artragis
zyhou
Blue Shark

Zeste de savoir

Créez votre site web avec ASP.NET

12 août 2019

Table des matières

Vous savez faire des sites web statiques, tel que votre *page perso* ? Et vous souhaitez aller plus loin pour créer votre blog ou votre forum ?

Ou bien peut être avez-vous l'idée de LA plateforme d'échange qui va révolutionner le web ?

Alors ce tutoriel est fait pour vous !

De nos jours, un site doit agir comme une véritable application :

— offrir un contenu *dynamique*, tel que la possibilité de commenter un article, de discuter sur un forum, etc.
— permettre d'interagir rapidement et avec fluidité avec l'utilisateur.

ASP.NET est une puissante technologie regroupant des fonctionnalités multiples qui va nous permettre de créer des applications web de manière simple et structurée.

Ce tutoriel est adressé aux débutants, vous n'avez besoin d'aucune notion à propos d'ASP.NET. Il est vivement conseillé de suivre chaque chapitre en entier et dans l'ordre.

Cependant avant d'utiliser ASP.NET, il faut impérativement connaître les langages HTML et CSS, et maîtriser les notions de base du langage **C#**.

Pour un avant goût de ce que vous permet ASP.NET, voici quelques grands sites qui utilises cette technologie :

- stackoverflow : un site qui reçoit plusieurs *millions* de visites par *jours* (54ème site au monde)
- bing : le célèbre moteur de recherche de Microsoft
- la caisse d'épargne
- microsoft.com
- et bien d'autres encore !

Il est aussi possible de développer des applications pour le web en VB.NET. Nous ne traiterons que le C# dans ce tutoriel.

Première partie

Vue d’ensemble

Vous avez décidé de vous lancer dans le développement web avec ASP.NET ? Félicitations, c'est un très bon choix ! Pourquoi ? Vous aurez l'occasion de le découvrir au travers ce tutoriel et notamment dans ce premier chapitre.

ASP.NET est un ensemble d'outils à disposition du développeur (vous). Voyez cela comme une grosse boîte contenant divers outils prêts à l'emploi pour permettre de développer simplement et sans réinventer la roue une application web.

Dans ce tutoriel, nous apprendrons à utiliser certains outils de cette boîte. Pas tous, car il y en a vraiment, *vraiment* beaucoup. Ce chapitre va introduire ASP.NET, de la définition de son architecture à l'installation des composants pour commencer à programmer.

Allez, c'est parti !

1. ASP.NET : le framework web de Microsoft

Lorsque vous visitez un site web, vous avez en général la possibilité de lire ou d'écrire du contenu, de créer un compte, d'envoyer des messages et d'administrer votre compte. Tout cela sans que vous ayez à vous soucier de comment ça fonctionne derrière.

Jusque là, vous avez appris le HTML et le CSS qui permettent respectivement de décrire et de mettre en forme l'interface visiteur de votre site. Alors comment fonctionne ASP.NET dans tout cela ? Quel rôle joue-t-il ?

Ce premier chapitre est là pour répondre à ces questions.

1.1. Qu'est-ce qu'un framework ?

Jusqu'à maintenant, nous avons décrit la technologie ASP.NET comme un **ensemble d'outils**. Il n'y a rien de faux là-dedans, mais en informatique nous préférons employer le terme **framework**. Un framework est très utilisé dans le domaine de la programmation et nous allons voir pourquoi.

Qu'est-ce que ça va m'apporter d'utiliser un framework ?

Tout d'abord, décortiquons le mot : *framework* vient de l'anglais et se traduit littéralement par « **cadre de travail** ». Pour simplifier la chose, nous pouvons dire qu'un framework est une grosse boîte à fonctionnalités et de règles de codage qui va nous permettre de réaliser nos applications informatiques.
Il est entre autres destiné au développement d'une application web, ce qui nous intéresse.

Lors du développement d'une application (web ou autre) le développeur utilise un langage de programmation, prenons par exemple le **C#**. Afin d'être efficace et d'éviter de se casser la tête à sans cesse repartir de zéro, le langage de programmation est accompagné d'un framework.

Celui-ci contient des espaces de noms, des classes, des méthodes, des structures et d'autres outils que le développeur peut utiliser en relation ici avec le C#.
Il s'occupe de la forme, et permet au développeur de se concentrer sur le fond.

La taille d'un framework comme ASP.NET est telle qu'il est impossible pour nous de tout connaître, c'est pour cette raison qu'il existe une documentation associée afin de s'y retrouver.

Utilisez la documentation le plus souvent possible ! Nous ne vous le rappellerons jamais assez.

De plus ASP.NET et C# possèdent une documentation commune extrêmement bien structurée, que vous découvrirez tout au long de ce tutoriel.

Cette documentation s'appelle la **MSDN Library**. Elle est accessible en **français** via ce lien : MSDN Library . Cet acronyme se traduira par "Réseau des Développeurs Microsoft" car il concentre l'essentiel des technologies de développement de l'entreprise.

1.1.0.1. Framework et développement

Après avoir introduit la notion de framework, nous allons exposer les avantages de l'utilisation d'un framework au sein d'une application web.

Nous pouvons commencer par énumérer la **rapidité** : le framework est un ensemble de briques qui nous évite de réinventer la roue. En effet, ces briques sont développées par des équipes de développeurs à plein temps, elles sont donc très flexibles et très robustes. Vous économisez ainsi des heures de développement !

Par exemple, un framework peut proposer des composants graphiques avec des propriétés toutes prêtes, sans que vous ayez à tout refaire vous même.

Passons maintenant à l'**organisation** du développement de l'application : utiliser un framework demande de respecter l'architecture de celui-ci. Une bonne utilisation de la documentation et le respect des règles du code permettent de créer une application maintenable par la suite.

D'autre part, le framework ASP.NET nous permet de nous baser sur plusieurs *architectures* différentes. Ce cours traitera de **MVC** ; nous verrons cela bientôt, mais sachez pour le moment que c'est une façon d'organiser son code pour séparer la partie interface graphique (HTML) de la partie logique (C#).

Utiliser un framework facilite le **travail en équipe** : du fait des conventions et des pratiques de code. La connaissance d'un framework vous permettra de vous intégrer rapidement dans des projets l'utilisant sans avoir besoin de formation.

Et enfin un point très important : l'esprit **communautaire**. Un framework se base généralement sur une grosse communauté de développeurs. Elle fournit toutes les ressources nécessaires afin d'éviter de se perdre lors du développement d'une application.

Citons comme exemple la documentation (MSDN) ou encore ce tutoriel sur l'ASP.NET. De plus, nous n'avons pas à nous soucier des bugs internes au framework, nous ne sommes qu'*utilisateurs* de celui-ci. Les développeurs travaillant sur le framework sont suffisamment chevronnées pour régler les problèmes rapidement, il y a donc un sentiment de **sécurité** supplémentaire.

Et pour renforcer l'esprit communautaire, toute la pile technologique autour de ASP.NET MVC est open source !

Résumons les avantages d'un framework :

- rapidité : nous utilisons des briques toutes prêtes ;
- organisation du code : production d'un code structuré et maintenable ;
- connaissances communes : le travail est basé sur une seule et même boîte à fonctionnalités ;
- communauté : en général, l'aide et les ressources de qualité ne manquent pas ;

— maintenance : en tant qu'utilisateurs, nous ne nous occupons que du fond de notre application, la forme est gérée par le framework lui-même ;
— sécurité : les développeurs de la communauté s'occuperont en principe de corriger les bugs, et vous n'aurez qu'à suivre les mises à jour.

? Il n'y a donc pas d'inconvénients dans tout cela ?

À vrai dire, nous n'en voyons pas mis à part le fait que pour utiliser un framework, il y a une courbe d'apprentissage plus ou moins raide, et donc un temps d'apprentissage plus ou moins long. Utiliser un framework s'apprend en plus de la programmation.

Mais à choisir, mieux vaut perdre un peu de temps pour apprendre à utiliser un framework afin de gagner du temps de développement et de maintenance pour vos applications web par la suite.

1.1.0.2. Un exemple concret : le framework .NET

Pour suivre ce tutoriel, nous vous avons conseillé d'avoir quelques notions de **C#** (ou de VB.NET). Ces deux langages reposent sur un framework : le **framework .NET** (prononcé *dot net*). Vous l'avez quasiment tout le temps utilisé, peut-être sans vous en rendre compte.

Pour illustrer cela, regardez bien ces deux bouts de code :

```
using System;

class Program
{
    static void Main(String[] args)
    {
        String message = "Bonjour";
        Console.WriteLine(message);
    }
}
```

```
Imports System;

Public Module Coucou
    Public Sub Main()
        Dim message As String = "Bonjour"
        Console.WriteLine(message);
    End Sub
End Module
```

Ces deux codes réalisent la même chose : afficher un message à l'écran. Malgré la différence entre les deux langages, nous retrouvons des éléments communs :

— utilisation de la même bibliothèque `System` ;
— la classe `Console` ;
— la méthode `WriteLine` qui permet d'écrire à l'écran.

Eh bien en réalité, dans les deux cas, **nous ne faisons qu'utiliser des éléments appartenant au framework .NET.**

C'est tout ce qu'il y a à savoir pour l'instant, nous reviendrons un peu plus en détail sur le .NET dans les chapitres suivants. Pour l'heure, il est important de retenir que c'est grâce au langage de programmation C# mêlés aux composants du framework .NET et bien sûr à **ASP.NET** que nous allons pouvoir développer des applications pour le web.

1.2. Le framework web ASP.NET

Passons à la partie qui nous intéresse le plus : **ASP.NET**. ASP.NET est un ensemble de technologies tournées vers le web créé par Microsoft. Il est utilisé pour dynamiser les sites web.

> ASP.NET est principalement supporté par **Windows**. C'est pourquoi la suite du tutoriel se basera sur un environnement de développement Windows. Néanmoins, grâce à mono , il est possible d'héberger le site sur un serveur Linux.

Avant de lire la suite, il faut savoir quelques petites choses. Lorsque vous surfez sur le web, il y a deux acteurs :

— le **client** : c'est vous. En lisant ce tutoriel, vous êtes client donc visiteur de la page web ;
— le **serveur** : ce sont des ordinateurs puissants qui stockent et délivrent des pages web aux internautes, c'est-à-dire aux clients.

Un client

Un serveur

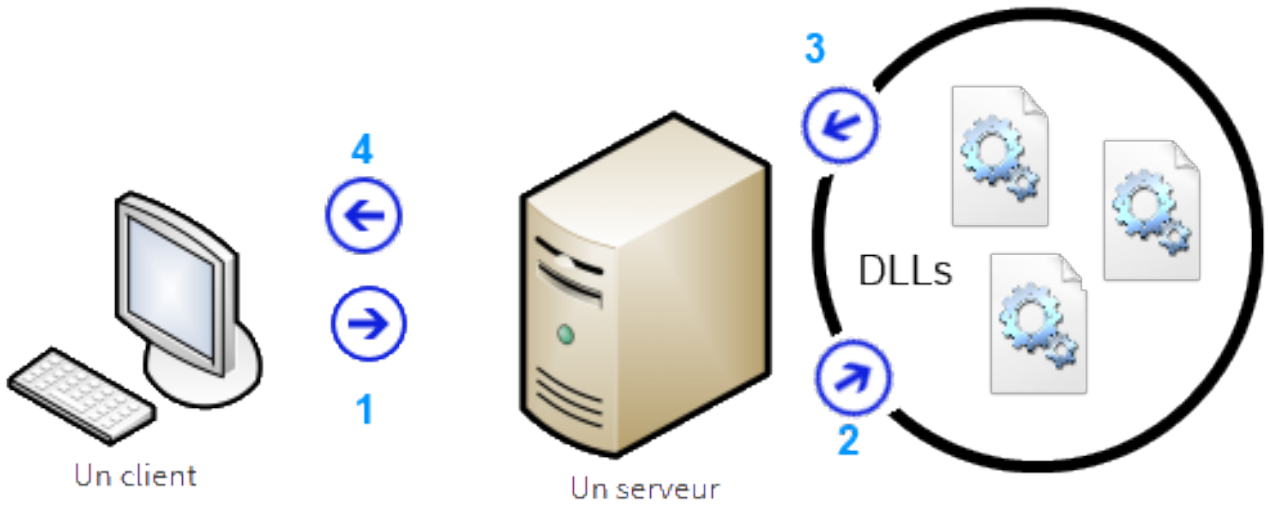
Passons à la suite. Le nom du framework est divisé en deux parties **ASP** et **.NET**. Analysons cela :

La première partie (ASP) est un acronyme de **A**ctive **S**erver **P**ages et symbolise le fait que le framework est un fournisseur de services applicatifs. Nous voyons dans l'acronyme le fait que pour dynamiser les pages HTML d'un site Web, il faut qu'il y ait du code logique côté serveur.

Attention à ne pas confondre ASP.NET et ASP : ce sont deux technologies différentes provenant du même éditeur. ASP est une technologie beaucoup plus vieille.

Et enfin la seconde partie signifie qu'il est basé sur **.NET**, la technologie phare de Microsoft. ASP.NET inclut la génération du code HTML côté serveur, l'utilisation de la programmation orientée objet et des bibliothèques du framework .NET.

À noter que ASP.NET offre de bonnes performances du fait que votre programme est *compilé*. Lorsque vous envoyez votre code sur le serveur, celui-ci reçoit un fichier *exécutable*, que le compilateur a déjà optimisé. Cela permet d'avoir une exécution accélérée, en comparaison à des langages comme Python ou PHP.



Lorsque le client va demander une page web, le moteur d'ASP.NET va décider quelle est l'**action** à effectuer, l'exécuter et générer la page HTML à partir des données traitées par l'action.

ASP.NET est un framework qui vous offrira énormément de possibilités pour créer des applications web. Ci-dessous un schéma montrant les différentes couches de ASP.NET.

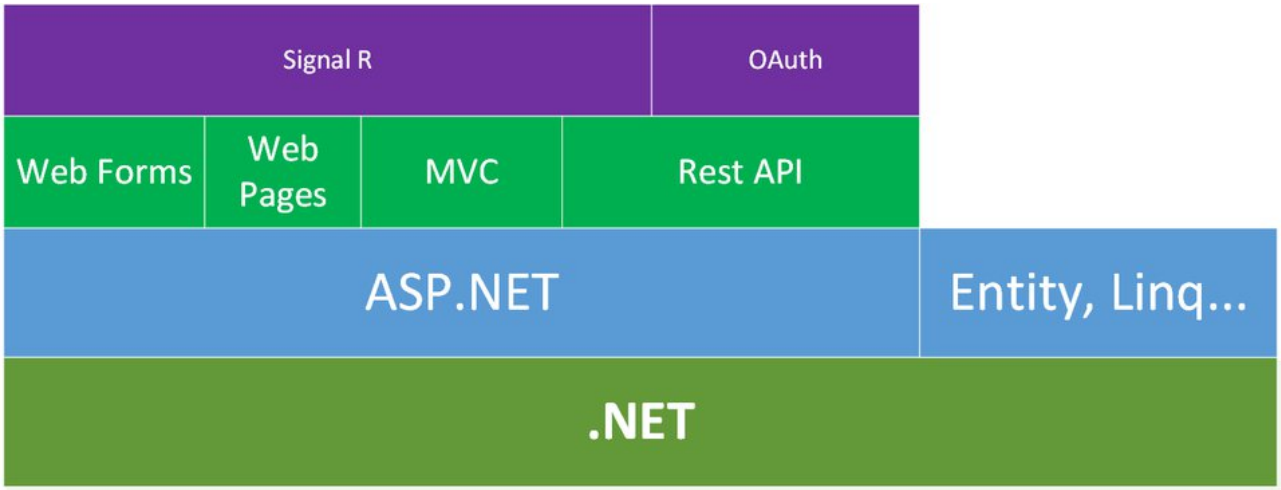


FIGURE 1.1. – Les possibilités d'ASP.NET

Le gros avantage d'ASP.NET, c'est qu'il se repose sur le framework .NET, ce qui lui donne accès à tout ce que .NET a de mieux à vous offrir. Citons par exemple l'accès aux données, les tests unitaires, linq to SQL, etc.

Nous découvrirons dans les chapitres suivants les deux dernières couches qui composent ASP.NET, à savoir ce qui concerne les Web Forms, les Web Pages et MVC.

1.3. Fonctionnement d'un site web

Lorsque vous avez appris à utiliser HTML et CSS, vous avez appris à créer des pages que le navigateur sait comprendre. Le problème, c'est qu'elles sont fixes et c'est pour cette raison que vous êtes en train de lire ce tutoriel.

Prenons un page web de base codée en HTML : son contenu n'évolue que si vous touchez **vous-même** au code source. Faire un blog, un site d'actualités, un comparateur de prix, etc., risque d'être très difficile uniquement en HTML. Sans compter qu'il faudra le mettre constamment à jour et le maintenir.

Ainsi nous pouvons différencier deux types de sites web : les **sites statiques** et les **sites dynamiques**. Les sites statiques sont donc bien adaptés pour réaliser des sites dit "vitrine", pour présenter par exemple son entreprise ou son CV, mais sans aller plus loin. Ce type de site se fait de plus en plus rare aujourd'hui, car dès que l'on rajoute un élément d'interaction (comme un formulaire de contact), on ne parle plus de site statique mais de site dynamique.

À l'inverse de cela, un site dynamique propose une interaction avec le visiteur.
Zeste de Savoir est un exemple de site dynamique. Le forum, par exemple change de contenu tous les jours de manière automatique, à chaque fois qu'un membre y écrit quelque chose.

Créer un site dynamique est un peu plus complexe dans le sens où il devient nécessaire d'utiliser des technologies différentes du tandem HTML/CSS et qui, quant à elles, s'exécuteront sur ce qu'on appelle un serveur. ASP.NET est bien entendu une de ces technologies.

Lorsque les utilisateurs viennent sur un site codé avec ASP.NET, ils envoient une requête au **serveur**.

Ce dernier, comprend la requête, fait des calculs, va chercher des données dans la base de données ou dans d'autres services (ex : Wikipédia, Twitter) et **crée** une page HTML.

Cette page HTML est alors envoyée au client, qui peut la lire. Le client ne verra jamais le code qui est exécuté sur le serveur, mais seulement le HTML généré. C'est le code C# qui va demander de générer cette page HTML. Plus précisément, c'est le rôle de ce qu'on appelle les **vues**.

Sur n'importe quel site il est possible pour le visiteur de voir le code HTML de la page. En général, depuis les navigateurs courants cela se fait via le raccourci de touches Ctrl + U.

1.3.0.0.1. Fonctionnement d'un site dynamique Voici un schéma résumant le fonctionnement d'un site web dynamique :

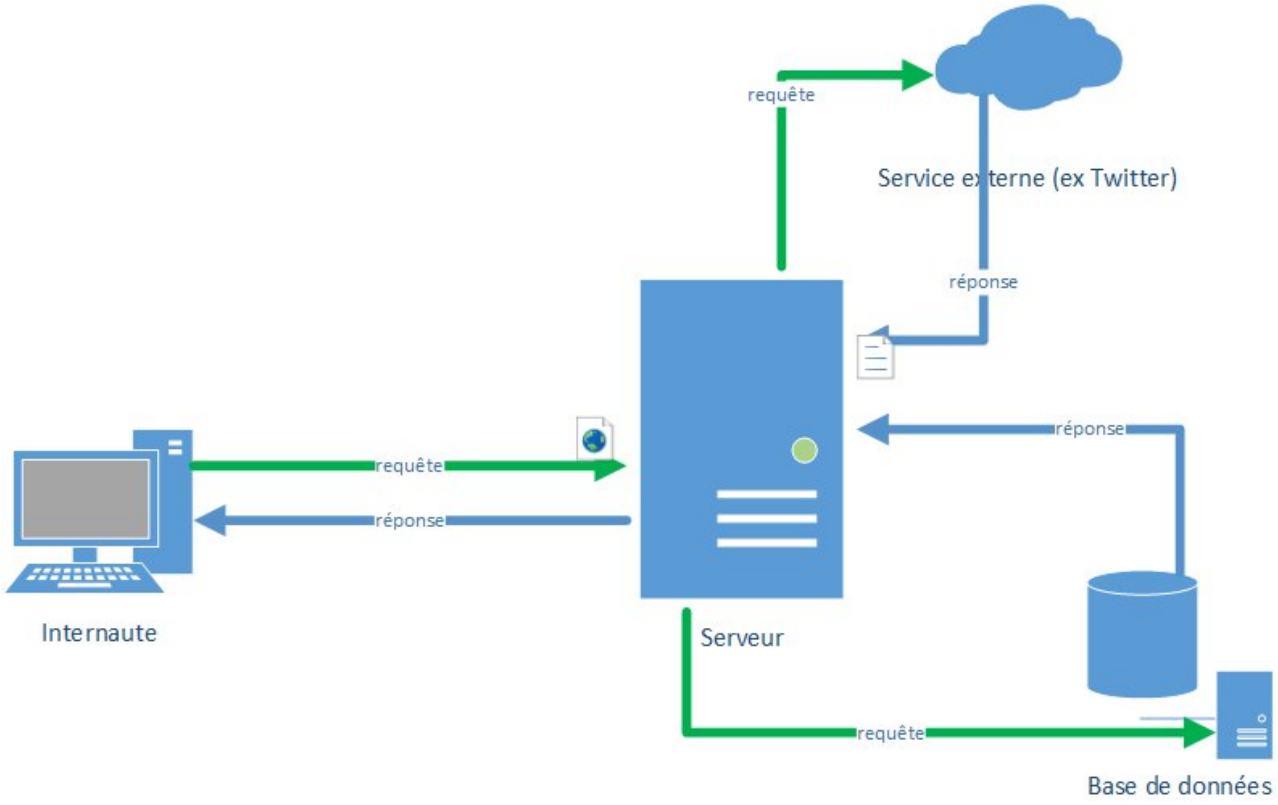


FIGURE 1.2. – L'architecture client/serveur

La page web est générée à chaque fois qu'un client la réclame. C'est précisément ce qui rend les sites dynamiques vivants : le contenu d'une même page peut changer d'un instant à l'autre.

Le **client** envoie des requêtes au serveur. C'est le **serveur**, via ASP.NET, qui fait tout le boulot via également des requêtes :

— chercher des données dans une base de données, par exemple le dernier message que vous avez posté ;

— chercher des données dans des flux externes, par exemple le nombre de *followers* de votre dernier article via Twitter.

Le tout est ensuite renvoyé au serveur, qui lui-même renvoie au client sous forme d'une page web HTML complète.

C'est ainsi que se termine ce premier chapitre. Retenez donc que ASP.NET regroupe plusieurs technologies qui permettent de dynamiser les sites web. J'espère que nous vous avons donné envie de créer votre prochaine application web avec ASP.NET.

Dans le prochain chapitre, nous découvrirons les outils à installer pour commencer à utiliser ASP.NET.

2. Installation des outils

Avant de pouvoir commencer à développer nos applications web, il y a une étape indispensable : **l'installation des bons outils**. La plupart du temps, si vous développez avec ASP.NET, vous développerez sous Windows même s'il est possible d'obtenir des outils de développement C#/ASP.NET pour Linux.

Ainsi, votre ordinateur va jouer le rôle de serveur lorsque vous travaillerez avec ASP.NET. Nous serons fin prêts à programmer après avoir lu ce chapitre !

2.1. L'environnement de développement

Jusqu'à maintenant, vous avez utilisé un simple éditeur de texte pour réaliser vos pages web en HTML et CSS. Dynamiser un site web nécessite quelques logiciels supplémentaires et par la même occasion, plus évolués !

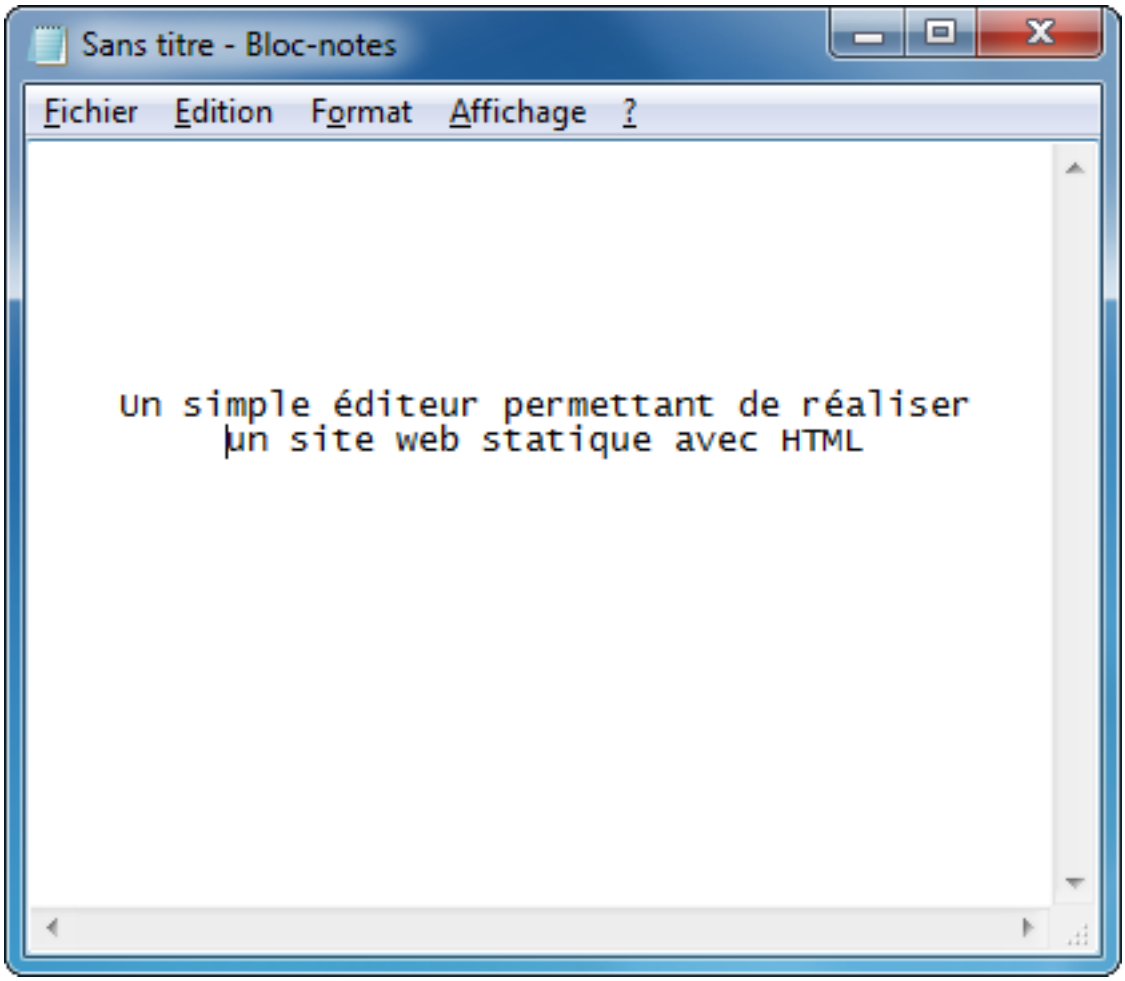


FIGURE 2.1. – Le Bloc-notes de Windows

Il y a un certain nombres d'outils permettant de développer avec ASP.NET et C#. C'est pourquoi, lorsque l'on fait de la programmation, il faut toujours passer par l'étape "choix des outils", des **bons** outils.

Lorsque vous faites ces choix, deux philosophies s'offrent à vous :

— prendre tout un tas de petits logiciels qui font chacun leur travail de manière très optimisée ;
— prendre un gros logiciel qui *intègre* toutes les fonctionnalités de base, mais aussi plus avancées, pour réaliser votre application.

Nous avons choisi la seconde méthode car elle est la plus simple pour un débutant. Nous allons donc utiliser un Environnement Intégré de Développement qui s'abrège, en anglais, IDE.

i

Un IDE comprend tous les outils nécessaires à la conception d'une application web : à savoir un environnement de débogage, un compilateur (C#), un éditeur avec auto-complétion, un gestionnaire de paquets et plein de modules qui peuvent vous faciliter la vie (git, bugtracker, colorpicker...).

2.1.0.0.1. Utiliser un IDE Pour simplifier les choses, nous avons choisi d'utiliser l'environnement de développement par défaut que propose Microsoft, à savoir Visual Studio[1].

Deux solutions s'offrent à vous :

— vous êtes étudiants dans une école partenaire du Microsoft Developper Network Academic Alliance, ou bien êtes titulaire d'une licence de Visual Studio Professional, Premium ou Ultimate. Dans ce cas, téléchargez un ISO du logiciel et installez-le ;
— vous n'avez aucune licence pour une version avancée de Visual Studio : rendez-vous sur cette page et prenez la version la plus récente de **Visual Studio Community edition** (que nous appellerons simplement "Visual Studio" dans la suite du cours, par commodité).

Nous verrons comment installer le logiciel dans la prochaine sous-partie. Visual studio intègre une large gamme d'outils permettant de faciliter la vie du développeur.

Visual Studio comprend de nombreuses fonctionnalités comme :

— un éditeur HTML / CSS qui vous permet de travailler en mode source ou directement en mode design ;
— un puissant débogueur intégré au compilateur C# ;
— un support pour les bibliothèques JavaScript externes, comme jQuery ;
— une coloration syntaxique et de l'auto-complétion (l'éditeur propose des choix au fur et à mesure de vos saisies au clavier) ;
— pouvoir tester directement votre application web avec un serveur web intégré ;
— et d'autres que nous découvrirons tout au long de ce tutoriel...

Dans la suite de ce chapitre, nous allons voir comment installer Visual Studio.

2.2. Installation de Visual Studio

Une fois le programme d'installation téléchargé, il ne reste plus qu'à le lancer et l'installation démarre. Installez Visual Studio en laissant toutes les options par défaut.

Un fois l'installation terminée, lancez Visual Studio, celui-ci va effectuer une première configuration.

1. Une version "en ligne" qui permet de faire du web et d'éditer vos sites de manière ad-hoc. Elle est néanmoins liée à Azure, qui est un hébergement payant.

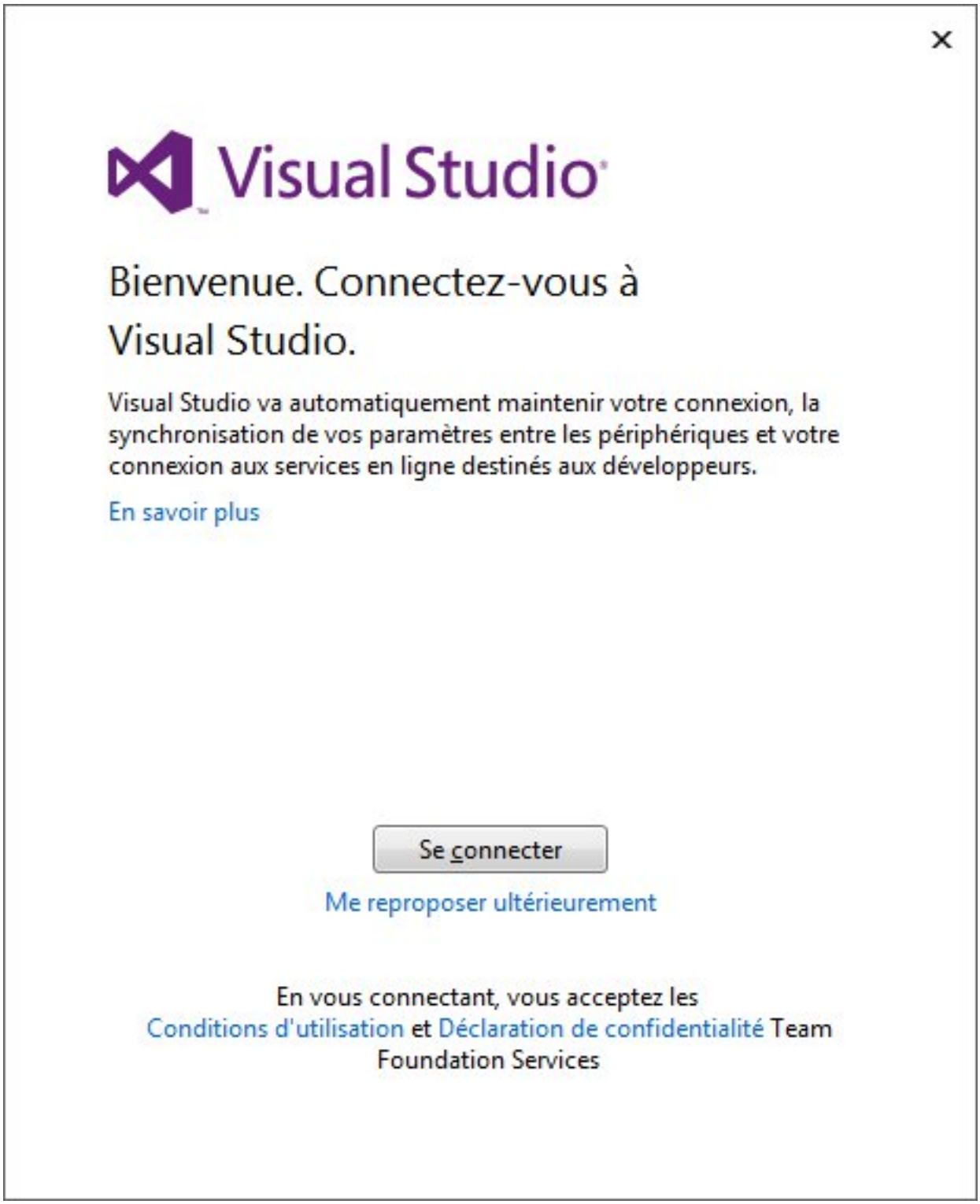


FIGURE 2.2. – Bienvenue dans Visual Studio

Visual Studio finalise nos paramètres, nous sommes prêts à commencer.

Il vous est proposé de vous connecter. Même si cela n'a rien de nécessaire, nous vous conseillons de le faire. Cela vous permettra de récupérer vos paramètres d'utilisation lorsque vous changerez de PC par exemple. Cliquez sur **Se connecter** pour démarrer la configuration utilisateur.

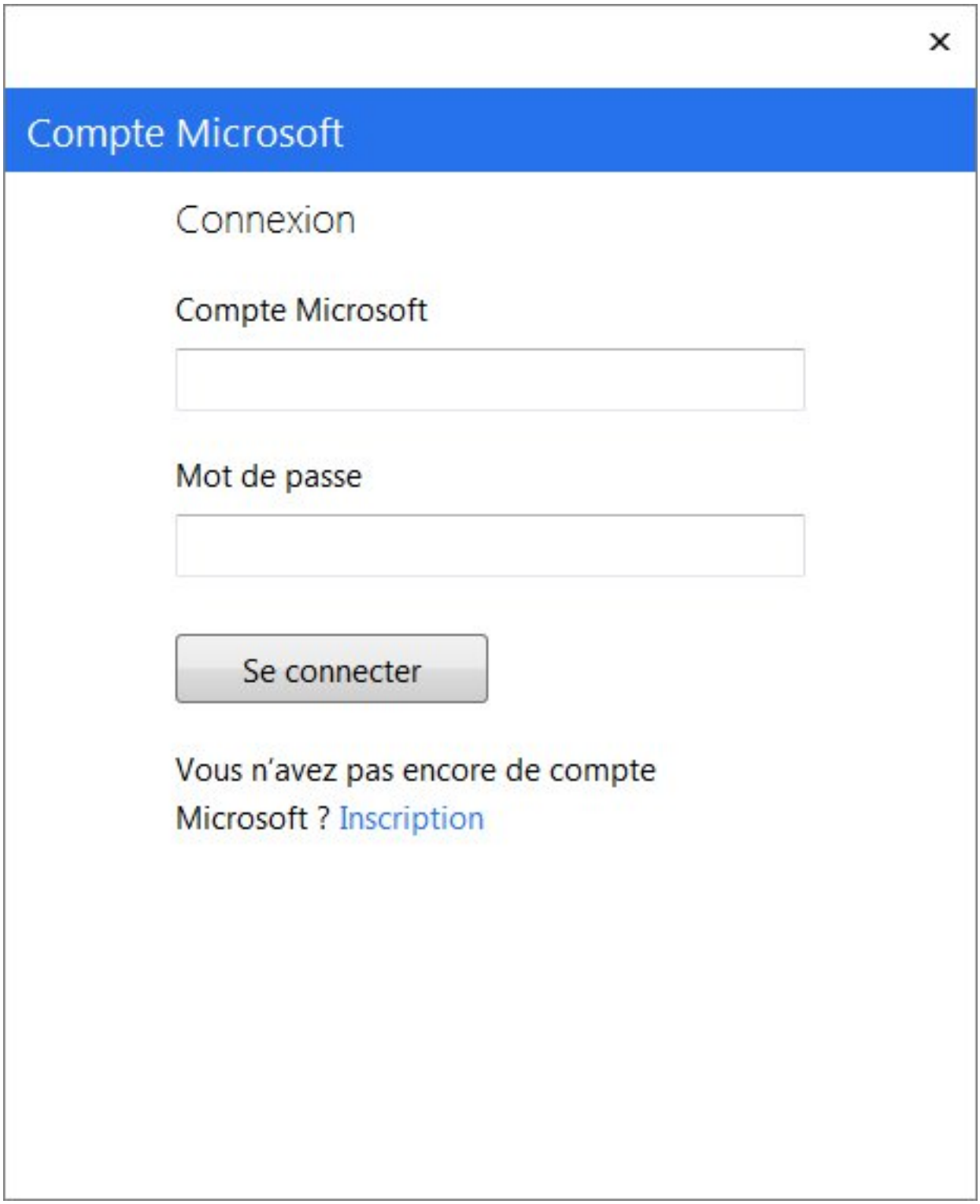


FIGURE 2.3. – Étape du compte Microsoft

Visual Studio nous demande notre adresse de compte Microsoft.

? Je n'ai pas de compte Microsoft !

Si vous n'avez pas d'adresse de messagerie, ce n'est pas un problème : le programme de mise en route vous permet d'en créer une.

Pour cela, cliquez sur **Inscription** pour obtenir une nouvelle adresse de messagerie. Vous devez alors choisir votre nouvelle adresse et entrer quelques informations utiles. Une fois que vous aurez terminé, vous pourrez passer à l'étape suivante.

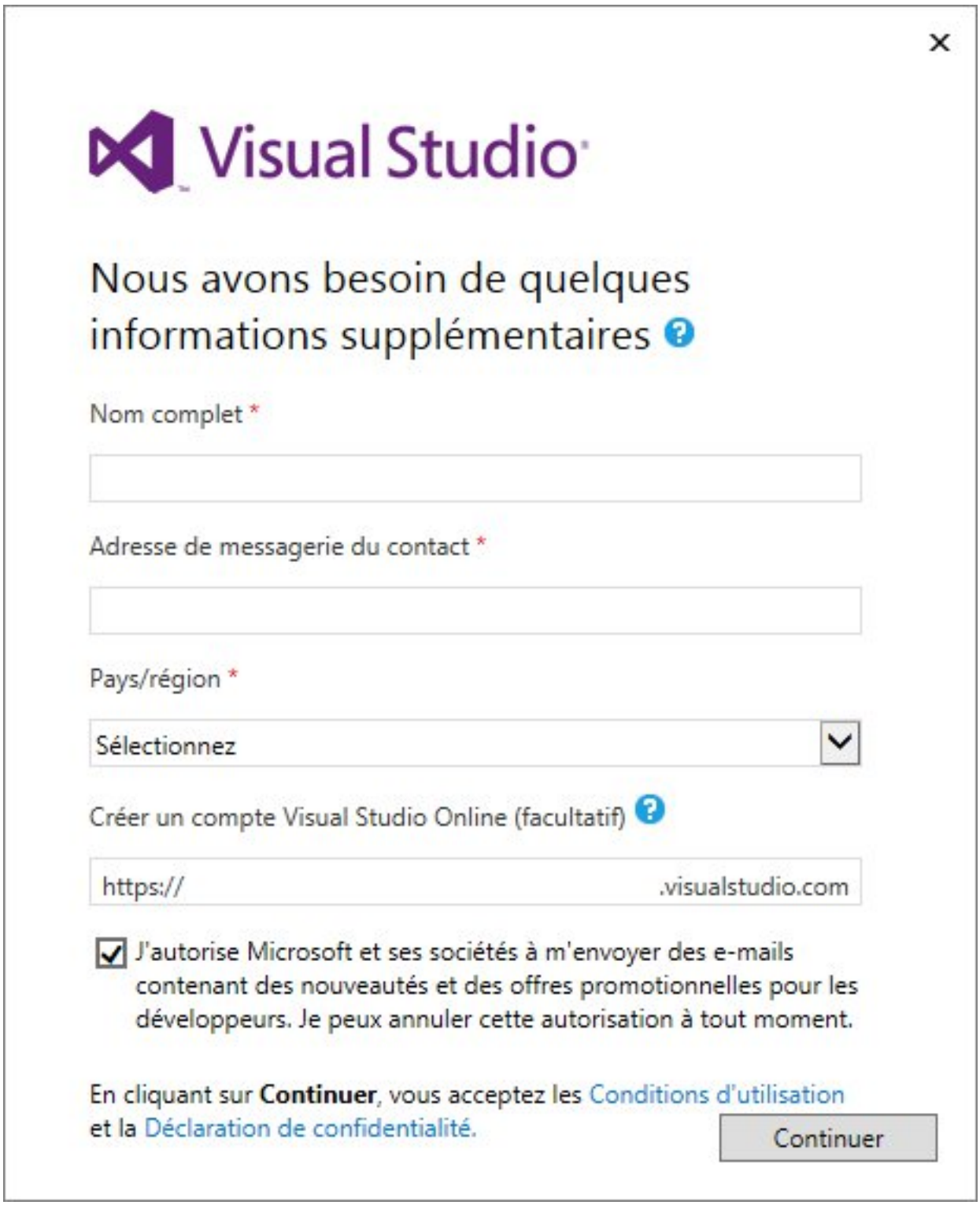


FIGURE 2.4. – Informations supplémentaires

Sur cette page, l'installateur vous propose de rentrer votre nom et une case à cocher. Si vous autorisez Microsoft à récupérer des informations sur votre ordinateur et des statistiques pour ses bases de données, laissez comme tel. Dans le cas contraire, décochez la case. Cliquez ensuite sur le bouton **Continuer**. En cliquant sur ce bouton, vous acceptez les conditions d'utilisation du logiciel.

2.3. Créez votre premier projet

2.3.1. L'interface de Visual Studio

Nous allons vérifier que l'installation de Visual Studio a bien fonctionné. Et pour ce faire, nous allons le démarrer et commencer à prendre en main ce formidable outil de développement.

Il y a une multitudes d'options mais c'est un outil très simple d'utilisation, si vous suivez ce tutoriel pas à pas, vous allez apprendre les fonctionnalités indispensables. Vous avez utilisé Visual C# pour faire vos applications en C# et maintenant vous utilisez Visual Studio Express pour le Web pour faire vos applications web.

Visual C# et Visual Studio Express pour le Web possèdent des fonctionnalités communes. Commencez par démarrer l'IDE. Le logiciel s'ouvre sur la page de démarrage :

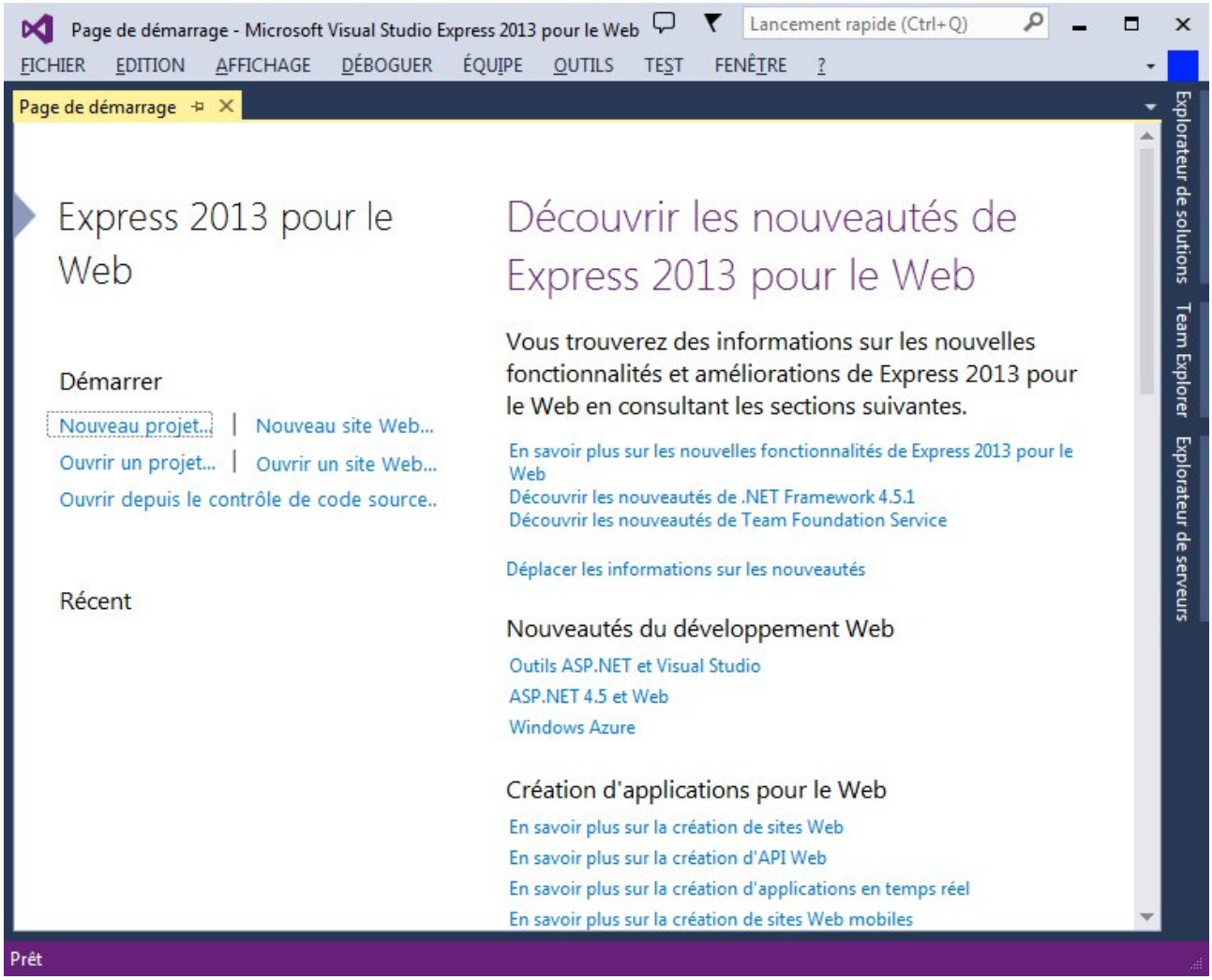


FIGURE 2.5. – Interface de Visual Studio

2.3.2. Nouveau projet

Je vous invite, seulement pour appréhender l'interface, à créer un projet Application Web ASP.NET MVC4 (voir figure suivante). Pour ce faire, trois solutions s'offrent à vous : cliquer sur le bouton **Nouveau projet**, se rendre dans le menu Fichier > Nouveau projet, ou utiliser le raccourci clavier Ctrl + N.

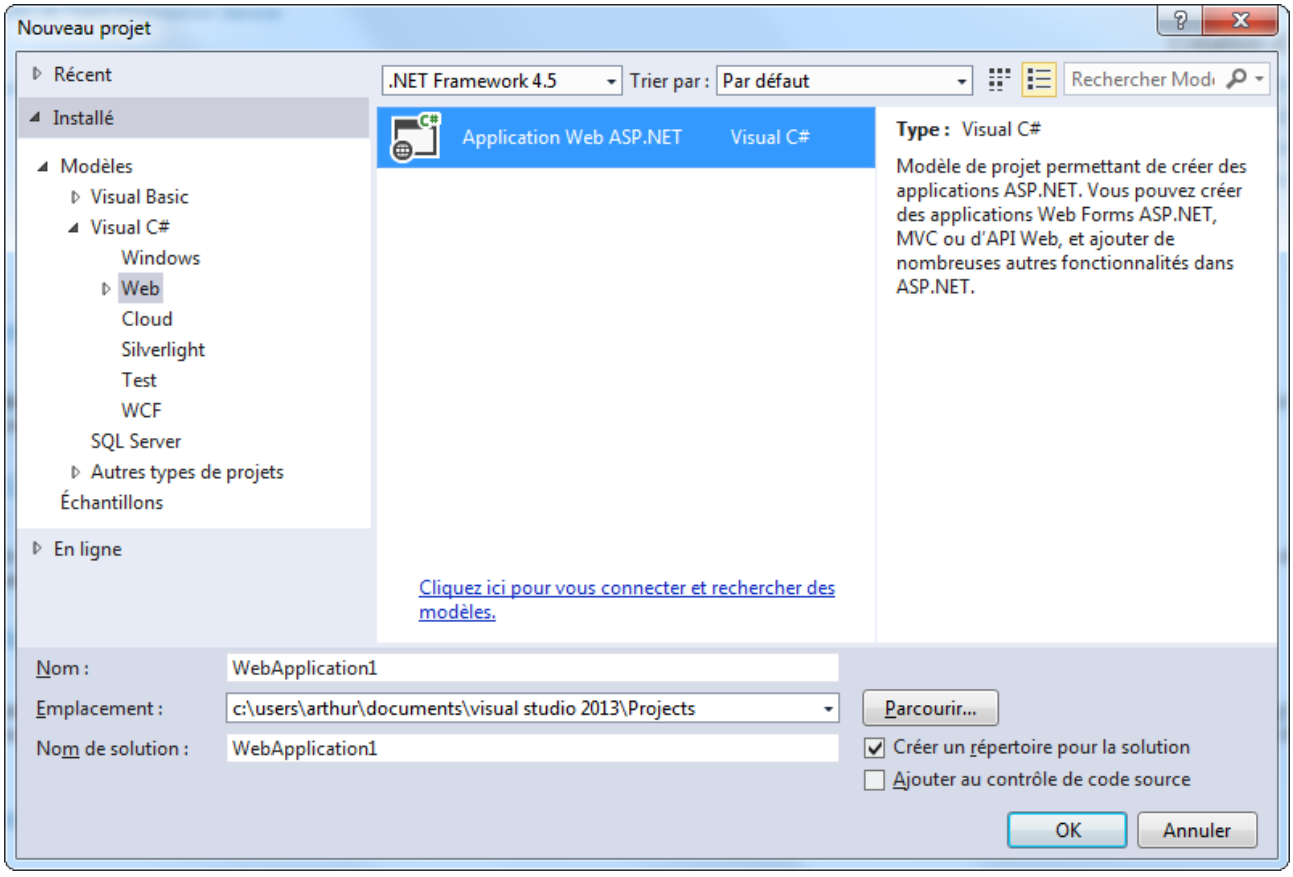


FIGURE 2.6. – Créer un nouveau projet

Allez dans le nœud Web de Visual Studio et sélectionnez Application Web ASP.NET. Tout comme avec Visual C#, vous avez la possibilité de changer le nom du projet et de changer le répertoire de l'application. Cliquons sur OK pour valider la création de notre projet. Vous remarquez que beaucoup plus de choses s'offrent à nous (voir figure suivante).

Dans cette fenêtre, il y a la liste des **modèles** de création de projets installés. Pour commencer, nous vous proposons de créer un modèle de type **application Web ASP.NET MVC**, nous reviendrons plus tard sur le terme MVC, retenez simplement que c'est une manière d'organiser son code.

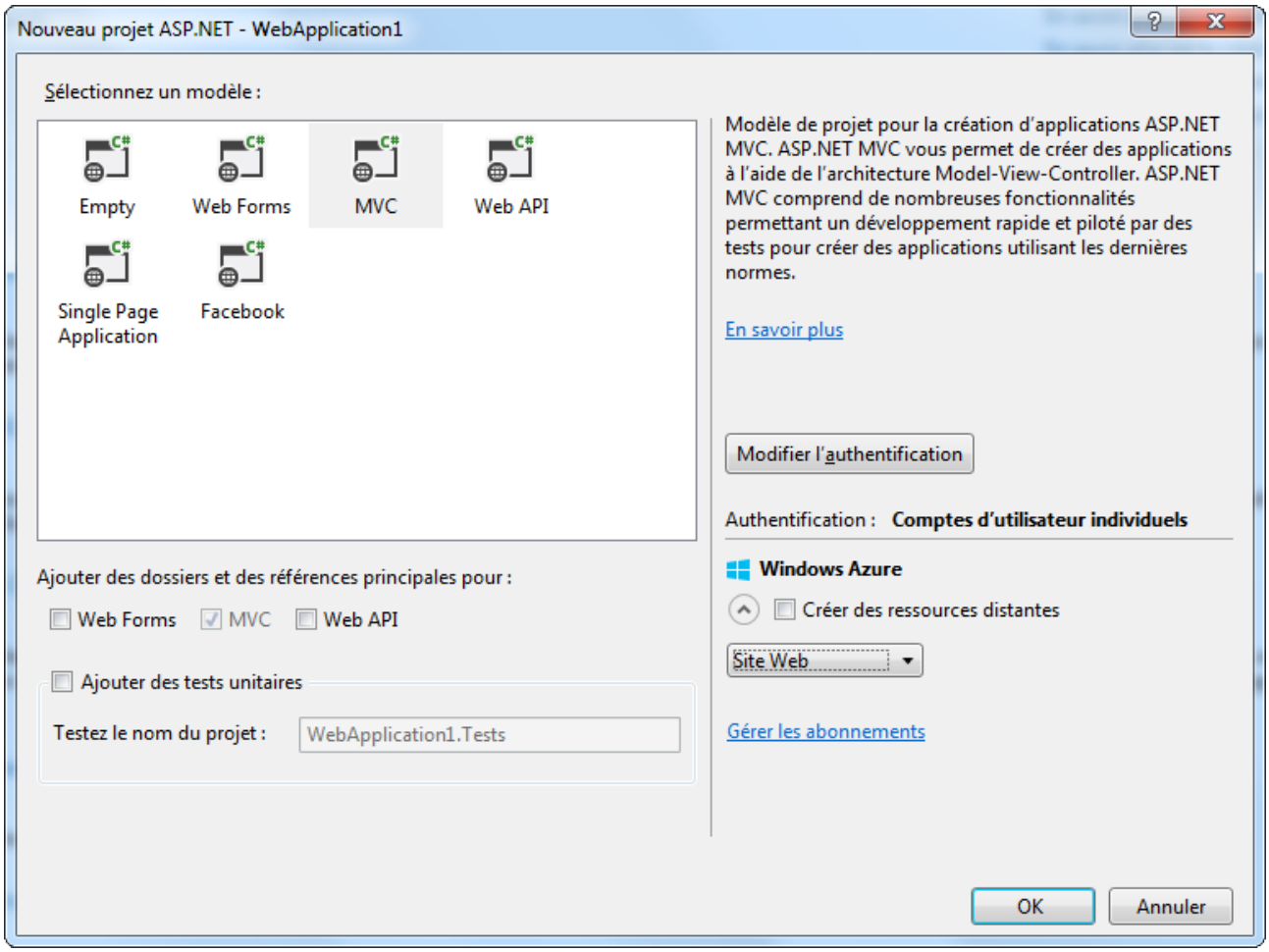


FIGURE 2.7. – Nouveau projet ASP.NET MVC

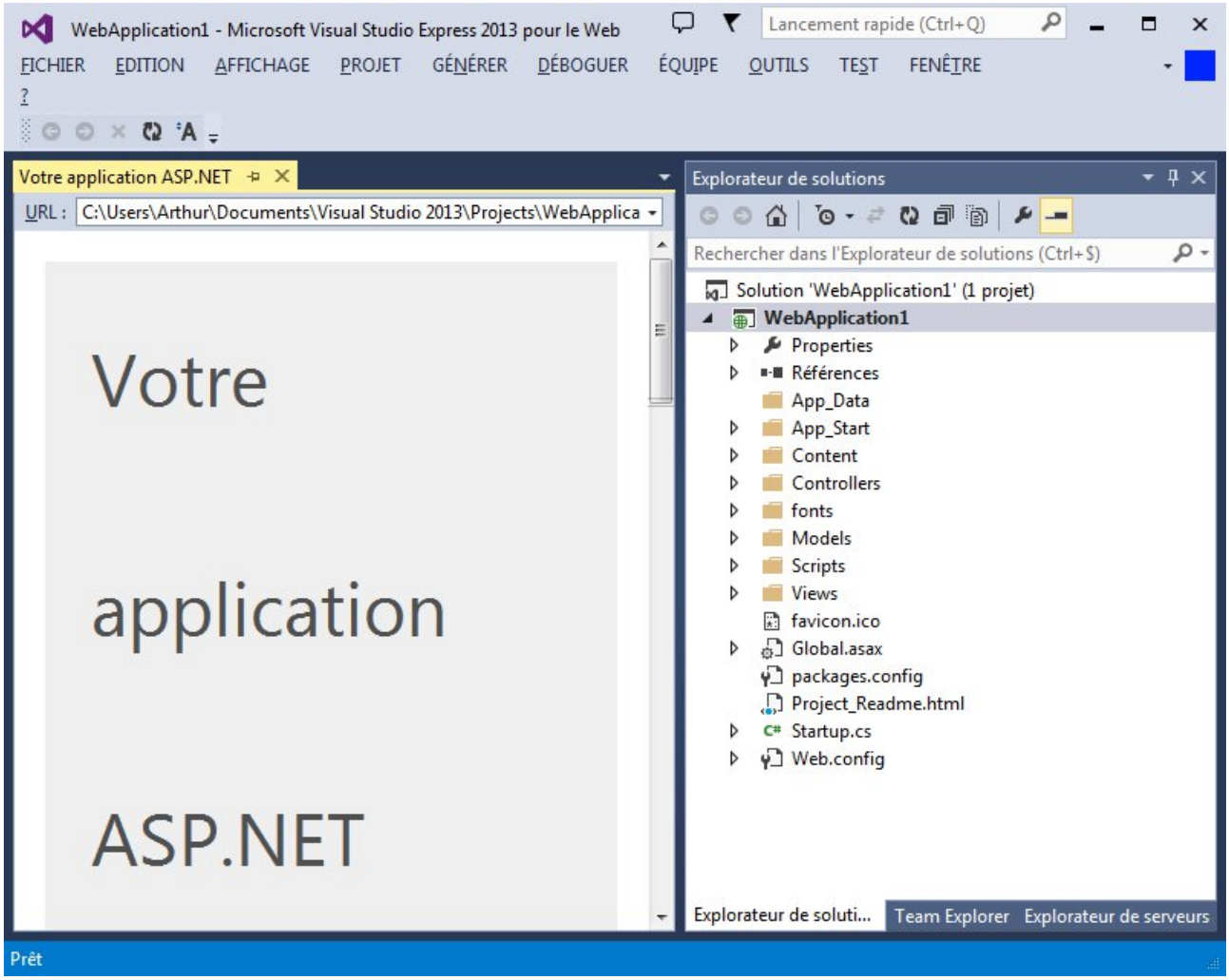
Maintenant, il faut choisir avec quoi nous allons commencer notre application web. ASP.NET propose plusieurs modèles :

- le projet **Vide** : comme son nom l'indique, nous créons un projet à partir de zéro, néanmoins Visual Studio y intégrera les références nécessaires vers les assemblies[2] ASP.NET, quelques bibliothèques JavaScript utiles pour une application web ainsi qu'un fichier de configuration ;
- le projet de **WebForms** : créer un projet ASP.NET Web Forms, basé sur les contrôles et le code évènementiel ;
- le projet **MVC** : un projet tourné vers MVC, ce qui nous intéresse pour commencer ;
- le projet **Web API** : une application Internet pré-configurée avec le contrôleur Web API. Nous aurons l'occasion de revenir dessus ;
- le projet Application avec **une seule page** ;
- le projet Application **Facebook**.

Ça fait beaucoup de modèles ! Alors lequel choisir ? Nous n'allons pas commencer à programmer directement, nous vous proposons donc de créer un projet **MVC** pour découvrir l'interface de Visual Studio.

Sélectionnez MVC et laissez le reste tel quel, décochez la case en dessous **Windows Azure**. Validez avec OK.

Windows Azure est une plateforme d'hébergement Cloud Microsoft. Windows Azure nous permettra de publier une application web, mais ça ne nous intéresse pas à ce stade du tutoriel.



Pas grand chose de différent comparé à Visual C# : nous retrouvons l'explorateur de solution avec toute l'organisation des fichiers et l'éditeur de code au milieu. Nous découvrons dans l'explorateur de solutions l'organisation des fichiers C# dans différents dossiers, ne vous attardez pas là-dessus car nous y reviendrons très vite.

2.4. Analyse de l'environnement de développement

Nous allons éplucher l'interface de Visual Studio pour parler de quelques éléments. Le fait que vous avez déjà touché au langage C# ou VB.NET vous permet de connaître quelques éléments de

2. Une assembly est un programme .NET qui a déjà été compilé. On les trouve souvent sous la forme de fichier `.dll`.

l'interface, comme l'explorateur de solutions, l'éditeur ou la barre d'outils. Regardons maintenant comment va s'organiser Visual Studio pour un projet d'application web.

2.4.0.0.1. L'organisation de Visual Studio Sachez avant tout que Visual Studio permet de faciliter la vie du développeur. Sa capacité de pouvoir se personnaliser facilement en fait de lui un outil très propre au développeur.

Nous allons commencer par parler de l'**explorateur de solutions**. C'est un outil intégré à Visual Studio qui présente les fichiers relatifs aux projets rattachés à la solution courante. Le plus souvent, nous ouvrons nos fichiers par un double clic. Visual Studio utilise des fichiers spécifiques pour gérer les solutions (extensions `.sln` et `.suo` pour les options utilisateurs) et les projets (`.csproj`).

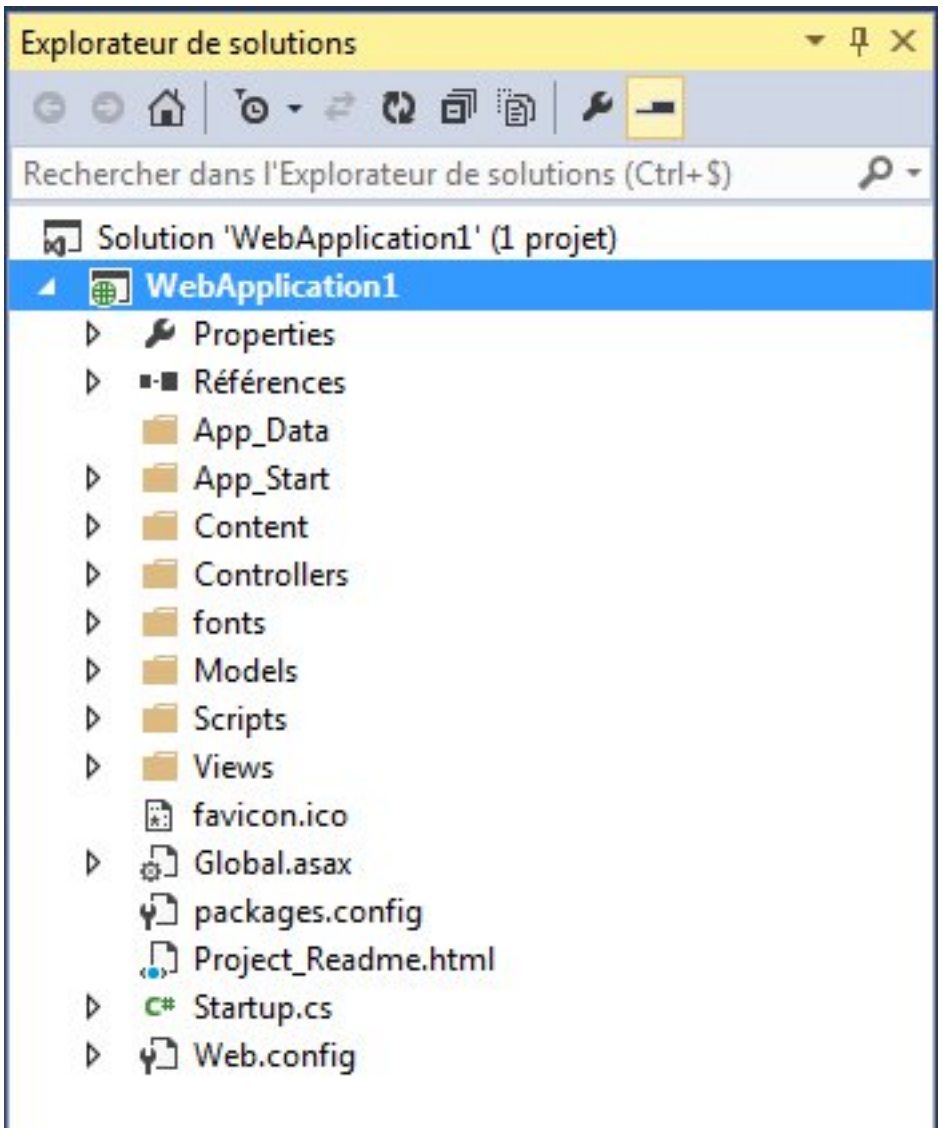


FIGURE 2.8. – L'explorateur de solutions Visual Studio

Un deuxième outil souvent important est la **barre de propriétés** : elle permet d'afficher et d'éditer les propriétés d'un objet sélectionné, comme un fichier de l'explorateur de solutions.

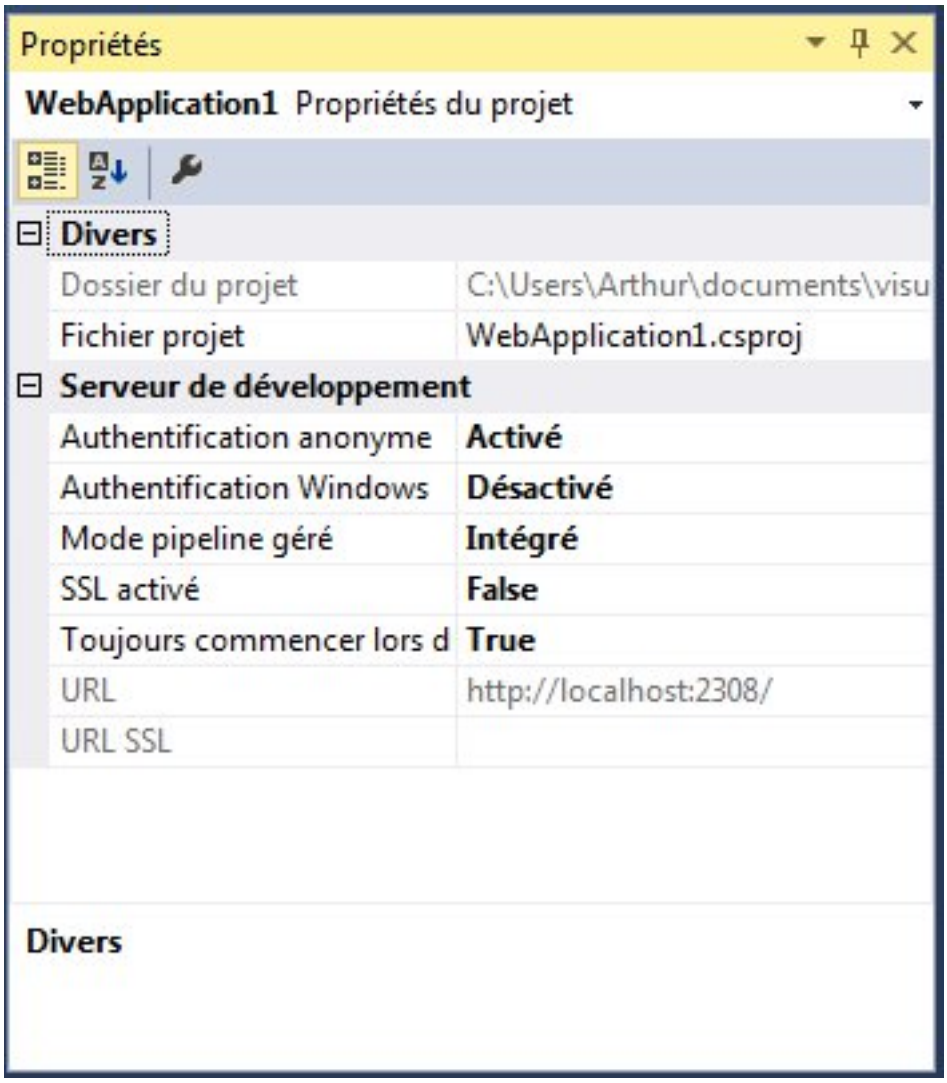


FIGURE 2.9. – Barre de propriétés

Enfin, nous allons parler d'un dernier outil important à nos yeux : l'**explorateur de serveurs**. Le plus souvent, l'explorateur de serveurs est utilisé pour accéder aux bases de données nécessaires au fonctionnement d'une application. Il donne aussi des informations sur l'ordinateur courant telles que les états Crystal report, les logs, les services Windows, etc.

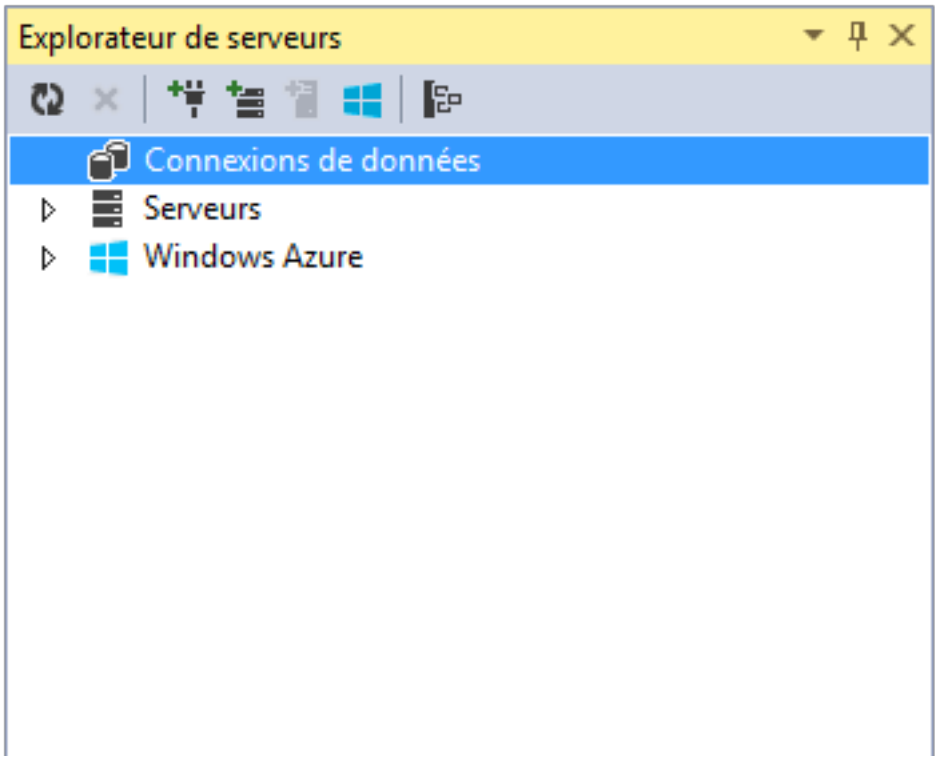


FIGURE 2.10. – Explorateur de serveurs

Si vous ne voyez pas ces fenêtres chez vous, vous pouvez les rendre visibles en allant dans le menu **Affichage** de Visual Studio.

L'élément le plus important reste l'éditeur de code qui offre une coloration de syntaxe, une auto-complétion et un soulignage d'erreurs dans le code.

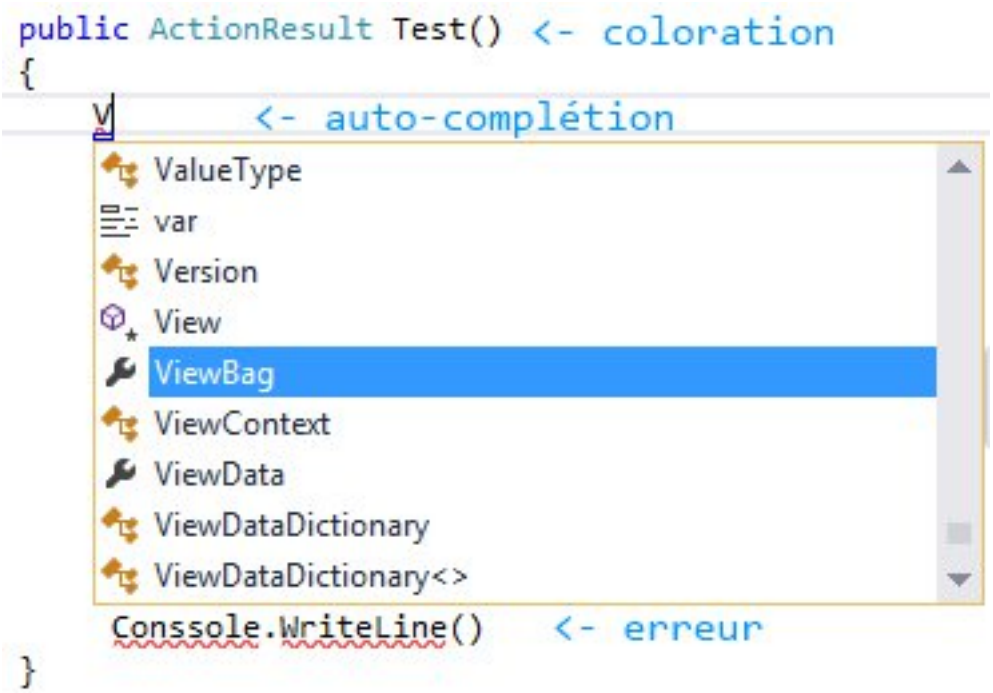


FIGURE 2.11. – Fonctionnalités de l'éditeur

2.4.0.0.2. Obtenir des informations complémentaires en cas d'erreur

Lorsque vous voulez corriger un bug, le plus souvent vous aller utiliser le débogueur pas à pas de Visual Studio.

Néanmoins il peut y avoir des cas - rares, mais existants - où le débogueur ne suffira pas et où vous aurez besoin de plus d'informations.

Pour cela, il faut aller regarder les *journaux* du serveur. Ces-derniers se trouvent dans le dossier `Mes Documents\IISExpress\TraceLogFiles` et `Mes Documents\IISExpress\logs`.
Leur allure vous rebutera peut être, mais ils vous permettront de voir, tel que le serveur le fait, comment votre site s'est exécuté.

Si vous avez un jour un problème, que vous posez votre question sur le forum, et que quelqu'un vous demande de lui fournir les logs, c'est là qu'il faudra aller les chercher !

2.5. Exécutez l'application web

Maintenant, il est temps de voir le rendu de l'application web sur navigateur. Il ne suffit pas de cliquer sur un fichier HTML comme vous avez l'habitude de le faire, il faut avant tout **générer** le projet. Cette phase permet de compiler le code côté serveur, le C#.

Allez dans le menu Déboguer et cliquez sur Démarrer le débogage, ou appuyez directement sur la touche F5 de votre clavier.

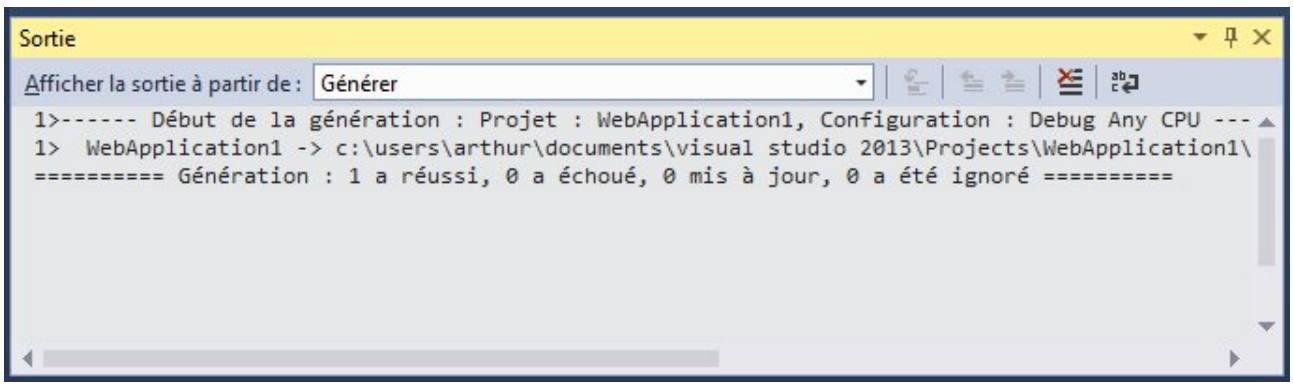


FIGURE 2.12. – Fenêtre de sortie

Remarquez qu'un fichier `.dll` est généré en sortie.

Ensuite, Visual Studio va démarrer **IIS Express**, qui est une version allégé de IIS. C'est le serveur web permettant d'exécuter une application ASP.NET. Visual Studio démarre ensuite le navigateur web et vous dirige automatiquement vers la page d'accueil.

La page web que vous voyez à l'écran vous a été envoyée par votre propre serveur IIS Express, que vous avez installé en même temps que Visual Studio. Vous êtes en train de simuler le

fonctionnement d'un serveur web sur votre propre machine. Pour le moment, vous êtes le seul internaute à pouvoir y accéder. On dit que l'on travaille **en local**. Notons que l'URL affichée par le navigateur dans la barre d'adresse est de la forme **http ://localhost :#numero de port/**, ce qui signifie que vous naviguez sur un site web hébergé sur votre propre ordinateur.

Lorsque Visual Studio exécute une application web, il génère un numéro de port permettant de faire fonctionner l'application.

Nous pouvons ensuite nous balader sur les différents liens qu'offre l'application toute prête. Notons qu'en fonction de la taille de votre fenêtre, les éléments sont agencés différemment.

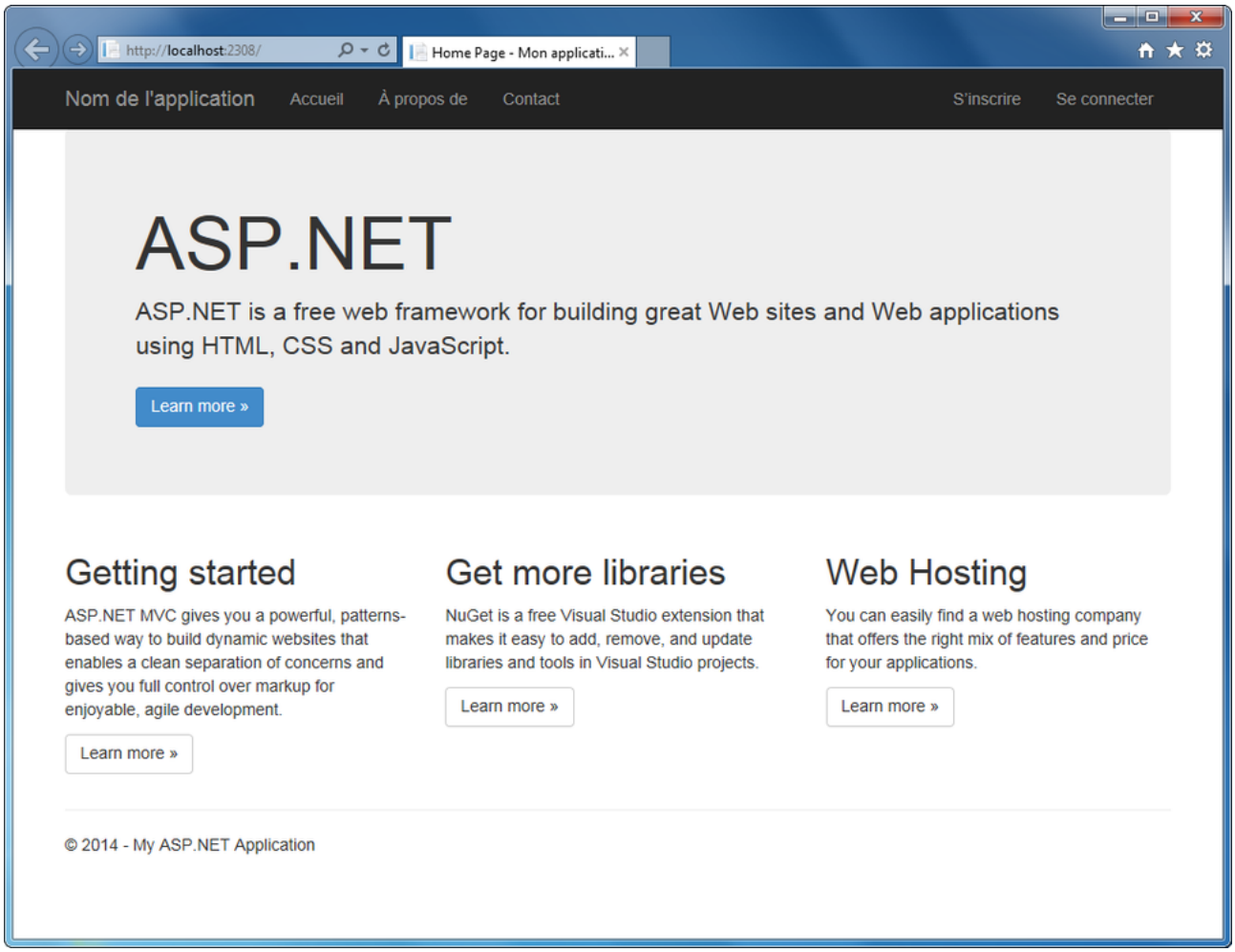


FIGURE 2.13. – rendu sur navigateur grand format

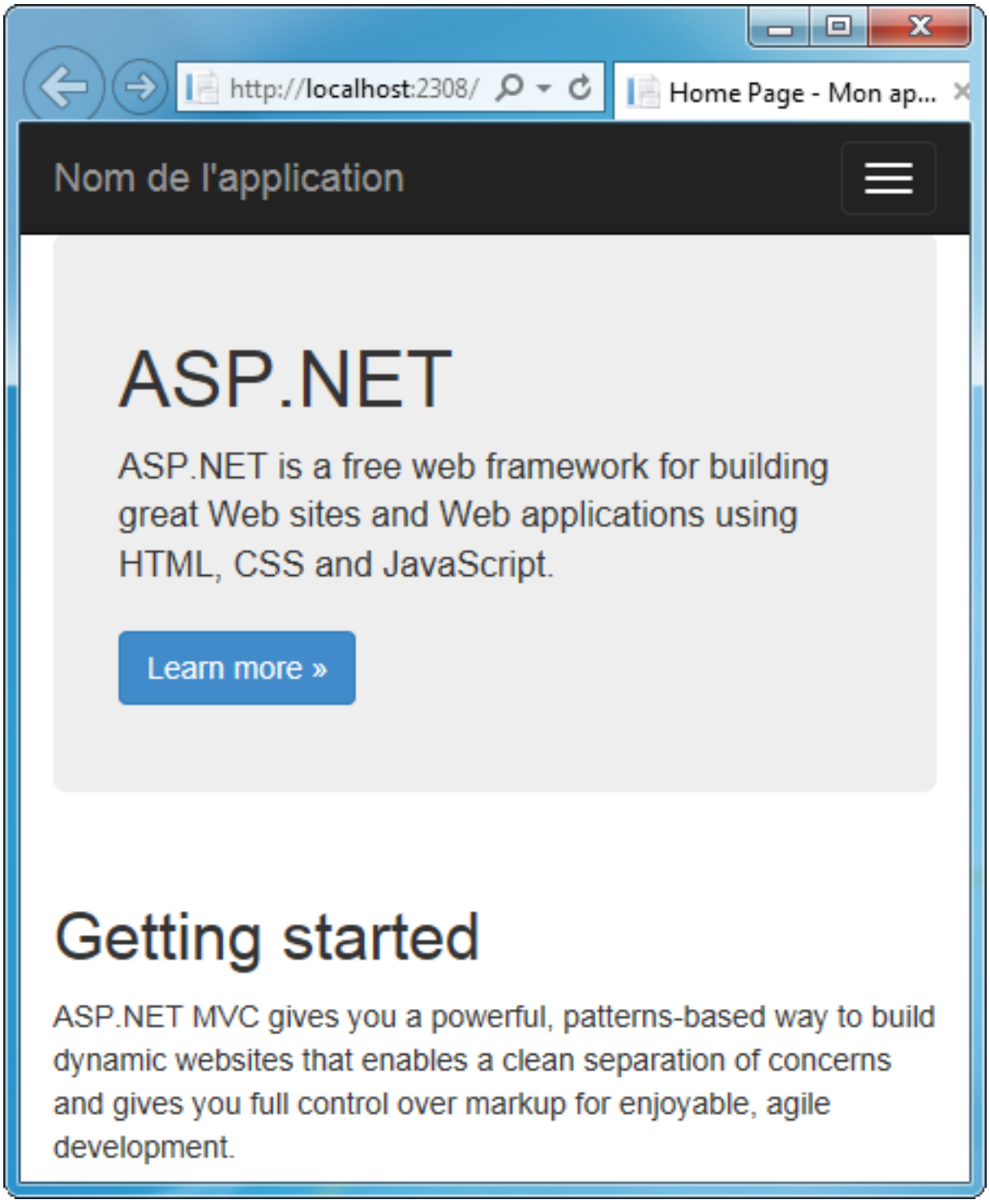


FIGURE 2.14. – rendu sur navigateur petit format

Ainsi le menu se retrouve soit directement sur la page de l'application, soit dans un menu déroulant. L'application propose aussi un support de connexion et d'enregistrement implémenté. Lorsque nous créerons notre application, nous partirons de zéro. Ce projet tout fait vous permet d'avoir un avant-goût de ce que nous pouvons faire avec ASP.NET.

Nous avons maintenant tout ce qu'il nous faut pour commencer à programmer. Il reste cependant

un point à éclaircir : les différents types de technologies que nous pouvons utiliser avec ASP.NET afin de créer une application web. La suite au prochain chapitre !

3. Les différents types d'applications

ASP.NET est un framework très complet et très souple.

Sa force, c'est son intégration au monde Windows, ce qui lui permet d'accéder à des fonctionnalités telles qu'Active Directory très facilement.

Par souci d'être en phase avec les technologies les plus modernes du web tout en vous permettant de développer **rapidement** vos applications, Visual Studio intègre plusieurs modèles de projets que nous avons brièvement découverts lors du chapitre précédent.

Nous employons le mot "modèle" pour décrire un projet créé avec Visual Studio. Sachez qu'il est normalement employé dans sa version anglaise : **Template**. Ainsi nous dirons Template de projet.

Ces *templates* permettent de partir sur des bases saines pour différents types d'applications, de technologies.

Ce chapitre montre les différentes possibilités. Lorsque cela s'y prête, une courte démonstration vidéo vous est proposée.

3.1. Choisir son template

ASP.NET est une technologie qui en regroupe plusieurs. Il y a plusieurs façons de créer une application web avec ASP.NET, ce tutoriel n'en traitera qu'une seule : nous verrons laquelle et pourquoi. Reprenons un schéma simplifié montrant les couches de ASP.NET :

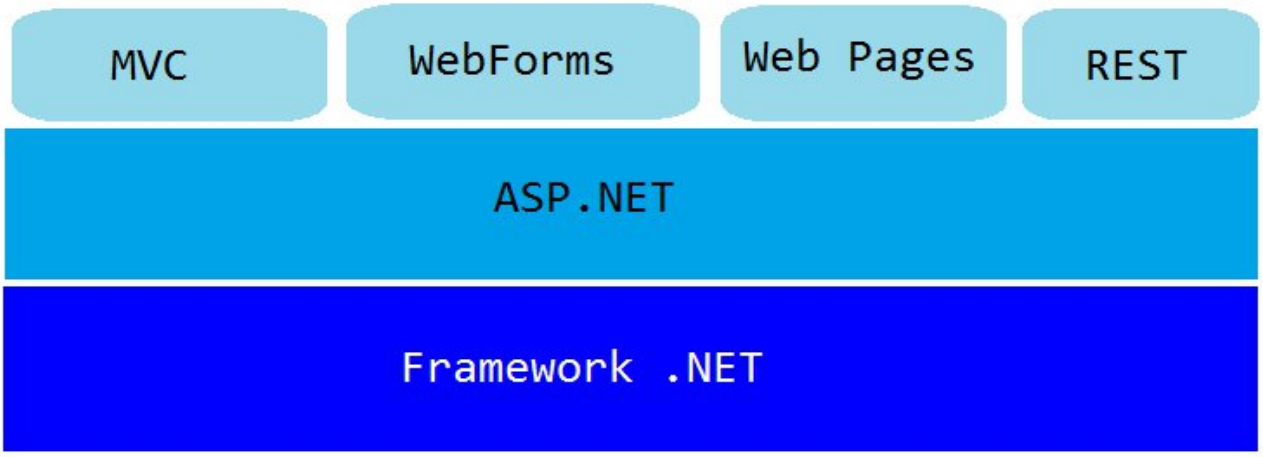


FIGURE 3.1. – les différentes couches du framework ASP.NET

Plusieurs choix s'offrent à nous, chacun proposant des approches différentes pour réaliser une application :

— l'approche MVC ;
— l'utilisation des Web Forms ;
— la page web ;
— l'API REST.

Nous les retrouvons dans les différents templates de Visual Studio :

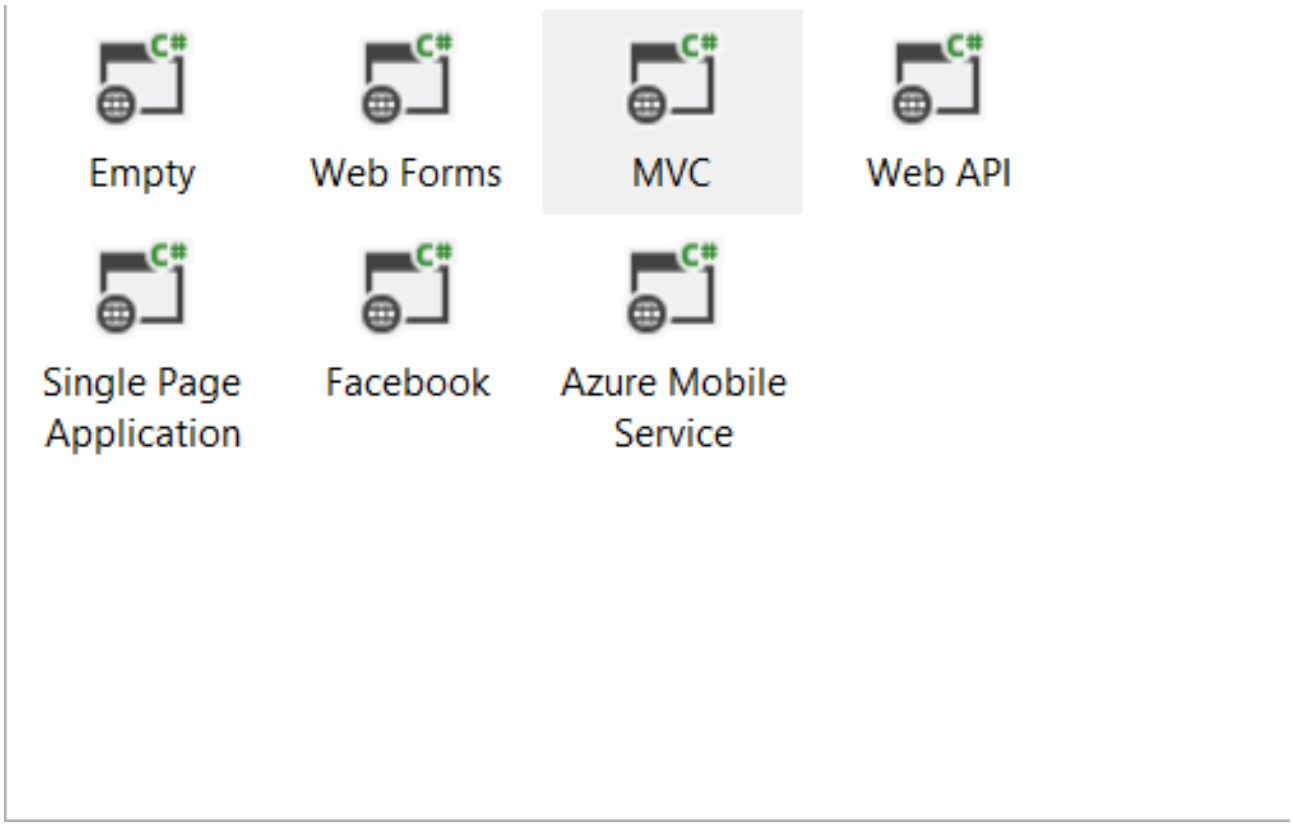


FIGURE 3.2. – Liste des applications

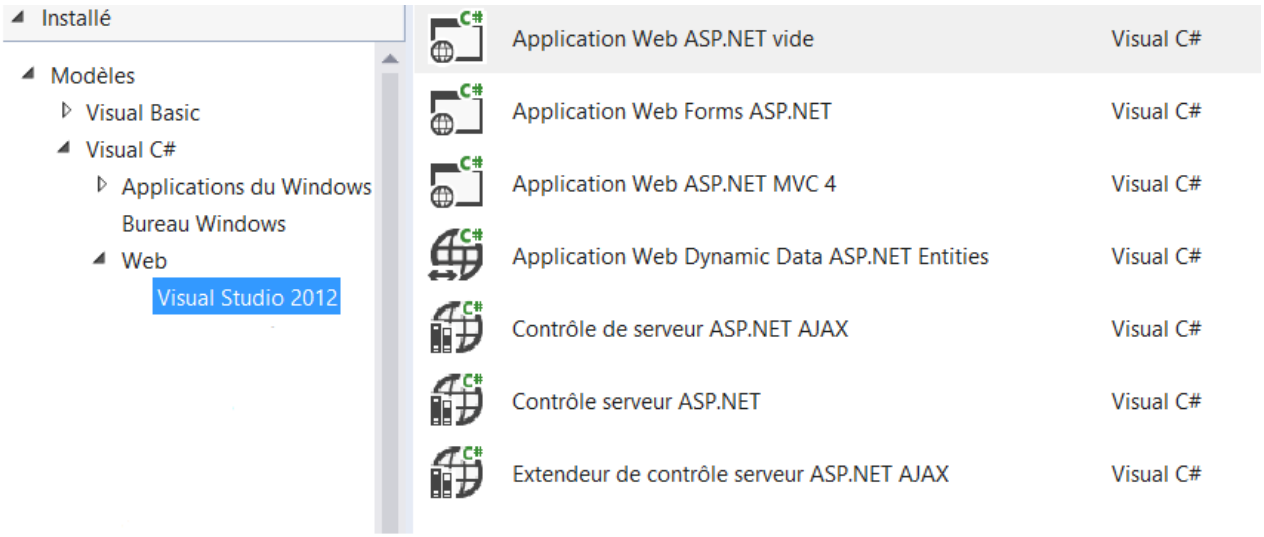


FIGURE 3.3. – Liste des applications dans l'ancienne version

? Il y a donc plusieurs ASP.NET. Lequel est le mieux ? Et lequel allons-nous utiliser ?

Nous parlerons plutôt de technologies liées au framework ASP.NET. Nous allons vous montrer et vous expliquer la physionomie de chacune de ces technologies. Et enfin, nous vous expliquerons laquelle nous allons utiliser pour ce tutoriel. Sachez que ça reste du C#, vous pourrez aisément passer de l'une à l'autre après avoir suivi ce tutoriel.

3.2. Projet basé sur l'architecture MVC

Vous est-il déjà arrivé de faire une application en C# en vous disant : *"Le code est vraiment mal organisé, mais bon, au moins ça marche !"* ?

Trop de développeurs, et pas seulement les débutants, ne savent pas organiser leur code, ce qui peut par la suite poser des problèmes. En fait, il y a des problèmes en programmation qui reviennent tellement souvent qu'on a créé toute une série de bonnes pratiques que l'on a réunies sous le nom de **design patterns**. En français, on dit "patron de conception".

Le MVC est un design pattern[3] très répandu. MVC est un acronyme signifiant **Modèle - Vue - Contrôleur**. Ce patron de conception, permet de bien organiser son code pour son application web.

Le code y est séparé en trois parties : la partie Modèle, la partie Vue et la partie Contrôleur. Les lecteurs attentifs auront remarqué que la solution du projet que nous avons créée précédemment comportait trois répertoires portant ces noms :

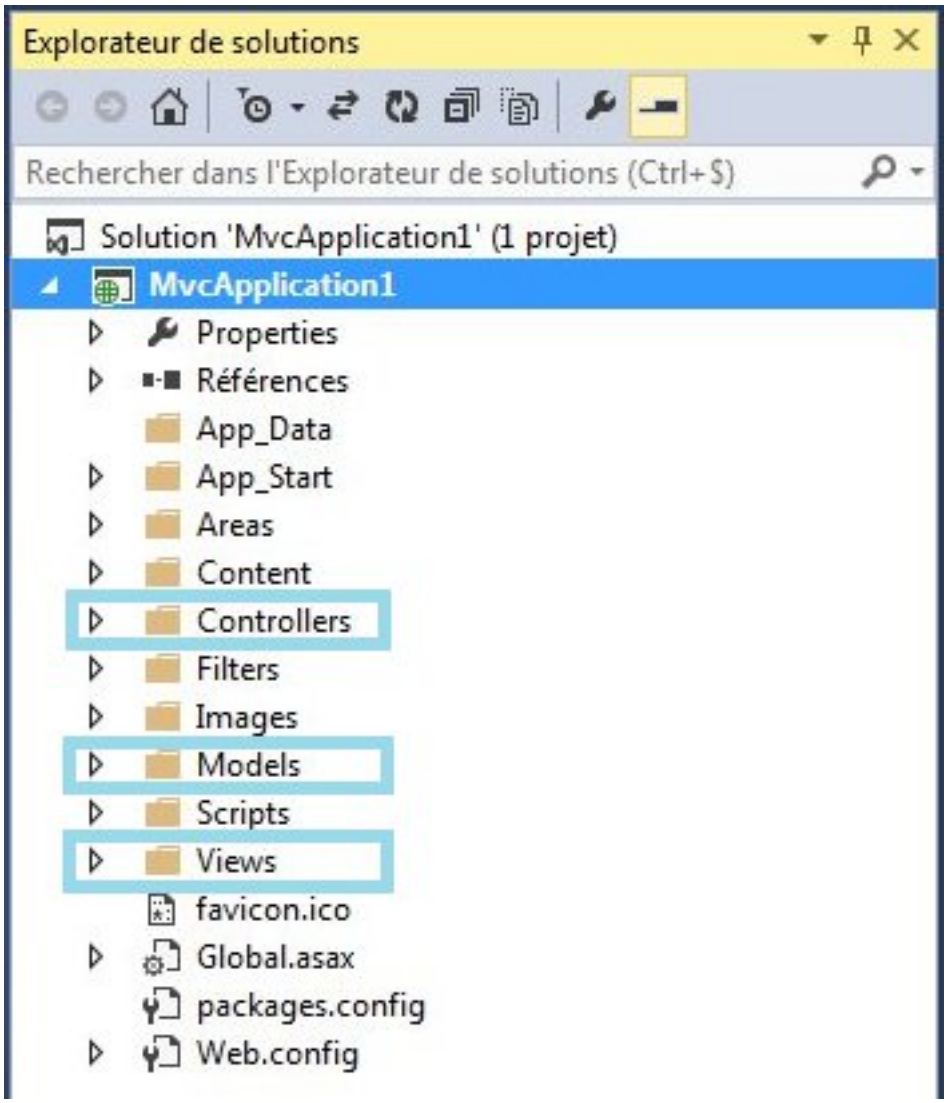


FIGURE 3.4. – Models, Views, Controllers

Ci-dessous, ce que nous trouvons dans chaque partie :

— **Modèle** : les modèles vont gérer tout ce qui a trait aux **données** de votre application. Ces données se trouvent en général dans une base de données. Les données vont être traitées par les modèles pour ensuite être récupérées par les contrôleurs.
— **Vue** : comme son nom l'indique, une vue s'occupe... de l'affichage. On y trouve uniquement du code HTML, les vues récupèrent la valeur de certaines variables pour savoir ce qu'elles doivent afficher. Il n'y a pas de calculs dans les vues.
— **Contrôleur** : un contrôleur va jouer le rôle d'intermédiaire entre le modèle et la vue. Le contrôleur va demander au modèle les données, les analyser, prendre des décisions et renvoyer le texte à afficher à la vue. Les traitements se font dans le contrôleur. Un contrôleur ne contient que du code en C# et réalise une **action**.

Résumons le tout schématiquement :

FIGURE 3.5. – modèle - données

FIGURE 3.6. – vue - affichage

FIGURE 3.7. – contrôleur - action

En suivant ce que nous avons écrit plus haut, les éléments s'agencent et communiquent entre eux de cette façon :

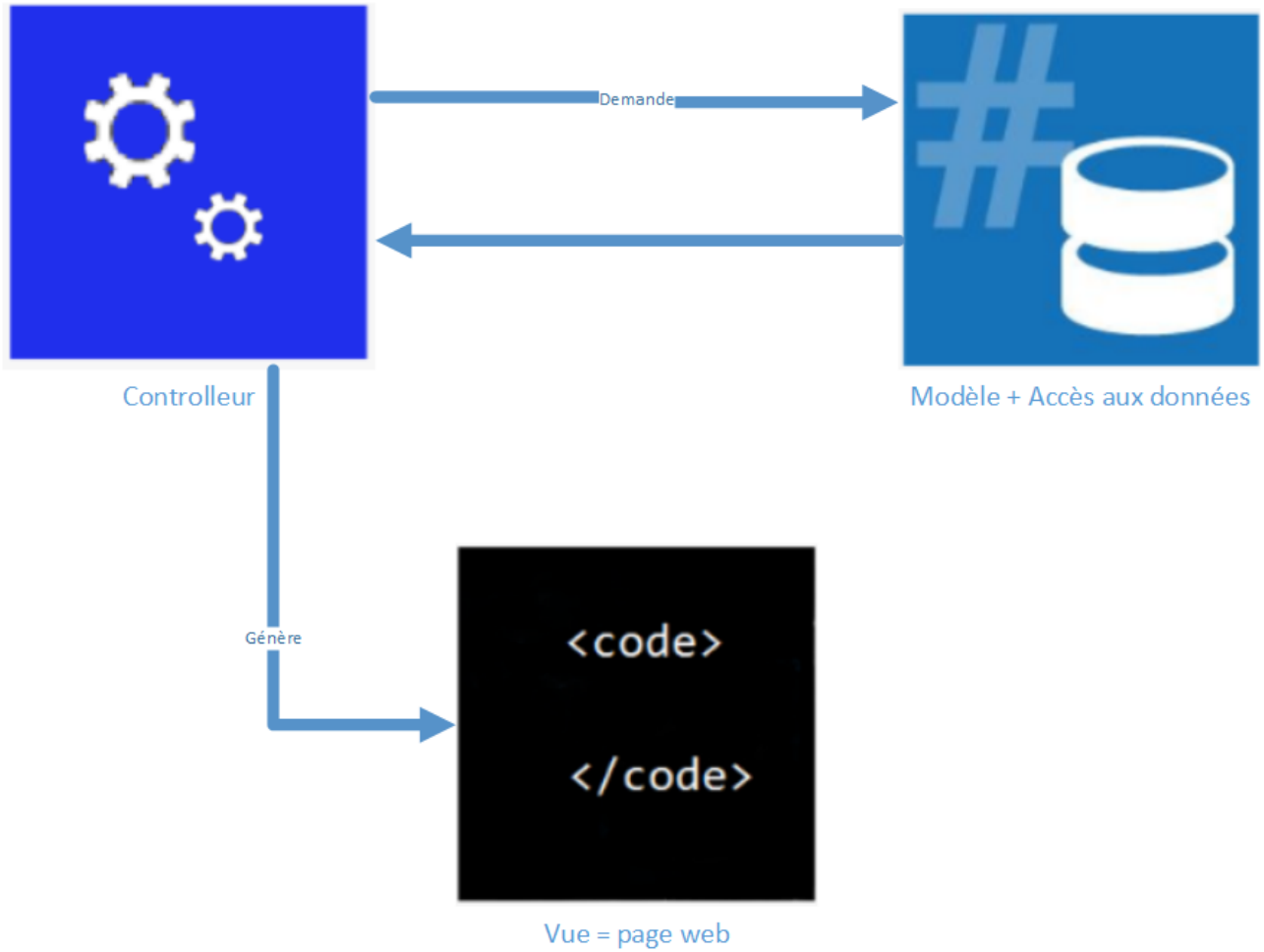


FIGURE 3.8. – Échange d'informations entre les éléments

Nous remarquerons que tout passe par le contrôleur : il va chercher les données dans le modèle, le modèle lui retourne ces données et enfin il transmet ces données à la vue. Lorsque qu'un visiteur demande une page web depuis son navigateur, sa requête est dans un premier temps traitée par le contrôleur, seule la vue correspondante est retournée au visiteur.

i Pour rester simple, j'ai volontairement présenté un schéma simplifié. Une vraie application ASP.NET MVC fera intervenir des éléments supplémentaires.

MVC est une architecture vraiment souple et pratique mais quelque peu floue, lorsque nous commencerons à pratiquer vous en comprendrez mieux les rouages.

3.3. Les WebForms

ASP.NET WebForms, c'est tout un mécanisme qui permet de faciliter la création d'une application web en faisant comme si c'était une application Windows. Plus précisément, nous allons

3. Les patrons de conceptions sont nombreux, et participent surtout à la *programmation orientée objet*. Une grande partie d'entre eux a été détaillée par le gang of four .

nous inspirer des applications Windows Form. Le design par défaut des composants sera le même.

Il permet de travailler avec une approche événementielle, comme une application Windows. Le premier but d'ASP.NET WebForms était de faire en sorte que les personnes qui avaient déjà fait du développement Windows (en C# ou en VB.NET) puissent facilement faire du développement web, dans un contexte qui leur était familier.

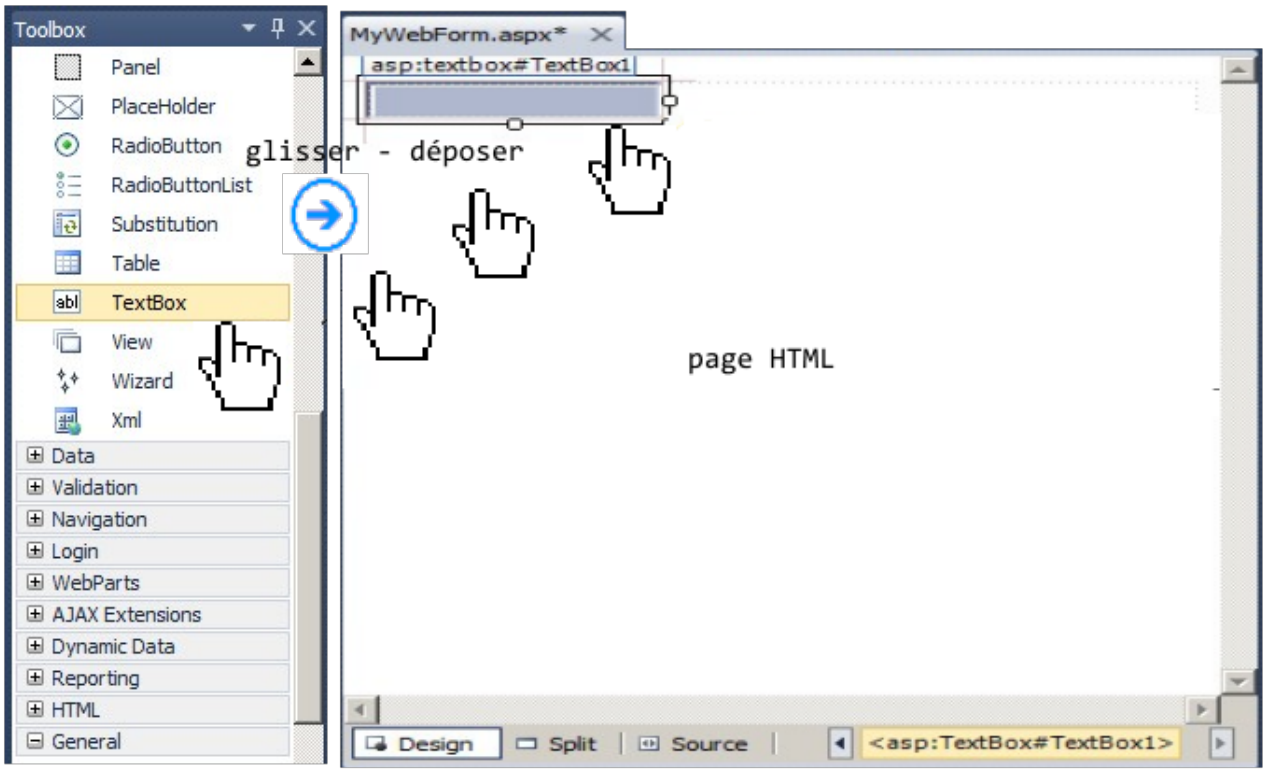


FIGURE 3.9. – Principe d'une application ASP.NET Web Form

Le web et les applications bureau sont deux mondes très différents. Si les grands principes sont les mêmes, beaucoup de différences existent. La plus évidente est la manière via laquelle sont gérés les événements. Les pages web retiennent beaucoup moins bien le contexte d'exécution, on dit qu'elles sont *stateless*.

Dans une application WebForm, une page HTML est constituée de contrôles serveur (un bouton, un champ de texte modifiable, un tableau, des cases à cocher, etc), de scripts clients et du code serveur (généralement du code C# directement inclus dans le HTML via des balises spéciales).

D'un autre côté, nous retrouvons aussi une séparation du code C# et du code HTML : les contrôles sont glissés sur la page web et le C# gère la partie évènementielle en arrière-plan : lorsque l'on clique sur ce bouton il va se passer ceci, etc. C'est une manière vraiment simple de réaliser une application web.

Les inconvénients avec une application WebForm résident dans l'organisation du code et de l'abstraction, nous entendons par là que le code est moins organisé et que lors du développement

d'une grosse application web, vous risquez de vous perdre facilement. Donc assez difficile à maintenir. Un autre défaut est le modèle de développement similaire à celui d'un application Windows, et donc un débutant aura tendance à développer de la même façon en WebForm, ce qui est une erreur, car les deux n'ont rien à voir.

3.4. Les api REST

3.4.1. Définition de API REST

API REST... Deux acronymes l'un à côté de l'autre et qui occupent une place extrêmement importante dans le web actuel.

API, cela signifie "interface pour programmer des applications". Basiquement, c'est un ensemble de fonctions qui vous permettent - dans le meilleur des cas - de faire très facilement des opérations souvent complexes.

Par exemple, lorsque vous utilisez LINQ, vous utilisez une API composée de tout un tas de fonctions telles que `Select` ou `Where`. Le mot API n'est donc pas directement lié au web. Pourtant, vous entendrez de temps en temps parler d'API Wikipédia, Twitter, IMDB... Et eux, ce sont des sites web.

Le principe de base d'une API web, c'est de dire que votre site devient un *service* et donc en envoyant une requête vous pourrez obtenir les données que ledit service accepte de vous donner. Cela peut être le contenu d'une page Wikipédia par exemple.

Le gros avantage de ce service, c'est qu'il ne vous fournit plus une page HTML avec tout le design associé mais uniquement le contenu. Ce qui permet d'accélérer globalement les choses, puisque s'il n'y a que le contenu, il y a moins de choses à télécharger, moins de choses à interpréter...

L'idée d'un service web n'est pas neuve. Il y a encore quelques années, un modèle de service était très à la mode : SOAP.

Ce dernier s'appuyait uniquement sur le XML et certaines en-têtes HTTP pour fournir un service aux utilisateurs. C'est aussi un système assez complexe qui s'adapte mal au web.

S'il n'a pas été aussi plébiscité que les API REST, c'est simplement qu'il était trop complexe à utiliser pour ceux qui voulaient se servir des données du service. En outre, le XML étant relativement lourd, il faut écrire beaucoup de code pour exprimer des choses même basiques.

Puis le JavaScript est devenu célèbre. Et avec lui le JSON. Un format très léger et qui permet de transporter des données de manière simple et rapide.

L'idée de REST est d'allier la légèreté de JSON (bien qu'il soit compatible avec XML) pour transmettre facilement la représentation des données tel que le service sait les manipuler.

Comme il n'y a qu'un simple "transfert de données", l'utilisation de REST va être simplifiée grâce à l'utilisation d'URL simples, comme l'adresse du site. Seule la **méthode HTTP** permettra de différencier l'action à faire.

Par exemple, le site Zeste De Savoir propose une API REST qui permet à certains développeurs de créer des applications mobiles par exemple. Lors qu'un utilisateur de ces applications envoie un message sur les forums, il envoie une requête **POST** avec une ressource "message" qui

contient le texte de son message et l'identifiant du sujet. De même lorsqu'il veut afficher les forum, il envoie une requête **GET** avec l'identifiant du forum ou du sujet à afficher.

3.4.2. Faire une API REST avec ASP.NET

Le *template* `Web API` de ASP.Net ressemble énormément à un site MVC.

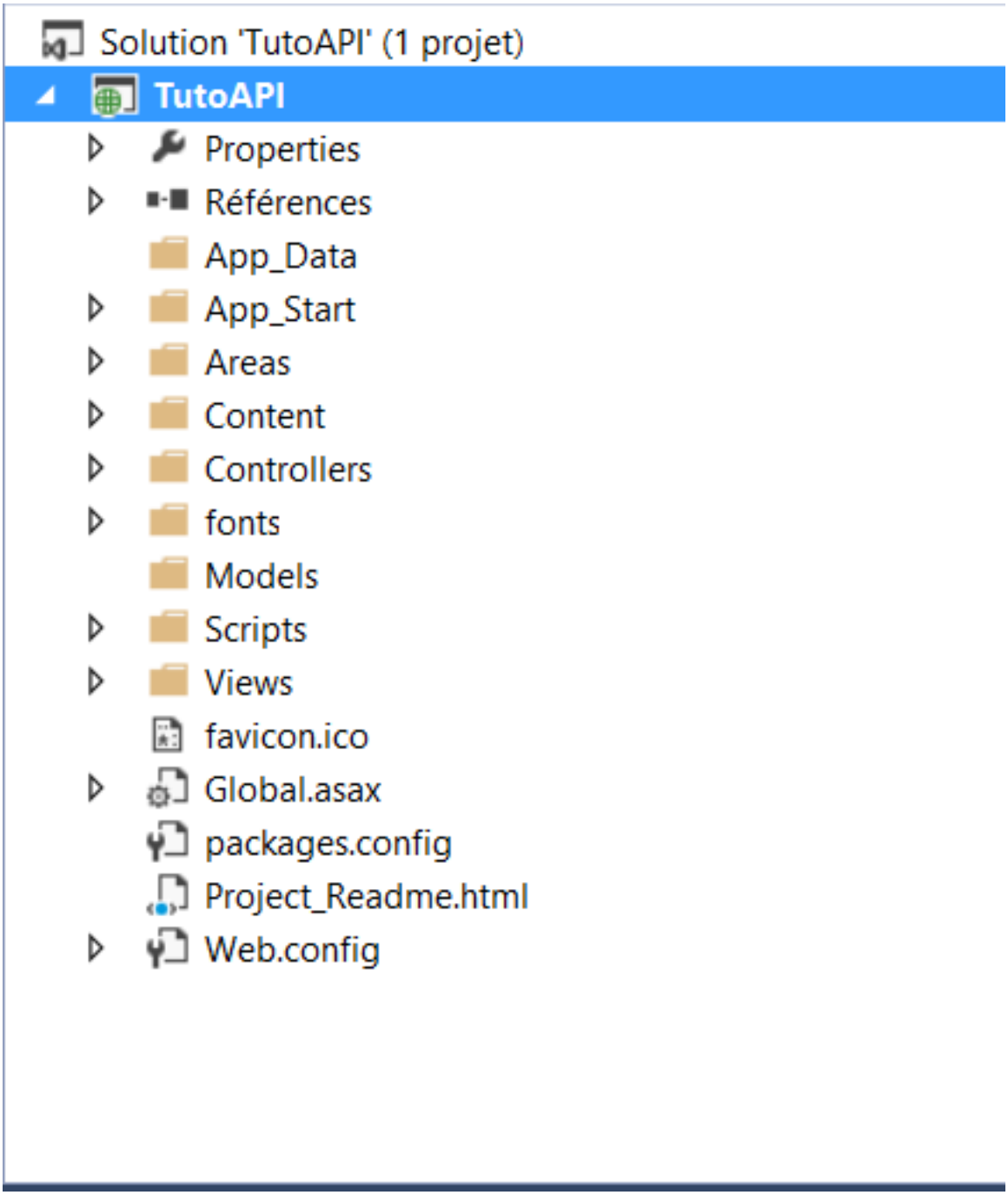


FIGURE 3.10. – Architecture d'un projet API

La raison est simple : vous allez utiliser du MVC sauf qu'au lieu d'avoir des vues, le contrôleur renverra juste du JSON ou du XML selon votre configuration.

? S'il n'y a pas de vue, pourquoi il y a un dossier vue ?

Une API doit être utilisée par des programmeurs pour faire une application mobile, enrichir leur site... S'ils ne comprennent pas comment fonctionne votre API, comment vont-ils faire ?

Le dossier vue contient les vues minimum pour que ASP.NET soit capable de générer **automatiquement** de la documentation qui expliquera à vos utilisateurs comment utiliser votre API.

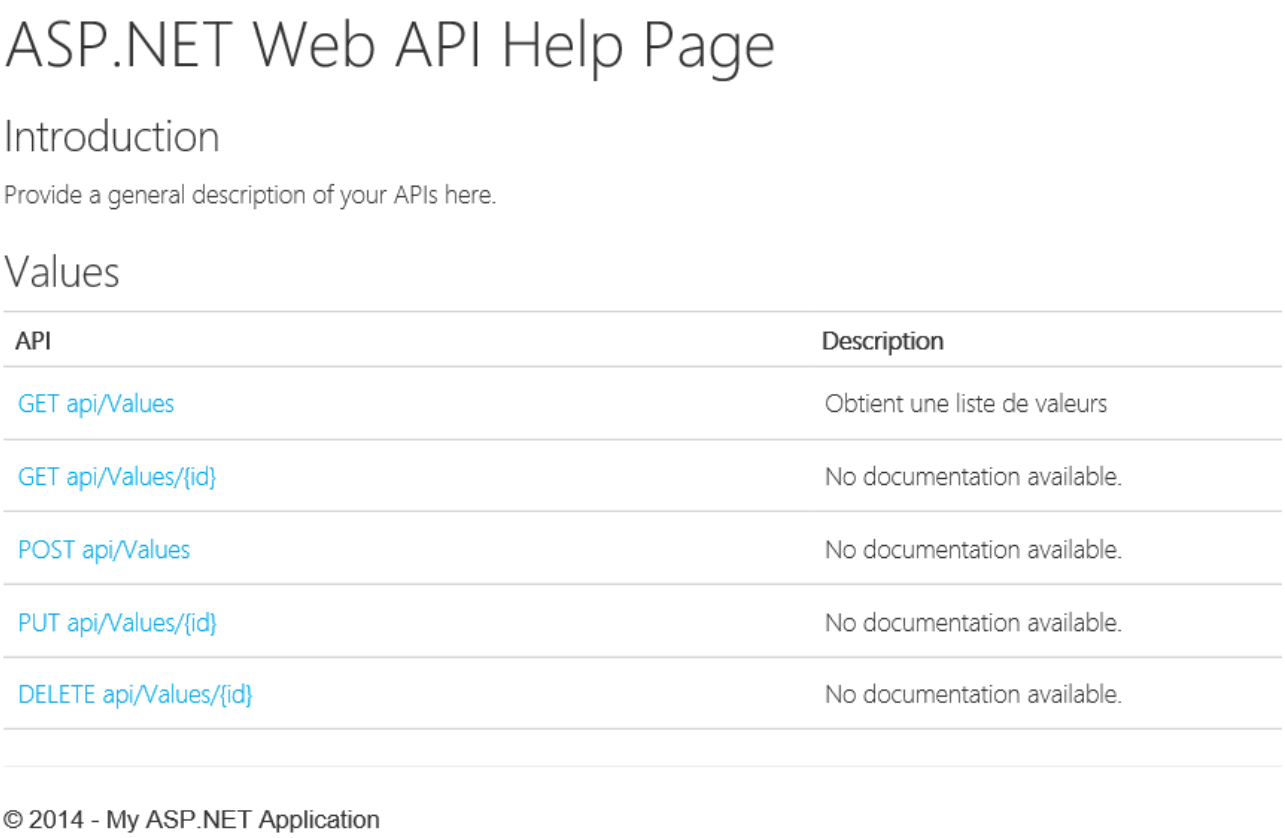


FIGURE 3.11. – Exemple de génération de documentation

3.5. Les applications sur une seule page

3.5.0.1. Définition

Application sur une seule page, cela peut sembler étrange à première vue.

Alors, qu'est-ce que c'est ? Une application sur une seule page est... une application web qui ne charge qu'une seule et unique page HTML.

? Donc on est obligé de tout mettre dans un fichier ?

Non, rassurez-vous. En fait, cela vaut juste dire que le serveur ne génèrera qu'une seule page HTML, pour le reste, il va être un peu plus fin.

Dans une application sur une seule page, le contenu se rafraîchit de manière "dynamique" au fur et à mesure de l'interaction avec le visiteur. Par exemple, si vous faites un blog sur une seule page, lorsque l'utilisateur cliquera sur "commenter", le formulaire se créera tout seul et quand vous enverrez le commentaire, il ne rechargera pas la page, il va juste dire au serveur "enregistre le commentaire" et c'est tout.

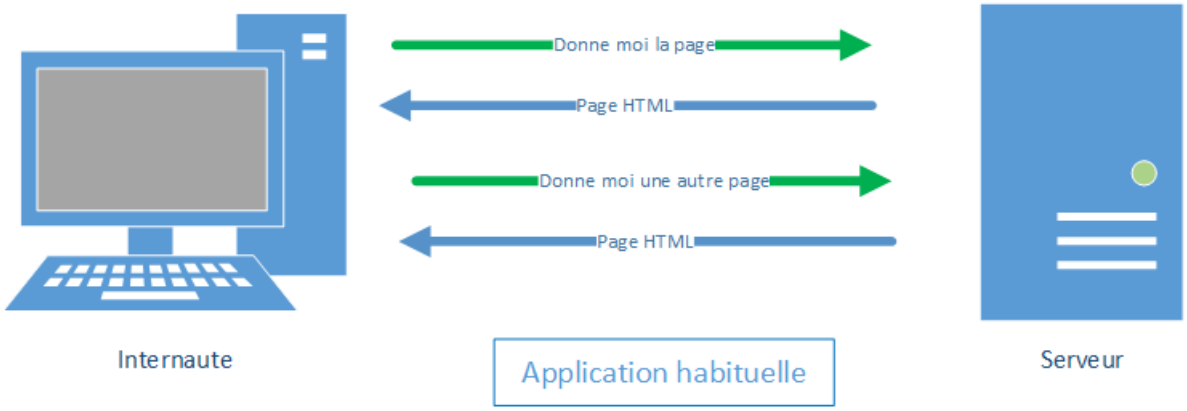
Ce modèle utilise massivement des technologies JavaScript pour créer des applications web fluides et réactives, qui n'ont pas besoin de recharger constamment les pages.

3.5.0.2. Différence avec une application web ASP.NET classique

Comme vous pouvez vous en douter, une application sur une seule page ne fonctionne pas de la même façon qu'une application web classique. Dans une application web classique, chaque fois que le serveur est appelé, il répond par une nouvelle page HTML. Cela déclenche une actualisation de la page dans le navigateur.

Tandis qu'avec une application sur une seule page, une fois la première page chargée, toute interaction avec le serveur se déroule à l'aide d'appels JavaScript.

Un schéma vaut mieux qu'un long discours :



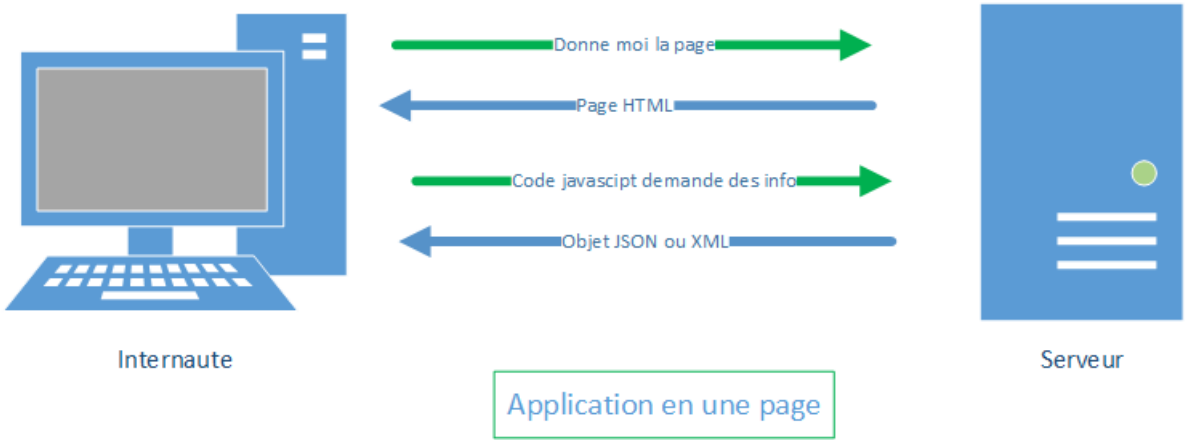


FIGURE 3.12. – Différence entre applications classique et sur une seule page

3.5.0.3. Avantages et inconvénients

Un avantage indéniable d'une application sur une seule page est la fluidité ; en effet les pages n'ont plus besoin d'être chargées, toute interaction avec l'interface utilisateur se produit côté client, par l'intermédiaire de JavaScript et CSS.

L'inconvénient est que la grosse partie de l'application se déroule côté client et non côté serveur : ce qui signifie pour un développeur ASP.NET d'utiliser plus de JavaScript que de C#.

3.6. Quelle technologie utiliser ?

Historiquement parlant, WebForm est la technologie qui a fait connaître ASP.NET.

Pour autant, les WebForms sont en perte de vitesse face à l'utilisation de MVC et des API REST.

Le MVC pour le web a trouvé ses lettres de noblesse dans bien des langages tels que Java, PHP ou Ruby. Les procédures utilisées sont donc assez standardisées, ce qui facilite la maintenance et l'arrivée de nouveaux développeurs sur un projet. Pratique, par exemple, si vous désirez partager votre code sur github comme le fait ZdS !

MVC permet une réelle **efficacité** lorsque vous codez, ASP.NET vous proposant déjà beaucoup de packages prêts à être utilisés via *nuget*.

La popularité de MVC est surtout due à une chose : lorsque vous êtes sur un site internet, les actions que vous réalisez sont découpées de manière à répondre toujours à ce schéma :

création->validation->enregistrement->affichage->modification->validation->enregistrement->affichage.

Imaginons que nous avons un blog.
En tant qu'auteur, nous voulons ajouter des articles.

1. Nous allons alors **créer** cet article ;
2. Puis le programme va vérifier qu'il est correct (par exemple : le titre est-il renseigné ?) ;
3. Puis nous allons l'afficher pour pouvoir le lire ;
4. Nous allons vouloir corriger nos fautes d'orthographe et donc nous allons le modifier ;
5. Une nouvelle fois, le programme vérifie que nous n'avons pas fait de bêtises ;
6. Et nous affichons la version corrigée.

Nous avons vu les différents modèles qu'offrait le framework web ASP.NET à travers ce chapitre. Il offre différents styles de programmation, ce qui montre son étendue. Dans ce tutoriel, nous utiliserons le modèle MVC (Modèle-Vue-Contrôleur) car il va nous permettre de conserver une bonne organisation dans le code.

Et nous allons commencer dès le prochain chapitre !

Pour conclure cette partie, rien de mieux qu'une petite vidéo qui vous propose une visite guidée dans les méandres de Visual Studio et ASP.NET.

!()

Deuxième partie

Les bases de ASP.NET

Démarrons donc le développement de notre application. Afin de vous fournir un exemple concret de ce qui est faisable en ASP.NET MVC, nous vous proposons de créer un blog complet.

Ce blog sera notre fil rouge, à la fin de ce tutoriel, il devrait être complet avec ajout et suppression des articles, des commentaires, affichage en temps réel...

Pour faire cela, créez une application avec le template MVC. Choisissez l'authentification "Comptes d'utilisateur individuels".

4. Ma première page avec ASP.NET

La tradition, chez les développeurs, quand on touche à une nouvelle technologie, c'est de dire bonjour au monde grâce à cette technologie.

Dans ce chapitre, nous allons enfin toucher à l'ASP.NET. Nous ne commencerons pas en force, nous verrons comment créer une petite application web avec l'organisation Modèle-Vue-Contrôleur. Alors en avant pour notre premier **Hello Word** !

4.1. Objectif : dire bonjour !

Pour commencer ce tutoriel dans de bonnes conditions, l'objectif ici est de réaliser un très petite application web afin d'intégrer la notion de Modèle-Vue-Contrôleur. Nous l'avons nommée **BonjourMVC**.

Dans cette application, le visiteur rentrera son nom dans un champ de texte modifiable et lorsqu'il cliquera sur le bouton Valider, nous lui afficherons "Bonjour" suivi du texte saisi. Voici la maquette de notre objectif :

FIGURE 4.1. – BonjourMVC, notre première application

Ça paraît simple, mais nous vous assurons que c'est déjà du travail !

4.2. Créer le projet MVC vide

Nous allons créer le projet. Cette fois-ci nous créerons un projet ASP.NET MVC vide afin de commencer notre application de zéro.

i

En réalité, le projet généré par Visual Studio ne sera pas vide au sens propre du terme car il contiendra une architecture de répertoires et des fichiers nécessaires au bon fonctionnement de l'application.

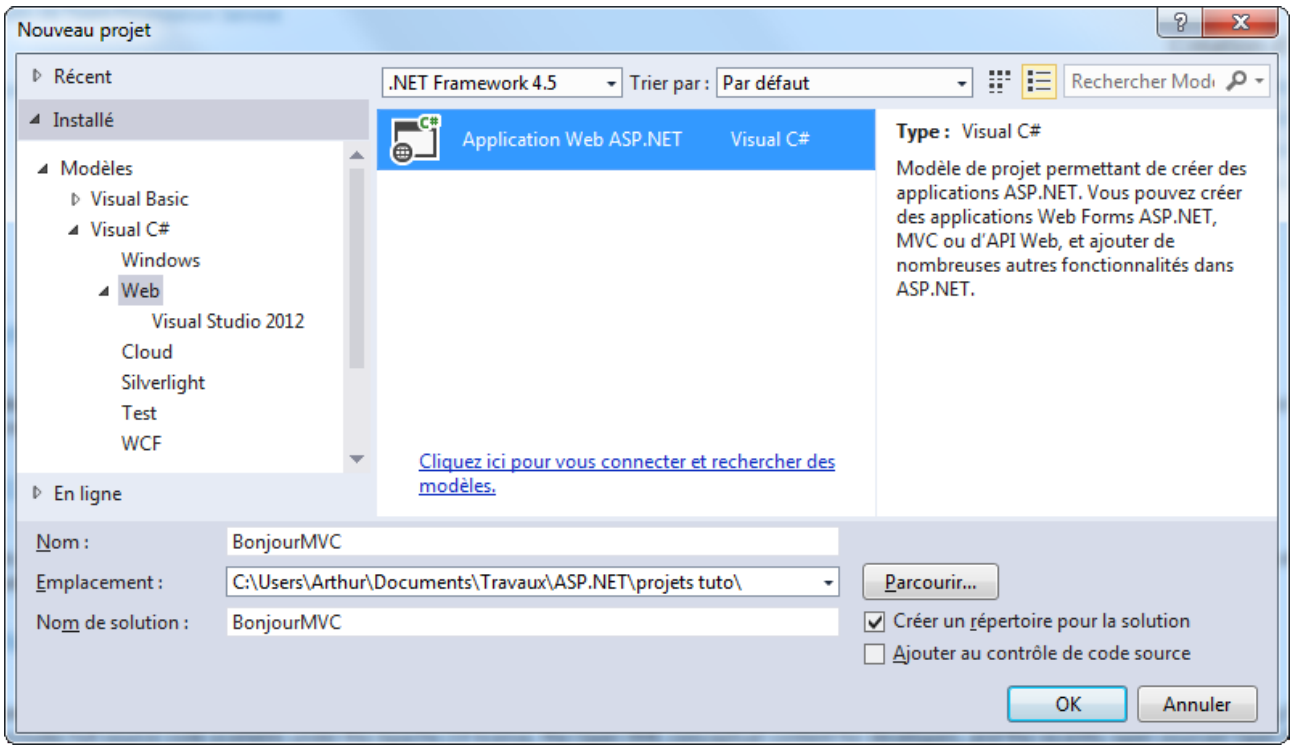


FIGURE 4.2. – Nouveau Projet application Web ASP.NET

Sélectionnez Application Web ASP.NET. Nous l'avons nommée **BonjourMVC**. Validez par OK.

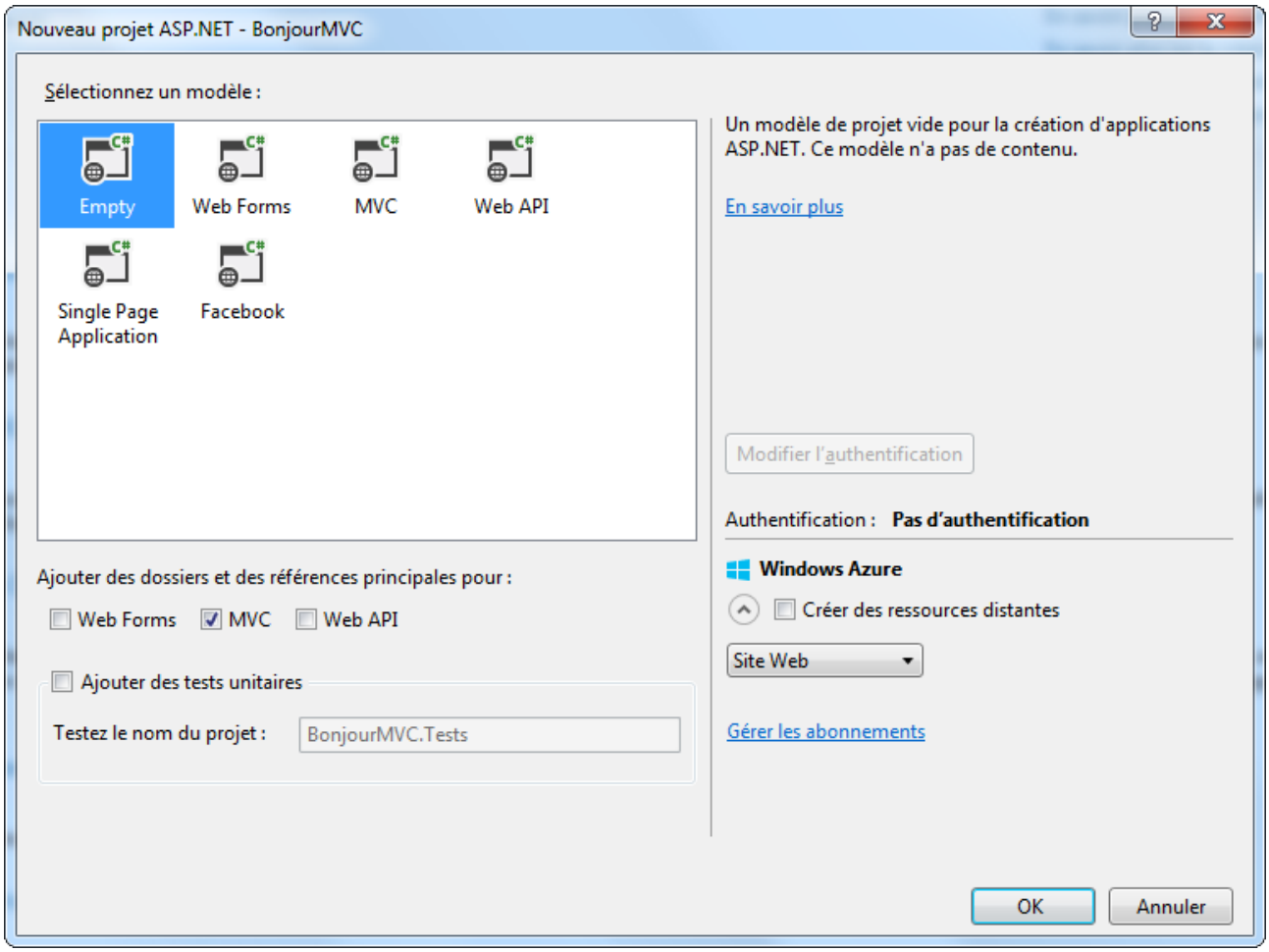


FIGURE 4.3. – Projet ASP.NET MVC vide

Cette fois-ci, sélectionnons le modèle de projet **Vide** en faisant attention à ce que la case **MVC** soit bien cochée en dessous de "*Ajouter des dossiers et des références principales pour*". Validez ensuite par OK. Visual Studio crée notre projet.

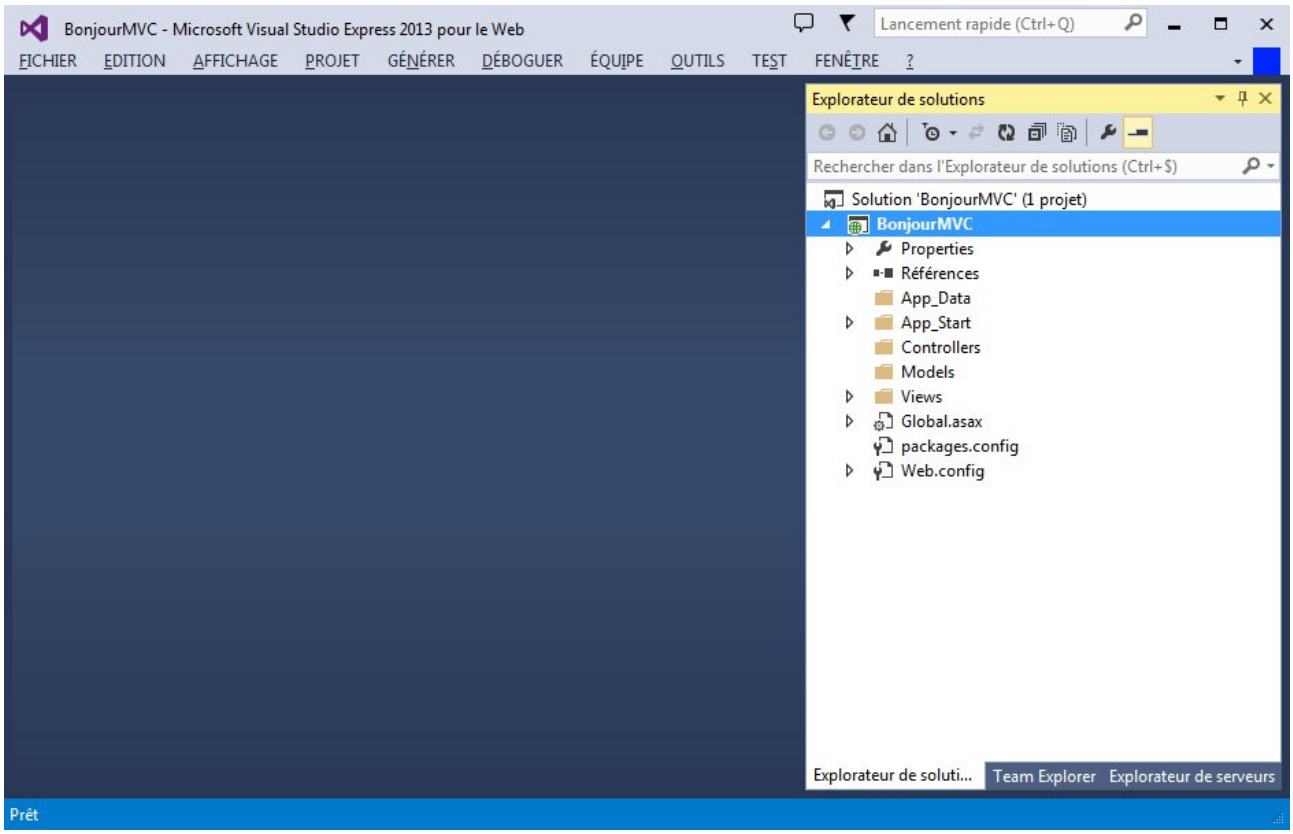


FIGURE 4.4. – Projet créé avec succès

Notons la similitude de l'organisation des dossiers entre ce projet et celui créé lors du chapitre précédent : nous retrouvons les dossiers Models - Views - Controllers. Bien évidemment, ce projet est moins fourni que le précédent.

Nous sommes prêts à commencer.

4.3. Décrire les informations : création du modèle

Commençons par créer un modèle assez simple. Le modèle va contenir les données de notre application. Pour celle-ci, ce sera simple : nous n'avons que le prénom du visiteur à récupérer.

Le visiteur va rentrer son prénom et celui-ci va être stocké dans une propriété **Prenom**. Nous allons donc créer une classe **Visiteur** contenant une unique propriété. Pour créer une classe, utilisons le clic droit de la souris sur le répertoire Models, puis Ajouter > Classe...

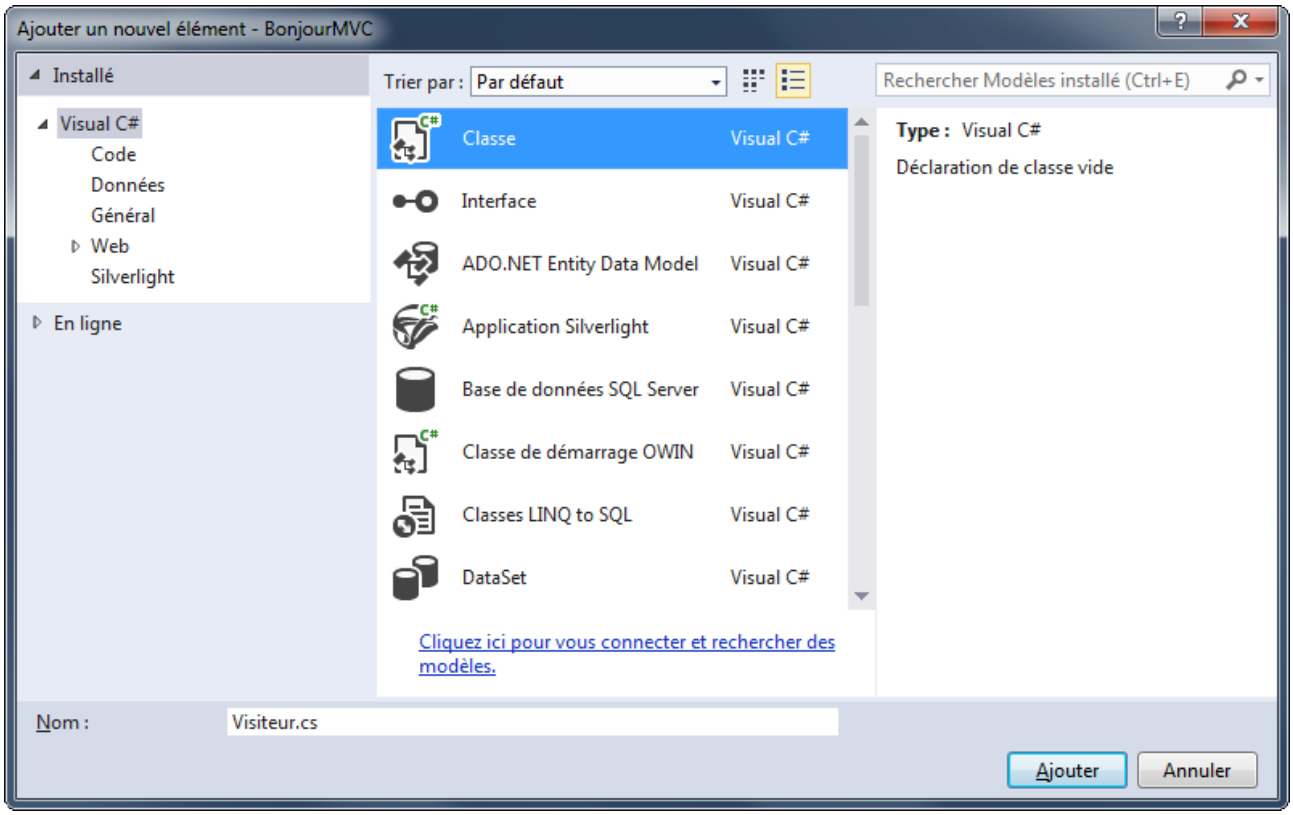


FIGURE 4.5. – Ajouter une classe de modèle

Visual Studio génère le code suivant :

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5
6  namespace BonjourMVC.Models
7  {
8      public class Visiteur
9      {
10     }
11 }
```

Rajoutons maintenant notre propriété `Prenom`. Le visiteur va rentrer son prénom dans un champ de texte, donc la propriété sera de type `string`. Ce qui donne :

```
1  public class Visiteur
2  {
3      public string Prenom { get; set; }
4  }
```

C'est bon pour le modèle. Notre classe ne contient quasiment rien mais ce sera suffisant pour notre application. Prochaine étape : le contrôleur.

4.4. Traiter les informations : création du contrôleur

Le contrôleur va faire la jonction entre le modèle que l'on a réalisé précédemment et la vue. Lorsque nous envoyons une requête au serveur, ce sont les contrôleurs qui s'occupent de la traiter. Dans un contrôleur, chaque méthode de classe représente une URL. Vous allez comprendre.

Créons donc un nouveau contrôleur. Pour ce faire, créez un répertoire **Salutation** dans le répertoire Controllers. Ensuite, faites un clic-droit sur le répertoire Salutation puis Ajouter > Contrôleur...

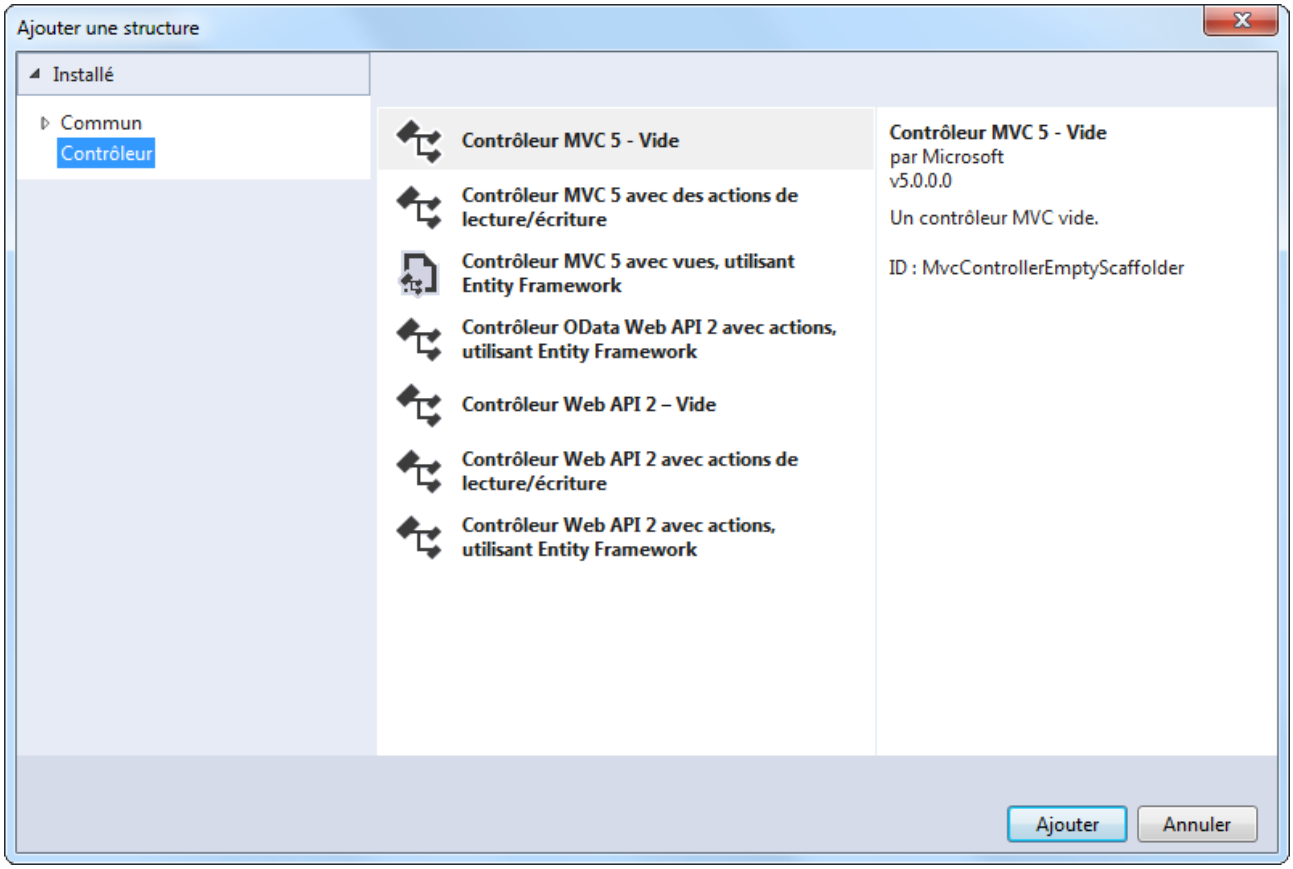


FIGURE 4.6. – Ajout d'un nouveau contrôleur

Comme nous le voyons, il existe plusieurs modèles de contrôleur. Pour l'instant, nous choisirons le **contrôleur MVC 5 - Vide**. Cliquez sur OK.

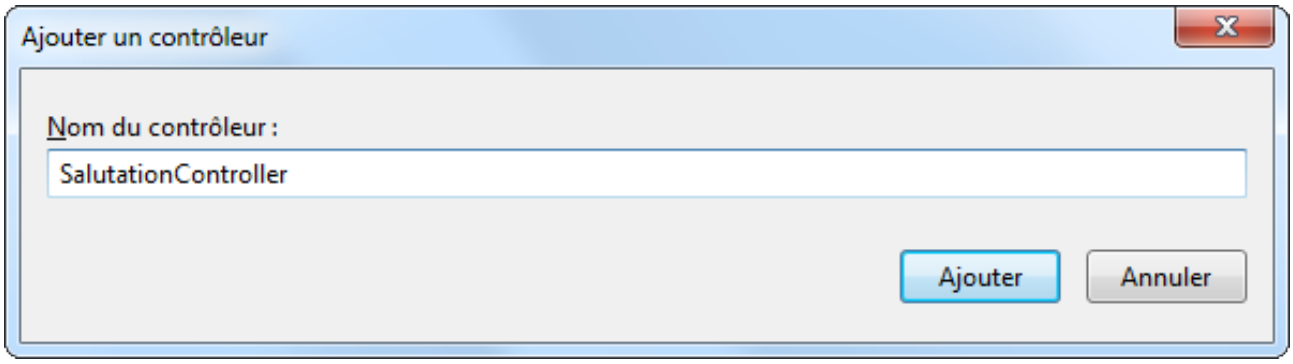


FIGURE 4.7. – Nommez le nouveau contrôleur

Nommons ce nouveau contrôleur **SalutationController**.

> *i* Le nom d'un contrôleur doit se terminer de préférence par Controller. C'est une convention à respecter.

Visual Studio nous génère le code suivant :

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.Web.Mvc;
6
7  namespace BonjourMVC.Controllers.Salutation
8  {
9      public class SalutationController : Controller
10     {
11         //
12         // GET: /Salutation/
13         public ActionResult Index()
14         {
15             return View();
16         }
17     }
18 }
```

La classe générée est un peu plus difficile à comprendre qu'une classe de base. Nous noterons que la classe hérite de la classe du framework ASP.NET MVC **Controller** (lien MSDN). Il y a en plus une méthode `Index` qui retourne un objet `ActionResult` (lien MSDN), ce qui signifie que la méthode va réaliser l'action correspondant à son nom.

Dans une classe de contrôleur, chaque méthode publique représente une **action**[4]. ASP.NET possède un mécanisme de **routage**, que nous découvrirons dans un prochain chapitre et qui permet de décrire comment interpréter l'URL et exécuter l'action correspondante. Ainsi, lorsque

que nous enverrons une requête vers la méthode **Index** de notre contrôleur **Salutation**, notre application donnera l'URL suivante : **http ://localhost/Salutation/Index** .

La méthode de classe du contrôleur va renvoyer une **vue** par défaut, grâce à la méthode `View()`. En fait, cette méthode va renvoyer la vue qui porte le nom par défaut **Index** et qui sera située dans le répertoire du même nom que le contrôleur.

Si nous exécutons l'application maintenant et que nous envoyons une requête vers la méthode Index du contrôleur Salutation à l'adresse http ://localhost :votre_numero_de_port/Salutation/Index , le serveur ASP.NET nous affichera l'erreur suivante :

> La vue « Index » ou son maître est introuvable.

Le méthode Index cherche à retourner une vue du même nom dans un répertoire similaire du répertoire **Views**. Comme nous n'avons pas encore créé de vue, il est normal que la méthode renvoie une exception.

Avant de créer une vue, il faut modifier la méthode Index pour que celle-ci réceptionne ce que saisit le visiteur et le stocke dans le modèle.

Lorsque l'on va faire appel à la méthode Index, celle-ci va instancier un objet Visiteur que l'on a créé dans les modèles. N'oubliez pas d'inclure l'espace de nom `BonjourMVC.Models`.

```
1  public class SalutationController : Controller
2  {
3      //
4      // GET: /Salutation/
5      public ActionResult Index()
6      {
7          Visiteur client = new Visiteur();
8          return View(client);
9      }
10 }
```

Le visiteur est donc créé. Maintenant, il faut aller chercher le contenu du champ de texte et remplir la propriété `Nom` de `Visiteur` au moment du clic sur le bouton Valider. Pour cela, nous allons créer une deuxième méthode avec un attribut spécial qui va faire que cette méthode ne sera appelée que si nous cliquons sur le bouton.

Cette attribut est **AcceptVerbs** (lien MSDN) : il précise de quelle façon nous voulons que les données soient transmises ; il y a deux manière de transmettre des données lorsque nous sommes visiteur sur un site web : par l'URL ou par formulaire. C'est pourquoi nous devons spécifier à l'attribut la façon dont nous souhaitons récupérer les données via l'énumération `HttpVerbs` (lien MSDN).

i

L'attribut `acceptVerbs` est utile quand vous désirez accepter *plusieurs* verbes. Si vous ne désirez accepter qu'un seul verbe (POST, GET), une version raccourcie existe : `[HttpPost]`, `[HttpGet]`.

Les données utilisateur entrées via l'URL sont des requêtes de type **GET** et celles entrées via les formulaires sont des requêtes de type **POST**.

```
1  public class SalutationController : Controller
2  {
3      //
4      // GET: /Salutation/
5      public ActionResult Index()
6      {
7          Visiteur client = new Visiteur();
8          return View(client);
9      }
10
11     [AcceptVerbs(HttpVerbs.Post)]
12     public ActionResult Index(Visiteur visiteur)
13     {
14         Visiteur client = new Visiteur();
15         string prenom = "";
16
17         prenom = Request.Form["prenom_visiteur"];
18         client.Prenom = prenom;
19         ViewData["message"] = "Bonjour à toi, " + prenom;
20         return View("Index", client);
21     }
22 }
```

Que de nouvelles choses ! Nous avons créé une deuxième surcharge de la méthode Index avec un paramètre qui ne nous sert à rien dans ce cas présent. Grâce à la propriété `Request.Form` (lien MSDN) nous récupérons la valeur de l'index placé entre crochets. Ce sera le même nom d'index du côté de la vue, donc le nom du champ de texte.

Ensuite, `ViewData` (lien MSDN) est une propriété de dictionnaire qui permet de passer des données entre le contrôleur et la vue. Nous l'avons nommé `message` et il prend pour valeur "Bonjour à toi" suivi du nom récupéré dans `Request.Form`.

La nouvelle méthode retourne la vue Index en lui passant en paramètre l'objet Visiteur avec la propriété du modèle remplie.

Voilà pour le contrôleur, nous apercevons déjà la liaison modèle - contrôleur ; il ne nous manque plus que la vue maintenant.

4.5. Afficher les informations : création de la vue

C'est la partie de l'application qui vous semblera la plus familière, car nous utiliserons du code HTML.

4. C'est très simplifié. En fait, vous pouvez dire à ASP.NET "ma méthode n'est pas une action" en utilisant l'attribut `[NonAction]`.

Comme nous vous l'avons dit, chaque méthode d'un contrôleur représente une action, liée à une vue ; pour ajouter une vue, retournons dans le code du contrôleur et faisons un clic-droit dans le code de la méthode voulue (`Index`). Visual Studio nous propose d'ajouter une vue :

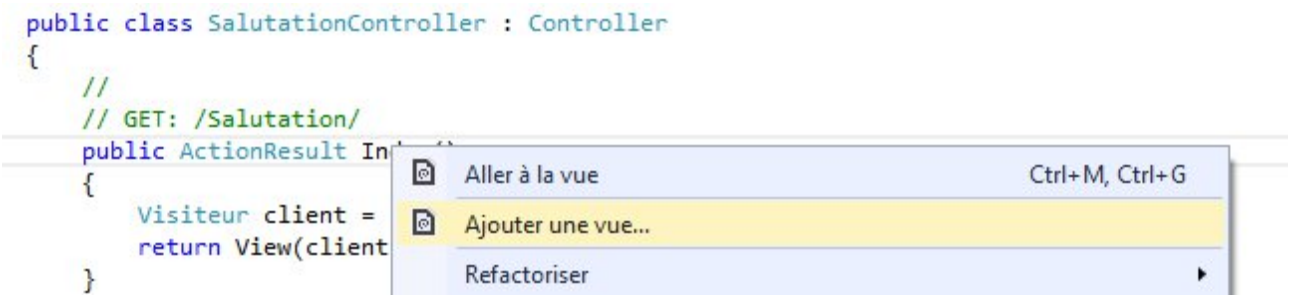


FIGURE 4.8. – Ajouter une vue depuis un contrôleur

Conservons le nom de vue Index, nous vous rappelons que le nom d'une vue doit être identique à celle de la méthode du contrôleur.

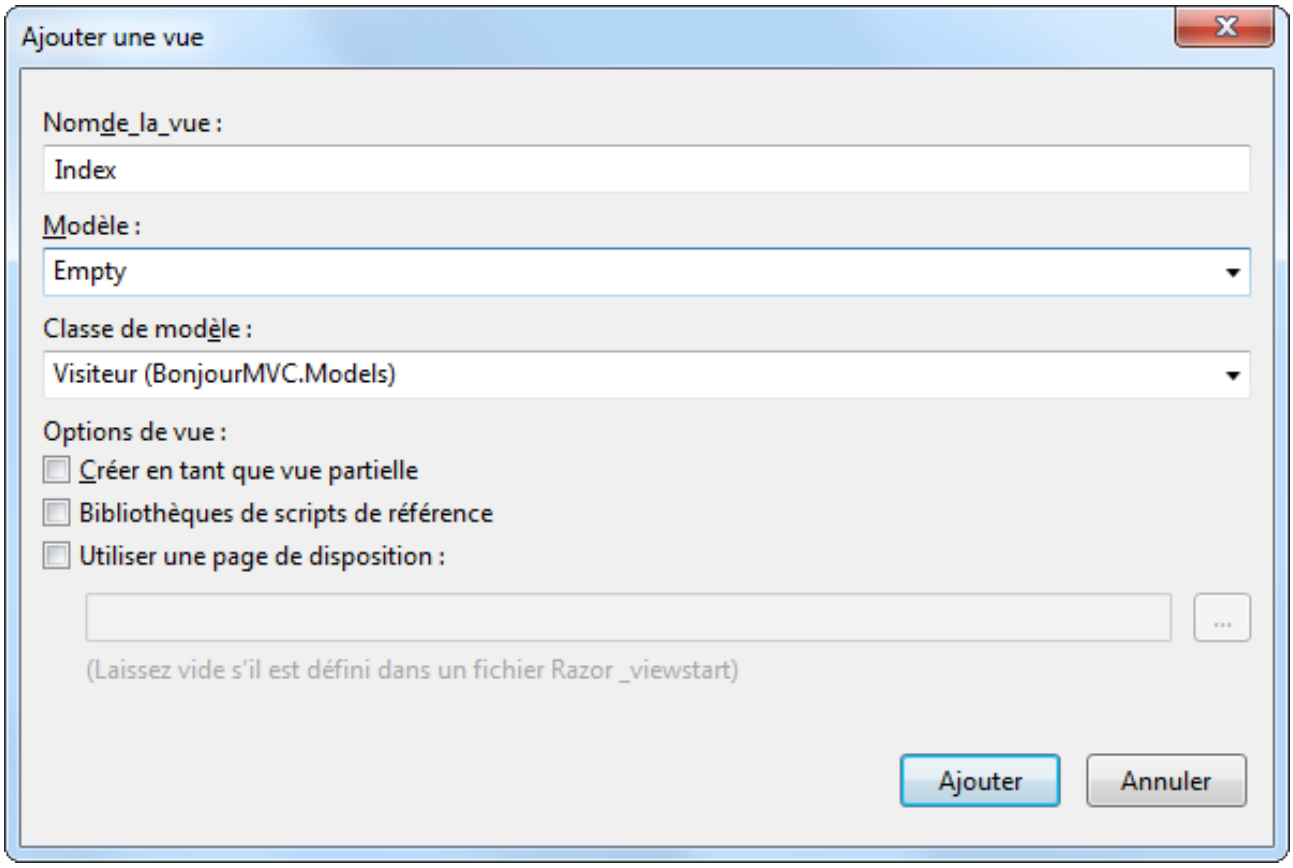


FIGURE 4.9. – Assistant d'ajout de vue

L'assistant nous propose de choisir un modèle pour notre vue : nous choisissons notre classe Visiteur en tant que classe de modèle pour pouvoir travailler directement avec le modèle dans la vue.

Cliquons sur Ajouter. Visual Studio génère le code suivant :

```
@model BonjourMVC.Models.Visiteur

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
    </div>
</body>
</html>
```

Nous reconnaissons le bon vieux HTML avec cependant quelques éléments que nous n'avons jamais vus, comme `@model` ou `@ { layout = null }`. Pour l'instant, considérons que ce sont des éléments inclus dans le HTML et qui interagissent avec le serveur ASP.NET.

Le fichier généré porte l'extension **.cshtml** (comme C# HTML) et est automatiquement situé dans le répertoire Views/Salutation. Nous allons maintenant garnir un peu notre page. Cela se fait en utilisant les balises HTML mais aussi avec des helpers HTML qui permettent de générer le code du contrôle HTML. Par exemple, ajoutons dans la balise `<body>` la mise en forme de notre application.

FIGURE 4.10. – Plan de notre application

Cette fois-ci, nous allons distinguer deux choses dans le code HTML : les éléments **statiques** que nous avons déjà vus (`<p>`, `<input>`, etc.) et les éléments **dynamiques**, qui ont une interaction avec le code serveur.

Dans notre cas, il y aura deux éléments principaux en dynamique : le champ de texte modifiable et le texte de salutation généré. Le champ de texte va associer la saisie de l'utilisateur à la clé **prenom_visiteur**, que l'on a appelée dans le contrôleur via `Request.Form`. Dans le texte de salutation, il faut afficher le **ViewData** que l'on a créé dans le contrôleur aussi.

```
ViewData["message"] = "Bonjour à toi, " + prenom;
```

Voici ce que notre page **.cshtml** donne :

```
@model BonjourMVC.Models.Visiteur

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @using (Html.BeginForm())
        {
            <h1>BonjourMVC</h1>
            <p>Comment t'appelles-tu jeune Zesteur ?</p>
            @Html.TextBox("prenom_visiteur", Model.Prenom)
            <input type="submit" value="Valider" />
        }
        <p>@ViewData["message"]</p>
    </div>
</body>
</html>
```

Nous retrouvons certains éléments que nous avons appelés dans la méthode du contrôleur. Les éléments dans le code HTML commençant par `@` sont les éléments dynamiques : c'est la syntaxe d'un moteur de vue appelé **Razor**, que nous découvrirons très bientôt.

Il y a deux **helpers** HTML dans le code : `Html.BeginForm` et `Html.TextBox`. Vous l'aurez compris, ils permettent respectivement de générer un formulaire et un champ de texte modifiable. Le champ de texte portera le nom prenom_visiteur et sera associée à la propriété Prenom de notre classe de modèle, représentée ici par la variable Model.

Après le formulaire, la balise `<p>` va contenir le message résultant. Il est temps de tester notre application maintenant !

4.6. Testons l'application

Vous connaissez la maison maintenant : pour démarrer l'application web, appuyez sur la touche F5. Le serveur va charger notre application sur localhost. Nous devrions tomber sur cette page :

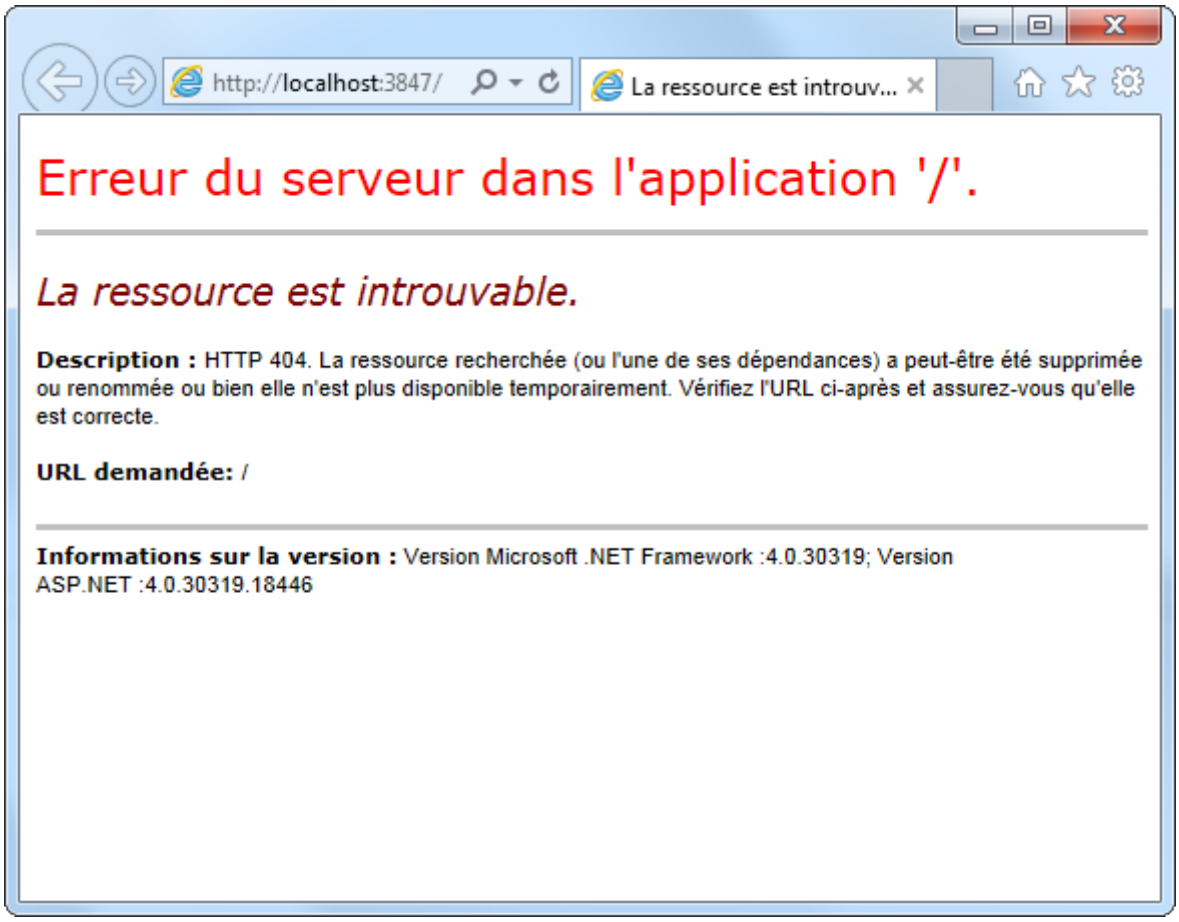


FIGURE 4.11. – Page d'accueil de notre application

Le navigateur nous affiche une page d'erreur indiquant que la page est introuvable. C'est normal, nous n'avons pas indiqué la route qui correspond à une action de notre contrôleur. Rappelez-vous, nous devons taper l'URL suivante : **http ://localhost :votre_numero_de_port/Salutation/Index** pour pouvoir appeler la méthode Index du contrôleur SalutationController.

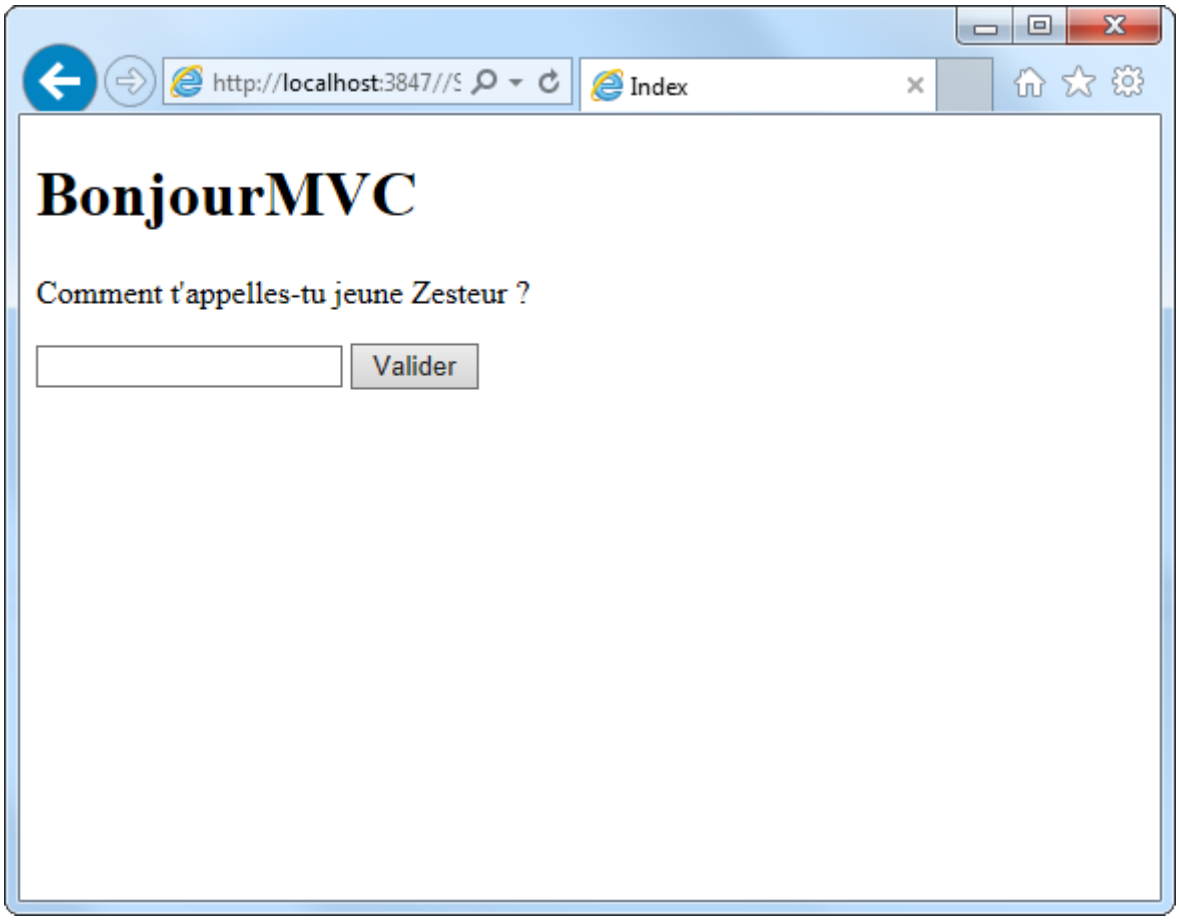
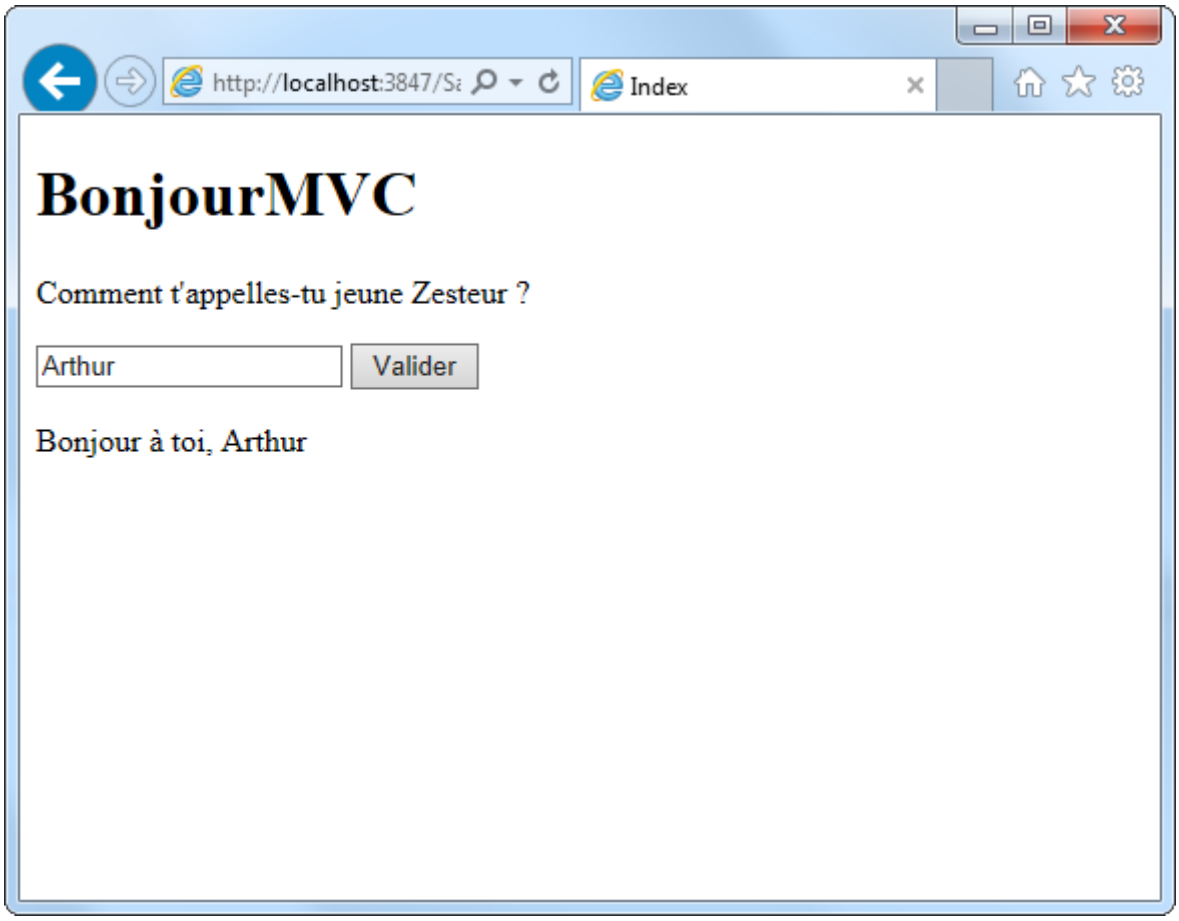


FIGURE 4.12. – Interface de notre application

Mettons notre prénom dans le champ de texte et soumettons le formulaire :



Au moment où nous soumettons le formulaire, tout le mécanisme s'enclenche et la phrase de bienvenue s'affiche. Si nous affichons le code source de la page ainsi obtenue, nous aurons quelque chose comme :

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
<form action="/Salutation/Index" method="post">
    <h1>BonjourMVC</h1>
            <p>Comment t'appelles-tu jeune Zesteur ?</p>
<input id="prenom_visiteur" name="prenom_visiteur" type="text"
    value="Arthur" />            <input type="submit"
    value="Valider" />
</form>        <p>Bonjour &#224; toi, Arthur</p>
    </div>

<!-- Visual Studio Browser Link -->
```

```
<script type="application/json"
    id="__browserLink_initializationData">
        {"appName":"Internet Explorer","requestId":"5c96bb4c6fa5470594489162026
</script>
<script type="text/javascript"
    src="http://localhost:1643/b70ba08d79874a7aa04dc5030e2c2d02/browserLink"
    async="async"></script>
<!-- End Browser Link -->

</body>
</html>
```

Les contrôles serveurs ont été transformés en HTML avec en complément, les éventuelles valeurs que nous avons mises à jour. Par exemple, le helper `Html.BeginForm` a été transformé en balise `<form>` que vous connaissez bien ou encore le helper `Html.TextBox` s'est lui transformé en `<input type="text ...>`.

Au travers de ce chapitre, nous avons pu toucher à différents éléments de la programmation en ASP.NET afin de construire notre première application. Certains éléments peuvent sembler flous, mais ce tutoriel est là pour enseigner la majeure partie de la programmation web en ASP.NET.

5. Le déroulement d'une requête : introduction aux filtres et aux routes

En ASP.NET, la notion de **filtre** et de **route** est importante pour comprendre comment va fonctionner notre application. Dans le chapitre de l'application BonjourMVC, nous avons utilisé ces deux notions, peut-être sans nous en rendre compte. Pour la suite, il est préférable de commencer à connaître les bases des filtres et des routes.

Les filtres sont des attributs personnalisés qui fournissent un moyen déclaratif pour ajouter un comportement pré et post-action aux méthodes d'action de contrôleur. Les routes, à partir d'une URL, vont déterminer quel contrôleur appeler et avec quels arguments.

5.1. Le déroulement d'une requête

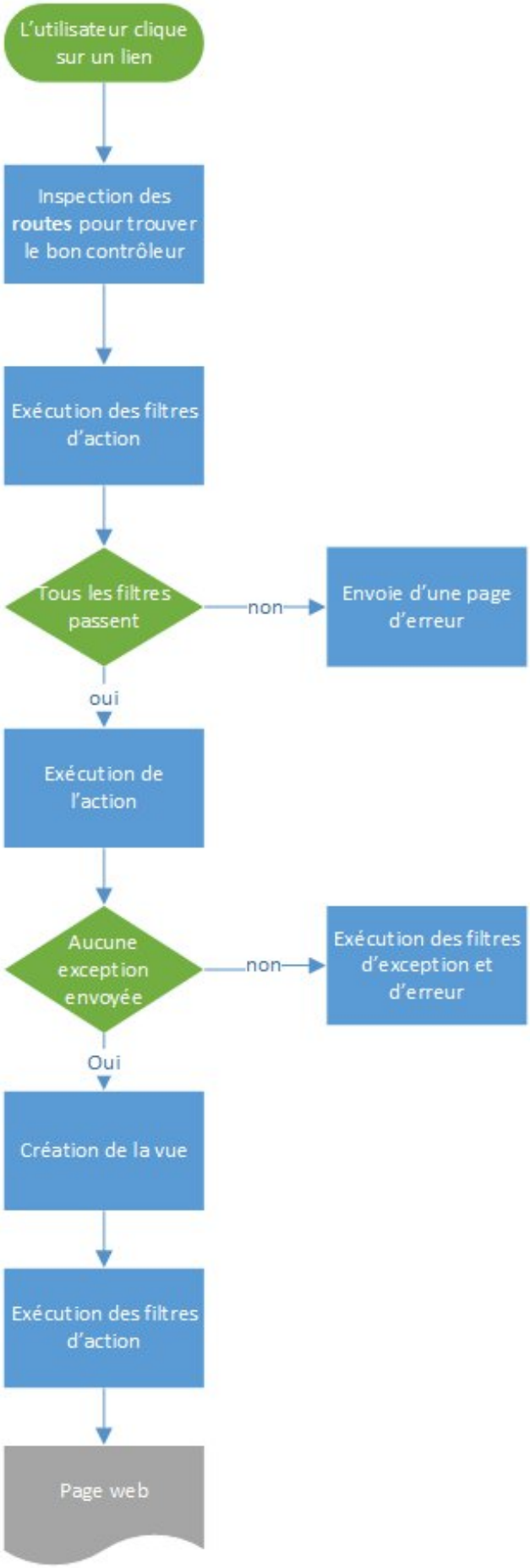
Comme nous l'avons vu dans la première partie, lorsque l'utilisateur veut obtenir une page, il envoie une *requête* au serveur.

Le premier défi du serveur sera de comprendre quel contrôleur vous désirez utiliser et quelle action vous désirez faire. Une **action** est simplement une méthode d'une classe de contrôleur. Cette étape s'appelle le **routage**[5].

Une fois que le serveur sait quelle action il doit appeler, il va regarder s'il a le droit de l'appeler : la requête va passer par des **filtres**.

Si les filtres acceptent tous la requête, l'action est appelée. À la fin de celle-ci vous allez générer une page grâce au moteur de template.

D'autres filtres vérifieront que le résultat est bon et si tout va bien, le visiteur verra sa page HTML.


L'utilisateur clique sur un lien
Inspection des routes pour trouver le bon contrôleur
Exécution des filtres d'action
Tous les filtres passent
non
Envoie d'une page d'erreur
oui
Exécution de l'action
Aucune exception envoyée
non
Exécution des filtres d'exception et d'erreur
Oui
Création de la vue
Exécution des filtres d'action
Page web

? Mais pour l'instant, on n'a jamais parlé de route ou de filtre, alors pourquoi ça marche bien ?

L'avantage de l'architecture MVC, c'est qu'elle est plutôt bien *normalisée*, du coup, quand vous avez créé votre projet, Visual Studio **savait** qu'il fallait configurer le routage de manière à ce que toute URL de la forme `nom du contrôleur/nom de l'action` soit automatiquement bien routée. Souvenez-vous du chapitre précédent avec l'exemple du BonjourMVC, nous avons appelé un contrôleur et chaque méthode de ce contrôleur renvoyait une vue. Au final, nous avions une URL de ce type : /Salutation/Index avec Salutation le nom du contrôleur et Index la méthode.

Visual Studio a donc, pour vous, généré un petit code qui se trouve dans le dossier **App_Start**, et plus précisément dans le fichier `FilterConfig.cs` et `RouteConfig.cs`.

```
1  public class RouteConfig
2  {
3      public static void RegisterRoutes(RouteCollection routes)
4      {
5          routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
6
7          routes.MapRoute(
8              name: "Default",
9              url: "{controller}/{action}/{id}",
10             defaults: new { controller = "Home", action = "Index",
                   id = UrlParameter.Optional }
11         );
12     }
13 }
```

Listing 1 – RouteConfig.cs

```
1  public class FilterConfig
2  {
3      public static void RegisterGlobalFilters(GlobalFilterCollection
           filters)
4      {
5          filters.Add(new HandleErrorAttribute());
6      }
7  }
```

Listing 2 – FilterConfig.cs

Dans ces deux fichiers, vous aurez les conventions de nommage pour les routes et les filtres qui s'appliquent dans **toute votre application**.

? Et si je veux faire une route spéciale juste à un endroit ?

La partie suivante est faite pour vous.

5.2. Les routes par défaut

Revenons sur l'instruction qui nous permet de créer une route :

```
routes.MapRoute(
     name: "Default",
     url: "{controller}/{action}/{id}",
     defaults: new { controller = "Home", action = "Index", id =
         UrlParameter.Optional }
);
```

Comme il s'agit d'une route globale, on lui donne un nom : "Default".
La partie intéressante vient avec la partie "url". Il vous dit que par défaut une URL doit être composée :

- du nom du contrôleur ;
- du nom de l'action ;
- d'un id.

`{controller}` et `{action}` sont des mots-clefs qui sont immédiatement reconnus par le routeur.
Par contre "id" est un paramètre qui est propre à notre application (UrlParameter).
Dans la ligne qui suit, on vous informe qu'il est optionnel.

? Pourquoi ils ont mis "id" s'il est optionnel, et puis à quoi sert-il ?

Souvenez-vous, je vous ai dit dès le départ que la grande force de MVC, c'est que c'était normalisé.
En fait vous faites cinq types d'actions le plus souvent :

- obtenir une liste ;
- créer un nouvel objet ;
- détailler un objet de la liste ;
- modifier cet objet ;
- supprimer un objet.

Dans les deux premiers cas, vous avez juste besoin de donner le nom de l'action (lister ou créer).
Par contre dans le second cas, vous avez besoin de retrouver un objet en particulier. Le plus souvent, pour cela, vous allez utiliser un **identifiant**. Pour aller plus vite, on écrit "id".
Prenons un exemple pour notre blog : nos visiteurs voudront avoir la liste de nos articles, puis pourquoi pas ne voir qu'un seul article en particulier. Ce qui équivaudrait à `/articles/liste` et `/articles/detail/5` (pour le 5ème article).
Pour les auteurs d'article, nous avons la même chose du côté de l'édition : `/articles/creer` et `/articles/modifier/5`.
Dans notre exemple, l'identifiant, c'est le numéro.

5. à ne pas confondre avec le routage réseau, qui n'a rien à voir.

Mais aujourd'hui, il est important d'avoir des URL qui montrent le titre de l'article, comme ça les gens et les *moteurs de recherche* comprennent mieux de quoi une page parle. On va donc donner un paramètre supplémentaire : le titre adapté (pas de caractères accentués, pas d'espace), on appelle ça un `slug`.
Si on avait fait une URL par défaut sur le même modèle, elle aurait été : `{controller}/{action}/{id}/{slug}`[6]
Et le code aurait donné :

```
routes.MapRoute(
     name: "Default",
     url: "{controller}/{action}/{id}/{slug}",
     defaults: new { controller = "Home", action="Index", id =
         UrlParameter.Optional, slug = UrlParameter.Optional }
);
```

Et pour afficher un tutoriel on aurait dû faire une action telle que :

```
//exemple d'url = /Article/Voir/1/slug
public ActionResult Voir(int id, string slug)
{
    ViewBag.slug = slug;
    return View();
}
```

Dans les routes par défaut, le paramètre "Action" est nécessaire.

Comme notre but est de faire un blog, il sera peut être intéressant de donner la possibilité aux gens de nous suivre via un flux RSS . Pour cela nous allons créer notre propre "url par défaut" qui donnera le prefix "flux-rss" et qui redirigera vers RSSFeedController. Nous allons proposer au gens de demander un nombre d'articles à afficher. Par défaut cela sera 5.
Je vous laisse le faire, la correction est masquée.

```
routes.MapRoute(
     name: "Flux",
     url: "flux-rss/{action}/{article_number}",
     defaults: new { controller = "RSSFeed", action = "Index",
         article_number = 5 }
);
```

Listing 3 – Correction

Nous verrons plus tard comment contourner le comportement "par défaut". Cela n'est pas nécessaire pour construire notre premier site.

6. Le mot "slug" est utilisé pour décrire une URL "lisible" telle que celle de ce tutoriel. Nous verrons plus tard comment en créer un.

5.3. Les filtres

À plusieurs étapes du rendu, vous pouvez vous rendre compte que des filtres sont placés : ils sont faciles à reconnaître, car ils sont placés au dessus des méthodes, commençant par '[' et finissant par ']'. Un exemple que nous avons déjà vu :

```
1 [HttpPost]
```

Ces filtres ont pour but d'automatiser certains traitements et en cas d'échec de ces derniers, ils redirigent l'utilisateur vers une page d'erreur.
Les filtres sont des attributs qui héritent de `FilterAttribute`. Comme ce sont des attributs, pour spécifier un filtre, il faut les placer au-dessus du nom de la méthode ou de la classe.
Lorsque vous spécifiez un filtre sur la classe, il s'appliquera à toutes les méthodes.
La plupart du temps, vous utiliserez les filtres [RequireRole], [Authorize], [AllowAnonymous].
Ces filtres ont la particularité de gérer les cas d'autorisation. Si Authorize ou RequireRole échouent, ils vous envoient une page avec pour erreur "403 : not authorized".
Il est **fortement conseillé** de mettre `[Authorize]` sur **toutes les classes** de contrôleur puis de spécifier les méthodes qui sont accessibles publiquement à l'aide de `[AllowAnonymous]`.
Prenons l'exemple de notre blog, créons un contrôleur dont le but est de gérer les articles.
Une bonne pratique pour bien sécuriser votre blog sera :

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.Web.Mvc;
6
7  namespace mvc_vide.Controllers
8  {
9      [Authorize]
10     public class ArticleController : Controller
11     {
12
13         // GET: Article
14         [AllowAnonymous]
15         public ActionResult Index()
16         {
17             return View();
18         }
19
20         // GET: Article/Details/5
21         [AllowAnonymous]
22         public ActionResult Details(int id)
23         {
24             return View();
25         }
26
27         // GET: Article/Create
```

```
28          public ActionResult Create()
29          {
30              return View();
31          }
32
33          // POST: Article/Create
34          [HttpPost]
35          public ActionResult Create(FormCollection collection)
36          {
37              try
38              {
39                  // TODO: Add insert logic here
40
41                  return RedirectToAction("Index");
42              }
43              catch
44              {
45                  return View();
46              }
47          }
48
49          // GET: Article/Edit/5
50          public ActionResult Edit(int id)
51          {
52              return View();
53          }
54
55          // POST: Article/Edit/5
56          [HttpPost]
57          public ActionResult Edit(int id, FormCollection collection)
58          {
59              try
60              {
61                  // TODO: Add update logic here
62
63                  return RedirectToAction("Index");
64              }
65              catch
66              {
67                  return View();
68              }
69          }
70
71          // GET: Article/Delete/5
72          public ActionResult Delete(int id)
73          {
74              return View();
75          }
76
77          // POST: Article/Delete/5
```

```
78          [HttpPost]
79          public ActionResult Delete(int id, FormCollection
                collection)
80          {
81              try
82              {
83                  // TODO: Add delete logic here
84
85                  return RedirectToAction("Index");
86              }
87              catch
88              {
89                  return View();
90              }
91          }
92      }
93  }
```

Listing 4 – Restreindre l'accès par défaut

Vous pouvez, bien entendu, ajouter plusieurs filtres à une même méthode. Dès lors, il est fortement conseillé de spécifier l'ordre d'exécution afin que tout soit prévisible.
Pour faire cela, il suffit de donner un paramètre nommé au filtre. Si vous désirez que l'autorisation soit exécutée en premier, faites : `[Authorize(Order:1)]`.

Ce chapitre est terminé, avec deux notions importantes et assez simples à assimiler :

- les filtres, qui permettent de spécifier les conditions à respecter au début ou à la fin d'un contrôleur ou d'une action ;
- les routes, qui quant à elles décrivent les demandes entrantes issues de navigateur via l'URL, pour des actions de contrôleur particulières.

Nous avons beaucoup parlé de **contrôleur** dans ce chapitre, il serait temps d'en savoir un peu plus sur eux. Et ça tombe bien, car c'est au programme du prochain chapitre !

6. Montrer des informations à l'utilisateur

Dans une application MVC web, le cœur de notre application, c'est le contrôleur. Il a pour rôle d'appliquer la logique de notre application et d'afficher les résultats.
La logique, c'est l'ensemble des étapes que vous voulez réaliser pour que votre site fonctionne. Principalement, le contrôleur va converser avec votre modèle.
Le modèle, c'est un ensemble de classes qui représentent les données manipulées par votre application.
Très souvent, le modèle est accompagné par une couche appelée DAL, ou "couche d'accès aux données". C'est cette couche qui accèdera à votre base de données par exemple. Elle fera l'objet de la partie IV.
Ce qu'il faut comprendre, c'est qu'a priori, le contrôleur ne suppose pas l'utilisation d'une base de données.
Vous pouvez faire pleins de calculs uniquement à partir de données contenues dans les sessions, les formulaires, ou encore les URL.

6.1. Un contrôleur de base

Souvenons-nous de comment fonctionne MVC : le contrôleur *traite des données* puis *génère* une vue ou bien délègue cette génération à un outil dédié.
Explorons un peu ces contrôleurs pour déterminer comment ils fonctionnent.
Tout d'abord, il faut bien comprendre qu'un contrôleur est un objet *comme les autres*. La seule contrainte, c'est qu'il hérite de l'objet `Controller`.
Visual studio possède un certain nombre de guides pour créer un contrôleur. Comme nous sommes encore en train d'apprendre, nous n'allons **pas** utiliser ces guides car ils génèrent beaucoup de code qui ne sont pas adaptés à l'apprentissage pas à pas.

> Assurez vous d'avoir créé un projet "Blog" avec le modèle `MVC` et l'authentification par défaut.

Pour créer un contrôleur, nous allons faire un clic droit sur le dossier `Controllers` puis `ajouter->Contrôleur`.
Là : sélectionnez "Contrôleur Vide", nommez-le `ArticleController`.

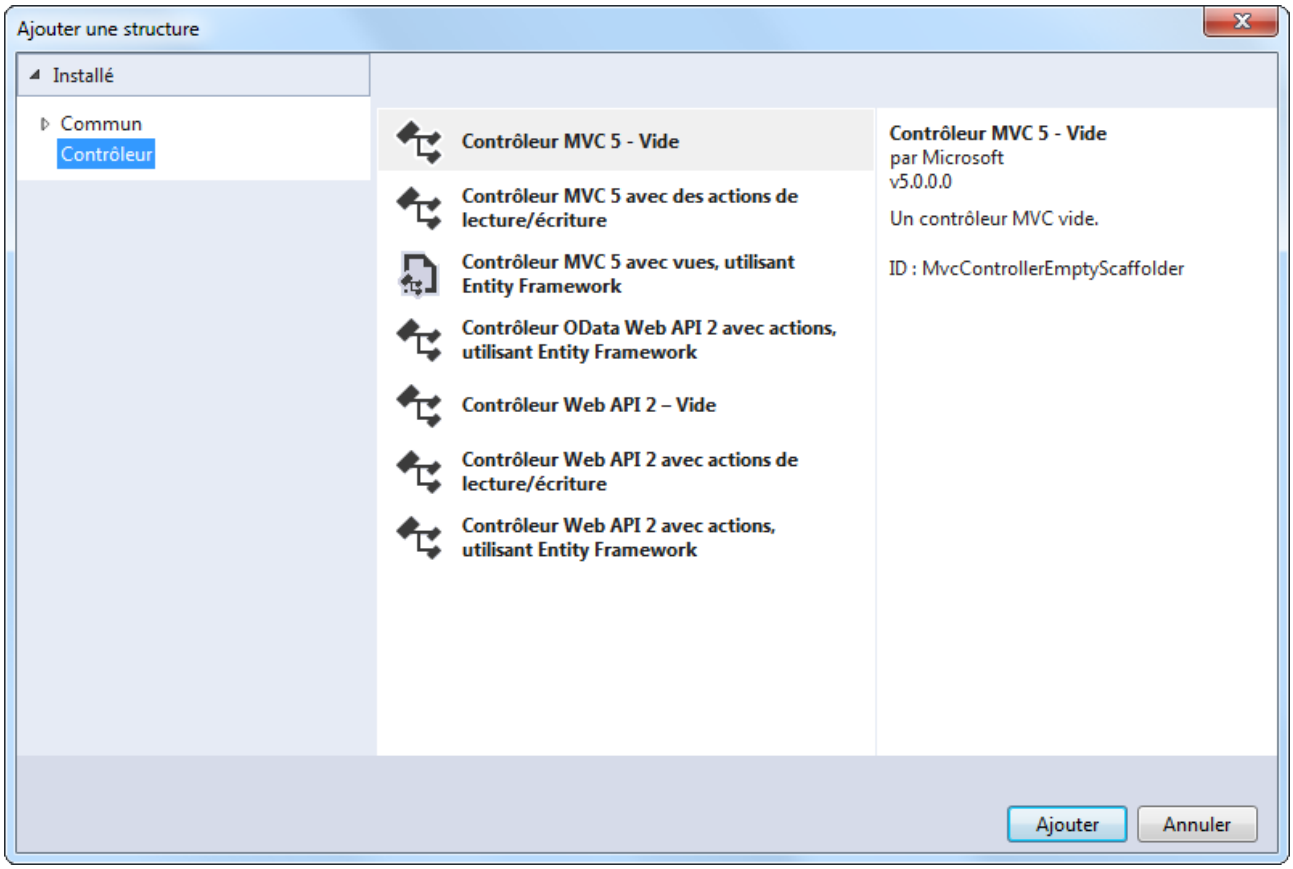


FIGURE 6.1. – Ajout d'un nouveau contrôleur

6.1.1. Qu'est-ce qu'une page ?

Lorsque vous utilisez MVC, une page correspond en fait à une `action` du contrôleur. Une action, c'est une méthode qui a deux contraintes :

— elle doit être publique ;
— elle doit retourner un objet `ActionResult`.

Et vous savez quoi ? Il existe une méthode qui a déjà été codée pour vous dans `Controller`, qui s'appelle `View` et qui vous permet de retourner un `ActionResult` qui contiendra tout le code HTML de votre page.
La seule condition : il faut que dans le dossier `Views`, il y ait un autre dossier qui porte le même nom que votre contrôleur (dans notre exemple `Article`) et qu'ensuite, il y ait un fichier `nom_de_votre_action.cshtml`. Par exemple, pour l'action `Index`, il faut un fichier `Index.cshtml`.
Nous verrons plus tard comment utiliser de manière avancée Razor. Pour l'instant, vous pouvez ne mettre que du code HTML dans cette page et ça sera parfait.

6.1.2. Page statique

Les pages statiques ne seront pas les pages les plus nombreuses de votre application, néanmoins, elles peuvent exister. Comme elles sont statiques, une bonne pratique consiste à indiquer au serveur qu'il ne doit pas refaire tout le déroulement de la requête mais aller tout de suite chercher la page déjà créée et la renvoyer à l'utilisateur.
Cela se fait grâce à un filtre appelé OutputCache .

```
// GET: MaPageStatique
        [OutputCache(Duration = 10000000, Order = 1, NoStore =
    false)]
        public ActionResult MaPageStatique()
        {
            return View();
        }
```

Listing 5 – Une page statique

6.1.3. Page dynamique

Les pages dynamiques sont utilisées quand vous voulez mettre en place une page qui change en fonction de l'utilisateur et de l'ancienneté du site.
Prenons en exemple Zeste de Savoir. Si vous êtes un utilisateur non connecté (on dit aussi *anonyme*), le haut de page vous propose de vous inscrire ou de vous connecter :

FIGURE 6.2. – Bannière pour les anonymes

Lorsque vous êtes connectés, vous allez par contre pouvoir gérer votre compte.

FIGURE 6.3. – Bannière pour les utilisateurs enregistrés

Mais le mieux, c'est que selon que vous soyez utilisateur lambda ou administrateur, vous n'aurez pas accès aux mêmes liens.

FIGURE 6.4. – Les menus utilisateurs et administrateurs sont différents

Vous pouvez - comme nous l'avons vu dans notre `Hello Word` - ajouter des données *dynamiquement* de deux manières :

— en donnant à la vue un *Modèle*, dans notre cas ça sera des `Article` ou des `List<Article>` selon notre besoin ;
— en ajoutant des clefs au `ViewBag` ou au `ViewData`

```
1        public ActionResult Index()
2        {
3            ViewBag.Bla = "bla"; //équivaut à ViewData["bla"] =
                 "bla";
4            return View();
5        }
```

Le `ViewBag` et le `ViewData` sont des propriétés qui font la même chose. Il est conseillé d'utiliser le `ViewBag` tout simplement parce qu'il respecte plus la notion d'objet (`ViewBag.UnePropriete`), de plus des erreurs de nommage peuvent arriver avec le `ViewData`.

6.2. Introduction à Razor

Les vues sont le point le plus important pour l'utilisateur de votre application. En effet, un visiteur se fiche bien de l'organisation de la logique de notre application web (bien que ce soit

important pour nous, développeur) : le seul composant qu'il verra et pourra juger est la vue. Les vues sont donc essentielles, elles doivent être agréables et ergonomiques pour le visiteur et d'un autre côté maintenables et lisibles pour le développeur.
Évidemment, nous ne vous conseillons pas de mettre de côté l'architecture et l'organisation de la logique des contrôleurs. Le code doit rester maintenable et bien pensé.

6.2.1. Bonjour Razor !

ASP.NET MVC utilise un moteur de vues appelé Razor. Comme son nom l'indique, un moteur de vues permet la création de vues (de pages HTML). Le moteur de vues va communiquer avec le code « serveur » afin de générer une réponse au format HTML. Le moteur de templates Razor est apparu avec le version 3 de ASP.NET MVC, c'est le moteur de templates par défaut avec ASP.NET MVC. Razor utilise sa propre syntaxe, l'avantage c'est que celle-ci a pour vocation d'être simple et de faciliter la lecture des pages de vues pour le développeur. Nous retrouvons une fluidification entre code HTML et code exécutable.
Pour une application utilisant le C#, les vues Razor porterons l'extension **.cshtml**, mais si vous utilisez VB.NET, l'extension **.vbhtml** sera utilisée.
Nous considèrerons même cela comme un moteur de conception et de composition de gabarits (**Template** en anglais).

Un gabarit (ou template) est un modèle de document (HTML) amené à être modifié. Une application web dynamique utilise les gabarits pour générer une page HTML (afficher votre pseudonyme, lister les derniers billets d'un blog, etc.).

Le moteur de templates Razor (et nous utiliserons dorénavant ce terme) offre un pseudo-langage, avec une syntaxe propre à lui pour permettre la séparation du code C# du code HTML tout en conservant une interaction avec la logique serveur (le HTML étant un langage de description côté client).

6.2.2. Syntaxe de Razor

Découvrons maintenant les rudiments de la syntaxe que nous offre Razor. Nous distinguons deux types de syntaxe utilisables avec Razor :

- les instructions uniques ;
- les blocs de code.

Le code Razor s'écrit directement dans le code HTML.

Nous parlons de code Razor, mais en vérité cela reste du code C#. Razor propose une syntaxe permettant de s'inclure directement dans le balisage HTML.

####### Les instructions uniques
Une instruction unique se fait avec le symbole arobase "@". Par exemple pour afficher la date courante :

```
1 <p>Bienvenue sur mon blog, nous sommes le @DateTime.Now</p>
```

Avec, vous l'aurez reconnu, l'objet DateTime donnant des informations sur le temps. Une instruction unique va directement générer du contenu au sein du code HTML, elle tient sur une

ligne.
####### Les blocs de code
Le second type de syntaxe Razor est le bloc de code. Comme son nom l'indique, un bloc de code est une section qui va contenir exclusivement du code Razor, sans HTML. Comme tout bloc en programmation, celui-ci est délimité par des symboles marquant le début et la fin du bloc d'instructions.
Un bloc de code est délimité par le symbole arobase et d'une paire d'accolades : le début du bloc, nous trouvons les signes "`@{`" et le signe "`}`" à la fin du bloc.
Tout ce qui se situe entre ces délimiteurs doivent respecter les règles du langage de programmation utilisé, c'est-à-dire le C#. Une page de vue écrite avec Razor respecte les mêmes règles qu'un fichier source C#.

6.2.2.0.1. Exemple Dans notre application de blog, créons un nouveau contrôleur vide. Nous l'appellerons **DemoController**.

```
public class DemoController : Controller
{
    //
    // GET: /Demo/
    public ActionResult Index()
    {
        return View();
    }
}
```

À l'aide du clic droit, ajoutons une vue appelée **Index**.

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
    </div>
</body>
</html>
```

En haut du code HTML, nous trouvons déjà une instruction Razor définie dans un bloc de code `@{ }`.

```
@{
    Layout = null;
}
```

Layout est une variable initialisé à `null` car nous n'utilisons pas de mise en page particulière. Nous reviendrons sur les mises en pages très prochainement. Dans la balise `<div>` ajoutons le code suivant :

```
<body>
    <div>
        @{
            String nom = "Clem";
            String message = "Salut " + nom;
        }

        <p>@message</p>
    </div>
</body>
```

Nous créons une variable contenant "Clem" et une autre concaténant la valeur de la variable `nom` avec "Salut". Ensuite, nous demandons d'accéder à la valeur de la variable dans le paragraphe `<p></p>`. Ce qui donne :

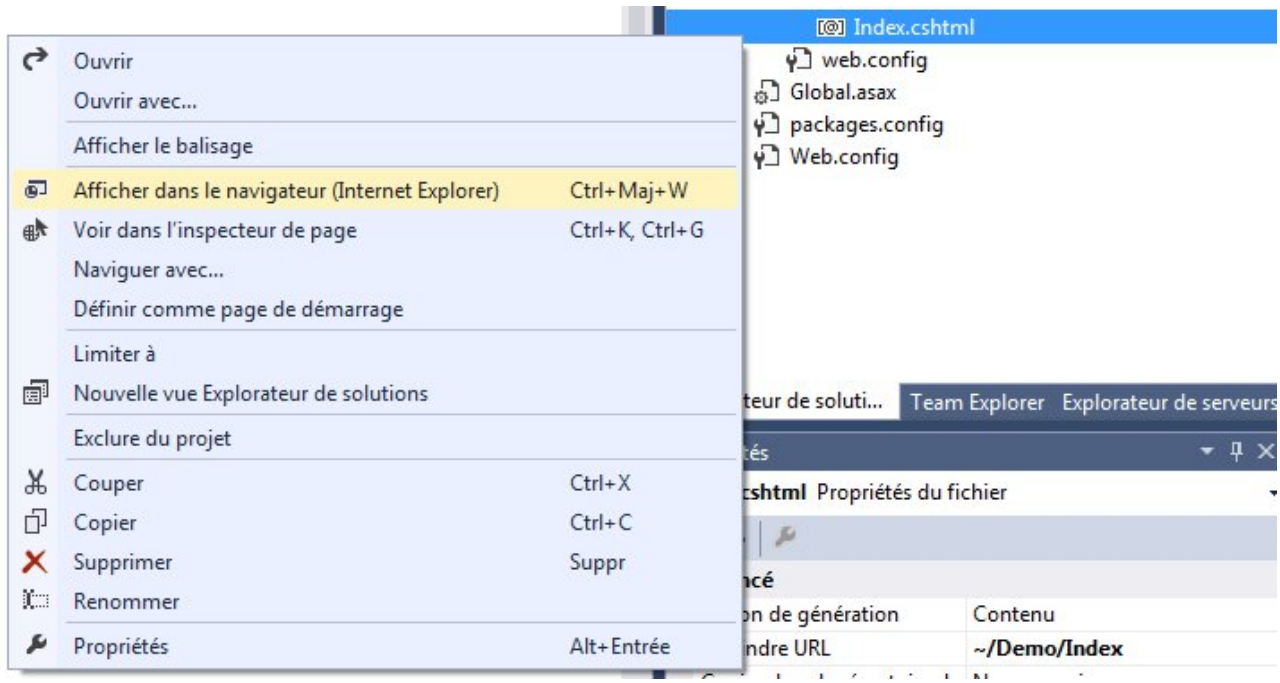


FIGURE 6.5. – Afficher la vue dans le navigateur

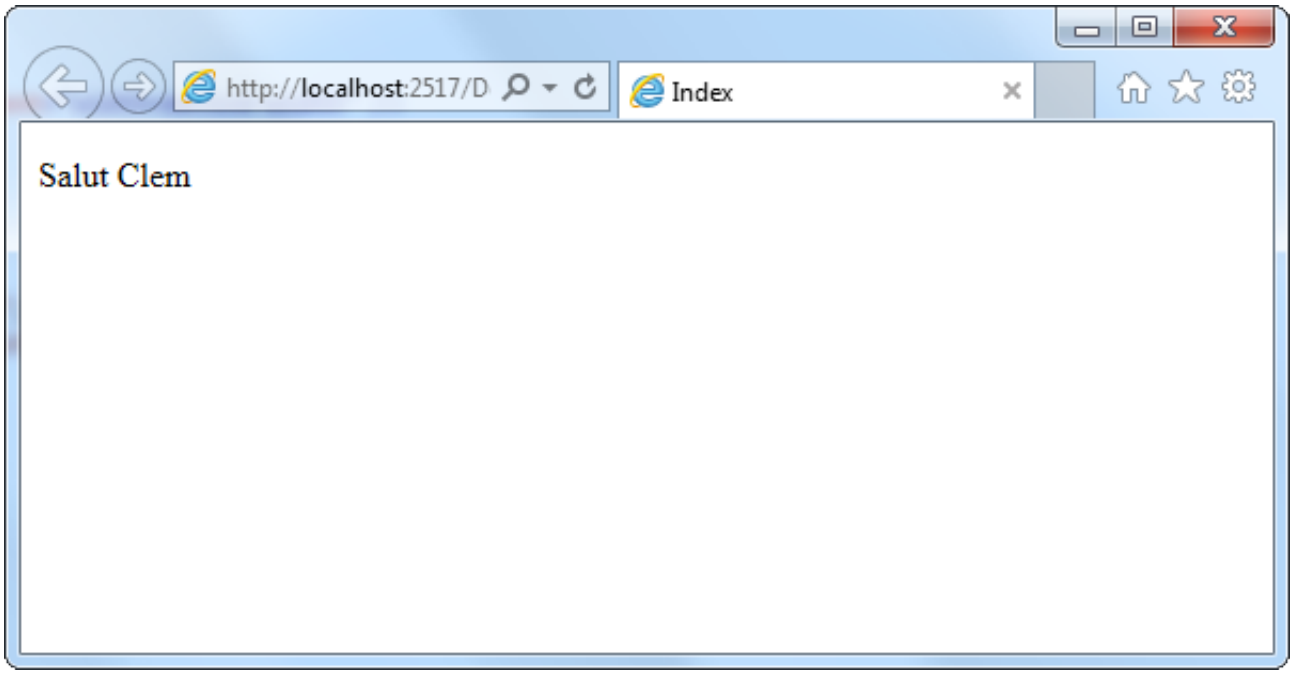


FIGURE 6.6. – Résultat de la vue

6.3. Le layout de base

Nous allons maintenant créer notre premier **layout** avec Razor.

? C'est quoi un layout ?

Layout est un mot anglais signifiant **mise en page**. Supposez que vous développez votre propre application avec ASP.NET et que vous souhaitez conserver un style cohérent pour chaque page de votre application. Prenons quelques pages de Zeste de Savoir :

FIGURE 6.7. – Page d'accueil

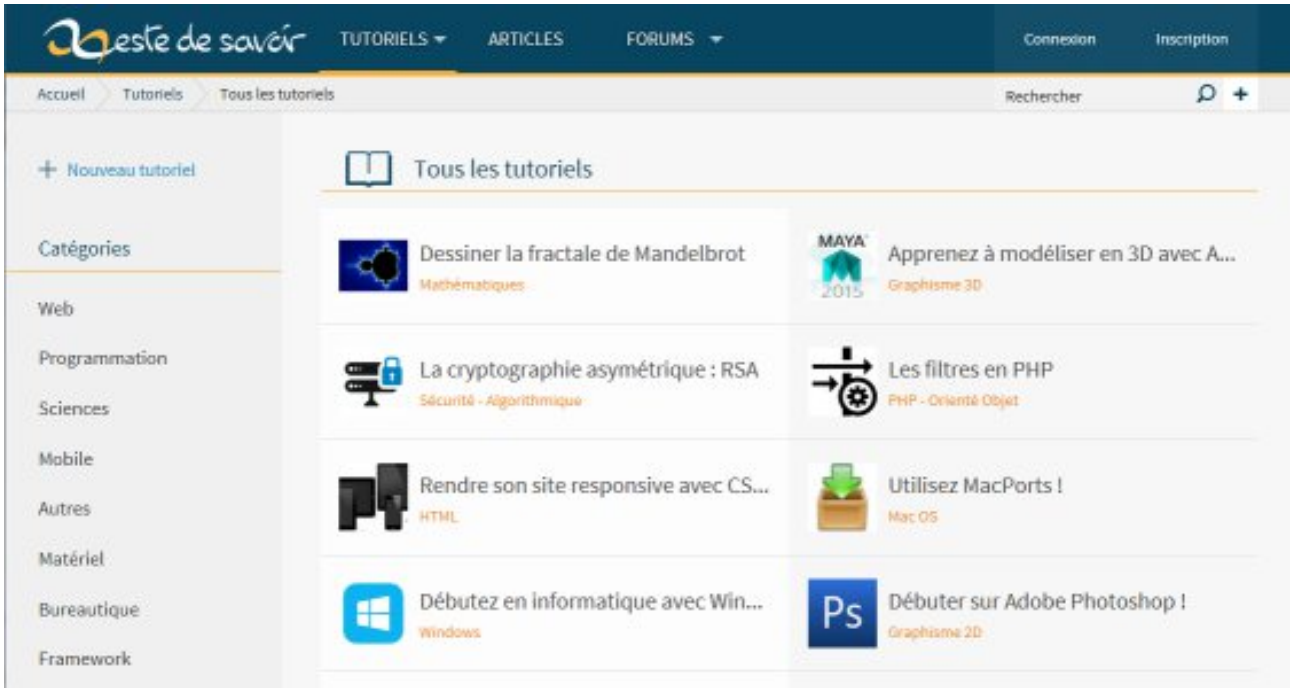


FIGURE 6.8. – Page des tutoriels

FIGURE 6.9. – Pages des articles

Nous remarquons qu'à chaque page, il existe une mise en page semblable : l'en-tête, le pied de page et la barre de navigation. Tout ce qui change est le **contenu** (qui se situe au centre). Il y a deux solutions pour faire cela :

— réécrire le code de l'en-tête, de la barre de navigation et du pied de page sur toutes les pages ;
— définir un modèle commun pour l'application, que chaque page pourra ou non utiliser.

La deuxième solution semble la meilleure et la moins fatigante. C'est justement le principe d'un layout. D'ailleurs, souvenez-vous de nos premières vues, en haut du fichier **cshtml** il y avait un bloc Razor :

```
1 @{
2     Layout = null;
3 }
```

C'est cette ligne qui indique si un layout est ou non utilisé.
Place à la pratique ! Nous allons créer notre premier layout. Pour cela, dans notre projet Blog, ajoutez un dossier **Content**. Ce dossier va contenir les fichiers de styles (CSS). Ensuite, dans ce dossier Content, ajoutez un fichier CSS que nous nommerons **Blog.css**.
Ensuite, ajoutez un dossier **Shared** dans le dossier Views. Par convention, en ASP.NET, le dossier Shared situé dans Views contient tous les layouts pour votre application. Il est destiné à contenir les éléments partagés entre les différentes vues.
Pour terminer, ajoutez un fichier **_Layout.cshtml** dans le dossier Shared de notre Blog. Ce fichier _Layout.cshtml va représenter notre fichier de mise en page.

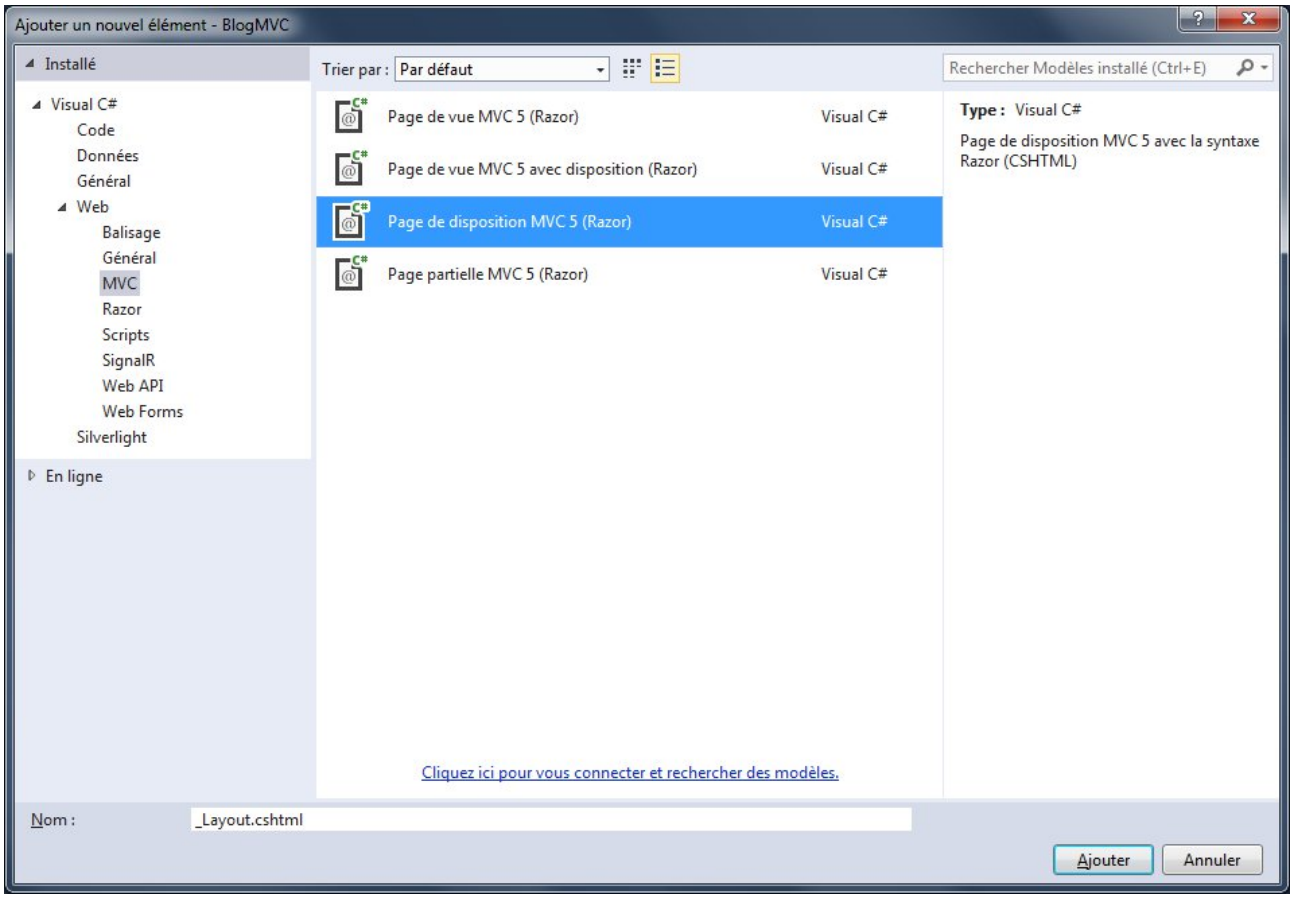


FIGURE 6.10. – Ajoutez une page de disposition(layout)

Notre solution doit ressembler à ceci maintenant :

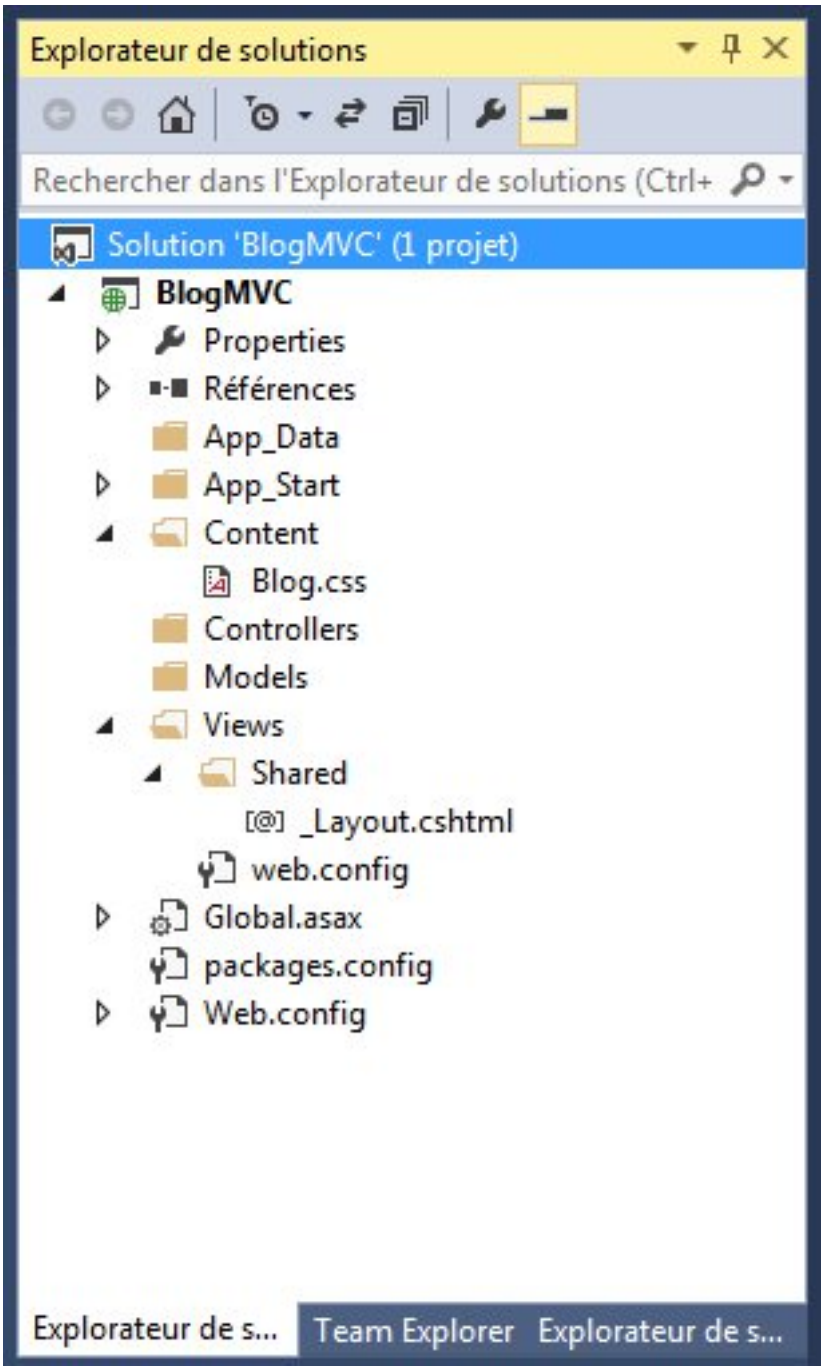


FIGURE 6.11. – Solutions après ajout des dossiers et des fichiers

Notre fichier _Layout.cshtml contient déjà un peu de code :

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
```

```
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Deux éléments vont porter notre attention : `@ViewBag.Title` et `@RenderBody`. Une page de disposition n'est pas un fichier HTML comme un autre, outre le fait qu'il contient du code Razor, cette page ne sera jamais affichée directement à l'utilisateur ! Ce sont les vues qui vont l'utiliser et pas l'inverse. Lorsque nous créerons une vue qui utilisera notre layout, cette dernière va transmettre des informations au layout : le titre de la page de vue (`@ViewBag.Title`) et le contenu de la vue (`@RenderBody`).
Cette présentation est un peu vide, nous allons la garnir un petit peu :

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title - Blog</title>
    <link rel="stylesheet"
        href="@Url.Content("~/Content/Blog.css")" type="text/css"
        />
</head>
<body>
    <header>
        <h1>Mon Blog ASP.NET MVC</h1>
        <ul id="navliste">
            <li class="premier"><a href="/Accueil/"
                id="courant">Accueil</a></li>
            <li><a href="/Accueil/About/">A Propos</a></li>
        </ul>
    </header>
    <div id="corps">
        @RenderBody()
    </div>
    <footer>
        <p>@DateTime.Now.Year - Mon Blog MVC</p>
    </footer>
</body>
</html>
```

Voici le contenu du fichier **Blog.css** :

Contenu masqué n°1

Nous avons notre page de layout. Maintenant, il faudrait placer une vue (par exemple celle de la page d'accueil) pour admirer le résultat. C'est ce que nous allons faire !
Pour qu'une page de vue utilise un layout, il y a deux solutions : soit placer le nom du layout

comme valeur de la variable Layout en haut de la vue, soit ajouter une page de vue qui va automatiquement ajouter le layout à chaque vue de notre application (cela signifie que nous n'aurons pas besoin d'indiquer le layout à chaque page).
Comme nous sommes fainéants et qu'ajouter `Layout = "/Views/Shared/_Layout.cshtml"` à chaque page est trop répétitif, nous allons utiliser la deuxième solution. Nous allons donc créer un fichier **_ViewStart**. Faites un clic droit sur le répertoire Views > Ajouter > **Nouvel élément**.

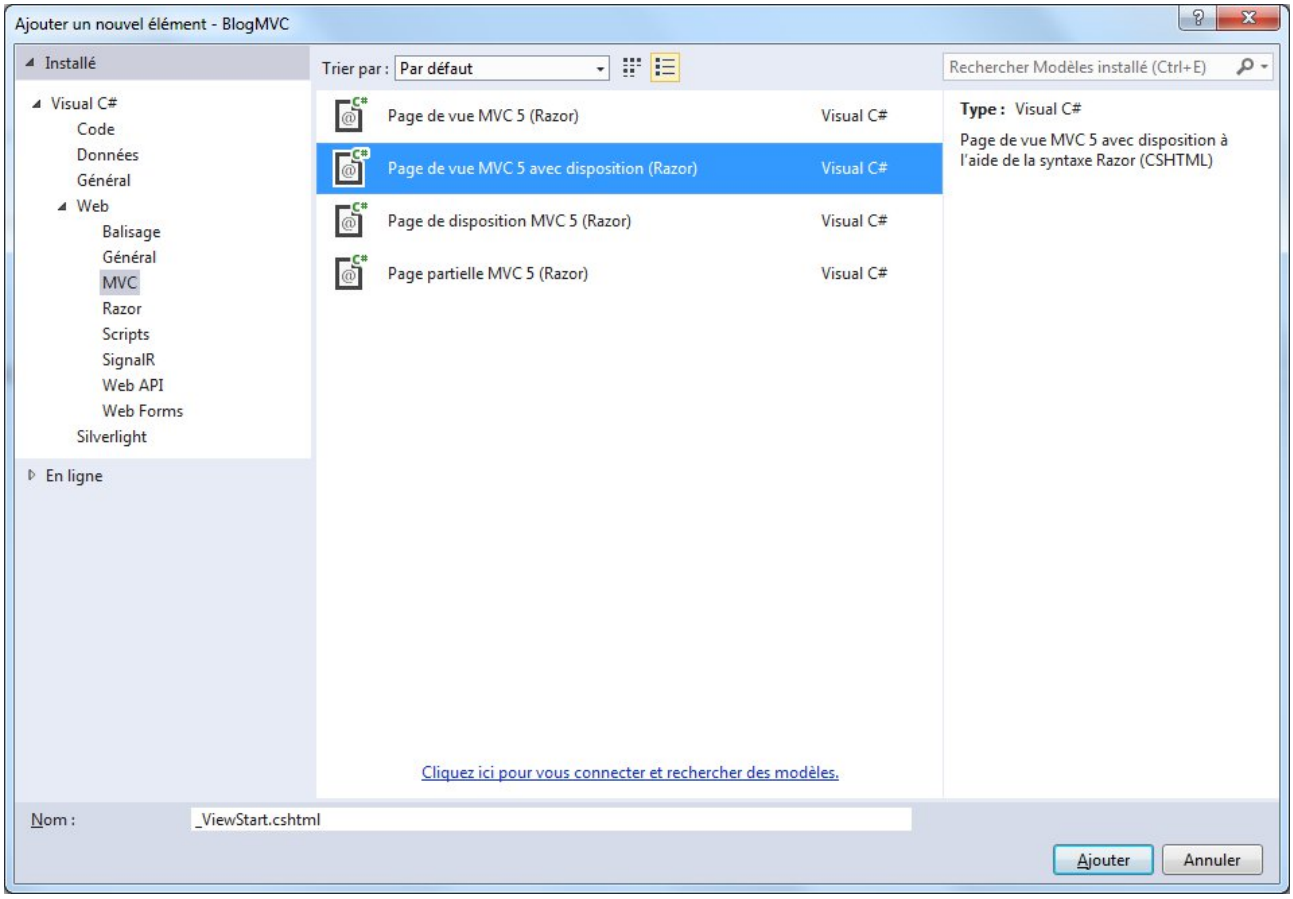


FIGURE 6.12. – Ajout du fichier _ViewStart

Sélectionnez Page de vue MVC avec disposition et nommez-là **_ViewStart.cshtml**. Ce fichier va simplement contenir :

```
1 @{
2     Layout = "~/Views/Shared/_Layout.cshtml";
3 }
```

Ce fichier de disposition doit se nommer _ViewStart. C'est une convention.

6.3.0.0.1. Ajout du contrôleur Ajoutons le contrôleur de notre page d'accueil. Ce contrôleur va contenir deux méthodes : **Index** et **About** (chacune désignant une page de l'application). C'est pour cela que nous vous avons fait placer les URL suivantes dans la barre de navigation :

```
<li class="premier"><a href="/Accueil/"
    id="courant">Accueil</a></li>
<li><a href="/Accueil/About/">A Propos</a></li>
```

Nous appellerons le contrôleur **AccueilController**.

```
namespace BlogMVC.Controllers
{
    public class AccueilController : Controller
    {
        //
        // GET: /Accueil/
        public ActionResult Index()
        {
            return View();
        }

        //
        // GET: /About/
        public ActionResult About()
        {
            return View();
        }
    }
}
```

Les méthodes ne contiennent, pour l'instant, rien de spécial. Ensuite, nous allons ajouter la vue correspondant à l'index. Faites un clic droit sur la méthode Index > **Ajouter une vue**...

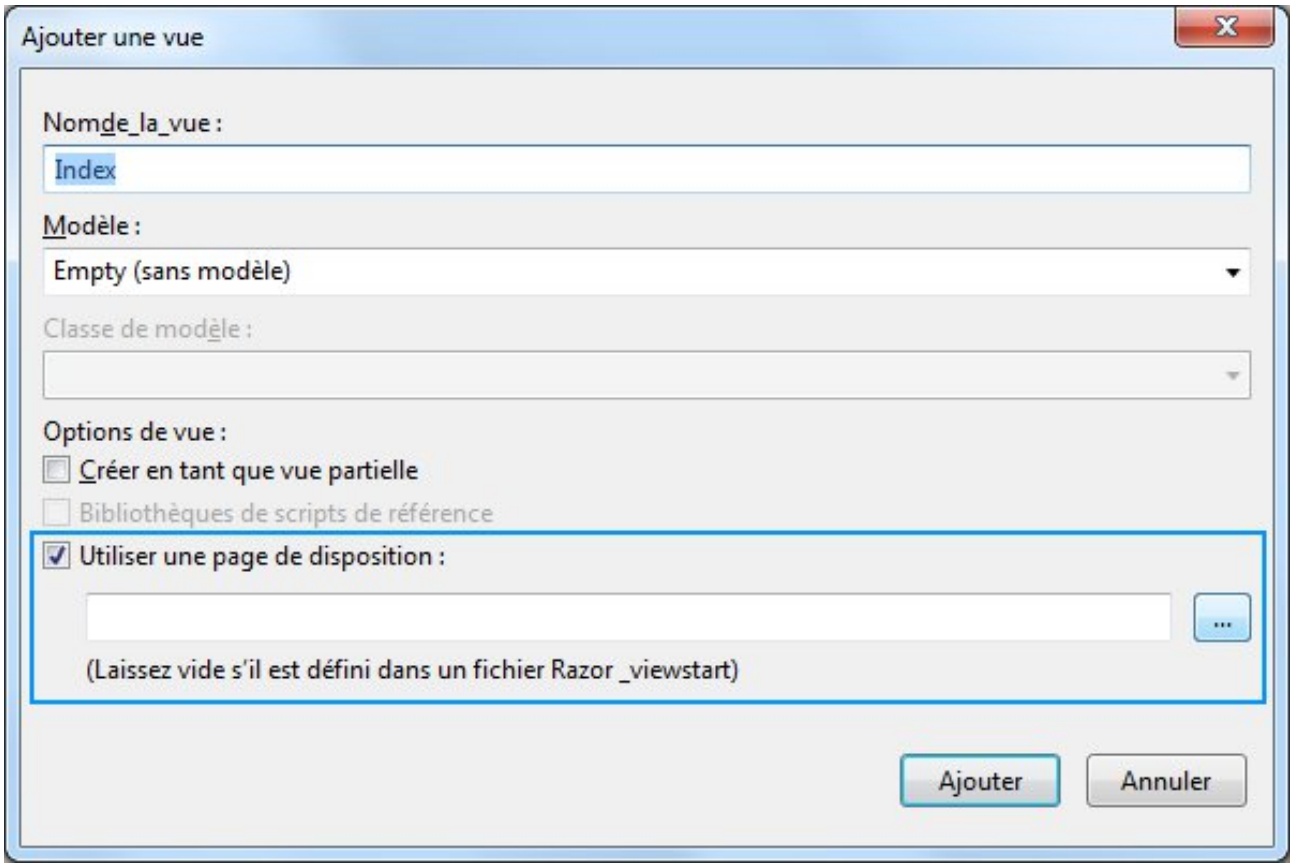


FIGURE 6.13. – Ajout de la vue Index du blog

Prenez soin de cocher la case "Utiliser une page de disposition". L'avantage de notre fichier ViewStart, c'est que nous n'avons pas besoin d'indiquer le layout. La vue Index est générée :

```
1 @{
2     ViewBag.Title = "Index";
3 }
4
5 <h2>Index</h2>
```

Ce sera suffisant. Testons notre application :

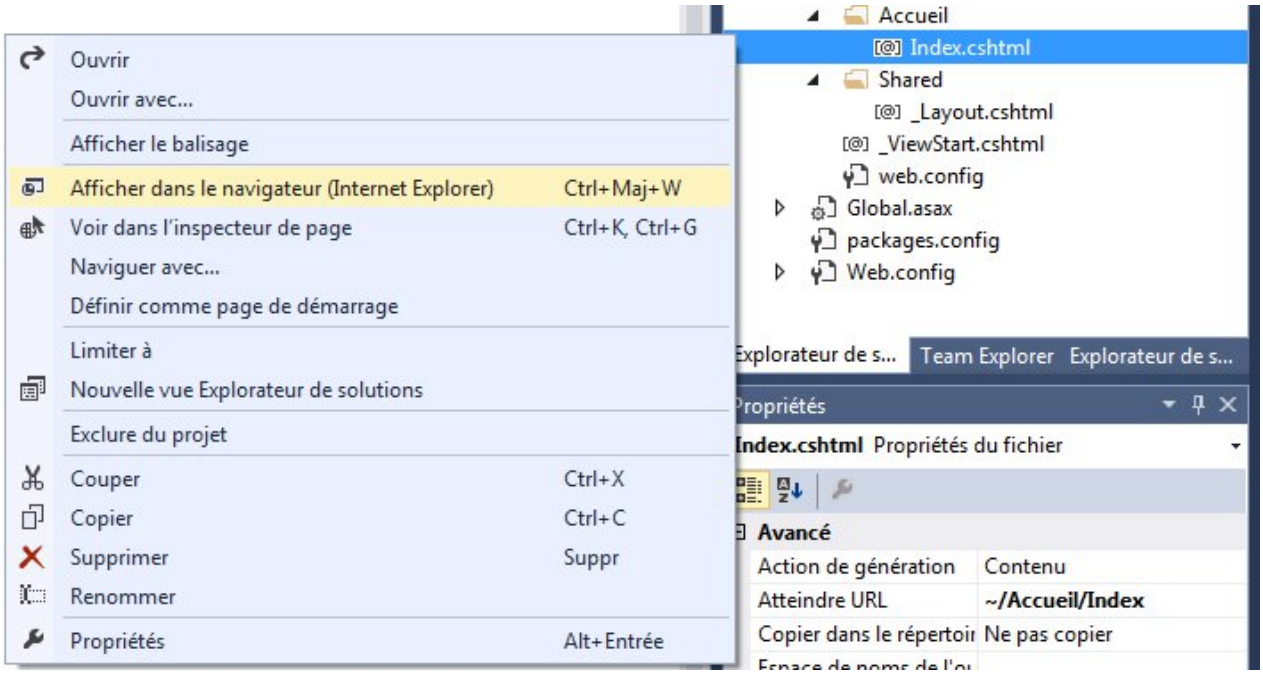


FIGURE 6.14. – Testez notre layout

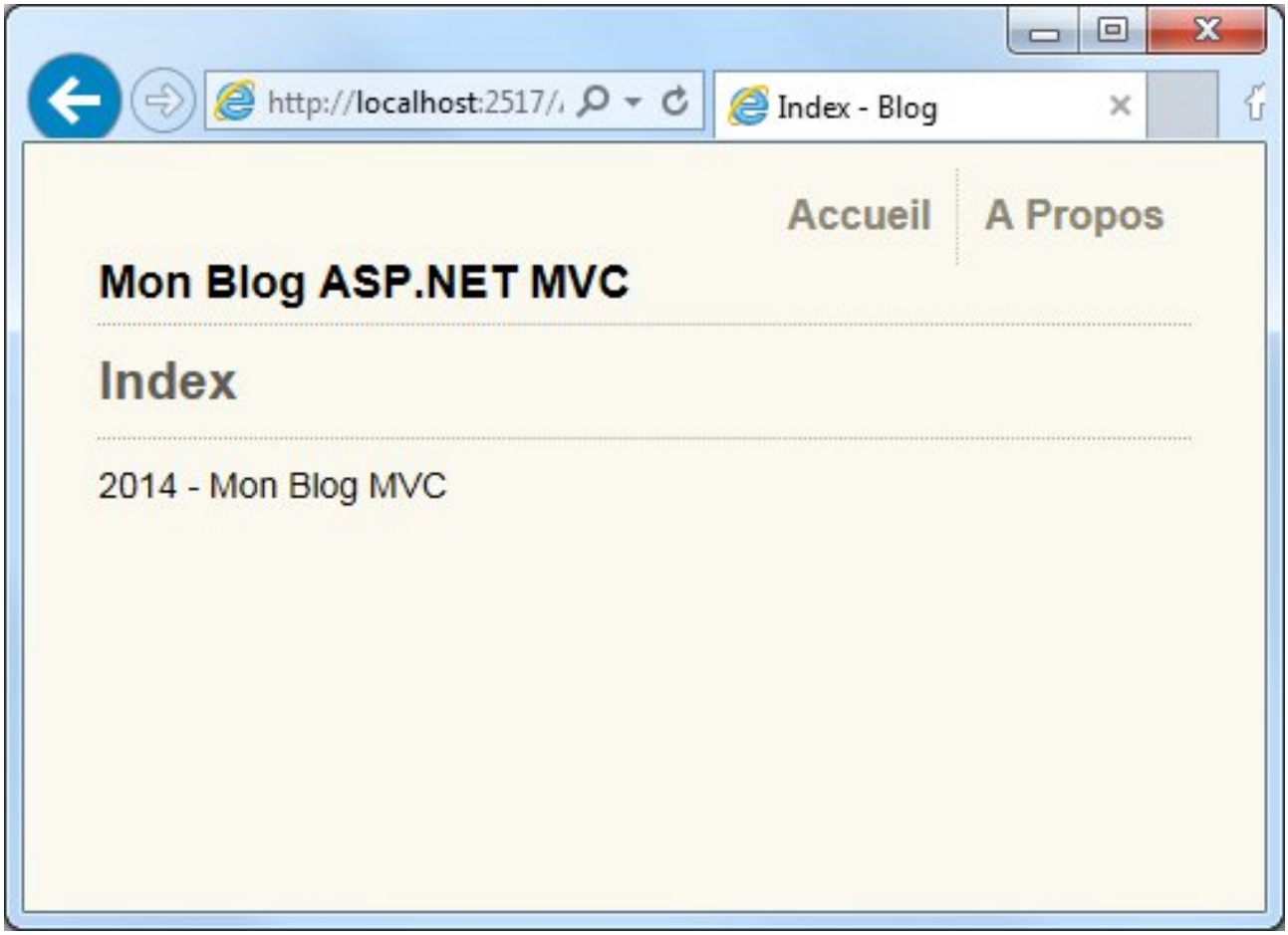


FIGURE 6.15. – Page d'accueil

Il suffit de recommencer la même opération sur la méthode `About` pour générer la vue /About/. Toutes nos vues auront désormais la même mise en page sans que nous ayons à répéter les

mêmes choses à chaque fois.

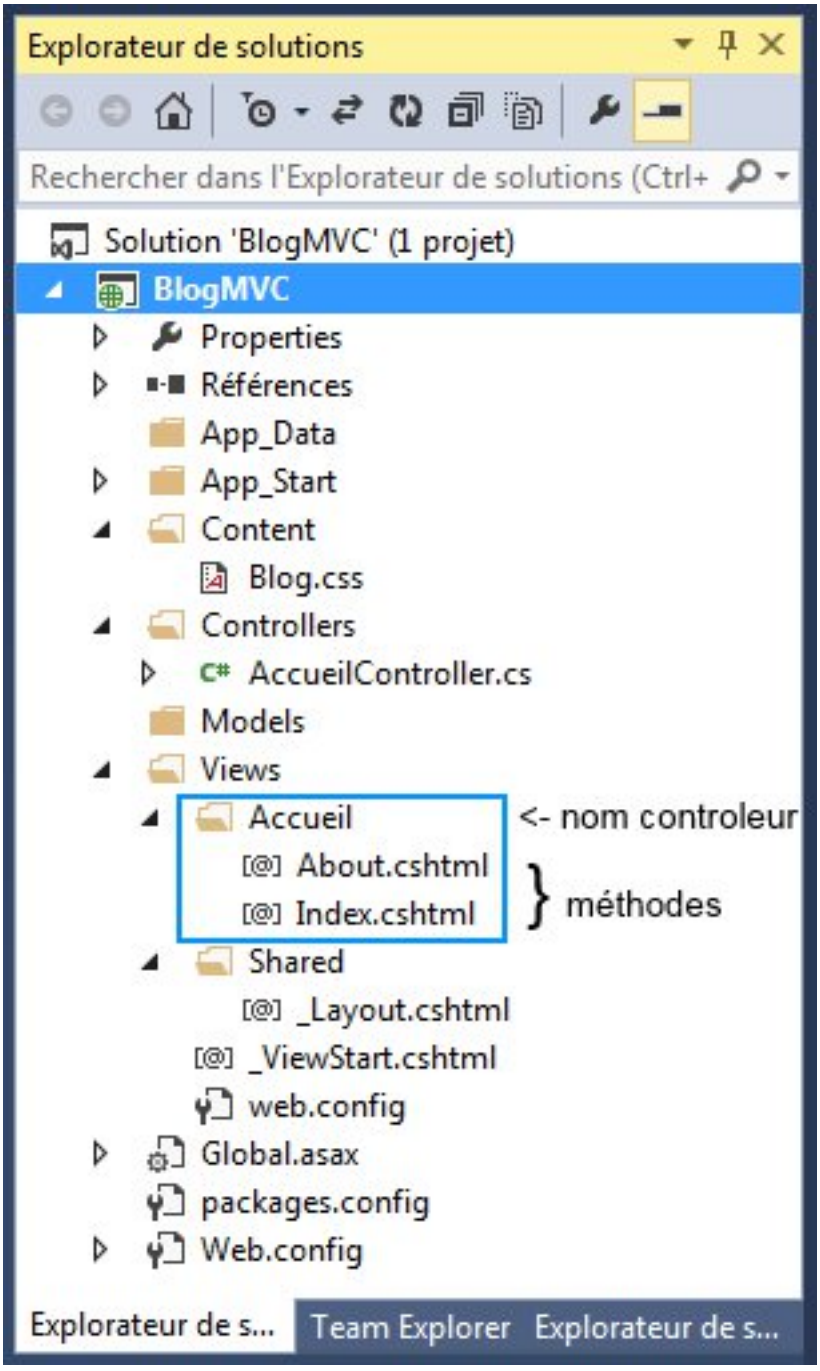


FIGURE 6.16. – Solution initiale de notre Blog

6.4. Les sessions et les cookies

Vous vous rendrez vite compte d'un problème récurrent lorsqu'on fait du développement web : quand on passe d'une page à l'autre, toutes les données qui n'ont pas été enregistrées "en dur" (dans un fichier ou dans la base de données) sont oubliées.
Ce comportement est dû à une propriété de base de HTTP, on dit qu'il est *stateless*. Ce mot signifie que deux requêtes successives sont indépendantes. Cela permet d'éviter les *effets de bord* qui seraient une source infinie de bugs.

? Alors comment retenir des informations de pages en pages ?

Il existe deux types d'informations :

— les informations qui n'ont d'intérêt que pendant la visite **actuelle** de l'utilisateur. La devise de ces données c'est "Demain est un autre jour". Pour leur cas particulier, nous utiliserons les **sessions**. C'est par exemple le cas lorsque sur un site de commerce électronique, vous utilisez un panier qui se complète au fur et à mesure ;

— les informations qui sont utiles à long terme. Leur devise c'est "souvenir, souvenir". L'exemple le plus parlant est celui de la case que vous cochez dans le formulaire de connexion "se souvenir de moi".

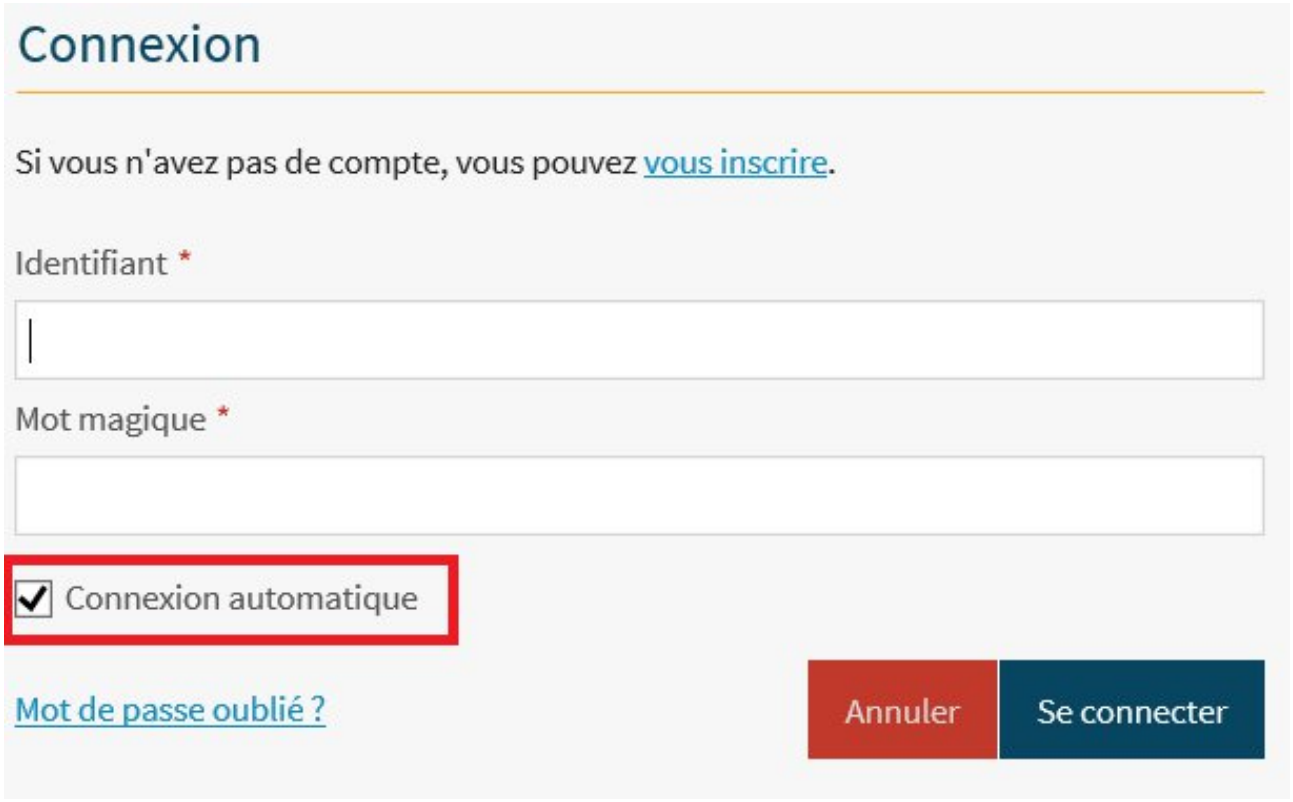


FIGURE 6.17. – Connexion automatique sur ZdS

6.4.1. Les sessions

Une session, c'est un tableau de données qui ne dure que pendant que l'utilisateur visite votre site.

Panier	["Pépites de chocolat", "farine", "œuf"]
Connecte_en_tant_que	"Artragis"
Nombre_Pages_Visitees	5
...	...

FIGURE 6.18. – Exemple de tableau de session

Pour déterminer que l'utilisateur est en train de visiter votre site, le serveur déclenche un compte à rebours de quelques minutes qui est remis à zéro à chaque fois que vous cliquez sur un lien du site.

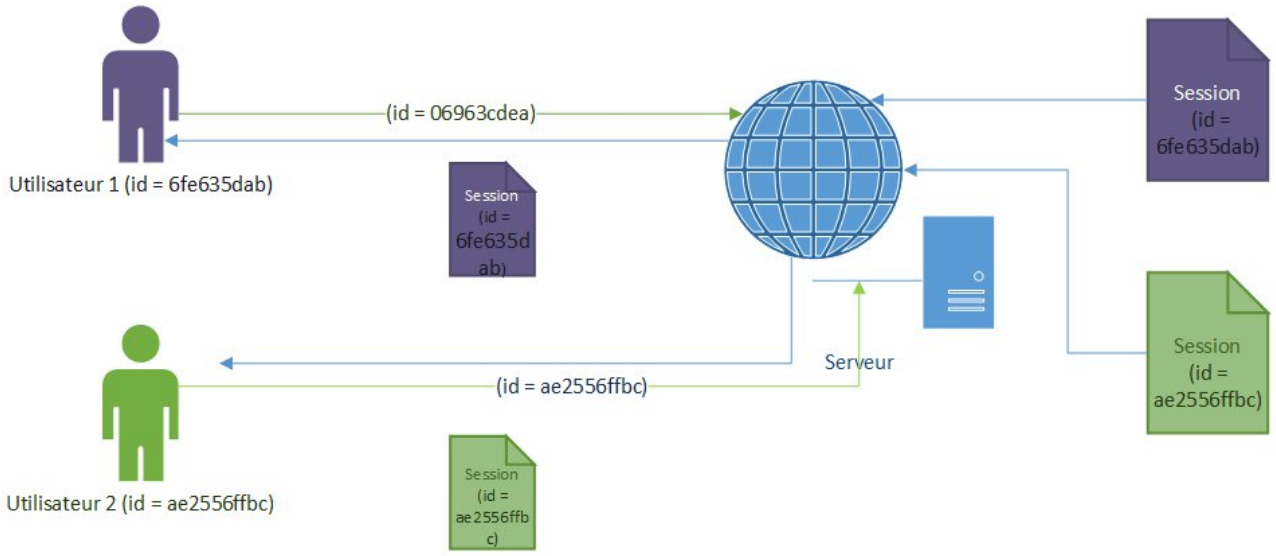


FIGURE 6.19. – Fonctionnement d'une session

Les données de la sessions sont stockées **sur le serveur**. Pour associer un utilisateur à une session le serveur utilise un **identifiant de session**. Par défaut cet identifiant est généré aléatoirement selon des standards sécurisés[7]. Vous pouvez définir vous-mêmes le moyen de générer les identifiants de session, mais je ne vous dirai pas comment car c'est hautement déconseillé dans 99.9% des cas.
Pour manipuler des données en session, il faut utiliser la propriété `Session` de votre contrôleur :

```
1         public ActionResult UneActionAvecSession()
2         {
3             Session["panier"] = "blabla";//on écrit un string dans
                  la clef "panier"
4             if (Session["Nombre_Pages_Visitees"] != null)
5             {
6                 Session["Nombre_Pages_Visitees"] =
                      (int)Session["Nombre_Pages_Visitees"] + 1;
7             }
8             else
9             {
10                Session["Nombre_Pages_Visitees"] = 1;
11            }
12            return View();
13        }
```

Code : Manipulation basique des sessions
Comme vous avez pu le voir, on peut mettre tout et n'importe quoi dans une session.
Cela signifie qu'il **faut** convertir à **chaque fois** les données pour pouvoir les utiliser.
Une façon plus *élégante* serait d'accéder à notre `Session["Nombre_Pages_Visitees"]` à partir d'une propriété, ce qui permettrait d'éviter un mauvais copier/coller de la clef.

Pour gérer les connexions d'utilisateurs, ASP.NET peut utiliser les sessions. Dans ce cas là, vous n'avez pas à vous préoccuper de ce qui se passe en interne, le framework gère tout lui-même et ça évite beaucoup de bugs.

Il se peut que vous observiez des clefs qui apparaissent toutes seules mais pour une seule page. On appelle ça des "flashes". Cette technique est très utilisée pour la gestion des erreurs dans les formulaires.

Par défaut, l'identifiant de session est enregistré dans un **cookie** appelé `SESSID`. Je vous conseille de laisser ce comportement par défaut.

Par défaut, votre navigateur partage la session à tous les onglets qui pointent vers le site web. Pour éviter cela, il faut activer la fonctionnalité "session multiple".

6.4.2. Les cookies

Pour retenir une information longtemps (selon les cas, la loi peut obliger un maximum de 13 mois), vous pouvez utiliser un **Cookie**.
Contrairement aux sessions, les cookies sont stockés dans le navigateur du client.
Les cookies sont de simples fichier contenant du texte, ils sont **inoffensifs**. Ils permettent de stocker un petit nombre d'informations afin de vous aider à les passer de page en page.
À chaque fois que vous envoyez une requête, les cookies sont envoyés au serveur. Dans ASP.NET vous manipulez les cookies comme une collection spécialisée :

```
1  public ActionResult UneActionAvecUnCookie()
2          {
3              if (Request.Cookies["valeur_simple"] != null) //on
                   vérifie que le cookie existe TOUJOURS
4              {
5                  //obtenir les valeurs
6                  string valeur =
                       Request.Cookies.Get("valeur_simple").Value;
7              }
8              //ajouter les valeurs
9              //un conseil : toujours mettre Secure à true
10             //on peut aussi définir le temps de validité du cookie
                   avec Expires
11             Response.Cookies.Set(new HttpCookie("valeur_simple",
                   "blabla") { Secure = true });
12
13             return View();
14         }
```

Listing 6 – Manipulation de base des cookies

Comme on peut le voir ci-dessus, on utilise deux objets différents pour mettre en place des cookies. Il y a `Request` qui permet de lire les données du navigateur et `Response` qui permet de sauvegarder des informations chez le client.

6.4.3. Mini TP : Un bandeau pour prévenir l'utilisateur ?

Ces derniers temps, on parle beaucoup des cookies, et les politiques, les médias, tout le monde s'est emparé du problème.
Cela signifie que vous avez des obligations légales à propos des cookies[8]. L'une d'entre elles est de demander à votre visiteur s'il accepte les cookies **dans le cas où ces derniers manipulent des données personnelles**, notamment pour tracer vos faits et gestes.
Les cookies ne sont **pas** dangereux en soi, et il n'est pas interdit d'utiliser des cookies.
Je le répète : seuls certains cookies particuliers[9] nécessitent l'approbation de vos visiteurs pour être utilisés.
Dans ce cas il faut mettre un bandeau dans votre site.
Cela peut être un exercice très intéressant pour vous de mettre en place ce bandeau. Considérez cela comme un mini TP dont la solution se trouve en dessous.

6.4.3.1. Quelques indices

Nous allons faire les choses simplement, il y aura qu'un seul contrôleur qui va afficher le bandeau (/Accueil/Index). Le code HTML sera simple, une phrase avec deux liens qui renverront vers un contrôleur du type /Accueil/Cookies ?accept=1 (ou 0).
Voici un petit diagramme qui explique comment les choses doivent se passer :

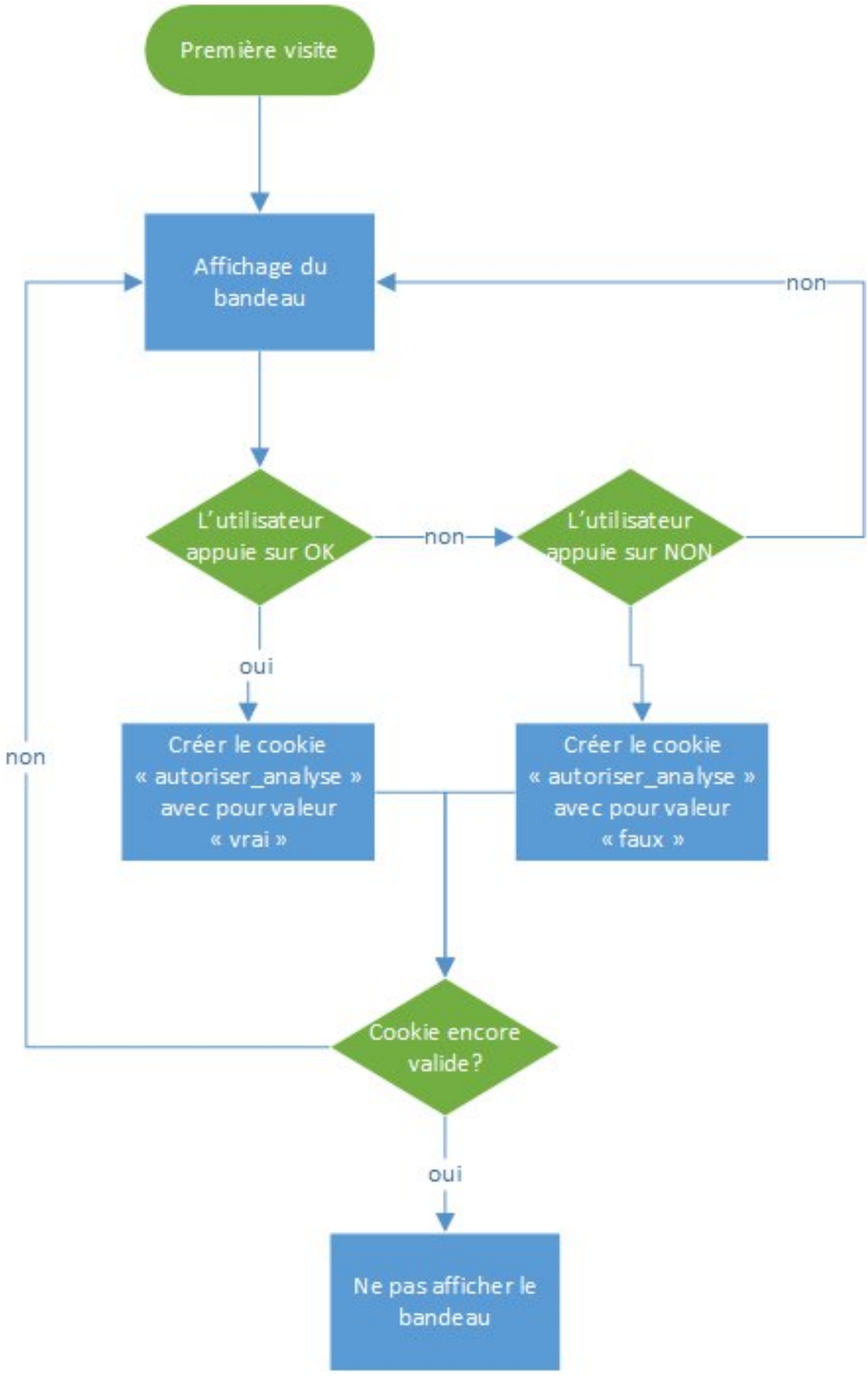


FIGURE 6.20. – Logique pour la création du bandeau

Comment tester notre solution ?

Ce qu'il faut savoir, c'est que pour récupérer nos cookies, il ne faut pas être en localhost. Pour changer cela il y a quelques petites manipulations à faire :

— allez dans les propriétés de votre projet puis onglet Web et définissez l'URL, comme sur cette image
— changez le fichier hosts (C :\Windows\System32\drivers\etc\hosts) et rajoutez simplement, 127.0.0.1 livesite.dev
— modifiez applicationhost.config (%USERPROFILE%\My Documents\IISExpress\config\applicationhost.config) et cherchez 2693 puis remplacez le mot localhost par livesite.dev

```
            <site name="Blog" id="VotreID">
                <application path="/"
                    applicationPool="Clr4IntegratedAppPool">
                    <virtualDirectory path="/"
                        physicalPath="VotreChemin" />
                </application>
                <bindings>
                    <binding protocol="http"
                        bindingInformation="*:2693:livesite.dev" />
                </bindings>
            </site>
```

Et voilà, on voit notre cookie, par exemple dans chrome .

6.4.3.2. Une solution

Contenu masqué n°2

Contenu masqué

Contenu masqué n°1

```
* {
    margin: 0px;
    padding: 0px;
    border: none;
```

7. Afin de s'assurer au maximum de la sécurité (dans notre cas de l'unicité de l'identifiant et de la difficulté à le deviner), les algorithmes utilisés demandent une grande entropie .
8. Notamment, un cookie ne doit pas avoir une durée de vie de plus de 13 mois. La liste complète se trouve sur le site de la CNIL .
9. Liste complète .

```
}

body {
    font-family: Arial, Helvetica, sans-serif;
    font-size: 14px;
    background-color: #FBF9EF;
    padding: 0px 6%;
}

header {
    float: left;
    width: 100%;
    border-bottom: 1px dotted #5D5A53;
    margin-bottom: 10px;
}

header h1 {
    font-size: 18px;
        float: left;
    padding: 45px 0px 5px 0px;
}

ul li a {
    font-size: 16px;
}

ul
{
        list-style-type: square;
        margin-left: 25px;
        font-size: 14px;
}

footer {
    width: 100%;
    border-top: 1px dotted #5D5A53;
    margin-top: 10px;
    padding-top: 10px;
}

/* barre de navigation header */

ul#navliste
{
        float: right;
}

ul#navliste li
{
        display: inline;
```

```
}

ul#navliste li a
{
        border-left: 1px dotted #8A8575;
        padding: 10px;
        margin-top: 10px;
        color: #8A8575;
        text-decoration: none;
        float: left;
}

ul#navliste li:first-child a
{
        border: none;
}

ul#navliste li a:hover
{
        color: #F6855E;
}

/* fin barre de navigation header*/

p
{
        margin-bottom: 15px;
        margin-top: 0px;
}

h2
{
        color: #5e5b54;
}

header h1 a
{
        color: #5E5B54;
}

a:link, a:visited
{
        color: #F6855E;
        text-decoration: none;
        font-weight: bold;
}

a:hover
{
        color: #333333;
```

```
        text-decoration: none;
        font-weight: bold;
}

a:active
{
        color: #006633;
        text-decoration: none;
        font-weight: bold;
}
```

Retourner au texte.

Contenu masqué n°2

```
// GET: Accueil
        public ActionResult Index()
        {
            bool afficheBandeau = true;
            if (Request.Cookies["autoriser_analyse"] != null)
            {
                afficheBandeau =
                    Request.Cookies.Get("autoriser_analyse").Value
                    == "vrai" ? false : true;
            }
            ViewBag.DisplayCookie = afficheBandeau;

            return View();
        }

        //
        // GET : /Cookies?accept=1
        public ActionResult Cookies(int accept)
        {
            string autorizeCookies = "faux";

            //Accepte le cookie
            if (accept == 1)
            {
                autorizeCookies = "vrai";
            }

            Response.Cookies.Set(new
                HttpCookie("autoriser_analyse", autorizeCookies) {
                Expires = DateTime.MaxValue });

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

Code : Le contrôleur

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title - Blog</title>
    <link rel="stylesheet" href="~/Content/Blog.css"
        type="text/css" />
</head>
<body>
    @{
        if (ViewBag.DisplayCookie != null)
        {
            <div id="cookies-banner" @(ViewBag.DisplayCookie ?
                "style=display:block;" : "")>
                <span>
                    En poursuivant votre navigation sur ce site,
                        vous nous autorisez à déposer des cookies.
                        Voulez vous accepter ?
                </span>
                <a href="/Accueil/Cookies?accept=1"
                    id="accept-cookies">Oui</a>
                <a href="/Accueil/Cookies?accept=0"
                    id="reject-cookies">Non</a>
            </div>
        }
    }

    <header>
        <h1>Mon Blog ASP.NET MVC</h1>
        <ul id="navliste">
            <li class="premier"><a href="/Accueil/"
                id="courant">Accueil</a></li>
            <li><a href="/Article/List/">Blog</a></li>
            <li><a href="/Accueil/About/">A Propos</a></li>
        </ul>
    </header>
    <div id="corps">
        @RenderBody()
    </div>
    <footer>
        <p>@DateTime.Now.Year - Mon Blog MVC</p>
    </footer>
</body>
</html>
```

Code : La vue

```css
/* barre cookie */

#cookies-banner {
    display: none;
    background: none repeat scroll 0% 0% #062E41;
    padding: 0px 2.5%;
}

#cookies-banner span {
    display: inline-block;
    margin: 0px;
    padding: 7px 0px;
    color: #EEE;
    line-height: 23px;
}

#cookies-banner #reject-cookies {
    display: inline-block;
    background: none repeat scroll 0px 0px transparent;
    border: medium none;
    text-decoration: underline;
    margin: 0px;
    padding: 0px;
    color: #EEE;
}

#cookies-banner a {
    display: inline-block;
    color: #EEE;
    padding: 4px 13px;
    margin-left: 15px;
    background: none repeat scroll 0% 0% #084561;
    text-decoration: none;
}

#cookies-banner #accept-cookies {
    text-decoration: none;
    background: none repeat scroll 0% 0% #EEE;
    color: #084561;
    padding: 4px 15px;
    border: medium none;
    transition: #000 0.15s ease 0s, color 0.15s ease 0s;
    margin-top: 3px;
}

/* fin barre cookie */
```

Code : le css Retourner au texte.

7. Intéragir avec les utilisateurs

Lorsque vous développez votre site, vous allez devoir demander à vos utilisateurs d'entrer des données.
Par exemple, vous allez lui demander quelle *page* d'une liste il veut afficher, quels sont son pseudonyme et son mot de passe, le texte d'un commentaire ou d'un article...
Ce chapitre vous permettra de percer les secrets des formulaires, d'envoyer des fichiers tels que des images et même afficher les articles de votre blog.
Nous parlerons aussi de *sécurité* parce que, en informatique, il y a une règle d'or : **Never Trust User Input**, c'est-à-dire *ne faites jamais confiance à ce que les utilisateurs saisissent.*

7.1. Préparation aux cas d'étude

Dans les sections suivantes, nous allons échanger des données avec nos utilisateurs. Pour cela, quatre cas d'études sont prévus :

- afficher vos articles ;
- la création d'un article ;
- l'envoi d'une image ;
- la sélection d'une page parmi plusieurs (la *pagination*).

Pour que ces cas d'études se passent de la meilleure des manières, il va falloir préparer votre espace de travail.
Comme nous allons manipuler des "articles", il faudra commencer par définir ce qu'est un article. Pour faire simple, nous allons supposer qu'un article possède les *propriétés* suivantes :

- le pseudo de son auteur ;
- le titre de l'article ;
- le texte de l'article.

Dans le dossier `Models`, créez la classe suivante :

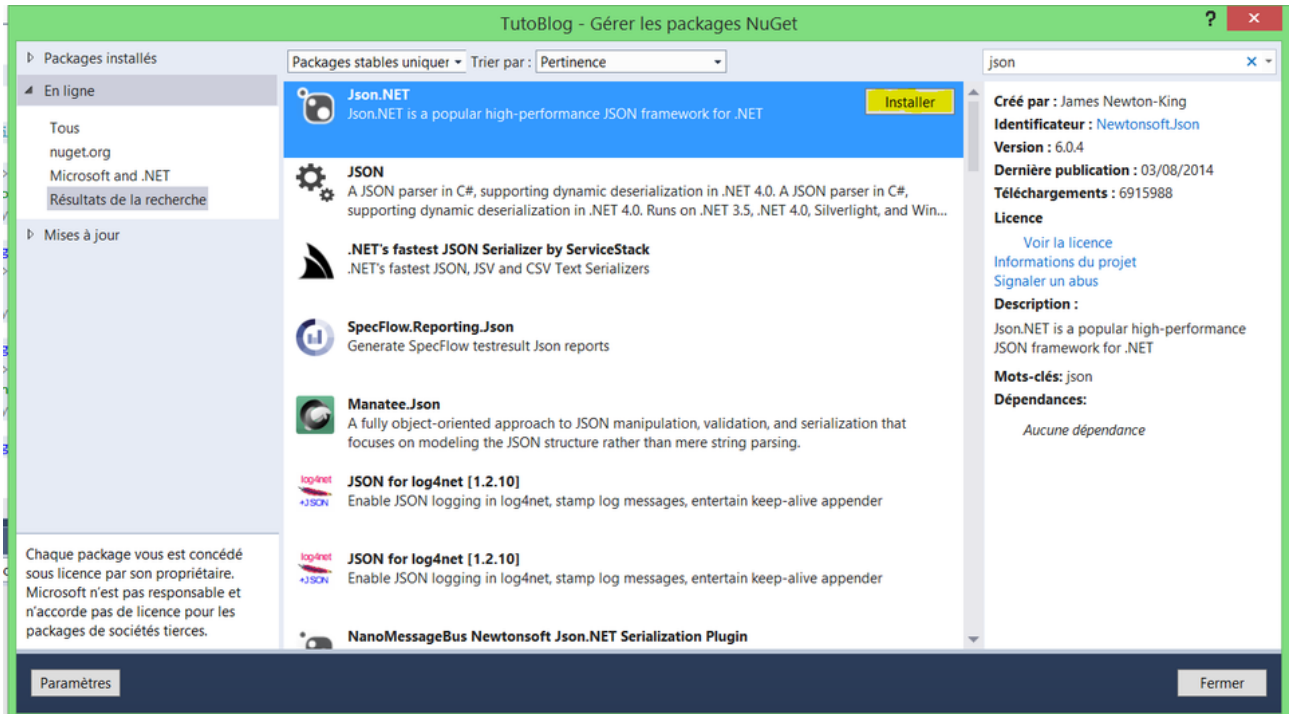
```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5
6  namespace Blog.Models
7  {
8      /// <summary>
9      /// Notre modèle de base pour représenter un article
10     /// </summary>
11     public class Article
12     {
13         /// <summary>
14         /// Le pseudo de l'auteur
```

```
15          /// </summary>
16          public string Pseudo { get; set; }
17          /// <summary>
18          /// Le titre de l'article
19          /// </summary>
20          public string Titre { get; set; }
21          /// <summary>
22          /// Le contenu de l'article
23          /// </summary>
24          public string Contenu { get; set; }
25      }
26  }
```

Pour aller plus vite, je vous propose un exemple de liste d'articles au format JSON. Ce fichier nous permet de stocker nos articles, nous utiliserons une base de données dans le prochain chapitre.
Téléchargez le fichier , et placez-le dans le dossier App_Data.
Puis, pour qu'il apparaisse dans Visual Studio, faites un clic droit sur le dossier App_Data, et "Ajouter un élément existant". Sélectionnez le fichier.
Il est extrêmement facile de lire un fichier de ce type, contrairement à un fichier texte. JSON est un format de données textuelles, tout comme le XML. Il est très répandu dans le monde du développement web. Le contenu d'un fichier JSON ressemble à cela :

```
1  {
2      "post": {
3          "pseudo": { "value": "Clem" },
4          "titre": { "value": "mon article" },
5          "contenu": { "value": "Du texte, du texte, du texte" }
6      }
7  }
```

Maintenant, nous allons utiliser un outil qui est capable de lire le JSON. Pour cela, nous allons utiliser un package : faites un clic droit sur le nom du projet puis "Gérer les packages nuget". Cherchez JSON.NET et installez-le.
Pour ceux qui veulent savoir ce qu'est NuGet , c'est simplement une bibliothèque de packages, qui regroupe de nombreuses librairies et DLL pour la plateforme de développement sous Microsoft.



Ensuite, dans le dossier `Models` créez le fichier `ArticleJSONRepository.cs` et entrez ce code :

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using Newtonsoft.Json;
6  using System.IO;
7
8  namespace Blog.Models
9  {
10     /// <summary>
11     /// permet de gérer les articles qui sont enregistrés dans un
       fichier JSON
12     /// </summary>
13     public class ArticleJSONRepository
14     {
15         /// <summary>
16         /// Représente le chemin du fichier JSON
17         /// </summary>
18         private readonly string _savedFile;
19
20         /// <summary>
21         /// Construit le gestionnaire d'article à partir du nom
           d'un fichier JSON
22         /// </summary>
23         /// <param name="fileName">nom du fichier json</param>
24         public ArticleJSONRepository(string fileName)
25         {
```

```
26              _savedFile = fileName;
27          }
28
29          /// <summary>
30          /// Obtient un unique article à partir de sa place dans la
                liste enregistrée
31          /// </summary>
32          /// <param name="id">la place de l'article dans la liste
                (commence de 0)</param>
33          /// <returns>L'article désiré</returns>
34          public Article GetArticleById(int id)
35          {
36              using (StreamReader reader = new
                    StreamReader(_savedFile))
37              {
38                  List<Article> list =
                        JsonConvert.DeserializeObject<List<Article>>(reader.ReadTo
39
40                  if (list.Count < id || id < 0)
41                  {
42                      throw new
                            ArgumentOutOfRangeException("Id incorrect");
43                  }
44                  return list[id];
45              }
46
47          }
48
49          /// <summary>
50          /// Obtient une liste d'article
51          /// </summary>
52          /// <param name="start">Premier article sélectionné</param>
53          /// <param name="count">Nombre d'article
                sélectionné</param>
54          /// <returns></returns>
55          public IEnumerable<Article> GetListArticle(int start = 0,
                int count = 10)
56          {
57              using (StreamReader reader = new
                    StreamReader(_savedFile))
58              {
59                  List<Article> list =
                        JsonConvert.DeserializeObject<List<Article>>(reader.ReadTo
60                  return list.Skip(start).Take(count);
61              }
62          }
63
64          /// <summary>
65          /// Obtient une liste de tout les articles
66          /// </summary>
```

```
67          public IEnumerable<Article> GetAllListArticle()
68          {
69              using (StreamReader reader = new
                    StreamReader(_savedFile))
70              {
71                  List<Article> list =
                        JsonConvert.DeserializeObject<List<Article>>(reader.ReadTo
72                  return list;
73              }
74          }
75
76          /// <summary>
77          /// Ajoute un article à la liste
78          /// </summary>
79          /// <param name="newArticle">Le nouvel article</param>
80          public void AddArticle(Article newArticle)
81          {
82              List<Article> list;
83              using (StreamReader reader = new
                    StreamReader(_savedFile))
84              {
85                  list =
                        JsonConvert.DeserializeObject<List<Article>>(reader.ReadTo
86              }
87              using (StreamWriter writter = new
                    StreamWriter(_savedFile, false))
88              {
89                  list.Add(newArticle);
90                  writter.Write(JsonConvert.SerializeObject(list));
91              }
92          }
93      }
94  }
```

Avec le code ci-dessus nous allons pouvoir récupérer nos articles qui se trouvent dans notre ficher JSON, et pouvoir en créer. Les méthodes parlent d'elles-mêmes : nous lisons et écrivons dans le fichier JSON.

7.2. Cas d'étude numéro 1 : afficher les articles

Ce premier cas d'étude vous permettra de mettre en œuvre :
- la création d'un contrôleur qui nous retourne une liste d'articles ;
- la création d'une vue avec un modèle fortement typé ;
- l'utilisation du layout.

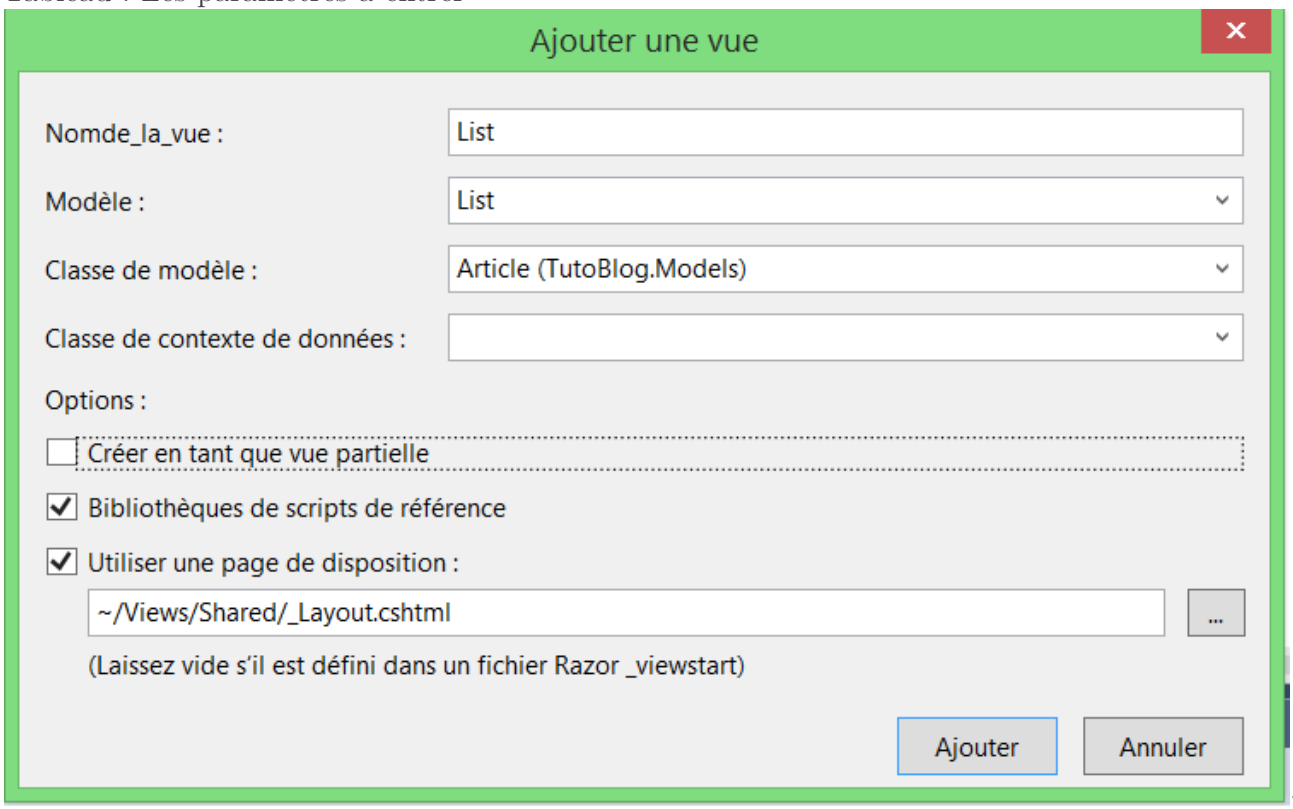
7.2.1. S'organiser pour répondre au problème

Comme c'est notre premier cas d'étude, je vais vous faire un pas à pas détaillé de ce qu'il faut faire puis vous coderez vous-mêmes les choses nécessaires.

1. Dans le contrôleur Article, créez une méthode List `public ActionResult List()`
2. Remplir la liste en utilisant la classe ArticleJSONRepository.cs
3. Ajouter une vue que le contrôleur va retourner, et entrez les paramètres décrits dans le tableau

paramètres	**valeur**
Nom	**List**
Modèle	List
Classe de modèle	Article (votre classe de modèle)
Utiliser une page de disposition	Oui

Tableau : Les paramètres à entrer



<-

Voici le code qui permet de récupérer le chemin de votre fichier qui se trouve dans le dossier App_Data :

```
1  string path = Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
       "App_Data", "liste_article_tuto_full.json");
```

Bien sur vous rajouterez un lien dans votre menu qui va permettre d'afficher votre liste d'articles.

7.2.2. Correction

Comme vous l'avez remarqué, c'est rapide de créer une vue qui affiche nos articles. Bien sur comme d'habitude, je vous invite à lancer votre application pour voir le résultat et le code source généré.

> Contenu masqué n°3

Jetons un coup d'œil à ce que nous a généré la vue :

- un tableau qui affiche les propriétés publiques de notre modèle Article ;
- différents liens qui pointent sur des méthodes du contrôleur : `Create` / `Edit` / `Details` / `Delete`.

7.3. Cas d'étude numéro 2 : publier un article

7.3.1. Les étapes de la publication

Dans ce cas d'étude, nous ne parlerons pas des autorisations, nous les mettrons en place **plus tard**, en partie III et IV.
Pour autant, il faut bien comprendre que la publication d'un article ne se fait pas au petit bonheur la chance. Il va falloir respecter certaines étapes qui sont **nécessaires**.
Comme pour tout ajout de fonctionnalité à notre site, nous allons devoir créer une action. Par *convention* (ce n'est pas une obligation, mais c'est tellement mieux si vous respectez ça), l'action de **créer** du contenu s'appelle `Create`, celle d'éditer `Edit`, celle d'afficher le contenu seul (pas sous forme de liste) `Details`.
Le seul problème, c'est que la création se déroule en deux étapes :

- afficher le formulaire ;
- traiter le formulaire et enregistrer l'article ainsi posté.

Il va donc bel et bien falloir créer deux actions. Pour cela, nous allons utiliser la capacité de C# à nommer de la même manière deux méthodes différentes du moment qu'elles n'ont pas les mêmes arguments.
La méthode pour afficher le formulaire s'appellera `public ActionResult Create()`.
Lorsque vous allez envoyer vos données, ASP.NET est capable de construire seul l'objet Article à partir de ce qui a été envoyé.
De ce fait, vous allez pouvoir appeler la méthode qui traitera le formulaire `public ActionResult Create(Article article)`. Elle devra comporter deux annotations qui seront détaillées ci-dessous :

```
1 [HttpPost]
2 [ValidateAntiForgeryToken]
3 public ActionResult Create(Article article)
4 {
5     //Votre code
6 }
```

Listing 7 – Protection contre la faille CSRF

C'est pour cela que de base, lorsque vous créerez la vue `Create.cshtml`, vous allez choisir comme modèle `Article` et comme template `Create`. Ce template génère un formulaire complet

qui possède deux capacités :

— envoyer un article quand vous le remplissez ;

— afficher les erreurs si jamais l'utilisateur a envoyé des données incomplètes ou mal formées.

Maintenant que vous avez vos deux actions et votre formulaire, nous allons pouvoir ajouter de la logique à notre application.

Principalement, cette logique sera toujours la même :

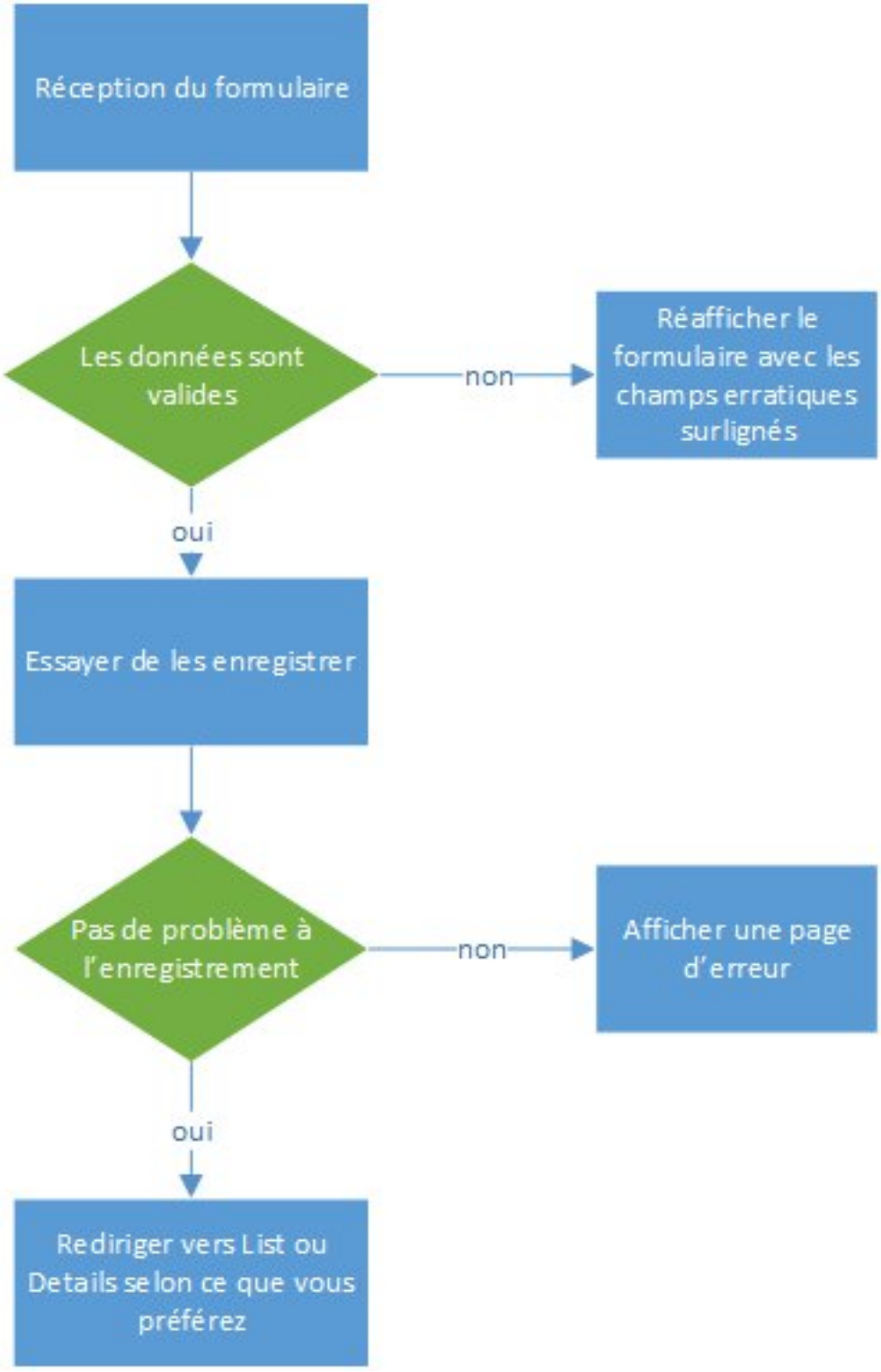


FIGURE 7.1. – Logique pour la création de contenu

Le jeu sera donc de savoir :
- comment on valide ;
- comment on fait une redirection.

Je vous conseille de lancer l'application pour voir le rendu du formulaire.
Faites un clic-droit sur "*afficher le code source de la page*", je vous conseille de comparer le code HTML généré avec votre code `create.cshtml` pour voir le rendu des différents éléments de la page.
Il y a trois *helpers* à retenir (et uniquement trois !) :
- `Html.LabelFor(model=>model.Propriete)` pour le label (exemple `Html.LabelFor(model => model.Titre)`) ;
- `Html.EditorFor(model=>model.Propriete)` pour le champ (exemple `Html.EditorFor(model => model.Titre)`) ;
- `Html.ValidationMessageFor(model=>model.Propriete)` pour le message d'erreur(exemple `Html.ValidationMessageFor(model => model.Titre)`).

À noter, vous pourrez aussi trouver, au début du formulaire, `Html.ValidationSummary()` qui sert pour la sécurité.

Comme pour le formulaire, vous pouvez *customiser* le résultat de chacun des trois *helpers* en utilisant une version de la fonction qui a un argument appelé **htmlattributes**.

```
1 @Html.LabelFor(model => model.Titre, htmlAttributes: new { @class =
      "control-label col-md-2" })
```

Listing 8 – Avec htmlattributes

En fin de tuto, vous pouvez retrouver un glossaire qui montre les différents contrôles Razor et leur résultat en HTML.
Mais avant d'aller plus loin, un petit point sur les formulaires s'impose.

7.3.2. Explication formulaire

Le formulaire est la façon la plus basique d'interagir avec l'utilisateur. Derrière le mot "interagir", se cachent en fait deux actions : l'utilisateur vous demande des choses ou bien l'utilisateur vous envoie du contenu ou en supprime.
Ces actions sont distinguées en deux verbes :
- `GET` : l'utilisateur veut juste **afficher** le contenu qui existe *déjà*. Il est important de comprendre qu'une requête qui dit "GET" ne doit **pas** modifier d'informations dans votre base de données (sauf si vous enregistrez des statistiques) ;
- `POST` : l'utilisateur **agit** sur les données ou en *crée*. Vous entendrez peut être parler des verbes `PUT` et `DELETE`. Ces derniers ne vous seront utiles que si vous utilisez le JavaScript[10] ou bien si vous développez une API REST. En navigation basique, seul `POST` est supporté.

Vous avez sûrement remarqué : je n'ai pas encore parlé de sécurité. La raison est simple, `GET` ou `POST` n'apportent **aucune** différence du point de vue de la sécurité. Si quelqu'un vous dit le contraire, c'est qu'il a sûrement mal sécurisé son site.

Pour rappel, un formulaire en HTML ressemble à ceci :

```
<!-- permet de créer un formulaire qui ouvrira la page zestedesavoir.com/rechercher
l'attribut method="get", permet de dire qu'on va "lister" des choses
-->
<form id="search_form" class="clearfix search-form" action="/rechercher" method="get">
    <!-- permet de créer un champ texte, ici prérempli avec "zep".-->
    <input id="id_q" type="search" value="zep" name="q"></input>
    <button type="submit"></button><!-- permet de valider le formulaire-->

</form>
```

Listing 9 – Un formulaire de recherche sur ZdS[11]

En syntaxe Razor, la balise `<form>` donne `HTML.BeginForm`, il existe différents paramètres pour définir si on est en "mode" `GET` ou `POST`, quel contrôleur appeler etc.. Bien sûr, on peut très bien écrire le formulaire en HTML comme ci-dessus.

7.3.3. Particularités d'un formulaire GET

Comme vous devez lister le contenu, le formulaire en mode `GET` possède quelques propriétés qui peuvent s'avérer intéressantes :

- toutes les valeurs se retrouvent placées à la fin de l'URL sous la forme : `url_de_base/?nom_champ_1=valeur1&nom_champ2=valeur2` ;
- l'URL complète (URL de base + les valeurs) ne doit pas contenir plus de 255 caractères ;
- si vous copiez/collez l'URL complète dans un autre onglet ou un autre navigateur : vous aurez accès au **même résultat** (sauf si la page demande à ce que vous soyez enregistré !)[12].

7.3.4. Particularités d'un formulaire POST

Les formulaires POST sont fait pour envoyer des contenus nouveaux. Ils sont donc beaucoup moins limités :

- il n'y a pas de limite de nombre de caractères ;
- il n'y a pas de limite de caractères spéciaux (vous pouvez mettre des accents, des smileys...) ;
- il est possible d'envoyer un ou plusieurs fichiers, seul le serveur limitera la taille du fichier envoyé.

Comme pour les formulaires `GET`, les données sont entrées sous la forme de clef=>valeur. À ceci près que les formulaires `POST` envoient les données dans la requête HTTP elle-même et donc sont invisibles pour l'utilisateur lambda.
Autre propriété intéressante : si vous utilisez HTTPS, les données envoyées par `POST` sont cryptées. C'est pour ça qu'il est fortement conseillé d'utiliser ce protocole (quand on a un certificat[13]) dès que vous demandez un mot de passe à votre utilisateur.

7.3.4.1. Validation du formulaire

Les formulaires Razor sont très intelligents et vous permettent d'afficher rapidement :
- le champ adapté à la donnée, par exemple si vous avez une adresse mail, il vous enverra un `<input type="email"/>` ;
- l'étiquette de la donnée (la balise `<label>`) avec le bon formatage et le bon texte ;
- les erreurs qui ont été détectées lors de l'envoi grâce à JavaScript (attention, c'est un confort, pas une sécurité) ;
- les erreurs qui ont été détectées par le serveur (là, c'est une sécurité).

Pour que Razor soit capable de définir le bon type de champ à montrer, il faut que vous lui donniez cette indication dans votre classe de modèle. Cela permet aussi d'indiquer ce que c'est qu'un article *valide*.
Voici quelques attributs que l'on rencontre très souvent :

Nom de l'attribut	Effet	Exemple
System.ComponentModel .DataAnnotations.RequiredAttribute	**Force la propriété à être toujours présente**	**`[Required(AllowEmptyStrings=false)]`**
System.ComponentModel .DataAnnotations .StringLengthAttribute	Permet de limiter la longueur d'un texte	`[StringLength(128)]`
System.ComponentModel .DataAnnotations.RangeAttriute	Permet de limiter un nombre à un intervalle donné	`[Range(128,156)]`
System.ComponentModel .PhoneAnnotations	Permet d'indiquer que la chaîne de caractères est un numéro de téléphone	`[Phone]`

TABLE 7.3. – Les attributs communs

Il en existe beaucoup d'autres, pour gérer les dates, les expressions régulières... Vous pouvez vous-mêmes en créer si cela vous est nécessaire.
Avec ces attributs, décrivons notre classe Article. Par exemple de cette manière :

Contenu masqué n°4

Maintenant, au sein de la méthode `Create(Article article)`, nous allons demander de valider les données.
Pour cela, il faut utiliser `ModelState.IsValid`.

```
1 if (ModelState.IsValid)
2             {
3                 _repository.AddArticle(article);
4                 return RedirectToAction("List", "Article"); // Voir
                      explication dessous
5             }
```

```
6            return View(article);
```

Listing 10 – Validation des données et enregistrement

7.3.4.2. Redirections

Comme nous l'avons vu dans le diagramme de flux, nous allons devoir rediriger l'utilisateur vers la page adéquate dans deux cas :

— erreur fatale ;
— réussite de l'action.

Dans le premier cas, il faut se rendre compte que, de base, ASP.NET MVC, enregistre des "filtres d'exception" qui, lorsqu'une erreur se passe, vont afficher une page "customisée". Le comportement de ce filtre est celui-ci :

— si l'exception attrapée hérite de `HttpException`, le moteur va essayer de trouver une page "personnalisée" correspondant au code d'erreur ;
— sinon, il traite l'exception comme une erreur 500.

Pour "customiser" la page d'erreur 500, il faudra attendre un peu, ce n'est pas nécessaire maintenant, au contraire, les informations de débogage que vous apportent la page par défaut sont très intéressantes.

Ce qui va nous intéresser, c'est plutôt la redirection "quand tout va bien".

Il existe deux méthodes pour rediriger selon vos besoin. La plupart du temps, vous utiliserez `RedirectToAction`. Cette méthode retourne un objet `ActionResult`.

Pour une question de lisibilité, je vous propose d'utiliser `RedirectToAction` avec deux (ou trois selon les besoins) arguments :

```
1 RedirectToAction("NomDeL'action", "Nom du Contrôleur");//une
      redirection simple
```

```
1 RedirectToAction("List","Article",new {page= 0});///Une redirection
      avec des paramètres pour l'URL
```

7.3.5. Le code final

À chaque partie le code final est donnée, mais il ne faut pas le copier bêtement. Pour bien comprendre les choses, il faut regarder pas à pas les différentes actions, mettre des points des arrêts dans les contrôleurs et regarder les différents objets.

FIGURE 7.2. – Mon point d'arrêt

7.3.5.1. L'entité Article

Contenu masqué n°5

7.3.5.2. La vue Create

Contenu masqué n°6

7.3.5.3. Le contrôleur

Contenu masqué n°7

7.4. Cas d'étude numéro 3 : envoyer une image pour illustrer l'article

Notre prochain cas d'étude sera le cas de l'ajout d'une image *thumbnail* (miniature) pour nos articles.
Comme nous allons lier un nouvel objet à notre modèle d'article, il nous faut modifier notre classe `Article` de manière à ce qu'elle soit prête à recevoir ces nouvelles informations.
En fait la difficulté ici, c'est que vous allez transmettre un **fichier**. Un fichier, au niveau de ASP.NET, c'est représenté par l'objet `HttpPostedFileBase`. Cet objet ne va pas être sauvegardé avec JSON ou SQL. Ce que vous allez sauvegarder, c'est le chemin vers le fichier.
Du coup, la première idée qui vous vient à l'esprit, c'est de rajouter une propriété dans votre classe Article :

10. Une technique avancée mais néanmoins courante dans le web moderne est l'utilisation de JavaScript avec l'objet XMLHttpRequest. L'acronyme qui désigne cette utilisation est *AJAX*.
11. La liste complète des types de champs HTML se trouve ici .
12. On parle d'indempotence .
13. Le protocole HTTPS garantit la confidentialité (i.e ce qui est transmis est secret) et l'authenticité (i.e que le site est bien qui il prétend être) et pour cela il faut acheter un certificat .

```
        /// <summary>
        /// Le chemin vers le fichier image
        /// </summary>
        public string ImageName { get; set; }
```

Alors comment faire pour qu'un fichier soit téléchargé ?
Une technique utilisable et que je vous conseille pour débuter, c'est de créer une nouvelle classe qu'on appelle souvent un **modèle de vue**[15]. Ce modèle de vue est une classe normale, elle va simplement être adaptée au formulaire que nous désirons mettre en place.
Créez un fichier `ArticleViewModels.cs` et à l'intérieur créez une classe "ArticleCreation" :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Web;

namespace TutoBlog.Models
{
    public class ArticleCreation
    {
        /// <summary>
        /// Le pseudo de l'auteur
        /// </summary>
        [Required(AllowEmptyStrings = false)]
        [StringLength(128)]
        [RegularExpression(@"^[^,\.\^+$")]
        [DataType(DataType.Text)]
        public string Pseudo { get; set; }
        /// <summary>
        /// Le titre de l'article
        /// </summary>
        [Required(AllowEmptyStrings = false)]
        [StringLength(128)]
        [DataType(DataType.Text)]
        public string Titre { get; set; }
        /// <summary>
        /// Le contenu de l'article
        /// </summary>
        [Required(AllowEmptyStrings = false)]
        [DataType(DataType.Text)]
        public string Contenu { get; set; }

        /// <summary>
        /// Le fichier image
        /// </summary>
        [DisplayName("Illustration")]
```

```
38          public HttpPostedFileBase Image{ get; set; }
39      }
40  }
```

Dans notre contrôleur, nous allons réécrire la méthode **Create**. Et au lieu de demander un **Article**, nous allons demander un **ArticleCreation**. Il faut aussi modifier notre vue **Create** pour prendre un **ArticleCreation** en entrée.
Il nous faudra aussi ajouter un champ au formulaire, qui accepte de télécharger un fichier :

```
1           <div class="form-group">
2               @Html.LabelFor(model => model.Image, htmlAttributes:
                    new { @class = "control-label col-md-2" })
3               <div class="col-md-10">
4                   @Html.TextBoxFor(model => model.Image, new { type =
                        "file", accept = "image/jpeg,image/png" })
5                   @* ou  <input type="file" name="Image"/>*@
6                   @Html.ValidationMessageFor(model => model.Image,
                        "", new { @class = "text-danger" })
7               </div>
8           </div>
```

Un formulaire, à la base, n'est pas fait pour transporter des données aussi complexes que des fichiers. Du coup, il faut les **encoder**. Et pour cela, il faut préciser un attribut de la balise `form` : `enctype="multipart/form-data"`. Si vous avez bien suivi les bases de Razor, vous avez compris qu'il faut modifier l'appel à la méthode `Html.BeginForm`.

Nous allons donc enregistrer le fichier une fois que ce dernier est reçu - surtout **quand** il est reçu.
La méthode `SaveAs` sert à cela.
Par convention, ce téléchargement se fait dans le dossier Content ou un de ses sous-dossiers.
Ensuite nous allons retenir le chemin du fichier.
Attention, pour des raisons de sécurité, il est impératif d'imposer des limites à ce qui est téléchargeable. Il est toujours conseillé de procéder à des vérifications dans le contrôleur :

- une image = des fichiers jpeg, png ou gif ;
- dans tous les cas, il faut limiter la taille du téléchargement (sur ZdS, la limite est par exemple à 1024 Ko).

Voici ce que cela peut donner :

```
1           private static string[] AcceptedTypes = new string[] {
                "image/jpeg", "image/png" };
2           private static string[] AcceptedExt = new string[] {
                "jpeg", "jpg", "png", "gif" };
3
4           //POST : Create
5           [HttpPost]
6           [ValidateAntiForgeryToken]
```

```
7           public ActionResult Create(ArticleCreation articleCreation)
8           {
9               if (!ModelState.IsValid)
10              {
11                  return View(articleCreation);
12              }
13
14              string fileName = "";
15              if (articleCreation.Image != null)
16              {
17                  bool hasError = false;
18                  if (articleCreation.Image.ContentLength > 1024 *
                        1024)
19                  {
20                      ModelState.AddModelError("Image",
                            "Le fichier téléchargé est trop grand.");
21                      hasError = true;
22                  }
23
24                  if
                        (!AcceptedTypes.Contains(articleCreation.Image.ContentType)
25                           ||
                                AcceptedExt.Contains(Path.GetExtension(articleCreat
26                  {
27                      ModelState.AddModelError("Image",
                            "Le fichier doit être une image.");
28                      hasError = true;
29                  }
30
31                  try
32                  {
33                      fileName =
                            Path.GetFileName(articleCreation.Image.FileName);
34                      string imagePath =
                            Path.Combine(Server.MapPath("~/Content/Upload"),
                            fileName);
35                      articleCreation.Image.SaveAs(imagePath);
36                  }
37                  catch
38                  {
39                      fileName = "";
40                      ModelState.AddModelError("Image",
                            "Erreur à l'enregistrement.");
41                      hasError = true;
42                  }
43                  if (hasError)
44                      return View(articleCreation);
45              }
46
47              Article article = new Article
```

```
48              {
49                  Contenu = articleCreation.Contenu,
50                  Pseudo = articleCreation.Pseudo,
51                  Titre = articleCreation.Titre,
52                  ImageName = fileName
53              };
54
55              _repository.AddArticle(article);
56              return RedirectToAction("List", "Article");
57          }
```

Nous souhaitons maintenant afficher l'image dans notre liste d'article, pour cela il n'y a pas de helper magique dans Razor. Il faudra utiliser simplement la balise `<img />`.

7.4.1. Les problèmes qui se posent

Je vous présente ce cas d'étude avant de passer à la base de données pour une raison simple : un fichier, qu'il soit une image ou non, ne doit **presque jamais** être stocké dans une base de données.
Si vous ne suivez pas cette règle, vous risquez de voir les performances de votre site baisser de manière affolantes.
Un fichier, ça se stocke généralement dans un **système de fichiers**. Pour faire plus simple, on le rangera dans un **dossier**.

7.4.1.1. Doublons

Essayons de donner à deux articles différents des images qui ont le même nom. Que se passe-t-il actuellement ?
L'image déjà présente est écrasée.
Pour éviter ce genre de problèmes, il faut absolument s'assurer que le fichier qui est créé lors d'un téléchargement est **unique**.
Vous trouverez deux écoles pour rendre unique un nom de fichier, ceux qui utilisent un marquage temporel (le nombre de secondes depuis 1970) et ceux qui utilisent un *identifiant*.
Comme nous n'utilisons pas encore de base de données, notre identifiant, c'est nous qui allons devoir le générer. Et une des meilleurs manières de faire un bon identifiant, c'est d'utiliser un *Global Unique Identifier* connu sous l'acronyme `GUID`.
Le code pour générer un tel identifiant est simple :

```
1 Guid id = Guid.NewGuid();
2 string imageFileName = id.toString()+"-"+model.Image.FileName;
```

7.4.1.2. Les caractères spéciaux

Nouveau problème : nous désirons **afficher** l'image. Pour cela, nous allons modifier le fichier de vue List pour qu'à chaque article, on ajoute une image.
Typiquement, la balise `<img>` s'attend à ce que vous lui donniez un lien. Comme nous avons sauvegardé notre image dans le dossier `Content/Uploads`, la balise ressemblera à : `<img`

`src="/Content/Uploads/135435464-nomdufichier.jpg/>`.
C'est là qu'arrive le problème : vos visiteurs n'utiliseront pas forcément des noms de fichiers qui donnent des URL sympathiques. En effet, les systèmes d'exploitation modernes savent utiliser les accents, les caractères chinois... mais ce n'est pas le cas des URL.
Pour régler ce problème, nous allons utiliser ce qu'on appelle un **slug**.
Les slugs sont très utilisés dans les liens des sites de presse, car ils ont souvent un impact positif sur le *SEO*[14]. Prenons un exemple :
`nextinpact.com/news/89405-gps-europeen-deux-satellites-galileo-bien-lances-mais-mal-positionnes.htm`
Vous avez un lien en trois parties :

- `nextinpact.com` : le nom de domaine ;
- `/news/` : l'équivalent pour nous du segment "{controller}" ;
- `89405-gps-europeen-deux-satellites-galileo-bien-lances-mais-mal-positionnes.htm` : le slug, qui permet d'écrire une URL sans accents, espaces blancs... Et pour bien faire, ils ont même un identifiant unique en amont.

Pour utiliser les slugs, une possibilité qui s'offre à vous est d'utiliser un package nommé *slugify*.
Dans l'explorateur de solutions, faites un clic-droit sur le nom du projet puis sur "gérer les packages" et installez slugify.

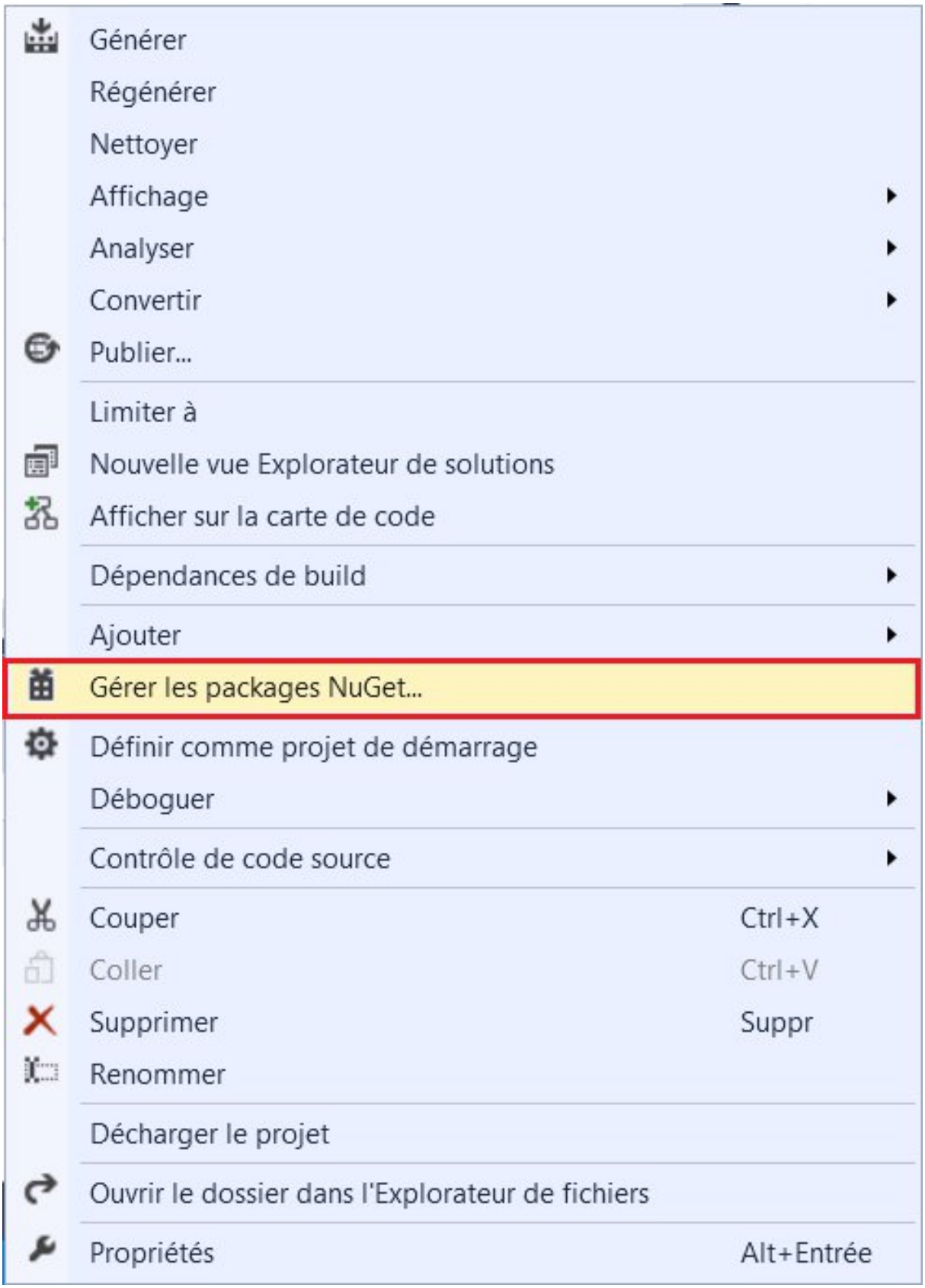


FIGURE 7.3. – Gérer les packages

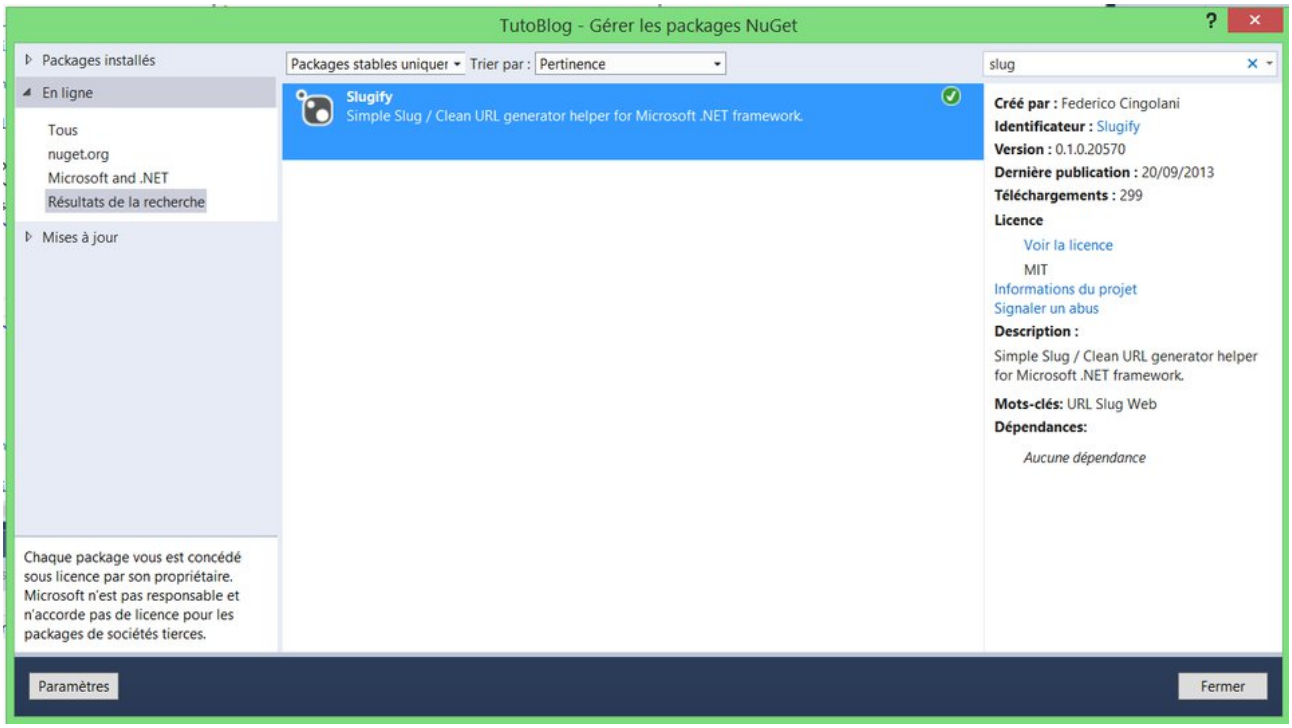


FIGURE 7.4. – Installer slugify

Pour utiliser slugify, cela se passe ainsi :

```
1 string fileSlug = new SlugHelper().GenerateSlug(filePath);
```

7.5. Cas d'étude numéro 4 : la pagination

7.5.1. Ce qu'est la pagination

La pagination, c'est une technique très utile quand vous devez afficher des listes très longues de contenu.
Sur un blog, vous aurez - à partir d'un moment - une liste assez importante d'articles. Disons pour faire simple que vous en avez 42.
Pensez-vous qu'afficher les 42 articles sur la page d'accueil soit une bonne chose ?
Autant d'articles, c'est long, c'est lourd. Cela ralentirait le chargement de votre page et ça forcerait vos visiteurs à utiliser l'ascenseur vertical un trop grand nombre de fois.
De ce fait, vous allez par exemple décider de n'afficher que 5 articles, et de mettre les 5 suivants dans une autre page et ainsi de suite.
Votre but sera donc d'afficher la liste des articles puis de mettre à disposition de vos utilisateurs un bouton "page précédente", un autre "page suivante", et un formulaire permettant de se rendre à une page quelconque.
Voici un petit aperçu du contrôle proposé :

14. Search Engine Optimisation : un ensemble de techniques qui permettent d'améliorer votre classement dans les résultats des grands moteurs de recherche.
15. En anglais on dit ViewModel .

7.5.2. S'organiser pour répondre au problème

Comme c'est notre dernier cas d'étude pour ce chapitre, vous allez utiliser toutes vos connaissances apprises jusqu'ici :

1. dans le contrôleur Article, modifiez la méthode List en `public ActionResult List(int page = 0)` ;
2. utilisez la bonne méthode pour récupérer vos articles, voir `ArticleJSONRepository` ;
3. ajoutez les liens Précédent/Suivant dans la vue, `@Html.ActionLink()` ;
4. déterminez si vous devez utiliser un formulaire GET ou POST et créez-le (URL de type url/Article/List ?page=0).

Contenu masqué n°8

7.6. Se protéger contre les failles de sécurité

Un site internet est un point d'entrée important pour pas mal d'attaques. Certaines de ces attaques sont brutales et pour vous protéger, vous aurez besoin d'une infrastructure importante, ce qui est souvent la responsabilité de votre *hébergeur*.
D'autres, à l'opposé, sont plus pernicieuses, car elles se servent de failles présentes au sein-même de votre code.
Nous allons voir quelques-unes des failles les plus connues sur le web, et surtout que ASP.NET vous aide énormément à vous en protéger.

7.6.0.1. La faille CSRF

Pour vous souhaiter la bienvenue dans le vaste monde des failles de sécurité, voici votre premier acronyme : CSRF, pour **Cross Site Request Forgery**[16].
Cette faille s'attaque aux formulaires qui demandent authentification. Elle tire profit du fait que créer une requête HTTP à partir de rien est très facile.
Surtout, elle s'aide beaucoup du *social engineering* et du réflexe pavlovien des gens : cliquer sur tous les liens qu'ils voient...
Cette attaque consiste à usurper l'identité d'un membre qui a le droit de gérer le contenu (créer, éditer, supprimer) et à envoyer un formulaire avec les données adéquates au site.
Faisons un petit schéma pour bien comprendre.
Au départ nous avons deux entités séparées :

— un utilisateur qui est connecté sur le vrai site avec ses identifiants ;
— un pirate qui va créer un site ou un email frauduleux pour piéger cet utilisateur.

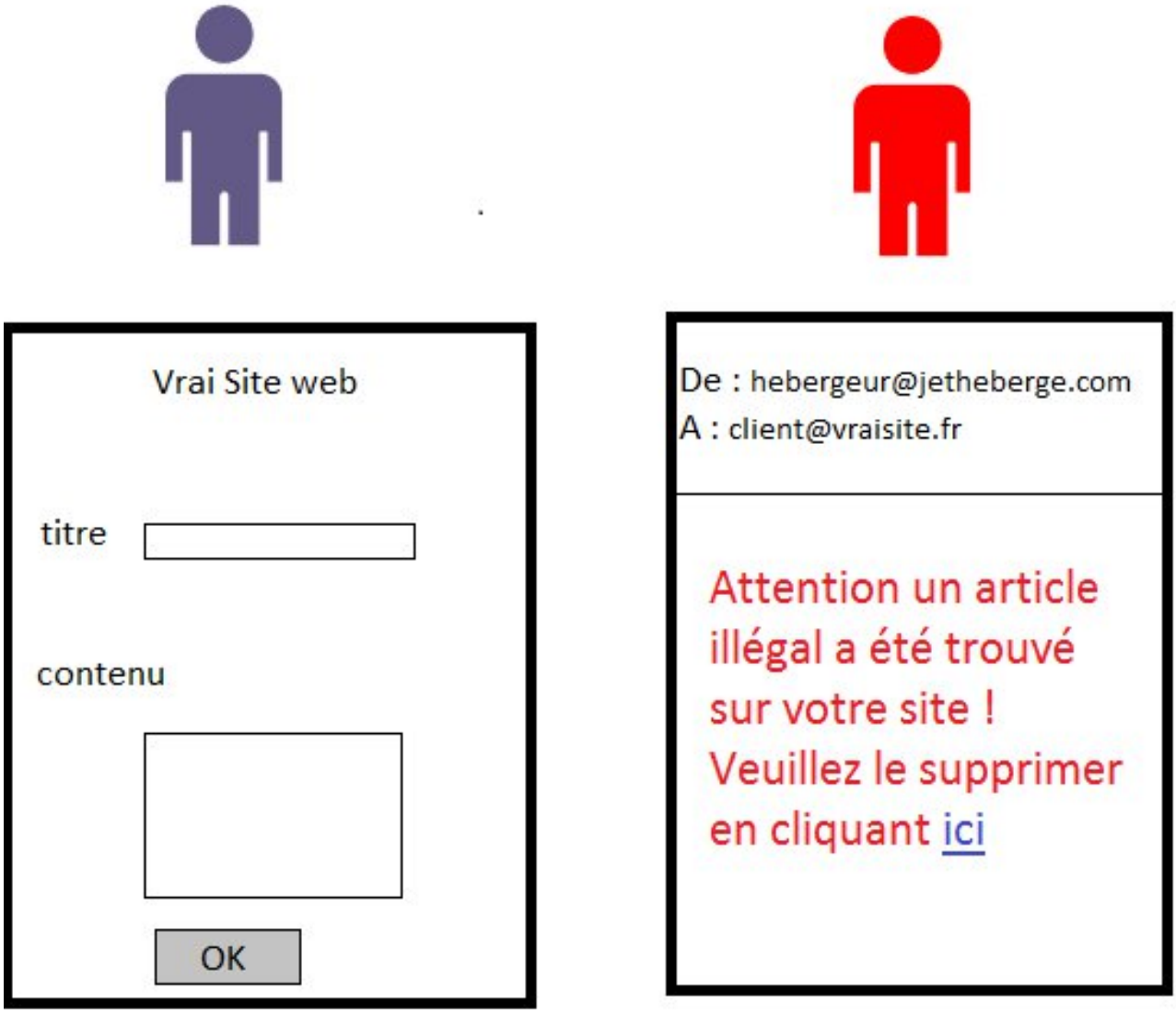


FIGURE 7.5. – Préparation de l'attaque

Le fameux lien sur lequel l'utilisateur va cliquer va en fait être un faux site qui ne fera qu'une chose : envoyer une requête pour supprimer tel ou tel contenu.
Heureusement, il existe une parade.
Premièrement il est absolument **primordial** que vous utilisiez des formulaires POST quand vous voulez modifier des données dans votre base. En effet si vous laissez un contrôleur supprimer des données alors que tout passe en GET, une simple balise image du type `<img src="http://votre.site.fr/supprimer/objet/1/?param=valeur"/>` suffira à vous "piéger". L'image ne s'affichera jamais mais votre navigateur interprétera automatiquement cette balise comme "envoie une requête au site" et vous allez silencieusement supprimer des données.
Le formulaire POST n'est pas suffisant, mais il permet d'utiliser une astuce pour se protéger définitivement contre cette faille.
En session, vous allez enregistrer un nombre généré aléatoirement à chaque fois que vous affichez une page.

Sur la page honnête, donc sur le vrai site, vous allez ajouter à tous les formulaires un champ caché qui s'appellera "csrf_token" par exemple et qui aura pour valeur ce nombre impossible à deviner.
Puis, dans votre contrôleur, vous allez vérifier que ce nombre est bien le même que celui enregistré en session.
Comme ça, tout autre site extérieur qui essaierait de créer une requête sera **immédiatement rejeté** et vous vous en rendrez compte.
Comme mettre en place tout ça risquerait d'être long à coder à chaque fois, ASP.NET vous aide.
Lorsque vous créez un formulaire de type POST, il suffit d'ajouter deux choses :

— dans le formulaire Razor, ajoutez : `@Html.AntiForgeryToken()` ;
— au niveau de la méthode `Create`, `Edit` ou `Delete` de votre contrôleur, ajoutez l'attribut `[ValidateAntiForgeryToken]`.

Voilà, vous êtes protégés !

7.6.0.2. La faille XSS

La faille "XSS" est une faille de type "injection de code malveillant", et elle est une des failles les plus connues et les plus faciles à corriger.
Une faille XSS survient lorsqu'une personne essaie d'envoyer du code HTML dans votre formulaire.
Imaginons qu'il envoie `<strong>Je suis important</strong>`. Si vous ne vous protégez pas contre la faille XSS, vous verrez sûrement apparaître la phrase "Je suis important" en gras.
Maintenant, cas un peu plus pernicieux, la personne vous envoie "</body>je suis important".
Et c'est le design complet de la page qui tombe.
Malheureusement, ça n'est pas tout. La personne peut envoyer un script JavaScript qui redirigera vos visiteurs vers un site publicitaire, ou bien illégal. C'est de ce comportement-là que vient le nom XSS : Cross Site Scripting.
Comme je vous l'ai dit, la parade est simple : il suffit de remplacer les caractères `><"` par leur équivalent HTML (`&gt; &lt; &quot;`). Ainsi les balises HTML sont considérées comme du texte simple et non plus comme des balises.
Et comme c'est un comportement qui est très souvent désiré, c'est le comportement par défaut de Razor quand vous écrivez `@votrevariable`.
Pour changer ce comportement, il faut créer un *helper* qui va par exemple autoriser les balises bien construites (`<strong></strong>`), interdire les balises `<script>` et les attributs spéciaux (`onclick` par exemple).

Notre blog commence à ressembler à quelque chose. C'est bien, mais la suite est encore longue.

16. Plus d'informations sur wikipédia

Contenu masqué

Contenu masqué n°3

```
using Blog.Models;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace Blog.Controllers
{
    public class ArticleController : Controller
    {
        /// <summary>
        /// Champ qui va permettre d'appeler des méthodes pour faire des actions sur notre fichier
        /// </summary>
        private readonly ArticleJSONRepository _repository;

        /// <summary>
        /// Constructeur par défaut, permet d'initialiser le chemin du fichier JSON
        /// </summary>
        public ArticleController()
        {
            string path = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "App_Data", "liste_article_tuto_full.json");
            _repository = new ArticleJSONRepository(path);
        }

        // GET: List
        public ActionResult List()
        {
            try
            {
                List<Article> liste = _repository.GetAllListArticle().ToList();
                return View(liste);
            }
            catch
            {
                return View(new List<Article>());
            }
        }
    }
```

```
41 }
```

Listing 11 – Le contrôleur

```
1  @model IEnumerable<Blog.Models.Article>
2
3  @{
4      ViewBag.Title = "Blog";
5  }
6
7  <p>
8      @Html.ActionLink("Create New", "Create")
9  </p>
10 <table class="table">
11     <tr>
12         <th>
13             @Html.DisplayNameFor(model => model.Pseudo)
14         </th>
15         <th>
16             @Html.DisplayNameFor(model => model.Titre)
17         </th>
18         <th>
19             @Html.DisplayNameFor(model => model.Contenu)
20         </th>
21         <th></th>
22     </tr>
23
24 @foreach (var item in Model) {
25     <tr>
26         <td>
27             @Html.DisplayFor(modelItem => item.Pseudo)
28         </td>
29         <td>
30             @Html.DisplayFor(modelItem => item.Titre)
31         </td>
32         <td>
33             @Html.DisplayFor(modelItem => item.Contenu)
34         </td>
35         <td>
36             @Html.ActionLink("Edit", "Edit", new { /*
                   id=item.PrimaryKey */ }) |
37             @Html.ActionLink("Details", "Details", new { /*
                   id=item.PrimaryKey */ }) |
38             @Html.ActionLink("Delete", "Delete", new { /*
                   id=item.PrimaryKey */ })
39         </td>
40     </tr>
41 }
```

```
42
43 </table>
```

Listing 12 – La vue

Retourner au texte.

Contenu masqué n°4

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel.DataAnnotations;
4  using System.Linq;
5  using System.Web;
6
7  namespace Blog.Models
8  {
9      /// <summary>
10     /// Notre modèle de base pour représenter un article
11     /// </summary>
12     public class Article
13     {
14         /// <summary>
15         /// Le pseudo de l'auteur
16         /// </summary>
17         [Required(AllowEmptyStrings=false)]
18         [StringLength(128)]
19         [RegularExpression(@"^[^,\.\^]+$")]
20         [DataType(DataType.Text)]
21         public string Pseudo { get; set; }
22         /// <summary>
23         /// Le titre de l'article
24         /// </summary>
25         [Required(AllowEmptyStrings = false)]
26         [StringLength(128)]
27         [DataType(DataType.Text)]
28         public string Titre { get; set; }
29         /// <summary>
30         /// Le contenu de l'article
31         /// </summary>
32         [Required(AllowEmptyStrings=false)]
33         [DataType(DataType.MultilineText)]
34         public string Contenu { get; set; }
35     }
36 }
```

Retourner au texte.

Contenu masqué n°5

```
public class Article
    {
        /// <summary>
        /// Le pseudo de l'auteur
        /// </summary>
        [Required(AllowEmptyStrings=false)]
        [StringLength(128)]
        [RegularExpression(@"^[^,\.\^]+$")]
        [DataType(DataType.Text)]
        public string Pseudo { get; set; }
        /// <summary>
        /// Le titre de l'article
        /// </summary>
        [Required(AllowEmptyStrings = false)]
        [StringLength(128)]
        [DataType(DataType.Text)]
        public string Titre { get; set; }
        /// <summary>
        /// Le contenu de l'article
        /// </summary>
        [Required(AllowEmptyStrings=false)]
        [DataType(DataType.MultilineText)]
        public string Contenu { get; set; }
    }
```

Retourner au texte.

Contenu masqué n°6

```
@model Blog.Models.Article

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>


@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Article</h4>
        <hr />
```

```
17          @Html.ValidationSummary(true, "", new { @class =
                "text-danger" })
18          <div class="form-group">
19              @Html.LabelFor(model => model.Pseudo, htmlAttributes:
                    new { @class = "control-label col-md-2" })
20              <div class="col-md-10">
21                  @Html.EditorFor(model => model.Pseudo, new {
                        htmlAttributes = new { @class = "form-control"
                        } })
22                  @Html.ValidationMessageFor(model => model.Pseudo,
                        "", new { @class = "text-danger" })
23              </div>
24          </div>
25
26          <div class="form-group">
27              @Html.LabelFor(model => model.Titre, htmlAttributes:
                    new { @class = "control-label col-md-2" })
28              <div class="col-md-10">
29                  @Html.EditorFor(model => model.Titre, new {
                        htmlAttributes = new { @class = "form-control"
                        } })
30                  @Html.ValidationMessageFor(model => model.Titre,
                        "", new { @class = "text-danger" })
31              </div>
32          </div>
33
34          <div class="form-group">
35              @Html.LabelFor(model => model.Contenu, htmlAttributes:
                    new { @class = "control-label col-md-2" })
36              <div class="col-md-10">
37                  @Html.TextAreaFor(model => model.Contenu, new {
                        htmlAttributes = new { @class = "form-control"
                        }, cols = 35, rows = 10 })
38                  @Html.ValidationMessageFor(model => model.Contenu,
                        "", new { @class = "text-danger" })
39              </div>
40          </div>
41
42          <div class="form-group">
43              <div class="col-md-offset-2 col-md-10">
44                  <input type="submit" value="Create"
                        class="btn btn-default" />
45              </div>
46          </div>
47      </div>
48  }
49
50  <div>
51      @Html.ActionLink("Back to List", "List")
52  </div>
```

```
53
54 <script src="~/Scripts/jquery-1.10.2.min.js"></script>
55 <script src="~/Scripts/jquery.validate.min.js"></script>
56 <script
       src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
```

Retourner au texte.

Contenu masqué n°7

```
1          //GET : Create
2          public ActionResult Create()
3          {
4              return View();
5          }
6
7          //POST : Create
8          [HttpPost]
9          [ValidateAntiForgeryToken]
10         public ActionResult Create(Article article)
11         {
12             if (ModelState.IsValid)
13             {
14                 _repository.AddArticle(article);
15                 return RedirectToAction("List", "Article");
16             }
17             return View(article);
18         }
```

Retourner au texte.

Contenu masqué n°8

```
1          public readonly static int ARTICLEPERPAGE = 5;
2
3          // GET: List
4          public ActionResult List(int page = 0)
5          {
6              try
7              {
8                  List<Article> liste =
                       _repository.GetListArticle(page *
                       ARTICLEPERPAGE, ARTICLEPERPAGE).ToList();
9                  ViewBag.Page = page;
10                 return View(liste);
11             }
```

```
            catch
            {
                return View(new List<Article>());
            }
        }
```

Code : le contrôleur

```
@model IEnumerable<Blog.Models.Article>

@{
    ViewBag.Title = "Blog";
}

<section>
    <p>
        @Html.ActionLink("Create New", "Create")
    </p>
    <table class="table">
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Pseudo)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Titre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contenu)
            </th>
            <th></th>
        </tr>

        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Pseudo)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Titre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Contenu)
                </td>
                <td>
                    @{
                        if
                            (!string.IsNullOrWhiteSpace(item.ImageName))
                        {
```

```
41                          <img
                                src="~/Content/Upload/@item.ImageName"
                                alt="@Path.GetFileNameWithoutExtension(item.ImageN
                                />
42                      }
43                  }
44              </td>
45              <td>
46                  @Html.ActionLink("Edit", "Edit", new { /*
                        id=item.PrimaryKey */ }) |
47                  @Html.ActionLink("Details", "Details", new { /*
                        id=item.PrimaryKey */ }) |
48                  @Html.ActionLink("Delete", "Delete", new { /*
                        id=item.PrimaryKey */ })
49              </td>
50          </tr>
51      }
52  </table>
53
54  <footer class="pagination-nav">
55      <ul>
56          <li>
57              @if ((int)ViewBag.Page > 0)
58              {
59                  @Html.ActionLink("Précédent", "List",
                        "Article", new { page = ViewBag.Page - 1 },
                        new { @class = "button" })
60              }
61          </li>
62          <li>
63              @if (Model.Count() ==
                    Blog.Controllers.ArticleController.ARTICLEPERPAGE
                    || ViewBag.Page > 0)
64              {
65                  using (Html.BeginForm("List", "Article",
                        FormMethod.Get))
66                  {
67                      @Html.TextBox("page", 0, new { type =
                            "number" })
68                      <button type="submit">Aller à</button>
69                  }
70              }
71          </li>
72          <li>
73              @if (Model.Count() ==
                    Blog.Controllers.ArticleController.ARTICLEPERPAGE)
74              {
75                  @Html.ActionLink("Suivant", "List", "Article",
                        new { page = ViewBag.Page + 1 }, new {
                        @class = "button" })
```

```
                }
            </li>
        </ul>
    </footer>
</section>
```

Code : La vue Retourner au texte.

Troisième partie

Gérer la base de données avec Entity Framework Code First

8. Entity Framework : un ORM qui vous veut du bien

Cette partie va présenter un aspect crucial de la programmation d'une application web : l'accès aux données. Comme vous avez fait un peu de C#, vous avez peut être l'habitude d'utiliser une base de données et d'effectuer des requêtes SQL via des méthodes fournies par le framework .NET. Un ORM va nous permettre d'accéder à une source de données sans que nous ayons la sensation de travailler avec une base de données.
Cela paraît étrange, mais signifie simplement que grâce à cet ORM, nous n'allons plus écrire de requêtes, ni créer de tables, etc., via un système de gestion de base de données mais directement manipuler les données dans notre code C#. Un ORM très populaire avec C# est **Entity Framework**.

8.1. ORM ? Qu'est-ce que c'est ?

8.1.1. Principe d'un ORM

Comme dit en introduction, ORM signifie que l'on va traduire de l'Objet en Relationnel et du Relationnel en Objet.

? Mais dans cette histoire, c'est quoi "objet" et "relationnel" ?

Le plus simple à comprendre, c'est "l'objet". Il s'agit ici simplement des classes que manipule votre programme. Ni plus, ni moins.
Par exemple, dans notre blog nous avons des articles, pour lesquels nous avons créé une classe `Article`, qui décrit ce qu'est un article du point de vue de notre programme. Chouette, non ?
Maintenant, le gros morceau : le "Relationnel".
Nous ne parlons pas ici des relations humaines, ni même des relations mathématiques mais de la *représentation relationnelle des données*.
Ce terme vient de l'histoire même des bases de données[17]. Ces dernières ont été créées pour répondre à des principes complexes[19] et une logique ensembliste.
De ce fait, les créateurs des bases de données, notamment M.Bachman vont créer une architecture qui va représenter les données en créant des *relations* entre elles.
Plus précisément, les données vont être *normalisées* sous forme **d'entités** qui auront entre elles trois types d'interactions : One-To-One, One-To-Many, Many-To-Many.
Nous verrons la traduction de ces termes au cours de nos cas d'étude.
Plus tard, avec l'arrivée de la programmation orientée objet, les chercheurs dans le domaine des bases de données, vont tenter de modéliser d'une manière compatible avec UML le fonctionnement des bases.
Grâce à ces travaux les programmes ont pu traduire de manière plus immédiate les "tables de données" que l'on trouve dans les bases de données relationnelles en instances d'objets.
Dans le cadre de .NET l'ORM est souvent constitué d'une API qui se découpe en deux parties :

un gestionnaire d'objet/données et un "traducteur" qui se sert de Linq To SQL.
Grâce à cette organisation, l'ORM n'a plus qu'à se brancher aux connecteurs qui savent converser avec le système choisi (MSSQL, MySQL, Oracle, PostgreSQL...).

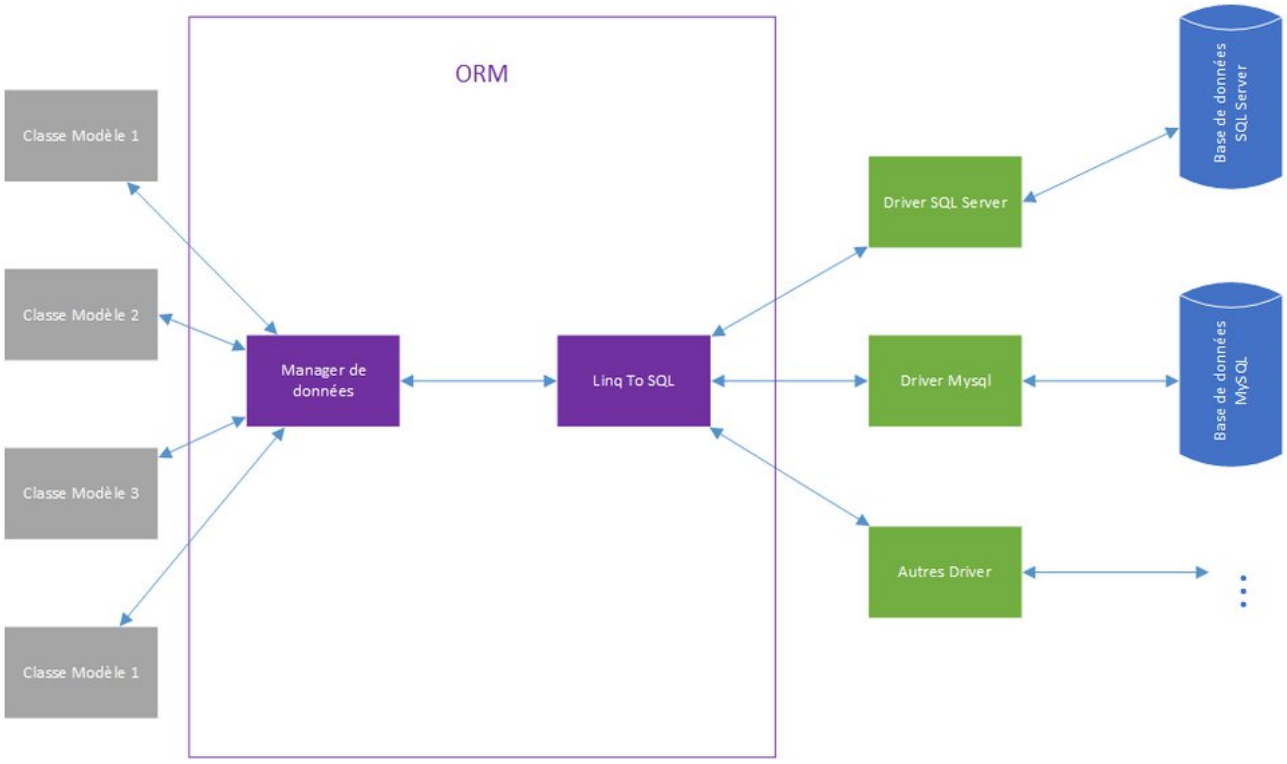


FIGURE 8.1. – Fonctionnement d'un ORM

8.1.2. Des alternatives à Entity Framework ?

Comme vous l'aurez compris, nous allons choisir *Entity Framework* en tant qu'ORM pour plusieurs raisons :

- c'est le produit mis en avant par Microsoft, il est régulièrement mis à jour (actuellement : version 6) ;
- il est bien documenté ;
- il est parfaitement intégré au reste du framework .NET (validation, exceptions, Linq...) ;
- il se base sur l'API standard ADO.NET ;
- il est complet tout en restant suffisamment simple d'utilisation pour le présenter à des débutants.

Comme nous avons pu nous en rendre compte dans les parties précédentes, ASP.NET ne nécessite pas un ORM. C'est juste un outil qui vous facilitera le travail et surtout qui vous assurera une véritable indépendance d'avec le système de gestion de bases de données que vous utilisez.
Le monde du .NET est néanmoins connu pour fournir un nombre important d'alternatives au cours du développement.
Une petite exploration des dépôts NuGet[18] vous permettra de vous en rendre compte.
Il existe en gros deux types d'ORM :

- les ORM complets, qui accèdent vraiment au meilleur des deux mondes objet et relationnel. Entity Framework en fait partie ;
- les ORM légers, qui sont là pour vous faciliter uniquement les opérations basiques, dites CRUD.

8.1.2.1. Un ORM complet : NHibernate

NHibernate est un ORM qui se veut être le concurrent direct de Entity Framework.
Son nom est un mot valise rassemblant ".NET" et "Hibernate". Il signifie que NHibernate est le portage en .NET de l'ORM Hibernate, bien connu des développeurs Java et Java EE.
De la même manière que Entity Framework, NHibernate se base sur ADO.NET. Par contre, il reprend un certain nombre de conventions issues du monde Java comme l'obligation de posséder un constructeur par défaut.

8.1.2.2. Un ORM léger : PetaPOCO

PetaPOCO est un ORM qui s'est construit par la volonté grandissante des développeurs à utiliser des *POCO*.
POCO, c'est un acronyme anglais pour Plain Old CLR/C# Object, autrement dit, un objet qui ne dépend d'aucun module externe. En Java, vous retrouvez également le concept de POJO, pour Plain Old Java Object.
Le problème posé par les ORM tels que NHibernate et EntityFramework, c'est que rapidement vos classes de modèles vont s'enrichir d'annotations, d'attributs ou de méthodes qui sont directement liés à ces ORM.
Or, en informatique, on n'aime pas que les classes soient fortement couplées, car cela complique beaucoup la maintenance et l'évolutivité du code. On essaie donc au maximum de respecter le principe de responsabilité unique (SRP).
Nous verrons plus tard que dès que l'accès à la base de données est nécessaire, nous préférons utiliser le patron de conception Repository .
L'utilisation d'un ORM complet peut rapidement mener à des *repository* assez lourds à coder et à organiser.
PetaPOCO vous permet donc d'utiliser vos classes de modèle tout simplement. Par contre certaines fonctionnalités ne sont pas implémentées.

8.2. Les différentes approches pour créer une base de données

Nous désirons créer un site web. Ce site web va devoir stocker des données. Ces données ont des relations entre elles...
Bref, nous allons passer par une base de données. Pour simplifier les différentes questions techniques, nous avons fait un choix pour vous : nous utiliserons une base de données **relationnelle**, qui se base donc sur le SQL.
Maintenant que ce choix est fait, il va falloir s'atteler à la création de notre base de données et à la mise en place de notre ORM. Comme ce tutoriel utilise Entity Framework, nous utiliserons les outils que Microsoft met à notre disposition pour gérer notre base de données avec cet ORM précis.
Avant toute chose, je voulais vous avertir : une base de données, quelle que soit l'approche utilisée, ça se réfléchit au calme. C'est souvent elle qui déterminera les performances de votre application, la créer à la va-vite peut même tuer un petit site.
Maintenant, entrons dans le vif du sujet et détaillons les trois approches pour la création d'une base de données.

17. wikipédia
18. liste complète des ORM publiés sur NuGet
19. Les bases de données sont ACID .

8.2.1. L'approche *Model First*

L'approche *Model First* est une approche qui est issue des méthodes de conception classique . Vous l'utiliserez le plus souvent lorsque quelqu'un qui a des connaissances en base de données, notamment la méthode MERISE[20].
Cette approche se base sur l'existence de diagramme de base de données parfois appelés *Modèle Logique de Données*. Ces diagrammes présentent les entités comme des classes possédant des attributs et notamment une *clef primaire* et les relient avec de simples traits.
Pour en savoir plus sur cette approche, je vous propose de suivre cette petite vidéo :

Élément externe (Video) —
Consultez cet élément à l'adresse `https://www.youtube.com/embed/aGUhBt1Cf9M?feature=oembed`.
L'approche model first

8.2.2. L'approche *Database First*

Comme son nom l'indique, cette méthode implique que vous créiez votre base de données de A à Z en SQL puis que vous l'importiez dans votre code. C'est une nouvelle fois le *Modèle Logique de Données* qui fera le lien entre la base de données et l'ORM. La seule différence c'est que cette fois-ci c'est Visual Studio qui va générer le schéma par une procédure de *rétro ingénierie*.

8.2.3. L'approche *Code First*

Cette dernière approche, qui est celle que nous avons choisi pour illustrer ce tutoriel, part d'un constat : ce qu'un développeur sait faire de mieux, c'est coder.
Cette approche, qui est la plus commune dans le monde des ORM consiste à vous fournir trois outils pour que vous, développeur, n'ayez qu'à coder des *classes* comme vous le feriez habituellement et en déduire le modèle qui doit être généré.
Nous avons donc de la chance, nous pourrons garder les classes que nous avions codées dans la partie précédente, elles sont très bien telles qu'elles sont.

? Trois outils, ça fait beaucoup non ? Et puis c'est quoi ces outils ?

Quand vous allez définir un modèle avec votre code, vous allez devoir passer par ces étapes :

1. coder vos classes pour définir le modèle ;
2. décrire certaines liaisons complexes pour que le système puisse les comprendre ;
3. mettre à jour votre base de données ;
4. ajouter des données de test ou bien de démarrage (un compte admin par exemple) à votre base de données.

Tout cela demande bien trois outils. Heureusement pour vous, ces outils sont fournis de base dans Entity Framework, et nous en parlons dans le point suivant, soyez patients.

8.2.4. Comment choisir son approche

Quand vous démarrez un projet, vous allez souvent vous poser beaucoup de questions. Dans le cas qui nous intéresse, je vous conseille de vous poser trois questions :

— Est-ce qu'il y a **déjà** une base de données ? Si oui, il serait sûrement dommage de tout réinventer, vous ne croyez pas ?

— Est-ce que vous êtes seuls ? Si oui, vous faites **comme vous préférez**, c'est comme ça que vous serez le plus efficace, mais si vous êtes à plusieurs il vous faudra vous poser une troisième question primordiale...

— Qui s'occupe de la BDD : un codeur, ou un expert qui a réfléchi son modèle via un diagramme ? Il y a fort à parier que s'adapter à l'expertise de la personne qui gère la BDD est une bonne pratique, non ?

Pour être plus visuel, on peut exprimer les choses ainsi :

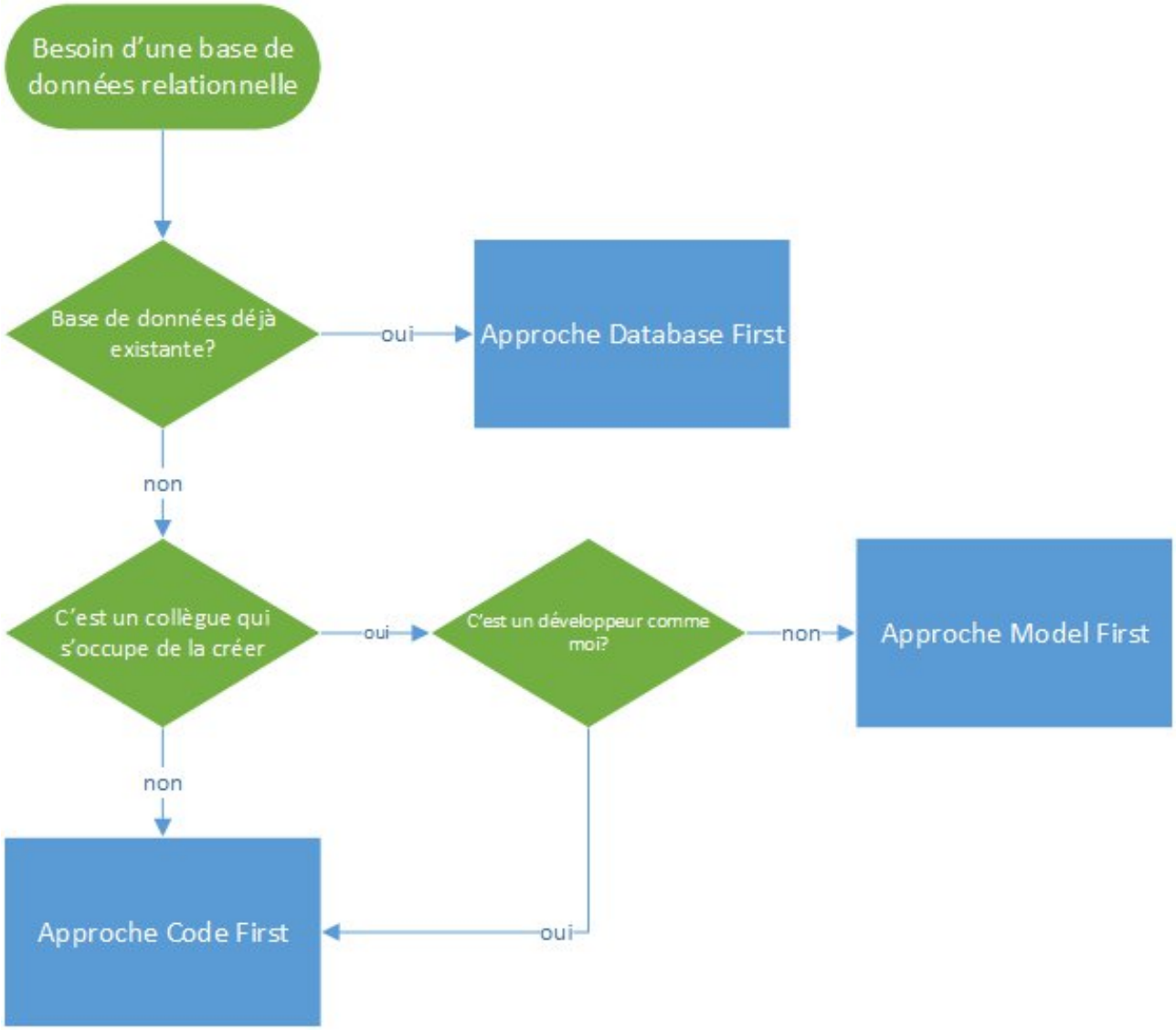


FIGURE 8.2. – Choix de l'approche

20. La Méthode MERISE est une méthode franco-française encore très répandue aujourd'hui pour gérer les projets et notamment les étapes de conception.

8.3. Présentation d'Entity Framework

À partir de ce chapitre nous utiliserons Entity Framework comme ORM et nous suivrons l'approche *Code First*.

Entity Framework, c'est le nom de l'ORM officiel de .NET. Développé par Microsoft, il est à l'heure actuelle en version 6. Vous pourrez observer l'arrivé d'une version mineure par semestre environ, parfois plus, parfois moins lorsque l'outil est très stable. Vous pouvez bien sûr le télécharger sur NuGet.
Je vous ai dit tout à l'heure qu'il fallait trois outils pour suivre l'approche *Code First*. Bien évidemment, Entity Framework les possède.

8.3.1. Un *mapper*

Le mot francisé serait "mappeur", et son principal but est de faire correspondre les objets POCO à des enregistrements en base de données.
C'est à ce *mapper* que sont accrochés les *connecteurs* qui vous permettront d'utiliser les différents systèmes de gestion de base de données. De base vous aurez droit à SQLServer LocalDB. Pour tester notre blog, c'est ce connecteur que nous utiliserons car il permet de créer rapidement des bases de données, et de les supprimer tout aussi rapidement une fois qu'on a tout cassé.

8.3.2. Une API de description

Quand vous utilisez un ORM, il faudra parfois décrire les relations entre les classes. En effet, certaines sont complexes et peuvent être interprétées de plusieurs manières, il vous faudra donc dire explicitement ce qu'il faut faire.
Vous pouvez aussi vouloir nommer vos champs et tables autrement que ce qu'indique la *convention*.
Pour cela, vous avez deux grandes méthodes avec EntityFramework.

8.3.2.1. Les annotations

Comme nous l'avions vu lorsque nous avons voulu vérifier la cohérence des données saisies par l'utilisateur, vous allez pouvoir décrire votre modèle avec des annotations.
Les annotations les plus importantes seront :

- `[Index]` : elle permet de s'assurer qu'une entrée est **unique** dans la table, et ainsi accélérer fortement les recherches.
- `[Key]` : si votre clef primaire ne doit pas s'appeler ID ou bien qu'elle est plus complexe, il faudra utiliser l'attribut Key.
- `[Required]` : il s'agit du même `[Required]` que pour les vues[21], il a juste un autre effet : il interdit les valeurs `NULL`, ce qui est une contrainte très forte.

21. Nous pourrons d'ailleurs observer que tous les filtres par défaut sont utilisés à la fois par les vues et Entity Framework. C'est une sécurité supplémentaire et une fonctionnalité qui vous évitera de produire encore plus de code.

8.3.2.2. La *Fluent API*

La seconde méthode a été nommée "Fluent API". Elle a été créée pour que ça soit encore une fois la production de code qui permette de décrire le modèle.
La Fluent API est **beaucoup plus avancée** en termes de fonctionnalités et vous permet d'utiliser les outils les plus optimisés de votre base de données.
Je ne vais pas plus loin dans l'explication de la Fluent API, nous ne l'observerons à l'action que bien plus tard dans le cours, lorsque nous aurons affaire aux cas complexes dont je vous parlais plus tôt.

8.3.3. Les migrations *Code First*

Troisième et dernier outil, qui bénéficiera de son propre chapitre tant il est important et puissant.
Les migrations sont de simples fichiers de code (on ne vous a pas menti en disant que c'était du Code First) qui expliquent à la base de données comment se mettre à jour au fur et à mesure que vous faîtes évoluer votre modèle.
Une migration se termine toujours par l'exécution d'une méthode appelée `Seed` qui ajoute ou met à jour des données dans la base de données, soit pour **tester** soit pour créer un environnement minimum (exemple, un compte administrateur).

9. La couche de sauvegarde : les entités

Nous allons coder notre application grâce à l'approche **code first**. L'avantage c'est que nous avons déjà codé une partie de nos modèles, il n'y aura plus qu'à les intégrer à Entity Framework.

9.1. L'entité, une classe comme les autres

Lorsque nous créons une application web, nous devons répondre à un **besoin métier**.
Ce besoin vous permet de comprendre qu'il y a des structures de données qui sont courantes.
C'est de là que naissent les **classes métier** aussi appelées "entités" dans notre cours.
Pour l'instant, ces classes métiers seront très légères, puisque leur premier but est avant tout de préparer la sauvegarde de leurs données en base. On dit alors que notre but est la *persistance des données*. C'est-à-dire qu'on veut les rendre disponibles même une fois que la page a été exécutée.

Plus tard dans le cours, nous étofferons ces classes qui mériteront alors leur nom de "classe métier".

Nous avons décidé que nous allions créer un *blog*. De manière basique, un blog se représente ainsi :
le contenu principal est appelé **article**.
Un article est composé de plusieurs choses :

- un titre ;
- un texte ;
- une thumbnail (miniature ou imagette en français).

Les articles pourront être **commentés** par différentes personnes.
Un commentaire est composé de plusieurs choses :

- un titre ;
- un texte ;
- un renvoi vers le site web de l'auteur du commentaire.

Et ainsi de suite.
Comme vous pouvez le constater, ces descriptions amènent à une conclusion simple : les "entités" peuvent être représentées par de simples classes telles qu'on a l'habitude de manipuler.
Il y a cependant une nécessité qui est **imposée** par le fait qu'on va devoir *persister* les objets : il faut qu'ils soient identifiables de manière unique.
A priori, deux articles peuvent avoir le même titre, ce dernier ne peut donc pas être l'identifiant de l'article.
Le texte peut être très long, comparer deux textes pour trouver l'identifiant du site ne serait donc pas performant.
C'est pour cela que nous allons devoir créer un attribut spécial. Par *convention*, cet attribut s'appellera `ID` et sera un entier.
Une version simplifiée (et donc incomplète) de notre classe `Article` serait donc :

```
1 public class Article{ //important : la classe doit être publique
2
3     public int ID {get; set;}
4     public string Titre{get; set;}
5     public string Texte{get; set;}
6     public string ThumbnailPath{get; set;}
7
8 }
```

Listing 13 – Structure simplifiée de la classe `Article`

Vous pouvez le constater, notre classe est surtout composée de propriétés simples (`int`, `string`, `float`, `decimal`) publiques avec un accès en **lecture et en écriture**.
Vous pouvez aussi utiliser certains objets *communs* tels que :

- `DateTime`
- `TimeSpan`
- `Byte[]`

Vous trouverez un récapitulatif complet des types supportés par défaut sur msdn .

Cela signifie que les `struct` ne sont pas compatibles par défaut. N'utilisez que les objets simples ou les objets que vous avez vous-même créés.

9.2. Customiser des entités

9.2.1. Le comportement par défaut

Avant de *personnaliser* les entités, tentons de comprendre leur comportement par défaut. Premièrement, si votre entité est bien enregistrée, elle sera transformée en **table** SQL. Cette **table** portera le même nom que l'entité. Les colonnes prendront le nom de l'attribut qu'elles cartographient.

ID(int)	Titre(Varchar128)	Texte(Text)	ThumbnailPath
1	Premier article	Bonjour à tous, bienvenue sur mon blog !	/images/coeur.png
2	Billet d'humeur	Je suis trop heureux d'avoir créé mon blog !	/images/content.png
...	...	.....	.....

TABLE 9.2. – La table SQL générée à partir de notre entité

<-

9.2.2. Changer le nom des tables et colonnes

Dans certaines applications, les gens aiment bien faire en sorte que les tables aient un préfixe qui symbolisent leur application. Cela permet, par exemple, de mettre plusieurs applications dans une seule et même base de données, sans changer l'annuaire des membres.
Pour cela, il faudra expliquer à EntityFramework que vous désirez changer le nom de la table voire des colonnes grâce aux **attributs** `[Table("nom de la table")]` et `[Column("Nom de la colonne")]`.

```
[Table("blog_Article")]
public class Article{
    public int ID {get; set;}
    [Column("Title")]
    [StringLength(128)]
    public string Titre{get; set;}
    [Column("Content")]
    public string Texte{get; set;}
    public string ThumbnailPath{get; set;}
}
```

Listing 14 – Personnalisation de la classe

9.2.3. Les données générées par la base de données

La base de données peut calculer certaines valeurs pour vos entités.
Inconsciemment nous avons déjà mis en place cette fonctionnalité précédemment. Rappelez-vous, je vous ai dit qu'une entité avait besoin d'un attribut `ID`.
Cet attribut, par défaut est compris par le programme comme possédant les particularités suivantes :

- nom de colonne = `ID` ;
- entiers positifs ;
- valeurs générées par la base de données : de manière séquentielle et unique à chaque instance.

Si nous devions coder cela nous mettrions :

```
public class Article{
    [Column("ID")]
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.IDENTITY)]
    public int ID_That_Is_Not_Conventional{get; set;}
    [Column("Title")]
    [StringLength(128)]
    public string Titre{get; set;}
    [Column("Content")]
    public string Texte{get; set;}
    public string ThumbnailPath{get; set;}
}
```

Listing 15 – Génération de l'ID par la base de données

9.3. Préparer le contexte de données

Nous allons vouloir *enregistrer* nos données dans la base de données. En génie logiciel on dit souvent que l'on assure la **persistance** des données. Comme dans les jeux en ligne *à monde persistant* en somme.
Lorsque vous utilisez EntityFramework, le lien fait entre la base de données et votre application se situe au niveau d'un objet appelé `DbContext`.
Si vous visitez un peu les classes que Visual Studio vous a générées, vous ne trouvez aucune classe appelée `DbContext`. Par contre vous trouvez une classe nommée `ApplicationDbContext` qui hérite de `IdentityDbContext`.

```
1  public class ApplicationDbContext :
       IdentityDbContext<ApplicationUser>
2      {
3          public ApplicationDbContext()
4              : base("DefaultConnection", throwIfV1Schema: false)
5          {
6          }
7
8          public static ApplicationDbContext Create()
9          {
10             return new ApplicationDbContext();
11         }
12     }
```

Listing 16 – ApplicationDbContext

En fait, comme nous avons créé une application web avec authentification, VisualStudio a préparé un contexte de données qui pré-enregistre **déjà** des classes pour :

- les utilisateurs ;
- leur informations personnelles (customisable) ;
- leur rôle dans le site (Auteur, Visiteur, Administrateur...) ;
- les liens éventuels avec les sites comme Facebook, Twitter, Google, Live...

`IdentityDbContext` est donc un `DbContext`, c'est là que la suite se passera.

9.3.1. Prendre le contrôle de la base de données

Avant toute chose, comme nous sommes en plein développement nous allons faire plein de tests. Ce qui signifie que nous allons vouloir souvent changer la structure du modèle.
Nous verrons plus tard qu'il existe un outil très puissant pour permettre de mettre à jour une base de données, qu'on appelle les *migrations*. Pour simplifier les choses, nous ne les mettrons en place qu'un peu plus tard.
Pour l'instant, il faudra que nous supprimions puis recréions la base de données à chaque changement.
Pour cela, regardez le dossier `App_Data`. Faites un clic droit, puis Ajouter > Nouvel élément.

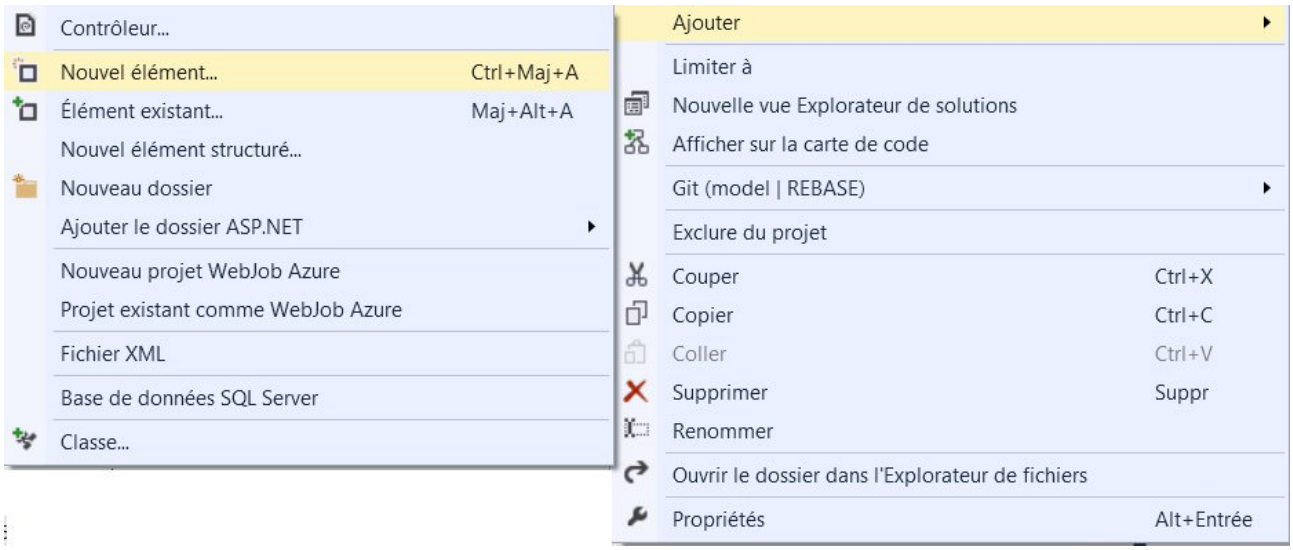


FIGURE 9.1. – Nouvel Élément

Sélectionnez ”Base de données SQL Server” (selon votre version de Visual Studio, il précisera ou pas (LocalDb)). Donnez-lui le nom `sqldb`.

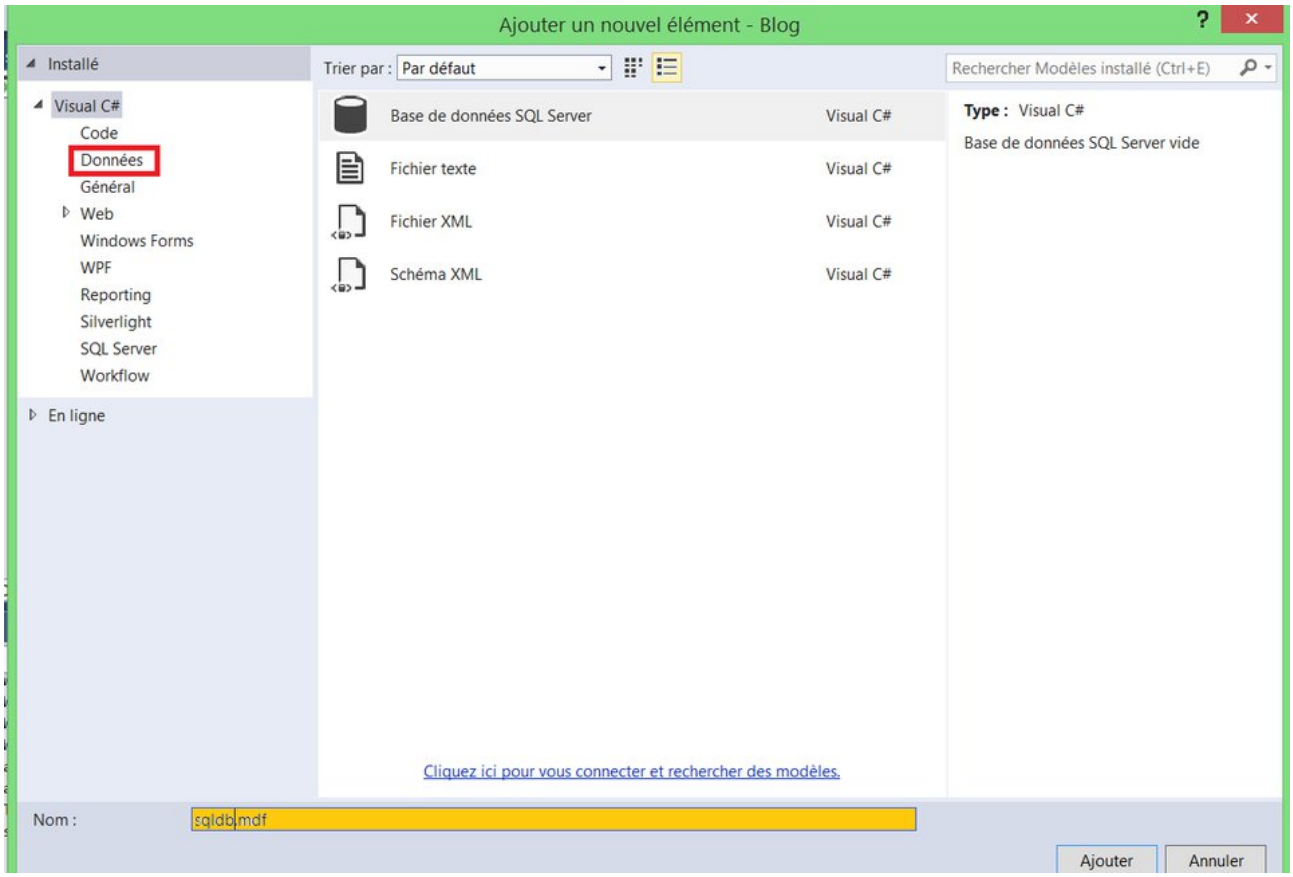


FIGURE 9.2. – Créez la base de données

Maintenant, allons à la racine du site et ouvrons le fichier Web.config. Trouvons la `connectionString` nommée `DefaultConnection` et changeons l'adresse de la base de données pour faire correspondre à `sqldb.mdf` :

```
<connectionStrings>
    <add name="DefaultConnection" connectionString="Data
        Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\sqldb.mdf;Init
        Catalog=aspnet-Blog-20141014093301;Integrated
        Security=True" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

Listing 17 – Web.config ConnectionString

9.3.2. Persister une entité

Pour persister une entité, il n'y a qu'une seule ligne à ajouter à notre DbContext : `public DbSet<TEntity> Nom { get; set; }`.
Ce qui donne, dans le cas des articles :

```
public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext()
            : base("DefaultConnection", throwIfV1Schema: false)
        {
        }
        public DbSet<Article> Articles { get; set; }
        public static ApplicationDbContext Create()
        {
            return new ApplicationDbContext();
        }
    }
```

Listing 18 – Les articles seront persistés

Et cela sera comme ça pour **toutes** nos entités.

i Petite astuce appréciable : lorsque vous créez une nouvelle entité, ajoutez-la tout de suite au contexte. Une fois cela fait, vous aurez accès au *scafolding* complet pour les contrôleurs.

10. Manipuler ses entités avec EF

C'est maintenant que tout commence. Nous allons apprendre à manipuler nos entités et à les **persister**, c'est-à-dire les enregistrer dans la base de données.
Comme la base de données est un système puissant, il nous permet de lui laisser faire un certains nombres de traitements que nous devrions théoriquement faire dans le contrôleur.
Cette partie vous présentera donc comment obtenir les données sauvegardées dans la base de données, les filtrer, comment les sauvegarder.
Nous terminerons en présentant le patron de conception "Repository" aussi appelé "Bridge", afin de rendre notre code plus propre et maintenable.

10.1. Obtenir, filtrer, ordonner vos entités

Allons donc dans notre contrôleur `ArticleController.cs`.
La première étape sera d'ajouter une instance de notre contexte de données en tant qu'attribut du contrôleur.

```
1 public class ArticleController{
2
3     private ApplicationDbContext bdd =
          ApplicationDbContext.Create();//le lien vers la base de
          données
4     /* reste du code*/
5 }
```

Listing 19 – Ajouter un lien vers la base de données

Une fois ce lien fait, nous allons nous rendre dans la méthode `List` et nous allons remplacer le vieux code par une requête à notre base de données.

```
1 try
2 {
3    List<Article> liste = _repository.GetAllListArticle().ToList();
4    return View(liste);
5 }
6 catch
7 {
8     return View(new List<Article>());
9 }
```

Listing 20 – L'ancien code

Entity Framework use Linq To SQL pour fonctionner. Si vous avez déjà l'habitude de cette technologie, vous ne devriez pas être perdu.

Sinon, voici le topo :
Linq, pour *Language Integrated Query*, est un ensemble de bibliothèques d'extensions codées pour vous simplifier la vie lorsque vous devez manipuler des collections de données.
Ces collections de données peuvent être dans des tableaux, des listes, du XML, du JSON, ou bien une base de données !
Quoi qu'il arrive, les méthodes utilisées seront toujours les mêmes.

10.1.1. Obtenir la liste complète

Il n'y a pas de code plus simple : `bdd.Articles.ToList();`. Et encore, le `ToList()` n'est nécessaire que si vous désirez que votre vue manipule une `List<Article>` au lieu d'un `IEnumerable<Article>`. La différence entre les deux : la liste est totalement chargée en mémoire alors que le `IEnumerable` ne fera en sorte de charger qu'un objet à la fois.
Certains cas sont plus pratiques avec une liste, donc pour rester le plus simple possible, gardons cette idée !

10.1.2. Obtenir une liste partielle (retour sur la pagination)

Souvenez-vous de l'ancien code :

```
1  public ActionResult List(int page = 0)
2          {
3              try
4              {
5                  List<Article> liste =
                       _repository.GetListArticle(page *
                       ARTICLEPERPAGE, ARTICLEPERPAGE).ToList();
6                  ViewBag.Page = page;
7                  return View(liste);
8              }
9              catch
10             {
11                 return View(new List<Article>());
12             }
13         }
```

Listing 21 – La pagination dans l'ancienne version

Notre idée restera la même avec Linq. Par contre Linq ne donne pas une méthode qui fait tout à la fois.
Vous n'aurez droit qu'à deux méthodes :

- `Take` : permet de définir le nombre d'objet à prendre ;
- `Skip` : permet de définir le saut à faire.

Néanmoins, avant toute chose, il faudra que vous indiquiez à EntityFramework comment ordonner les entités.
Pour s'assurer que notre liste est bien ordonnée par la date de publication (au cas où nous aurions des articles dont la publication ne s'est pas faite juste après l'écriture), il faudra utiliser la méthode `OrderBy`.
Cette dernière attend en argument une fonction qui retourne le paramètre utilisé pour ordonner.

Par exemple, pour nos articles, rangeons-les par ordre alphabétique de titre :

```
bdd.Articles.OrderBy(a => a.Titre).ToList();
```

Listing 22 – La liste est ordonnée

Ainsi, avec la pagination, nous obtiendrons :

```
public ActionResult List(int page = 0)
        {
            try
            {
                //on saute un certain nombre d'article et on en prend la quantité voulue
                List<Article> liste = bdd.Articles
                                          .OrderBy(a => a.ID)
                                          .Skip(page * ARTICLEPERPAGE)
                                          .Take(ARTICLEPERPAGE).ToList();
                ViewBag.Page = page;
                return View(liste);
            }
            catch
            {
                return View(new List<Article>());
            }
        }
```

Listing 23 – Nouvelle version de la pagination

Par exemple si nous avions 50 articles en base de données, avec 5 articles par page, nous afficherions à la page 4 les articles 21, 22, 23, 24 et 25.
Mais le plus beau dans tout ça, c'est qu'on peut chaîner autant qu'on veut les Skip et les Take. Ainsi en faisant : `Skip(3).Take(5).Skip(1).Take(2)` nous aurions eu le résultat suivant :

N° article	Pris
1	non
2	non
3	non
4	oui
5	oui
6	oui
7	oui

8	oui
9	non
10	oui
11	oui

Si vous voulez que votre collection respecte un ordre spécial, il faudra lui préciser l'ordre **avant** de faire les `Skip` et `Take`, sinon le système ne sait pas comment faire.

10.1.3. Filtrer une liste

10.1.3.1. La méthode Where

Avant de filtrer notre liste, nous allons légèrement modifier notre entité `Article` et lui ajouter un paramètre booléen nommé `EstPublie`.

```
1  public class Article
2      {
3          public Article()
4          {
5              EstPublie = true;//donne une valeur par défaut
6          }
7          public int ID { get; set; }
8          [Column("Title")]
9          [StringLength(128)]
10         public string Titre { get; set; }
11         [Column("Content")]
12         public string Texte { get; set; }
13         public string ThumbnailPath { get; set; }
14         public bool EstPublie { get; set; }
15     }
```

Listing 24 – Ajout d'un attribut dans la classe Article

Maintenant, notre but sera de n'afficher que les articles qui sont marqués comme publiés. Pour cela il faut utiliser la fonction `Where` qui prend en entrée un délégué qui a cette signature :

```
1  delegate bool Where(TEntity entity)
```

Listing 25 – Signature de la méthode Where

Dans le cadre de notre liste, nous pourrions donc utiliser :

```
public ActionResult List(int page = 0)
        {
            try
            {
                //on saute un certain nombre d'articles et on en
                    prend la quantité voulue
                List<Article> liste = bdd.Articles.Where(article =>
                    article.EstPublie)
                                          .Skip(page *
                                              ARTICLEPERPAGE).Take(ARTICLEPERPAG
                ViewBag.Page = page;
                return View(liste);
            }
            catch
            {
                return View(new List<Article>());
            }
        }
```

Listing 26 – Filtrage de la liste en fonction de l'état de l'article

Dans vos filtres, toutes les fonctions ne sont pas utilisables dès qu'on traite les chaînes de caractères, les dates...

10.1.3.2. Filtre à plusieurs conditions

Pour mettre plusieurs conditions, vous pouvez utiliser les mêmes règles que pour les embranchements `if` et `else if`, c'est-à-dire utiliser par exemple `article.EstPublie && article.Titre.Length < 100` ou bien `article.EstPublie || !string.IsNullOrEmpty(article.ThumbnailPath)`.
Néanmoins, dans une optique de lisibilité, vous pouvez remplacer le `&&` par une chaîne de `Where`.

```
List<Article> liste = bdd.Articles.Where(article =>
    article.EstPublie)
                                  .Where(article =>
                                      !string.IsNullOrEmpty(article.ThumbnailPa
```

Listing 27 – Chaînage de Where

10.1.4. N'afficher qu'une seule entité

Afficher une liste d'articles, c'est bien, mais rapidement la liste deviendra longue. Surtout, nos articles étant des textes qui peuvent être assez denses, il est souhaitable que nous ayons une page où un article sera affiché seul.

Cette page vous facilitera aussi le partage de l'article : la liste est mouvante, mais la page où seul l'article est affiché restera toujours là, à la même adresse.
![[a]] | Il est vraiment important que la page reste à la même adresse et ce même si vous changez le texte ou le titre de l'article.
Pour le cours, cette page sera créée plus tard, lorsque nous verrons les liens entre les entités car nous voulons bien sûr afficher les commentaires de l'article sur cette page. Néanmoins, cela sera un bon exercice pour vous de la coder maintenant.

10.1.4.1. Indices

— Pour sélectionner un article, il faut pouvoir l'**identifier** de manière *unique*.
— La fonction `db.VotreDBSet.FirstOrDefault(m=>m.Parametre == Valeur)` vous permet d'avoir le premier objet qui vérifie la condition ou bien null si aucun article ne convient.
— Par convention, en ASP.NET MVC on aime bien appeler cette page Details.

10.2. Ajouter une entité à la bdd

10.2.1. Quelques précisions

Avant de montrer le code qui vous permettra de sauvegarder vos entités dans la base de données, je voudrais vous montrer plus en détail comment Entity Framework gère les données.

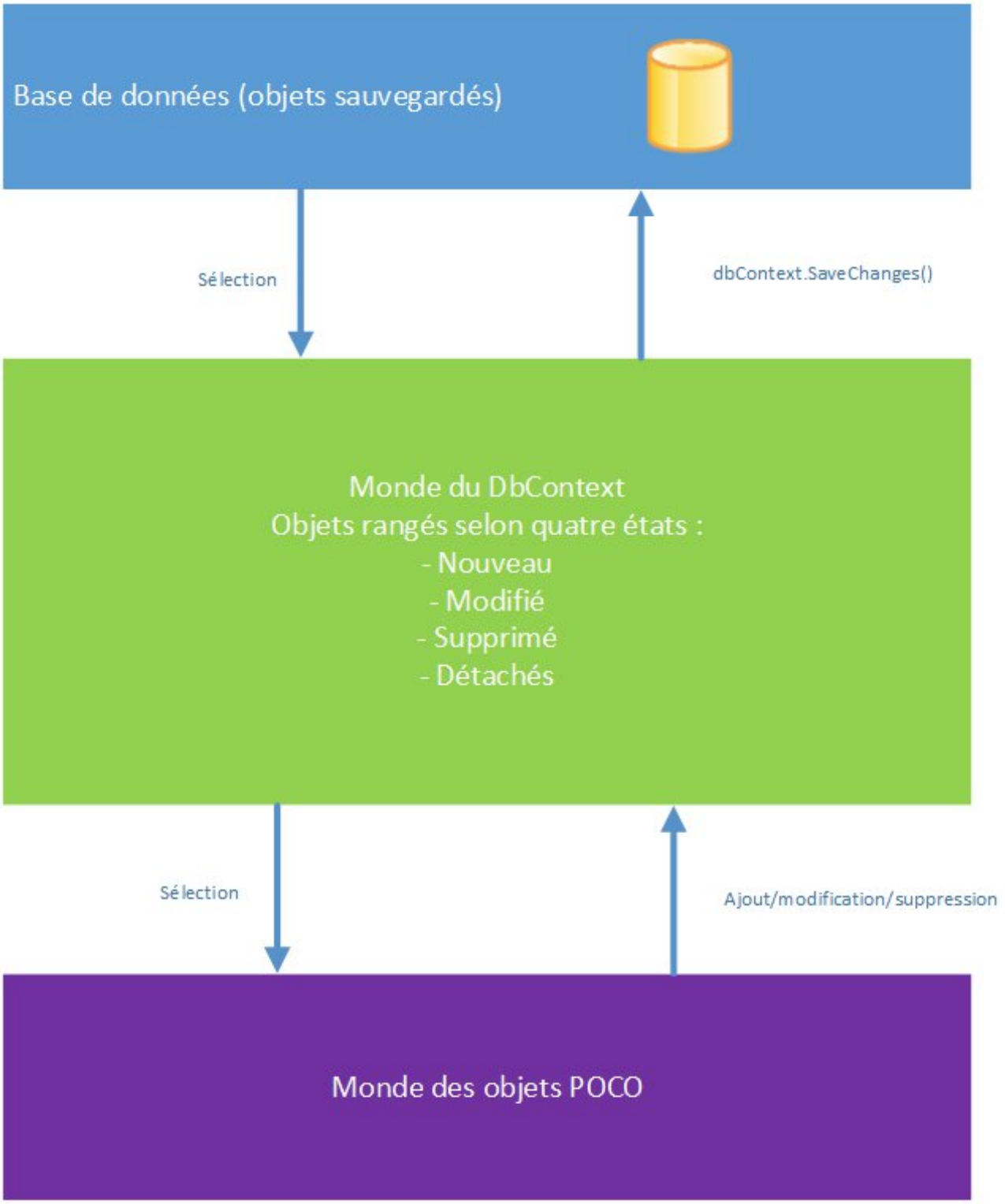


FIGURE 10.1. – Fonctionnement de EntityFramework

Comme le montre le schéma, une entité n'est pas *tout de suite* envoyée à la base de données. En fait pour que cela soit fait il faut que vous demandiez à la base de données de **persister** tous les changements en attente.

> ”détachés” et ”supprimés” sont des états qui se ressemblent beaucoup. En fait ”détaché” cela signifie que la suppression de l'entité a été persistée.

À partir de là, le système va chercher parmi vos entités celles qui sont marquées :
- créées ;
- modifiées ;
- supprimées.

Et il va mettre en place les actions adéquates.
La grande question sera donc :

> Comment marquer les entités comme ”créées” ?

10.2.2. Ajouter une entité

C'est l'action la plus simple de toutes : `bdd.VotreDbSet.Add(votreEntite);` et c'est tout !
Ce qui transforme notre code d'envoi d'article en :

Listing 28 – Persistance de l'article en base de données

10.3. Modifier et supprimer une entité dans la base de données

10.3.1. Préparation

1.
2.
3.

```
1  [HttpGet]
2          public ActionResult Edit(int id)
3          {
4              return View();
5          }
6          [HttpPost]
7          [ValidateAntiForgeryToken]
8          public ActionResult Edit(int id, ArticleCreation
               articleCreation)
9          {
10             return RedirectToAction("List");
11         }
```

Listing 29 – Les méthodes d'édition

Toutes les étapes sont importantes. Si vous ne faites pas la première, EF va possiblement croire que vous êtes en train de créer une **nouvelle** entité.

```
List
```

```
@Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ })
    |
                    @Html.ActionLink("Details", "Details", new { /*
                        id=item.PrimaryKey */ }) |
                    @Html.ActionLink("Delete", "Delete", new { /*
                        id=item.PrimaryKey */ })
```

Listing 30 – Les liens incomplets

```
@Html.ActionLink("Edit", "Edit", new {  id=item.ID  }) |
                    <!-- si vous n'avez pas créé la page Details
                        dans l'exercice précédent, laissez
                        inchangé-->
                    @Html.ActionLink("Details", "Details", new {
                        id=item.ID  }) |
                    @Html.ActionLink("Delete", "Delete", new {
                        id=item.ID })
```

Listing 31 – Les liens sont corrects.

10.3.2. Modifier l'entité

```
Where
```

```
FirstOrDefault
Find
```

```
Entry<TEntity>()
```

```
DbEntityEntry
```

Listing 32 – La méthode Edit

J'ai encapsulé la gestion de l'image dans une méthode pour que ça soit plus facile à utiliser.

Contenu masqué n°9

10.3.3. La suppression de l'entité

`Entry` `Deleted`

`bdd.Articles.Remove`

`POST`

Contenu masqué n°10

10.4. Le patron de conception Repository

`Repository` `ApplicationDbContext`

`bdd = ApplicationDbContext.Create();`

`ApplicationDbContext`

Tu nous apprends Entity Framework, mais tu veux le remplacer par autre chose, je ne comprends pas pourquoi !

ORM

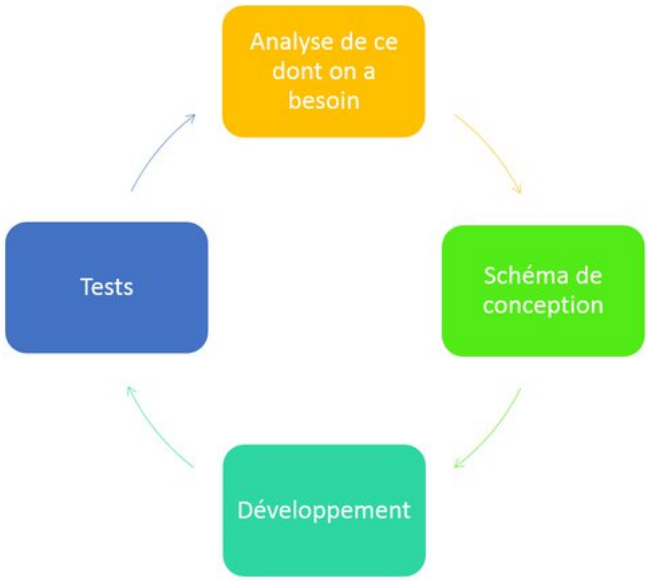


FIGURE 10.2. – Méthode de développement d'un projet

22

```
1 public interface EntityRepository<T> where T: class{
2     IEnumerable<T> GetList(int skip = 0, int limit = 5);
3     T Find(int id);
4     void Save(T entity);
```

22. Vous trouverez des renseignements complémentaires ici et ici .

```
5       void Delete(T entity);
6       void Delete(int id);
7   }
```

Listing 33 – L'interface commune à tous les Repository

```
1   public class EFArticleRepository: EntityRepository<Article>{
2       private ApplicationDbContext bdd = new ApplicationDbContext();
3       public IEnumerable<Article> GetList(int skip = 0, int limit =
            5){
4           return bdd.Articles.OrderBy(a =>
                a.ID).Skip(skip).Take(limit);
5       }
6       public Article Find(int id){
7           return bdd.Articles.Find(id);
8       }
9       public void Save(Article entity){
10          Article existing = bdd.Articles.Find(entity.id);
11          if(existing == null){
12             bdd.Articles.Add(entity);
13          }else{
14             bdd.Entry<Article>(existing).State =
                   EntityState.Modified;
15
                   bdd.Entry<Article>(existing).CurrentValues.SetValues(entity);
16          }
17          bdd.SaveChanges();
18       }
19      public void Delete(Article entity){
20         bdd.Articles.Remove(entity);
21      }
22      public void Delete(int id){
23          Delete(bdd.Articles.Find(id));
24      }
25  }
```

Listing 34 – Le Repository complet qui utilise Entity Framework

Contenu masqué

Contenu masqué n°9

```
1  private bool handleImage(ArticleCreation articleCreation, out
       string fileName)
2          {
3              bool hasError = false;
4              fileName = "";
5              if (articleCreation.Image != null)
6              {
7
8                  if (articleCreation.Image.ContentLength > 1024 *
                       1024)
9                  {
10                     ModelState.AddModelError("Image",
                           "Le fichier téléchargé est trop grand.");
11                     hasError = true;
12                 }
13
14                 if
                       (!AcceptedTypes.Contains(articleCreation.Image.ContentType)
15                        ||
                              AcceptedExt.Contains(Path.GetExtension(articleCreat
16                 {
17                     ModelState.AddModelError("Image",
                           "Le fichier doit être une image.");
18                     hasError = true;
19                 }
20
21                 try
22                 {
23                     string fileNameFile =
                           Path.GetFileName(articleCreation.Image.FileName);
24                     fileName = new
                           SlugHelper().GenerateSlug(fileNameFile);
25                     string imagePath =
                           Path.Combine(Server.MapPath("~/Content/Upload"),
                           fileName);
26                     articleCreation.Image.SaveAs(imagePath);
27                 }
28                 catch
29                 {
30                     fileName = "";
31                     ModelState.AddModelError("Image",
                           "Erreur à l'enregistrement.");
32                     hasError = true;
33                 }
34
```

```
35              }
36              return !hasError;
37          }
```

Listing 35 – gestion de l'image

Retourner au texte.

Contenu masqué n°10

```
1  [HttpPost]
2          [ValidateAntiForgeryToken]
3          public ActionResult Delete(int id)
4          {
5              Article a = bdd.Articles.Find(id);
6              if (a == null)
7              {
8                  return RedirectToAction("List");
9              }
10             bdd.Articles.Remove(a);
11             bdd.SaveChanges();
12             return RedirectToAction("List");
13         }
```

Listing 36 – le contrôleur de suppression

```
1  @using (Html.BeginForm("Delete", "Article", new { id = item.ID },
       FormMethod.Post, null))
2                      {
3                          Html.AntiForgeryToken();
4                          <button type="submit">Supprimer</button>
5                      }
```

Listing 37 – le code à changer dans la vue

Retourner au texte.

11. Les relations entre entités

11.1. Préparation aux cas d'étude

Article/Details

Contenu masqué n°11

11.2. Cas d'étude numéro 1 : les commentaires

11.2.1. Relation 1->n

```
1  public class Commentaire{
2
3      public int ID {get; set;}
4      [DataType(DataType.Email)]
5      public string Email {get; set;}
6      [StringLength(75)]
7      public string Pseudo {get; set;}
8      [DataType(DataType.MultilineText)]
9      public string Contenu {get; set;}
10     public DateTime Publication{get; set;}
11 }
```

Listing 38 – Un commentaire basique

FIGURE 11.1. – Relation 1-n non inversée

```
1  public class Commentaire{
2
3      public int ID {get; set;}
4      [DataType(DataType.EmailAddress)]
5      public string Email {get; set;}
6      [StringLength(75)]
7      public string Pseudo {get; set;}
8      [DataType(DataType.MultilineText)]
9      public string Contenu {get; set;}
10     public DateTime Publication{get; set;}
11     //On crée un lien avec l'article
12     public Article Parent {get; set;}
13 }
```

Listing 39 – Relation 1-n

i N'oubliez pas de créer un DbSet nommé Commentaires dans votre contexte

Include

Parent

```
1  //obtient le premier commentaire (ou null s'il n'existe pas) du
       premier article du blog
```

```
Commentaire com = bdd.Commentaires.Include('Parent').Where(c =>
    c.Id = 1).FirstOrDefault();
Article articleLie = com.Article;//
```

Listing 40 – Trouver l'article lié

? J'ai un problème : ce que je veux, moi, c'est afficher un article grâce à son identifiant puis, en dessous afficher les commentaires liés.

11.2.2. Inverser la relation 1-n

11.2.2.1. Le principe

FIGURE 11.2. – Relation 1-n inversée

List<Commentaire>

```

```

Listing 41 – Inversion de la relation.

11.2.2.2. Lazy Loading

ORM

ORM

Include

Le lazy loading est une fonctionnalité très appréciable des ORM, néanmoins, n'en abusez pas, si vous l'utilisez pour des gros volumes de données, votre page peut devenir très longue à charger.

`virtual`

Virtual ? comme pour surcharger les méthodes héritées ?

```
1  public class Article
2      {
3          public int ID { get; set; }
4          /// <summary>
5          /// Le pseudo de l'auteur
6          /// </summary>
7          [Required(AllowEmptyStrings = false)]
8          [StringLength(128)]
9          [RegularExpression(@"^[^,\.\^]+$")]
10         [DataType(DataType.Text)]
11         public string Pseudo { get; set; }
12
13         /// <summary>
14         /// Le titre de l'article
15         /// </summary>
16         [Required(AllowEmptyStrings = false)]
17         [StringLength(128)]
18         [DataType(DataType.Text)]
19         public string Titre { get; set; }
20
21         /// <summary>
22         /// Le contenu de l'article
23         /// </summary>
24         [Required(AllowEmptyStrings = false)]
25         [DataType(DataType.MultilineText)]
26         public string Contenu { get; set; }
27
28         /// <summary>
29         /// Le chemin de l'image
30         /// </summary>
31         public string ImageName { get; set; }
32
```

```
33          public virtual List<Commentaire> Comments { get; set; }
34      }
```

Listing 42 – La classe Article complète

11.2.3. Interface graphique : les vues partielles

11.2.3.1. Étape 1 : la liste des commentaires

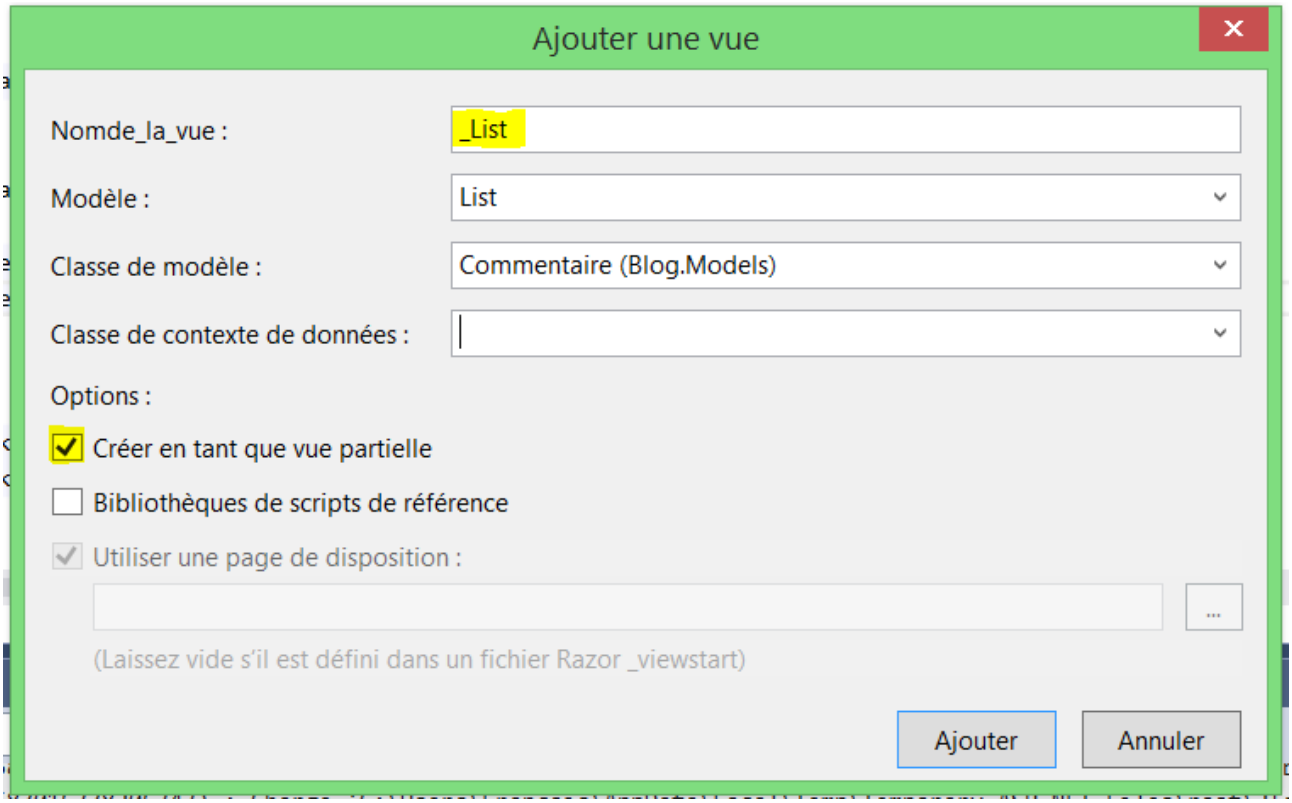


FIGURE 11.3. – Ajouter une vue partielle

i Vous remarquerez l'underscore utilisé dans le nom de la vue. C'est une convention pour les vues partielles.

11.3. Cas d'étude numéro 2 : les tags

11.3.1. Définition et mise en place

FIGURE 11.4. – Liaison many to many

```
1  public class Tag
2      {
3          public int ID { get; set; }
4
5          public string Name { get; set; }
6
7          [StringLength(255)]
8          [Display(AutoGenerateField = true)]
9          [Index(IsUnique=true)]//permet de rendre unique le slug
       comme ça le tag "Mon Tag" et "mon tag" et "MOn Tag" seront le
       même :-)
10         public string Slug { get; set; }
11         public virtual List<Article> LinkedArticles { get; set; }
12     }
13
```

```
14      public class Article
15      {
16          public int ID { get; set; }
17          /// <summary>
18          /// Le pseudo de l'auteur
19          /// </summary>
20          [Required(AllowEmptyStrings = false)]
21          [StringLength(128)]
22          [RegularExpression(@"^[^,\.\^]+$")]
23          [DataType(DataType.Text)]
24          public string Pseudo { get; set; }
25
26          /// <summary>
27          /// Le titre de l'article
28          /// </summary>
29          [Required(AllowEmptyStrings = false)]
30          [StringLength(128)]
31          [DataType(DataType.Text)]
32          public string Titre { get; set; }
33
34          /// <summary>
35          /// Le contenu de l'article
36          /// </summary>
37          [Required(AllowEmptyStrings = false)]
38          [DataType(DataType.MultilineText)]
39          public string Contenu { get; set; }
40
41          /// <summary>
42          /// Le chemin de l'image
43          /// </summary>
44          public string ImageName { get; set; }
45
46          public virtual List<Commentaire> Comments { get; set; }
47          public virtual List<Tag> Tags { get; set; }
48      }
```

Listing 43 – Les classes Tag et Article

Vous l'avez peut-être remarqué, j'ai utilisé l'attribut `[Index]`. Ce dernier permet de spécifier que notre Slug sera unique dans la bdd ET qu'en plus trouver un tag à partir de son slug sera presque aussi rapide que de le trouver par sa clef primaire.

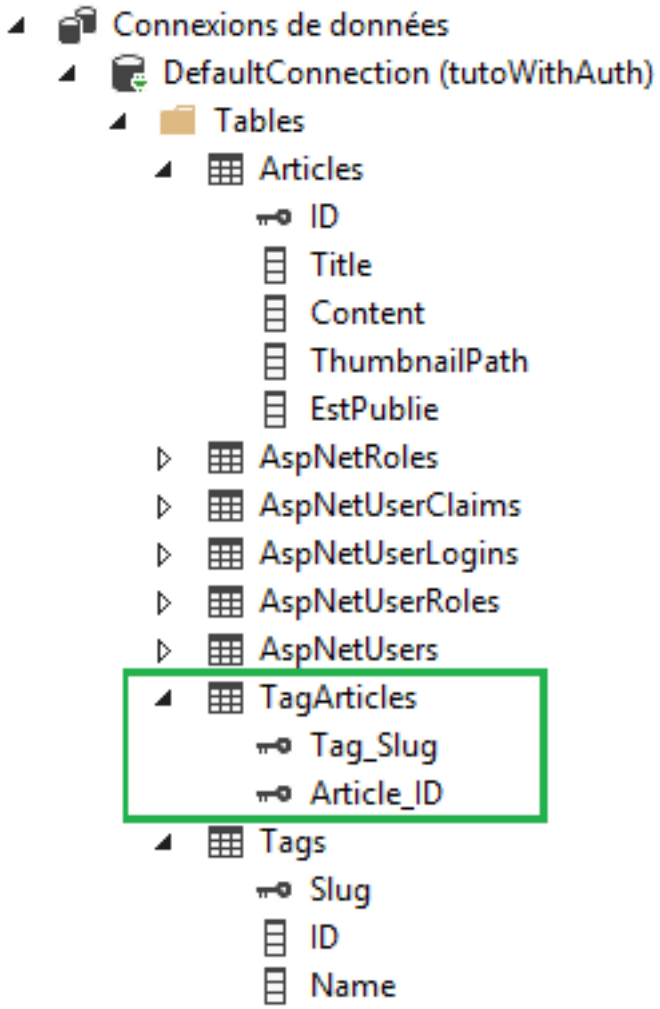


FIGURE 11.5. – L'explorateur de serveurs

Retenez bien cela : toute relation many to many crée en arrière plan une nouvelle table. Mais ne vous inquiétez pas, dans bien des cas cette table n'impacte pas les performances de votre site, en fait si vous avez bien réfléchi, cela va même les améliorer car la bdd est faite **pour ça** !

11.3.2. Un champ texte, des virgules

1.

2.
3.
4.
5.

> Contenu masqué n°12

> Contenu masqué n°13

> Vous l'aurez peut-être remarqué, nous avons plusieurs fois dupliqué du code. Je sais, c'est mal ! Ne vous inquiétez, pas nous y reviendrons. En effet, pour simplifier énormément notre code, nous allons devoir apprendre comment créer nos propres validateurs, ainsi que la notion de ModelBinder. Cela sera fait dans la prochaine partie.

11.3.3. Allons plus loin avec les relations Plusieurs à Plusieurs

```
public class MemberComment
{
    [Key, Column(Order = 0)]
    public int MemberID { get; set; }
    [Key, Column(Order = 1)]
    public int VotedArticleID { get; set; }

    //les deux attributs virtuels suivants vous permettent
        d'utiliser les propriétés comme d'habitude
    //elles seront automatiquement liées à MemberID et
        VotedArticleID
    public virtual Member Member { get; set; }
    public virtual Article VotedArticle { get; set; }

    public DateTime VoteDate { get; set; }

}
```

Listing 44 – La classe de vote

Contenu masqué

Contenu masqué n°11

```
public ActionResult Details(int id)
        {
            Article art = bdd.Articles.Find(id);
            if (art == null)
            {
                return new HttpNotFoundResult();
            }
            return View(art);
        }
```

Listing 45 – L'action Details dans le contrôleur

```
@model Blog.Models.Article
@using Blog.Models;
<section>
    <h1>@Html.DisplayFor(model => model.Titre)</h1>
    <hr />
    <header>
        <dl class="dl-horizontal">
            <dt>
```

```
                @Html.DisplayNameFor(model => model.Pseudo)
            </dt>

            <dd>
                @Html.DisplayFor(model => model.Pseudo)
            </dd>
            <dd>
                <img
                    src="@Html.DisplayFor(model => model.ImageName)"
                    alt="@Html.DisplayFor(m => m.Titre)" />
            </dd>

        </dl>
    </header>
    <article>
        @Html.DisplayFor(model => model.Contenu)
    </article>
</section>
<footer>
    @Html.ActionLink("Edit", "Edit", new { id = Model.ID }) |
    @Html.ActionLink("Retour à la liste", "List")
</footer>
```

Listing 46 – La vue Article/Details.cshtml

Retourner au texte.

Contenu masqué n°12

```
[HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Create(ArticleCreation articleCreation)
        {
            if (!ModelState.IsValid)
            {
                return View(articleCreation);
            }

            string fileName = "";
            if (!handleImage(articleCreation, out fileName))
            {
                return View(articleCreation);
            }

            Article article = new Article
            {
                Contenu = articleCreation.Contenu,
                Pseudo = articleCreation.Pseudo,
```

```
20                  Titre = articleCreation.Titre,
21                  ImageName = fileName,
22                  Tags = new List<Tag>()
23              };
24              IEnumerable<Tag> askedTags = new List<Tag>();
25
26              SlugHelper slugifier = new SlugHelper();
27              if(articleCreation.Tags.Trim().Length > 0)
28              {
29                  askedTags = from tagName in
                        articleCreation.Tags.Split(',') select new Tag
                        { Name = tagName.Trim(), Slug =
                        slugifier.GenerateSlug(tagName.Trim()) };
30              }
31              foreach (Tag tag in askedTags)
32              {
33                  Tag foundInDatabase = bdd.Tags.FirstOrDefault(t =>
                        t.Slug == tag.Slug);
34                  if (foundInDatabase != default(Tag))
35                  {
36                      article.Tags.Add(foundInDatabase);
37                  }
38                  else
39                  {
40                      article.Tags.Add(tag);
41                  }
42              }
43
44              bdd.Articles.Add(article);
45
46              bdd.SaveChanges();
47              return RedirectToAction("List", "Article");
48          }
```

Listing 47 – La méthode Create du contrôleur avec les tags.

Retourner au texte.

Contenu masqué n°13

```
1  [HttpPost]
2          [ValidateAntiForgeryToken]
3          public ActionResult Edit(int id, ArticleCreation
               articleCreation)
4          {
5              //on ne peut plus utiliser Find car pour aller chercher
                   les tags nous devons utiliser Include.
```

```
            Article entity =
                bdd.Articles.Include("Tags").FirstOrDefault(m =>
                m.ID == id);
            if (entity == null)
            {
                return RedirectToAction("List");
            }
            string fileName;
            if (!handleImage(articleCreation, out fileName))
            {

                return View(articleCreation);
            }
            DbEntityEntry<Article> entry = bdd.Entry(entity);
            entry.State = System.Data.Entity.EntityState.Modified;
            Article article = new Article
            {
                Contenu = articleCreation.Contenu,
                Pseudo = articleCreation.Pseudo,
                Titre = articleCreation.Titre,
                ImageName = fileName
            };
            entry.CurrentValues.SetValues(article);
            IEnumerable<Tag> askedTags = new List<Tag>();

            SlugHelper slugifier = new SlugHelper();
            if (articleCreation.Tags.Trim().Length > 0)
            {
                askedTags = from tagName in
                    articleCreation.Tags.Split(',') select new Tag
                    { Name = tagName.Trim(), Slug =
                    slugifier.GenerateSlug(tagName.Trim()) };
            }

            foreach (Tag tag in askedTags)
            {
                Tag foundInDatabase = bdd.Tags.FirstOrDefault(t =>
                    t.Slug == tag.Slug);
                if (foundInDatabase != default(Tag) &&
                    !article.Tags.Contains(foundInDatabase))
                {
                    article.Tags.Add(foundInDatabase);
                }
                else if(foundInDatabase == default(Tag))
                {
                    article.Tags.Add(tag);
                }
            }

            bdd.SaveChanges();
```

```
49
50              return RedirectToAction("List");
51          }
```

Listing 48 – La méthode d'édition avec la prise en charge des tags

Retourner au texte.

12. Les Migrations codefirst

12.1. Définition

12.1.1. Cas d'utilisation

ce tuto

12.1.2. La solution : les migrations

—

—

—

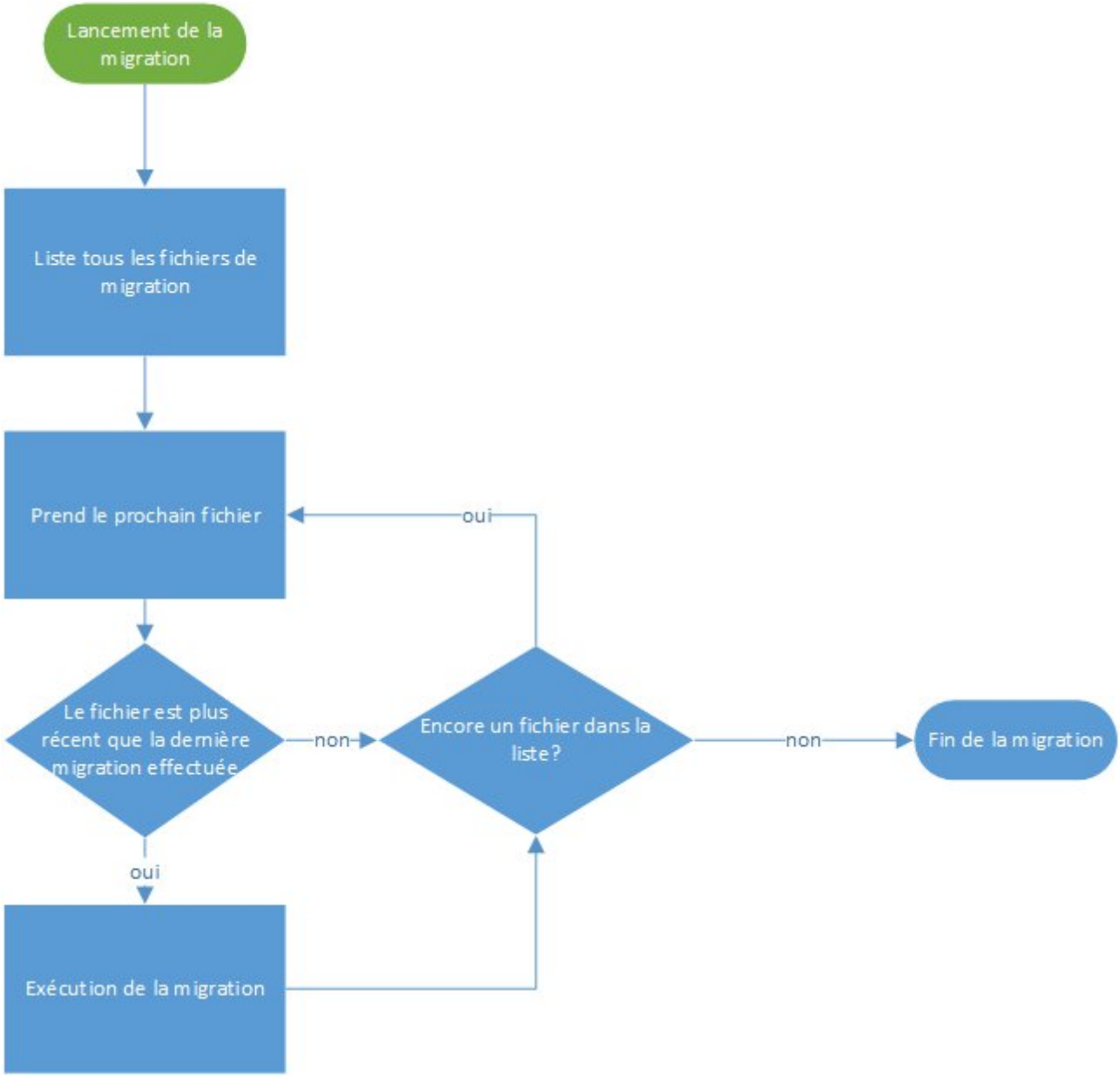


FIGURE 12.1. – Fonctionnement des migrations

reverse

12.1.3. Mettre en place les migrations

`Enable-Migrations` `Add-Migration initial`

`Add-Migration nom_lisible_par_un_humain_pour_la_migration`

`Update-Database`

Élément externe (Video) — https://www.youtube.com/embed/Co0pi4hsZPg?feature=oembed

12.2. Créer un jeu de données pour tester l'application

12.2.1. Un jeu de données simple

`Configuration.cs` `Migrations` `Seed`

12.2.1.1. Ajoutons un article

`Article`

```
context.Articles.Add(new Article
{
    Contenu = "plop !",
    Pseudo = "un pseudo",
```

```
5       Titre = "mon titre",
6   });
7
8   context.SaveChanges();
```

Listing 49 – la Seed a un article !

Update-Database

	ID	Pseudo	Titre	Contenu	ImageName
▶	1	un pseudo	mon titre	plop !	*NULL*
*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*

FIGURE 12.2. – Résultat de seed

DbSet

12.2.1.2. Créons un compte administrateur

? Créer un compte utilisateur, c'est si différent que ça de créer un article ?

AccountController.cs

Context.Users.Add()

```
1  if (!context.Roles.Any(r => r.Name == "Administrator"))
2  {
3      RoleStore<IdentityRole> roleStore = new
           RoleStore<IdentityRole>(context);
4      RoleManager<IdentityRole> manager = new
           RoleManager<IdentityRole>(roleStore);
5      IdentityRole role = new IdentityRole { Name = "Administrator"
           };
6
7      manager.Create(role);
8  }
9
10 if (!context.Users.Any(u => u.UserName == "Proprio"))
11 {
12     UserStore<ApplicationUser> userStore = new
           UserStore<ApplicationUser>(context);
13     UserManager<ApplicationUser> manager = new
           UserManager<ApplicationUser>(userStore);
14     ApplicationUser user = new ApplicationUser { UserName =
           "Proprio" };
15
16     manager.Create(user, "ProprioPwd!");
17     manager.AddToRole(user.Id, "Administrator");
18 }
```

Listing 50 – La création du compte administrateur

12.2.2. Un jeu de données complexes sans tout taper à la main

—

—

JSON

Faker.NET

Lorem Words(1).First()

```
Tag tag1 = new Tag { Name = Faker.Lorem.Words(1).First() };
Tag tag2 = new Tag { Name = Faker.Lorem.Words(1).First() };
context.Tags.AddOrUpdate(tag1, tag2);
Article article = new Article
    {
        Contenu = Faker.Lorem.Paragraphs(4),
        Pseudo = Faker.Internet.UserName(),
        Titre = Faker.Lorem.Sentence(),
        Tags = new List<Tag>()
    };
article.Tags.Add(tag1);
article.Tags.Add(tag2);
context.Articles.Add(article);
```

Listing 51 – Débutons avec Faker

12.2.3. Un jeu de données pour le debug et un jeu pour le site final

—
—

—
—

—

`Web.config`

`appSettings`

`<add key="data_version" value="debug"/>`

12.2.3.1. Pour les données de tests :

1.

```
<appSettings>
  <add xdt:Transform="Replace" xdt:Locator="Match(key)"
      key="data_version" value="debug"/>
</appSettings>
```

`Migrations/Configuration.cs` `seed`
`Debug` `Seed`
`seedDebug`

`seedDebug`

```
protected override void Seed(Blog.Models.ApplicationDbContext
    context)
{
    if (ConfigurationManager.AppSettings["data_version"] ==
        "debug")
    {
         seedDebug(context);
    }
```

```
}
```

12.2.3.2. Pour le jeu de données de production

```
<appSettings>
  <add xdt:Transform="Replace" xdt:Locator="Match(key)"
      key="data_version" value="production"/>
</appSettings>
```

`Migrations/Configuration.cs`

```
        protected override void
            Seed(Blog.Models.ApplicationDbContext context)
        {
            if (ConfigurationManager.AppSettings["data_version"] ==
                "debug")
            {
                seedDebug(context);
            }
            else if
                (ConfigurationManager.AppSettings["data_version"]
                == "production")
            {
                seedProduction(context);
            }

        }
        private void seedProduction(ApplicationDbContext context)
        {

        }
```

`seedProduction`

JSON

AddOrUpdate

12.3. Améliorer notre modèle avec la FluentAPI

Les explications qui vont suivre s'adressent surtout aux personnes qui ont une bonne connaissance des bases de données. Nous l'avons mis dans ce tutoriel car vous trouverez rapidement des cas où il faut en passer par là, mais ils ne seront pas votre quotidien, rassurez-vous.

API

API

Fluent API

12.3.1. Quand les relations ne peuvent être pleinement décrites par les annotations

API

FIGURE 12.3. – Cycle de relations

FIGURE 12.4. – Relation en losange

OnModelCreating

```
1  public class ApplicationDbContext :
       IdentityDbContext<ApplicationUser>
2  {
3          public ApplicationDbContext()
4              : base("DefaultConnection", throwIfV1Schema: false)
5          {
6
7          }
8          protected override void OnModelCreating(DbModelBuilder
               modelBuilder)
9          {
10             base.OnModelCreating(modelBuilder);
11         }
12 }
```

Listing 52 – Surcharge de la méthode OnModelCreating

DbModelBuilder

```
modelBuidler.Entity<A>()
            .HasMany(obj => obj.BProperty)
            .WithOptional(objB -> objB.AProperty)
            .WillCascadeOnDelete(false);// on dit explicitement
                qu'il n'y a pas de suppression en cascade
```

Quatrième partie

Allons plus loin avec ASP.NET

13. Sécurité et gestion des utilisateurs

13.1. Authentifier un utilisateur, le login et le mot de passe

[23]

Le saviez vous ? Le mot ”authentification” se distingue du mot ”identification” par le fait qu'une authentification assure que la personne est bien celle qu'elle prétend être. À l'opposé, une simple identification permet à la personne de déclarer qui elle est. C'est pour ça que vous avez un ”identifiant” et un ”mot de passe”. Le premier vous permet de déclarer votre identité, le second, comme il n'est connu que de vous (même pas de la plateforme) permet de vous authentifier.

Controllers\AccountController.cs
return View();

23. Et n'oubliez pas de bien définir votre mot de passe.

— méthodes asynchrones
— `Microsoft.AspNet.Identity.Owin`

```
using System;
using System.Globalization;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using AuthDemo.Models;

namespace AuthDemo.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private ApplicationSignInManager _signInManager;
        private ApplicationUserManager _userManager;

        public AccountController()
        {
        }

        public AccountController(ApplicationUserManager
            userManager, ApplicationSignInManager signInManager )
        {
            UserManager = userManager;
            SignInManager = signInManager;
        }

        public ApplicationSignInManager SignInManager
        {
            get
            {
                return _signInManager ??
                    HttpContext.GetOwinContext().Get<ApplicationSignInManager>
            }
            private set
            {
                _signInManager = value;
            }
        }

        public ApplicationUserManager UserManager
        {
```

```
45              get
46              {
47                  return _userManager ??
                        HttpContext.GetOwinContext().GetUserManager<ApplicationUser
48              }
49              private set
50              {
51                  _userManager = value;
52              }
53          }
54          //
55          // POST: /Account/Login
56          [HttpPost]
57          [AllowAnonymous]
58          [ValidateAntiForgeryToken]
59          public async Task<ActionResult> Login(LoginViewModel model,
                string returnUrl)
60          {
61              if (!ModelState.IsValid)
62              {
63                  return View(model);
64              }
65
66              // Ceci ne comptabilise pas les échecs de connexion
                    pour le verrouillage du compte
67              // Pour que les échecs de mot de passe déclenchent le
                    verrouillage du compte, utilisez shouldLockout:
                    true
68              var result = await
                    SignInManager.PasswordSignInAsync(model.Email,
                    model.Password, model.RememberMe, shouldLockout:
                    false);
69              switch (result)
70              {
71                  case SignInStatus.Success:
72                      return RedirectToLocal(returnUrl);
73                  case SignInStatus.LockedOut:
74                      return View("Lockout");
75                  case SignInStatus.RequiresVerification:
76                      return RedirectToAction("SendCode", new {
                            ReturnUrl = returnUrl, RememberMe =
                            model.RememberMe });
77                  case SignInStatus.Failure:
78                  default:
79                      ModelState.AddModelError("",
                            "Tentative de connexion non valide.");
80                      return View(model);
81              }
82          }
83  /// ...
```

```
84      }
85  }
```

Listing 53 – Le code de login est asynchrone et utilise le composant `Microsoft.AspNet.Identity.Owin` comme `_loginManager`

—
—
— SSO

Adapter Bridge

public AccountController(ApplicationUserManager userManager, ApplicationSignInManager signInManager)

ApplicationSignInManager

13.1.1. Le modèle de données

— IdentityModels.cs
— AccountViewModels.cs
— ManageViewModel.cs

IdentityModels

IdentityModels.cs
— ApplicationUser IdentityUser
—

DbContext IdentityDbContext

```
public bool RequireUniqueEmail { get; set; }
public virtual IDbSet<TRole> Roles { get; set; }
public virtual IDbSet<TUser> Users { get; set; }
protected override void OnModelCreating(DbModelBuilder
    modelBuilder);
protected override DbEntityValidationResult
    ValidateEntity(DbEntityEntry entityEntry, IDictionary<object,
    object> items);
```

Listing 54 – les ajouts de `IdentityDbContext`

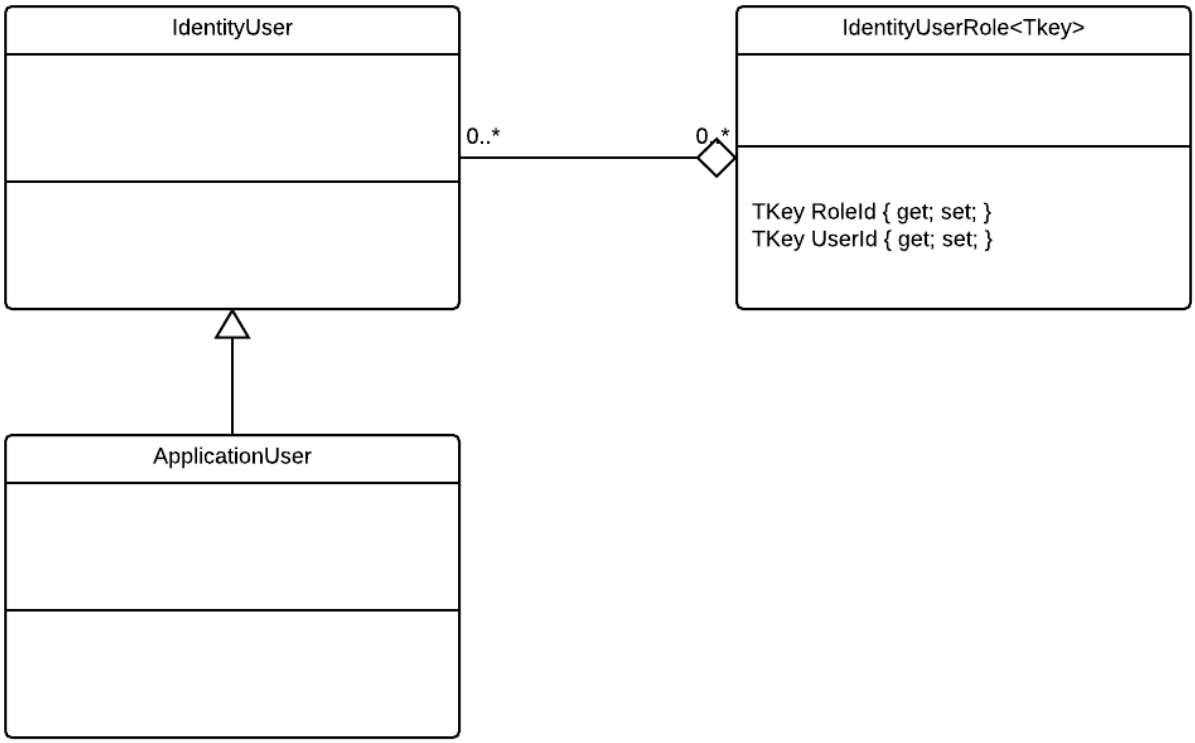


FIGURE 13.1. – Diagramme de classe des rôles par défaut

—
—
—

Role Community Staff
Technical admin
_userManager.AddToRole(user.UserName, "Staff");

Le choix de la méthode est très important. Si vous pensez que votre application sort des 99% évoqués au dessus, n'hésitez pas à venir poster la question sur le forum avec le tag [asp.net], nous serons heureux de vous répondre.

```
class RoleWithAvatar: IdentityRole:
{
public RoleWithAvatar(): super(){
    AvatarUrl = "DefaultAvatarUrl";
};
public RoleWithAvatar(string name): super(name){
   AvatarURL = "DefaultAvatarUrl";
}
public RoleWithAvatar(string name, string url): super(name){
   AvatarURL = url;
}
    public string AvatarURL {get; set;}
}
```

Listing 55 – Notre nouveau groupe

```
public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser, RoleWithAvatar, string,
    IdentityUserLogin, IdentityUserRole, IdentityUserClaim>
    {
        public ApplicationDbContext()
            : base("DefaultConnection", throwIfV1Schema: false)
```

```
5           {
6
7           }
8
9           public static ApplicationDbContext Create()
10          {
11              return new ApplicationDbContext();
12          }
13      }
```

Listing 56 – Un petit changement dans les rôles.

Si dans un de vos contrôleurs vous avez besoin d'accéder aux rôles, il suffit d'utiliser la ligne suivante `RoleManager roleManager = new RoleManager<IdentityRole>(new RoleStore<RoleWithAvatar>(context));`

13.1.2. [Mini TP] Restreindre les accès

Peut être vous souvenez-vous de ce conseil : *"Il est fortement conseillé de mettre [Authorize]sur toutes les classes de contrôleur puis de spécifier les méthodes qui sont accessibles publiquement à l'aide de[AllowAnonymous]"*. C'est maintenant que nous allons pleinement le mettre en œuvre.

— `AuthorizeAttribute`
—

```

```

`IAuthorizeAttribute`
`OnAuthentication`
`FilterContext` `Result`
`HttpUnauthorizedResult`

Contenu masqué n°14

Contenu masqué

Contenu masqué n°14

```
public AuthorizedForAttribute:AuthorizeAttribute
{
    public AuthorizedForAttribute(string group) super(){
        Role = group;
    }
    public string Role{ get; private set;}
    public override void OnAuthorization(AuthorizationContext
        filterContext){
         super.OnAuthorization(filterContext);
         if(filterContext.Result != null && filterContext.Result
             instanceof HttpUnauthorizedResult){
            // on sait que ce n'est pas autorisé
            return;
          }
         if(!filterContext.HttpContext.User.IsInRole(Role)){
              filterContext.Result = new HttpUnauthorizedResult();
         }
    }
}
```

Listing 57 – Notre attribut, ce n'est qu'une solution possible.

Retourner au texte.

14. Validez vos données

14.1. Ajouter des contraintes aux entités

Élément externe (Video) — https://www.dailymotion.com/embed/video/x1vyn8x

ValidationAttribute Mo delState IsValid false IsValid ModelState false

```
1  public class StringLengthRangeAttribute : ValidationAttribute
2  {
3      public int Minimum { get; set; }
4      public int Maximum { get; set; }
5
6      public StringLengthRangeAttribute()
7      {
8          this.Minimum = 0;
```

```
9           this.Maximum = int.MaxValue;
10      }
11
12      public override bool IsValid(object value)
13      {
14          string strValue = value as string;
15          if (!string.IsNullOrEmpty(strValue))
16          {
17              int len = strValue.Length;
18              return len >= this.Minimum && len <= this.Maximum;
19          }
20          return true;
21      }
22  }
```

Listing 58 – un exemple de `ValidationAttribute`

— `IsValid(object value)` `IsValid(object value, ValidationContext context)` pas mal d'informations

— `RequiresValidationContext` `true`

14.1.0.1. Bonnes pratiques

— `Required`

`true`

— `as XXX` `null`

14.2. Un peu plus loin : les MetaType

14.2.1. Le concept de méta données

—

—

`ElementWithAntiSpamText`

```
1  [MetadataType(typeof(AntiSpamProtectedMetaData))]
2  public partial class ContactForm
3  {
4      [MaxLength(128)]
5      public String Subject {get; set;}
6      [EmailAddress]
7      public String UserAddress {get; set;}
8  }
9
10 [MetadataType(typeof(AntiSpamProtectedMetaData))]
11 public partial class Comment
12 {
13     [MaxLength(256)]
14     [RegularExpression("/^https?://[a-zA-Z0-
       9_.-]+\.[a-z]{2,10}(/[a-zA-Z_%0-9.\?/-])*")]
15     public String WebSite {get; set;}
16     [EmailAddress]
17     public String UserAddress {get; set;}
18 }
19
20 public class AntiSpamProtectedMetaData
21 {
22     [Display(Name = "Votre message")]
23     [AVeryPowerfulAntiSpam(ErrorMessage = "Votre message a été
       détecté comme spam.")]
24     [Required(ErrorMessage = "Merci d'entrer un message.")]
25     public string Text { get; set; }
26 }
```

Listing 59 – Deux types qui se servent d'un `MetaType`

15. Aller plus loin avec les routes et les filtres

15.1. Les routes personnalisées

forums

souscategorie/un_numero/un_titre forums/titre_categorie/titre_sous_categorie

slug_category slug_sub_category

http://localhost:port/Forum/Index/cat/subcat

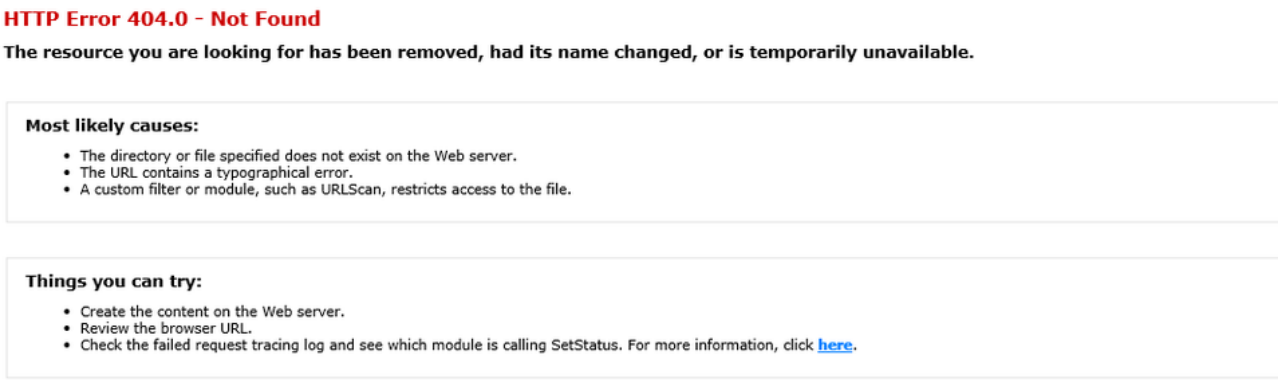


FIGURE 15.1. – page introuvable

http://localhost:port/Forum/Index/?slug_category=cat&slug_sub_category=subcat

Route

```
[Route("Forum/Index/{slug_category}/{slug_sub_category}")]
public ActionResult Index(string slug_category, string
    slug_sub_category)
{
    return View();
}
```

Route

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapMvcAttributeRoutes();
    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id =
            UrlParameter.Optional }
    );
}
```

? Si deux routes se ressemblent, comment je fais pour les départager ?

```
[Route("Forum/Index/{slug_category}/{slug_sub_category}")]
public ActionResult Index(string slug_category, string
    slug_sub_category)
{
    return View();
}
[Route("Forum/Index/{slug_category}/{author_email}/")]
///
/// Imaginons que nous voulons avoir la liste des postes par une
    personne précise dans une catégorie.
///
public ActionResult Index(string slug_category, string
    author_email)
{
    return View();
}
```

Listing 60 – Un conflit possible

`Forum/Index/un_string/un_string/`

[24]

`IRouteConstraint` `Match`

```
public class IsSlugConstraint : IRouteConstraint
{
    public bool Match(HttpContextBase httpContext, Route route,
        string parameterName, RouteValueDictionary values,
        RouteDirection routeDirection)
    {
        return values[parameterName].toLower() ==
            values[parameterName] && values.Replace("-",
            "").Replace("_", "").isAlphaNumeric();
    }
}
```

Listing 61 – Notre contrainte de slug.

```
routes.MapRoute(
                name: "Default",
                url:
                    ""Forum/Index/{slug_category}/{slug_sub_category}"",
                defaults: new { controller = "Forum", action =
                    "Index"},
                constraints: new { slug_sub_category = new
                    IsSlugConstraint()}
            );
```

24. Un exemple souvent donné est forcer la requête à venir soit du localhost soit d'un réseau particulier. Vous pouvez trouver l'exemple sur developpez.com .

Listing 62 – `RouteConfig.cs`

Cette méthode nous fait perdre l'utilisation des attributs `Route`, par soucis de cohérence n'utilisez-la que si vous définissez toutes vos routes dans `RouteConfig.cs`.

`Route`

`RouteFactoryAttribute`

```
class SlugRouteAttribute: RouteFactoryAttribute{
    public SlugRouteAttribute(string template, params string[]
        slugParameters)
                : base(template)
        {
                Slugs = slugParameters;
        }
    public string[] Slugs {
       get;
       private set;
    }
    public override RouteValueDictionary Constraints
        {
                get
                {
                        var constraints = new
                            RouteValueDictionary();
                        foreach(string element in Slugs){
                                constraints.Add(element , new
                                    IsSlugConstraint());
                        }
                             return constraints;
                }
        }
    }
}
```

Listing 63 – notre attribut personnalisé

```
[SlugRouteAttribute("Forum/Index/{slug_category}/{slug_sub_category}",
    "slug_sub_category")]
public ActionResult Index(string slug_category, string
    slug_sub_category)
{
    return View();
}
```

15.2. Les filtres personnalisés

Action

ViewBag

ActionFilterAttribute

System.Web.Mvc.ActionFilterAttribute

```
1  public class MyFilterAttribute : ActionFilterAttribute{
2      public override void OnActionExecuted(ActionExecutedContext
           context){
3          // appelé lorsque la méthode d'action est exécutée.
4      }
5      public override void OnActionExecuting(ActionExecutingContext
           context){
6          //appelé avant la méthode d'action.
7      }
8      public override void OnResultExecuted(ResultExecutedContext
           context){
9          // appelé une fois le rendu fait .Vous pouvez "annuler"
               l'exécution grâce à l'attribut Cancel
10     }
11     public override void OnResultExecuting(ResultExecutingContext
           context){
12        // appelé avant le rendu. Vous pouvez "annuler" l'exécution
              grâce à l'attribut Cancel
13     }
14 }
```

Listing 64 – Les méthodes de notre filtre

OnActionExecuting OnActio

nExecuted

25

enum

Notons que le fait de surcharger ActionFilterAttribute permet d'avoir une classe parente qui implémente déjà `IFilter`, `IActionFilter` et `IResultFilter`. Ce sont en fait ces

trois classes qui sont primordiales.

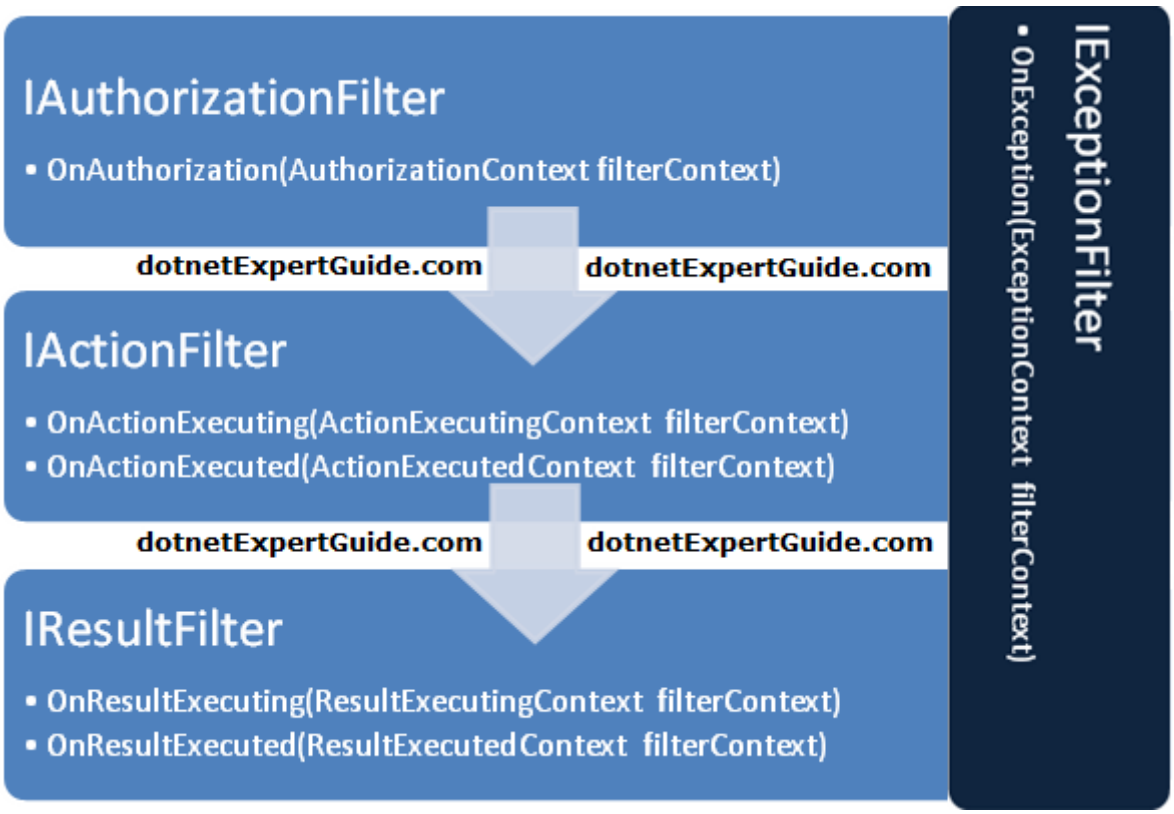


FIGURE 15.2. – L'ordre d'exécution des filtres[26]

15.3. Les filtres en pratique : mini TP calculer le temps de chargement

15.3.1. Quelques indications si l'énoncé ne vous suffit pas :

—

—

— actionExecu
tedContext.ActionDescriptor.ActionName

— DateTime

25. on peut par exemple utiliser influxdb ou prometheus pour stocker les métriques et les afficher dans des beaux dashboard. C'est très pratique quand votre application grandit.
26. Source http ://www.dotnetexpertguide.com/2013/02/aspnet-mvc-action-filter-life-cycle.html

15.3.2. La correction

> Contenu masqué n°15

`ExceptionFilterAttribute`

Contenu masqué

Contenu masqué n°15

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http.Controllers;
using System.Web.Http.Filters;

namespace Blog.Tools
{
    public class
        TimeTracingFilterAttribute:System.Web.MVC.ActionFilterAttribute
    {
        private DateTime start;
        private DateTime end;
        public override void
            OnActionExecuted(ActionExecutedContextactionExecutedContext)
        {
            base.OnActionExecuted(actionExecutedContext);
            end = DateTime.Now;
            double milliseconds =
                end.Subtract(start).TotalMilliseconds;

                log4net.LogManager.GetLogger(actionExecutedContext.ActionDescr
                .Info(String.Format("temps_de_chargement=%sms",
                    milliseconds));
        }
        public override void
            OnActionExecuting(ActionExecutingContext actionContext)
        {
            base.OnActionExecuting(actionContext);
            start = DateTime.Now;
        }

```

```
28      }
29  }
```

```
1  [TimeTracingFilter]
2  public ActionResult MonAction(){
3
4  }
```

Retourner au texte.

16. Les API REST

au début du tutoriel
API REST

API REST
API REST

—
—
—
—

API

16.1. Un peu d'organisation : les zones

Ce que nous allons voir n'est pas spécifique aux API REST, néanmoins ces dernières sont un sujet parfait pour vous permettre d'aborder le concept des zones.

API REST

API REST

Je sais pas si ça me rassure, j'ai déjà beaucoup de contrôleurs, on peut pas faire mieux "organisé" ? Après tout c'est le titre de ton chapitre

Administration Backend

Recovery

API API REST

API

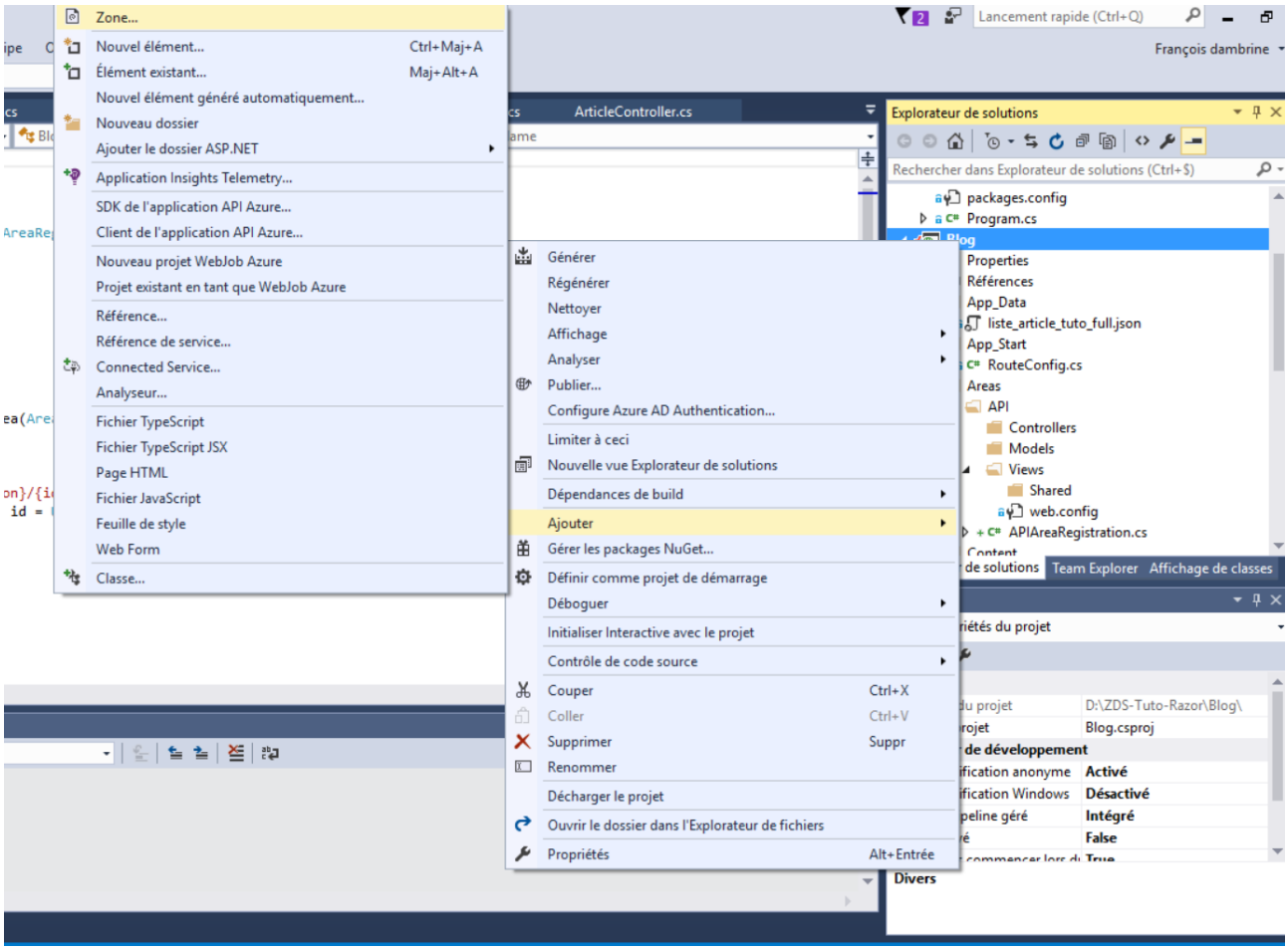


FIGURE 16.1. – Créer une nouvelle zone

IDE

API

`Controller` `Model`

`APIAreaRegistration.cs`

API

```
using System.Web.Mvc;

namespace Blog.Areas.API
{
    public class APIAreaRegistration : AreaRegistration
    {
        public override string AreaName
        {
            get
            {
```

```
11                  return "API";
12              }
13          }
14
15          public override void RegisterArea(AreaRegistrationContext
                context)
16          {
17              context.MapRoute(
18                  "API_default",
19                  "API/{controller}/{action}/{id}",
20                  new { action = "Index", id = UrlParameter.Optional
                        }
21              );
22          }
23      }
24  }
```

Listing 65 – le fichier `APIAreaRegistration.cs`

`RegisterArea`
`RouteConfig.cs` API

Le système fonctionne car durant la génération du code, votre IDE a ajouté la ligne `AreaRegistration.RegisterAllAreas();` à votre fichier `Global.asax`.

16.2. Les bases d'une bonne API REST

API REST

JSON XML
API REST

un tutoriel dédié

16.2.1. Le CRUD

API API
API REST

API REST

API REST

API REST

27

API

16.2.2. Les contrôleurs

CRUD

Controllers

API

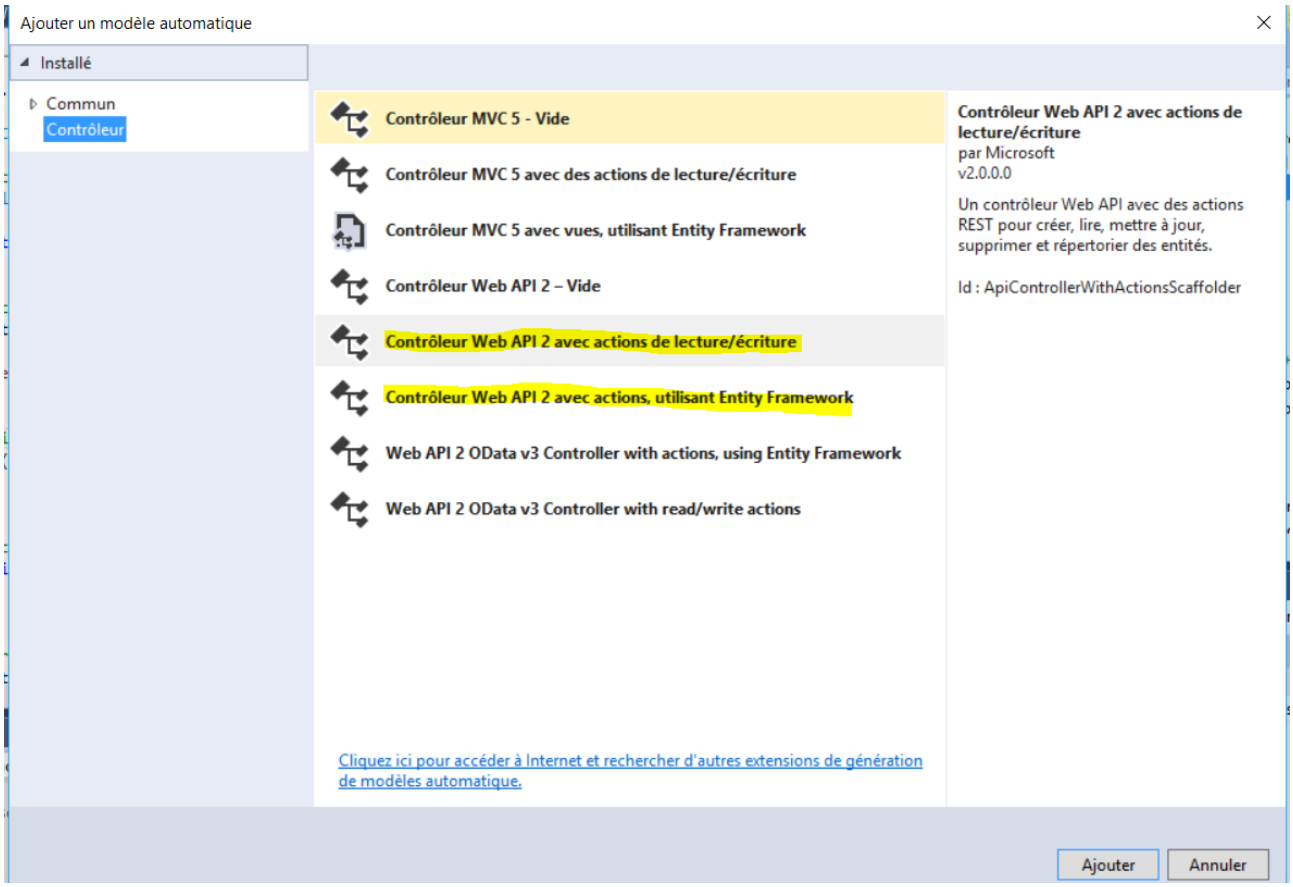


FIGURE 16.2. – Créer un contrôleur : par la suite vous pourrez générer automatiquement le code qui lie votre API à votre base de données.

Global.asax

```
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Routing;

namespace Blog
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            GlobalConfiguration.Configure(WebApiConfig.Register);
        }
    }
}
```

Listing 66 – Le fichier Global.asax

ArticlesController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace Blog.Areas.API.Controllers
{
    public class ArticlesController : ApiController
    {
        // GET: api/Articles
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET: api/Articles/5
        public string Get(int id)
        {
            return "value";
        }
```

```
23
24          // POST: api/Articles
25          public void Post([FromBody]string value)
26          {
27          }
28
29          // PUT: api/Articles/5
30          public void Put(int id, [FromBody]string value)
31          {
32          }
33
34          // DELETE: api/Articles/5
35          public void Delete(int id)
36          {
37          }
38      }
39  }
```

Listing 67 – Le squelette de l'API

`Authorize` `Crea` `teArticle`

`Request.QueryString`

16.3. Le bonus : la documentation

API

—

— API

— API

— API

`HelpPage`

27. Le verbe OPTIONS permet de savoir quelles sont les actions disponibles pour une ressources dans notre cas POST, GET, PUT, DELETE lorsqu'on est auteur. Une ressource en lecture seule n'aura que GET par exemple. Il est très souvent utilisé par les framework tels qu'angularjs .

```
Install-Package Microsoft.AspNet.WebApi.HelpPage
```

Listing 68 – Exécutez cette commande dans la console du gestionnaire de package et vous aurez votre documentation qui se créera.

`http://localhost:2693/help/`

API

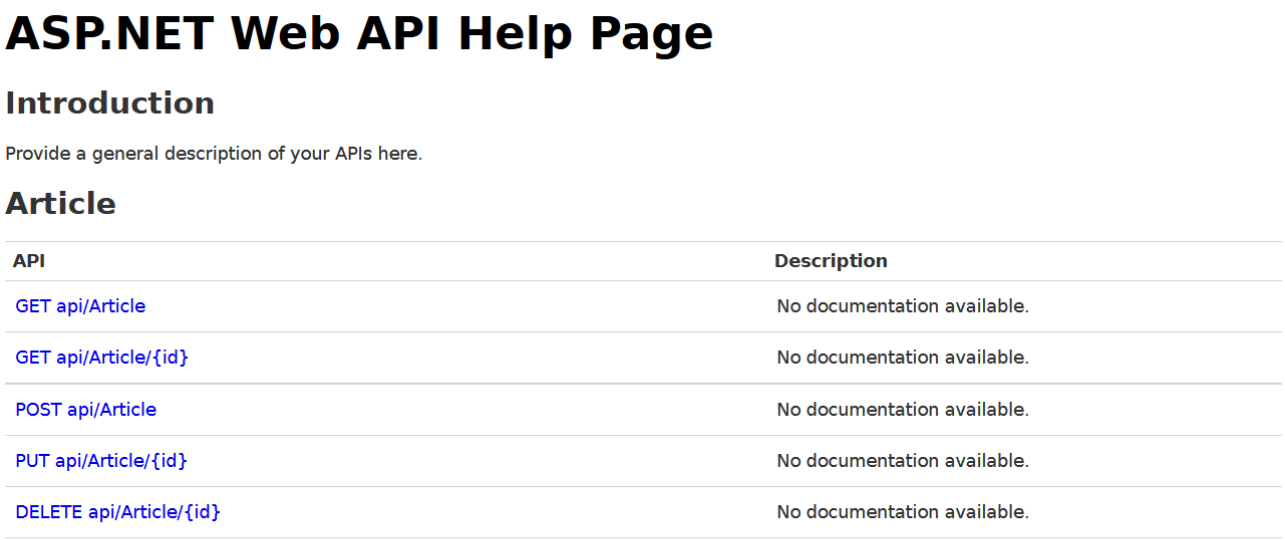


FIGURE 16.3. – La base de votre documentation

`Post`

`String`

`Areas/API/Models`

`ArticleModels.cs`

? Et pour le texte qui dit qu'on n'a aucune documentation ?

ce tutoriel

API

c'est par là

Liste des abréviations

API 1 3 4 25 35 41 43 45 111 136 137 141 142 185 208 215

CRUD 4 137 210 211

IDE 17 22 209 210

JSON 41 42 104 107 115 150 181 184 210

ORM 2 136 139 141 158 165 166

REST 1 4 35 41 42 45 111 208 211

XML 41 42 104 150 210