Chapitre 6
Les nouveaux mécanismes d'ASP.NET Core

1. Introduction

La nouvelle version d'ASP.NET Core intègre plusieurs nouveaux mécanismes qui permettent aux développeurs de mieux gérer certains aspects de leurs projets. Dans les versions précédentes, lorsque le projet nécessitait l'exposition d'API au travers de l'application web, la brique Web API proposait ses propres API et classes afin d'exposer des services vers l'extérieur. Par exemple, il existait une classe de base `ApiController` pour les API et une classe `Controller` pour les pages de l'application, alors que le fonctionnement d'un contrôleur est toujours le même : récupérer la requête HTTP, traiter les données et renvoyer une réponse.

Il en est de même pour l'accès aux données qui était souvent fastidieux. Le code métier était pollué de plusieurs blocs *using* afin de s'assurer que la connexion à la base de données était bien fermée. Cela est sans conteste une bonne pratique que de fermer la connexion au plus tôt, mais la lisibilité du code subissait les conséquences. ASP.NET Core intègre un nouveau système de dépendances permettant de mieux gérer ce genre de cas, et d'augmenter la maintenabilité du code métier.

Ce chapitre va traiter de quelques nouveautés extrêmement pratiques du framework. Tout d'abord, la prochaine section traitera du sujet de l'injection de dépendances afin de bien gérer les dépendances entre les services d'une application ASP.NET Core. Ensuite, la section suivante traitera des middlewares et de l'importance d'utiliser uniquement ce dont on a besoin. Puis, le chapitre finira par un tour d'horizon concernant les Web API.

2. L'injection de dépendances

L'injection de dépendances d'ASP.NET Core est un ensemble de services et de mécanismes préintégrés au framework afin d'injecter des services dans toute l'application. Dans les versions précédentes, le développeur avait besoin d'un framework externe alors que maintenant ce n'est plus nécessaire.

Le principe de l'injection de dépendances est une technique consistant à coupler faiblement les objets et les classes de services entre elles et leurs dépendances. Au lieu d'instancier directement les services dans les méthodes par l'intermédiaire des constructeurs ou des `using`, la classe va déclarer quelles sont les dépendances dont elle a besoin pour fonctionner. La plupart du temps, la classe va déclarer ses dépendances dans son constructeur : ce procédé est appelé "*constructor injection*", et permet de respecter les bonnes pratiques intitulées *Explicit Dependencies Principle*. Le but étant que la classe expose de manière explicite ses dépendances dans le constructeur.

Cependant, il est important de concevoir sa classe en gardant le principe de DI (*Dependency Injection*) en tête et de garder ses services faiblement couplés avec ses dépendances. Une autre bonne pratique intitulée *Dependency Inversion Principle* énonce une phrase qui résume très bien la philosophie de DI :

"*High level modules should not depend on low level modules; both should depend on abstractions.*"

Cette phrase est très révélatrice de la méthode à adopter lorsqu'on fait de l'injection de services dans les classes : il faut injecter une abstraction de cette classe, et non la classe elle-même. En C#, cela reviendrait à déclarer une interface comme étant une dépendance de la classe. Ce principe de déporter les dépendances dans des interfaces et de fournir des implémentations concrètes est également un exemple du pattern *Strategy Design*.

Lorsqu'un système est conçu pour utiliser l'injection de dépendances, il a besoin d'un conteneur de dépendances pour répertorier tous les services qui sont potentiellement injectables dans des classes. On parlera alors de conteneur d'inversion de contrôle ou de conteneur d'injection de dépendances. L'inversion de contrôle est également une bonne pratique qui inverse la responsabilité de la création de l'instance lors de la résolution des dépendances : c'est le framework qui va décider quelle instance utiliser pour telle dépendance de tel type.

Remarque

L'injection de dépendances rassemble un bon nombre de bonnes pratiques, et elle-même en est déjà une. Nombreux sont les articles faisant l'éloge de ces concepts, mais un article que nous aimerions vous conseiller a particulièrement retenu notre attention :
http://www.martinfowler.com/articles/injection.html

Avec ce genre de conteneur et l'inversion de contrôle, le développeur peut facilement déclarer un arbre complexe de dépendances, et pendant le temps d'exécution le framework va automatiquement résoudre toutes les dépendances, laissant un code déclaratif simple et épuré. En plus de créer les objets qui correspondent aux types déclarés, le conteneur de services gère tout seul le cycle de vie des objets. ASP.NET Core intègre déjà un conteneur de services matérialisé par l'interface *IServiceProvider*, qui est elle-même injectable dans des classes.

Remarque

Nous parlerons souvent de "service" dans ce chapitre, et probablement à d'autres endroits du livre. Ces services représentent simplement des classes qui sont enregistrées dans le conteneur de services, et donc injectables dans d'autres classes de l'application. Nous verrons comment, plus loin dans cette section.

Le conteneur de services est configurable dans la méthode `ConfigureServices` de la classe `Startup`, et doit être configuré uniquement à cet endroit.

```
public void ConfigureServices(IServiceCollection services)
{
     services.AddMvc();
}
```

Dans l'exemple ci-dessus, la méthode `ConfigureServices` configure l'application pour injecter les services liés à MVC. C'est ici que le développeur va injecter d'autres services liés à Entity Framework ou à ASP.NET Identity via les méthodes d'extensions `AddEntityFramework` ou `AddIdentity`. Cependant, et c'est là tout l'intérêt de ce mécanisme, le développeur peut injecter ses propres services.

Dans le template de base d'ASP.NET Core, les services injectés sont les suivants :

```
services.AddTransient<IEmailSender, AuthMessageSender>();
services.AddTransient<ISmsSender, AuthMessageSender>();
```

La méthode `AddTransient` permet de mapper le type `AuthMessageSender` avec l'interface `IEmailSender`. Concrètement, cela veut dire qu'à chaque fois qu'une classe requiert l'interface `IEmailSender`, le conteneur de services va répondre avec une instance de `AuthMessageSender`. Ensuite, la méthode définit également le cycle de vie du service, et il est important de bien choisir la durée de vie de votre service. Doit-il exister tout le temps de la requête HTTP ? Doit-il être instancié uniquement pour la classe qui l'a demandé ? Ou alors doit-il persister pendant toute la durée de vie de l'application ?

Le contrôleur ci-dessous est un exemple simple d'utilisation de l'injection de dépendances. Il déclare un service dans son constructeur, qu'il utilise ensuite dans sa méthode `Index`. L'important à noter ici est que le service est injecté via le constructeur, ce qui est une bonne pratique. **Ne jamais injecter un service via des propriétés injectées directement dans le contrôleur**. Cela peut produire des comportements hasardeux dans certains cas.

```
    public class ProductsController : Controller
    {
        private readonly IProductsRepository _productsRepository;

        public ProductsController(IProductsRepository
productsRepository)
        {
            _productsRepository = productsRepository;
        }

        // GET: /products/
```

```
        public IActionResult Index()
        {
            return View(this._productsRepository.ListAll());
        }
    }
```

L'interface utilisée est très simple, et permet d'exposer des services de bas niveau pour gérer les produits de l'application.

```
    public interface IproductsRepository
    {
        IEnumerable<Product> ListAll();
        void Add(Product product);
    }
```

Et l'implémentation de cette interface utilise à nouveau l'injection de dépendances pour communiquer avec la base de données, et ainsi effectuer les opérations nécessaires.

```
    public class ProductsRepository : IproductsRepository
    {
        private readonly ApplicationDbContext _dbContext;

        public ProductsRepository(ApplicationDbContext dbContext)
        {
            _dbContext = dbContext;
        }

        public IEnumerable<Product> ListAll()
        {
            return _dbContext.Products.AsEnumerable();
        }

        public void Add(Product character)
        {
            _dbContext.Products.Add(character);
            _dbContext.SaveChanges();
        }
    }
```

Il ne manque plus que l'enregistrement du service dans le conteneur :

```
services.AddScoped<ICharacterRepository, CharacterRepository>();
```

Remarque

L'injection de `ApplicationDbContext` se fait déjà via la méthode d'extension `.AddDbContext<ApplicationDbContext>`.

Ici, `ProductsController` a besoin d'un `ProductsRepository` qui lui-même a besoin d'un `ApplicationDbContext`. Cela crée un arbre de dépendances que le framework sait résoudre automatiquement, à partir du moment où les types sont correctement renseignés dans le conteneur de services. La complexité de résolution des dépendances est à la charge du framework et non plus du développeur.

La deuxième chose importante ici est la transparence totale du code métier se trouvant dans le contrôleur. En effet, ce dernier utilise une interface pour gérer les produits, mais il ne sait rien de l'implémentation. Dans l'exemple ci-dessus, la gestion se fait via la base de données, mais plus tard elle pourrait se faire via des fichiers. Dans ce cas, il suffira simplement de réimplémenter l'interface et fournir la nouvelle implémentation au conteneur de services, mais le code du contrôleur ne changera pas du tout. C'est bien là tout l'intérêt de l'injection de dépendances : le code métier devient totalement transparent pour les consommateurs de ces services qui n'ont pas besoin de se soucier de l'implémentation.

Enfin, la testabilité des classes est bien améliorée grâce à ce mécanisme. Si la classe à tester dépend d'une interface, il suffit de fournir une implémentation simpliste de ce service, et de l'utiliser afin de tester la classe. Si l'on reprend l'exemple ci-dessus, afin de tester le code du contrôleur, il suffirait de réimplémenter l'interface `IProductsRepository` et de fournir ainsi des données fictives plutôt que des données provenant d'une base de données.

Comme indiqué plus haut dans cette section, le choix de la durée de vie des services est important. Le framework fournit quatre moyens d'injecter les services en fonction de la durée de vie choisie :

– **Transient** : une nouvelle instance du service est envoyée à chaque nouvelle demande d'une classe. Cela veut dire que sur une même requête HTTP, le développeur peut se retrouver avec plusieurs instances du même service.
– **Scoped** : le service est créé une fois par requête. C'est le mode le plus utilisé et le plus préconisé, car il permet de ne pas gaspiller des instances inutiles et garantit l'unicité du service pour la requête en cours.