# **Etude de cas : Restaurants**

**L’objectif de ce document** : présentation d’une web API permettant de faire la gestion des menus pour des restaurants.

L’une des principales contraintes était que les plats des restaurants doivaient provenir de l’API TheMealDb. L’application ne prenant pas en compte la gestion du prix des plats, il ne sera présenté dans ce document que la partie BackEnd, à savoir L’API et la base de données. En ce qui concerne l’API externe TheMealDb, nous ne reviendrons que sur les fonctionnalités utilisées dans ce projet. Ce document expliquera donc : la conception, les choix effectués, la présentation technique et la présentation fonctionnelle.

Sommaire

[**Etude de cas : Restaurants** 1](#_Toc133996973)

[1.Récupération du projet 3](#_Toc133996974)

[1.1 Prérequis d’installation 3](#_Toc133996975)

[1.2 Récupération du projet 3](#_Toc133996976)

[1.3 Générer automatiquement la base de données 3](#_Toc133996977)

[2 Architecture de l’application 5](#_Toc133996978)

[2.1 La base de données 5](#_Toc133996979)

[2.2 Les couches applicatives 6](#_Toc133996980)

[3 Spécifications techniques 7](#_Toc133996981)

[3.1 Présentation des technologies employées 7](#_Toc133996982)

[3.2 Présentation global du projet 8](#_Toc133996983)

[3.3 Présentation des deux fonctionnalités à spécifier. 9](#_Toc133996984)

[3.3.1 Ajout d’un restaurant 9](#_Toc133996985)

[3.3.2 Affichage détaillé d’un plat 10](#_Toc133996986)

[4 Présentation fonctionnelle de l’API 11](#_Toc133996987)

[4.1 Lister les restaurants disponibles : 11](#_Toc133996988)

[4.2 Ajouter de nouveaux restaurants 12](#_Toc133996989)

[4.3 Afficher les plats disponibles dans un restaurant (Nom + Image) 12](#_Toc133996990)

[4.4 Afficher une vue détaillée d’un plat (Nom, Image, Catégorie, Ingrédients et quantités) 12](#_Toc133996991)

[4.5 Pouvoir ajouter de nouveaux plats dans le menu d’un restaurant grâce à des propositions de l’API TheMealDb 12](#_Toc133996992)

[**Conclusion** 13](#_Toc133996993)

# 1.Récupération du projet

## 1.1 Prérequis d’installation

Quelques prérequis sont nécessaires pour récupérer le projet. A minima, la version 7.0.0 de dotnet est requise sur le poste de travail. Pour le vérifier, lancez la commande suivante :

dotnet --v

 Si la commande ne renvoie pas la version 7.0.0 minimum alors vous pouvez installer Visual Studio 2022 pour mettre à jour dotnet. Il est également nécessaire que GIT soit installé sur le poste de travail.

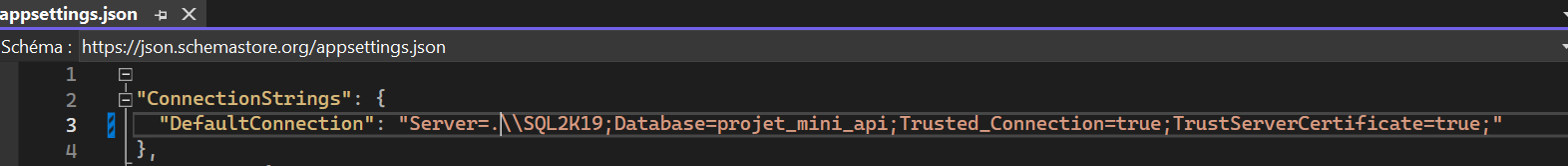
## 1.2 Récupération du projet

Pour récupérer le projet, placez-vous dans le répertoire de destination et lancez la commande suivante :

git clone <https://github.com/lucasmtvilain/projet-mini-api-exercice.git>

## 1.3 Générer automatiquement la base de données

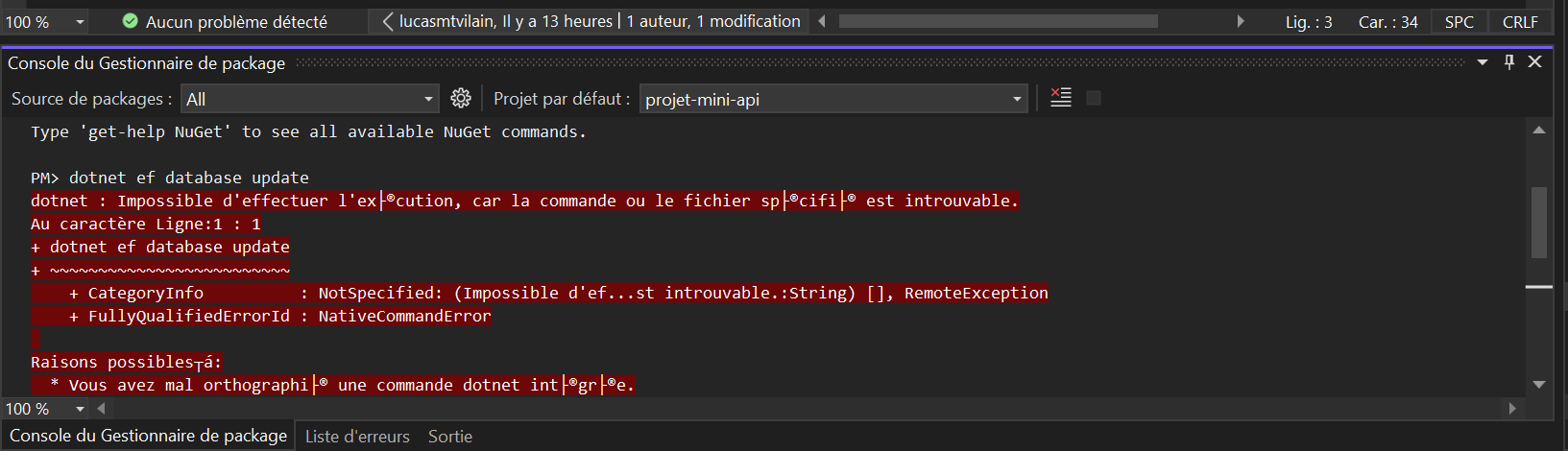
Une fois que vous avez récupéré le projet et lancé Visual Studio, ouvrez le fichier appsettings.json et remplacez la configuration « DefaultConnection » par la configuration qui correspond à votre poste local.



Placez-vous ensuite dans la console de gestionnaire de package et lancez la commande suivante :

dotnet ef database update

Cette commande va générer automatiquement la base de données sur votre instance de sqlServer. Si le message d’erreur suivant apparaît, cela signifie que Entity Framework Tool n’est pas installé sur votre poste.

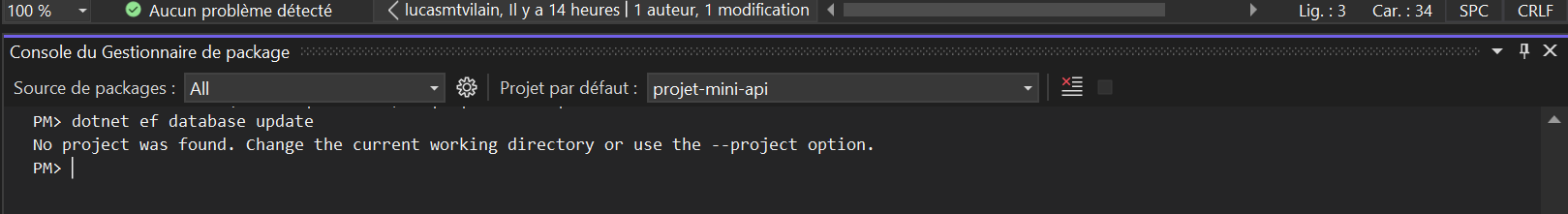


Pour l’installer, lancez la commande suivante :

dotnet tool install --global dotnet-ef --version 7.0.0

(si votre version de dotnet est plus récente que la version 7.0.0, alors vous pouvez installer une version plus récente de dotnet-ef).

Si ce message d’erreur apparaît, cela signifie que vous n’êtes pas à la racine du projet.

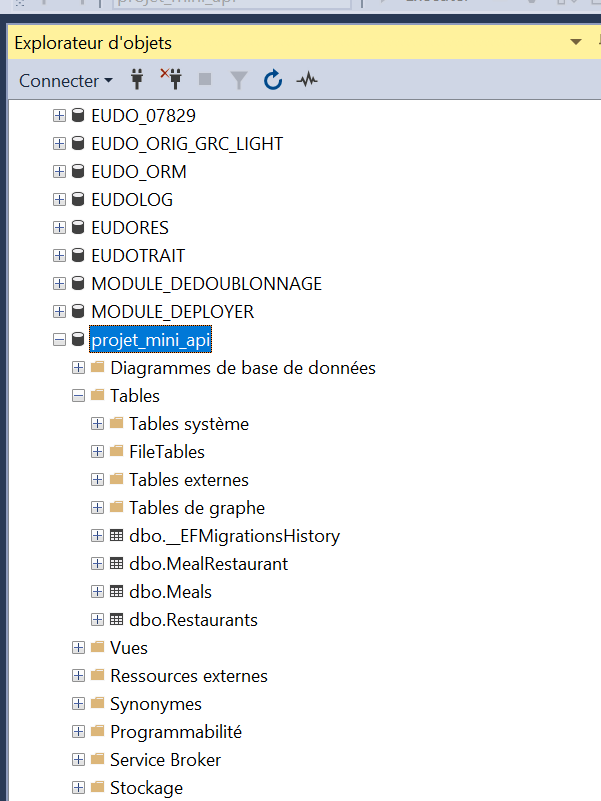


Lancez la commande suivante et recommencez :

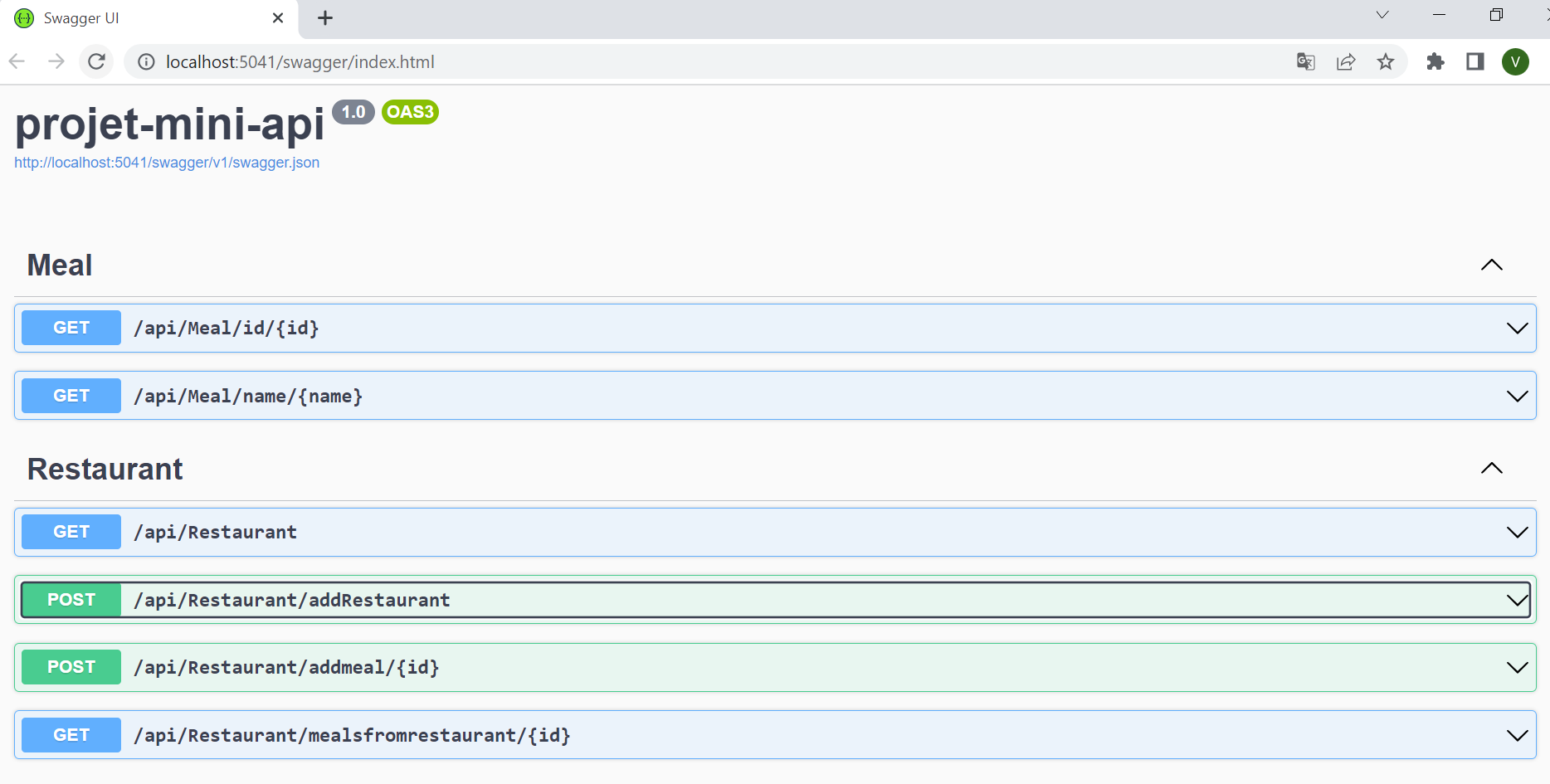
cd .\projet-mini-api

dotnet ef database update

Une fois la commande lancée avec succès, la base de données projet-mini-api sera automatiquement générée sur l’instance de sqlServeur que vous avez paramétrée dans appsettings.Json. Les tables seront générées en fonction du paramétrage du projet (nous y reviendrons dans la partie présentation technique des fonctionnalités).



Une fois la base générée vous pouvez exécuter. Le comportement attendu et que le swagger de l’API se lance dans une fenêtre de votre navigateur par défaut.



# 2 Architecture de l’application

## 2.1 La base de données

Dans ce projet nous avons les entités suivantes :

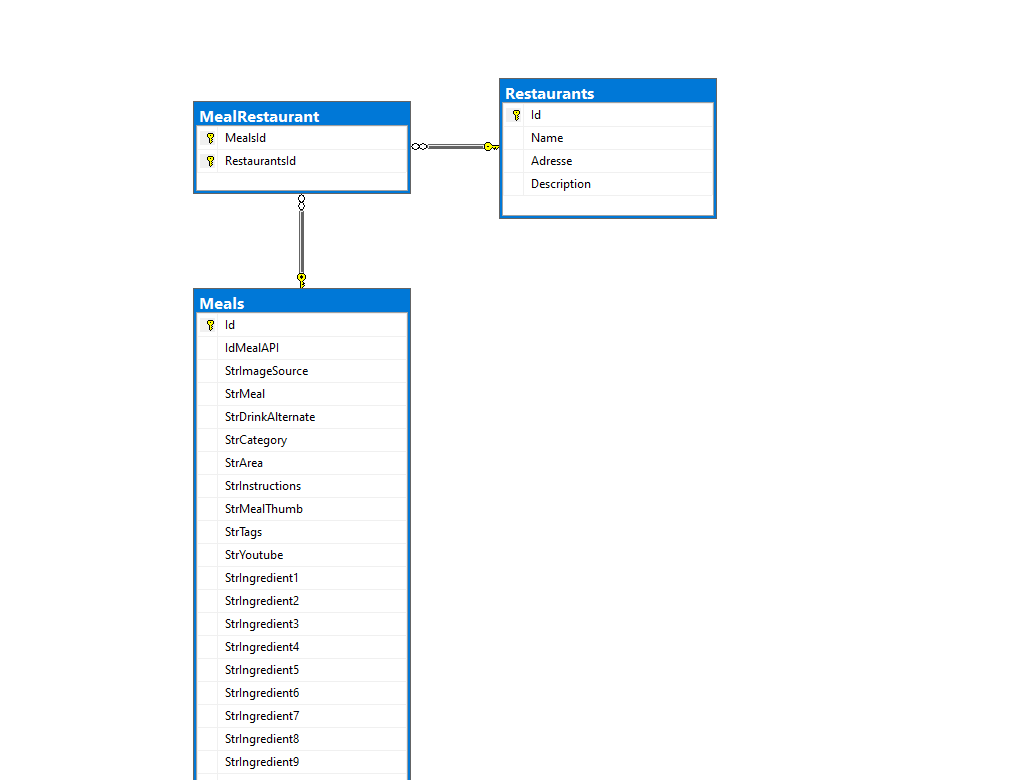
* Les plats qui proviennent de l’API TheMealDb. Les plats ont plusieurs informations comme le lien d’une image, le nom, la liste des ingrédients et leurs quantités.
* Les restaurants ne proviennent pas de TheMealDb. Nous devons donc les stocker dans notre propre base de données. Les restaurants ont également une liste de plats associés.

Ainsi l’une des principales contraintes de ce projet est que les plats et les restaurants ne sont pas stockés dans la même base. Il existe selon moi deux manières de gérer ce problème :

**Option #1 :** Nous pourrions stocker uniquement les restaurants dans notre base de données avec une liste d’entiers correspondants aux identifiants des plats de TheMealDb (idMeal). Ainsi avec l’identifiant idMeal nous pourrions faire des requêtes API pour récupérer toutes les informations d’un plat si besoin. Cela garantit l’exactitude des données des plats de TheMealDb. Le problème de cette solution est que notre API est obligé de faire des requêtes vers l’API de TheMealDb dès qu’un client voudra obtenir des informations sur un plat. Cela n’est pas très gênant pour récupérer les informations détaillées d’un seul plat mais cela devient problématique dès lors qu’un client voudra obtenir la liste de tous les plats d’un restaurant. C’est pourquoi cette solution ne me semble pas être la bonne.

**Option #2 :** Une autre façon de faire serait de stocker les restaurants, les plats ainsi qu’une table qui fait la liaison n/n entre les restaurants et plats sur notre propre base de données. Ainsi nous n’aurions à interroger l’API de TheMealDB que pour ajouter de nouveaux plats. Cette solution présente beaucoup d'avantages car contrairement à la précédente nous n’interrogerons pas l’API TheMealDb dès que les données d’un plat sont demandées car les données sont hébergées dans notre propre base. L’inconvénient avec cette solution est que si les données d’un plat sont mises à jour sur TheMealDb alors les données de notre propre base de données seront obsolètes. Pour palier à ce problème, des mises à jour de plats doivent être proposées par notre API. Pour que cela soit transparent, pour l’utilisateur, nous pourrions mettre à jour les données d’un plat lorsque celui-ci est rajouté à la liste des plats d’un restaurant.

La deuxième solution me semble être la meilleure. C’est pourquoi le diagramme de la base de données est le suivant :



La table [Restaurants] contient les informations d’un restaurant, [Meals] contient les informations des plats de l’API TheMealDb et [MealRestaurant] fait le lien entre ces deux tables.

## 2.2 Les couches applicatives

Notre API doit être capable d’assurer la communication entre 3 éléments :

* Le client FrontEnd qui souhaite récupérer les données des restaurants/plats. Il peut s’agir d’une application mobile ou bien d’un front web.
* La base de données du serveur qui contient les restaurants et leur liste de plat.
* L’API de TheMealDb qui contient les données sur les plats.

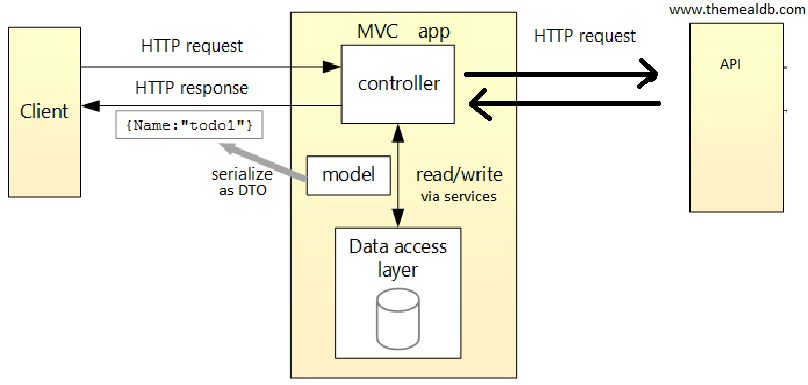


Schéma des couches applicatives de l’API.

(Une partie de ce schéma proviens du lien suivant: https://learn.microsoft.com/fr-fr/aspnet/core/tutorials/first-web-api?view=aspnetcore-7.0&tabs=visual-studio)

Le schéma ci- dessus illustre les différentes couches de l’API. Nous avons ainsi plusieurs cas d’utilisation :

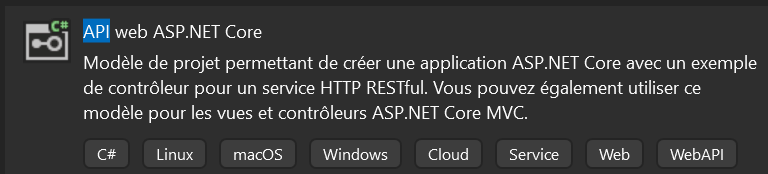
* Si un client effectue une requête http, celle-ci est traitée par un contrôleur qui va réceptionner la requête ainsi que les données de cette dernière (variables get, post dans le body…). Les données sont récupérées au travers de classes DTO et envoyées à des services qui vont les traiter. Les DTO seront alors converties en classes «model » et ajoutées dans la base de données. Une fois le traitement dans le service effectué, le contrôleur renvoie une réponse au client.
* Si le client envoie une requête qui a pour but d’interroger themealdb alors, de la même manière que dans le cas d’utilisation précédent, un contrôleur réceptionne la requête. Cette fois le service instancié ne va pas requêter la base de données mais effectuer une requête client vers www.themealdb.com. La réponse est alors renvoyée à travers une DTO au client.

Pour notre cas, j’ai choisi de créer 2 contrôleurs pour notre API. Un premier contrôleur qui traite les requêtes qui interrogent la base de données (restaurant, liste de plats du restaurant,…) et un autre contrôleur qui interroge uniquement TheMealDB. Ceci afin de ne pas instancier inutilement un context ORM entity framework core si le seul but de la requête n’est pas de requêter des données en base.

# 3 Spécifications techniques

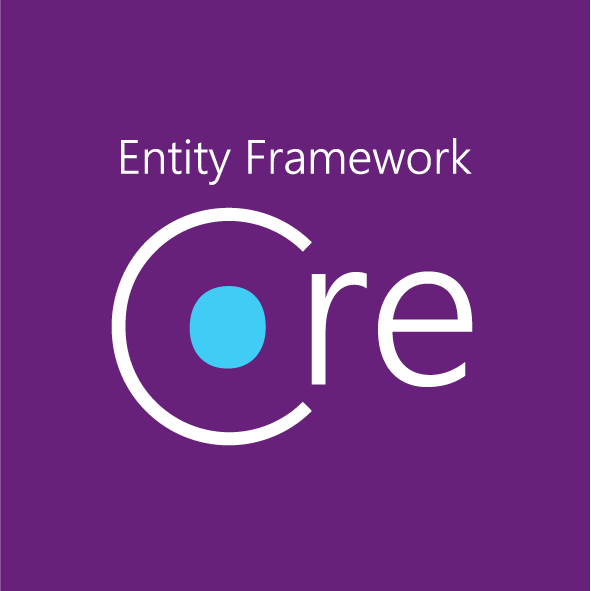
## 3.1 Présentation des technologies employées

Pour ce projet, je suis parti de la template « API web ASP.NET Core ». Cette template offre une base solide pour créer une API REST avec un design pattern cohérent.



https://learn.microsoft.com/fr-fr/aspnet/core/tutorials/first-web-api?view=aspnetcore-7.0&tabs=visual-studio

Pour effectuer la communication avec la base de données j’ai choisi d’utiliser entity framework core car il s’agit d’une solution orm très simple d’utilisation et très documentée, facilitant ainsi l’éventuelle passation du projet par un autre développeur. De plus, EF permet de générer automatiquement les bases de données et de faire automatiquement les migrations en cas d’évolution du modèle de données.



https://learn.microsoft.com/en-us/ef/core/

Etant donnée que j’utilise entity framework core, je dois passer par des classes models. Ces classes models sont des représentations des tables de la base de données. Elles doivent donc respecter des normes de nommage précises pour profiter le plus simplement possible de toutes les fonctionnalités offertes par EF. C’est pourquoi il faut utiliser des classes DTO pour faire l’interface entre les données renvoyées par l’API et les classes models. Faire la conversion de DTO en classe model et inversement, peut être long et fastidieux. C’est pourquoi j’ai choisi d’utiliser la librairie automapper.extensions.microsoft.dependencyinjection. Cette librairie est une surcouche de la librairie automapper et permet de faire les conversions entre DTO et classes models très simplement.



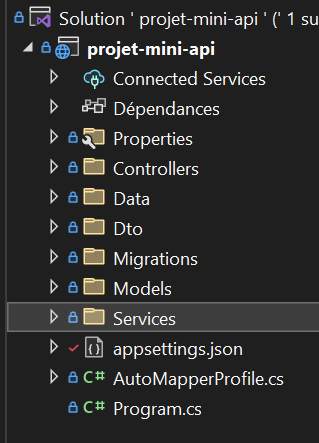
(<https://www.nuget.org/packages/AutoMapper.Extensions.Microsoft.DependencyInjection>)

Pour faciliter les requêtes vers l’API TheMealDB, j’ai choisi d’implémenter un service de requêtage http client. Pour cela j’ai employé IHttpClientFactory. Cela permet de paramétrer dans le programme.cs les instances de HttpClient en fonction du type de la classe service.

<https://learn.microsoft.com/en-us/aspnet/core/fundamentals/http-requests?view=aspnetcore-7.0#typed-clients>

## 3.2 Présentation global du projet

Le projet a la structure suivante :



La première chose que l’on remarque est la disparition du web.config. Maintenant le paramétrage s’effectue directement dans le programme.cs. L’écriture de pragrame.cs a également été simplifiée (disparition de la classe Main et des namespaces).

Les dossiers contiennent les éléments suivants :

* AutoMapperProfile.cs : classe de paramétrage des mappings entre les classes models et DTO.
* Model : contient les classes qui correspondent aux tables de la base de données
* Service : contient les classes de service. Ces classes ont pour rôle de requêter dans la base de données ou bien encore d’interroger l’API TheMealDB.
* Migrations : contient les classes générées automatiquement par Entity Framework core. Ces classes ont pour but de donner le format de la base de données et des tables.
* Dto : contient les classes DTO.
* Data : contient les classes qui permettent d’implémenter le context orm du projet.
* Properties : contient un ensemble de paramétrages pour le projet (dans le cas présent, notamment le swagger).

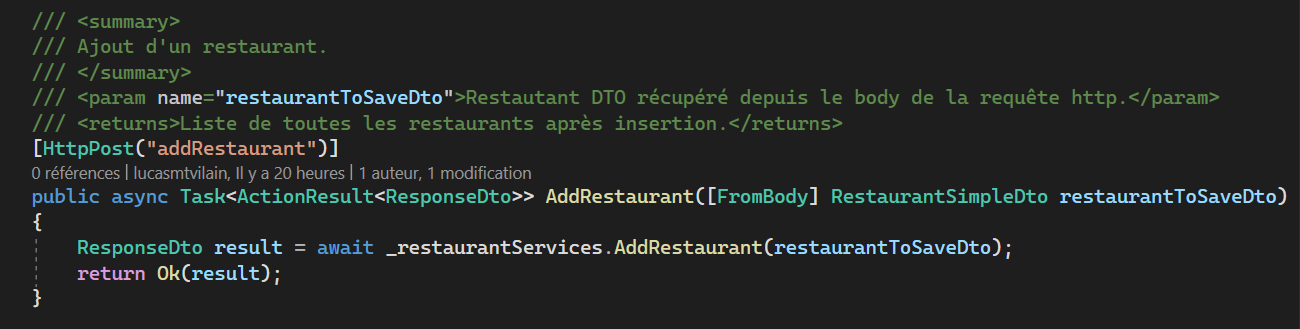
## 3.3 Présentation des deux fonctionnalités à spécifier.

Dans la partie suivante nous présenterons les fonctionnalités d’ajout de restaurants et d’affichage détaillé d’un plat.

### 3.3.1 Ajout d’un restaurant

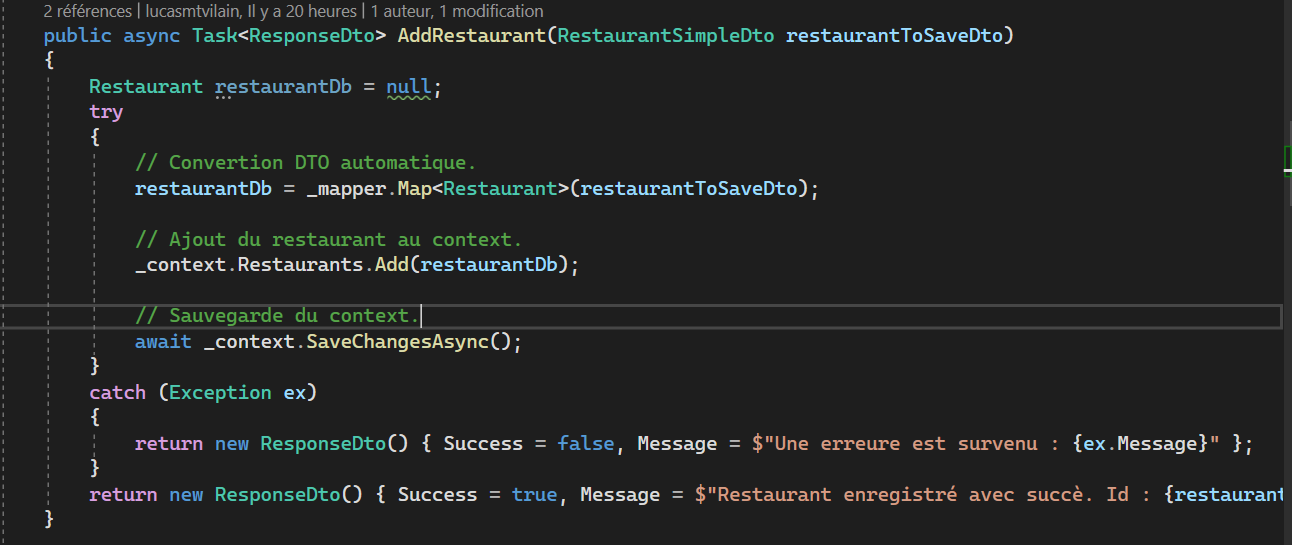
Pour ajouter un plat dans la base de données, il suffit de mettre en place une route API Post qui attend dans le body de la requête http un objet JSON. Pour cela nous devons mettre en place un contrôleur (RestaurantController.cs), un service à lui associer (RestaurantService.cs), une classe model Restaurant ainsi qu’une DTO AddRestaurant.

Le contrôleur :



(RestaurantController.cs)

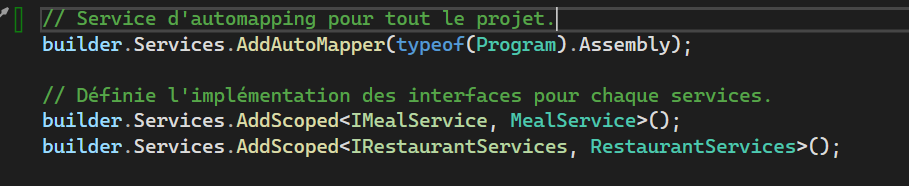
Lorsque vous mettez en place un contrôleur, il est obligatoire d’indiquer via un attribut le type de requête http attendu. Vous pouvez également ajouter entre parenthèses une route supplémentaire ; ceci n’est pas obligatoire mais facilite l’utilisation des routes API. Les variables en entrées proviennent du body de la requête (comme indiqué par le [FromBody]). Ces variables correspondent ici à une DTO. Le contrôleur renvoie un Task car la méthode est asynchrone avec une ActionResult qui embarque une DTO de réponse. Le résultat de la réponse dépend du retour du service.



(RestaurantServices.cs)

Le service récupère en entrée la DTO restaurant avec les données à enregistrer en base. Comme on peut le voir la DTO de réponse dépend du résultat de l’insertion en base. Pour insérer les données en base, nous devons convertir la DTO en model. Ce travail est effectué par la méthode Map() de automapper. Nous ajoutons ensuite notre restaurant à la liste des restaurants du context puis nous sauvegardons le context grâce à la méthode SaveChangesAsync() afin d’insérer le restaurant en base.

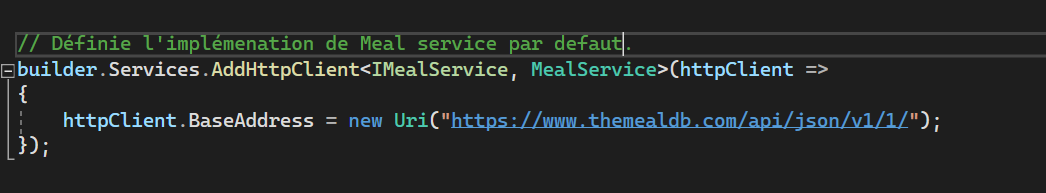
Pour que tout cela fonctionne correctement, il est important de paramétrer la classe programme.cs avec les éléments suivants :



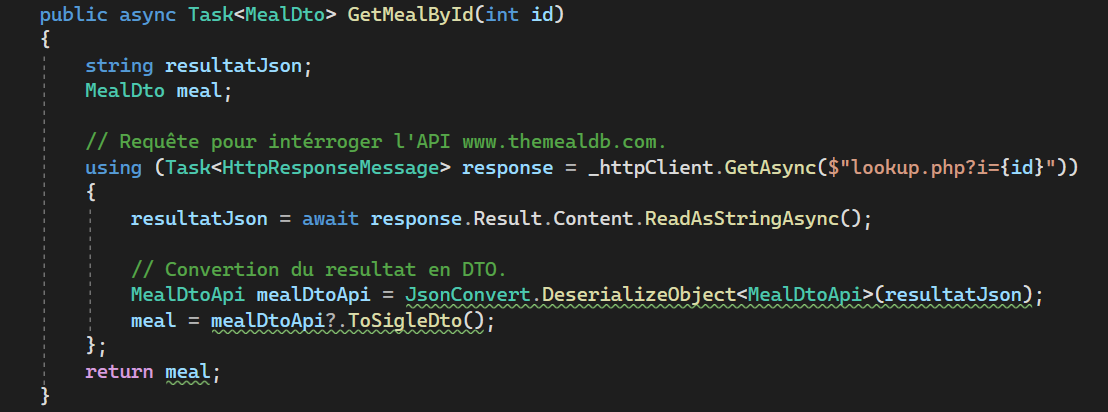
Ces lignes définissent les services d’automapping et les classes services MealService.cs et RestaurantService.cs.

### 3.3.2 Affichage détaillé d’un plat

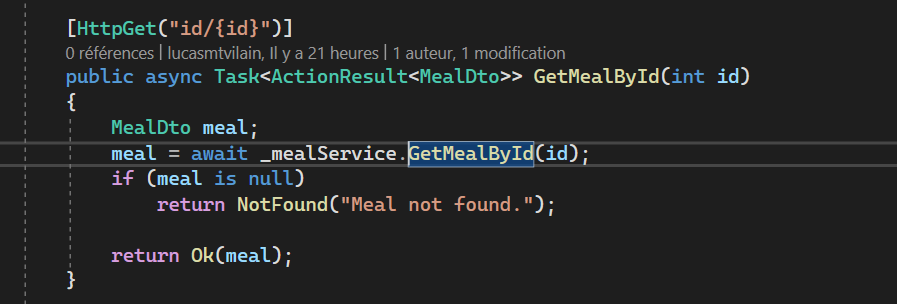
De la même manière que pour le cas précédent, il faut mettre en place le contrôleur, le service et la DTO. Pour cette fonctionnalité, nous interrogerons directement l’API TheMealDB. Il va donc être nécessaire d'utiliser IHttpClientFactory et de le paramétrer pour notre service Meal. Donc dans le programme.cs, il faut ajouter les lignes suivantes afin de spécifier que si une instance de MealService est appelée il faut que le paramétrage de l’url de base soit comme suit :



Nous pouvons alors mettre en place notre contrôleur et le service :



Le service ci-dessus récupère la réponse de l’API et désérialise le JSON dans le body de la réponse.

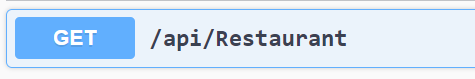


Le contrôleur ci-dessus récupère l’identifiant du plat depuis la requête GET du client et l’envoie au service pour obtenir une réponse. Si aucun plat ne correspond à l’identifiant, alors la réponse paramétrée dans la méthode NotFound est envoyée.

# 4 Présentation fonctionnelle de l’API

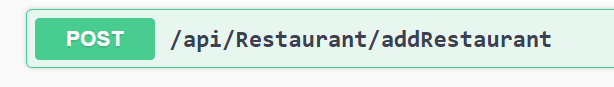
Afin de répondre aux besoins des cinq fonctionnalités demandées, les routes API suivantes ont été mises en place :

## 4.1 Lister les restaurants disponibles :



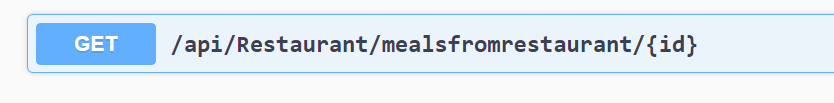
Cette route renvoie la liste des restaurants. Il s‘agit d’une requête GET sans paramètre.

## 4.2 Ajouter de nouveaux restaurants



Cette route prend en paramètre d’entrée un JSON qui correspond à une DTO d’enregistrement du restaurant. Le contenu de la réponse renvoyée dépend de l’insertion en base de données.

## 4.3 Afficher les plats disponibles dans un restaurant (Nom + Image)



Cette route prend en paramètre l’identifiant du restaurant dont la liste des plats est à renvoyer. Les noms, images et images/preview sont inclus dans les données renvoyées.

## 4.4 Afficher une vue détaillée d’un plat (Nom, Image, Catégorie, Ingrédients et quantités)

Pour afficher les vues détaillées de chaque plat, les 2 routes suivantes sont disponibles :



La première route effectue la recherche d’un plat via l’identifiant de ce plat, alors que la seconde route fait la recherche par caractères (ex : premières lettres du plat). Ces 2 routes requêtent les données de [www.themealdb](http://www.themealdb).com.

## 4.5 Pouvoir ajouter de nouveaux plats dans le menu d’un restaurant grâce à des propositions de l’API TheMealDb



Cette route attend en paramètre l’identifiant du restaurant dont le plat est à ajouter. Le seul paramètre attendu dans le body est l’idMeal du plat sur [www.themealdb.com](http://www.themealdb.com). Les informations du plat sont automatiquement alimentées par l’API via une requête sur TheMealDB.

# **Conclusion**

Comme cela a été démontré, la template « API web ASP.NET Core » offre une base solide pour mettre en place une API. Le swagger est très simple à paramétrer. L’implémentation des services est également facilitée, puisque tout est géré dans le programme.cs.

Ce type d’API est idéal pour alimenter les données d’une application web ou mobile. A noter que plusieurs évolutions sont possibles. Nous pourrions mettre en place des routes API PUT pour mettre à jour les données d’un restaurant ou bien une route DELETE. Nous pourrions également ajouter des tokens d’identification pour sécuriser certaines routes API.