.GetPersonneById(id);
            if (personneFromRepo == null)
            {
                return NotFound();
            }
            personneFromRepo.Dump();
            _mapper.Map(personne, personneFromRepo);
            personneFromRepo.Dump();
            personne.Dump();
            // inutile puisque la fonction ne fait rien, mais garde la cohérence
            _service.UpdatePersonne(personneFromRepo);

            return NoContent();

        }
```

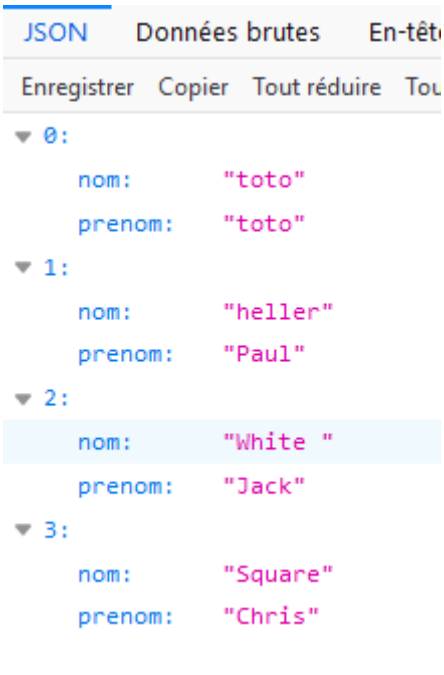
On précise l'id dans l'URL, on passe le nouvel objet en POST

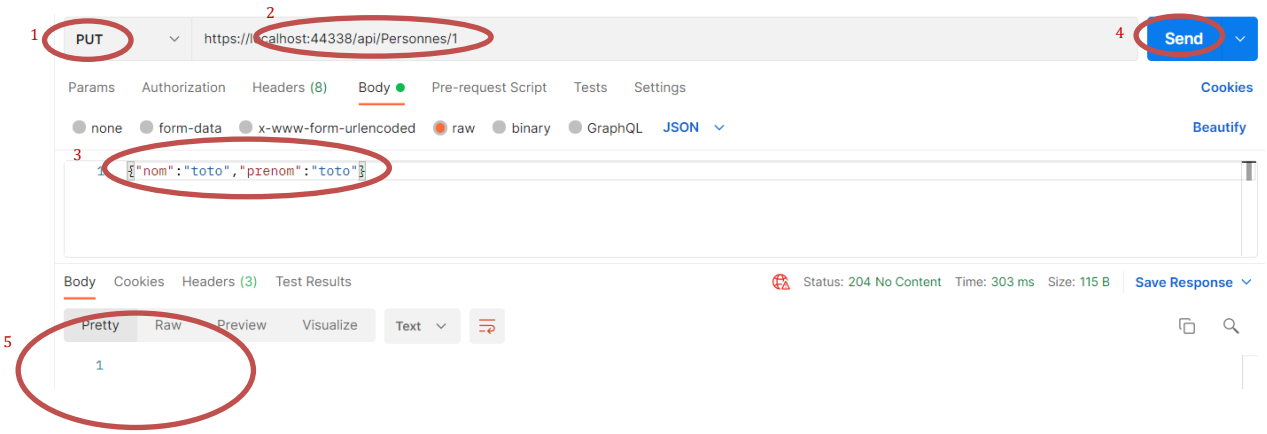
On récupère l'objet en base, s'il n'existe pas, on renvoi une erreur.

On propage les modifications grâce au mapper.

On persiste les données.

Avant : Après





➡ La méthode PATCH avec un id dans l'url et un paramètre caché correspond à Update partielle

Un update partiel signifie que l'on va indiquer au système quel champ modifier pour quel enregistrement.

```
Exemple d'appel
          [{  "op":"replace",
            "path":"prenom",
            "value":"test"
          }]
```

```
       //PATCH api/personnes/{id}
        [HttpPatch("{id}")]
        public ActionResult PartialPersonneUpdate(int id, JsonPatchDocument<Personne>
patchDoc)
        {
            var personneFromRepo = _service.GetPersonneById(id);
            if (personneFromRepo == null)
            {
                return NotFound();
            }
```

```
            var personneToPatch = _mapper.Map<Personne>(personneFromRepo);
            patchDoc.ApplyTo(personneToPatch, ModelState);

            if (!TryValidateModel(personneToPatch))
            {
                return ValidationProblem(ModelState);
            }

            _mapper.Map(personneToPatch, personneFromRepo);

            _service.UpdatePersonne(personneFromRepo);

            return NoContent();
        }
```

On précise l'id dans l'URL, on passe les instructions de modification en POST

On récupère l'objet en base, s'il n'existe pas, on renvoi une erreur.

On vérifie que les modifications sont cohérentes

On propage les modifications grâce au mapper.

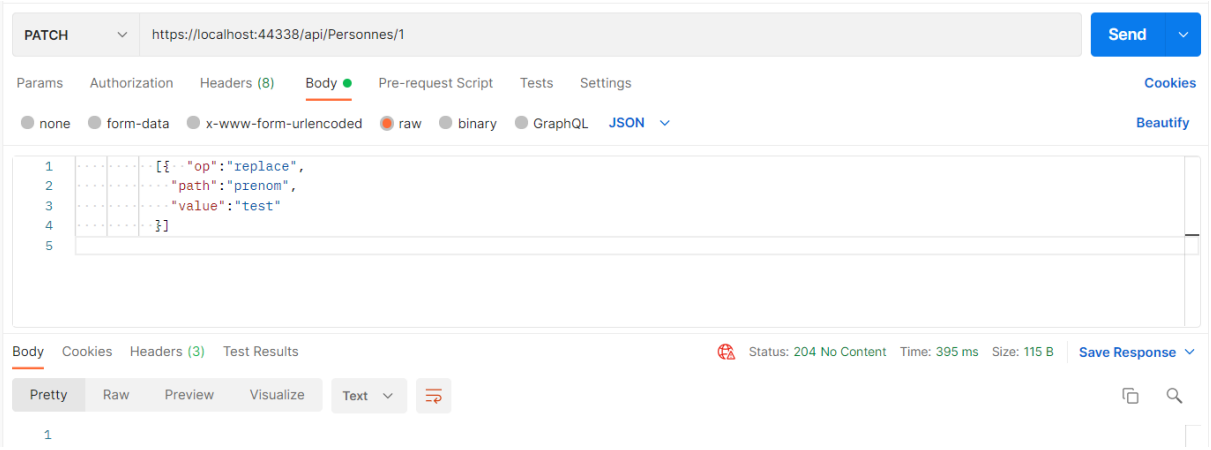
On persiste les données.

Il faut ajouter une indication dans le program.cs

```
builder.Services.AddControllers().AddNewtonsoftJson();
```

Avant :

Après



La liste des opérations est disponible dans la doc

Operations [edit]

The operations do the following:

Add: adds a value into an object or array.

Remove: removes a value from an object or array.

Replace: replaces a value. Logically identical to using remove and then add.

Copy: copies a value from one path to another by adding the value at a specified location to another location.

Move: moves a value from one place to another by removing from one location and adding to another.

Test: tests for equality at a certain path for a certain value.[3]

➡ La méthode DELETE avec un id dans l'url correspond à Delete

```
        //DELETE api/personnes/{id}
        [HttpDelete("{id}")]
        public ActionResult DeletePersonne(int id)
        {
            var personneModelFromRepo = _service.GetPersonneById(id);
            if (personneModelFromRepo == null)
            {
                return NotFound();
            }
            _service.DeletePersonne(personneModelFromRepo);

            return NoContent();

        }
```

Cette méthode permet de supprimer un enregistrement de la table