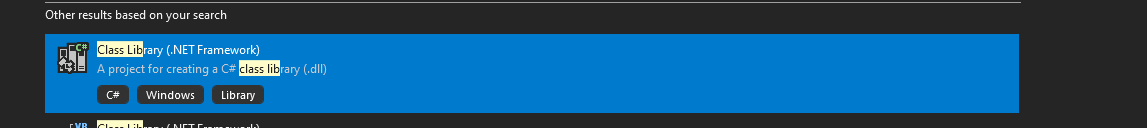
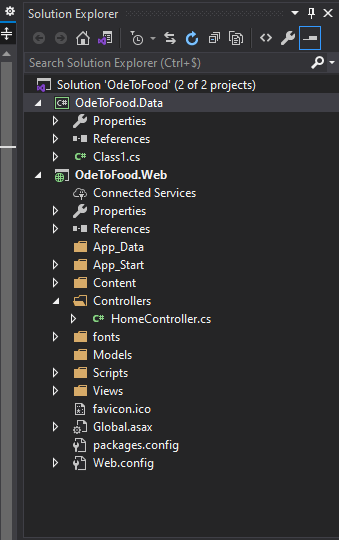
Pour créer un site Web MVC 5 DOT NET Framework C#

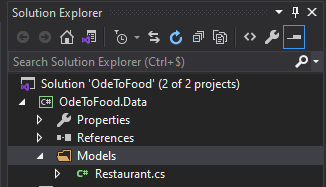
1. Créer un Empty Solution
2. Créer un nouveau projet MVC avec l’extension. Web (même nom que la solution mais avec en plus **« .Web »)**
3. Créer une Classe library .NET Framework, qui pourra être réutilisée ailleurs, et avec le nom du projet et l’extension **« .Data »**



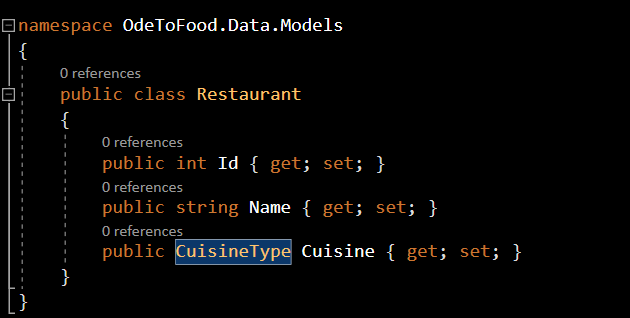
On se retrouve avec ceci :

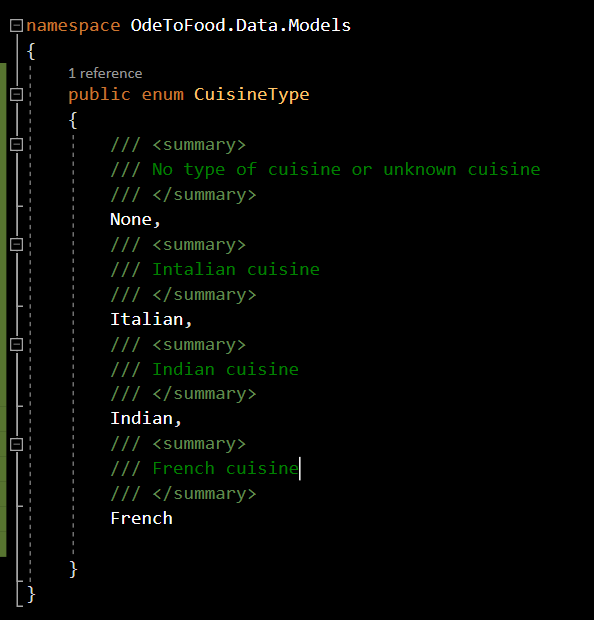


1. On supprime ensuite Class1.cs du projet OdeToFood.Data qui va être remplacée par une classe plus représentative du site (ici un site de restaurant) et qui représente le M de MVC
2. Pour le site il existe au moins 2 catégories d’abstractions :
   1. Une catégorie qui définit la forme des données (Data), comme par exemple une classe Restaurant qui contiendra les informations du restaurant
   2. Une catégorie d’abstraction qui comporte des services qui accèdent à ces données
3. Pour cela dans le projet OdeToFood.Data, on ajoute un répertoire « Models » qui représente le Business Object, ou le model de domaine, ou encore entité (entity) que l’on souhaite stocker dans la base de données
   1. Dans ce dossier on ajoute une classe qui va représenter l’objet Restaurant et que l’on rend public (par défaut elle est private)



1. Dans la classe il nous faut définir des attributs qui représentent la base de données
2. Il nous faut également définir une classe qui fait office d’énumération des types de cuisines

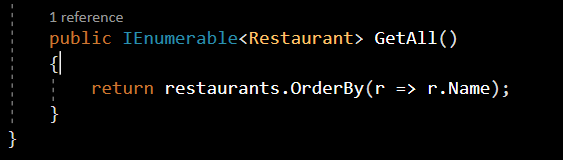




1. On doit ensuite ajouter une abstraction qui simule l’accès au serveur de base de données (**MOCK**)
   1. On ajoute pour cela un répertoire appelé « Services » et une Interface appelée « RestaurantData.cs » (il existe d’autres façons de faire)
   2. On passe l’interface en public
   3. On doit être capable d’ajouter, de mettre à jour, et de supprimer des éléments de la base de données
   4. On crée également une classe qui va hériter de l’interface, ou on va implémenter le code de l’interface
   5. Le **MOCK** ici sera défini en « *hard-code* » dans un premier temps afin de pouvoir tester rapidement le site avec des éléments, on créera la base plus tard



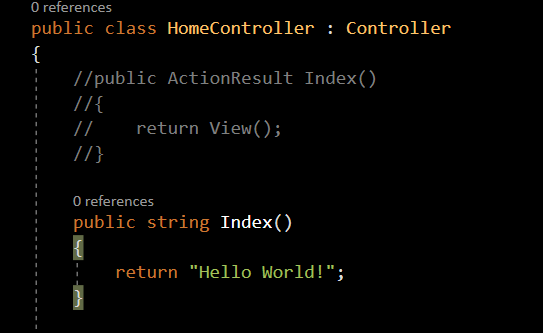
* 1. Enfin, on retourne la liste des restaurants avec la méthode héritée de l’interface, avec l’usage d’un OrderBy par nom (LINQ)

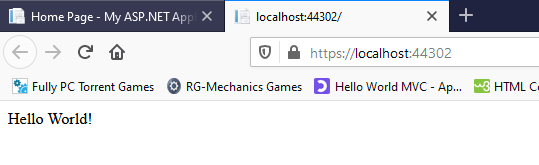


1. Répondre à un message http, signifie comment définir une réponse à un clique sur le GUI, ici on veut pouvoir renvoyer la liste des restaurants si on clique sur un menu « liste des restaurants » par exemple
   1. Pour cela on s’attaque au « **Controller** » qui correspond au C de MVC, par défaut on a lors de la création ceci :

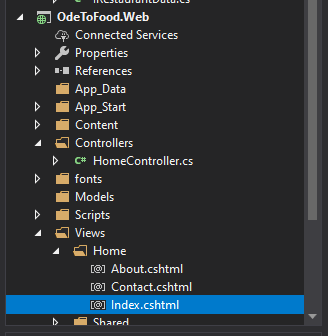


* 1. On a la méthode Index qui permet de faire une requête sur toutes les pages situées à la racine du site. La méthode retourne une « *View* », ce qui correspond au V de MVC
     1. Si on remplace le code existant par une méthode Index qui retourne une chaine de caractères à la place d’un View, et qu’on lance le site (sans debug), on peut s’apercevoir que la page initiale est remplacée par la chaine de caractères souhaitée :





* + 1. On peut inclure dans la chaine de caractères des balises HTML et le Controller va les interpréter comme du code HTML, mais ce n’est pas souhaitable car non lisible et brouillon quant au code C# et à la séparation des codes dans un projet, donc à éviter.
    2. C’est pour cette raison que nous utilisons dans le modèle MVC, la View, qui comportera les éléments à afficher dans la page web
    3. Lorsque j’invoque la View, le Framework va aller regarder dans le dossier « Views », dans « Home », et spécifiquement dans **Index.cshtml** lorsque j’invoque la méthode « Index » (idem pour les autres méthodes reliées à un choix dans la page principale (*root*)

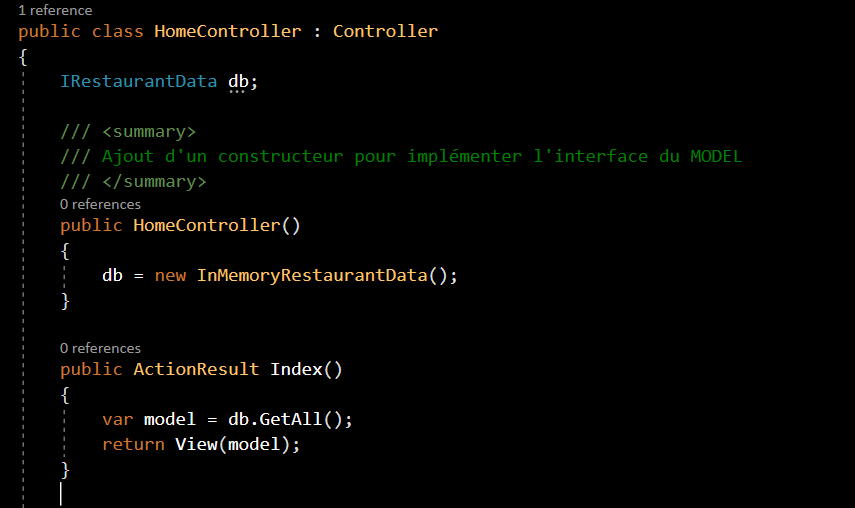


* + 1. Index.cshtml est une extension de View Razor qui est une synthèse de code C# et de balises HTML
    2. Ici on ne veut plus afficher les textes par défauts dans Index, mais la liste des restaurants. On parle du MODEL (M de MVC) et donc on va modifier le code dans la méthode Index afin de renvoyer la liste des restaurant dans la VIEW.

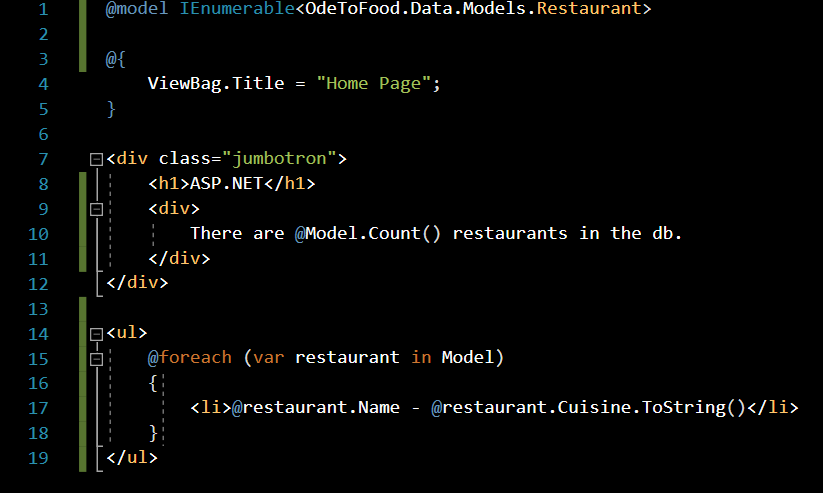
1. On va donc créer le MODEL dans le CONTROLLER qui sera renvoyé dans la VIEW à partir de la méthode index

|  |
| --- |
| **CONTROLLER** : Est destiné à recevoir une requête du client et savoir quoi renvoyer du serveur, dans le cas de la méthode Index de *HomeController.cs*, on renvoi une VIEW. Le CONTROLLER est responsable de la construction du MODEL  **VIEW** : la VIEW s’en fout pas mal des données, de la logique etc… Elle se préoccupe de la présentation des données uniquement  **MODEL** : Le MODEL est responsable de transmettre les données et la logique dans le site uniquement |

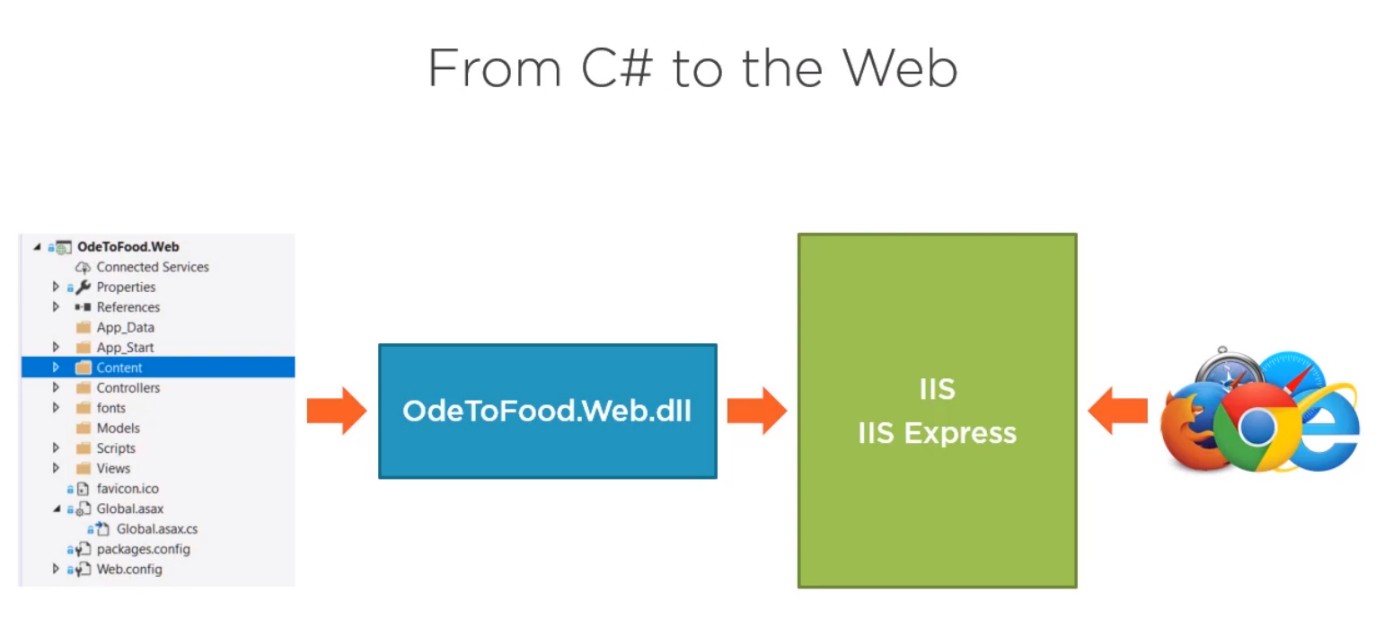
1. Pour faire appel au MODEL, on va implémenter l’interface **IRestaurantData** depuis le CONTROLLER en l’ajoutant dans les références du projet (depuis le projet OdeToFood.Web on charge le using du projet OdeToFood.Data)
2. On ajoute ensuite dans la classe un constructeur du HomeController afin de créer une nouvelle instance de l’interface IRestaurantData
3. Ensuite on va créer un model à partir de l’instance de IRestautantData qui implémente la classe InMemoryRestaurantData(), et faire appel à la méthode GetAll() afin de récupérer une liste des restaurants depuis cette méthode. Le résultat va être implémenté dans les paramètres de l’objet View (model).



1. Dans le fichier Razor VIEW index.cshtml, on peut y mixer du code HTML avec du code C#



**Application startup et configuration**



Fichier qui représente l’application :

* **Global.asax** : On définit ici ce qu’il doit se passer quand l’application démarre.
  + Les différentes parties du code appelées dans Application\_Start() se situent dans le dossier « App\_Start »

|  |  |
| --- | --- |
|  |  |

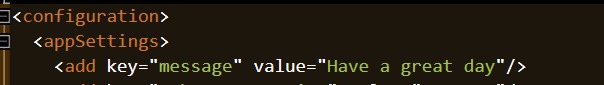
* + - Si on prend **BundleConfig.RegisterBundles** par exemple (dernière ligne) on voit que la méthode statique réunie tous les CSS (Bootstrap + site.css) dans un seul et même fichier de configuration (durant le runtime). Pour s’en rendre compte il suffit de taper dans l’URL /content/css.
    - Si on prend **RouteConfig.RegisterRoutes** qui permet de mapper les URL (celles qu’on entre dans la barre de l’explorer web).
      * La méthode statique **RegisterRoutes** (RouteCollection routes) dit quoi faire avec les requêtes, notamment ignorer certain type de routes avec **routes.IgnoreRoute**
      * La méthode statique **MapRoute** de RouteCollection défini quoi faire lorsqu’il y a une requête http
        + **Controller** est responsable du traitement de requêtes http
        + **Action** est responsable de la réponse qui va être retournée à l’aide de la méthode qui va être appelée dans le ***controller***
        + **Id** est optionnelle mais donne un paramètre de la méthode que l’on veut afficher
        + **On voit dans « defaults » que controller = « Home », action = « Index » signifie que si on tape dans l’URL uniquement** [**https://localhost:44302/**](https://localhost:44302/)**, la route sera reconstituée avec les chemins par défaut « home/Index**

On peut créer un nouveau controller afin de résoudre la route suivante :

[**https://localhost:44302/greeting**](https://localhost:44302/greeting)

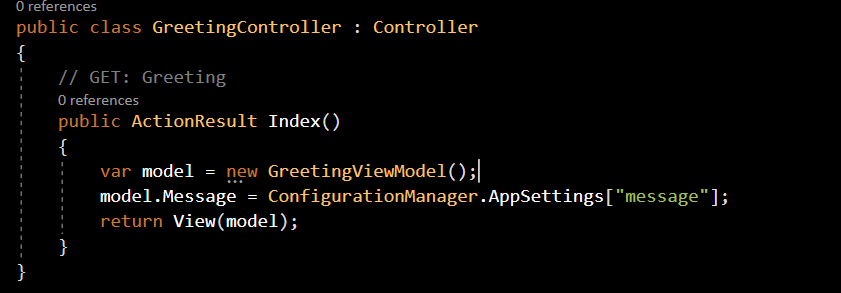
Pour ce faire on va créer un nouveau controller dans le répertoire « **Controllers** » et dans la méthode Index() (qui est la méthode par défaut appelée dans MapRoute comme vu ci-dessus), on fait un clique-droit et on créé une VIEW

* **Web.config** : Qui représente et influence également la configuration de l’application. Dans la section <**appSettings**>, on peut ajouter quelle base de données par exemple utiliser. On y indiquera par exemple l’adresse du serveur qui héberge la base de données.
  + **On peut y inclure un message spécifique pour l’application par exemple**:

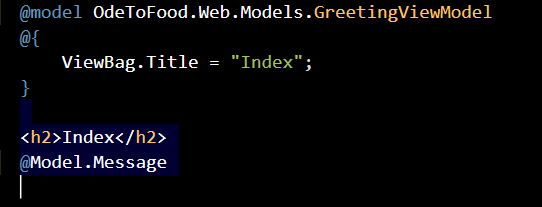


Pour créer donc un nouveau message dans la VIEW à partir de web.config, dans le CONTROLLER et le MODEL on procède comme suit :

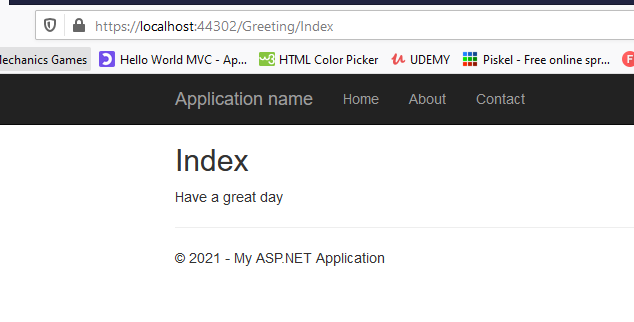
1. On ajoute la clé/valeur dans le xml web.config comme décrit au-dessus
2. On va dans le CONTROLLER prévu pour le « greeting » appelé GreetingController.cs
3. On ajoute une classe qui sera le MODEL (répertoire Models) appelée par exemple GreetingViewModel.cs
4. On y inclus une propriété Message de type string
5. Dans la classe GreetingController.cs dans la méthode Index(), on instancie un nouveau model et on lui dit d’aller chercher le message dans web.config à l’aide de la classe ConfigurationManager et de sa méthode statique AppSettings



1. Pour finir, il reste à afficher le résultat de la méthode Index() qui retourne le View/Model avec le message. Pour se faire on retourne dans le dossier **Greeting** et dans index.cshtml on ajoute l’alias :

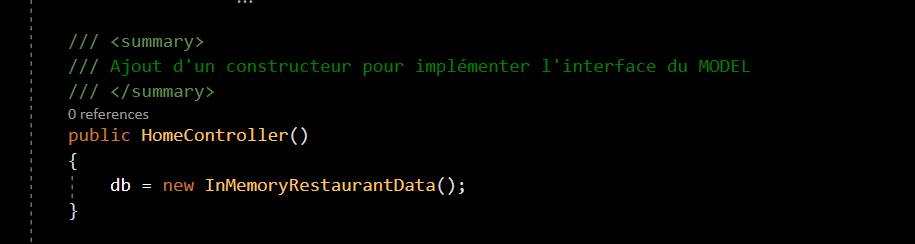


1. On compile et on lance l’appli sans le mode Debug et on obtient bien le message de bienvenue fourni dans le fichier web.config :



Swticher entre une base d’intégration, une base de tests unitaires, une base de production en ASP.NET MVC. Nous souhaitons en fait une flexibilité. Cette flexibilité nous pouvons la trouver dans le Design Pattern « Dependancy injection ».

Le code existant est le suivant :



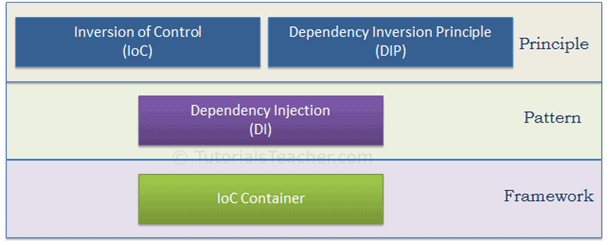
Différence entre Design Principle et Design Pattern

Design Principle signifie mettre en œuvre un haut niveau de bonnes pratiques en vue de développer des applications robustes. SOLID (SRP, OCP, LSP, ISP, DIP) sont des principes de design très populaires.

Par exemple, le principe SPR qui signifie Single Responsability Principle, suggère qu’une classe devrait n’avoir qu’une seule raison de changer.

Un Design Pattern propose des solutions de développement en vue de l’implémentation dans le cadre de problèmes communs au développement orienté objet.

Par exemple, le Design Pattern Singleton permet d’implémenter un objet spécifique qui ne génère qu’une seule implémentation possible de cet objet.



**Design pattern dependency injection** : Permet de créer une classe indépendante de ses dépendances. Cela signifie que l’on peut utiliser un objet indépendamment de ses créations.

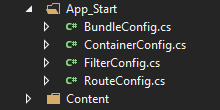
Ce Design Pattern à 4 rôles :

1. Le service que l’on veut utiliser (une classe)
2. Le client qui utilise le service (une autre classe)
3. Une interface utilisée par le client et implémentée par le service
4. L’injecteur qui créé une instance du service et injecte celui-ci dans le client

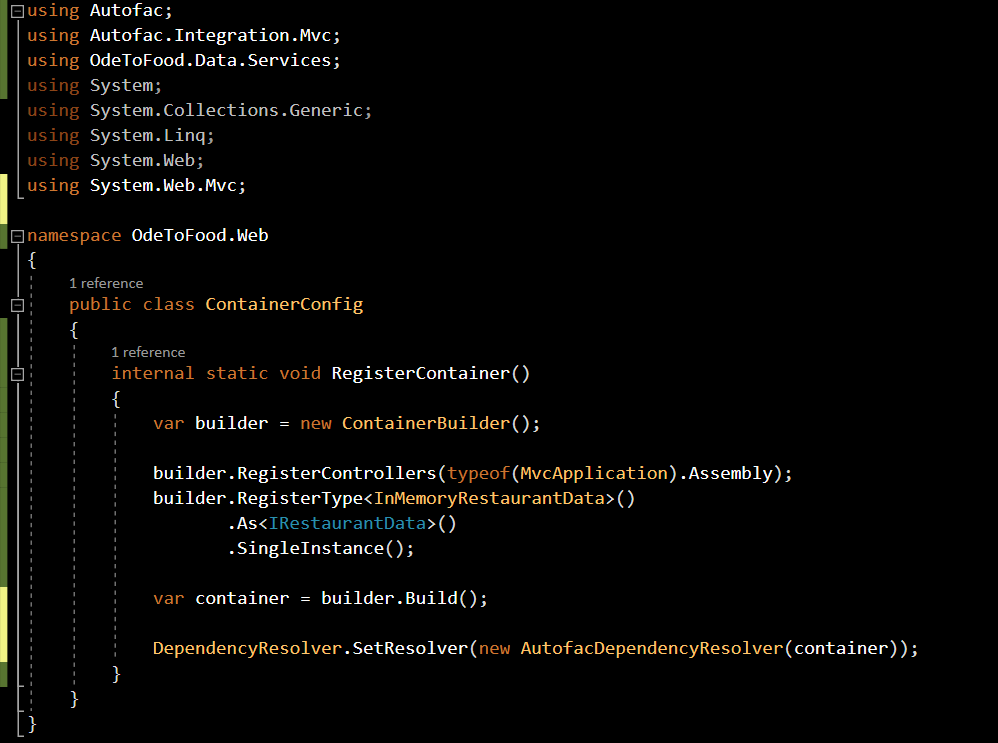
Pour implémenter un injecteur of Control **Container** : on utilisera **Autofac MCV5**. On clique sur référence du projet OdeToFood.Web, add référencé Nugget Packages, et dans Browse, Autofac MVC5

**Autofac** va créer un container qui permet *d’injecter* un une *interface* qui ne se préoccupe pas de ce qu’il y a comme source de données derrière (base de données, xml, json, string…)

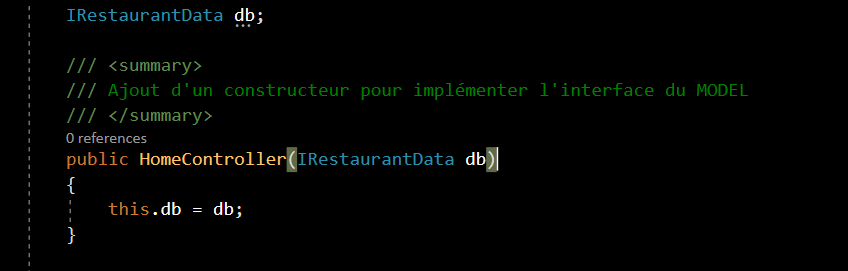
On crée ensuite une nouvelle classe dans App\_Start :



Puis on entre le code suivant (solution tout en un qui permet d’implémenter un Container):



A l’aide de **Autofac** et de la nouvelle classe, nous pouvons modifier le code de **HomeController** afin de recevoir **IRestaurantData**, sans se préoccuper du type d’implémentation qu’il y a derrière :

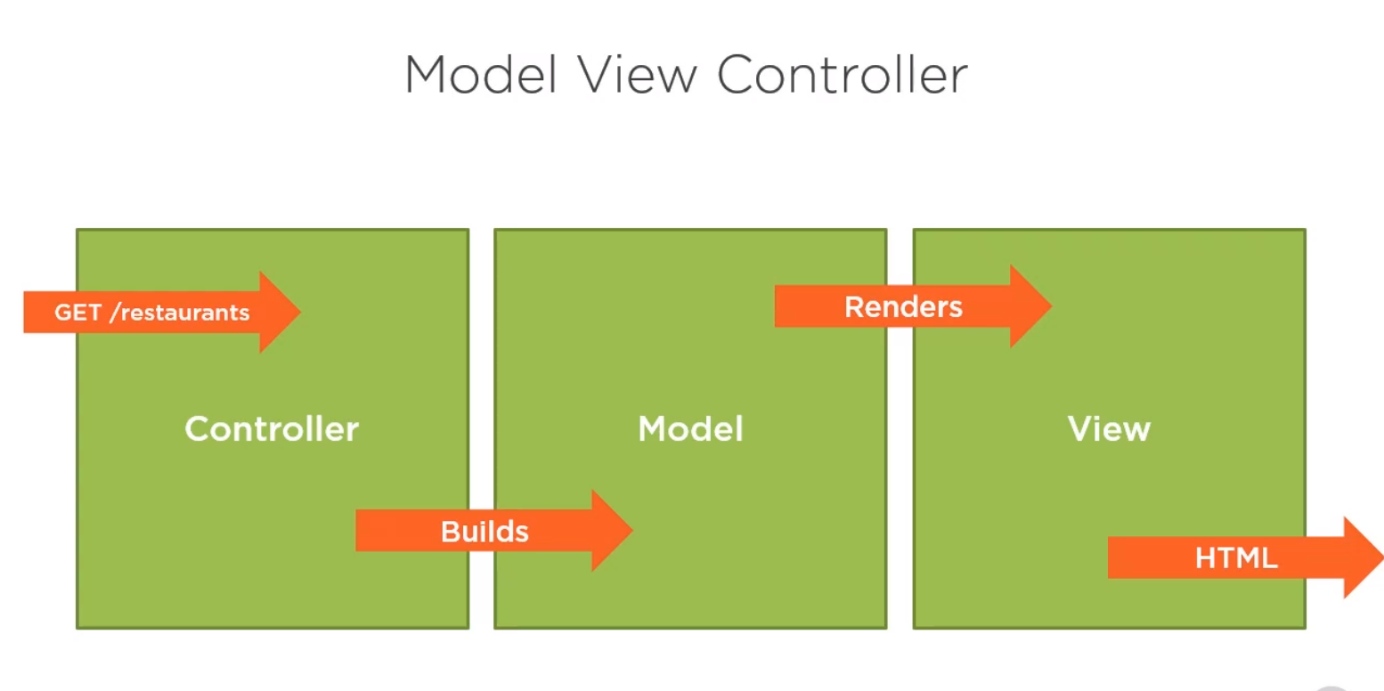


|  |
| --- |
| **Tight Coupling**, signifie qu’une classe est dépendante d’une autre classe.  **Loose Coupling** signifie qu’une classe est dépendante d’une interface. |

On utilise Dependancy Injection Pattern pour créer des applications qui dépendent d’interfaces (**Loosely-coupled**) au lieu de classes (**Tightly-coupled**)

Comprendre le Controller dans MVC

MVC permet de séparer les responsabilités.



Le CONTROLLER reçoit un GET, et devra répondre. Il va construire la réponse en faisant appel au MODEL.

Le MODEL encapsule tout ce que l’utilisateur a besoin de voir sur la page web. Après que le CONTROLLER ait construit le MODEL, il va générer la VIEW qui va présenter les données sous formes élégantes à l’utilisateur.

La VIEW va générer la page HTML qui sera présentée à l’utilisateur.

Un **CONTROLLER** est quelque chose qui répond à un stimuli extérieur telle qu’une requête **http**. Le **CONTROLLER** va construire des **MODEL** et rendre la **VIEW** pour la présenter au client web.

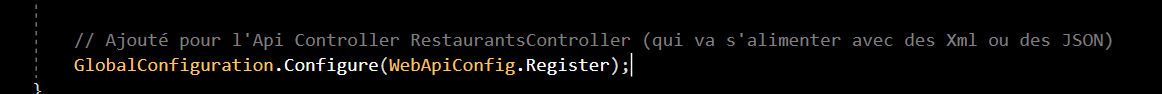
**Dans ASP.NET il existe 2 types de CONTROLLER** :

* Un **CONTROLLER** qui répond à une requête http tel que celui qu’on voit dans notre exemple. La classe hérite de Controller. Ce type de **CONTROLLER** est optimisé pour générer une page web.
* L’autre type de **CONTROLLER** est un **web API CONTROLLER**, qui est fait et optimisé pour construire une API. Ce type de CONTROLLER est fait pour recevoir un **JSON** ou un **XML** ou pour produire un **JSON** ou un **XML**.

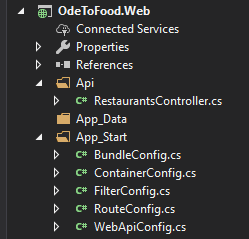
Exemple : On va ajouter un nouveau répertoire dans le projet **OdeToFood.Web** qu’on va appeler API

On va ensuite ajouter un « **New Scaffolded Item** », Web API 2 Controller- Empty appelé ***RestaurantsController.cs***

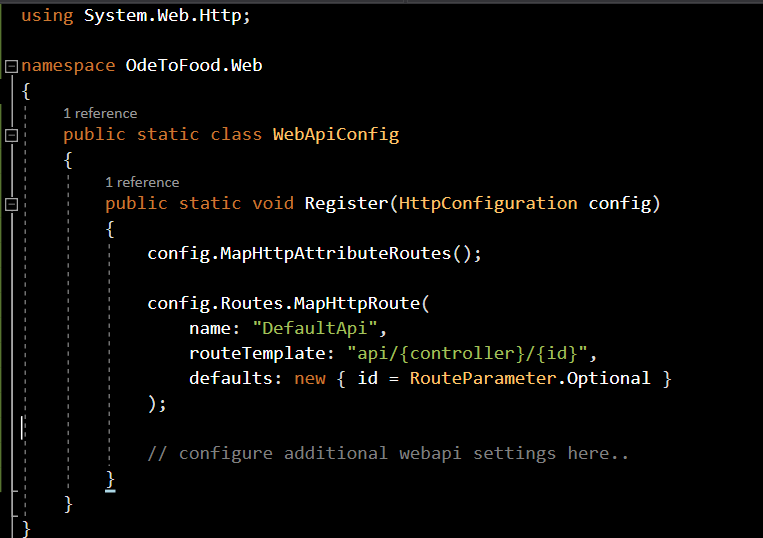
Dans **Global.asax.cs**, on ajoute la ligne suivante qu’il faut placer après AreaRegistration.RegisterAllAreas() (*et non à la fin, car l’ordre est important*) :



**WebApiConfig.cs** devra se trouver dans le répertoire suivant **App\_Start**, qui sont les classes qui s’exécutent lorsque mon application démarre :



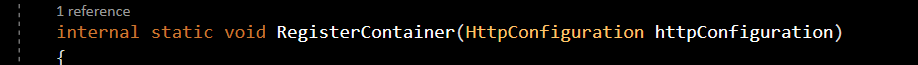
Enfin, cette classe **WebApiConfig.cs** devra comporter les éléments suivants :



On doit ajouter dans ContainerConfig.cs la ligne suivante pour que l’Api Web soit prise en compte :



Et on doit ajouter aussi un paramètre dans la signature de la méthode RegisterContainer :



Enfin il faut ajouter cette ligne aussi pour l’ApiController dans cette même méthode :



Exemple ici : <https://www.c-sharpcorner.com/article/create-simple-web-api-in-asp-net-mvc/>

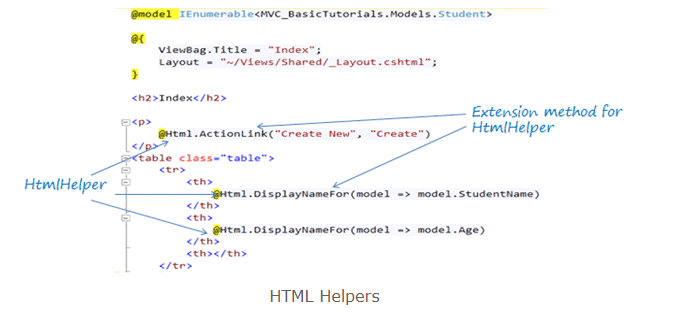
**HTMLHelpers** :

**Razor** est un moteur qui permet de générer du contenu **HTML** à partir de la compilation du code. Si ce code fait appel aux **HTML helpers**, **Razor** va réinterpréter le code des **helpers** pour les traduire en code pure **HTML**.

Lorsqu’on veut créer un formulaire en ASP.NET MVC avec une ***Razor view***, on doit connaitre certains mots clés qui correspondent à des **HTML helpers** :

Les **Html helpers** permettent de rendre le code HTML dans une *view Razor*. Il est conseillé fortement d’utiliser des **Html helpers** plutôt que de créer des tags **HTML** manuellement.

**En revanche, ce n’est pas obligatoire d’utiliser Html helpers pour créer une application ASP.NET MVC. Mais si on les connait bien, on peut créer des View rapidement avec.**



Le but de **Html Helper** est donc de simplifie l’écriture des éléments HTML. Cette classe permet de générer des chaines de caractères qui représentent des tags HTML liés à des données du MODEL.

Par exemple, ***@Html.ActionLink(« Create New », « Create »)*** va générer le tag suivant :

***<a href= « /Student/Create »>Create New</a>***

Plus de méthodes d’extension dans ce lien : <https://docs.microsoft.com/en-us/previous-versions/aspnet/dd493095(v=vs.118)>

Et pour les utiliser :

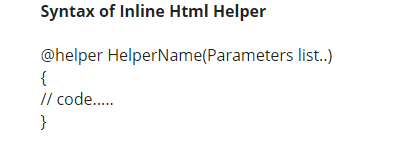
<https://www.c-sharpcorner.com/article/html-helpers-in-asp-net-mvc-5/>

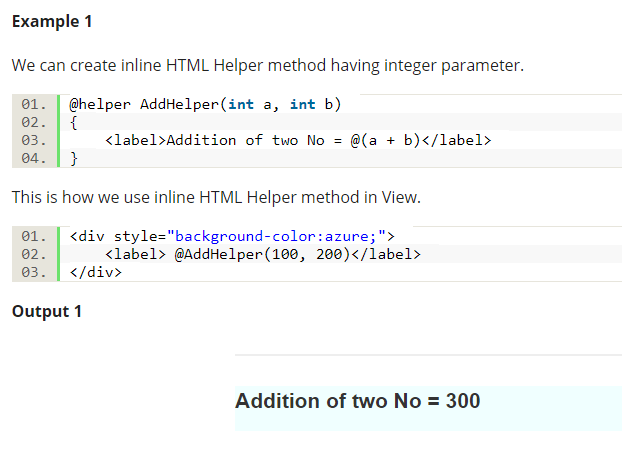
Quelques exemples :

* **@using(Html.BeginForm()){ … }** : Correspond à la balise HTML **<form method= « post » action= «  »></form>**
* **@Html.AntiForgeryToken() :** Correspond à la balise permettant d’éviter qu’un pirate utilise la forme à votre place

Il existe 3 types de HTML Helpers :

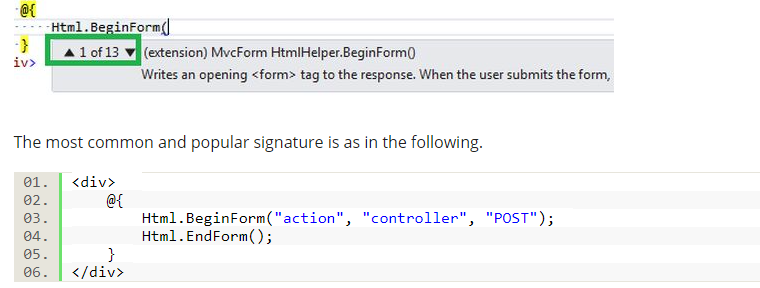
* **Inline HTML helpers** : permet dans une view définie de créer un tag qui va être utilisé dans la view comme une méthode C# pouvant être reprise dans le code de la View (HTML)



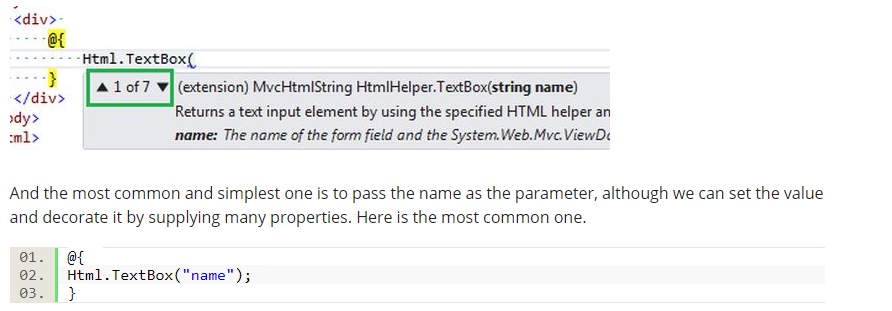


* **Built-In HTML helpers** : Ce sont les helpers standards les plus couramment utilisés pour générer des contrôles HTML tels que des text boxes, des checkboxes et bien d’autres contrôles très fréquemment utilisés dans une page web.

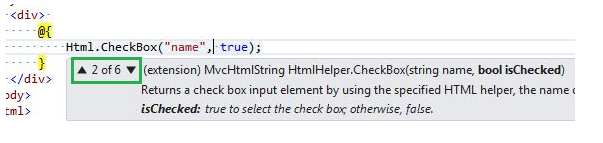
Pour créer une FORM HTML, on peut commencer le code par un Html.BeginForm() et un Html.EndForm()



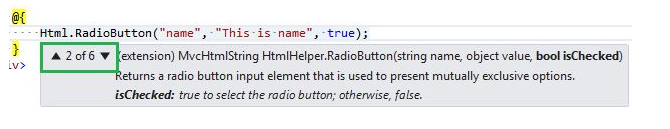
**Textbox** :



**CheckBox** :

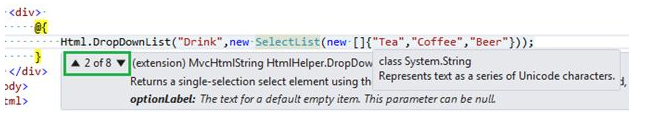


**Radio Button** :



**Drop Down List** :





* **Custom HTML helpers** :

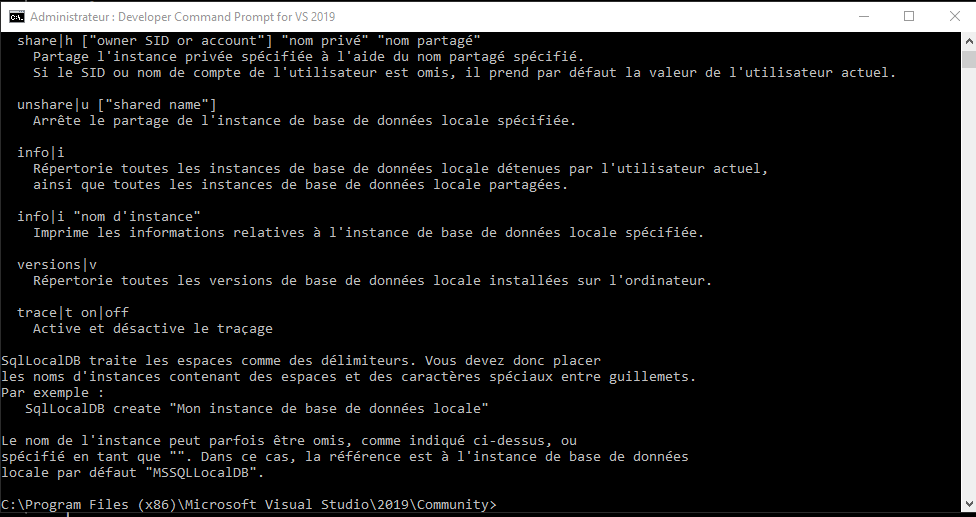
**Entity Framework**

Pour s’exécrer avec SQL Server, il faut vérifier que local.db est bien présent sur le disque.

Lancer dans la barre de recherche Windows Developer Command Prompt for VS 2019

Dans le command prompt exécuter la commande : sqllocaldb

Si la localdb est bien installée, on doit pouvoir trouver la liste des commandes que l’on peut passer avec :

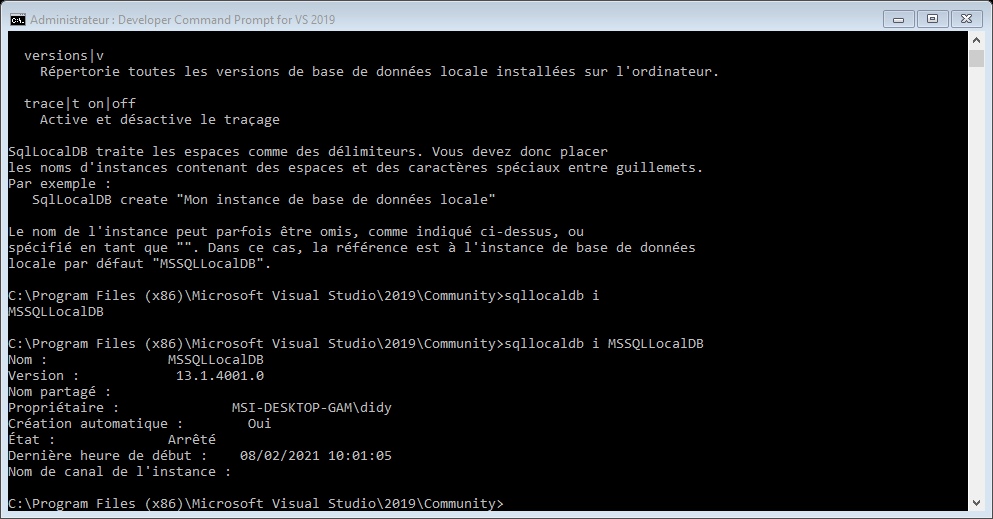


Pour voir quelle instance est présente :

Sqllocaldb i

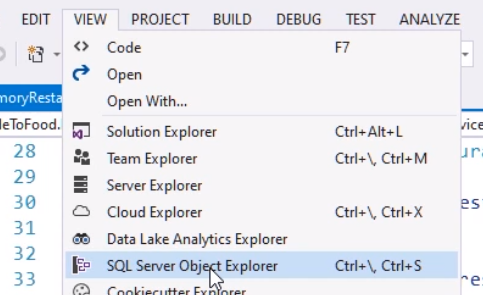


Pour avoir l’état de la base :

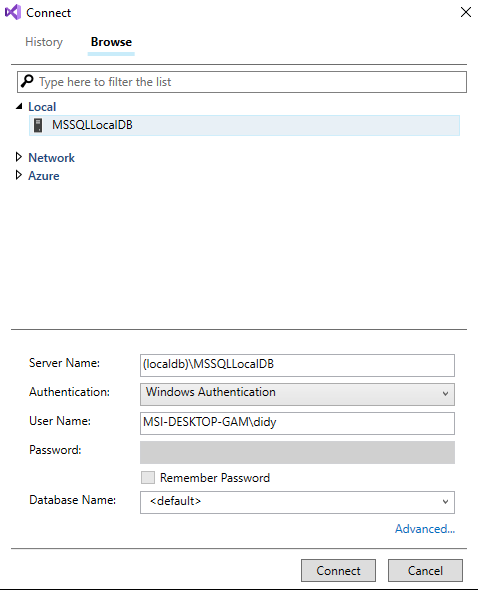


On voit qu’elle est arrêtée

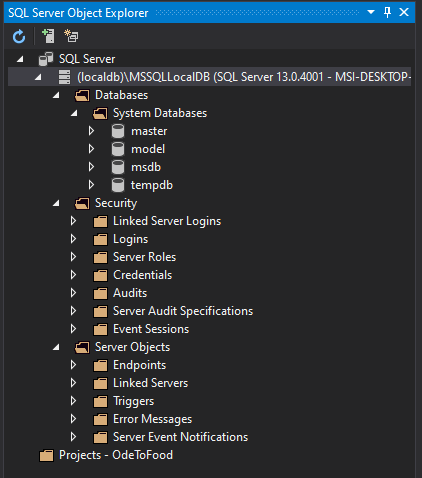
Pour ouvrir la base SQL Server depuis VS 2019 :



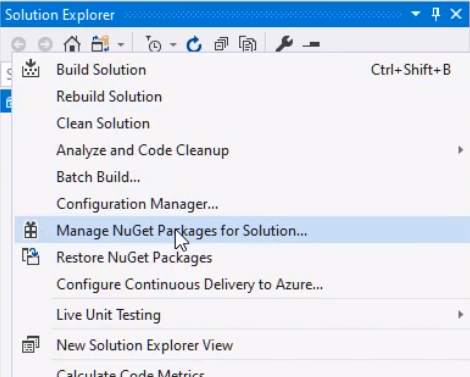
Puis Dans la racine de l’arborescence clique-droit et connecter à une base :



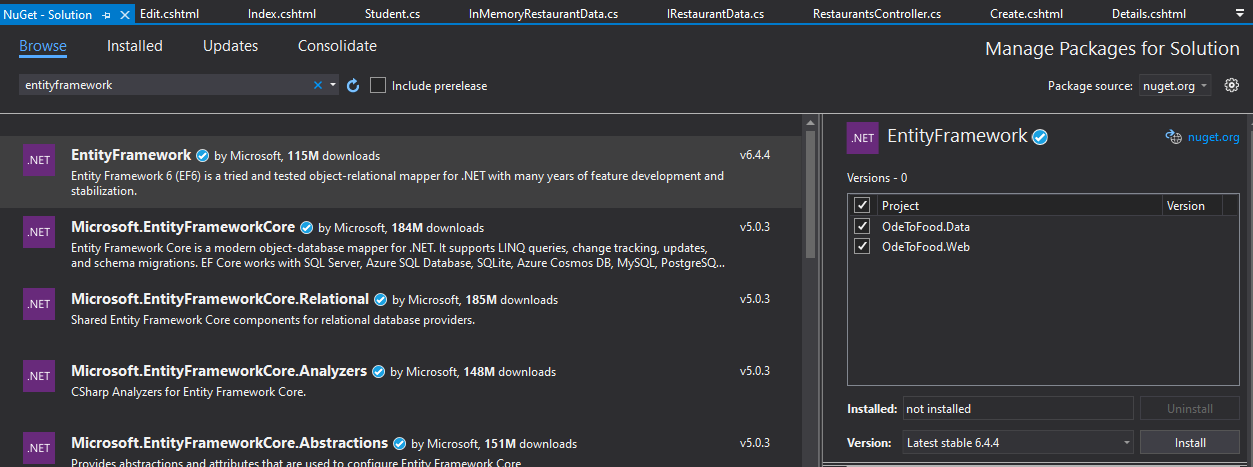
Une fois connecté, la base de données est accessible :



1. Une fois la base locale SQL connectée est accessible, il faut installer quelques nuggets packages
   1. On va installer ces nuggets packages pour les projets OdeToFood.Data et OdeToFood.Web
2. En cliquant droit sur la solution :



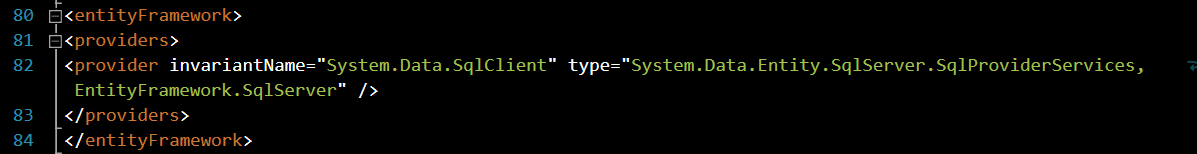
1. Dans « Browse » taper EntityFramework est lancer l’installation sur les 2 projets :



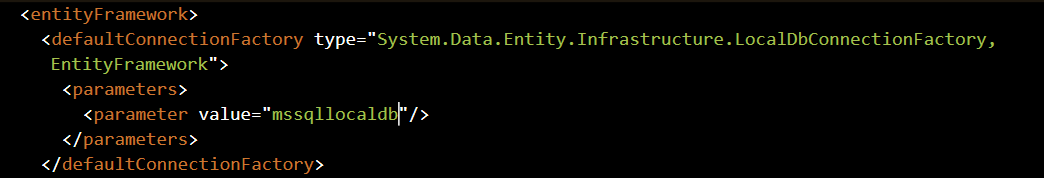
On peut voir dans packages.config que la ligne EntityFramework a bien été ajoutée :



Et dans Web.config :



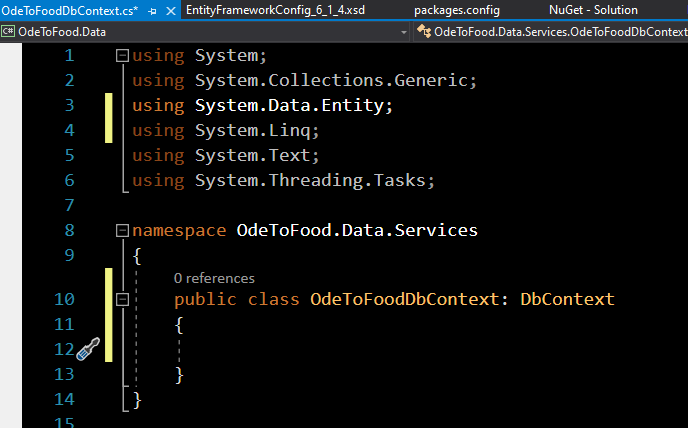
Il manquait cette ligne alors j’ai ajouté :



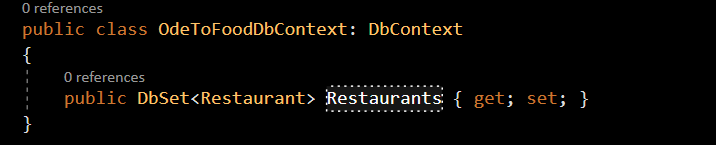
**Stocker les données en dure dans une base de données à l’aide de EntityFramework**

Pour accéder aux données de EntityFramework, il faut créer une DbContext class.

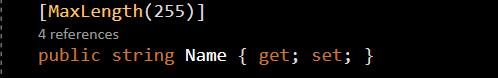
On va la créer dans Services une classe qui va dériver de DbContext :



On va créer une propriété de type DbSet<T> qui permet de pointer une table de type T. Ici la table est Restaurant :



On doit également préciser le type de données dans la classe Restaurant.cs, par rapport aux propriétés :



EntityFramework va créer à notre place la table correspondante dans la base de données

Il faut maintenant créer une classe qui va remplir les mêmes fonctions que la classe de substitution (celle qui nous a permis de faire un Mock) InMemoryRestaurantData.cs.

La nouvelle classe sera celle qui fera les transactions entre la base de données et les objets, elle sera appelée SqlRestaurantData.cs et instanciera IRestaurantData :

