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.

Chapitre 4
Les bases du langage

1. Introduction

Comme tous les langages de programmation, C# impose certaines règles au développeur. Ces règles se manifestent au travers de la syntaxe du langage mais elles couvrent le large spectre fonctionnel proposé par C#. Avant d'explorer en profondeur les fonctionnalités du langage au chapitre suivant, nous étudierons donc les notions essentielles et fondamentales de C# : la création de données utilisables par une application et le traitement de ces données.

2. Les variables

Les données utilisables dans un programme C# sont représentées par des variables. Une variable est un espace mémoire réservé auquel on assigne arbitrairement un nom et dont le contenu est une valeur dont le type est fixé. On peut manipuler ce contenu dans le code en utilisant le nom de la variable.

2.1 Nommage des variables

La spécification du langage C# établit quelques règles à prendre en compte lorsque l'on nomme une variable :

– Le nom d'une variable ne peut comporter que des chiffres, des caractères de l'alphabet latin accentués ou non, le caractère ç ou les caractères spéciaux _ et μ.

– Le nom d'une variable ne peut en aucun cas commencer par un chiffre. Il peut en revanche en comporter un ou plusieurs à toute autre position.

– Le langage est sensible à la casse, c'est-à-dire qu'il fait la distinction entre majuscules et minuscules : les variables `unevariable` et `uneVariable` sont donc différentes.

– La longueur d'un nom de variable est virtuellement illimitée : le compilateur ne remonte pas d'erreur lorsqu'il est composé de 30 000 caractères ! Il n'est évidemment pas recommandé d'avoir des noms de variables aussi longs, le maximum en pratique étant plus souvent de l'ordre de la trentaine de caractères.

– Le nom d'une variable ne peut pas être un mot-clé du langage. Il est toutefois possible de préfixer un mot-clé par un caractère autorisé ou par le caractère @ pour utiliser un nom de variable similaire.

Les noms de variables suivants sont acceptés par le compilateur C# :

– `maVariable`

– `maVariableNumero1`

– `@void` (`void` est un mot-clé de C#)

– `µn3_VàRïãbl3`

De manière générale, il est préférable d'utiliser des noms de variables explicites, c'est-à-dire permettant de savoir à quoi correspond la valeur stockée dans la variable, comme `nomClient`, `montantAchat` ou `ageDuCapitaine`.

2.2 Type des variables

Une des caractéristiques de C# est la notion de typage statique : chaque variable correspond à un type de données et ne peut en changer. De plus, ce type doit être déterminable au moment de la compilation.

2.2.1 Types valeurs et types références

Les différents types utilisables en C# peuvent être décomposés en deux familles : les types valeurs et les types références. Cette notion peut être déconcertante au premier abord, puisqu'une variable représente justement une donnée et donc une valeur. Ce concept est en fait lié à la manière dont est stockée l'information en mémoire.

Lorsque l'on utilise une variable de type valeur, on accède directement à la zone mémoire stockant la donnée. Au moment de la création d'une variable de type valeur, une zone mémoire de la taille correspondant au type est allouée. Chaque octet de cette zone est automatiquement initialisé avec la valeur binaire 00000000. Notre variable aura donc une suite de 0 pour valeur binaire.

Dans le cas d'une variable de type référence, le comportement est différent. La zone mémoire allouée à notre variable contient une adresse mémoire à laquelle est stockée la donnée. On passe donc par un intermédiaire pour accéder à notre donnée. L'adresse mémoire est initialisée avec la valeur spéciale `null`, qui ne pointe sur rien, tandis que la zone mémoire contenant les données n'est pas initialisée. Elle sera initialisée lorsque la variable sera instanciée. Dans le même temps, l'adresse mémoire stockée dans notre variable sera mise à jour.

Une variable de type référence pourra donc ne contenir aucune donnée, tandis qu'une variable de type valeur aura forcément une valeur correspondant à une suite de 0 binaires.

Cette différence de fonctionnement a une conséquence importante : la copie de variable se fait par valeur ou par référence, ce qui signifie qu'une variable de type valeur sera effectivement copiée, tandis que pour un type référence, c'est l'adresse que contient la variable qui sera copiée, et il sera donc possible d'agir sur la donnée réelle indifféremment à partir de chacune des variables pointant sur ladite donnée.

2.2.2 Types intégrés

La plateforme .NET embarque plusieurs milliers de types différents utilisables par les développeurs. Parmi ces types, nous en avons une quinzaine que l'on peut considérer comme fondamentaux : ce sont les types intégrés (aussi nommés types primitifs). Ce sont les types de base à partir desquels sont construits les autres types de la BCL ainsi que ceux que le développeur crée dans son propre code. Ils permettent de définir des variables contenant des données très simples.

Ces types ont comme particularité d'avoir chacun un alias intégré à C#.

Types numériques

Ces types permettent de définir des variables numériques entières ou décimales. Ils couvrent des plages de valeurs différentes et ont chacun une précision spécifique. Certains types seront donc plus adaptés pour les calculs entiers, d'autres pour les calculs dans lesquels la précision décimale est très importante, comme les calculs financiers.

Les différents types numériques sont énumérés ci-dessous avec leurs alias ainsi que les plages de valeurs qu'ils couvrent.

Type	Alias C#	Plage de valeurs couverte	Taille en mémoire
`System.Byte`	`byte`	0 à 255	1 octet
`System.SByte`	`sbyte`	-128 à 127	1 octet
`System.Int16`	`short`	-32768 à 32767	2 octets
`System.UInt16`	`ushort`	0 à 65535	2 octets
`System.Int32`	`int`	-2147483648 à 2147483647	4 octets
`System.UInt32`	`uint`	0 à 4294967295	4 octets
`System.Int64`	`long`	-9223372036854775808 à 9223372036854775807	8 octets
`System.UInt64`	`ulong`	0 à 18446744073709551615	8 octets
`System.Single`	`float`	±1,5e-45 à ±3,4e38	4 octets

Type	Alias C#	Plage de valeurs couverte	Taille en mémoire
`System.Double`	`double`	±5,0e-324 à ±1,7e308	8 octets
`System.Decimal`	`decimal`	±1,0e-28 à ±7,9e28	16 octets

Les types numériques primitifs sont tous des types valeurs. Une variable de type numérique et non initialisée par le développeur aura pour valeur par défaut 0.

Remarque

.NET 4 a apporté le type `System.Numerics.BigInteger` afin de manipuler des entiers d'une taille arbitraire. Ce type est aussi un type valeur, mais il ne fait pas partie des types intégrés.

Les valeurs numériques utilisées au sein du code peuvent être définies à l'aide de trois bases numériques :

– **Décimale** (base 10) : c'est la base numérique utilisée par défaut.

  Exemple : 280514

– **Héxadécimale** (base 16) : elle est très utilisée dans le monde du Web, pour la définition de couleurs, et plus généralement pour encoder des valeurs numériques dans un format plus condensé. **Une valeur héxadécimale est écrite en la préfixant par 0x.**

  Exemple : 0x0447C2

– **Binaire** (base 2) : cette base est celle qui permet de s'approcher au plus près de la manière dont la machine interprète le code compilé. L'utilisation du binaire en C# peut par conséquent se révéler précieuse dans le cadre d'optimisations (décalages de bits au lieu de multiplications par 2, stockage de multiples données au sein d'une même variable, etc.). **Les valeurs binaires sont préfixées par 0b.**

  Exemple : 0b01000100011111000010

La lecture de valeurs numériques peut s'avérer difficile dans certains cas, notamment lorsqu'elles sont représentées sous forme binaire. L'équipe en charge des évolutions de C# a intégré, dans la septième version du langage, la possibilité d'ajouter des séparateurs dans les valeurs numériques afin d'améliorer la lisibilité globale. Le caractère utilisé pour cela est _ :

– 280_514
– 0x04_47_C2
– 0b0100_0100_0111_1100_0010

Types textuels

Il existe dans la BCL deux types permettant de manipuler des caractères Unicode et des chaînes de caractères Unicode : `System.Char` et `System.String`. Ces types ont respectivement pour alias `char` et `string`.

Le type `char` est un type valeur encapsulant les mécanismes nécessaires au traitement d'un caractère Unicode. Par conséquent, une variable de type char est stockée en mémoire sur deux octets.

Les valeurs de type `char` doivent être encadrées par les caractères ' ' : 'a'.

Certains caractères ayant une signification particulière pour le langage doivent être utilisés avec le caractère d'échappement \ afin d'être correctement interprétés. D'autres caractères n'ayant pas de représentation graphique doivent être aussi déclarés avec des séquences spécifiques. Le tableau suivant résume les séquences qui peuvent être utilisées.

Séquence d'échappement	Caractère associé
\'	Simple quote '
\"	Double quote "
\\	Backslash \
\a	Alerte sonore
\b	Retour arrière
\f	Saut de page
\n	Saut de ligne