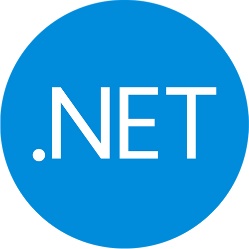
CCI Nîmes

Présentation .Net 

API .Net



Réalisé et enseigné par

Jérôme Ferratier



Table des matières

Table des matières

[Pré requis 4](#_Toc129895673)

[Fonctionnement général d’une API 5](#_Toc129895674)

[Etape 1 Création d’une base de données. 7](#_Toc129895675)

[Etape 2 Création du projet 7](#_Toc129895677)

[Etape 3 Accès à la base de données via Entity Framework. 9](#_Toc129895681)

[Etape 4 Contrôleurs de l'API Web. 12](#_Toc129895683)

[Etape 5 Ajoutons un swagger. 14](#_Toc129895684)

* 1. Pré requis
* MySQL.
* SDK .NET 6.
* Visual Studio 2019/2022.
* .Net C#

# Liens utiles

Création d’une API .Net C# / Mysql

<https://www.c-sharpcorner.com/article/rest-api-with-asp-net-6-and-mysql/>

Création d’un Swagger dans une API .NET C#

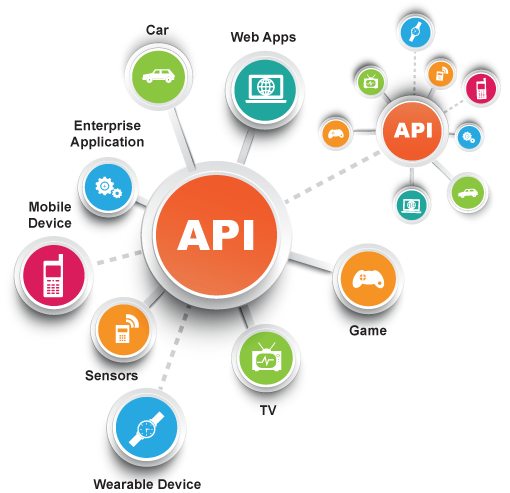
https://learn.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-7.0&tabs=visual-studio

# Lexique

# Fonctionnement général d’une API

Le fonctionnement d’une API REST repose sur les notions de client, serveur, ressources/méthodes et terminaisons (Endpoints).

Une image contenant diagramme

Description générée automatiquement

**Les types d’API :**

* **REST :**
  + **Conditions à respecter :**

Une image contenant diagramme

Description générée automatiquement

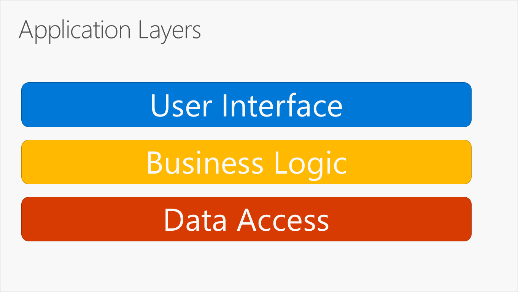
* **RESTFUL :**
  + **Conditions à respecter :**

Une image contenant texte, intérieur

Description générée automatiquement

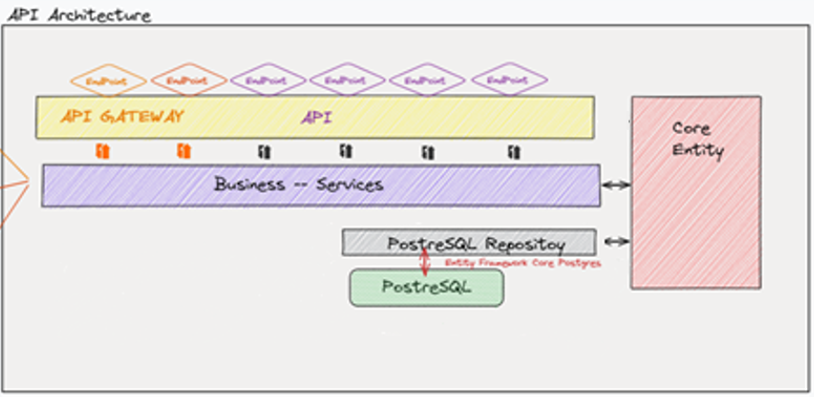
Les différentes couches de conceptions d’une API

L’architecture type de Microsoft mise en place pour nos besoins est composée de 3 couches conventionnelles.

Ce sont les couches les plus couramment adoptées pour les logiques d’application en couches.

Ces couches sont souvent abrégées comme ceci :

* UI (couche d’interface utilisateur : API)
* BL (couche métier- : Business Services)
* DAL (couche d’accès aux données : Core Entity)



# Etape 1 Création d’une base de données.

Commençons par créer cette base de données « demo ».

Créons ensuite une table user avec le script suivant :

CREATE TABLE `user` (

`Id` INT NOT NULL PRIMARY KEY,

`FirstName` VARCHAR(45) NOT NULL,

`LastName` VARCHAR(45) NOT NULL,

`Username` VARCHAR(45) NOT NULL,

`Password` VARCHAR(45) NOT NULL,

`EnrollmentDate` datetime NOT NULL

);

# Etape 2 Création du projet

Dans Visual Studio, nous allons créer un nouveau projet du type API Web ASP.NET Core :

Une image contenant texte, écran, noir, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran, écran, noir

Description générée automatiquement

Une image contenant texte, écran, capture d’écran, téléphone cellulaire

Description générée automatiquement

# Etape 3 Accès à la base de données via Entity Framework.

Avec ce projet, nous allons créer un accès à la base de données et implémenter un contrôleur correspondant pour travailler avec ces données.

**Accès à la base de données avec Entity Framework.**

Pour établir les entités à travers les classes et la connexion de la base de données, nous pouvons utiliser l’approche *Database First* de Entity Framework, qui nous permet d’échafauder la base de données du projet, c'est-à-dire de générer automatiquement des classes en fonction des entités établies dans le base de données et la connexion dans le projet.

Pour cela, il est nécessaire d'installer trois packages NuGet,

* Microsoft.EntityFrameworkCore.Design
* Microsoft.EntityFrameworkCore.Tools
* MySql.EntityFrameworkCore

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Avec l'installation de ces packages NuGet, nous allons maintenant ouvrir la console du gestionnaire de packages pour écrire une commande qui nous permettra d'effectuer *une construction des modèles* à partir de la base de données :

Une image contenant texte, capture d’écran, noir, écran

Description générée automatiquement

Scaffold-DbContext "server=localhost;port=3306;user=root;password=;database=demo" MySql.EntityFrameworkCore -OutputDir Entities -f

Une fois la commande exécutée, les templates sont crées automatiquement ainsi que le « database contexte ».

Une image contenant texte

Description générée automatiquement

Ce n'est pas le plus approprié que la chaîne de connexion à la base de données soit spécifiée dans la méthode OnConfiguring. Pour faire mieux, nous utiliserons le fichier appsettings.json, dans lequel nous pouvons définir cette configuration.

Ce fichier est fait pour ca , et nous allons donc l’utiliser.

* Effacer le contenu de la méthode OnConfiguring

Une image contenant texte

Description générée automatiquement

* Ajouter le DB context dans program.cUne image contenant texte, capture d’écran, écran, fermer

  Description générée automatiquement
* Ajouter la connexionstring retiré précédemment dans appsettings.json

Une image contenant texte

Description générée automatiquement

# Etape 4 Contrôleurs de l'API Web.

Maintenant, nous allons ajouter des contrôleurs.

Ils permettront d'établir des méthodes pour effectuer des opérations CRUD sur les tables de la BDD et de les exposer via l'API Web.

Depuis le dossier Controllers, nous ajouterons un contrôleur appelé UserController:

Une image contenant texte, capture d’écran, écran

Description générée automatiquement

Le code suivant permet d’avoir un CRUD sur une entité « bot ». Ajouter ce code dans votre projet et adapter à votre entité User.

using WebApplication2.Entities;

using WebApplication2.Controllers;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using System.Net;

namespace WebApplication2.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class BotController : ControllerBase

{

private readonly DBContext DBContext;

public BotController(DBContext DBContext)

{

this.DBContext = DBContext;

}

/// <summary>

/// Definition du Web Service

/// </summary>

/// <remarks>Je manque d'imagination</remarks>

/// <param name="id">id du client a retourné</param>

/// <response code="200">client selectionné</response>

/// <response code="404">client introuvable pour l'id specifié</response>

/// <response code="500">Oops! le service est indisponible pour le moment</response>

[HttpGet("GetBots")]

public async Task<ActionResult<List<Bot>>> Get()

{

var List = await DBContext.Bots.Select(

s => new Bot

{

Id = s.Id,

FirstName = s.FirstName,

LastName = s.LastName,

Botname = s.Botname,

Password = s.Password,

EnrollmentDate = s.EnrollmentDate

}

).ToListAsync();

if (List.Count < 0)

{

return NotFound();

}

else

{

return List;

}

}

[HttpGet("GetBotById")]

public async Task<ActionResult<Bot>> GetBotById(int Id)

{

Bot Bot = await DBContext.Bots.Select(

s => new Bot

{

Id = s.Id,

FirstName = s.FirstName,

LastName = s.LastName,

Botname = s.Botname,

Password = s.Password,

EnrollmentDate = s.EnrollmentDate

})

.FirstOrDefaultAsync(s => s.Id == Id);

if (Bot == null)

{

return NotFound();

}

else

{

return Bot;

}

}

[HttpPost("InsertBot")]

public async Task<HttpStatusCode> InsertBot(Bot Bot)

{

var entity = new Bot()

{

Id = Bot.Id,

FirstName = Bot.FirstName,

LastName = Bot.LastName,

Botname = Bot.Botname,

Password = Bot.Password,

EnrollmentDate = Bot.EnrollmentDate

};

DBContext.Bots.Add(entity);

await DBContext.SaveChangesAsync();

return HttpStatusCode.Created;

}

[HttpPut("UpdateBot")]

public async Task<HttpStatusCode> UpdateBot(Bot Bot)

{

var entity = await DBContext.Bots.FirstOrDefaultAsync(s => s.Id == Bot.Id);

entity.FirstName = Bot.FirstName;

entity.LastName = Bot.LastName;

entity.Botname = Bot.Botname;

entity.Password = Bot.Password;

entity.EnrollmentDate = Bot.EnrollmentDate;

await DBContext.SaveChangesAsync();

return HttpStatusCode.OK;

}

[HttpDelete("DeleteBot/{Id}")]

public async Task<HttpStatusCode> DeleteBot(int Id)

{

var entity = new Bot()

{

Id = Id

};

DBContext.Bots.Attach(entity);

DBContext.Bots.Remove(entity);

await DBContext.SaveChangesAsync();

return HttpStatusCode.OK;

}

}

}

# Etape 5 Ajoutons un swagger.

<https://learn.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-7.0&tabs=visual-studio>

Ajouter cette ligne dans program.cs :

builder.Services.AddSwaggerGen(options =>

{

options.SwaggerDoc("v1", new OpenApiInfo { Version = "v1", Title = "ToDo API", Description = "An ASP.NET Core Web API for managing ToDo items", TermsOfService = new Uri("https://example.com/terms"), Contact = new OpenApiContact { Name = "Example Contact", Url = new Uri("https://example.com/contact") }, License = new OpenApiLicense { Name = "Example License", Url = new Uri("https://example.com/license") } });

});

Ajouter cette ligne dans le fichier Csproj :

<GenerateDocumentationFile>true</GenerateDocumentationFile>

<NoWarn>$(NoWarn);1591</NoWarn>

Une image contenant texte

Description générée automatiquement