Développement web avec C#

Qu'est ce que le C#

C# ou csharp, est un langage développé par Microsoft pour concurrencer Java. Comme Java, c'est un langage semi-compilé et interprété par un environnement d'exécution. L'environnement d'exécution du C# est le .NET Runtime (prononcé dotnet), l'équivalent de la JVM de Java.

En Java, on compile les sources (fichiers `.java`) pour obtenir le bytecode (fichiers `.class`) que l'on package en exécutable (fichier `.jar`) pour les exécuter avec le Java Runtime Environnement. En C#, oncompile les sources (fichiers `.cs`) pour obtenir le code en Intermediate Language (fichiers `.il`) que l'on package en exécutable (fichiers `.dll` - attention les DLL .NET n'ont rien à voir avec les DLL natives).

Contrairement à certaines idées reçues, C# et .NET sont :

- Entièrement gratuits et open-source (licences MIT)
- Cross-platforme (comme Java, *write once, run anywhere*)

Mettre en place son environnement de développement

Installer .NET CLI

Avec Winget

```
winget install Microsoft.dotnet
```

Avec Chocolatey

```
choco install dotnet
```

Manuellement

Téléchargez et exécutez l'installer du **.NET SDK** via ce lien

Créer un projet web

Vous pouvez consulter les templates de projet en faisant :

```
dotnet new -l
```

Pour créer un nouveau projet d'application web :

```
dotnet new mvc -o "mon-projet"
```

Nous venons de créer une application web à page avec le framework ASP .NET Core MVC. Vous pouvez

Environnement de développement

Ce tutoial n'est pas dépendant d'un IDE particulier, il utilise la CLI .NET pour créer et exécuter les projets. Ainsi, vous pouvez utiliser n'importe quel IDE ou éditeur de texte. Je recommande les suivants :

- Jetbrains Rider : IDE Jetbrains dédié au développement .NET
- Visual Studio Code avec le plugin "C#" de Microsoft

Présentation de ASP .NET Core MVC

Le pattern MVC

Comme son nom l'indique, ce framework se base sur le pattern MVC, qui divise l'architecture de l'application en trois parties :

- Modèle : les données de l'application
- Vue : les pages web de l'application
- Controlleur : gestion des actions des utilisateurs, coordonne données et pages

Explication du layout du projet

- Dossier `Controllers` : Contient les classes controlleurs de votre application. Vous pouvez supprimer le fichier d'exemple créé par le projet
- Dossier `Models` : Contient les classes de logique et de données de votre application. Vous pouvez supprimer le fichier d'exemple créé par le projet
- Dossier `Views` : Contient les templates HTML du projet. Vous pouvez supprimer le dossier `Home` d'exemple créé par le projet
  - Dossier `Shared` : Contient les éléments de template ré-utilisables :
    - `_Layout.cshtml` : Squelette de toutes vos pages, l'appel `@RenderBody()` rend le contenu de la page. Vous pouvez vider les `<header>` et `<footer>` d'exemple créés par le projet, et les remplacer par votre propre `<header>` et `<footer>`. Tout ce que vous mettrez dans ce template sera rendu sur toutes les pages de votre application
    - `_Layout.css` : le CSS propre à votre squelette de pages
    - `_ValidationScriptsPartial.cshtml` et `Erros.cshtml` : vous pouvez les supprimer, ce sont des exemples du projet
  - `_ViewImports.cshtml` : Imports des namespaces dans les templates. Comme on a vider le namespace `Models`, son import ne compile plus, vous pouvez le commenter pour l'instant (la syntaxe pour les commenaire ici est `@* commenaire *@`
  - `_ViewStart.cshtml` est le point d'entrée de vos template
- Dossier `wwwroot` : les fichiers statiquement servis par votre application
  - `css` : le CSS de votre application
  - `js` : le JS de votre application
  - `lib` : les libraires JS et CSS téléchargées localement
  - `favicon.ico` : l'icone de votre site
- `Program.cs` : le point d'entré de votre application

Controllers & Routes

Pour créer un controlleur, créez une classe dont le nom finit par "Controller" dans le dossier `Controllers`. Par exemple, "HelloWorldController", qui étend la classe `Controller`, par exemple :

```
namespace net_web_tuto_2.Controllers
{
    public class HelloWorldController : Controller
    {
```

```
    }
}
```

Pour ajouter un préfixe de route à votre controleur, mettez lui l'attribut [Route("maroute")], par exemple :

```
    [Route("hello-world")]
    public class HelloWorldController : Controller
    {

    }
```

Ensuite, vous pouvez créer une méthode endpoint, avec l'annotation `[HttpGet]` :

```
        [HttpGet]
        public string HelloWorld()
        {
            return "Hello world !";
        }
```

La route de ce endpoint est dont `GET /hello-world`. On peut constater en lançant l'application avec un :

```
dotnet run
```

Et en en allant à https://localhost:7229/hello-world

On peut donner des suites de route particulières aux méthodes endpoint :

```
    [HttpGet("fr")]
    public string HelloWorldFr()
    {
        return "Bonjour, le monde";
    }

    [HttpGet("en")]
    public string HelloWorldEn()
    {
        return "Hello, world";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à :

- https://localhost:7229/hello-world/fr
- https://localhost:7229/hello-world/en

Paramètres de requête et endpoints

Paramètre de requête

Pour récupérer une paramètre de requête, il suffit de mettre un paramètre avec le même nom à la méthode endpoint :

```
    [HttpGet]
    public string HelloWorld(string name)
    {
        return $"Hello, {name}";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world?name=Arsène.

Paramètre de chemin

Pour récupérer un paramètre de chemin, il suffit de mettre ce paramètre entre accolades dans la route du endpoint, puis ajouter n paramètre avec le même nom à la méthode endpoint :

```
    [HttpGet("{id}")]
    public string HelloWorld(int id)
    {
        return $"Id : {id}";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world/1.

Vues

Retourner une vue

Avant de retourner une vue, il faut d'abord créer un dossier dans `Views` pour notre controleur qui correspond à son nom. Ici notre controleur s'appelle `HelloWorldController`, notre dossier va donc s'appeller `HelloWorld`.

On va ensuite créer dedans un fichier `hello-world.cshtml`, qui sera notre template :

```
<h1>Hello, world</h1>
```

Pour le retourner depuis une méthode endpoint, il faut utiliser la fonction `View()` du controleur :

```
    [HttpGet]
    public IActionResult HelloWorld()
    {
        return View("hello-world");
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world.

Templater une vue

Avec une classe modèle

On peut templater une vue avec un modèle de façon fortement typée, ce qui est pratique car cela permet de répérer d'eventuelles erreurs plus vites.

Soit la classe de modèle (dans le dossier `Model`, et n'oublier pas de décomenter l'import du namespace dans `_ViewImports.cshtml`) :

```
namespace net_web_tuto.Models {
    public class Person {
```

```
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

Dans la méthode endpoint, on ajouter le modèle dans l'appel à `View` :

```
    [HttpGet]
    public IActionResult HelloWorld()
    {
        var person = new Person(){ FirstName = "John", LastName = "Shepard"};
        return View("hello-world", person);
    }
```

Et pour le template, la syntaxe est la suivante :

```
<h1>Hello, @Model.FirstName @Model.LastName</h1>
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world.

Avec d'autres données

Pour templater la vue avec d'autres infos que celles d'un modèle, on peut utiliser le `ViewBag`, exemple :

Dans la méthode endpoint, on ajouter le modèle dans l'appel à `View` :

```
    [HttpGet]
    public IActionResult HelloWorld()
    {
        var person = new Person(){ FirstName = "John", LastName = "Shepard"};
      ViewBag.Today = DateTime.Now;
        return View("hello-world", person);
    }
```

```
<h1>Hello, @Model.FirstName @Model.LastName, The Date is : @ViewBag.Today</h1>
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world.

Rendu conditionnel

Pour faire du rendu conditionnel il faut utiliser `@if` :

```
@if((20 % 2) == 0) {
<p> 20 is even </p>
}

@if((21 % 2) == 0) {
<p> 21 is not even </p>
}
```

Seulement "20 is even" sera rendu sur la page.

Boucles

Pour faire du rendu en boucle, il faut utiliser `@foreach`, exemple :

```
@foreach (var person in Model.Persons)

    <div>@item.FirstName @item.LastName</div>
}
```

Soumission de formulaire

Pour gérer une soumission de formulaire, il faut un modèle dont les noms des champs correspondent aux champs du formulaire, puis ajouter ce modèle en paramètre la méthode enpoint. La méthode endpoint doit également utiliser la méhotde HTTP POST, et donc l'attribut `[HttpPost]`. Exemple :

Les deux templates, pour le formulaire et le résultat :

`hello-world-form.cshtml` :

```
<form action="/hello-world/salute" method="post">
    <input name="FirstName" type="text"/>
    <input name="lastName" type="text"/>
    <input type="submit" value="Say hello !"/>
</form>
```

`hello-world.cshtml` :

```
<h1>Hello, @Model.FirstName @Model.LastName</h1>
```

Les méthodes endpoint :

```
    [HttpGet]
    public IActionResult HelloWorldForm()
    {
        return View("hello-world-form");
    }

    [HttpPost("salute")]
    public IActionResult SayHello(Person person)
    {
        return View("hello-world", person);
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world.

Session HTTP

On peut stocker des données dans la session HTTP. Pour cela il faut d'abord la convigurer en rajoutant dans le `Program.cs` (après la ligne `var builder = WebApplication.CreateBuilder(args);`) :

```
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromSeconds(10);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});
```

Et après `app.UseAuthorization();` :

```
app.UseSession();
```

Pour enregistrer des données dans la session :

```
HttpContext.Session.SetString("key", "value");
```

Pour récupérer des données de la session :

```
string value = HttpContext.Session.GetString("key");
```

Interaction avec la base de donnée

L'interaction avec la base de donnée se fait via l'ORM (Object-Relationnal Mapper) officiel de Microsoft, Entity Framework Core.

Installer Entity Framework Core

Il faut installer les packages nugets via les commandes suivantes :

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 6.0.0
dotnet add package Pomelo.EntityFrameworkCore.MySql --version 6.0.0
```

Il faut également installer l'outil en ligne de commande d'EF Core pour gérer les migrations :

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

Le `DbContext`

Il faut ensuite créer notre classe `DbContext`, point d'entrée de notre base de donnée. Créez un dossier `Database` et dans ce dossier une classe `ApplicationDbContext` qui étend `DbContext`:

```
using Microsoft.EntityFrameworkCore;

namespace net_web_tuto.Database {

    public class ApplicationDbContext : DbContext {

        public ApplicationDbContext(DbContextOptions options) : base(options) {
        }
    }

}
```

Il faut ensuite rattacher notre `ApplicationDbContext` à notre application, dans votre `Program.cs` (après la ligne `var builder = WebApplication.CreateBuilder(args);`), rajoutez :

```
// Database
builder.Services.AddDbContext<ApplicationDbContext>(options => {
    string databaseHost = Environment.GetEnvironmentVariable("DATABASE_HOST");
    string databaseName = Environment.GetEnvironmentVariable("DATABASE_NAME");
    string databaseUsername = Environment.GetEnvironmentVariable("DATABASE_USERNAME");
    string databasePassword = Environment.GetEnvironmentVariable("DATABASE_PASSWORD");

    var connectionString =
$"server={databaseHost};database={databaseName};user={databaseUsername};password={databasePass
word}";
    options.UseMySql(connectionString, ServerVersion.AutoDetect(connectionString));
});
```

On récupère les identifiant depuis des variables d'environnement, c'est plus projet que des les avoir dans le code. Vous devriez aussi rajouter des `using` au début du fichier :

```
using Microsoft.EntityFrameworkCore;
using net_web_tuto.Database;
```

Définir les entités

Revenons à notre `ApplicationDbContext` auquel on va rajouter la méthode suivante pour définir nos entités :

```
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {

        }
```

Il faut aussi rajouter en attributs un `DbSet` qui va resprésenter notre table :

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

Ensuite dans la méthode `OnModelCreating`, on peut définir notre entité (on a ajouté un champs `Id` de type `Guid` à la classe `Person` pour servir d'identifiant dans la base :

```
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
```

```
□□□ modelBuilder.Entity<Person>().HasKey(person => person.Id);
            modelBuilder.Entity<Person>().Property(person => person.FirstName);
            modelBuilder.Entity<Person>().Property(person => person.LastName);
        }
```

On peut également définir des contraintes sur les attributs de l'entité, avec des méthodes comme `HasMaxLength`, `HasMinLength` chainées après l'appel à `Property`. Vous pouvez trouver plus de précisions sur [la documention d'EF Core]

Migrations

Après avoir défini notre modèle, il faut créer notre migration vers la base de donnée et l'exécuter. On va utiliser l'invite de commande EF Core pour ça.

Dans un premier temps on définit notre variables d'environnement (exemple avec Powershell, la syntaxe change selon votre shell):

```
$Env:DATABASE_HOST = "localhost"
$Env:DATABASE_NAME = "nomDeLaDb"
$Env:DATABASE_USERNAME = "userDeLaDb"
$Env:DATABASE_PASSWORD = "mdpDeLaDb"
```

> Avant de faire une migration, faite bien attention d'avoir éteint votre application, sinon vous allez avoir des err

Ensuite, exécutez la commande suivante pour créer la migrations :

```
dotnet ef migrations add maMigration
```

Enfin pour exécuter la migration sur votre base :

```
dotnet ef database update
```

Si vous vous connectez à votre base avec un client, vous deviez voir le schéma déployé.

Utiliser le `DbContext` dans le controleur

>

> Attention, si vous utiliser Rider, reportez les varibales d'environnement dans le fichier `Properties/launchSettings.json` dans l'objet `environmentVariables` de l'objet profile que vous utilisez pour lancer votre app (si vous avez un doute vous pouvez le mettre dans les deux).

Pour interagir avec la base de donnée depuis notre controleur, on peut demander au framework de nous l'injecter :

```
using Microsoft.AspNetCore.Mvc;
using net_web_tuto.Database;
using net_web_tuto.Models;

[Route("persons")]
public class PersonsController : Controller {

    private readonly ApplicationDbContext context;

    public PersonsController(ApplicationDbContext context){
        this.context = context;
    }

}
```

Créons maintenant un endpoint servant une vue avec :

- Une liste des personnes dans la base
- Un formulaire pour ajouter une personne

```
@foreach (Person person in @Model)
{
    <p>@person.FirstName - @person.LastName</p>
}

<form action="/persons" method="post">
    <input name="FirstName" type="text"/>
    <input name="LastName" type="text"/>
    <input value="Create" type="submit"/>
</form>
```

D'abord, pour récupérer les personnes, nous allons accèder à notre `DbSet` et le transformer en liste, pour EF Core, cela revient à faire une requête pour récuprer tous les enregistrement de la

table :

```
    [HttpGet]
    public IActionResult PersonsPage(){
        List<Person> persons = this.context.Persons.ToList();

        return View("persons", persons);
    }
```

Ensuite, on peut ajouter notre endpoint d'insertion :

```
    [HttpPost]
    public IActionResult AddPerson(Person p){
        this.context.Persons.Add(p);
        this.context.SaveChanges();

        return new RedirectResult("/persons");
    }
```

On ajoute la `Person` récupérée depuis le formulaire au `DbSet`, puis on appelle `SaveChanges` afin de valider la transaction sur la base.

Enfin, on retourne une redirection vers notre endpoint d'affichage pour réafficher la page avec les données mises à jours.

> Vous pouvez regarder dans la console les requêtes effectuées par EF Core sur la base de données, qui correspondent à nos appels

On va ensuite ajouter une vue pour voir une seule personne, avec un lien dans la liste. Créons d'abord notre endpoint :

```
    [HttpGet("{id}")]
    public IActionResult GetOnePerson(Guid id){
        Person person = this.context.Persons
            .First(p => p.Id == id);

        return View("person", person);
    }
```

On utilise un paramètre de route pour récupérer l'identifiant. Ensuite, la méthode `First` prend un prédicat (fonction retournant un booléen) sur les personnes et permet de filtrer la liste pour ne

retourner que le premier enregistrement qui valide le prédicat. EF Core compile cette méthode sous la forme d'une clause `WHERE` dans la requête SQL. L'appel à `First` permet ensuite de récupérer le premier élément du résultat (ici logiquement il n'y en a qu'un).

Pour faire un filtrage sur la table et récupérer une collection d'enregistrement, il faut utiliser la méthode `Where` qui prend également un prédicat, mais retourne tous les enregistrement qui le valident :

```
        List<Person> persons = this.context.Persons
            .Where(p => p.FirstName.Contains("e"))
            .ToList();
```

On retourne ensuite la vue suivante :

```
@Model.FirstName - @Model.LastName
```

On rajoute dans notre vue `persons` des liens vers notre nouveau endpoint :

```
@foreach (Person person in @Model)
{
    <p><a href="/persons/@person.Id">@person.FirstName - @person.LastName</a></p>
}
```

Les relations avec EF Core

Pour créer des relations, il faut utiliser les méthodes `HasOne`, `HasMany`, `WithOne`, `WithMany` dans la méthode `OnModelCreating` du `DbContext`. Les méthodes "Has" permettent de définir le premier dans de la relation, et les méthodes With, permettent de définir l'autre coté. Cela permet de créer toutes les relations possibles. Exemple avec une relation "1 lié à n" (avec un livre lié à des pages) :

```
        modelBuilder.Entity<Book>()
            .HasMany(book => book.Pages) // Premier sens
            .WithMany(page => page.Book); // Sens inverse
```

On est pas obligé de passer de paramètre dans la méthode de sens inverse, si on a pas besoin de la navigation en sens inverse (ici : si on a pas besoin d'avoir de référence au livre dans la page).

Ensuite, pour récupérer dans une requête le conteu de la propriété de navigation, il faut utiliser `Include` :

```
context.Books
    .Include(book => book.Pages)
    .ToList();
```

Dockeriser une application ASP .NET Core

D'abord on build un exécutable de notre application :

```
dotnet publish -c Release
```

On crée ensuite le Dockerfile suivante à la racine du projet

```
# On part d'une image microsoft pour ASP .NET Core
FROM mcr.microsoft.com/dotnet/aspnet:6.0

# On copie notre résultat de publication dans le conteneur
COPY bin/Release/net6.0/publish/ App/

# On se place là où on a copié
WORKDIR /App

# On expose le port 80
EXPOSE 80

# On lance l'application
ENTRYPOINT ["dotnet", "net-web-tuto.dll"]
```

On peut ensuite build l'image :

```
docker build -t mon-image -f Dockerfile .
```

Et enfin lancer l'image :

> Attention, si votre base de donnée est aussi un conteneur Docker tournant sur votre machine, dans l'hote de la base de donnée, notez non pas "localhost" mais

> l'adresse locale de votre machine pour permettre à la connexion de loopback entre vos conteneurs.

```
docker run -p 80:80 -e DATABASE_HOST=<IP de votre base de donnée> -e DATABASE_NAME=testdb -e DATABASE_USERNAME=root -e DATABASE_PASSWORD=monMdpTresSecret -t mon-image
```

Sources

- Documentation d'ASP .NET Core MVC
- Documentation d'Entity Framework Core

Pour aller plus loin

Si vous voulez aller plus loin, je vous conseille de vous intéresser aux sujet suivants :

- API ReST avec ASP .NET Core WebAPI
- Architectures Logicielles Backend :
  - N-Tiers
  - Clean Architecture

Des tutos sur ces sujets apparaitront peut être sur ce site à l'avenir !

Revision #36
Created 2021-11-17 19:18:58 UTC by Arsène Lapostolet
Updated 2022-09-28 12:47:02 UTC by Arsène Lapostolet