Cyril Seguenot

2019

Résumé

Support d’accompagnement d’une formation à ASP.Net Core Web API faite en présentiel

ASP.Net Core Web API

Support de cours

Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l’auteur ou de ses ayants droit ou ayants cause est illicite selon le Code de la propriété intellectuelle (Art L 122-4) et constitue une contrefaçon réprimée par le Code pénal.

Seules sont autorisées (Art L122-5) les copies ou reproductions strictement réservées à l’usage privé du copiste et non destinées à une utilisation collective, ainsi que les analyses et courtes citations justifiées par le caractère critique, pédagogique ou d’information de l’œuvre à laquelle elles sont incorporées, sous réserve, toutefois, du respect des dispositions des articles L 122-10 à L 122-12 du même Code, relatives à la reproduction par reprographie.

©Cyril Seguenot 2018

[1 Notions de base 3](#_Toc25882524)

[1.1 Les applications web 3](#_Toc25882525)

[1.2 Les API Web 3](#_Toc25882526)

[1.3 ASP.Net Core 4](#_Toc25882527)

[1.4 Le serveur web 5](#_Toc25882528)

[1.5 Les requêtes http 5](#_Toc25882529)

[2 Créer une API Web avec ASP.Net Core 7](#_Toc25882530)

[2.1 Projet Visual Studio 7](#_Toc25882531)

[2.2 Contrôleurs et actions 8](#_Toc25882532)

[2.3 Routage 10](#_Toc25882533)

[2.4 Paramétrer les actions 11](#_Toc25882534)

[2.5 Gérer un contexte de données 13](#_Toc25882535)

[2.6 Générer une réponse http 15](#_Toc25882536)

[3 Tester une API 17](#_Toc25882537)

[3.1 Requêtes POST 18](#_Toc25882538)

[3.2 Requêtes PUT 19](#_Toc25882539)

[3.3 Requêtes DELETE 20](#_Toc25882540)

[4 Documenter une API 20](#_Toc25882541)

[5 Créer une appli cliente 21](#_Toc25882542)

[6 Références 23](#_Toc25882543)

# Notions de base

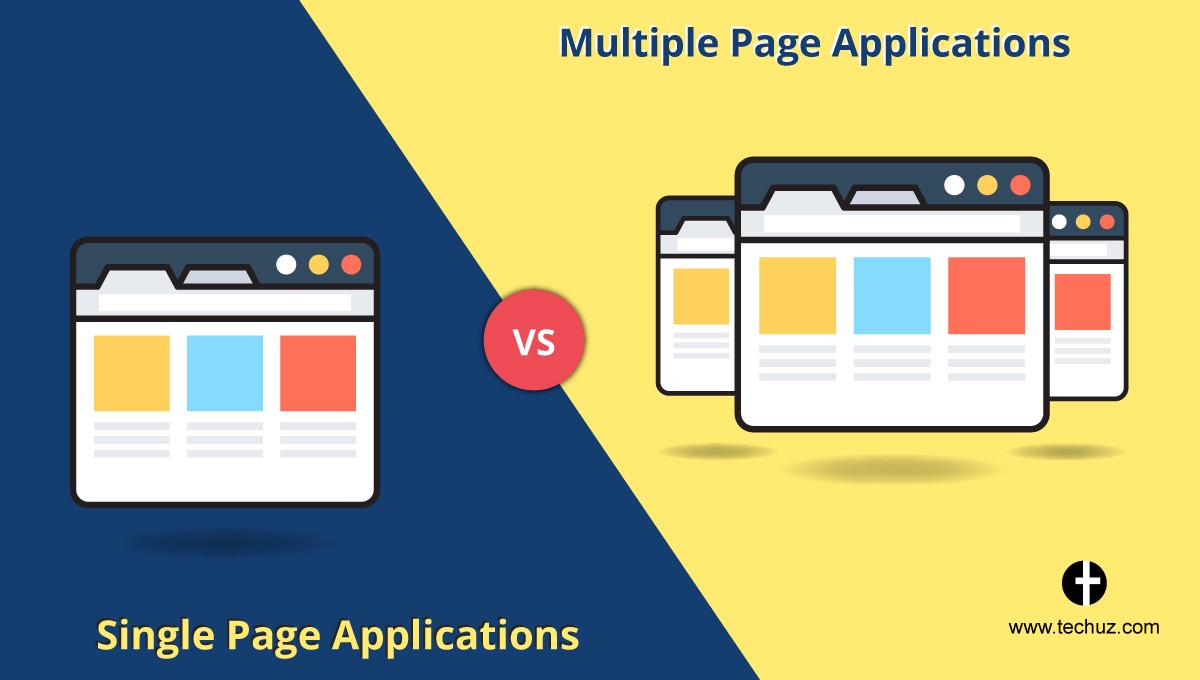
## Les applications web

Une application web est une application composée de 2 parties :

* La partie cliente, composée d’une ou plusieurs pages web pouvant contenir du code JavaScript qui les rend dynamiques et interactives (ex : validation des saisies, calculs, UI dynamique…etc.)
* La partie serveur, qui est hébergée et s’exécute sur une machine serveur. Elle exploite généralement une base de données

La partie cliente envoie des requêtes à partir du navigateur et obtient des réponses de la partie serveur sous différentes formes.

On distingue 2 grands types d’applications web :



### Les applis monopages (SPA)

Dans une application SPA (Single Page Application), la partie cliente est constituée d’une page HTML unique, qui échange des données avec la partie serveur sous forme de flux JSON ou XML.

La partie cliente est dans ce cas une application JavaScript complexe, qui permet d’adapter la page en fonction du contexte (actions faites par l’utilisateur, données reçues...). Tandis que la partie serveur est assez simple puisqu’elle n’a pas à gérer l’aspect visuel. Souvent il s’agit d’une API Web.

### Les applis multipages

Dans ce type d’application, les pages HTML de la partie cliente sont construites côté serveur et envoyées côté client en réponse aux requêtes du navigateur.

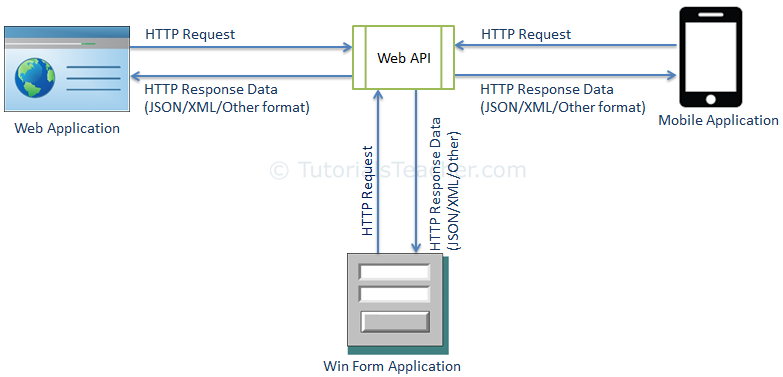
La partie serveur est assez complexe, car elle gère aussi la construction des pages HTML. Tandis que la partie cliente est au contraire plus simple, car gérée en grande partie par le serveur.

## Les API Web

Une API web est un ensemble de méthodes exposées sur un serveur pour répondre à des requêtes http. Une API web ne gère pas le visuel de la partie cliente. Elle se contente de répondre aux requêtes de l’application cliente (généralement une appli SPA) en traitant les données reçues et en lui renvoyant éventuellement des données au format JSON ou XML.

Techniquement, les API Web sont une mise en œuvre simplifiée de services WCF REST, basée sur le protocole http. Leurs méthodes sont accessibles au travers d'URI appelés points de terminaison (endpoints en anglais).

L’utilisation du protocole http, qui est à la fois simple et universel, rend ce type d’API exploitable par tous types d’applications clientes (web, mobiles ou bureau), comme le montre le schéma ci-dessous :



Il existe de nombreuses API web publiques exploitables gratuitement. Par exemple, l’API de Twitter expose des méthodes de lecture et d’écriture de données, qui nous permettent d’intégrer les fonctionnalités de Twitter dans nos propres applications.

## ASP.Net Core

Les API web peuvent être créées avec différentes technologies, telles que Java, ASP.Net, PHP…etc.   
Nous nous intéressons ici au framework ASP.Net Core, qui met en œuvre les notions de contrôleurs et d’actions (une action est une méthode d’API).

Une appli ou API ASP.Net Core peut être déployée avec toutes les dll nécessaires pour être autonome. Ou bien elle peut utiliser un framework partagé dans C:/Program Files/dotnet/shared.

Ce dossier contient les sous-dossiers suivants :

* Microsoft.NETCoreApp : contient les dll de base pour tous les types d’applications .Net Core.
* Microsoft.AspNetCore.App : contient les dll spécifiques aux applications et API ASP.Net Core.
* Microsoft.WindowsDesktop.App : contient les dll spécifiques aux applications de bureau Windows

**Remarque** : avant .Net Core 3.0, il y avait aussi un dossier Microsoft.AspNetCore.All contenant les dll du framework ASP.Net Core, plus des dll diverses ne faisant pas partie des frameworks, mais utiles au développement des applications. Ce dossier n’existe plus. Les dll supplémentaires éventuellement nécessaires sont fournies par des packages NuGet référencés individuellement dans l’application.

Les frameworks partagés sont installés par défaut avec Visual Studio, mais peuvent aussi être téléchargés sur <https://dotnet.microsoft.com/download>

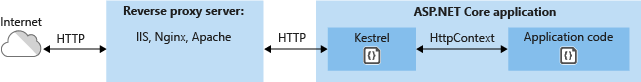
## Le serveur web

L’application ou API web côté serveur est exposée par un programme particulier qu’on appelle « Serveur Web ». Les serveurs web les plus connus sont Microsoft IIS, IBM Websphere, Oracle Weblogic, Apache Tomcat, Nginx...etc. Microsoft IIS ne fonctionne que sur Windows.

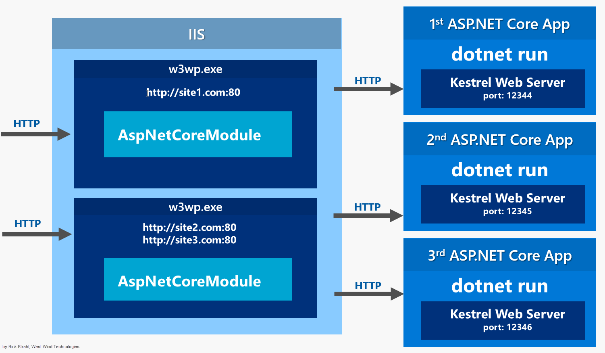
Visual Studio est fourni avec une version gratuite, simple et autonome de IIS, optimisée pour le développement et nommée **IIS Express**.

Tandis que les applications ASP.net étaient hébergées directement par le serveur web IIS, les applications ASP.Net Core sont auto-hébergées par une instance de serveur web basic nommé Kestrel.

**Kestrel** fait partie intégrante de .Net Core et peut donc s’exécuter sur différents systèmes d’exploitation. Il a été conçu pour être léger et le plus rapide possible.  
Il n’offre toutefois pas toutes les fonctionnalités d’un serveur web classique. Par exemple, il ne prend pas en charge les logs d’accès, le partage de ports, les certificats SSL, la console de gestion du serveur, le filtrage de requêtes, les redirections, la mise en cache de réponses...etc. C'est pourquoi Microsoft recommande de toujours utiliser un autre serveur web comme proxy de Kestrel pour les applications web publiques, comme illustré par le schéma de principe ci-dessous :



Le schéma suivant illustre l’interaction entre IIS et Kestrel pour le partage de ports :



## Les requêtes http

### Structure d’une requête

Une requête http est constituée de deux parties :

* Des en-têtes décrivant la demande
* Un corps, contenant les données à envoyer

Celles-ci peuvent être observées à l’aide des outils de développement intégrés au navigateur, ou bien à l’aide d’outils externes installés sur le PC, tels que Fiddler.

Voici un exemple d’affichage d’une requête POST par les outils de développement de Microsoft Edge :

|  |  |
| --- | --- |
| **En-têtes de la demande et de la réponse** | **Corps de la demande** |
|  |  |

La capture de gauche montre un exemple d’en-têtes, et la partie droite le corps de la demande.

La réponse est également constituée d’une partie en-têtes et d’un corps. Le corps n’est autre que le code html de la page.

### Les types de requêtes

La partie serveur traite essentiellement quatre méthodes de requêtes : GET, PUT, POST et DELETE

Voici leurs caractéristiques :

**Requêtes GET**

* Utilisées pour récupérer des données
* Les paramètres de requêtes sont passés dans l’url (donc attention à ne pas mettre des données sensibles dans ces paramètres)
* Les paramètres restent dans l’historique du navigateur
* Peuvent être mises en cache et en favoris dans le navigateur
* Sont restreintes par la longueur de l’url à 2048 caractères
* Ne peuvent véhiculer que des données au format ASCII

**Requêtes POST**

* Utilisées pour envoyer de nouvelles données sur le serveur
* Les paramètres de requêtes sont passés dans le corps de la requête
* Les paramètres ne restent pas dans l’historique du navigateur
* Ne peuvent pas être mises en cache ni dans les favoris
* Ne sont pas restreintes en volume de données

**Requêtes PUT**

* Utilisées pour remplacer (mettre à jour) des données sur le serveur. Il faut donc spécifier l’url de la ressource à mettre à jour
* Idem POST pour le reste

**Requêtes DELETE**

* Utilisées pour supprimer les données spécifiées sur le serveur
* Les paramètres de requêtes sont passés dans l’url
* Idem POST pour le reste

Côté client, la méthode est spécifiée dans le code html du formulaire qui émet la requête :

<form action="/Movies/Edit/5" method="post">

Get est la valeur par défaut si aucune méthode n’est spécifiée dans le code html

Côté serveur, l’action qui traite la requête spécifie la méthode prise en charge au moyen d’un attribut :

[HttpPost]

public async Task<IActionResult> Edit(int id, Movie movie)

Get est la méthode prise en charge si on ne spécifie pas d’attribut.

### Actions associées

Bien que chaque méthode de requête ait une finalité particulière, rien n’est imposé techniquement sur la nature du traitement réellement réalisé par l’action côté serveur. Il est donc possible (mais pas conseillé) d’utiliser une méthode dans un but différent de ce pourquoi elle est prévue. Par exemple, on pourrait utiliser systématiquement la méthode POST à la place de PUT et de DELETE...

Au minimum, les opérations qui modifient les données (Create Update Delete) ne devraient pas être réalisées en réponse à une requête de type GET.

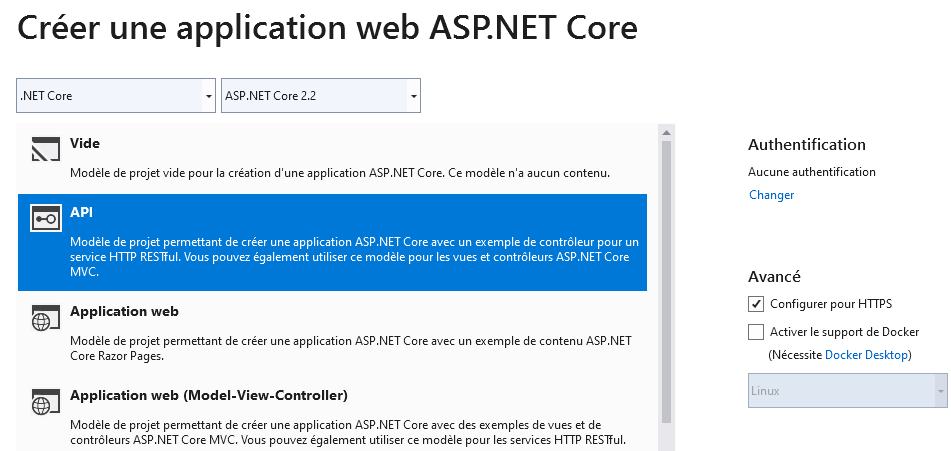
Lorsque les données sont transmises par le corps du message (requêtes POST ou PUT), elles doivent être au format xml ou json.

# Créer une API Web avec ASP.Net Core

## Projet Visual Studio

Lors de la création d’un nouveau projet, on passe par les étapes suivantes :

* On choisit tout d’abord le type « Application web ASP.Net Core »
* On sélection l’emplacement de la solution
* On arrive ensuite à la fenêtre suivante, où on spécifie le type API et la version d’ASP.Net Core à utiliser :



Après validation, Visual Studio génère un projet ayant l’arborescence suivante :

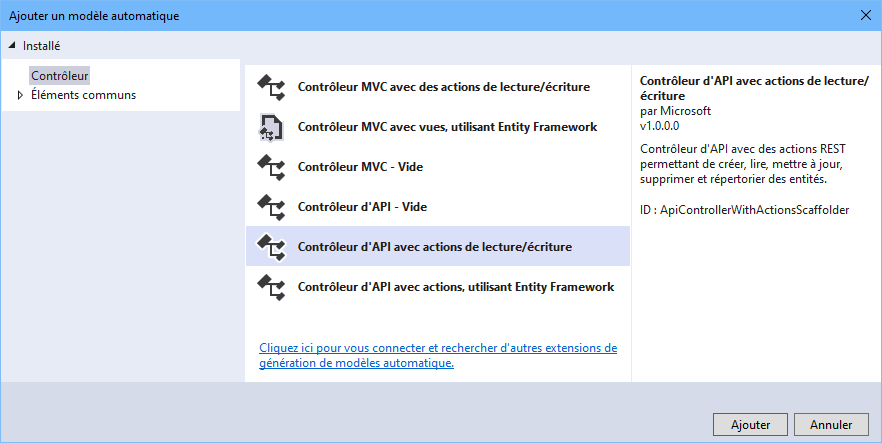
|  |  |
| --- | --- |
|  | Dépendances :   * Analyseurs : bibliothèques pour l’analyse de code (qualité) * Frameworks : frameworks partagés utilisés par l’API * Packages : modules externes apportant des fonctionnalités complémentaires (exemple : Entity Framework Core)   Le dossier Controllers contient des classes dérivant de ControllerBase et qui implémentent les méthodes de l’API.  Un contrôleur WeatherForecast est créé par défaut à titre d’exemple. |

## Contrôleurs et actions

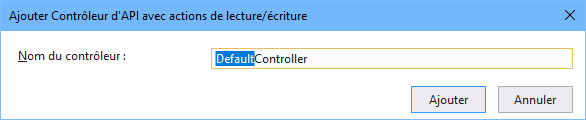
Un contrôleur est une classe chargée de répondre à des requêtes http. A chaque requête valide reçue par le serveur, un contrôleur est instancié et l’une de ses méthodes est appelée pour répondre à la requête.

Pour ajout un contrôleur, dans le menu contextuel du dossier Controllers, on choisit Ajouter \ Contrôleur...

Dans la fenêtre qui s’ouvre, on peut sélectionner le type de contrôleur. En l’occurrence, on choisit Contrôleur d’API avec des actions de lecture / écriture (ou vide si on préfère que Visual Studio ne génère aucune méthode d’action) :



On est ensuite invité à nommer la classe :



Visual Studio génère alors une classe xxxController (xxx étant le nom choisi ci-dessus), ressemblant à ceci :

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

[ApiController]

public class EssaiController : ControllerBase

{

    // GET: api/Essai

    [HttpGet]

    public IEnumerable<string> Get()

    {

        return new string[] { "value1", "value2" };

    }

    // GET: api/Essai/5

    [HttpGet("{id}", Name = "Get")]

    public string Get(int id)

    {

        return "value";

    }

    // POST: api/Essai

    [HttpPost]

    public void Post([FromBody] string value)

    {

    }

    // PUT: api/Essai/5

    [HttpPut("{id}")]

    public void Put(int id, [FromBody] string value)

    {

    }

    // DELETE: api/Essai/5

    [HttpDelete("{id}")]

    public void Delete(int id)

    {

    }

}

Les noms de classes de contrôleurs doivent se terminer par « Controller »

Les méthodes du contrôleur sont appelées **actions** et sont nommées par défaut comme les différents verbes de requêtes http : Get, Post, Put, Delete. On peut les renommer pour préciser davantage la nature des données manipulées (exemple : GetEmployees).

2 actions répondant au même type de requête doivent obligatoirement se distinguer par leurs paramètres ou attributs, comme nous allons le voir plus loin.

La classe de contrôleur dérive de ControllerBase, qui fournit de nombreuses méthodes utilisables à l’intérieur des actions.

## Routage

### Points de terminaison et routes

Dans une classe de contrôleur, chaque action est accessible via un point de terminaison (endpoint), dont l’url est de la forme : **https://nomdomaine/route**

**La route** peut comprendre plusieurs **segments** (parties séparées par des /).

Par exemple, en saisissant l’url <https://localhost:12345/essai/5> dans un navigateur, on émet une requête GET qui appelle la méthode Get de la classe EssaiController, en lui passant la valeur 5 en paramètre.

**Remarques** :

* https signifie que la connexion est sécurisée par chiffrement SSL ou TLS. Même si ce n’est pas une obligation, il est vivement recommandé d’utiliser https pour contrer les fuites de données et autres attaques.
* Bien que les classes de contrôleurs soient toujours suffixées par « Controller », il ne faut pas mettre ce suffixe dans les url.

Pour définir le format de chaque route, on utilise des attributs placés sur le contrôleur et sur les actions.

### Modèle de routes du contrôleur

Lorsque Visual Studio génère une classe de contrôleur, il la décore par défaut de l’attribut suivant :

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

La valeur entre crochets [controller] est un joker qui remplace le nom du contrôleur.

Pour un contrôleur nommé « Essai », cet attribut spécifie que tous les points de terminaison commenceront par quelque-chose du type :

https://nomdomaine/api/essai

On peut modifier librement la valeur de l’attribut, par exemple en enlevant ou en modifiant le segment « api ».

### Routes des actions

On spécifie les fins des routes par des attributs décorant les actions.

Pour cela, on peut utiliser l’attribut Route, comme pour le contrôleur :

// GET: api/essai/produits

[HttpGet]  
[Route("liste")]  
public IEnumerable<string> Get()

{

    ...

}

... ou bien intégrer le segment de route dans l’attribut [http...] :

// GET: api/essai/produits

[HttpGet("liste")]  
public IEnumerable<string> Get()

{

    ...

}

Dans cet exemple, l’action Get est donc accessible via l’url :

https://localhost:52902/api/essai/liste

**Remarque** : on peut aussi enlever l’attribut de routage qui décore la classe et définir une route complète au niveau de chaque action, mais cela s’avère peu commode lorsqu'on veut modifier de façon globale le routage sur un contrôleur.

### Paramètres d’url

Lorsque le routage est correctement défini, une url permet donc d’appeler une action précise. Mais elle peut aussi servir à transmettre des données à l’action, en tant que paramètres d’url. Voici la syntaxe à utiliser pour cela :

https://nomdomaine/route?param1=valeur1&param2=valeur2

On place un point d’interrogation à la fin de la route et on spécifie une ou plusieurs paires « param=valeur » séparées par des « & » (éperluette).

Exemple :

https://localhost:52902/api/essai?rech=truc&annee=2020

Cette possibilité est souvent exploitée pour envoyer la valeur saisie par l’utilisateur dans une zone de recherche.

## Paramétrer les actions

Pour paramétrer correctement une action, il faut définir :

* Sa route
* A quel type de requête elle répond
* Comment les données doivent lui être transmises

Nous allons voir comment définir les 2 derniers éléments.

### Type de requête

Pour spécifier à quel type d’action doit répondre la requête, on utilise l’attribut **[Http...] :**

// GET: api/Essai

[HttpGet]

public IEnumerable<string> Get()

// PUT: api/essai/5  
[HttpPut("{id}")]

public void Put(int id, [FromBody] string value)

Même si le nom de l’action comprend aussi le nom de la méthode http, c’est bien l’attribut qui indique le type de requête gérée. Le nom de l’action n’a pas d’importance.

L’attribut [http...] peut prendre un ou plusieurs paramètres.

Le premier indique un segment de route qui s’ajoute au point de terminaison de l’action, comme nous l’avons vu précédemment.   
Lorsque ce paramètre est entre accolades, cela signifie que le segment peut avoir une valeur variable et que cette valeur détermine celle d’un paramètre de la méthode. Dans l’exemple ci-dessus, le segment de route « {id} » détermine donc la valeur du paramètre « id » de l’action.

### Provenance des données

Les requêtes http peuvent envoyer des données aux actions via le corps de la requête, l’en-tête, l’url...etc.

Sur l’action, il faut donc préciser à quel endroit doivent être récupérées les données reçues. On utilise pour cela des attributs placés devant les paramètres :

public void Post([FromBody] string value)

Voici les attributs possibles :

|  |  |
| --- | --- |
| **Attribut** | **Donnée recherchée dans** |
| [[FromBody]](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.frombodyattribute) | Le corps de la requête |
| [[FromForm]](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.fromformattribute) | L’objet FormData du corps de la requête |
| [[FromHeader]](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.fromheaderattribute) | L’en-tête de la requête |
| [[FromQuery]](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.fromqueryattribute) | Un paramètre dans l’url de la requête |
| [[FromRoute]](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.fromrouteattribute) | Un morceau de la route de la requête |
| [[FromServices]](https://docs.microsoft.com/fr-fr/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-3.0#action-injection-with-fromservices) | Le service injecté en tant que paramètre |

Ainsi dans les exemples précédents, les données correspondant au paramètre « value » des méthodes Post et Put sont recherchées dans le corps de la requête.

Si une action possède plusieurs paramètres, ces derniers peuvent éventuellement être décorés d’attributs différents.

**Attributs par défaut**

Si on ne spécifie aucun attribut sur un paramètre, un attribut par défaut lui est affecté. Les règles d’affectation sont définies par l’attribut [ApiController] de la classe de contrôleur. Les voici :

|  |  |
| --- | --- |
| **Type de paramètre** | **Attribut appliqué par défaut** |
| Type complexe (classe) | [FromBody] |
| IFormFile ou IFormFileCollection | [FromForm] |
| Paramètre dont le nom correspond à celui d’un segment dans le modèle de route | [FromRoute] |
| Type simple | [FromQuery] |

## Gérer un contexte de données

Une API web reçoit des données des requêtes http et renvoie des données dans les réponses qu’elle fournit. Elle doit donc avoir :

* Un ou plusieurs **systèmes de stockage de données**. Ce peut être la mémoire vive, des fichiers (xml, json ou autres), ou plus généralement une base de données
* Un ou plusieurs **contextes de données** qui sont des classes possédant des méthodes pour récupérer, créer, mettre à jour et supprimer des données dans le système de stockage.

Un contexte de données est destiné à être utilisé dans les actions des contrôleurs. Il peut mettre en œuvre différentes technologies adaptées au système de stockage : LINQ, LINQ To XML, SqlCommand, Entity Framework Core...etc.

Nous allons voir dans ce chapitre comment créer un contexte et le référencer pour qu’il soit utilisable dans les contrôleurs.

**Remarque** : lorsqu'on utilise Entity Framework Core, l’utilitaire en ligne de commande Scaffold-DbContext génère automatiquement le contexte de données.

#### Connexion à une base

Si les données sont stockées dans une base de données, le contexte a besoin d’une chaîne de connexion pour accéder à cette base. Il est recommandé de centraliser cette chaîne dans le fichier de paramètres appsettings.json, dont voici un extrait :

{

  "ConnectionStrings": {

    "Northwind2Connect": "Server=(localdb)\\mssqllocaldb;Database=Northwind2;Trusted\_Connection=True;"

  },

  ...

}

Dans cet exemple, on déclare à l’intérieur de la section ConnectionStrings, une chaîne de connexion nommée Northwind2Connect, permettant d’accéder à une base de données SQL Server locale. L’authentification se fait via le compte Windows.

Pour que la chaîne soit accessible aux classes de contextes de données, il faut l’enregistrer dans la collection des services de l’application, qui est accessible depuis la méthode ConfigureService de la classe Startup. Pour simplifier, on peut enregistrer un singleton donnant accès à l’ensemble des paramètres du fichier appsettings, comme illustré ci-dessous :

public void ConfigureServices(IServiceCollection services)

{

   // Crée un singleton donnant accès à l'ensemble des paramètres de l'appli

   services.AddSingleton<IConfiguration>(Configuration);

   ...

}

**Remarque** : il existe une autre façon de faire un peu plus complexe, qui consiste à créer une classe spécifique pour la section « ConnectionStrings » du fichier de paramètres. Cela permet ensuite de restreindre l’accès à la chaîne de connexion uniquement aux classes de contexte, et ainsi de sécuriser davantage l’application.

#### Créer un contexte

Un contexte de données est décrit par une classe créée manuellement ou générée automatiquement (si on utilise Entity Framework Core).

Voici un exemple de classe de contexte qui utilise un service pour récupérer une chaîne de connexion à une base de données :

public class Northwind2Context

{

   private readonly string \_connect;

   public Northwind2Context(IConfiguration config)

   {

      \_connect = config.GetConnectionString("Northwind2Connect");

   }

   public List<Employee> GetEmployees()

   {

      ...

   }

}

On crée un constructeur possédant un paramètre de type Iconfiguration. C’est ce qu’on appelle une **injection de dépendance.**

On extrait la chaîne de connexion souhaitée de ce paramètre, grâce à la méthode GetConnectionString(). Puis on mémorise la chaîne dans un champ privé en lecture seule pour pouvoir y accéder dans toutes les méthodes de la classe.

Quand la classe de contexte sera instanciée, un objet de type IConfiguration sera automatiquement fourni à son constructeur par ASP.Net, grâce au fait qu’on a enregistré un singleton de type IConfiguration dans la liste des services de l’API.

**Remarques** :

* La liste des services de l’API est gérée par ce qu’on appelle un **conteneur IoC** (pour Inversion of Control). C’est lui qui instancie les classes enregistrées en tant que services et passe à leurs constructeurs les paramètres dont ils ont besoin. Le conteneur IoC joue donc un rôle central dans ASP.Net Core.
* Grâce à l’objet IConfiguration passé à son constructeur, le contexte peut aussi avoir accès à d’autres paramètres applicatifs qui peuvent lui être utiles.
* Si le contrôleur n’a besoin d’aucune chaîne de connexion ni autre paramètre, il est bien entendu inutile de les passer en paramètre à son constructeur.
* Les classes de contexte sont généralement placées dans un dossier sous-dossier Data du projet

#### Référencer le contexte

La classe de contexte est destinée à être utilisée par les actions des contrôleurs. Pour qu’un contrôleur ait accès à une instance du contexte, on utilise comme précédemment l’injection de dépendance et le conteneur IoC.

On commence donc par enregistrer la classe de contexte dans la liste des services de l’application, au moyen de la ligne ci-dessous :

public void ConfigureServices(IServiceCollection services)

{

   // Crée un singleton donnant accès à l'ensemble des paramètres de l'appli

   services.AddSingleton<IConfiguration>(Configuration);

   // Enregistre la classe de contexte de données comme service

   services.AddTransient<Northwind2Context>();  
 ...

}

**Remarque** : La méthode AddTrancient indique à ASP.Net Core de créer une instance ordinaire de la classe et non un singleton. Si la classe de contexte implémente une interface, on utilise la syntaxe suivante pour l’enregistrer :

services.AddTransient(typeof(INorthwindContext), typeof(Northwind2Context));

Puis on injecte une dépendance vers le contexte dans la classe du contrôleur :

public class EmployeesController : ControllerBase

{

   private readonly Northwind2Context \_context;

   public EmployeesController(Northwind2Context context)

   {

      \_context = context;

   }

}

Ainsi, quand le contrôleur sera instancié (lors de la réception d’une requête http), ASP.Net Core instanciera automatiquement un contexte de données, et le passera en paramètre au constructeur du contrôleur.

## Générer une réponse http

Lorsqu’une action traite une requête http, elle doit renvoyer une réponse. Nous allons donc voir maintenant de quoi est constituée une réponse et comment la générer avec ASP.Net Core.

### Les codes HTTP

Une réponse http est constituée d’un corps et d’un en-tête.

Le corps est un flux JSON ou autre format. L’en-tête contient notamment **un code http** qui est un entier à 3 chiffres dont le premier indique le type de résultat :

* 1XX : Information
* 2XX : Succès
* 3XX : Redirection
* 4XX : Erreur du client
* 5XX : Erreur du serveur

Quelques codes parmi les plus courants :

* 200 : succès de la requête
* 201 : requête exécutée avec pour résultat la création d’une nouvelle ressource
* 204 : requête exécutée sans résultat retourné
* 301 et 302 : redirection, respectivement permanente et temporaire
* 401 : utilisateur non authentifié
* 403 : accès refusé par le serveur
* 404 : la ressource demandée n’existe pas sur le serveur
* 500 : erreur générique sur le serveur
* 503 : serveur temporairement indisponible (en maintenance)
* 504 : le serveur n'a pas répondu dans le délai imparti

La liste complète des valeurs est donnée par le [cette page](https://msdn.microsoft.com/en-us/library/windows/desktop/aa384325(v=vs.85).aspx) de doc Microsoft. On peut également consulter [cette page](https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP) Wikipédia.

Le code http est modélisé par la propriété StatusCode de la classe StatusCodeResult.

### Les types de réponses possibles

L’action qui a traité la requête du client génère la réponse http. Pour cela, elle doit retourner un résultat de l’un des 3 types suivants :

* Type spécifique (une classe qu’on a définie soi-même)
* IActionResult
* ActionResult<T> (introduit dans ASP.Net Core 2.1)

Un type spécifique est suffisant lorsque le résultat à retourner est toujours du même type. En revanche, si l'action peut retourner des types de résultats différents selon certaines conditions, il faut utiliser l'un des 2 autres types.

ASP.Net Core fournit des classes spécialisées qui implémentent IActionResult. Il s’agit des classes dérivées de [ActionResult](https://docs.microsoft.com/fr-fr/dotnet/api/microsoft.aspnetcore.mvc.actionresult). Cette classe abstraite a une vingtaine de dérivées directes, dont certaines ont également des dérivées. Utilisez le lien précédent pour explorer ces classes.

De plus, la classe [ControllerBase](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase?view=aspnetcore-3.0) fournit des méthodes qui génèrent automatiquement des instances de ces classes, ce qui simplifie beaucoup le code. En voici quelques-unes parmi les plus courantes :

|  |  |
| --- | --- |
| **Méthode** | **Code http en-tête de la réponse** |
| BadRequest | 400 |
| CreatedAtAction | 201 |
| Forbid | 403 |
| NoContent | 204 |
| NotFound | 404 |
| Ok | 200 |
| Redirect | 302 |
| Unauthorized | 401 |
| ValidationProblem | 400 |

**Remarque** : beaucoup de ces méthodes possèdent plusieurs surcharges pour passer également des données dans le corps de la réponse http.

Voici un exemple typique de mise en œuvre :

|  |
| --- |
| [HttpGet("{id}", Name = "GetEmp")] [Produces(TypeOf(Employee))]  public IActionResult GetEmployee(int id)  {     var employee = \_context.GetEmployee(id);     if (employee == null) return NotFound();       return Ok(employee);  }    // POST: Employees  [HttpPost]  public ActionResult Post([FromBody] Employee emp)  {     if (string.IsNullOrWhiteSpace(emp.FirstName) ||        string.IsNullOrWhiteSpace(emp.LastName))        return BadRequest();       // on crée l’employé dans la base et on récupère son Id     emp.EmployeeId = \_context.CreateEmployee(emp);       // On renvoie une réponse avec les en-têtes "status Code: 201 Created"     // et "location: <url d'accès à l’employé>", et un corps contenant l’employé     return CreatedAtAction(nameof(GetEmployee), new { id = emp.EmployeeId }, emp);  } |

Dans la méthode Get :

* Si l’employé demandé est trouvé, la réponse contiendra un en-tête « Status Code : 200 Ok » et son corps contiendra l’employé.
* Sinon, la réponse contiendra un en-tête « Status Code : 404 Not Found » et son corps sera vide
* Le type IActionResult ne permet pas de connaître le type d’objet renvoyé dans le corps de la réponse. C’est pourquoi on spécifie ce dernier grâce à l’attribut [Produces] à des fins de documentation.

Dans la méthode Post :

* Si le nom ou le prénom de l’employé à créer est vide, on retourne une réponse avec un en-tête « Status Code : 400 Bad Request » et un corps vide.
* Après création de l’employé dans la base, la méthode CreatedAtAction construit une réponse http avec des en-tête « Status Code : 201 Created », et « Location : https://localhost:44383/Employees/13 » et un corps contenant l’employé créé

En effet, dans une action Post il est d’usage de renvoyer l’url d’accès à l’entité créée, ainsi que l’entité elle-même, car ces informations peuvent être très utiles à l’application cliente.  
L’url est générée grâce aux 2 premiers paramètres de la méthode CreatedAtAction, qui représentent :

* Le nom de l’action Get permettant de récupérer l’entité créée (cette action doit donc être nommée avec [HttpGet(Name=... )])
* Les éventuelles valeurs à fournir pour générer la route complète (ici, l’id de l’employé)

A partir de .Net Core 2.1, Microsoft a ajouté le type de résultat [ActionResult<T>](https://docs.microsoft.com/fr-fr/dotnet/api/microsoft.aspnetcore.mvc.actionresult-1?f1url=https%3A%2F%2Fmsdn.microsoft.com%2Fquery%2Fdev16.query%3FappId%3DDev16IDEF1%26l%3DFR-FR%26k%3Dk(Microsoft.AspNetCore.Mvc.ActionResult%601);k(DevLang-csharp)%26rd%3Dtrue&view=aspnetcore-3.0) qui est semblable au précédent, avec l’avantage que le type des données renvoyées est spécifié directement dans le type de retour ; il n’y a donc plus besoin d’ajouter d’attribut [Produces] :

|  |
| --- |
| [HttpGet("{id}")] public ActionResult<Employee> GetById(int id) {  ... } |

**Remarque** : si l’attribut [ApiController] décore la classe de contrôleur, les actions du contrôleur renvoient automatiquement une réponse http 400 (bad request) si les données qu’elles reçoivent ne sont pas conformes aux types des paramètres. Cf. [cette page](https://docs.microsoft.com/fr-fr/aspnet/core/web-api/?view=aspnetcore-3.0) pour voir tous les effets de cet attribut.

# Tester une API

Les requêtes http de type Get sont simples à tester, car les données à passer éventuellement à la requête sont incluses dans l’url. Il suffit donc de saisir l’url souhaitée et d’observer la réponse qui s’affiche dans la page. Pour avoir plus de détails, on peut observer la requête et sa réponse dans les outils de développement du navigateur, comme vu plus haut.

Les requêtes http de type Post et PUT nécessitent en revanche de spécifier un corps de requête. Pour cela, on a besoin d’un module de navigateur supplémentaire ou bien d’un logiciel indépendant. On peut citer :

* PostMan : un logiciel gratuit et populaire avec des fonctionnalités avancées. Il nécessite de créer un compte sur le [site de Postman](https://www.getpostman.com/).
* RestClient : un module pour Firefox simple et pratique

Nous allons voir comment tester chaque type de requête http avec ces logiciels.

## Requêtes POST

Supposons qu’on souhaite tester la création d’un employé via l’action suivante :

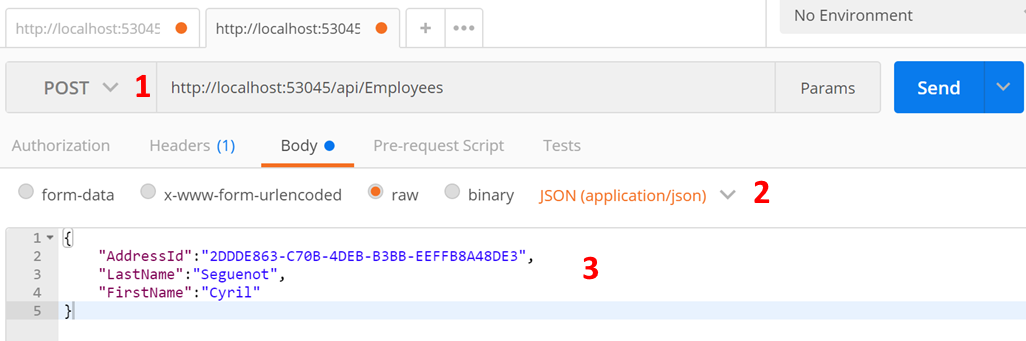
// POST: api/Employees

[HttpPost]

public async Task<IActionResult> PostEmployee([FromBody] Employee employee)

Ce code nous indique que les données doivent être envoyées dans le corps de la requête sous forme d’un objet Employee.

L’image ci-dessous montre comment envoyer ce type de requête :



1. On sélectionne le type POST et on saisit l’url
2. Dans l’onglet Body, on spécifie le format de données JSON et on sélectionne le bouton radio « raw » (brut) de façon à saisir les données sous forme d’un flux JSON brut.
3. On saisit les données au format JSON

Dans cet exemple, seules les données obligatoires ont été renseignées.

Lorsqu’on clique sur le bouton « Send », on obtient le visuel suivant :



Le code de statut de la réponse est indiqué dans la partie supérieure droite, avec le temps d’exécution de la requête, et la taille des données renvoyées.

Ici, le corps de la réponse contient l’employé complet qui a été créé, ce qui permet notamment de récupérer son Id, qui a été généré par la base.

## Requêtes PUT

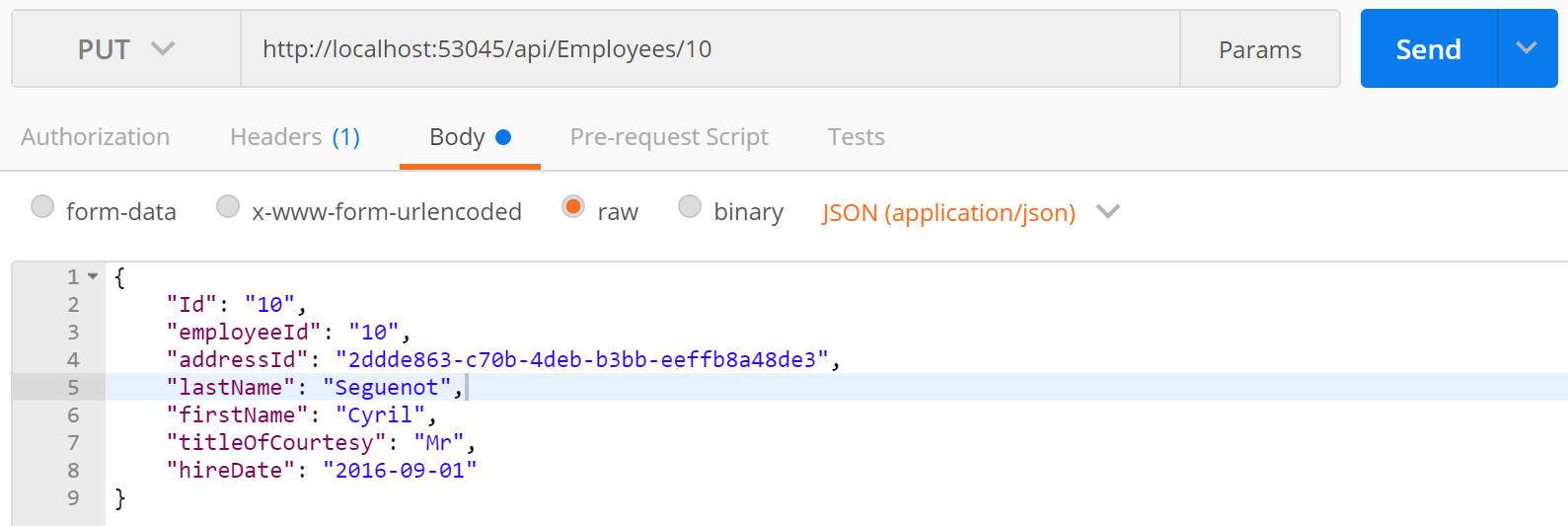
Pour tester la mise à jour d’un employé via l’action suivante :

// PUT: api/Employees/5

[HttpPut("{id}")]

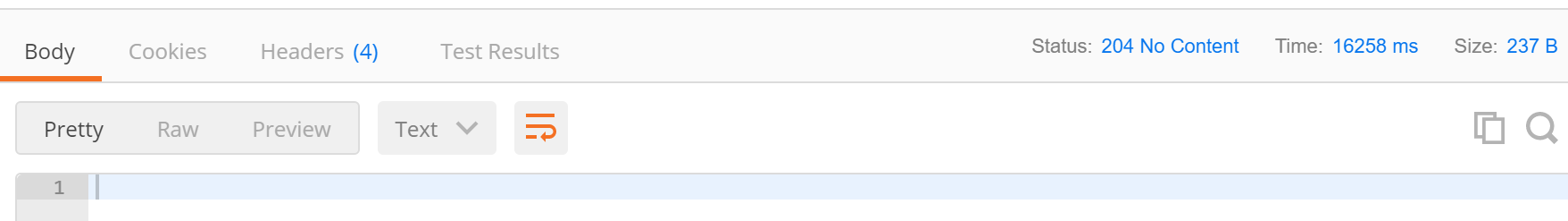
public async Task<IActionResult> PutEmployee([FromRoute] int id, [FromBody] Employee employee)

… on exécuter la requête suivante dans PostMan :



Dans cet exemple, on met à jour les champs titleOfCourtesy et hireDate. La propriété Id correspond au premier paramètre de l’action.

La réponse obtenue est la suivante :



Le code de statut 204 nous indique que l’action a réussi et qu’elle ne retourne pas de contenu dans le corps de la réponse.

## Requêtes DELETE

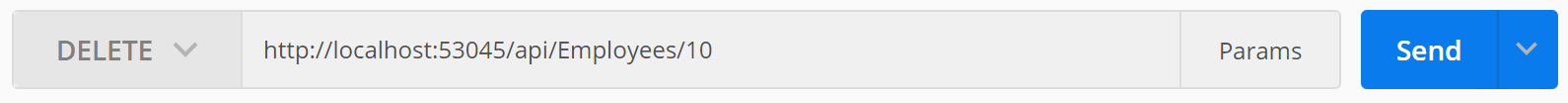
Pour tester la suppression d’un employé via l’action suivante :

// DELETE: api/Employees/5

[HttpDelete("{id}")]

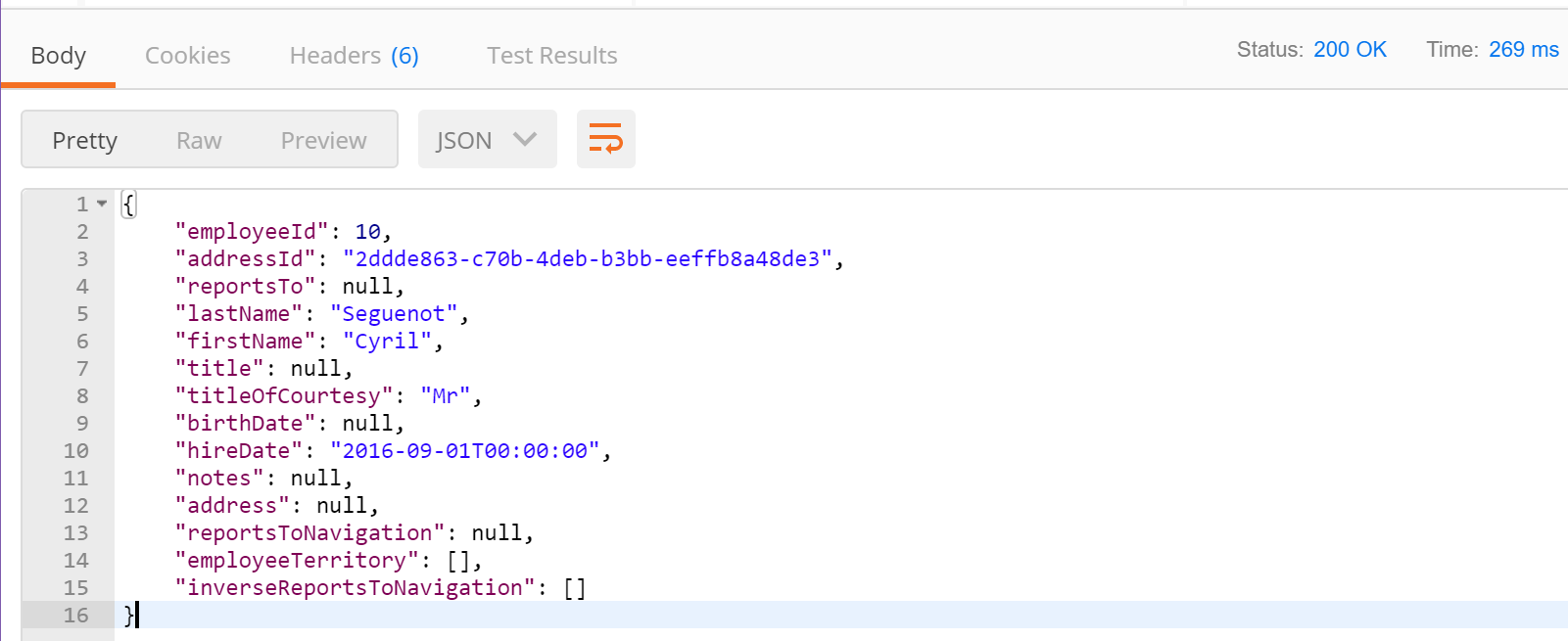
public async Task<IActionResult> DeleteEmployee([FromRoute] int id)

… on exécuter la requête suivante dans PostMan :



Il n’y a cette fois besoin d’aucun corps de requête.

La réponse est la suivante :



On obtient le code de statut 200, et on récupère dans le corps de la réponse l’entité qui a été supprimée.

# Documenter une API

Lorsqu'on créer une API web, il est important de la documenter afin que les développeurs d’applications clientes sachent comment l’utiliser. C’est d’autant plus vrai que l’API est riche et exposée à un public large.

La documentation peut être générée automatiquement grâce à un module tel que [Swagger](https://docs.microsoft.com/fr-fr/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-3.0). Ce dernier s’appuie sur des attributs ajoutés spécifiquement sur les actions à des fins de documentation.

En voici quelques exemples :

|  |
| --- |
| [HttpGet("{id}")] [Produces(TypeOf(Employee))] [ProducesResponseType(StatusCodes.Status200OK)] [ProducesResponseType(StatusCodes.Status404NotFound)] public IActionResult GetById(int id) {  if (!\_context.TryGetEmployee(id, out var employee))  {  return NotFound();  }    return Ok(employee); } |

Dans l’exemple ci-dessus :

* L’attribut [Produces] permet de spécifier le type des données renvoyées dans le corps de la réponse
* Les attributs [ProducesResponseType] permettent de préciser les différents types de réponses que l’action peut renvoyer

# Créer une appli cliente

Les API web de type REST utilisent le protocole http qui est simple et universel. De plus, elles séparent bien les responsabilités entre client et serveur. Cela permet donc de faire des applications clientes de toutes sortes : web, natives ou hybrides, mobiles ou de bureau...

Pour la création d’applications clientes web avec HTML, CSS et JavaScript, on se référera au cours sur ces technologies. Nous allons voir ici comment créer une application de bureau de type console avec .Net Core.

Voici les étapes à suivre :

* Créer un projet de type Application console (.Net Core)
* Ajouter des références vers system.net et System.Net.Http

Puis utiliser un code semblable à ce qui suit pour les méthodes de tests :

class Program

{

    static HttpClient client = new HttpClient();

    static void ShowEmployee(Employee emp)

    {

        Console.WriteLine($"{emp.LastName} {emp.FirstName},  
  embauché le {emp.HireDate}");

    }

    static async Task<Uri> CreateEmployeeAsync(Employee emp)

    {

        HttpResponseMessage response = await client.PostAsJsonAsync(

            "api/employees", emp);

        response.EnsureSuccessStatusCode();

        // retourne l'uri de la ressource créée

        return response.Headers.Location;

    }

    static async Task<Employee> GetEmployeeAsync(string path)

    {

        Employee emp = null;

        HttpResponseMessage response = await client.GetAsync(path);

        if (response.IsSuccessStatusCode)

        {

            emp = await response.Content.ReadAsAsync<Employee>();

        }

        return emp;

    }

    static async Task<Employee> UpdateEmployeeAsync(Employee emp)

    {

        HttpResponseMessage response = await client.PutAsJsonAsync(

            $"api/employees/{emp.EmployeeId}", emp);

        response.EnsureSuccessStatusCode();

        // Deserialise l'employé mis à jour depuis le corps de la réponse

        emp = await response.Content.ReadAsAsync<Employee>();

        return emp;

    }

    static async Task<HttpStatusCode> DeleteEmployeeAsync(int id)

    {

        HttpResponseMessage response = await client.DeleteAsync(

            $"api/employees/{id}");

        return response.StatusCode;

    }

NB/ Ces méthodes de tests sont toutes asynchrones, car elles utilisent les méthodes de la classe HttpClient, qui sont elles-mêmes asynchrones.

Voici un exemple de code qui exploite les méthodes ci-dessus :

static void Main()

{

    RunAsync().GetAwaiter().GetResult();

}

static async Task RunAsync()

{

    // Modifier le port selon les besoins

    client.BaseAddress = new Uri("http://localhost:53045/");

    client.DefaultRequestHeaders.Accept.Clear();

    client.DefaultRequestHeaders.Accept.Add(

        new MediaTypeWithQualityHeaderValue("application/json"));

    try

    {

        // Create a new emp

        Employee emp = new Employee

        {

            AddressId = new Guid("2DDDE863-C70B-4DEB-B3BB-EEFFB8A48DE3"),

            LastName = "Durant",

            FirstName ="Eric"

        };

        var url = await CreateEmployeeAsync(emp);

        Console.WriteLine($"Employé créé à l'url {url}");

        // Get the emp

        emp = await GetEmployeeAsync(url.PathAndQuery);

        ShowEmployee(emp);

        // Update the emp

        Console.WriteLine("Mise à jour de la date d'embauche...");

        emp.HireDate = new DateTime(2016, 9, 1);

        await UpdateEmployeeAsync(emp);

        // Get the updated emp

        emp = await GetEmployeeAsync(url.PathAndQuery);

        ShowEmployee(emp);

        // Delete the emp

        var statusCode = await DeleteEmployeeAsync(emp.EmployeeId);

        Console.WriteLine($"Employé supprimé (statut HTTP = {(int)statusCode})");

    }

    catch (Exception e)

    {

        Console.WriteLine(e.Message);

    }

    Console.ReadLine();

}

# Références

[Documentation Microsoft sur ASP.Net Core](https://docs.microsoft.com/en-us/aspnet/core/)

[Documentation Microsoft sur la création d’une API web avec ASP.Net Core](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api)