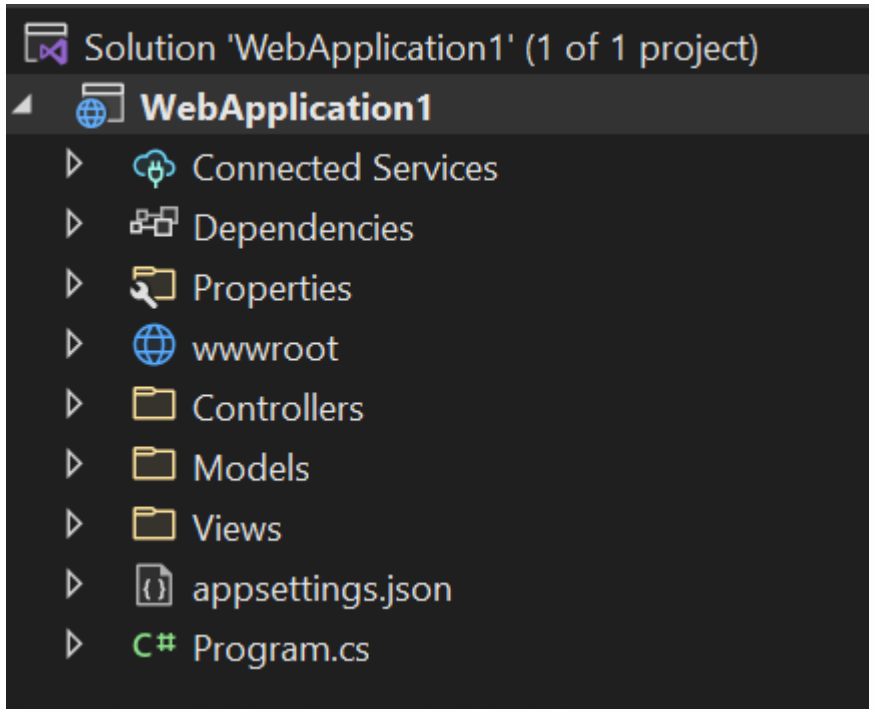
ASP.NET Core MVC

Création d'Applications Webs et d'APIs

Antoine Dieudonné

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

La Structure d'un Projet



Une application Web de type ASP.NET MVC se compose
de trois grandes parties :

Les **Models :** Ce sont les classes qui représentent les entités
de votre application, tout comme les classes d'une application console
représentent les objets de la vie courante

Les **Controllers :** Ce sont les éléments de votre application
se concentrant sur l'exécution et la réalisation de votre logique métier.
Les controlleurs manipulent les modèles et retransmettent
ces changements dans les vues.

Les **Views :** Ce sont les élements servant à la présentation de l'application à l'utilisateur, votre interface en somme.
Les vues sont au format **.cshtml**, une variante de l'HTML permettant l'incorporation de C# dans votre HTML

Program.cs

Le fichier ***Program.cs*** joue le rôle de chef d'orchestre de votre application, c'est ici que se jouent l'**inversion de contrôle** et l'**injection de dépendances**.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Les Middlewares

ASP.NET repose sur le principe de l'utilisation de multiples Middlewares qui se succèdent
lors des requêtes utilisateurs et des réponses du serveur. (Une demande de l'utilisateur traverse une série de modules
Avant d'être réceptionnée par le contrôleur et vis-versa)
Les middlewares sont configurés par les méthodes ***Map()***, ***Run()*** et ***Use()***

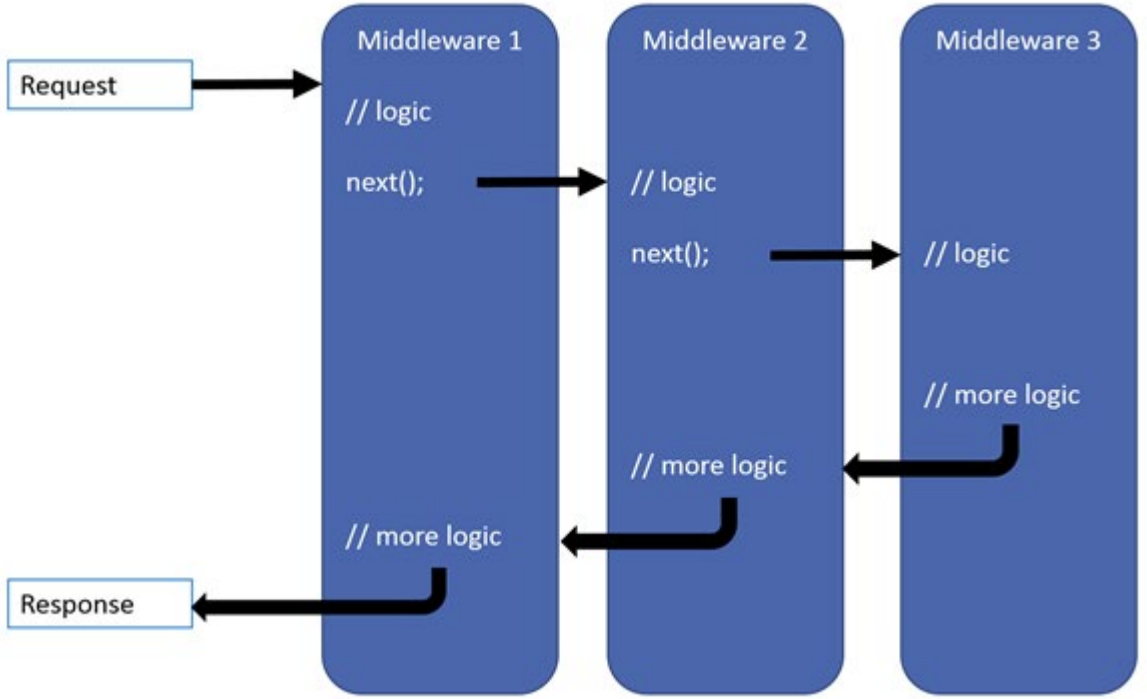
Un middleware est une classe qui est configurée par l'application dans le cas où on en spécifie l'utilisation,
comme par exemple ***app.UseRouting();***

L'objectif de l'utilisation des middlewares est de réaliser une
configuration de l'application répondant à nos besoins
(réalisation d'une pipeline d'interlogiciels)

```
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();
```



Injection de Dépendances

L'injection de dépendance est une notion qui permet d'éviter ce qu'on appelle un **couplage fort**,
et qui améliore une application par une meilleure maintenance et une meilleure scalability.

En effet, lors de l'instanciation d'une dépendance dans un autre objet, un couplage fort se crée, rendant l'objet A dépendant de l'utilisation d'un objet B pour son fonctionnement.

En utilisant à la place des **interfaces** et en injectant en fonction de nos besoin des classes respectant ces interfaces,
il est possible de briser ce couplage fort. C'est ce qu'on appelle l'injection de dépendances.

```
1 reference
public class BaseController : Controller
{
    private UploadService _uploadService;
    0 references
    public BaseController()
    {
        _uploadService = new UploadService();
    }

    0 references
    public IActionResult Index()
    {
        return View();
    }
}
```

Couplage fort

```
1 reference
public class BaseController : Controller
{
    private IUpload _uploadService;
    0 references
    public BaseController(IUpload uploadService)
    {
        _uploadService = uploadService;
    }

    0 references
    public IActionResult Index()
    {
        return View();
    }
}
```

Couplage faible

Inversion de Contrôle

L'inversion de contrôle est une notion qui fournit une abstraction pour l'instanciation des objets dans les contrôleurs.
Pour cela, on se sert d'un conteneur, qui dans notre cas sera le configurateur des services.
En ajoutant des méthodes à ***builder.Services***, on peut injecter des services selon trois cycles de vie :

Transient : Le service est recréé à chaque fois que le contrôleur doit en avoir besoin

Scoped : Le service est créé une seule fois par connexion cliente

Singleton : Le service est créé une fois puis est utilisé par toute l'application

```
// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddScoped<IUpload, UploadService>();
```

```
builder.Services.AddControllersWithViews();
builder.Services.AddSingleton<IUpload, UploadService>();
```

```
builder.Services.AddControllersWithViews();
builder.Services.AddTransient<IUpload, UploadService>();
```

Les Contrôleurs

Les contrôleurs se chargent de renvoyer les vues après l'exécution ou non d'une logique métier.

Pour cela, il y a plusieurs façon de retourner une vue (on ne considère actuellement pas l'utilisation du routing) :

La **première** consiste à renvoyer une vue basée sur le nom de la méthode du contrôleur, via ***return View()***
Cette vue devra alors se trouver dans le dossier Views, dans un sous-dossier portant le nom du contrôleur
et enfin avoir le nom de la méthode employée. Si aucune vue n'est trouvée dans cette hierarchie de dossier, alors le logiciel ira chercher dans le dossier **Shared.**

La **seconde** consiste à préciser la vue demandée (via une string). La vue demandée devra alors suivre la même logique de positionnement que la Première méthode mais pourra porter un nom différent.

Enfin, il est possible qu'une méthode de contrôleur ne renvoie pas Forcément de vue, mais renvoie vers une autre action, via l'utilisation d'un ***return RedirectToAction()***

```
0 references
public IActionResult Index()
{
    return View();
}

0 references
public IActionResult TestA()
{
    return View("Index");
}

0 references
public IActionResult TestB()
{
    return RedirectToAction("TestA");
}
```

Les Contrôleurs

Il est également possible d'envoyer des informations dans la vue, et pour cela nous disposons de trois méthodes :

L'utilisation du **ViewData :** Le Viewdata est un dictionnaire d'objets, auquel on accède par utilisation d'une clé de type string.

L'utilisation du **ViewBag :** Le Viewbag se sert de variables de type dynamic, ce qui lui permet d'emporter avec lui n'importe quel type de variables

L'utilisation d'un **Model :** Il n'est possible d'envoyer qu'un seul modèle dans notre vue, et son utilisation au sein de la vue devra être précédée d'un typage de ce modèle pour accéder à ses méthodes et propriétés.

```
0 references
public IActionResult Index()
{
    ViewData["Message"] = "Ceci est un string";
    ViewData["HasPets"] = true;

    ViewBag.Blabla = "Blabla";
    ViewBag.IsMarried = true;
    ViewBag.Age = 254;

    List<Dog> _dogs = new List<Dog>()
    {
        new Dog() {Name = "Albert", Color = "Beige"},
        new Dog() {Name = "Rex", Color = "Black"}
    };

    return View(_dogs);
}
```

Les Contrôleurs

En plus d'envoyer des valeurs, il est possible d'en recevoir et de s'en servir. Pour exemples, les deux méthodes ci-dessous :

La première est une méthode permettant au navigateur de réaliser un **GET** pour obtenir les informations du chien voulu. Pour cela, l'utilisateur devra respecter le chemin spécifié par le routage défini dans le fichier ***Program.cs***
et envoyer comme paramètre un ***int*** représentant l'Id du chien voulu.

La seconde est une méthode de type édition, qui permettrait l'édition d'une couleur de chien en envoyant depuis le navigateur un **POST** contenant en paramètres à la fois l'Id du chien en ***int*** à modifier et la couleur du chien en ***string***

```
[HttpGet]
0 references
public IActionResult GetDog(int id)
{

    return View("DogDetails", _dogs[id]);
}

[HttpPost]
0 references
public IActionResult TestB(int dogId, string newColor)
{
    _dogs[dogId].Color = newColor;

    return RedirectToAction("Index");
}
```

```
[HttpPost]
0 references
public IActionResult TestB(int dogId, string newColor)
{
    Dog dogFound = _dogs.Find(d => d.Id == dogId);

    if(dogFound == null) return View("Error");

    dogFound.Color = newColor;

    ViewBag.Message = $"{dogFound.Name} a bien été modifié ! ";

    return RedirectToAction("Index");
}
```

Version plus poussée

Les Vues

De leur côté, les vues servent à la présentation de l'application à l'utilisateur. Pour cela, on se sert d'HTML, de CSS et de C#.

Que ce soit le **ViewData**, le **ViewBag** ou le **Model**, les trois peuvent être accessibles depuis une vue. Cependant, des différences se posent lors de leur utilisation :

Le fait que le **Model** soit typé tout en haut de la vue permet d'avoir accès à ses méthodes et à ses champs depuis l'intellisense.

Cela n'est pas possible pour le **ViewBag** et le **ViewData**, qui doivent être castés pour profiter d'intellisense.

ViewBag étant un objet, il dispose de base des méthodes de la classe ***System.Object,*** ce qui n'est pas le cas pour le ViewBag.

Attention donc à leur utilisation, sous peine d'avoir de nombreuses erreurs…

```
@model List<Dog>

@{
    ViewData["Title"] = "Mes chiens";
}
<div class="row container">
    <div class="col-md-8 offset-2">
        <h2>Ma liste de chiens</h2>

        @ViewBag.Blabla.sasdfgfgd
        @(((Dog) ViewBag.DogA).Name);

        @ViewData["abracadabra"]?.ToString();
        @(((Dog) ViewData["ChienA"]).Name);

        <table class="table">
            <thead>
                <tr>
                    <th>Id.</th>
                    <th>Nom</th>
                    <th>Couleur</th>
                </tr>
            </thead>

            <tbody>
                @foreach(Dog dog in Model)
                {
                    <tr>
                        <td>@dog.Id</td>
                        <td>@dog.Name</td>
                        <td>@dog.Color</td>
                    </tr>
                }
            </tbody>
        </table>
    </div>
</div>
```