Tutoriel phase 3

But

- Faire une application Web API
- Gérer la persistance des données avec « Entity Framework Core »
- Gérer l'asynchronisme

Développement d'une API en ASP.NET Core Web API pour gérer une bibliothèque

Avant de commencer à faire ce projet il est conseillé d'installer plusieurs extensions de Visual Studio Code :

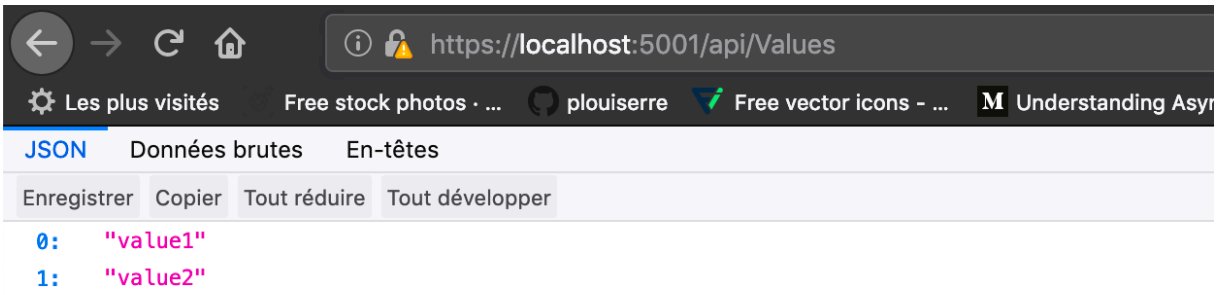
- C# for Visual Studio Code
- ASP.NET core VS Code Extension Pack

Tout d'abord comme pour le projet précédent il vous faut créer votre projet ASP.NET Core Web API. Pour cela il vous suffit de taper la commande **dotnet new webapi --name *nomDeVotreProjet*.**

Votre projet sera ainsi créer avec :

- Program.cs permet de créer l'host pour l'application web
- Startup.cs permet de créer le pipeline de traitement des requêtes de l'application et peut aussi configurer les services de l'application
- DevoxxBooksApi.csproj
- Un dossier Controllers avec un Controller « ValuesController.cs » créé par défaut.

Pour voir si votre projet fonctionne correctement il suffit de lancer l'application soit par visual studio code ou soit de taper dans le dossier de votre projet la commande **dotnet run.**
Ensuite à l'aide de votre navigateur aller à l'adresse https://localhost:5001/api/Values et vous devriez obtenir cet écran.



Votre API REST aura pour charge de gérer le contenu d'une bibliothèque en exposant 4 méthodes :

- Ajouter un livre dans votre bibliothèque
- Supprimer un livre dans votre bibliothèque
- Retrouver un livre particulier dans votre bibliothèque par son Id
- Retourner tous les livres de la bibliothèque

Pour cela vous devrez utiliser Entity Framework Core avec SqlLite. Entity Framework Core est un ORM en Dotnet Core permettant de gérer avec des objets des bases de données relationnelles. SqlLite est une base de données relationnelles mais ne fonctionne pas avec le système client-serveur. Il s'intègre directement aux programmes. Pour plus de détail n'hésitez pas à consulter ce lien https://docs.microsoft.com/fr-fr/ef/core/get-started/netcore/new-db-sqlite

Le modèle de donnée est une classe livre composée de 5 champs :

- Un Id
- Un Titre
- Un Type de livre
- Une année de sortie
- Un nom d'auteur

Ensuite pour pouvoir avoir une API rapide je vous demande d'utiliser l'asynchronisme dans vos API. Pour faire de l'asynchronisme il faut mettre le mot clé async dans la signature de la méthode. De plus quand elle renvoie en mode synchrone un objet X en étant elle renvoie Task<X>. Quand vous appelez une méthode asynchrone dans votre code il faut utiliser le mot clé await avant l'appel de cette méthode. Voici deux liens qui expliquent bien :

- https://docs.microsoft.com/fr-fr/dotnet/csharp/async
- https://docs.microsoft.com/fr-fr/dotnet/standard/async-in-depth

Pour initialiser le projet :

- Faire un git clone de ce projet github :
  - git clone https://github.com/plouiserre/DevoxxBooksApi.git
- L'initialisation du projet se trouve sur la branche master

Pour réaliser ce TP, vous aurez 45 minutes pour le réaliser, vous pouvez le tenter seul ou suivre la solution détaillée juste après.

Correction

Phase 1 : créer l'API sans base de donnée.

Récupérer directement la solution

Si vous voulez rapidement la solution il suffit de :

- Récupérer en local la branche « CrudWithoutDatabase », en utilisant ces commandes
  - git checkout CrudWithoutDatabase

Développement de cette partie

Tout d'abord créez le dossier Models à la racine du projet, puis la classe BookModel composée sous cette forme

```
using System;

namespace DevoxxBooksApi.Models
{
    public class BookModel
    {
        public int BookId { get; set; }

        public string BookTitle { get; set; }

        public string BookType { get; set; }

        public int BookPublication { get; set; }

        public string AutorName { get; set; }
    }
}
```

Warning : retirer l'héritage de DBContext de cette classe si vous voulez passer par cette étape intermédiaire

Une liste static va être utilisée « BookDataContext.cs » à la racine de ce projet pour gérer les données.

```
using System.Collections.Generic;
using DevoxxBooksApi.Models;

namespace DevoxxBooksApi
{
    public static class BookDataContext
    {
        public static List<BookModel> Books = new List<BookModel>();

    }
}
```

Dans le dossier Controllers, créez la classe BooksController. Elle hérite de la classe Controller. Elle possède deux attributs :

- [ApiController] : définit que c'est bien un controller de web api
- [Route("api/[controller]/[action]")] définit la route pour accéder à cette ressource

Ensuite il faut définir les méthodes pour votre API dans ce controller :

- L'attribut [HttpGet] est à mettre sur les méthodes pour retourner le livre par id ou tous les livres de votre bibliothèque.
- L'attribut [HttpPost] est à mettre quand un livre est ajouté à la bibliothèque pour faire une requête POST.
- L'attribut [HttpDelete] indique que la méthode supprime un livre précis de votre bibliothèque.
- Pour rappel la liste Static **Books** de la classe **BookDataContext** est utilisée comme moyen pour stocker les données ici en mémoire.

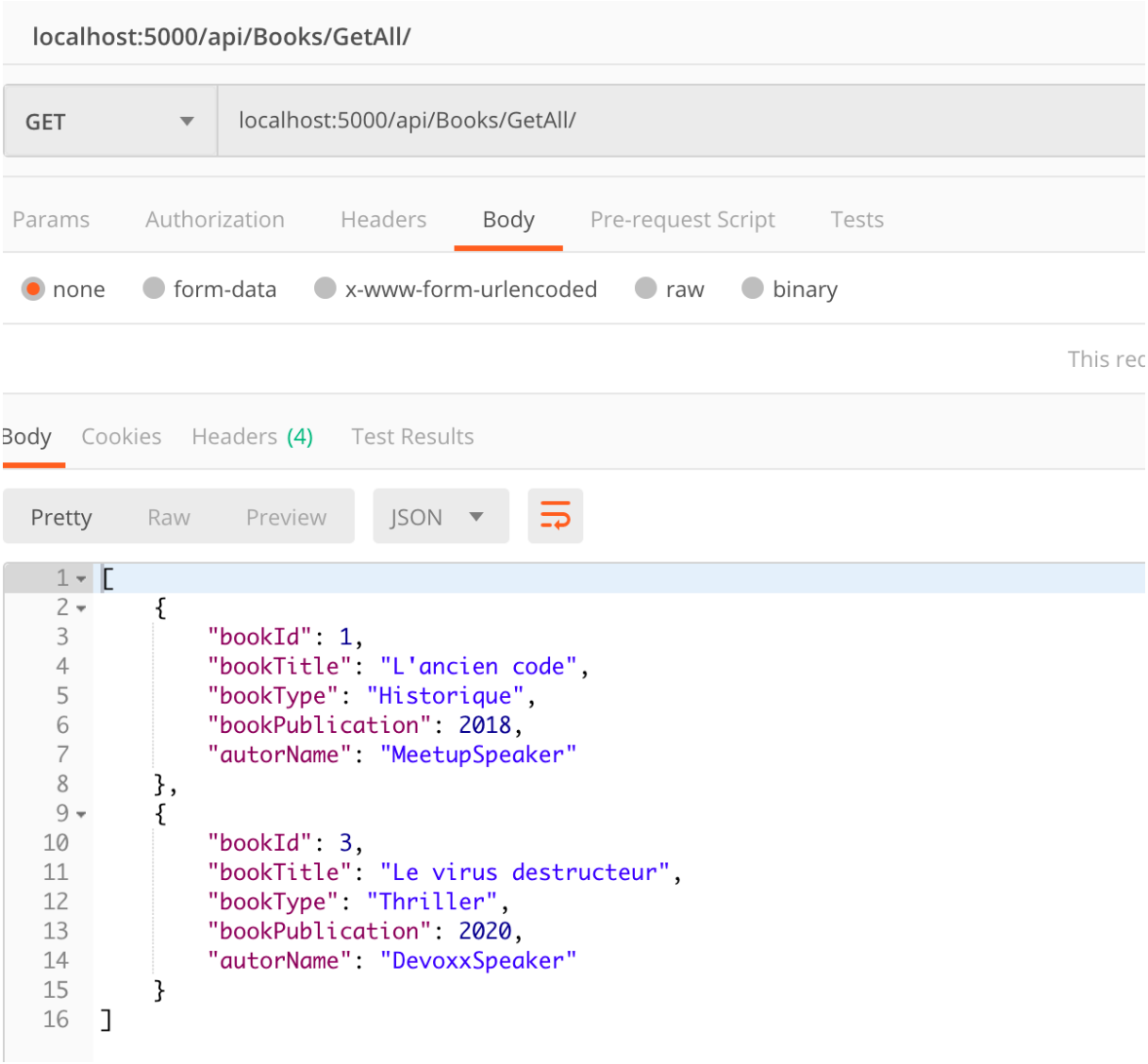
Voici le rendu de la classe controller

```
using System;
using System.Collections.Generic;
using System.Linq;
using DevoxxBooksApi.Models;
using Microsoft.AspNetCore.Mvc;

namespace DevoxxBooksApi.Controllers
{
    [Route("api/[controller]/[action]")]
    [ApiController]
    public class BooksController : Controller
    {


        [HttpGet]
        public IEnumerable<BookModel> GetAll()
        {
            return BookDataContext.Books;
        }

        [HttpGet]
        public  BookModel Get(int bookId)
        {
            return BookDataContext.Books.FirstOrDefault(o => o.BookId == bookId);
        }

        [HttpPost]
        public string Create(BookModel book)
        {
            try{
                BookDataContext.Books.Add(book);
                return "OK";
            }
            catch(Exception ex){
                return string.Format("erreur {0}", ex.Message);
            }
        }

        [HttpDelete]
        public string Delete(int bookId)
        {
            try{
                BookDataContext.Books.RemoveAll(o => o.BookId == bookId);
                return "OK";
            }
            catch(Exception ex){
                return string.Format("erreur {0}", ex.Message);
            }
        }
    }
}
```

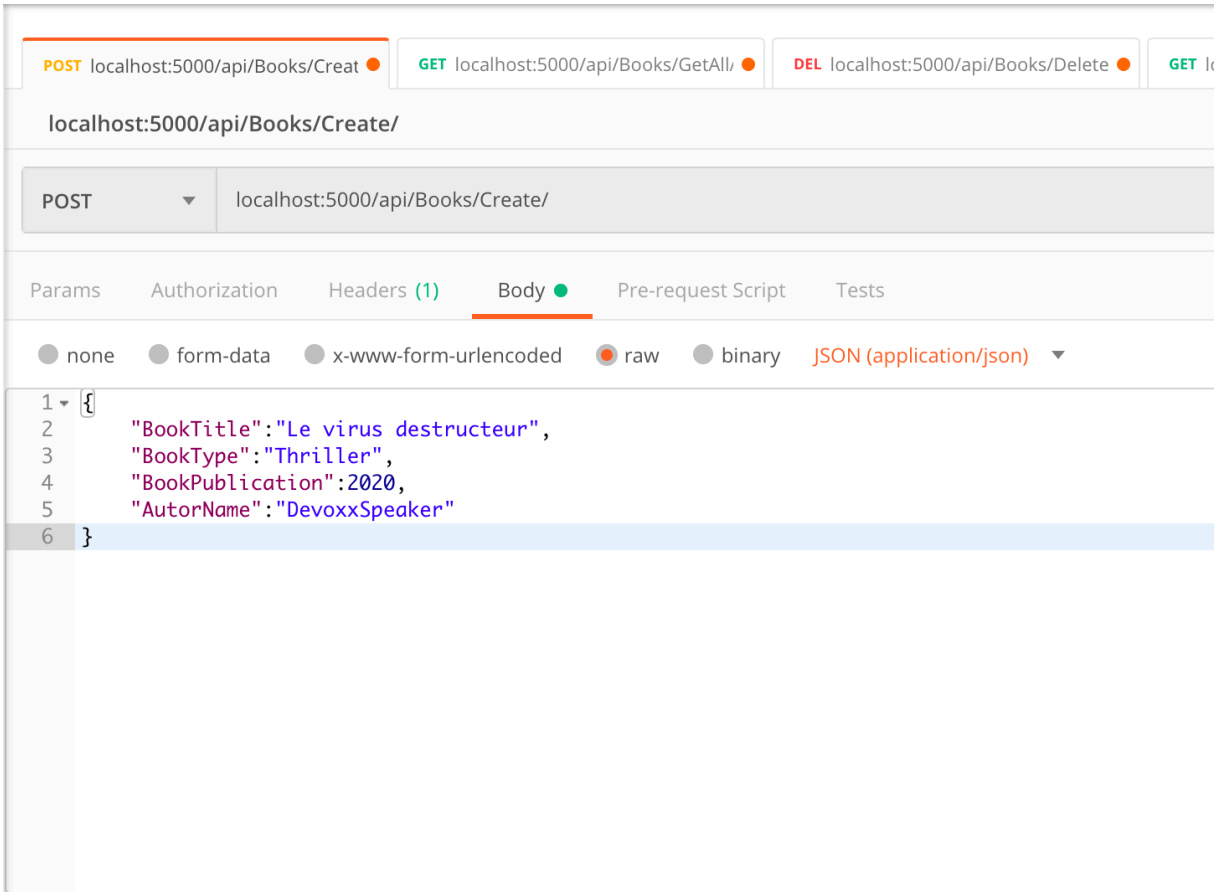
Pour tester il faut installer Postman qui permet de contacter directement notre API en REST. Voici un lien pour installer le logiciel https://www.getpostman.com/

Dans le dossier Properties se trouve le fichier lauchsettings.json. Dans l'objet json « DevoxxBooksApi » mettait pour la propriété « applicationUrl » la valeur « http://localhost:5000 ».

Ensuite pour utiliser postman voici deux exemples. Le premier fait un get



Le deuxième un post :



Phase 2 : Ajouter une base de données

Récupérer directement la solution

Si vous voulez rapidement la solution il suffit de :

- Récupérer en local la branche « crudWithDatabase », en utilisant ces commandes
  - git checkout crudWithDatabase
- Vous avez le code gérant cette partie

Développement de cette partie

Avant de toucher au code, il faut importer le package nuget Microsoft.EntityFrameworkCore.Sqlite. Pour cela, la commande **dotnet add package Microsoft.EntityFrameworkCore.Sqlite** doit être utilisée.

Pour que le package qui vient juste d'être installé soit compatible avec la version Asp.net core du projet, allez dans le csproj et vérifiez que la version de ce dernier soit bien dans la version 2.1.0. comme ci-dessous :

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.1.2" PrivateAssets="All" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="2.1.0" />
  </ItemGroup>

</Project>
```

Dans votre classe BookModel il faut ajouter l'attribut [Key]. Il indique à Entity Framework Code que cette propriété est une primary Key.

```
9 references
public class BookModel
{
    [Key]
    2 references
    public int BookId { get; set; }
```

Après la classe BookDataContext est modifiée pour :

- Hériter de DbContext
- Transformer Books en DbSet de BookModel
- Overrider la méthode OnConfiguring comme ci-dessous

```
using System.Collections.Generic;
using DevoxxBooksApi.Models;
using Microsoft.EntityFrameworkCore;

namespace DevoxxBooksApi
{
    public class BookDataContext : DbContext
    {
        public DbSet<BookModel> Books {get; set;}

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite("Filename=./devoxxbooksapi.sqlite");
        }
    }
}
```

Dans la classe Startup, le code suivant doit être ajouter pour demander à l'application qu'à chaque démarrage on supprime la base de données créé au précédent démarrage pour en créer une nouvelle. Bien sûr, il ne faut pas faire ça en production

```
using (var context = new BookDataContext())
{
    context.Database.EnsureDeleted();
    context.Database.EnsureCreated();
}
```

Maintenant il suffit de modifier les méthodes http de BookController pour aller utiliser entityFramework.Core pour la gestion des données.

```
[HttpGet]
public IEnumerable<BookModel> GetAll()
{
     List<BookModel> books = new List<BookModel>();
    using (var context = new BookDataContext()){
        foreach(var book in context.Books){
            books.Add(book);
        }
    }
    return books;
}

[HttpGet]
public  BookModel Get(int bookId)
{
    BookModel book = new BookModel();
    using (var context = new BookDataContext()){
        book = context.Books.FirstOrDefault(o => o.BookId == bookId);
    }
    return book;
}
```

```
[HttpPost]
public string Create(BookModel book)
{
    try{

        using(var context = new BookDataContext()){
           context.Books.Add(book);
           context.SaveChanges();

        }
        return "OK";
    }
    catch(Exception ex){
        return string.Format("erreur {0}", ex.Message);
    }
}

[HttpDelete]
public string Delete(int bookId)
{
    try{
        using(var context = new BookDataContext()){
            BookModel book = context.Books.FirstOrDefault(o => o.BookId == bookId);
            context.Books.Remove(book);
            context.SaveChanges();
        }
        return "OK";
    }
    catch(Exception ex){
        return string.Format("erreur {0}", ex.Message);
    }
}
```

Phase 3 : Asynchronisme

Récupérer directement la solution

Si vous voulez rapidement la solution il suffit de :

- Récupérer en local la branche « asynchroneApi », en utilisant ces commandes
  - git checkout asynchroneApi
  -

Développement de la solution

Tout d'abord toutes les méthodes devront comporter le mot **async.** Il indique au compilateur que cette méthode est asynchrone. Elle ne retourne pas uniquement un objet mais un Task de cet objet. Par exemple une méthode synchrone retournant un string retournera un Task<string>.
Pour que ces méthodes soient asynchrones, elles devront appeler des API asynchrones avec le mot clé await. En prenant les API pour ajouter un élément en base de donnée :

- context.Books.Add(book); -> await context.Books.AddAsync(book);
- context.SaveChanges(); -> await context.SaveChangesAsync() ;

Voilà ce que donne la classe BooksController en permettant l'asynchronisme :

```
[HttpGet]
public async Task<IEnumerable<BookModel>> GetAll()
{
    List<BookModel> books = new List<BookModel>();
    using (var context = new BookDataContext()){
        books = await context.Books.ToListAsync();
    }
    return books;
}

[HttpGet]
public async  Task<BookModel> Get(int bookId)
{
    BookModel book = new BookModel();
    using (var context = new BookDataContext()){
        book = await context.Books.FirstOrDefaultAsync(o => o.BookId == bookId);
    }
    return book;
}


[HttpPost]
public async Task<string> Create(BookModel book)
{
    try{

        using(var context = new BookDataContext()){
           await context.Books.AddAsync(book);
           await context.SaveChangesAsync();

        }
        return "OK";
    }
    catch(Exception ex){
        return string.Format("erreur {0}", ex.Message);
    }
}
```

```
[HttpDelete]
public async Task<string> Delete(int bookId)
{
    try{
        using(var context = new BookDataContext()){
            BookModel book = await context.Books.FirstOrDefaultAsync(o => o.BookId == bookId);
            context.Books.Remove(book);
            await context.SaveChangesAsync();
        }
        return "OK";
    }
    catch(Exception ex){
        return string.Format("erreur {0}", ex.Message);
    }
}
```