ASP.NET Core : inversion de contrôle (IOC) et injection de dépendance

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

ASP.NET Core

Dépendance entre objets ?

Les objets de la classe C_1 dépendent des objets de la classe C_2 si :

- C_1 a un attribut objet de la classe C_2
- C_1 hérite de la classe C_2
- C_1 dépend d'un autre objet de type C_3 qui dépend d'un objet de type C_2
- Une méthode de C_1 appelle une méthode de C_2

ASP.NET Core

En programmation objet classique

Le développeur :

- Instancie les objets nécessaires pour le fonctionnement de son application (avec l'opérateur `new`)
- Prépare les paramètres nécessaires pour instancier ses objets
- Définit les liens entre les objets

ASP.NET Core

Avec **ASP.NET Core**

- Programmation par interface : couplage faible
- Injection de dépendance

ASP.NET Core

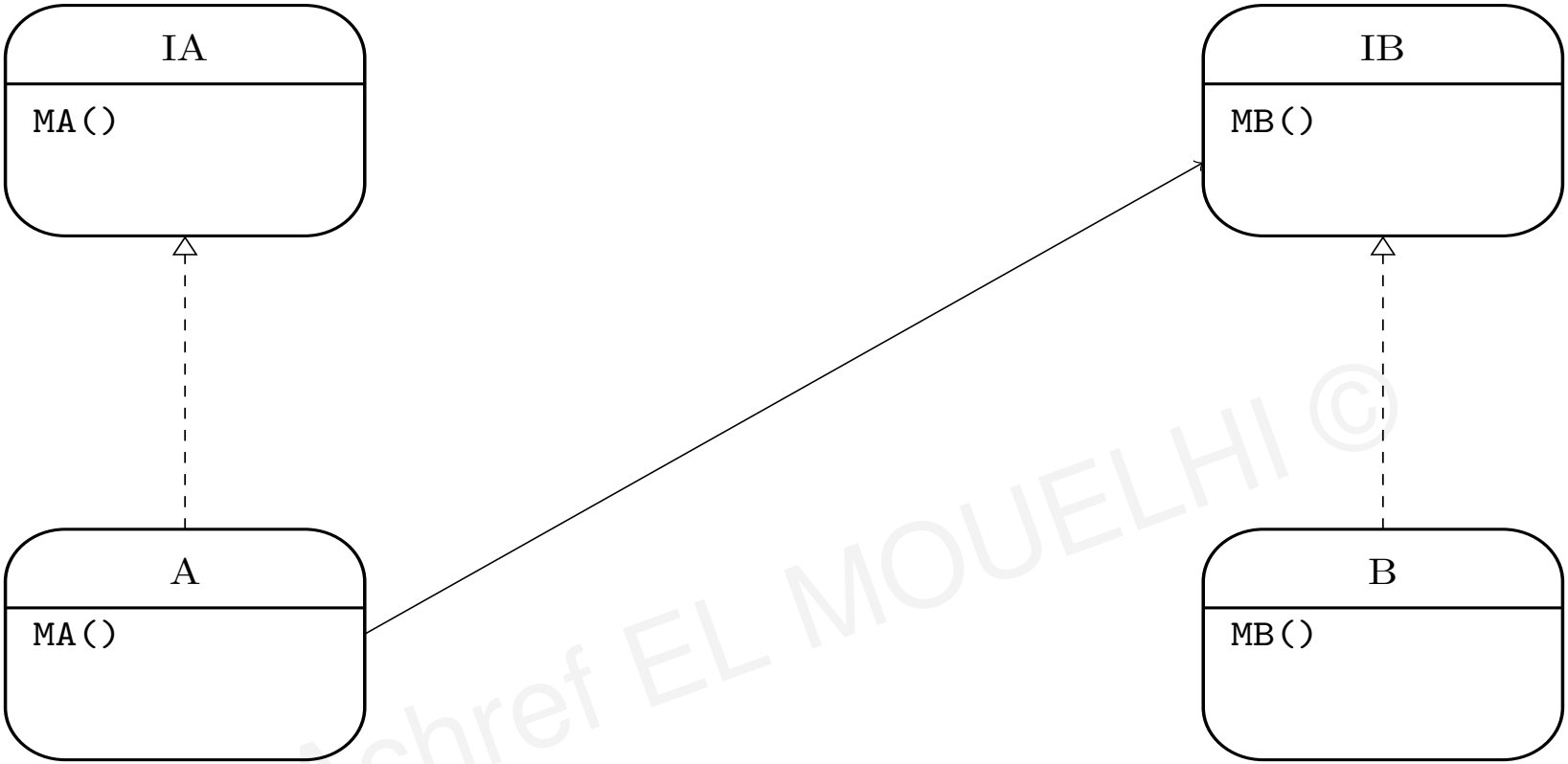
ASP.NET Core

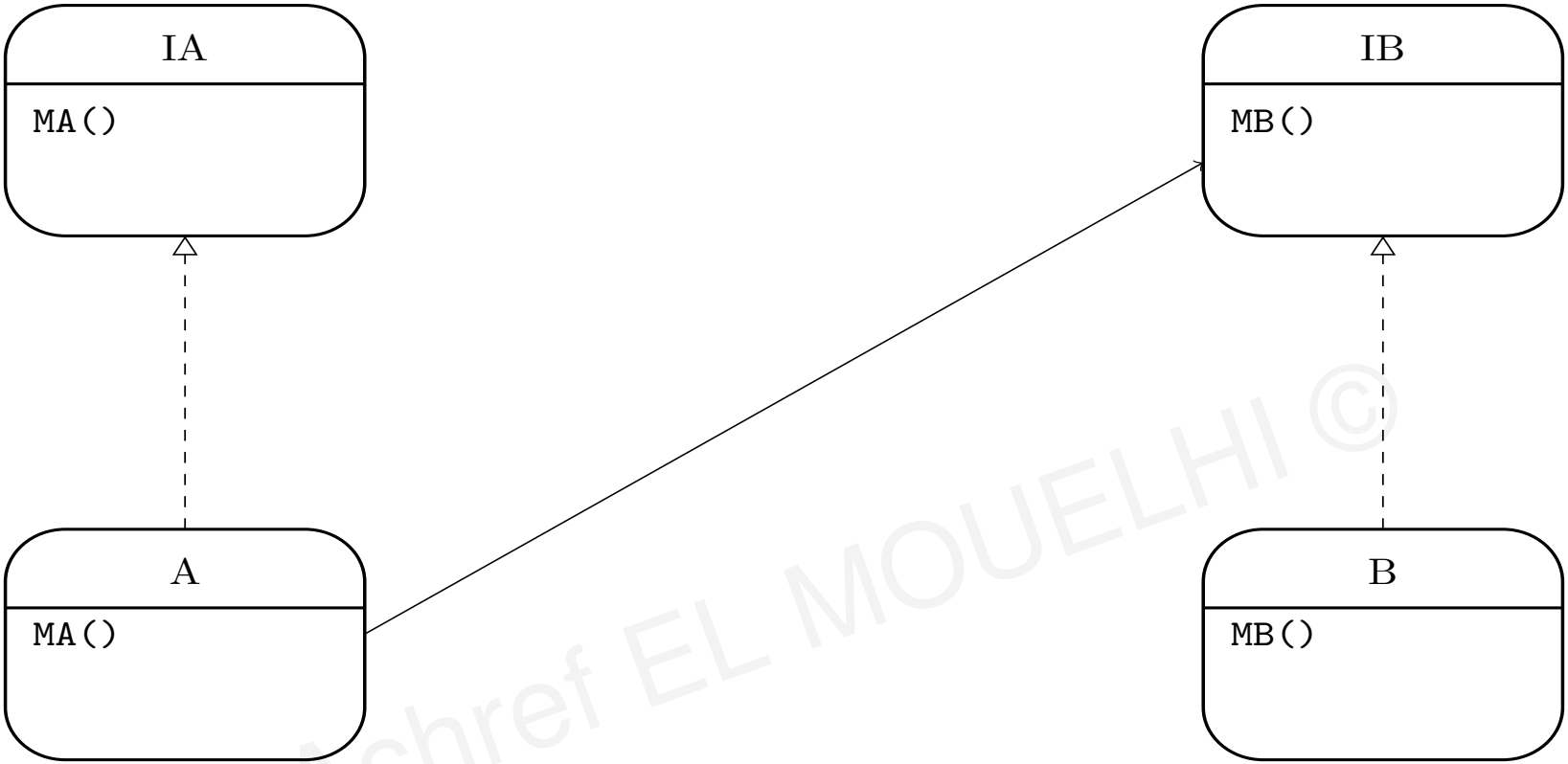
Contenu de la méthode `main`

```
main () {
   A obj = new A ();
   obj.MA();
}
```

Contenu de la classe `A`

```
B obj = new B ();
MA () {
   obj.MB();
}
```





Contenu de la méthode `main`

```
main () {
  IA ia;
  IB ib;
  ia.obj = ib;
  ia.MA();
}
```

Contenu de la classe `A`

```
IB obj;
MA () {
  obj.MB();
}
```

ASP.NET Core

Remarques

- On ne peut instancier les interfaces `IA` et `IB`
- Mais, on peut injecter les classes qui les implémentent pour créer les deux objets `ia` et `ib`

ASP.NET Core

Inversion de contrôle par injection de dépendance

- Patron de conception
- Permettant de dynamiser la gestion de dépendance entre objets
- Facilitant l'utilisation des composants
- Minimisant l'instanciation statique d'objets (avec l'opérateur `new`)

ASP.NET Core

Code simplifié avec **.NET 6** et **C# 10**

- Simplification de l'écriture du `Main` : point d'entrée
- Simplification de l'utilisation des namespaces avec `using`

ASP.NET Core

Code des versions précédentes

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ProjectName
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```


Code des versions précédentes

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

ASP.NET Core

Explication

- L'instruction précédente est supposée être dans le `Main` de la classe `Program`
- L'espace de noms `System` ainsi que certains autres sont automatiquement importés.

ASP.NET Core

Explication

- L'instruction précédente est supposée être dans le `Main` de la classe `Program`
- L'espace de noms `System` ainsi que certains autres sont automatiquement importés.

Liste d'espaces de noms importés (peut changer en fonction du type de projet)

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

ASP.NET Core

Remarques

- Il est possible d'utiliser `using` pour importer un autre espace de noms en cas de besoin.
- Il est possible de définir un `using` global pour tous les fichiers du projet
  - soit en utilisant la syntaxe `global using`,
  - soit en définissant un nouvel item dans le fichier de propriétés.

ASP.NET Core

Remarques

- Il est possible d'utiliser `using` pour importer un autre espace de noms en cas de besoin.
- Il est possible de définir un `using` global pour tous les fichiers du projet
  - soit en utilisant la syntaxe `global using`,
  - soit en définissant un nouvel item dans le fichier de propriétés.

Pour définir un `using` global, on double clique sur le projet et on ajoute le code suivant

```
<Using>NameSpace.Souhaité</Using>
```

ASP.NET Core

Étapes

- Créer un nouveau projet `Fichier > Nouveau > Projet`
- Choisir `C#` dans `Tous les langages`
- Sélectionner `ASP.NET Core Vide`
- Cliquer sur `Suivant`
- Remplir les champs
  - `Nom` avec `CoursInjectionDeDependance`
  - `Solution` avec `CoursAspNetCore`
- Valider

ASP.NET Core

Structure du projet

- `Connected Services` : liste des services Web consommés par l'application.
- `Properties/launchSettings.json` : fichier de configuration utilisé par **Visual Studio** au démarrage de l'application.
- `appsettings.json` : fichier de configuration pouvant contenir les variables globales, les données de connexion à la base de données...
- `Dépendances` : répertoire contenant les librairies indispensables pour le démarrage de l'application.
- `Program.cs` : point d'entrée d'une application **ASP.NET Core**.

ASP.NET Core

Créons un premier POCO `Personne` dans `Models`

```
namespace CoursInjectionDeDependance.Models
{
    public class Personne
    {
        public int Num { get; set; }
        public string? Nom { get; set; }
        public string? Prenom { get; set; }
        public int Age { get; set; }
    }
}
```

ASP.NET Core

Dans `Services`, créons l'interface service suivante

```
using CoursInjectionDeDependance.Models;

namespace CoursInjectionDeDependance.Services
{
    public interface IPersonneService
    {
        List<Personne> GetAll();
        Personne? GetOneById(int id);
    }
}
```

ASP.NET Core

Dans `Services`, créons une classe `PersonneService` qui implémente `IPersonneService`

```
namespace CoursInjectionDeDependance.Services
{
    public class PersonneService: IPersonneService
    {
    }
}
```

ASP.NET Core

Utilisons Visual Studio pour ajouter les méthodes manquantes

```
namespace CoursInjectionDeDependance.Services
{
    public class PersonneService: IPersonneService
    {
        public List<Personne> GetAll()
        {
            throw new NotImplementedException();
        }

        public Personne? GetOneById(int id)
        {
            throw new NotImplementedException();
        }
    }
}
```

Ajoutons le constructeur et implémentons les méthodes de l'interface `IPersonneService`

```
using CoursInjectionDeDependance.Models;

namespace CoursInjectionDeDependance.Services
{
    public class PersonneService : IPersonneService
    {
        public List<Personne> Personnes { get; set; }
        public PersonneService()
        {
            Personnes = new List<Personne>()
            {
                new Personne(){ Num = 1, Nom = "Wick", Prenom = "John", Age = 45},
                new Personne(){ Num = 2, Nom = "Dalton", Prenom = "Jack", Age = 40},
                new Personne(){ Num = 3, Nom = "Maggio", Prenom = "Sophie", Age = 20},
            };
        }
        public List<Personne> GetAll()
        {
            return Personnes;
        }
        public Personne? GetOneById(int id)
        {
            foreach (Personne personne in Personnes)
            {
                if (personne.Num == id)
                {
                    return personne;
                }
            }
            return null;
        }
    }
}
```

ASP.NET Core

Dans `Controllers`**, créons une classe** `PersonnesController`

```
namespace CoursInjectionDeDependance.Controllers
{
    public class PersonnesController
    {
    }
}
```

ASP.NET Core

Ajoutons le service `PersonneService` **dans** `PersonnesController`

```
using CoursInjectionDeDependance.Services;

namespace CoursInjectionDeDependance.Controllers
{
    public class PersonnesController
    {
        PersonneService personneService = new PersonneService();
    }
}
```

ASP.NET Core

Ajoutons ensuite les deux actions suivantes

```
using CoursInjectionDeDependance.Models;
using CoursInjectionDeDependance.Services;

namespace CoursInjectionDeDependance.Controllers
{
    public class PersonnesController
    {
        PersonneService personneService = new PersonneService();

        public List<Personne> Get()
        {
            return personneService.GetAll();
        }

        public Personne? Get(int id = 1)
        {
            return personneService.GetOneById(id);
        }
    }
}
```

ASP.NET Core

Dans `Program`, utilisons l'espace de noms `Controllers`

```
using CoursInjectionDeDependance.Controllers;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();
app.MapGet("/", () => "Hello World!");

app.Run();
```

ASP.NET Core

Instancions ensuite `PersonnesController`

```
using CoursInjectionDeDependance.Controllers;

var builder = WebApplication.CreateBuilder(args);

PersonnesController personnesController =  new PersonnesController();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");
app.MapGet("/personnes", () => personnesController.Get());

app.Run();
```

ASP.NET Core

Préparons l'appel de nos deux actions

```
using CoursInjectionDeDependance.Controllers;

var builder = WebApplication.CreateBuilder(args);

PersonnesController personnesController =  new PersonnesController();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");
app.MapGet("/personnes", () => personnesController.Get());
app.MapGet("/personnes/1", () => personnesController.Get(1));

app.Run();
```

ASP.NET Core

Avant de tester

Installer l'extension **JSON Formatter** pour mieux visualiser le résultat.

ASP.NET Core

En allant à `/personnes` le résultat est


```
[
    {
        "num": 1,
        "nom": "Wick",
        "prenom": "John",
        "age": 45
    },
    {
        "num": 2,
        "nom": "Dalton",
        "prenom": "Jack",
        "age": 40
    },
    {
        "num": 3,
        "nom": "Maggio",
        "prenom": "Sophie",
        "age": 20
    }
]
```


En allant à `/personnes/1` le résultat est


```
{
    "num": 1,
    "nom": "Wick",
    "prenom": "John",
    "age": 45
}
```

ASP.NET Core

Hypothèse

Proposer une nouvelle version de la méthode `GetOneById`, **plus courte, en utilisant LINQ**.

ASP.NET Core

Hypothèse

Proposer une nouvelle version de la méthode `GetOneById`, plus courte, en utilisant **LINQ**.

Remarque

On ne modifie pas la classe `PersonneService` mais on crée une nouvelle.

ASP.NET Core

Notre nouvelle classe de service utilisant LINQ

```
using CoursInjectionDeDependance.Models;

namespace CoursInjectionDeDependance.Services
{
    public class PersonneServiceLinq : IPersonneService
    {
        private List<Personne> Personnes { get; set; }

        public PersonneServiceLinq()
        {
            Personnes = new List<Personne>()
            {
                new Personne(){ Num = 1, Nom = "Wick", Prenom = "John", Age = 45},
                new Personne(){ Num = 2, Nom = "Dalton", Prenom = "Jack", Age = 40},
                new Personne(){ Num = 3, Nom = "Maggio", Prenom = "Sophie", Age = 20},
            };
        }

        public List<Personne> GetAll()
        {
            return Personnes;
        }

        public Personne? GetOneById(int id)
        {
            return Personnes.Find(elt => elt.Num == id);
        }
    }
}
```

ASP.NET Core

Utilisation du nouveau service : première solution

Parcourir tous les fichiers et remplacer toutes les occurrences de `PersonneService` par `PersonneServiceLinq`.

ASP.NET Core

Utilisation du nouveau service : première solution

Parcourir tous les fichiers et remplacer toutes les occurrences de `PersonneService` par `PersonneServiceLinq`.

Utilisation du nouveau service : deuxième solution

Injection de dépendance.

ASP.NET Core

Injection de dépendances : comment ça marche dans **.NET Core 6** ?

Conteneur de services (appelé `Services` dans une application **ASP.NET Core 6**)

- Annuaire de mapping entre types abstraits (interfaces...) et leurs implémentations.
- Techniquement représenté par `IServiceProvider` contenant une structure pour stocker les services : `IServiceCollection`.

ASP.NET Core

Injection de dépendances : comment ça marche dans **.NET Core 6** ?

Conteneur de services (appelé `Services` dans une application **ASP.NET Core 6**)

- Annuaire de mapping entre types abstraits (interfaces...) et leurs implémentations.
- Techniquement représenté par `IServiceProvider` contenant une structure pour stocker les services : `IServiceCollection`.

Remarque

Service $\neq$ Service Web, bien que le service puisse utiliser un Service Web.

ASP.NET Core

Étapes

- Ne plus instancier des classes services dans les contrôleurs.
- Injecter les interfaces services dans le constructeur des contrôleurs.
- Indiquer dans `Program.cs` ce qu'il faut instancier quand on référence l'interface.

Nouveau contenu de `PersonnesController`

```
using CoursInjectionDeDependance.Models;
using CoursInjectionDeDependance.Services;

namespace CoursInjectionDeDependance.Controllers
{
    public class PersonnesController
    {
        private readonly IPersonneService personneService;

        public PersonnesController(IPersonneService personneService)
        {
            this.personneService = personneService;
        }
        public List<Personne> Get()
        {
            return personneService.GetAll();
        }
        public Personne? Get(int id = 1)
        {
            return personneService.GetOneById(id);
        }
    }
}
```

ASP.NET Core

Constat

Dans `Program.cs`, l'instanciation du contrôleur est soulignée en rouge car le constructeur de ce dernier prend un paramètre (l'interface service).

ASP.NET Core

Constat

Dans `Program.cs`, l'instanciation du contrôleur est soulignée en rouge car le constructeur de ce dernier prend un paramètre (l'interface service).

Solution

- Ne pas instancier le contrôleur et demander à **.NET** de le faire.
- Définir les routes dans les classes contrôleurs.
- Préciser ce le service qu'il faut instancier lorsqu'on référence l'interface service.

ASP.NET Core

Commençons par ajouter notre contrôleur en tant que service ASP.NET (après avoir supprimée l'instanciation du contrôleur et de ses routes

```
using CoursInjectionDeDependance.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

ASP.NET Core

Ajoutons les routes dans `PersonnesController`

```
using CoursInjectionDeDependance.Models;
using CoursInjectionDeDependance.Services;
using Microsoft.AspNetCore.Mvc;

namespace CoursInjectionDeDependance.Controllers
{
    public class PersonnesController
    {
        private readonly IPersonneService personneService;

        public PersonnesController(IPersonneService personneService)
        {
            this.personneService = personneService;
        }

        [Route("/personnes")]
        public List<Personne> Get()
        {
            return personneService.GetAll();
        }

        [Route("/personnes/1")]
        public Personne? Get(int id = 1)
        {
            return personneService.GetOneById(id);
        }
    }
}
```

ASP.NET Core

Chargeons les routes dans `Program.cs`

```
using CoursInjectionDeDependance.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");
app.MapControllers();

app.Run();
```

ASP.NET Core

Indiquons à ASP.NET ce qu'il faut instancier lorsqu'on référence l'interface service

```
using CoursInjectionDeDependance.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IPersonneService, PersonneServiceLinq>();
builder.Services.AddControllers();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");
app.MapControllers();

app.Run();
```

ASP.NET Core

En allant à `/personnes` le résultat est


```
[
    {
        "num": 1,
        "nom": "Wick",
        "prenom": "John",
        "age": 45
    },
    {
        "num": 2,
        "nom": "Dalton",
        "prenom": "Jack",
        "age": 40
    },
    {
        "num": 3,
        "nom": "Maggio",
        "prenom": "Sophie",
        "age": 20
    }
]
```


En allant à `/personnes/1` le résultat est


```
{
    "num": 1,
    "nom": "Wick",
    "prenom": "John",
    "age": 45
}
```

ASP.NET Core

Constat

Grâce à la méthode `AddScoped`, on a indiqué ce qu'il faut instancier lorsqu'on référence une interface.

ASP.NET Core

Constat

Grâce à la méthode `AddScoped`, on a indiqué ce qu'il faut instancier lorsqu'on référence une interface.

Question

Y en a-t-il d'autres ?

ASP.NET Core

Réponse

- `AddTransient` : une instance par appel.
- `AddScoped` : une instance par requête **HTTP**.
- `AddSingleton` : une seule instance pour tous les appels.

ASP.NET Core

Quelques règles pour l'injection de dépendances par constructeur

- Le constructeur doit être `public`.
- Une seule surcharge du constructeur avec paramètre.

ASP.NET Core

L'injection de dépendance, dans **ASP.NET Core**, peut se faire par

- Le constructeur,
- Le setter.

ASP.NET Core

L'injection de dépendance, dans **ASP.NET Core**, peut se faire par

- Le constructeur,
- Le setter.

Remarque

Dans le cas où une seule action a besoin du service, on peut directement injecter ce dernier dans la méthode avec l'attribut `[FromServices]`.

ASP.NET Core

Exemple

```
using CoursInjectionDeDependance.Models;
using CoursInjectionDeDependance.Services;
using Microsoft.AspNetCore.Mvc;

namespace CoursInjectionDeDependance.Controllers
{
    public class PersonnesController
    {
        [Route("/personnes")]
        public List<Personne> Get([FromServices] IPersonneService personneService)
        {
            return personneService.GetAll();
        }
    }
}
```