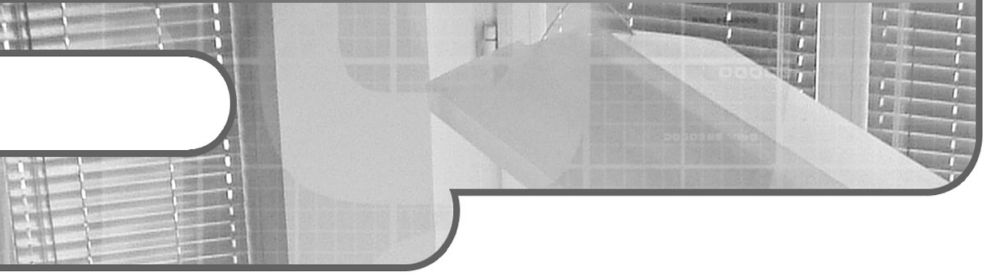


Chapitre 3
Les nouveautés d'ASP.NET Core

1. Introduction

ASP.NET existe depuis 2002 et bien des changements ont été effectués sur le framework depuis sa première version. Il faut rappeler une chose sur ASP.NET Core : la nouvelle plateforme web de Microsoft n'est en aucun cas une suite de la version 4.6 du framework que nous connaissions, mais bien un renouveau qui doit marquer une nouvelle ère de la technologie de Microsoft dans le Web moderne.

Certains diront que le framework n'a pas tant changé que cela (surtout la partie MVC), pourtant c'est bien "sous le capot" que les changements ont été les plus profonds, en commençant par le namespace `System.Web` qui n'existe plus. Ensuite, annoncé comme cross-platform, ASP.NET Core est plus modulable qu'il ne l'a été dans les années précédentes. Via ***NuGet*** pour les composants serveurs, puis via ***Grunt*** ou ***Gulp*** pour la partie cliente du site web, le nouveau framework bénéficie également d'un nouveau runtime, appelé ***CoreCLR***, permettant l'exécution d'une application web Microsoft sur Linux ou Mac.

2. Les nouveaux outils open source

ASP.NET Core arrive avec un tout nouveau panel d'outils open source permettant la gestion des nouveaux projets web. Fort heureusement pour le développement, l'ensemble de ces outils est réuni autour d'une seule et même interface en ligne de commande : dotnet.

2.1 L'environnement d'exécution dotnet

dotnet a été conçu afin de faire fonctionner des applications .NET de manière cross-platform sur Windows, Mac et Linux et ceci sans développer un runtime différent pour chaque plateforme. C'est à la fois un environnement d'exécution et un SDK embarquant ainsi tout ce qui est nécessaire pour le bon fonctionnement des applications web ASP.NET cross-platform.

Totalement orienté ***package-first***, Microsoft a poussé le concept de modularité très loin, permettant même à l'environnement d'exécution d'embarquer, de gérer et de créer de lui-même les packages dont il a besoin, et ceci de manière automatique via NuGet. dotnet est capable de cibler plusieurs frameworks (.NET Core ou le framework .NET Full), et ainsi générer les packages NuGet directement. De plus, dotnet embarque le nouveau moteur d'exécution CoreCLR conçu spécialement pour les problématiques de compatibilité sur les autres plateformes.

dotnet est intégré à Visual Studio 2015 pour offrir une expérience développeur enrichie, mais l'environnement d'exécution est également pilotable via les lignes de commande. Dans un projet ASP.NET Core, il est possible de rajouter des outils Microsoft et ainsi piloter son projet avec dotnet en ligne de commande. Ces outils se rajoutent en même temps que le paquet NuGet dans le fichier *.csproj* :

```
<ItemGroup>
    <DotNetCliToolReference Inclu-de="Microsoft.VisualStudio.Web.
CodeGeneration.Tools" Version="2.0.4" />
    <DotNetCliToolReference Include="BundlerMinifier.Core"
Version="2.0.238" />
  </ItemGroup>
```

Les commandes déclarées permettent par exemple de lancer un outil de génération de code, anciennement appelé *scaffolding*. Ce dernier peut permettre de générer le contrôleur et les vues qui correspondent à un modèle de données bien précis. La commande est utilisable via `dotnet <command>` :

```
dotnet Microsoft.VisualStudio.Web.CodeGeneration.Tools
```

Au sein de l'application elle-même, il est possible d'utiliser les services de dotnet via certaines interfaces C# disponibles via injection de dépendances. On retrouve des services comme `IServiceEnvironment` permettant de modifier la configuration de l'environnement d'exécution pendant que l'application elle-même fonctionne.

L'utilitaire `dotnet` contient également une liste de commandes prédéfinies permettant d'effectuer certaines actions sur le projet ASP.NET Core :

– `dotnet new` : initialise un projet C# console simple.

– `dotnet restore` : restaure les dépendances du projet selon le *.csproj*.

– `dotnet build` : construit l'application .NET Core.

– `dotnet publish` : publie une application .NET Core.

– `dotnet run` : lance l'application depuis les sources.

– `dotnet test` : lance les tests utilisant le *test runner* défini dans le *.csproj*.

– `dotnet pack` : crée un paquet NuGet depuis le code source.

Cet outil supporte ainsi plusieurs processus de lancement de l'application ASP.NET Core. Le premier consiste simplement à restaurer les packages nécessaires à l'application, puis à lancer le projet depuis les sources.

```
dotnet new
dotnet restore
dotnet run
```

Cependant, l'utilitaire permet également de lancer l'application directement depuis une DLL après avoir lancé la génération du projet.

```
dotnet build
dotnet run bin/Debug/netcoreapp1.0/test-app.dll
```

Avec les deux processus ci-dessus, l'outil montre qu'il est capable d'exécuter plusieurs types d'applications :

– Des applications "portables", c'est-à-dire des applications qui dépendent de la version de .NET Core installée sur la machine. Cela veut dire que ce type d'application sera capable de fonctionner sur différentes installations de .NET Core, peu importe les systèmes d'exploitation utilisés. Les applications "portables" permettent également de centraliser les librairies de .NET : elles embarqueront donc uniquement les librairies externes.
– Des applications "autonomes", c'est-à-dire des applications contenant toutes les dépendances dont elles ont besoin pour fonctionner, incluant le runtime .NET Core faisant entièrement partie de l'application. Cela permet de faciliter le déploiement de l'application mais alourdit le package. De plus, il convient à l'application de spécifier la version du runtime à utiliser dans le *.csproj*.

2.2 L'utilitaire dotnet restore

Un projet web ASP.NET Core possède plusieurs dépendances à des packages externes provenant souvent de NuGet. Pour faire fonctionner une application web, il va tout d'abord falloir restaurer les packages nécessaires au bon fonctionnement du projet. C'est une des nouveautés d'ASP.NET Core : le projet embarque uniquement ce dont il a besoin, et donc il a fallu concevoir un outil permettant de restaurer ces dépendances, ceci de manière cross-platform et totalement transparente. dotnet restore permet de faire cette restauration.

Intégré à .NET Core et installé avec dotnet, dotnet restore permet de scruter le projet ASP.NET Core et de récupérer les packages dont il a besoin dans le cloud. Visual Studio 2015 utilise automatiquement cet utilitaire lorsque les dépendances du projet sont mises à jour. Il est cependant possible de lancer le processus à la main grâce à la commande suivante :

```
> dotnet restore
```

Remarque

Il faut noter toutefois que, dans la console depuis laquelle la commande est lancée, il faut se placer à la racine du projet. En effet, `dotnet restore` *va inspecter le fichier .csproj pour identifier les dépendances à récupérer.*

Grâce à dotnet restore et à sa simplicité d'utilisation, il est très facile de restaurer les dépendances d'un projet ASP.NET Core, et ceci indépendamment de la plateforme.

2.3 Gérer ses paquets NuGet avec dotnet pack

L'outil `dotnet pack` est utilisé pour générer un package NuGet à partir d'un projet .NET. Il peut être utilisé de différentes manières pour gérer ses paquets NuGet. Dans un premier temps, nous pouvons utiliser la commande suivante, à la racine d'un projet .NET, afin de simplement générer un paquet NuGet :

```
dotnet pack mon_projet.csproj
```

Cette commande générera un package NuGet à partir du projet spécifié et l'enregistrera dans le répertoire `bin\Debug` ou `bin\Release`, selon le mode de compilation utilisé.

Afin d'inclure les informations de version et de métadonnées dans le package NuGet généré, nous pouvons utiliser les options `--version` et `--metadata` :

```
dotnet pack mon_projet.csproj --version 1.2.3
--metadata authors="John Doe"
```

Dans cet exemple, nous avons spécifié la version `1.2.3` du package et ajouté une métadonnée `authors` avec la valeur "`John Doe`".

Pour publier le package NuGet généré sur un dépôt NuGet, nous pouvons utiliser l'option `--publish` en spécifiant l'URL du dépôt :

```
dotnet pack mon_projet.csproj --publish https://mynugetrepo.com
```

Cette commande publiera le package NuGet généré sur le dépôt NuGet spécifié.

Pour plus d'informations, nous pouvons consulter la documentation officielle ou utiliser la commande `dotnet pack --help` pour afficher la liste complète des options disponibles :

- `--output` : répertoire de sortie pour les paquets générés.
- `--no-build` : génère un paquet NuGet sans lancer la génération du projet .NET.
- `--no-restore` : génère un paquet NuGet sans lancer la restauration des paquets NuGet du projet.
- `--include-symbols` : inclue les symboles de compilations à côté du paquet généré dans le dossier de sortie.
- `--include-source` : inclut les fichiers PDB et les fichiers sources. Les sources iront dans le dossier `src`.
- `--serviceable` : définit le niveau de maintenance du paquet.
- `--nologo` : n'affiche pas le logo lors du démarrage de la commande.
- `--interactive` : permet d'attendre ou non une interaction utilisateur si nécessaire.
- `--verbosity` : indique le niveau de verbosité des logs affichés lors du lancement de la commande.
- `--version-suffix` : définit la valeur de la propriété `$(VersionSuffix)` à utiliser lors de la génération du projet.
- `--configuration` : définit la configuration à utiliser lors de la génération. Les valeurs peuvent être `Debug` ou `Release`.
- `--use-current-runtime` : définit s'il faut utiliser le runtime actuel comme runtime cible.

La commande `dotnet pack` est très utile pour la gestion des paquets NuGet des projets .NET et permet au développeur de créer et publier des paquets facilement.

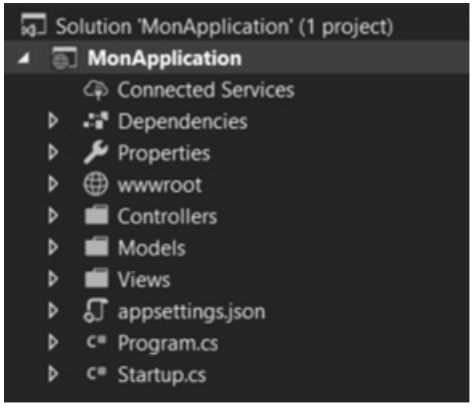
3. La structure d'une solution

Une solution ASP.NET Core est la base d'un projet web utilisant les technologies Microsoft. Elle permet rapidement de déployer un site et de structurer le code utilisé pour faire fonctionner l'application. Une solution peut comporter à la fois le code côté serveur et le code côté client, tout en incluant des mécanismes afin de bien séparer les deux parties. Ce chapitre va traiter des différents composants d'une solution ASP.NET Core et expliquer leurs rôles dans la configuration ou le déploiement de l'application web.

3.1 Les fichiers .csproj

Un projet ASP.NET Core met en œuvre une toute nouvelle philosophie de conception d'applications web issue de chez Microsoft s'inspirant beaucoup de l'open source.

Le nouveau template ressemble à ceci :



Nouveau template de projet ASP.NET Core

Chapitre 4
L'accès aux données avec ADO.NET

1. Les bases d'ADO.NET

Sous .NET, l'accès aux données s'effectue à l'aide du bloc de services ADO.NET. Bien que le framework ASP.NET ait été enrichi de nouveaux contrôles facilitant la lecture et la présentation des données SQL, le développeur devrait toujours considérer l'emploi du mode connecté pour élaborer une application ASP.NET. En effet, les contraintes de charge, d'intégration et d'exécution du web ont une influence très forte sur l'efficacité finale d'une application ASP.NET.

1.1 Le mode connecté

En mode connecté, tous les formats de base de données adoptent le même fonctionnement. Nous l'illustrerons avec SQL Server, et pour passer à d'autres formats, il n'y aura qu'à changer d'espace de noms et à modifier le préfixe de chaque classe.

Le tableau suivant donne la marche à suivre pour appliquer ces changements :

	Espace de noms	Connexion	Commande
SQL Server	System.Data.SqlClient	**Sql**Connection	**Sql**Command
Access	System.Data.OleDB	**OleDB**Connection	**OleDB**Command
Oracle	System.Data.OracleClient	**Oracle**Connection	**Oracle**Command

Les autres classes sont nommées sur le même principe.

1.1.1 La connexion

La connexion SqlConnection désigne un canal par lequel sont échangés les ordres et les lignes SQL. Ce canal relie le programme C# et la base de données.

L'objet SqlConnection possède plusieurs états, dont deux remarquables : ouvert et fermé. Les autres états sont actifs en régime transitoire ou en cas d'erreur.

Le programme interagit avec une connexion par le biais de la propriété **`ConnectionString`** et des méthodes **`Open()`** et **`Close()`**. L'essentiel des opérations liées à une base ne peut se faire que sur une connexion ouverte.

```
// initialisation de la connexion
string c_string=@"data source=.\SQL2019; initial catalog=
annuaire; integrated security=true";
SqlConnection cx_annuaire;
cx_annuaire=new SqlConnection();
cx_annuaire.ConnectionString=c_string;

// ouverture
cx_annuaire.Open();

// opérations SQL


// Fermeture
cx_annuaire.Close();
```

La chaîne de connexion est formée de différents segments indiquant le nom de la machine abritant la base, le nom de la base, les crédits de l'utilisateur. La syntaxe dépend de chaque format de base. Pour SQL Server, la chaîne de connexion comprend les informations suivantes :

data source	nom de la machine et de l'instance exécutant SQL Server.
initial catalog	nom de la base de données.
integrated security	true ou sspi : la sécurité intégrée est activée. false : la sécurité intégrée n'est pas activée.
user id	nom de l'utilisateur accédant à la base.
password	mot de passe de l'utilisateur accédant à la base.

La documentation MSDN fournit les détails des commutateurs constituant la chaîne de connexion.

Le programmeur se doit d'être particulièrement attentif à la manipulation de la connexion. Une fois ouverte, elle consomme des ressources systèmes et ne peut pas être réouverte avant d'être fermée ; une connexion mal fermée représente ainsi un danger pour l'intégrité et les performances du système.

La syntaxe `try... catch... finally` est alors la seule construction possible pour être certain de fermer une connexion et de libérer des ressources acquises depuis l'ouverture :

```
try
{
    // ouverture
    cx_annuaire.Open();

    // opérations SQL
    // ...
}
catch (Exception err)
{
    Trace.Write(err.Message);
}
finally
{
    try
    {
```

```
        // fermeture
        cx_annuaire.Close();
    }

    catch (Exception err2)
    {
        Trace.Write(err2.Message);
    }
}
```

Authentification et chaîne de connexion

SQL Server dispose de deux modes d'authentification. Le premier consiste en la fourniture, à chaque connexion, d'un couple (utilisateur, mot de passe) vérifié dans une table des utilisateurs. Cette approche très classique peut s'avérer délicate lorsque les informations de connexion sont consignées dans un fichier de configuration textuel ou lorsqu'elles sont transmises très souvent sur le réseau.

Le deuxième mode d'authentification, appelé sécurité intégrée, n'utilise pas de nom d'utilisateur et de mot de passe transmis sur le réseau. Le système Windows authentifie le programme client (dans notre cas, ASP.NET) et transmet le jeton d'authentification à SQL Server. Ce jeton est codé et d'une durée de vie limitée, ce qui augmente la sécurité d'ensemble.

Lorsque la chaîne de connexion comprend le segment `integrated security=true` (ou `=sspi`), la sécurité intégrée est activée. L'utilisateur ASP.NET doit être préalablement autorisé par SQL Server à l'accès à la base cible. Dans les cas où la sécurité intégrée n'est pas activée, doivent figurer dans la chaîne de connexion les segments `user id=xxx` et `password=xxx`.

```
string c_string = @"data source=.\SQL2019; initial catalog=annuaire;
integrated security=false; user id=sa; password=password";
```

1.1.2 La commande

La commande correspond à une instruction SQL exécutée depuis le programme et appliquée à une base désignée par la connexion associée.

```
// ouverture
cx_annuaire.Open();

// opérations SQL
string rq = "select count(*) from utilisateur";
```

```
SqlCommand sql;
sql = new SqlCommand();
sql.CommandText = rq;
sql.CommandType = CommandType.Text; // valeur par défaut
sql.Connection = cx_annuaire; // association connexion

int cu = (int)sql.ExecuteScalar();
Label1.Text = string.Format("Nombre d'utilisateurs : {0}", cu);

// fermeture
cx_annuaire.Close();
```

La propriété **`CommandType`** précise si **`CommandText`** contient une instruction SQL – comme dans cet exemple – ou bien désigne une procédure stockée. **`CommandType.Text`** étant la valeur par défaut, la ligne d'affectation de CommandType n'est pas nécessaire.

L'objet commande expose quatre méthodes pour exécuter des requêtes SQL. Chacune d'elles est indiquée dans une situation précise :

`ExecuteScalar()`	Exécute une instruction SQL ne retournant qu'une seule valeur (agrégat).
`ExecuteReader()`	Exécute une instruction SQL retournant au moins une valeur ou un enregistrement.
`ExecuteNonQuery()`	Exécute une instruction SQL ne retournant pas de valeur. Convient aux requêtes de création, de mise à jour, aux appels de certaines procédures stockées.
`ExecuteXmlReader()`	Méthode spécifique à SQL Server. Exécute une instruction SELECT et retourne un flux XML.

1.1.3 Le DataReader

Le DataReader – ou plutôt le **`SqlDataReader`** – est un curseur se positionnant d'enregistrement en enregistrement. Il est instancié par la méthode `ExecuteReader`, et avance de ligne en ligne par le biais de la méthode `Read`.

Le programme ne peut qu'avancer le curseur jusqu'à ce qu'il atteigne le dernier enregistrement de la sélection ou qu'il soit fermé. Les données indexées par le curseur ne peuvent pas être modifiées par son intermédiaire. En contrepartie de ces contraintes, le DataReader se révèle être la méthode la plus efficace pour lire des enregistrements.

Exécuter une requête SELECT retournant un DataReader

Les requêtes de type SELECT concernant plusieurs enregistrements ou plusieurs colonnes retournent un DataReader. Leur application suit toujours la même logique :

```
// préparer la connexion
string c_string=@"data source=.\SQL2019; database=annuaire;
integrated security=true";
SqlConnection cx_annuaire=new SqlConnection(c_string);

// une requête select donne lieu à un SqlDataReader
string rq="select idu,nom,telephone,ids from utilisateur";
SqlCommand sql=new SqlCommand(rq,cx_annuaire);

// ouvrir la connexion
cx_annuaire.Open();

// exécuter la requête et récupérer le curseur
SqlDataReader reader=sql.ExecuteReader();

// avancer de ligne en ligne
while(reader.Read())
{

}

// toujours fermer le reader après usage
reader.Close();

// fermer la connexion
cx_annuaire.Close();
```

Comme des clauses `where` ou `having` figurant dans la requête `SELECT` sont à même d'éliminer tous les enregistrements, le DataReader renvoyé est toujours positionné avant le premier enregistrement, s'il existe. La boucle de parcours est donc bien un `while` (et non un `do`).

Il est impératif de fermer le DataReader avant d'exécuter une autre requête sur la même connexion. La structure de contrôle `try... catch ... finally` s'avère alors très utile.

Lire les valeurs des colonnes

Positionné sur un enregistrement, le DataReader se comporte comme un dictionnaire dont les entrées sont indexées par des numéros d'ordre dans la requête (première colonne, seconde colonne...) et par des noms de colonnes. Les syntaxes correspondantes ne sont pas facilement interchangeables et le programmeur doit être très rigoureux quant aux transtypages nécessaires.

```
while(reader.Read())
{
    int idu;
    idu = (int)reader[0];         // syntaxe 1
    idu = (int)reader["idu"];     // syntaxe 2
    idu = reader.GetInt32(0);     // syntaxe 3, sans transtypage
    string nom;
    nom = (string)reader[1];      // syntaxe 1
    nom = (string)reader["nom"];  // syntaxe 2
    nom = reader.GetString(1);    // syntaxe 3
}
```

Les trois syntaxes présentées fournissent la même donnée, et globalement avec les mêmes temps de réponse. La deuxième syntaxe, très directe, nécessite quand même un **cast** (transtypage) car l'indexeur de SqlDataReader est typé `object`. Il est très important de distinguer transtypage et conversion. Un transtypage est un mécanisme accordant les types de part et d'autre de l'opérateur d'affectation =. Le compilateur vérifie que le type de la valeur à affecter n'est pas promu sans que le programmeur n'en prenne la responsabilité effective. À l'exécution, le CLR applique une nouvelle vérification et déclenche une exception si le type de la colonne interrogée ne correspond pas au type indiqué dans le programme.

Au contraire, la conversion constitue un changement de type. Pour les types numériques, l'opérateur **( )** est applicable et peut donner lieu à un double transtypage. Pour les types chaînes à convertir en numérique ou en date, les méthodes d'analyse textuelle (parsing) sont exposées par les types cibles (`int.Parse` ou `DateTime.Parse`).

La troisième syntaxe ne donne pas lieu à un transtypage car le DataReader expose des méthodes spécifiques à chaque type. Néanmoins, le framework vérifiera lors de l'exécution que ces méthodes ont été appliquées à bon escient. Il n'est pas possible de lire un nombre double à partir d'une colonne **`varchar`** sans opérer de conversion.

Finalement, le programmeur applique la syntaxe qui lui paraît la plus directe, ceci n'a pas beaucoup de conséquences pour le reste du code.

```
while(reader.Read())
{
    int idu;
    idu = (int)reader["idu"];

    string nom;
    nom = (string)reader["nom"];

    ListBox1.Items.Add(new ListItem( nom, idu.ToString()));
}
```