|  |  |
| --- | --- |
| **Auteur**  Fred BERTON  Je pratique le développement logiciel depuis plus de 35 ans, depuis 13 ans chez Capgemini dans une approche en phase avec la mouvance des artisans du logiciel “Software Craftsmanship”. Ma motivation : relever le niveau du développement professionnel de logiciels par la pratique et en aidant les autres à acquérir le savoir-faire. |  |

**Titre :**

* C’est quoi un bon webservces RestFull avec C# ?

ou

* Les bonnes pratiques de développement de web service RESTFull avec C#

**Sous titre :**

C#, et particulièrement le Framework .NET offre un environnement extrêmement pratique pour développer des webservices RESTFull conforme aux spécifications OpenAPI. Encore faut-il bien l’utiliser en mettant en œuvre l’ensemble des bonnes pratiques : de documentation, de gestion des erreurs, de logs, de respect des normes.

**Corps Article**

Nous allons commencer par créer avec visual Studio 2022, un projet de Webservice. Il faut utiliser le template : « ASP.NET Core Web API » qui permet d’initialiser un projet dans Visual Studio avec l’ensemble des composants de base pour un projet de webservices conforme au standard RESTFull HTTP.

Vous pouvez faire ça avec Visual Studio, sinon pour faire geek on peut aussi le faire ne ligne de commande comme suit :

md MonWebService

cd MonWebService

dotnet new sln -n MonWebService

dotnet new webapi -n MonWebService

dotnet sln MonWebService.sln add ./MonWebService/MonWebService.csproj

MonWebService.sln

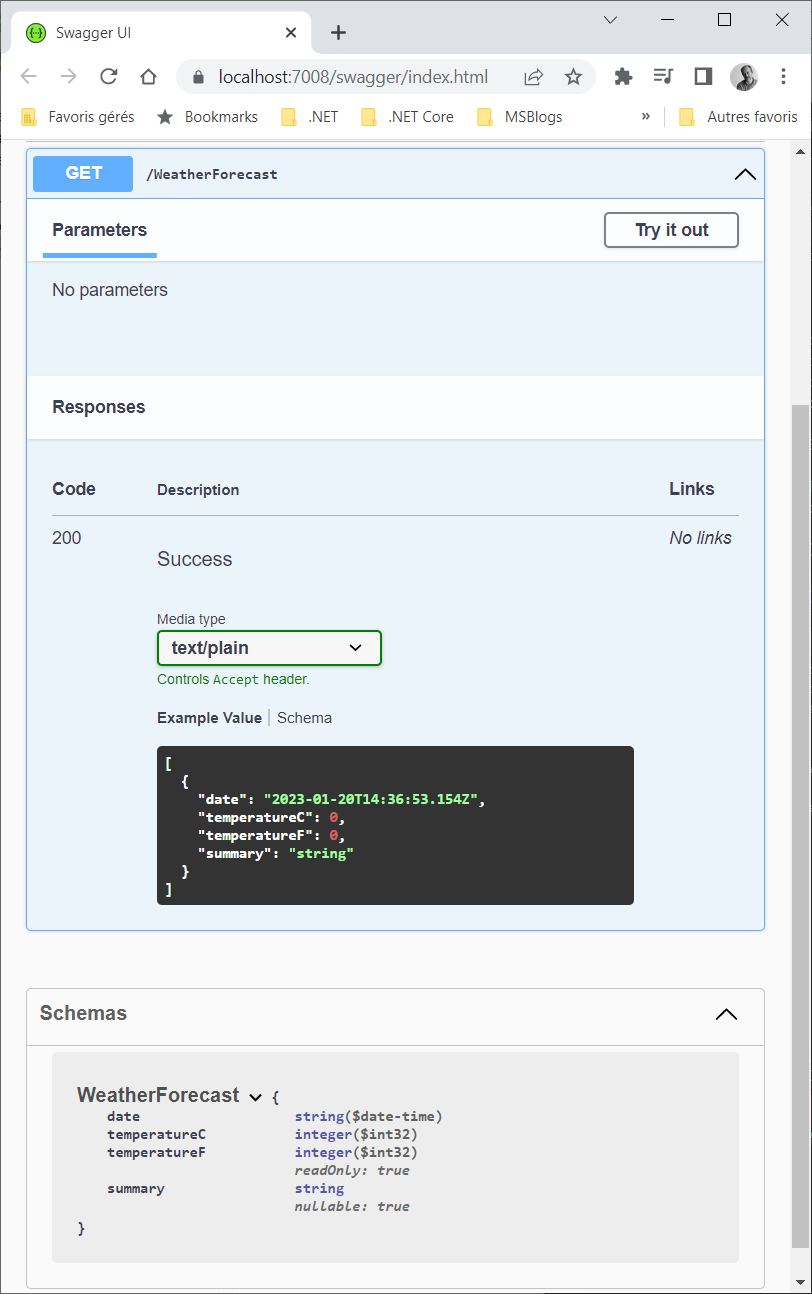
En résumé on créer un dossier qui contiendra notre solution et le dossier du projet (ici MonWebService). Puis on crée la solution, puis le projet, on ajoute le projet dans la solution. Puis on ouvre la solution dans Visual Studio.

Les Frameworks .NET nécessaires pour le développement de web services sont ajouté automatiquement : Microsoft.AspNetCore.App et Microsoft.NETCore.App.

Par défaut le package Swashbuckle.AspNetCore est installé pour générer le site d’auto-documentation Swagger.

# Premier test

Si vous lancez l’exécution, vous allez vous retrouver dans votre navigateur avec la documentation Swagger affichée, pour l’exemple de webservice WeatherForecast (Fig 1). Comme on peut le voir la page permet de consulter la documentation du service. Ici très faible car nous n’avons pas encore documenté notre service et les données qu’il utilise.

 Figure 1

Cette jolie interface « Swagger UI» permet de consulter la documentation et de tester des web services conforme à la norme OpenAPI. Pour la petite histoire, Il faut savoir qu’historiquement la documentation qui va donner son nom au standard OpenAPI s’appelée spécification Swagger (<https://swagger.io/solutions/getting-started-with-oas/>). La société smartbear développe des outils open source (ou non) permettant de consulter ou de générer du code à partir de spécification swagger (cad : OpenAPI, sauf suivre) sans avoir besoins de lire le fichier JSON du swagger (cf figure 2) plutôt illisible.

Dans .NET cette interface Swagger UI peut-être produite via deux services : Swashbuckle ou NSwag. Dans le cas de .NET 6 c’est l’implémentation Swashbuckle qui est utilisée par le template de projet (« ASP.NET Core Web API ») via le nugget Swashbuckle.AspNetCore.

Cette interface web est générée à partir de l’introspection de votre code et complétée par les descriptions de celui-ci que vous ne manquerez pas d’ajouter. L’activation (ou pas) de cette interface web est réalisée par l’invocation de la méthode : app.UseSwaggerUI() dans le code du fichier program.cs. Attention cette interface graphique n’est activée qu’en mode développement donc elle ne sera pas présente lors d’un déploiement sur un environnement cible (azure ou autre). Il vous suffit de modifier le code pour supprimer cette limitation.

La consultation du site Swagger UI est réalisé sur l’URL : <https://localhost:7008/swagger/index.html> (le port pouvant changer en fonction de votre installation).

De même Swashbuckle vous amène un service qui permet de générer automatiquement votre fichier de description de votre API de web service au format OpenAPI v3. L’activation de la génération du fichier swagger est réalisé par l’invocation de la méthode d’extension : app.UseSwagger()

La consultation du fichier swagger est réalisé sur l’url (fig 2) : <https://localhost:7008/swagger/v1/swagger.json>

Bon vous le verrez, lire en fichier openApi n’est pas une partie de plaisir. La plupart du temps il n’est d’ailleurs pas lu par un développeur, mais par un générateur de code pour produire la partie client permettant de solliciter le web service.

Figure 2

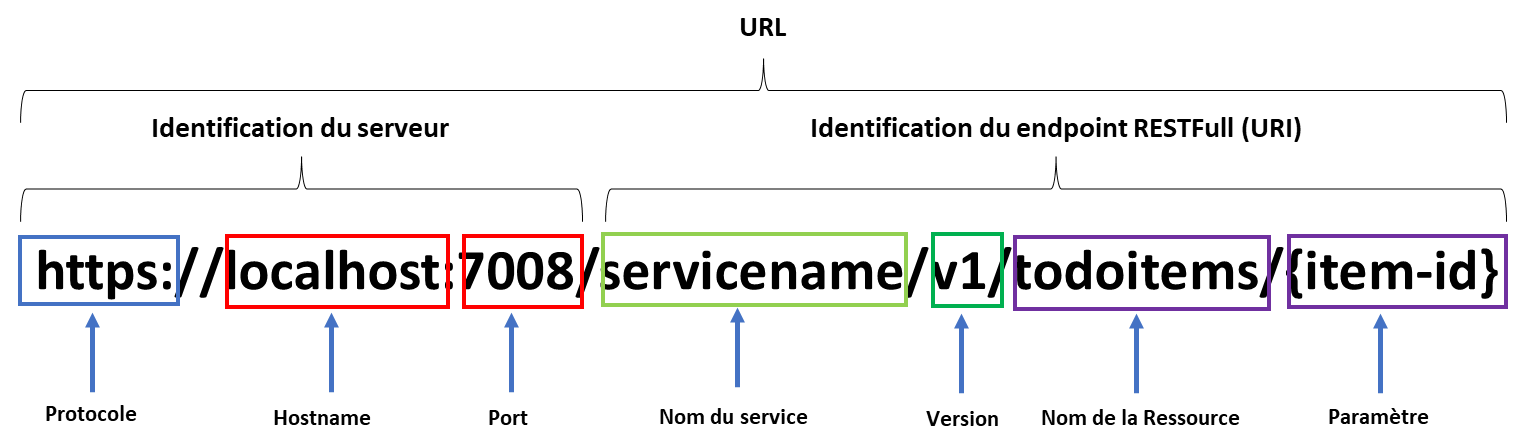
Pour que les services apportés par ces fonctions soient disponibles le middleware HTTP doit être étendue pour offrir le service de génération de Swagger. C’est l’appel de la méthode builder.Services.AddSwaggerGen() sur le bus de service HTTP qui permet de mettre en place cette extension.

Nous allons maintenant laisser là l’exemple de web service (WeatherForecast) produit par le template et produire le web service de notre exemple. Ici nous partons sur un service qui gère une liste d’actions à faire aussi connu sous le nom de Todo List.

Les actions de notre web services seront le traditionnelle CRUD (Create / Read / Update / Delete) permettant de gérer la création, la suppression, la mise à jour, et la suppression d’un TodoItem. Evidement nous complèteront celui-ci par une méthode de recherche des TodoItems en fonction de critère de recherche.

# Le standard RESTFull

Le standard RESTFull s’appuie sur les verbes ou méthodes fournis par le protocole HTTP pour définir des actions qui peuvent être réalisées sur une ressource. Les ressources sont identifiées par leur URI (Uniform Resource Identifier). Une URL d’identification du service est bien formée au sens RESTFull lorsqu’elle est conforme à la présentation de la figure 3. Nous verrons plus tard qu’il ne faut pas oublier la version…

Figure 3

Il est important de bien choisir le nom de ces ressources voici par exemple quelques bonnes idées :

http://api.exemple.fr/todolist/v101/items/12

http://api.exemple.fr/gestcom/v101/factures/fac001

Mais aussi quelques mauvaises :

http://api.exemple.fr/gestcom/CreateClient

http://api.exemple.fr/gestcom/delete-client/cli001

http://api.exemple.fr/todolist/CreateItem

http://api.exemple.fr/gestcom/v101/get-facture/fac001

Voici quelques bonnes pratiques, pour ne pas dire règle dans la création de vos noms URL et URI :

* Ne pas oublier la base une URI identifie une ressource, donc il faut données des noms de ressource, pas de verbes ou autres choses.
* Généralement dans le cas de gestion de collection de données on utilise le nom de la collection au pluriel comme nom de ressource.
* Ne pas oublier qu’une URL doit être écrite en minuscule donc facture-client est préférable à FactureClient bien que les deux soient valide.
* Une URI doit impérativement contenir une version. Vos API subiront des évolutions sinon c’est qu’elles ne sont pas utilisées. Donc il est très important de différencier la version de celle-ci. Plusieurs moyens sont possibles mais avoir la version dans l’URL est le plus simple et le plus évidant.

Pour solliciter les URI de nos ressources nous utiliserons les méthodes HTTP comme le préconise le standard :

* **POST**, pour créer une ressource. Les données sont dans le corps de la requête.
* **GET**, pour lire les données d’une ressource. Les données sont dans le corps de la requête, et la clé d’accès à l’information que l’on veut lire est passée en paramètres de l’URI. Le GET est également utilisé pour faire une interrogations sur l’ensemble des données d’une ressource.
* **PUT**, pour mettre à jour une ressource. Les données sont dans le corps de la requête, et la clé d’accès à l’information que l’on veut mettre à jour est passée en paramètres de l’URI. Attention cette mise à jour va écraser toutes les données présente dans le moyen de stockage utilisée. Il ne s’agit pas d’une mise à jour partielle, mais d’un remplacement. Dans la vraie vie beaucoup de problèmes sont liés à l’utilisation simpliste de cette méthode. Effectivement l’objet qui est passé en paramètre pour mettre à jour la données présente en base et va écraser toutes les données présentes en base. Il faut donc trouver un moyen de vérifier que la donnée que l’on va écrire n’écrase pas des modifications réalisées potentiellement par un autre appel. Généralement on règle ce problème par un numéro de version attaché à la donnée. Mais le problème se complique si la donnée contient des relations ou des listes de valeurs. D’où l’intérêt de la méthode patch.
* **DELETE**, pour supprimer une ressource. Il n’y a pas de données, et la clé d’accès à l’information a supprimé est passé en paramètres de l’URI.
* **PATCH**, pour mettre à jour partiellement une ressource.

On retrouve ici tous le nécessaire pour implémenter notre CRUD, avec en plus la notion de mise à jour partielle. Par exemple un post sur l’uri /Todolist va permettre la création d’un TodoItem dont les données sont dans le body de la requête HTTP, de même un get sur l’uri /Todolist/12 permet de lire le TodoItem identifié par 12.

Si vous avez des paramètres multiples utilisez les paramètres de requête. Par exemple : http://api.exemple.fr/gestcom/v101/factures?debut=010123&fin=310123

N’inventez pas un format de stockage pour vos données de body, généralement on utilise du JSON (JavaScript Object Notation) même si cette notation a plein de défaut. Elle reste la référence commune pour échanger des données. Il est aussi possible de passer du XML, verbeux, mais qui offre l’avantage de pouvoir être contrôlé et validé par un schéma XMLSchema.

# RESTFull et HATEOAS (Hypermedia as the Engine of Application State)

HATEOAS est une recommandation du standard RESTFull visant à renvoyer des réponses qui indiquent les liens de navigation afférant à une ressource.

Par exemple dans une réponse de type GET, il faut indiquer les liens qui permettent par exemple : la suppression, l’édition, et la recherches. L’avantage de cette approche c’est que l’application utilisatrice de votre web service va pouvoir gérer automatiquement son interface graphique pour faire le routage vers ces fonctions.

Malheureusement le .Net ne propose pas de solution pour mettre en place le standard HATEOS. Je n’en traiterais donc pas ici, mais vous pourrez trouver une solution élégante dans l’article de Vladimir PECANAC sur code-maze.com (HATEOS1).

# Création contrôleur de ressource du webservice

Je ne passerais pas de trop sur ce sujet car il est largement automatisé, documenté. Je vais plutôt insister sur tout ce qui nécessaire autour et dans la manière de le faire.

Dans .Net la logique d’implémentation d’un web service est qu’une classe est créé pour implémenter pour gérer chaque ressource exposée par le web service. Si vous avez un web service qui gérer une seule ressource par exemple des todo items vous avez donc un seul contrôler à définir. Si maintenant vous souhaitez gérer des listes de todo, alors vous allez devoir implémentée plusieurs contrôleurs dans votre web service car vous avez plusieurs ressources à gérer : d’un côté la gestion des listes de todo (liste de cours, liste de jardinage, list de bugs, etc…), et de l’autre les items de chaque todo liste.

La définition d’un contrôleur de ressource est réalisée par une déclaration comme suit.

[ApiController]

[Route("api/v{version:apiVersion}/[controller]")]

public class TodoListController : ControllerBase

{

// bla bla bla

}

Une classe contrôleur de ressource qui portera le nom de la ressource et sera postfixé par « Controller ». Cette classe doit héritée de la classe de base des contrôleurs de ressources de web service ControllerBase fournit par l’assembly Microsoft.AspNetCore.Mvc.

Elle doit également être décorée par l’attribut [ApiController].

Pour que le contrôleur de votre ressource soit accessible via une requête HTTP, il faut spécifier la route qui permet de l’atteindre depuis la racine du site web de déploiement. C’est l’objet de l’attribut [Route("api/v{version:apiVersion}/[controller]")]. Dans notre exemple nous gérerons une route avec un numéro de version, nous y reviendrons plus tard. Ce qui est important ici c’est que nous souhaitons que notre route commence par « api » ce qui est une bonne pratique pour différencier l’usage de nos URL, puis un numéro de version, et le nom de la ressource. Comme le tag [controller] n’indique pas, le nom de la ressource est défini par le tag. Le Framework comprend qu’il doit remplacer le tag par le nom de la classe de notre contrôleur auquel, il aura pris soins de supprimer le mot « Controller » s’il est présent. La route produite est donc : **api/v1/todolist**

# Implémentation des actions sur une ressource

Notre ressource étant déclarée, nous allons maintenant définir les actions qu’elle peut supporter. Actions qui correspondent au verbes HTTP définit par l’attributs : [HttpGet()], [HttpPost()], [HttpPut()], [HttpDelete()], [HttpPatch()] qui doit être placé devant le nom de la méthode.

Dans l’exemple ci-dessous nous faisons un GET pour lire une donnée, il nous faudra donc un paramètre qui est l’id de l’enregistrement à lire. Ce paramètre qui doit être ajouté à la route et peut être déclarer soit dans le verbe de l’action [HttpGet("{id}")], soit via l’attribut [Route("{id}")] qui permet de spécifier les informations pour compléter la route sur la ressource. Notre URL d’accès en lecture à notre ressource est : **api/v1.1/todolist/12**. La méthode GetTodoItem qui prend en charge l’exécution du code doit définir un paramètre avec le même nom pour récupérer la valeur de notre id.

[HttpGet("{id}")]

[Produces("application/json")]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

[ProducesResponseType(StatusCodes.Status200OK)]

public ActionResult<TodoItem> GetTodoItem(int id)

{

if (id <= 0)

return BadRequest("Id doit être supérieur à 0.");

TodoItem? data = Datas.FirstOrDefault(item => item.ID == id);

if (data == null)

return NotFound($"Non trouvé avec l'ID = {id}");

return Ok(data);

}

L’attribut [Produces("application/json")] permet de forcer un résultat au format JSON, ce qui est le cas par défaut. Masi il peut être utiliser pour indiquer un autre format.

Dans certains cas nous pouvons avoir besoins de plus de paramètres qui soit récupérés dans l’URL. Par exemple pour prendre en compte les URL suivant :

**api/v1.1/todolist?pageindex=12**

**api/v1.1/todolist?pageSize=10&pageindex=2**

Dans ce cas on utilisera l’attribut [FromQuery] en préfixe devant le nom des attributs qui doivent être passé à notre méthode. Le nom de l’attribut sera aussi attendu dans l’URL de sollicitation. On voie également que l’on peut passer des valeurs par défaut.

[HttpGet()]

public ActionResult<IEnumerable<TodoItem>> GetTodoItems([FromQuery]int pageSize = 10,[FromQuery] int pageIndex = 0)

Pour la création d’une nouvelle ressource on utilise un [HttpPost()] qui prendra ces données dans le body du message HTTP via l’attribut [FromBody] déclarer dans les paramètres de la fonction. Un mapping sera fait entre le contenu du body qui sera en JSON et la classe déclarer par le FromBody

public ActionResult<TodoItem> CreateTodoItem([FromBody] CreateOrUpdateTodoItem dataToAdd)

Pour la Mise à jour d’une ressource c’est le [HttpPut()] qui doit être utilisé, ici nous avons besoins de compléter les données utilisées pour la mise à jour par l’id de la données à mettre à jour d’où la combinaison [HttpPut("{id}")] et [FromBody].

[HttpPut("{id}")]

public ActionResult UpdateTodoItem(int id, [FromBody] UpdateTodoItem dataToUpdate)

J’en vois certains qui aurait utilisé une propriété passé par le mapping de donnée du body et du coup récupéré dans la classe UpdateTodoItem. Mais cette approche introduit une incohérence dans le contenu de la cette classe, effectivement nous ne pourrons pas mettre à jour le record numéro. La classe doit contenir uniquement les données qui doivent être modifiées et seulement ça. De même une classe passé en paramètre d’un Create ne peut pas contenir un id comme attribut, sauf si c’est vraiment celui-ci qui est utilisé pour identifier la donnée. Or généralement les ids sont générés par le service en charge du stockage de la donnée.

# Déclarer et respecter les codes erreurs HTTP

Les fonctions C# qui prennent en charge l’exécution des méthodes HTTP, doivent renvoyer un code de statuts résultat de l’exécution de l’appel, tel que défini par le standard HTTP. (<http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml>). D’une façon générale les codes erreurs sont des nombres >= 100 et <600 et sont regroupés par thématique, chaque thématique étant séparées de la suivante par 100.

* **1xx :** information générale.
* **2XX**: indique un succès dans la réalisation d’une opération.
* **3XX :** Indique une redirection nécessaire pour terminer la requête.
* **4XX**: Indique une erreur issue des informations fournis par le client.
* **5XX :** Indique une erreur produite par le serveur suite à une requête a priori valide.

Avec .NET vous avez la possibilité d’utiliser ces valeurs à partir de l’énumération définit dans l’assembly : Microsoft.AspNetCore.Http qui contient une classe statique StatusCodes voici par exemple quelques valeurs bien utile :

StatusCodes.Status200OK

StatusCodes.Status400BadRequest

StatusCodes.Status404NotFound

Pour déclarer les codes erreurs possible sur un web service, et documenter ceux-ci au niveau du swagger. Il faut d’utiliser l’attribut *ProducesResponseType* avec le StatusCodes des erreurs qui peuvent être générées en résultat de l’exécution du code.

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

Dans le code de la méthode, pour renvoyer le statut d’exécution il faut utiliser une des fonctions prédéfinies sur la classe de base du contrôleur ControllerBase. Ces fonctions renvoient une instance de classe dérivant de StatusCodeResult (elle-même dérivant de ActionResult) si la valeur de retour ne doit pas renvoyer de données ; dans le cas contraire, il s’agit d’une sous classe de ObjectResult. Par exemple la fonction Ok() renvois une instance de OkObjectResult et Ok(object?) renvois une instance de la classe OkObjectResult dont le code ci-dessous illustre le principe de fonctionnement.

public class OkObjectResult : ObjectResult

{

private const int DefaultStatusCode = StatusCodes.Status200OK;

public OkObjectResult(object? value)

: base(value)

{

StatusCode = DefaultStatusCode;

}

}

Vous trouverez toutes ces fonctions dans la classe ControllerBase, mais voici les principales :

* [BadRequest](https://learn.microsoft.com/fr-fr/dotnet/api/microsoft.aspnetcore.mvc.badrequestresult?view=aspnetcore-7.0), StatusCodes.Status400BadRequest
* [NoContentResult](https://learn.microsoft.com/fr-fr/dotnet/api/microsoft.aspnetcore.mvc.nocontentresult?view=aspnetcore-7.0), StatusCodes. Status204NoContent
* [NotFound](https://learn.microsoft.com/fr-fr/dotnet/api/microsoft.aspnetcore.mvc.notfoundresult?view=aspnetcore-7.0), StatusCodes.Status404NotFound
* Conflict StatusCodes Status409Conflict
* [Ok](https://learn.microsoft.com/fr-fr/dotnet/api/microsoft.aspnetcore.mvc.okresult?view=aspnetcore-7.0), StatusCodes.Status200OK

Dans l’exemple de code GetTodoItems() ci-dessous nous implémentons une méthode Get qui va renvoyer les données lues dans notre source de données, ici une liste en mémoire, et qui déclare trois types de statuts de retour classique pour une opérations de lecture de données.

[HttpGet()]

[Produces("application/json")]

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

public ActionResult<IEnumerable<TodoItem>> GetTodoItems([FromQuery]int pageSize = 10,[FromQuery] int pageIndex = 0)

{

      if (Datas == null)

        return NotFound($"Pas de données dans la liste");

      List<TodoItem> datas = Datas.Skip(pageSize \* pageIndex)

                                  .Take(pageSize)

                                  .ToList();

      return Ok(datas);

}

# Le contrôle des données d’entrée avec les data annotations

Pour transmettre des données vers ou depuis un web service. Le plus simple est d’utilisé une classe C# correspondant à la structure des données. C# offre la possibilité d’ajouter des attributs d’annotation des données. Au-delà de documenté les valeurs acceptables par votre classe. Ce mécanisme va permettre à .Net de contrôler que les données fournit sont valides. Ce travail de validation est réalisé par le contrôleur avant l’invocation de la méthode qui reçois le paramètre. C’est donc la classe de base ControllerBase qui va faire le job avant d’invoquer votre méthode, et qui d’ailleurs n’invoquera pas votre méthode si la validation n’est pas bonne. A la place le contrôleur de base généré lui-même une erreur.

Par exemple avec une classe Personne comme suit :

public class PersonCreateDTO

{

[Required]

public string Nom { get; set; }

[Required(ErrorMessage ="Le prénom est obligatoire")]

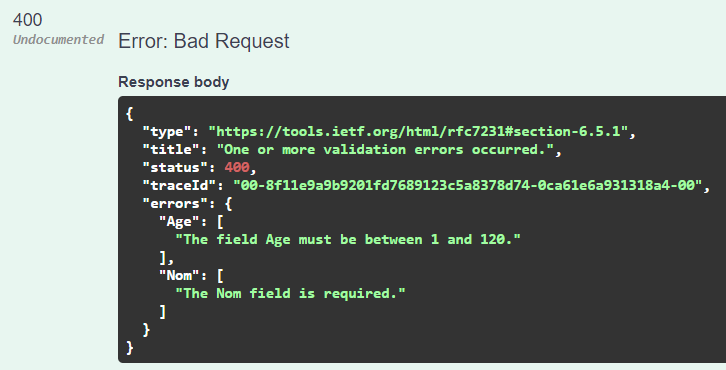
public string Prenom { get; set; }

[Range(1, 120)]

public int Age { get; set; }

}

On demande à avoir obligatoirement les propriété Nom et Prenom avec des valeurs et l’Age doit être compris entre 1 et 120 ans. Si on appelle le web service avec les valeurs suivantes : { "nom": null, "prenom": "string", "age": 250}. Nous obtiendrons directement une erreur 400 Bad Request.



Ce qui est vraiment parfait, avec finalement pas beaucoup de travail à faire.

C’est l’assembly System.ComponentModel.DataAnnotations qui contient les attributs disponible [DA1,DA2] . Ils vont permettre d’indiquer qu’une valeur est obligatoire pour un attribut, ou par exemple de fixer une valeur par défaut, une valeur min et max. Ce mode de fonction est déjà connu si vous avez fait de l’ASP.NET MVC. Les valeurs les plus classique tant

* **Required**, pour un champ obligatoire
* **Range,** pour indiquer un intervalle de valeur autorisé.
* **MinLength,** pour donner une taille minimum pour une chaine
* **MaxLength**, pour donner une taille maximum pour une chaine
* **Compare,** pour comparer la valeur de 2 propriétés. Par exemple pour valider un email saisi avec eMailConfirmation saisie en confirmation.
* **RegularExpression**, pour valider la donnée avec un expression régulière

Vous pouvez bien sûr ajouter votre propre logique de contrôle dans la méthode en utilisant la propriété ModelState, comme dans l’exemple ci-dessous.

[HttpPost()]

[Produces("application/json")]

[ProducesResponseType(StatusCodes.Status200OK)]

public ActionResult<PersonCreateDTO> CreatePerson([FromBody]PersonCreateDTO createDTO)

{

if (ModelState.IsValid == true)

{

logger.LogInformation("Mode state Valid");

ModelState.AddModelError("Nom", "Ici un erreur forcée ");

return BadRequest(ModelState);

}

else

{

logger.LogInformation("Mode state NOT VALID");

}

return Ok();

}

# Journaliser le fonctionnement d’un web service

Faire des logs dans une applications est fondamentale, je ne parle pas de logs de debug, mais de logs qui vont permettre aux équipes de supervision ou de maintenance de comprendre le fonctionnement et particulièrement les erreurs qui se produisent dans votre logiciel. Ce besoin est encore plus important dans le développement de webservice du fait que ceux-ci sont utilisées par des applications et non directement par un utilisateur qui pourrait consulter une erreur sur l’écran et la transmettre à un service de maintenance. Dans le cas des webservices pour tracer un problème, ou le bon fonctionnement de vos services, le seul moyen est de lire les logs qui vous produisez.

Il est important de loguer les informations relatives au fonctionnement générale d’un service : Le démarrage, l’arrêt, ou la présence d’une exception gérée ou non.

D’autre part les méthodes exposés par le web service doivent également loguer les informations sur leurs exécutions. Dans ce cas, ce sont les conditions liées à la demande d’exécution de la méthode, et le résultat qui vont nous intéresser. Lors du logue de la demande d’exécution il faut tracer le nom de la fonction, et les paramètres fournit. Ce qui permettra en cas de problème d’identifier les paramètres d’exécution qui ont générés le problème. De même si l’exécution c’est bien passé, il faut tracer le fait que l’exécution c’est déroulé sans problème.

Pour utiliser le service de log, il suffit au niveau du constructeur de la classe du contrôleur du webservice de mettre en place l’injection du service de log. Comme l’illustre le code ci-dessous.

// Service de log injecté par le constructeur

private readonly ILogger logger;

public TodoListController(ILogger<TodoListController> \_logger)

{

logger = \_logger;

logger.LogInformation("CTOR");

Pour faire dans log durant l’initialisation du web service cela se passe dans le fichier *Program.cs* qui fournit au niveau de la classe *WebApplication* une propriété Logger.

var app = builder.Build();

app.Logger.LogInformation("Démarrage du Web services");

La mise en place du service de log dans une application .Net est très simple, mais de nombreuse option de configuration sont possibles. Ce qui sort du cadre de cet article. Je vous invite à prendre connaissance de la doc MS qui donne toutes les informations nécessaires ici [LOG1]

# Versionner son API

La durée de vie d’une API est parfois très longtemps (même très très longtemps), mais les applications cliente aussi.

Il faut donc garantir le service dans la durée, et permettre aux applications client de continuées a utilisé le service même si l’API évolue. Il est évidant que vos API subiront des évolutions sinon c’est qu’elles ne sont pas utilisées.   
Donc il est très important de différencier la version de celle-ci. Généralement les évolutions sur un URI existant concerne les informations qui seront véhiculé dans le body.

Le fait générateur [GV1, GV2] d’un changement de version dans votre API est une modification avec rupture (breaking change). C’est-à-dire un changement qui va avoir un impact sur le code des applications cliente par exemple :

* Supprimer une API ;
* Renommer une API ;
* Supprimer des paramètres d'API ;
* Renommer des paramètres d'API ;
* Modification (ajout, suppression, renommage de champs) dans les données de réponse JSON ;
* Supprimer des paramètres d'API ;
* Renommer des paramètres d'API ;
* Changements de comportement pour une API existante ;
* Modifications des codes d'erreur.

Vous devez/pouvez complétés cette liste avec d’autres raisons, ou même la réduire. Mais dans tous les cas les changements ou évolutions de votre API ne doivent pas changer son comportement d’un point de vue du client. Si c’est tels est le cas, alors vous devez changer de version.

En conséquence vous devez aussi définir une politique de changement de version. Lorsque votre API change de version : est-ce que vous gardez disponible les versions précédentes ? si oui combien de temps ? Cette notion de gestion du cycle de vie des versions doit être documentée et largement publié auprès des utilisateurs de votre API.

Généralement la solution retenue est d’utilisé une gestion de version codé dans l’URL du chemin de votre API.

http://api.exemple.fr/todolist/v11/items/12

http://api.exemple.fr/gestcom/v12/factures/fac001

La gestion du versionning d’API est apporté par le package nuget *Microsoft.AspNetCore.Mvc.Versioning* qui faudra référencé dans votre projet. La forme la plus courante utilisé pour spécifier la version d’une API repose sur l’utilisation des attributs *ApiVersion* qui permet d’indique la version supportée par un contrôleur

[ApiVersion("1.0")]

L’attribut Route va nous permettre de spécifier le format de la route avec l’encodage de la version comme ci-dessous.

[Route( "api/v{version:apiVersion}/[controller]" )]

Cette route permet de générer des url comme :

api/v1.0/MonService

api/v1.11/MonService

L’avantage, c’est qui si vous n’indiquez rien de particulier, les méthodes HTTP que vous exposez supportent les deux versions. Pour restreindre ou différencier le support d’une méthode HTTP vous pouvez utiliser l’attribut *MapToApiVersion* pour spécifier le support d’une version particulière comme le montre le code suivant.

[ApiVersion( "1.0" )]

[ApiVersion( "2.0" )]

[Route( "api/v{version:apiVersion}/[controller]" )]

[ApiController]

**public** **class** MonServiceController : ControllerBase

{

    [HttpGet]

**public** **string** Get() { return "MonService GET en version 1.0" ; }

    [HttpGet, MapToApiVersion( "2.0" )]

**public** **string** Get() { return "MonService GET en version 2.0" ; }

}

Attention si vous utilisez *SwaggerGen* pour la génération du fichier swagger. Vous obtiendriez une exception en raison de la duplication de la méthode Get. Il faut modifier la configuration de votre appel à la méthode *AddSwaggerGen()* comme suit pour éviter ce problème.

c.ResolveConflictingActions(apiDescriptions => apiDescriptions.First());

Si les évolutions réalisées entre les versions sont très importantes, il est généralement préférable de créer un nouveau contrôleur qui sera routé sur la même URL pour la version spécifiée comme ci-dessous. Attention comme le nom du contrôleur change, il faut écrire explicitement l’URI pour qu’elle corresponde avec le service dans les versions précédente. Ici j’ai indiqué *MonServiceVersion3Controller* comme nom de contrôleuret j’ai modifié la règle de routage en indiquant */MonService.*

[ApiVersion( "**3.0**" )]

[Route( "api/v{version:apiVersion}/**MonService**" )]

**public** **class** MonServiceVersion3Controller: ControllerBase {

    [HttpGet

**public** **string** Get() => " MonService GET en version 2.0 !";

}

Il faut savoir que le moteur de routage utilise des ‘a priori’ par exemple si votre contrôleur s’appelle MonService3Controller la subtilité ici est que le moteur de routage fait le choix de supprimer les chiffres qui précédent le mots clef contrôleur. En conséquence les contrôleurs : *MonServiceController* et *MonService3Controller* correspondent tous les deux à une route *MonService.*

A partir du moment où vous avez plusieurs versions de votre API, il est important de fournir ces informations aux consommateurs de votre API. C’est justement le but de la méthode *AddApiVersioning* et de l’attribut *ReportApiVersions* dans ces options de configuration.

Dans l’exemple ci-dessous, nous configurons à la fois une version par défaut ce qui évite le recourt à l’attribut *[ApiVersion]* sur le contrôleur ou les fonctions, et nous activons la fonctionnalité de reporting de version.

builder.Services.AddApiVersioning(optVersion =>

{

optVersion.DefaultApiVersion = new ApiVersion(1, 2);

optVersion.ReportApiVersions = true;

});

Pour illustre l’intérêt de *ReportApiVersions*, nous allons utiliser curl pour invoquer une méthode de mon webservice. Intentionnellement, j’ai indiqué dans mon url que je souhaite utiliser la version v1.1 qui n’ait justement pas implémenté.

**curl -X 'GET' \**

**'http://localhost:5279/api/v1.1/TodoList?pageSize=10&pageIndex=0' \**

**-H 'accept: application/json'**

L’effet immédiat, est de provoquer une exception, car mon API est définie pour la v1.2, et uniquement pour celle-là.

**{**

**"error": {**

**"code": "UnsupportedApiVersion",**

**"message": "The HTTP resource that matches the request URI 'http://localhost:5279/api/v1.1/TodoList' does not support the API version '1.1'.",**

**"innerError": null**

**}**

**}**

Si l’on regarde la réponse header HTTP de ma requête on constate l’effet de *ReportApiVersions* qui affiche les versions supportées par mon API, ici la v1.2. S’il y a plusieurs versions supportées, elles sont affichées séparées par des virgules.

api-supported-versions: 1.2

content-type: application/json; charset=utf-8

date: Fri,21 Apr 2023 14:52:31 GMT

server: Kestrel

transfer-encoding: chunked

# Générer la documentation swagger et tester le web service

OpenAPI est une spécification qui permet de documentée une API de web service. Elle est produite dans un fichier JSON (qui respecte la grammaire JSON spécifiée par l’OpenAPI) pour documenter votre API. Ce fichier est par défaut appelé openapi.json

SwaggerUI est un service qui implémente une interface web pour consulter la documentation d’une API, il permet même d’invoquer l’API pour évaluer/tester son fonctionnement.

L’activation de Swagger UI sur votre API doit être réalisé dans le middleware en ajoutant le service de génération Swagger via la méthode d’extension AddSwaggerGen().

Builder.Services.AddSwaggerGen() ;

Puis il faut enregistré le middleware de génération de l’OpenAPI sur le pipeline HTTP via la commande UseSwagger() et activer l’interface utilisateur de consultation de la documentation et de test des services avec UseSwaggerUI().

app.UseSwagger();

app.UseSwaggerUI();

Maintenant que nous avons activé la génération du Swagger et l’activation de l’interface SwaggerUI pour consulter la documentation et tester le web Service. Comment faire pour documenter notre API ?

Vous savez déjà qu’avec C#, vous devez documenter votre code avec une syntaxe XML spécifique [Doc1] qui permet de générer la documentation de votre code. Voici un exemple classique de documentation d’une méthode en C#.

/// <summary>

/// Lecture d'un Todo Item

/// Retourne un todo item à partir de son identifiant ID

/// </summary>

/// <remarks>ici une remarque</remarks>

/// <param name="id">id du todo item doit être supérieur à 0.</param>

/// <response code="200">OK, le todo item est retourné.</response>

/// <response code="400">Mauvaise requete, id non invalide.</response>

/// <response code="404">Erreur, item non trouvé avec l'id spécifié.</response>

[HttpGet()]

[Route("{id}")]

[Produces("application/json")]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

[ProducesResponseType(StatusCodes.Status200OK)]

public ActionResult<TodoItem> GetTodoItem(int id)

Swagger à la bonté de reprendre vos efforts pour générer la documentation de votre web service. Ce qui vous évitera peut-être de rédiger un document de références pour votre API 😉. Il est toutefois est d’avoir en tête quelques petite subtilité dans la transformation de la documentation C# en documentation Swagger.

Le contenu de la balise <summary> se retrouve dans la balise "summary" du swagger, jusqu’à là rien d’étonnant. En revanche, la balise <remarks> se retrouve dans la balise « description » du swagger. Il est donc important de prendre cette subtilité en considération, pour avoir une documentation lisible. Si vous avez beaucoup de choses à raconter, c’est bien dans le tag <remarks> qu’il faut le mettre et non dans <summary> comme c’est généralement l’habitude en C#.

Ce qui donne en swagger la documentation suivante :

"/api/v{version}/TodoList/{id}": {

"get": {

"tags": [ "TodoList" ],

"summary":   
 "Lecture d'un Todo Item\r\nRetourne un todo item a partir de son identifiant ID",

"description": "ici une remarque",

"parameters": [

{

"name": "id",

"in": "path",

"description": "id du todo item doit être supérieur à 0.",

"required": true,

"schema": {

"type": "integer",

"format": "int32"

} },

{

"name": "version",

"in": "path",

"required": true,

"schema": {

"type": "string"

} }],

"responses": {

"400": {

"description": "Mauvaise requete, id non invalide.",

"content": {

"application/json": {

"schema": {

"$ref": "#/components/schemas/ProblemDetails"

} } } },

"404": {

"description": "Erreur, le todo item n'est pas trouvé à partir de l'id spécifié.",

"content": {

"application/json": {

"schema": {

"$ref": "#/components/schemas/ProblemDetails"

} } } },

"200": {

"description": "OK, le todo item est retourné.",

"content": {

"application/json": {

"schema": {

"$ref": "#/components/schemas/TodoItem"

} } } } } },

Et dans swagger UI



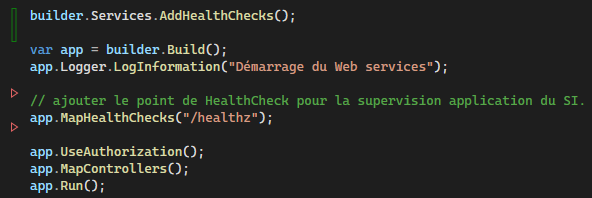
# Avoir un point de HealthCheck

Les applications modernes sont de plus en plus décomposées en une partie front qui va constituer l’interface graphique mis à disposition des utilisateurs (une application mobile par exemple), et un ensemble de services back office qui sont agrégés par celle-ci pour offrir un service métiers. Dans ce type d’architecture l’application front peut semblait totalement opérationnelle car le site web fonctionne : les pages s’affichent, la navigation fonctionne, mais il n’y a pas de données dans certaines zones de l’interface, ou certaines pages ne fonctionne pas. Il est donc important de donner un moyen à l’équipe de supervision d’analyser l’état de fonctionnement de l’ensemble des composants nécessaires au fonctionnement d’une application. Ces composants peuvent faire partie de l’application elle-même comme : le site web du front, ou les webservices. Mais il peut aussi s’agir de service externe a l’application comme des webservices tiers, ou des webservices d’une autre application.

Lorsque l’on développe un webservice, il faut donc trouver un moyen pour indiquer si notre webservice est opérationnel ou pas. Ce qui permettra à l’application cliente de prendre les décisions qui s’impose comme : indiquer le problème à l’utilisateur, ou choisir un autre webservice pour obtenir des informations identiques à celle fournis par le webservice non opérationnel. D’un autre côté l’équipe de supervision pourra utiliser des outils de supervision applicative pour détecter le non-fonctionnement d’un web service et mettre en action un plan de remédiation.

Le .Net Framework fournit un service dit « HealthCheck » qui permet très simplement de fournir un point d’entrée pour indiquer l’état de santé d’une application. Le service de HealthCheck est rendu disponible par l’assembly : Microsoft.Extensions.Diagnostics.HealthChecks qui est elle-même apporté par le framework Microsoft.AspNetCore.App.

Pour activé le HealthCheck dans votre webservice, il faut ajouter ce service dans le middleware HTTP via la méthode d’extension AddHealthChecks qui doit être invoquée sur le bus de service HTTP comme suit : builder.Services.AddHealthChecks(). Puis il faut indiquer le point de contrôle qui sera ajouté à l’URL de votre site pour solliciter le point de contrôle. C’est l’objet de la méthode MapHealthChecks qui doit être invoqué la variable d’instance de la WebApplication comme ceci : app.MapHealthChecks("/healthz").



Généralement le point de contrôle s‘appelle healthz et peut être invoqué avec url de check : <http://localhost:5279/healthz>

L’invocation de ce point de contrôle va principalement renvoyer le StatusCodes.Status200OK, et le body contiendra une des valeurs de l’énumération HealthStatus, dont les valeurs possibles sont :

* [HealthStatus.Healthy](https://learn.microsoft.com/fr-fr/dotnet/api/microsoft.extensions.diagnostics.healthchecks.healthstatus#microsoft-extensions-diagnostics-healthchecks-healthstatus-healthy), qui indique que le composant et ces éventuelles sous composant sont opérationnelle.
* [HealthStatus.Degraded](https://learn.microsoft.com/fr-fr/dotnet/api/microsoft.extensions.diagnostics.healthchecks.healthstatus#microsoft-extensions-diagnostics-healthchecks-healthstatus-degraded), lors le fonctionnement est dégradé.
* [HealthStatus.Unhealthy](https://learn.microsoft.com/fr-fr/dotnet/api/microsoft.extensions.diagnostics.healthchecks.healthstatus#microsoft-extensions-diagnostics-healthchecks-healthstatus-unhealthy), lorsque le service n’est pas opérationnel probablement à la suite d’une exception.

Il est possible d’ajouter des contrôles d’intégrité sur des services nécessaire au web service par exemple, si une base de données SQL Server est utilisée on peut ajouter une demande de contrôle de celle-ci.

string connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddHealthChecks()  
 .AddSqlServer(connectionString);

Vous trouverez sur le net de nombreux nugget package en faisant une recherche depuis le gestionnaire de package nuget avec « AspNetCore.HealthChecks. » ; je ne peux pas vous donner la liste complète bien trop longue. Mais dans les principaux services sont disponibles comme par exemple : Oracle, Cosmosdb, RabbitMQ, AzureServiceBus, Sqlite, SqlServer, Redis, etc.

Il est également possible de développer son propre contrôleur HealthCheck avec une classe qui implémentera de l’interface IHealthCheck, mais je garde ceci pour un autre article...

# Avoir un point de HealthCheckUI

url de check avec UI :

<http://localhost:5279/healthchecks-ui#/healthchecks>

<https://github.com/Xabaril/AspNetCore.Diagnostics.HealthChecks#HealthCheckUI>

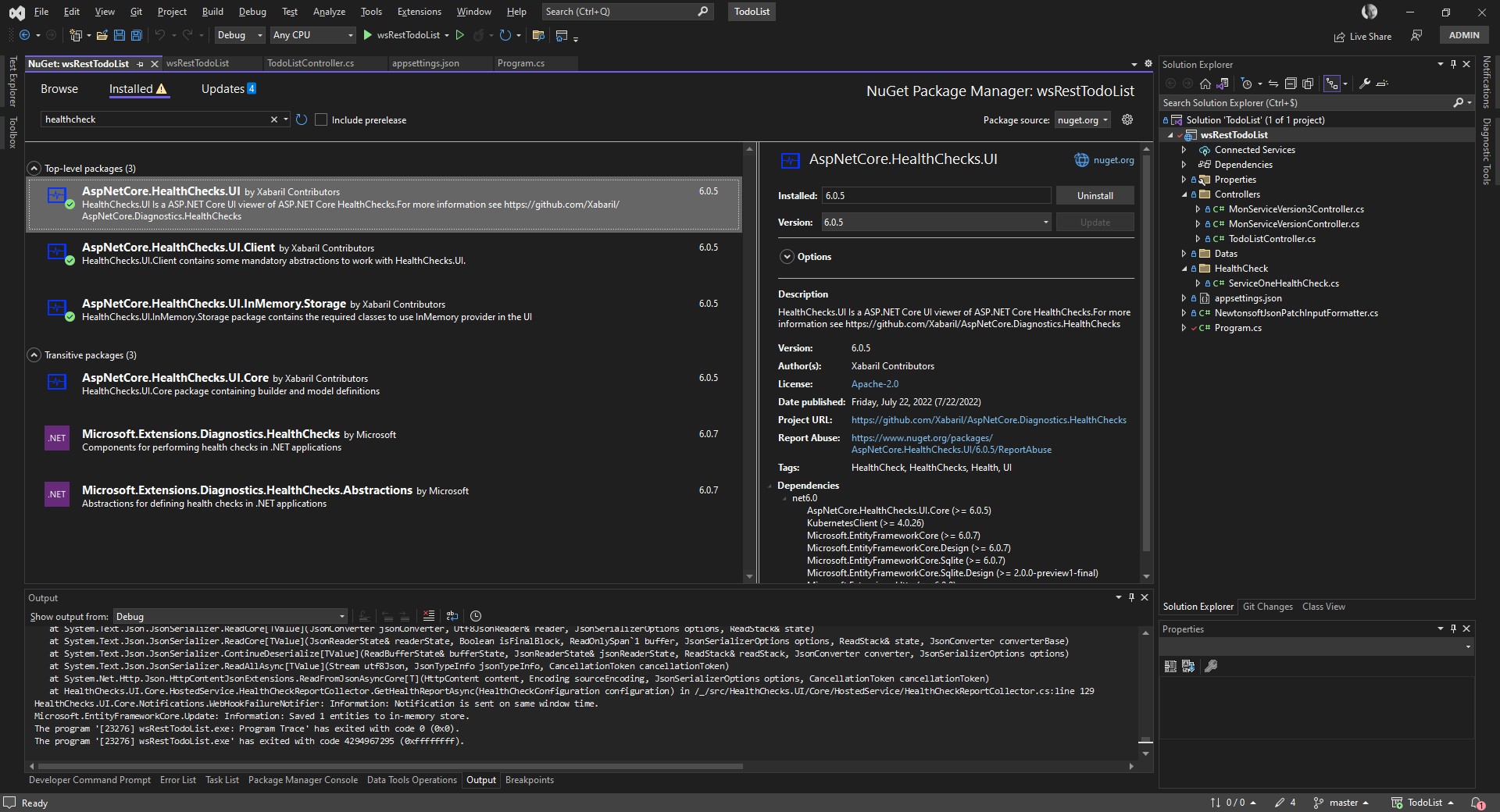
Microsoft.Extensions.Diagnostics.HealthChecks

nuget necessaure :

* AspNetCore.HealthChecks.UI
* AspNetCore.HealthChecks.UI.Client
* AspNetCore.HealthChecks.UI.InMemory.Storage

<https://rmauro.dev/adding-health-checks-ui/>

<https://balta.io/blog/aspnet-health-check>



# Références

|  |  |  |
| --- | --- | --- |
| Ref | Sujet | Référence |
| HATEOS1 | Implémentation du standard Resfull HATEOS en C#. | <https://code-maze.com/hateoas-aspnet-core-web-api/> |

Documentation du code

|  |  |  |
| --- | --- | --- |
| Ref | Sujet | Référence |
| Doc1 | Syntaxe de documentation des commentaires dans C# | <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/documentation-comments> |

|  |  |  |
| --- | --- | --- |
| Ref | Sujet | Référence |
| 1 | Information sur les valeurs de retour d’une API WEB | <https://medium.com/awesome-net/web-api-return-types-in-net-94715415ae88> |
|  | Utilisation de swagger | <https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger> |
|  | Créer la documentation des Web API ASP.NET Core avec Swagger | <https://rdonfack.developpez.com/tutoriels/documenter-web-api-aspnet-core-swagger/> |
|  | Api controller documentation | <https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-6.0#apicontroller-attribute> |

Gestion des versions

|  |  |  |
| --- | --- | --- |
| GV1 | Gestion de version sur les Web API, Scott Hanselman | <http://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx> |
| GV2 | Identification des causes de changement de version dans OData | <https://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part1-protocol/odata-v4.0-errata02-os-part1-protocol-complete.html#_Toc406398209> |

<https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md#12-versioning>

<https://semver.org/lang/fr/>

Data Annotation

|  |  |  |
| --- | --- | --- |
| DA1 | Aide sur la validation des données en se basant sur les data annotations | <https://learn.microsoft.com/fr-fr/aspnet/web-api/overview/formats-and-model-binding/model-validation-in-aspnet-web-api?source=recommendations> |
| DA2 | Liste des Data Annotation | <https://learn.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations?view=net-7.0> |

Logs

|  |  |  |
| --- | --- | --- |
| LOG1 | La journalisation dans .Net core et ASP.NET Core | https://learn.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-7.0 |
| DA2 | Liste des Data Annotation | <https://learn.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations?view=net-7.0> |

|  |  |  |
| --- | --- | --- |
|  | Liste des codes d’erreurs HttpStatusCode | https://learn.microsoft.com/fr-fr/dotnet/api/system.net.httpstatuscode?view=net-6.0 |
|  | Validation des données | <https://code-maze.com/aspnetcore-modelstate-validation-web-api/> |
|  | Analyseur de valeur de retour | <https://learn.microsoft.com/en-us/aspnet/core/web-api/advanced/analyzers?view=aspnetcore-6.0> |
|  | Actions asynchrone | <https://learn.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-6.0> |
|  | Activer le support update partiel d'objet avec JSonPatch | <https://learn.microsoft.com/fr-fr/aspnet/core/web-api/jsonpatch?view=aspnetcore-6.0>  avec le nuget : Microsoft.AspNetCore.JsonPatch le jsonpatch est décodée par : t Microsoft.AspNetCore.Mvc.NewtonsoftJson |
|  | article JSON Patch With ASP.NET Core | <https://dotnetcoretutorials.com/2017/11/29/json-patch-asp-net-core/> |

Création de contrôle d’intégrité

|  |  |  |
| --- | --- | --- |
|  | Surveillance intégré applicative  Hébergement et supervision applicative / Contrôles d’intégrité  (builder.Services.AddHealthChecks() | <https://learn.microsoft.com/fr-fr/aspnet/core/host-and-deploy/health-checks?view=aspnetcore-6.0> |
|  | Créer un contrôle d’intégrité spécifique | <https://learn.microsoft.com/fr-fr/aspnet/core/host-and-deploy/health-checks?view=aspnetcore-7.0#create-health-checks> |
|  | Surveillance de l’intégrité  HealthCheck UI : AspNetCore.HealthChecks.UI | <https://learn.microsoft.com/fr-fr/dotnet/architecture/microservices/implement-resilient-applications/monitor-app-health?source=recommendations> |
|  |  |  |