Cyril Seguenot

2018

Résumé

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

ASP.Net Core MVC

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 4](#_Toc506248702)

[1.1 Application web 4](#_Toc506248703)

[1.2 Serveur web 4](#_Toc506248704)

[1.3 Différence entre un service web et une application web 4](#_Toc506248705)

[1.4 Les requêtes http 5](#_Toc506248706)

[2 ASP.Net MVC 7](#_Toc506248707)

[2.1 ASP.Net 7](#_Toc506248708)

[2.2 Le modèle MVC 7](#_Toc506248709)

[3 Structure d’un projet 8](#_Toc506248710)

[3.1 Gestion des packages 9](#_Toc506248711)

[3.2 Démarrage de l’application 10](#_Toc506248712)

[3.3 Configuration 11](#_Toc506248713)

[4 Le modèle 13](#_Toc506248714)

[4.1 Création 13](#_Toc506248715)

[4.2 Affichage et validation 14](#_Toc506248716)

[4.3 Modèles pour vues 15](#_Toc506248717)

[5 Les contrôleurs 15](#_Toc506248718)

[5.1 Création d’un contrôleur 15](#_Toc506248719)

[5.2 Points de terminaison et actions 17](#_Toc506248720)

[5.3 Les attributs d’actions 17](#_Toc506248721)

[5.4 Liaison de modèle 18](#_Toc506248722)

[6 Les vues 20](#_Toc506248723)

[6.1 Concepts 20](#_Toc506248724)

[6.2 Création d’une vue 20](#_Toc506248725)

[6.3 Rattachement aux actions des contrôleurs 22](#_Toc506248726)

[6.4 Passage des données du contrôleur à la vue 23](#_Toc506248727)

[6.5 Formats d’affichage 25](#_Toc506248728)

[6.6 Moteur de vue et syntaxe Razor 27](#_Toc506248729)

[6.7 Les fichiers multimédias 29](#_Toc506248730)

[7 HtmlHelper et TagHelper 30](#_Toc506248731)

[7.1 Les HTML Helper 30](#_Toc506248732)

[7.2 Les Tag Helpers 32](#_Toc506248733)

[8 Les pages de disposition 35](#_Toc506248734)

[8.1 Contenu 36](#_Toc506248735)

[8.2 Référencement 37](#_Toc506248736)

[9 Les formulaires 38](#_Toc506248737)

[9.1 Le Tag Helper Form 38](#_Toc506248738)

[9.2 Attributs de l’élément <form> 39](#_Toc506248739)

[9.3 Tag Helpers de formulaires 39](#_Toc506248740)

[10 La validation 44](#_Toc506248741)

[10.1 Validation côté serveur 44](#_Toc506248742)

[10.2 Validation côté client 49](#_Toc506248743)

[11 Les ViewModels 50](#_Toc506248744)

[11.1 Principe 50](#_Toc506248745)

[11.2 Mise en œuvre 51](#_Toc506248746)

[11.3 Agrégation 53](#_Toc506248747)

[12 Tri, pagination, filtrage 55](#_Toc506248748)

[12.1 Tri 55](#_Toc506248749)

[12.2 Pagination 57](#_Toc506248750)

[12.3 Filtrage 61](#_Toc506248751)

[13 Le routage 63](#_Toc506248752)

[13.1 Routage conventionnel 63](#_Toc506248753)

[13.2 Routage par attributs 64](#_Toc506248754)

[14 Les middlewares 65](#_Toc506248755)

[15 La sécurité 66](#_Toc506248756)

[15.1 Identification, authentification et autorisation 66](#_Toc506248757)

[15.2 ASP.NET Core Identity 67](#_Toc506248758)

[16 API Web 73](#_Toc506248759)

[16.1 Création d’une API Web 74](#_Toc506248760)

[16.2 Actions 74](#_Toc506248761)

[16.3 Réponse http 75](#_Toc506248762)

[16.4 Test de l’API avec PostMan 75](#_Toc506248763)

[16.5 Création d’une application cliente 78](#_Toc506248764)

[17 Références 80](#_Toc506248765)

# Notions de base

## Application web

Une application web est une application hébergée sur un serveur, et utilisée au travers d’un navigateur.

L’utilisateur envoie des requêtes grâce au navigateur, et obtient des réponses de l’application sous forme de pages HTML affichées par le navigateur. Ces pages peuvent contenir du code JavaScript de façon à les rendre dynamiques et interactives (ex : validation des saisies, calculs, UI dynamique…etc.)

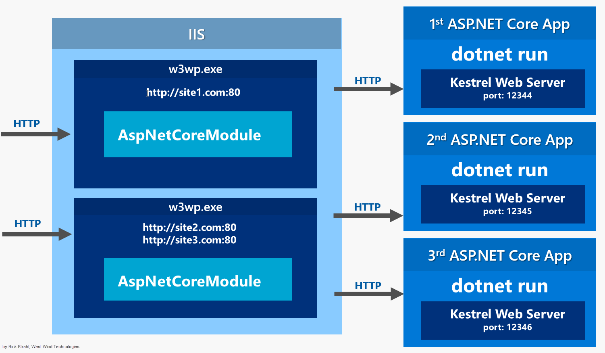
Une application web se compose donc d’au moins deux « couches » : la couche cliente, qui s’exécute dans le navigateur web, et la couche serveur, qui s’exécute sur la machine serveur.

## Serveur web

L’application web côté serveur est elle-même hébergée par un programme particulier qu’on appelle « Serveur Web ». Les serveurs web les plus connus sont Microsoft IIS (Internet Information Service), Apache, Tomcat et Nginx.

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

Tandis que les application ASP.net classiques étaient hébergées par le serveur web IIS, les applications ASP.Net Core sont désormais auto-hébergées par une instance de serveur web basic nommé **Kestrel**, qui fait partie intégrante de .Net Core. Elles sont cependant par défaut encore exposées en façade par un serveur IIS, qui apporte des fonctionnalités supplémentaires.



## Différence entre un service web et une application web

Une application web est destinée à être utilisée directement par un utilisateur final, à partir de son navigateur web. Elle fournit une interface visuelle en HTML pour afficher des informations et permettre à l’utilisateur d’interagir avec l’application.

Un service web est destiné à être utilisé par une autre application. Il peut recevoir et renvoyer des données au format texte (généralement xml ou json), mais ne construit pas de page html.

Une application web peut inclure des services web. Des services web gratuits peuvent être utilisés par exemple sur [cette page](http://www.webservicex.net/new/Home/Index). Application et service web utilisent tous deux le protocole http pour communiquer avec la couche cliente.

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

### Les types de requêtes

La couche 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
* Peuvent être mises en caches
* Les paramètres restent dans l’historique du navigateur
* Peuvent être mises en favori
* Ne doivent pas être utilisées avec des données sensibles (mots de passe ou autres)
* Sont restreintes par la longueur de l’url à 2048 caractères
* Ne peuvent véhiculer que des données ASCII

**Requêtes POST**

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

**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 peut spécifier 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.

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.

Une opération de suppression peut nécessiter deux actions, l’une en réponse GET pour vérifier que l’enregistrement à supprimer existe, l’autre en réponse POST, pour réaliser la suppression.

# ASP.Net MVC

## ASP.Net

ASP.Net est la technologie de Microsoft pour concevoir des applications web. Elle comprend trois frameworks, qui diffèrent surtout par leur style de développement, et non par leur potentiel :

* **Web Forms** : framework de développement rapide proche de WinForms
* **MVC** : permet une bonne séparation de la logique applicative et de l’interface utilisateur (UI), ce qui rend l’application plus facilement testable. Ce framework est notamment bien adapté aux applis mobiles, et aux applis à page unique (Single Page Applications)
* **Razor Pages** (anciennement **Web Pages**) : modèle mélangeant le html et le code serveur

MVC permet de mieux architecturer l’application, mais demande plus d'expertise. Les trois frameworks peuvent être utilisés ensemble dans la même appli.

## Le modèle MVC



Les flèches représentent le sens des dépendances, ce qui revient à définir qui peut utiliser qui. Par exemple, le contrôleur peut utiliser le modèle et les vues.

**Le modèle** représente l'état de l'appli et la logique applicative. Il fournit les données pour remplir les vues créées par le contrôleur. Il y a plusieurs types de modèles, ayant chacun une organisation propre.

**Les vues** affichent l'interface utilisateur, avec les données issues du modèle. Elles doivent contenir le minimum d'intelligence. Elles sont générées au moyen de templates.

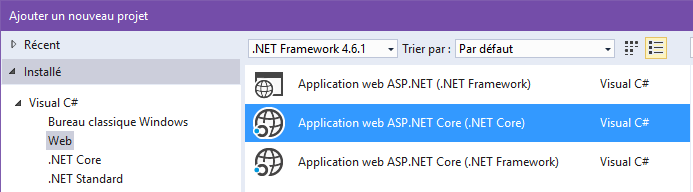
**Les contrôleurs** gèrent les interactions utilisateurs (les requêtes http), récupèrent les données du modèle et utilisent les templates de vue pour renvoyer une réponse. Les vues ne font qu'afficher les infos. Ce sont les contrôleurs qui gèrent l'interaction avec l'utilisateur

ASP.Net Core MVC est basé sur des patterns, et utilise les derniers standards du web. Il inclut :

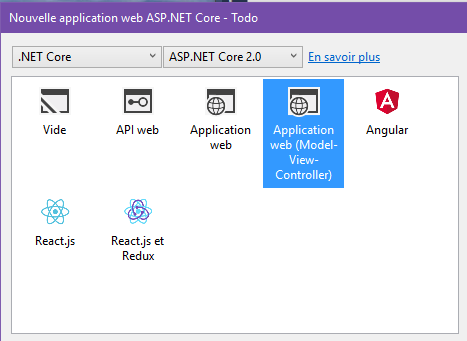
* Le routage
* La liaison de modèle
* La validation de modèle
* L'injection de dépendances
* Les filtres
* Les aires
* Les Web APIs
* La testabilité
* Le moteur visuel Razor (à partir de ASP.Net Core 2.0)
* Les vues fortement typées
* Les tag helpers
* Les composants de vues

# Structure d’un projet

Visual Studio fournit un modèle de projet pour une application ASP.Net Core :



On choisit ensuite le type d’architecture de l’application :



Visual Studio génère automatiquement une application déjà fonctionnelle avec une page d’accueil, une page A propos et une page Contact.

La structure du projet avec l’architecture MVC ressemble à ce qui suit :

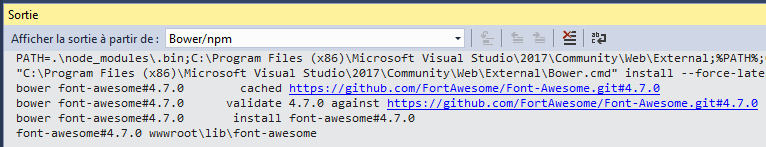
|  |  |
| --- | --- |
|  | Dépendances :   * Bower : bibliothèques JavaScript gérées par Bower et référencées dans le fichier bower.json * NuGet : packages gérés par NuGet, tels que ASP.Net Core et Entity Framework Core * SDK : dll du kit de développement pour applis .Net Core   wwwroot contient les ressources statiques utilisées par la couche cliente : fichiers html, css, JavaScript et images  Les contrôleurs, les modèles et les vues sont rangés dans les répertoires Controllers, Models et Views  Les vues sont rangées dans des sous-répertoires correspondant aux noms des différents contrôleurs  Appsettings.json est le fichier de config principal de l’appli  Bower.json ressence les fichiers gérés par le gestionnaire de packages Bower  Bundleconfig.json est le fichier de config du regroupement de fichiers et de la minification (techniques qui améliorent les perfs des requêtes http) |

## Gestion des packages

### Partie cliente de l’application

**Bower** est un gestionnaire de packages open source, permettant de gérer des packages de fichiers pour la partie cliente de l’application (html, css, JavaScript…). Les packages utilisés par l’application sont recensés dans le fichier bower.json, et affichés sous le nœud « Dépendances \ Bower ». Par défaut, l’application utilise les packages Bootstrap et jQuery.

Pour ajouter de nouveaux packages, il suffit d’éditer le fichier bower.json. L’IntelliSense propose une liste de tous les packages disponibles. Lorsqu’on enregistre le fichier, les packages ajoutés sont automatiquement téléchargés dans le répertoire wwwroot\lib, et leurs références apparaissent dans le nœud « Dépendances \ Bower ». On peut suivre le déroulement de l’opération dans le panneau de sortie de Visual Studio :



Le répertoire **wwwroot** contient toutes les ressources statiques du projet utilisées par la couche cliente, c’est-à-dire les fichiers qui sont envoyés par l’appli au navigateur de l’utilisateur. Il s’agit de fichiers html, css, JavaScript et images accessibles publiquement, c’est-à-dire non soumis à autorisation.  
Les fichiers de ce répertoire sont accessibles dans le code, comme si wwwroot était la racine du projet. Pour qu’ils puissent être envoyés au client par le serveur, il faut ajouter le code suivant dans la méthode Configure de la classe Startup :

app.UseStaticFiles();

### Partie serveur de l’application

La partie serveur de l’application utilise de nombreux packages binaire, notamment ceux d’ASP.Net Core, et d’Entity Framework Core. Ces packages sont téléchargés par NuGet, le gestionnaire de packages de Visual Studio, accessible via le menu Outils.

Les packages NuGet sont stockés dans le répertoire global de packages, dont le chemin par défaut est : C:\Users\xxxx\.nuget\packages

Le nœud SDK affiche les dll qui composent le kit de développement pour .Net Core

## Démarrage de l’application

### Point d’entrée

La fonction Main de la classe Program est le point d’entrée de l’application. Elle contient le code suivant :

public static void Main(string[] args)

{

   var host = WebHost.CreateDefaultBuilder(args)

            .UseStartup<Startup>()

            .UseApplicationInsights()

            .Build();

   host.Run();

}

Les différentes méthodes qui sont appelées ici renvoient toutes l’objet de type IWebHostBuilder qui les a appelées, de sorte qu’elles peuvent être chaînées. Elles ont respectivement pour rôle de :

* Allouer un serveur Kestrel qui sera exposé par IIS s’il est disponible
* Définir la classe Startup comme classe de démarrage de l’application
* Utiliser les services Application Insights pour mesurer et gérer les performances de l’application
* Construire le serveur web hôte avec la configuration définie par les méthodes précédentes

Une fois construit et configuré, le serveur hôte est exécuté via sa méthode Run.

La méthode Run instancie la classe **Startup** et appelle sa méthode ConfigureServices pour configurer les différents services de l’application. Les services sont des composants de l’application, tels que la gestion des données, l’identification, la gestion de la configuration, le modèle MVC…etc.

La liste des services est gérée par un conteneur IoC (Inversion of Control), qui se charge de leur instanciation dès qu’ils sont appelés.

La séquence de démarrage de l’application peut donc être représentée de façon simplifiée comme ceci :

Main 🡪 IWebHost.Run 🡪 Startup.ConfigureServices  
 🡪 Startup.Configure

### Lancement

Le serveur web hôte peut être lancé en ligne de commande, en se plaçant dans le répertoire du projet et en exécutant la commande « dotnet run » :

D:\Workspaces\Web\ASP.NET\CoursASPNet\Northwind2>dotnet run

Hosting environment: Production

Content root path: D:\Workspaces\Web\ASP.NET\CoursASPNet\Northwind2

Now listening on: http://localhost:5000

Application started. Press Ctrl+C to shut down.

Il suffit de saisir l’adresse indiquée par la console, dans la barre d’adresse d’un navigateur pour accéder à l’interface cliente de l’application.

Bien entendu, on peut aussi démarrer l’application directement depuis Visual Studio en mode debug en appuyant sur F5. Dans ce cas, Visual Studio démarre automatiquement une instance IIS Express pour exposer la partie serveur (celui-ci est visible dans la barre des tâches), et lance le navigateur à l’adresse de l’application. Celle-ci est du type <http://localhost:XXXXX>, où XXXXX est un numéro de port choisi aléatoirement.

**Astuce** : utiliser CTRL + F5 pour lancer l’appli en mode non debug. Si on modifie le code, il suffit d’enregistrer les modifications, et de rafraîchir la page dans le navigateur (avec F5 ou CTRL + F5) pour tester le résultat. Il n’est pas nécessaire de recompiler l’appli, ce qui est plus rapide.

## Configuration

L’application et ses services peut être configurée au moyen de différents fournisseurs :

* Des fichiers json, xml ou ini
* Des arguments de ligne de commande
* Des variables d’environnement
* …etc.

Les infos de config sont stockées sous forme de paires clé-valeur

Lorsqu'on crée une nouvelle application à l’aide du modèle de projet fourni par Visual Studio, celle-ci contient par défaut plusieurs fichiers json, le principal étant **appsettings.json** (remplaçant de web.config dans les versions antérieures d'ASP.Net)

Ce fichier contient les informations de configuration initiale de l'application. Le constructeur de la classe Startup charge ces informations dans une propriété en lecture seule nommée Configuration, afin de les rendre accessibles aux autres méthodes de la classe, en particulier ConfigureServices et Configure.

public Startup(IHostingEnvironment env)

{

   var builder = new ConfigurationBuilder()

       .SetBasePath(env.ContentRootPath)

       .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)

       .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)

       .AddEnvironmentVariables();

   Configuration = builder.Build();

}

Dans le fichier appsettings.json, les informations peuvent être stockées sous forme arborescente, et classées en sections.

Exemple :

{

   "Logging": {

      "IncludeScopes": false,

      "LogLevel": {

         "Default": "Warning"

      }

   },

   "ConnectionStrings": {

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

   }

}

Logging et ConnectionStrings sont des sections de même niveau

LogLevel est une sous-section de Logging

**Accès aux sections de paramètres depuis le code**

Dans les versions précédentes d’ASP.Net MVC, les paramètres étaient accessibles via une propriété statique ConfigurationManager.AppSettings. ASP.Net MVC Core met en œuvre une logique différente, basée sur l’injection de dépendance et le conteneur IoC, grâce à laquelle chaque contrôleur a accès uniquement aux sections de paramètres dont il a besoin. Voici comment l’implémenter :

Supposons que le fichier appsettings.json contienne la section suivante :

"AppSettings": {

   "ItemsPerPage": "20"

}

…et que cette section soit utile pour le contrôleur nommé EmployeesController.

Pour rendre cette section de paramètres accessible à la classe EmployeesController, il nous faut tout d’abord créer une classe qui représente la section :

public class AppSettings

{

   public int ItemsPerPage { get; set; }

}

…puis enregistrer une instance de cette classe dans le conteneur IoC :

public void ConfigureServices(IServiceCollection services)

{

   services.AddMvc();

   // Enregistre une instance de section de config dans le conteneur IoC

   services.Configure<AppSettings>(Configuration.GetSection("AppSettings"));

}

Pour pouvoir utiliser les valeurs de la section AppSettings dans un contrôleur particulier, on injecte une dépendance vers la classe correspondante dans le constructeur du contrôleur :

using Microsoft.Extensions.Options;

namespace Northwind2.Controllers

{

   public class EmployeesController : Controller

   {

      private readonly Northwind2Context \_context;

      private readonly AppSettings \_appSettings;

      // Contructeur avec injection de dépendance vers le DbContext  
 // et la section de paramètres AppSettings

      public EmployeesController(Northwind2Context context,  
 IOptions<AppSettings> appSettings)

      {

         \_context = context;

         \_appSettings = appSettings.Value;

      }

IOptions est une interface générique possédant une simple propriété Value du type spécifié en paramètre. Elle permet ainsi de passer n’importe quelle classe de paramètre au contrôleur.

Grâce à ce code, les actions de EmployeeController peuvent désormais utiliser \_appSettings.ItemsPerPage.

# Le modèle

La couche modèle de l’application est celle qui permet de dialoguer avec la base de données.

Le mot « modèle » désigne deux choses légèrement différentes, selon le contexte dans lequel il est utilisé :

* Quand on parle d’Entity Framework, il désigne l’ensemble des classes POCO qui représentent les tables de la base de données.
* Quand on parle du modèle utilisé par une vue, il désigne une classe particulière du modèle EF. En ce sens, l’application utilise plusieurs modèles.

## Création

Le modèle peut être implémenté de différentes façons. Si l’on utilise Entity Framework Core dans l’approche Database First, l’utilitaire scaffold-dbcontext génère une classe dérivée de DbContext et un ensemble de classes POCO à partir des tables d’une base de données existante (cf. chapitre EF du cours ADO.Net)

Pour que le code généré s’intègre bien dans l’architecture de l’application, nous devons faire ce qui suit :

* Placer la chaine de connexion à la base dans un fichier de configuration, pour faciliter son accès
* Ajouter le contexte à la liste des services de l’application (dans son conteneur IoC), de façon à ce qu’il soit facilement utilisable dans l’application.
* Faire en sorte qu’il reçoive les options de configuration (contenant la chaîne de connexion) en paramètre de son constructeur

Pour cela :

**Dans la classe Northwind2Context** :

* Ajouter le constructeur suivant, afin que le contexte puisse recevoir les options de configuration :

public Northwind2Context(DbContextOptions<Northwind2Context> options) :  
  base(options) { }

* Supprimer la méthode OnConfiguring en ayant au préalable copié la chaîne de connexion à la base

**Dans le fichier appsettings.json**, ajouter un paramètre, et y copier la chaîne de connexion :

"ConnectionStrings": {

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

}

**Dans la classe Startup**, ajouter le code suivant pour enregistrer le contexte dans la liste des services, et lui passer la chaîne de connexion :

using Northwind2.Models;

using Microsoft.EntityFrameworkCore;

...

public void ConfigureServices(IServiceCollection services)

{

   // Add framework services.

   services.AddMvc();

   var connect = Configuration.GetConnectionString("Northwind2Connect");

   services.AddDbContext<Northwind2Context>(opt => opt.UseSqlServer(connect));

}

Nous avons ainsi déporté la chaîne de connexion dans le fichier de paramétrage général de l’appli, et passé ce paramétrage au contexte, via un constructeur spécifique.

## Affichage et validation

Les propriétés des classes de modèle peuvent être décorées d’attributs, de façon à modifier leur affichage ou à leur appliquer des règles de validation à la saisie.

Exemple :

[Range(1, 100), DataType(DataType.Currency)]

public decimal Price { get; set; }

Dans cet exemple, on affecte à la propriété Price un type monétaire, pour qu’elle s’affiche sous une forme adéquate, et on contraint sa valeur à l’intervalle [0 – 100]

Les attributs d’affichage et de validation seront abordés plus en détail dans les chapitres consacrés aux vues et à la validation.

**Règles de validation et contraintes sur la base**

/!\ Dans une approche code first avec nouvelle base de données, il est important de décrire les règles de validation sur le modèle avant de générer la base de données, car cela affecte les contraintes sur le schéma de la base.

Quelques exemples de correspondance entre les règles de validation sur les propriétés et les contraintes sur les champs de la table :

|  |  |
| --- | --- |
| **Propriété de l’entité** | **Champ dans la table correspondante** |
| string Libelle | Libelle(nvarchar(max), NULL) |
| [Required, MaxLenght(100))  string Libelle | Libelle(nvarchar(250), non NULL) |
| DateTime DateNais | DateNais(datetime2(7), non NULL) |
| DateTime? DateNais | DateNais(datetime2(7), NULL) |
| … |  |

## Modèles pour vues

Comme nous le verrons dans le chapitre consacré au ViewModels, il est parfois nécessaire de créer des classes de modèles nommées « modèles pour vues » ou « entités visuelles », dédiées à l’affichage et la saisie d’informations dans les vues. Ces entités ne correspondent pas directement à des tables de la base de données.

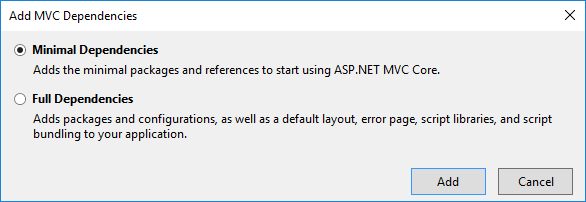
# Les contrôleurs

Un contrôleur est une classe chargée de répondre à des requêtes http. Elle utilise la couche modèle pour récupérer ou stocker des données dans la base de données.

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.

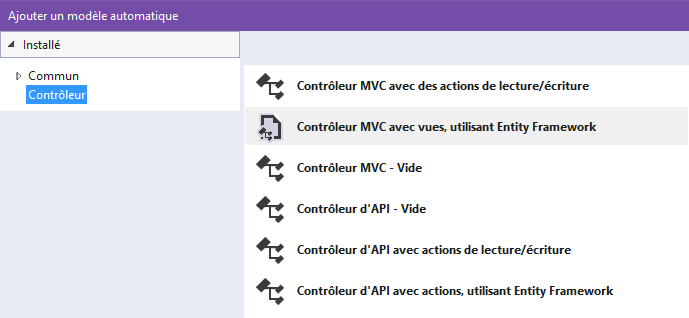
## Création d’un contrôleur

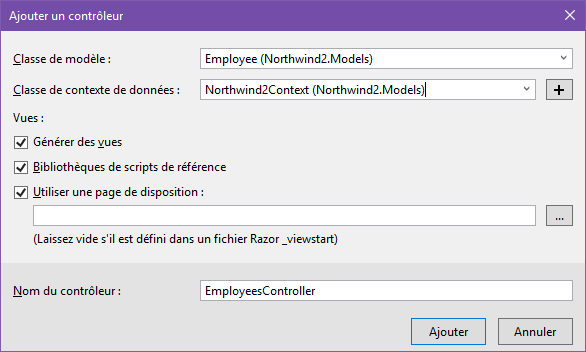
Lors du premier ajout d’un contrôleur au projet, via le menu « Ajouter \ Contrôleur », la boîte de dialogue suivante s’ouvre :



Elle permet d’ajouter au projet toutes les références nécessaires pour utiliser ASP.Net MVC Core

Il faut de nouveau cliquer sur « Ajouter \ Contrôleur » pour réellement créer le contrôleur. Les fenêtres suivantes s’affichent alors pour choisir et configurer le contrôleur :

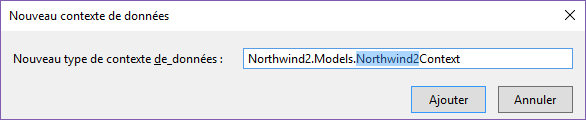




La première liste déroulante contient les classes susceptibles d’être utilisées comme modèles.

La seconde permet de sélectionner la classe de contexte, pour l’accès aux données.

NB/ Le bouton + permet d’ouvrir la boîte de dialogue ci-dessous dans laquelle, on peut spécifier le nom d’un nouveau contexte à créer. Ceci est utile dans une approche EF Code First avec nouvelle base de données.



La première case à cocher permet de générer ou non des vues pour afficher, créer, supprimer et éditer les entités manipulées par le contexte.

La seconde case à cocher permet d’ajouter ou non des références vers les bibliothèques JavaScript jQuery, qui peuvent être utiles aux vues, par exemple pour faire de la validation des saisies côté client.

Nous parlerons des pages de disposition plus loin dans ce cours.

Lorsqu’on clique sur le bouton Ajouter, Visual Studio crée automatiquement :

* Une classe de contexte dérivant de DbContext, dans le répertoire Data. C’est cette classe qui permet au contrôleur d’interagir avec le modèle de données.
* Un contrôleur, qui exploite le contexte
* Des templates de vues Razor (nous verrons ce concept plus loin)

Le contexte doit être ajouté à la liste des services de l’application (cf. paragraphe sur la création du modèle vu précédemment).

Les services sont instanciés automatiquement par un conteneur IoC (Inversion of Control) lorsqu’on en a besoin. Il s’agit d’un mécanisme intégré dans ASP.Net Core.

Ainsi, il suffit de définir un paramètre de type contexte dans le constructeur du contrôleur (c’est ce qu’on appelle une injection de dépendance), pour qu’une instance de ce contexte soit automatiquement fournie par le conteneur IoC au moment de la création du contrôleur. C’est ce qui est fait dans le code généré par l’assistant :

public class EmployeesController : Controller

{

    private readonly Northwind2Context \_context;

    public EmployeesController(Northwind2Context context)

    {

        \_context = context;

    }

}

## Points de terminaison et actions

Dans une classe de contrôleur, chaque méthode publique représente un point de terminaison (endpoint), auquel on s’adresse via une url de la forme : http://localhost:12345/controleur/action

L’url combine :

* Le protocole
* L’adresse de l’application sur le serveur web (incluant le port)
* Une action à exécuter, qui est une méthode d’un contrôleur

Exemple : en allant à l’url <http://localhost:12345/Home/Contact>, on appelle la méthode Contact du contrôleur HomeController.

/!\ Par convention, les noms de classes de contrôleurs doivent se terminer par « Controller », mais il ne faut pas mettre ce suffixe dans les url.

Un contrôleur doit fournir une méthode Index(), qui est appelée par défaut lorsqu’on ne spécifie pas de nom de méthode dans l’url. Ainsi l’url [http://localhost:12345/Home](http://localhost:12345/Home/Contact) appelle la méthode Index du contrôleur HomeController.

## Les attributs d’actions

Une action peut être décorée d’un attribut permettant de spécifier le type de requête qu’elle prend en charge. Ainsi, à chaque type de requête http (GET, POST, PUT, DELETE) correspond un attribut : HttpGet, HttpPost, HttpPut, HttpDelete.

Par défaut, une action décorée d’aucun attribut n’accepte que les requêtes GET

L’attribut ActionName permet de préciser un alias pour l’action, c’est-à-dire que l’action devra être appelée côté client par le nom d’alias spécifié.  
/!\ Le nom de la vue renvoyée comme réponse doit alors être précisé explicitement.

Exemple :

[ActionName("Aboutus")]

public IActionResult About()

{

    ViewData["Message"] = "A propos de nous";

    return View("About");

}

[HttpPost]

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

[HttpPost, ActionName("Delete")]

public async Task<IActionResult> DeleteConfirmed(int id) { ... }

L’action About ne sera déclenchée que si on utilise le nom « Aboutus » dans l’url de la requête. Le nom de la vue renvoyée est spécifié explicitement, sinon ASP.Net rechercherait une vue nommée « Aboutus » qui n’existe pas.

L’attribut HttpPost sur les actions Edit et DeleteConfirmed fait que ces actions ne peuvent être déclenchées que par des requêtes POST. La seconde doit être appelée par le nom « Delete » dans l’url.

Sur l’action DeleteConfirmed, on aurait pu mettre l’attribut HttpDelete au lieu de HttpPost. Avec une requête POST, l’Id de l’élément à supprimer doit être passé dans le corps de la requête, tandis qu’avec une requête DELETE, il doit être passé dans l’url.

## Liaison de modèle

Lorsqu’une requête http appelle une action d’un contrôleur, elle peut bien entendu lui transmettre des données. [Le Model Binding](https://docs.microsoft.com/fr-fr/aspnet/core/mvc/models/model-binding) (qu’on peut traduire par « liaison de modèle ») est ce qui permet de mapper ces données aux paramètres des méthodes du contrôleur.

NB/ On comprend mieux l’expression « Liaison de modèle » si l’on considère que les paramètres des actions du contrôleur sont généralement des entités de modèle.

Les données peuvent être véhiculées via l’url, ou via le corps de la requête. Elles peuvent être de type simple ou complexe.

Quand une requête http est reçue par l’application, elle est routée vers une méthode spéciale du contrôleur, qui détermine quelle méthode du contrôleur doit être exécutée, et qui relie les données transmises par la requête http aux paramètres de la méthode, en se basant sur leurs noms.

Les données transmises par la requête http peuvent être stockées sous forme de paires (nom – valeur) à différents endroits, et sont recherchées par le Model Binding dans l’ordre suivant :

* Dans le corps de la requête, pour des données de formulaire envoyées par une requête de type POST
* Dans des segments d’url, à la suite du nom de l’action
* Dans la partie paramètres (requête) de l’url, après le point d’interrogation

Nous allons examiner en détail ces trois possibilités.

### Corps de la requête

Supposons par exemple que la page de contact de l’application affiche un formulaire, dans lequel l’utilisateur laisse ses coordonnées. La soumission du formulaire par l’url suivante :

http://localhost:52902/Home/Contact

… appelle la méthode suivante :

public IActionResult Contact(Address adresse)

{

   ViewData["Message"] = "Tel : " + adresse.Phone;

   return View();

}

Les coordonnées de l’utilisateur sont passées dans le corps de la requête http, sous forme d’objet de type Address sérialisé.

### Segments d’URL

Le format global de l’url est défini par le **modèle de route**, que nous verrons en détail dans un prochain chapitre. Lorsqu’on crée un nouveau projet ASP.Net Core, un modèle de route par défaut est défini de la façon suivante dans la classe Startup :

app.UseMvc(routes =>

{

    routes.MapRoute(

        name: "default",

        template: "{controller=Home}/{action=Index}/{id?}");

});

Les segments d’url sont séparés par des « / ».

Dans le modèle de route ci-dessus :

* Les url peuvent contenir des segments pour le contrôleur, l’action et un identifiant facultatif.
* Si le contrôleur n’est pas spécifié dans l’url, c’est HomeController qui sera utilisé par défaut.
* Si l’action n’est pas spécifiée dans l’url, c’est Index() qui sera appelée.

Exemple : si un contrôleur nommé HomeController possède la méthode suivante :

public IActionResult Contact(int id)

{

   ViewData["Message"] = "Id = " + id.ToString();

   return View();

}

… alors on peut appeler cette action par une url ressemblant à ceci :

http://localhost:52902/Home/Contact/5

Si on ne met pas la valeur 5 à la fin, c’est la valeur par défaut du type du paramètre qui sera utilisée (0).

### Requête dans l’url

Si une action prend quelques valeurs simples et non confidentielles en paramètres, on peut les passer par l’url, plutôt que par le corps de la requête http. Cette possibilité peut être utilisée par exemple pour récupérer les valeurs saisies par l’utilisateur dans une ou plusieurs zones de recherche de la page.

Exemple : supposons que notre application permette de faire une recherche de produit par catégorie + nom. Cette requête est traitée par l’action suivante :

public IActionResult Search(string categorie, string nom)

{

   ViewData["Produit"] = "Catégorie : " + categorie + ", Nom : " + nom;

   return View();

}

La recherche de l’utilisateur peut être transmise via une url du style :

http://localhost:52902/Home/Search?categorie=pain&nom=baguette%20campagne

Le point d’interrogation permet de spécifier que tout ce qui suit est un ensemble de paramètres.

Les paramètres de la requête sont des paires « nom=valeur », séparées par le caractère « & »

Si une valeur contient des caractères spéciaux, ils doivent être encodés en ASCII au format « % + 2 caractères hexadécimaux ». Par exemple la valeur hexadécimale 20 correspond au caractère espace. Cet encodage est géré automatiquement par les templates de vues.

Remarque : la valeur d’un segment de route peut aussi être passée dans la partie paramètre. Ainsi, l’id passé à la méthode Contact de l’exemple précédent peut aussi être passé comme ceci :

http://localhost:52902/Home/Contact?id=5

# Les vues

## Concepts

Les vues sont des fichiers à l’extension **cshtml**, placés dans des sous-dossiers du dossier **Views**.

Chaque action d’un contrôleur répond à une demande du client en lui renvoyant une page html construite par le **moteur de vue Razor**, à partir d’une **vue** et d’une page de disposition. On peut donc considérer les vues comme des descriptions génériques de réponses html.

La réponse html est générée en appelant la méthode View du contrôleur. Celle-ci permet de spécifier la vue à utiliser et les données à transmettre à la vue (ex : une liste de personnes, la fiche d’une personne…).

Exemple de code d’une action qui passe un employé à la vue :

public async Task<IActionResult> Details(int? id)

{

    if (id == null) return NotFound();

    var employee = \_context.Employee.SingleOrDefault(m => m.EmployeeId == id);

    if (employee == null) return NotFound();

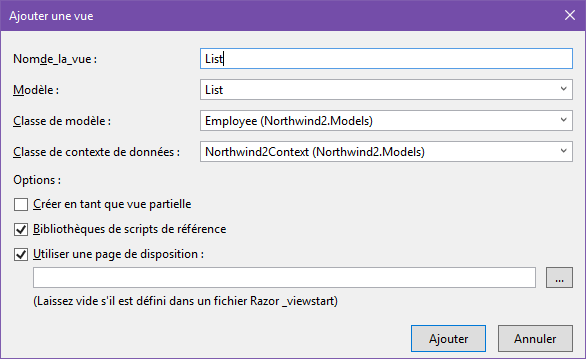
    return View(employee);

}

Nous verrons plus loin que les conventions de nommage suffisent à MVC pour déterminer quelle vue utiliser lors de l’appel à View.

## Création d’une vue

Les vues peuvent être générées automatiquement en même temps que le contrôleur, si on coche la case « Générer des vues » dans la boîte de dialogue d’ajout d’un nouveau contrôleur. Mais on peut aussi ajouter une nouvelle vue après coup, en faisant un clic droit sur le sous-répertoire correspondant au contrôleur souhaité, et en cliquant sur « Ajouter \ Vue », ce qui ouvre la boîte de dialogue suivante :



Voici la description des zones de saisie de cette boîte de dialogue :

**Modèle**: liste déroulante permettant de choisir parmi cinq templates de vues prédéfinis pour la création, l’affichage du détail, l’édition, la suppression ou l’affichage en liste d’une classe de modèle. Un choix « Empty » est également disponible si l’on préfère créer une vue vide.

NB/ Ici, le mot « Modèle » ne désigne donc pas une entité de données, mais un modèle de vue. Il correspond au mot anglais « template ».

**Classe de modèle** : permet de spécifier quel type de données la vue gérera, que ce soit en saisie ou en affichage.

**Classe de contexte de données** : le fait de spécifier le contexte de données permet à Visual Studio d’ajouter automatiquement un DbSet dans ce contexte pour la classe de modèle sélectionnée.

NB/ Les listes déroulantes « Classe de modèle » et « Classe de contexte » ne sont activées que si on sélectionne un templates autre que « Empty » dans la liste déroulante « Modèle ».

Description des options :

**Créer en tant que vue partielle** : les vues partielles sont des morceaux de vues, qu’on peut intégrer à l’intérieur d’autre vues. Elles servent à découper les vues complexes ou à partager des morceaux de vues entre plusieurs vues. Dans ce cas, elles peuvent être placées dans le répertoire Shared afin de faciliter le partage. Cf. [doc Microsoft](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/partial) pour plus de détail sur le partage de données entre une vue partielle et sa vue parente. Pour intégrer une vue partielle dans une vue parente, placer le code suivant à l’endroit souhaité :

@Html.Partial("VuePartielle")

**Bibliothèque de scripts de référence** : ajoute des références à des bibliothèques JavaScript pour la validation des saisies.

**Utiliser une page de disposition** : permet de préciser le nom d’une page de disposition. Par défaut, l’application contient un fichier \_ViewStart.cshtml qui définit la page \_layout.cshtml comme page de disposition, et qui est exécuté avant l’affichage de chaque vue, sauf des vues partielles.

## Rattachement aux actions des contrôleurs

**Conventions**

Chaque vue est automatiquement rattachée à une action de contrôleur par les conventions suivantes :

* La vue est placée dans un sous-dossier de même nom que le contrôleur (sans le suffixe « Controller »)
* La vue a le même nom que l’action du contrôleur

Voici un extrait de la structure d’une solution d’application ASP.Net MVC :

|  |  |
| --- | --- |
|  | Le contrôleur EmployeesController est associé aux vues du dossier Views\Employees  Le contrôleur HomeController est associé aux vues du dossier Views\Home  Les 5 vues ci-contre sont rattachées au contrôleur EmployeesController, car elles sont placées dans le sous-dossier Employees.    Les 3 vues ci-contre sont rattachées au contrôleur HomeController, car elles sont placées dans le sous-dossier Home. |

**Dossier Shared**

Si une vue n’est pas présente dans le sous-dossier correspondant au contrôleur, MVC la recherche dans le sous-dossier Shared. Ce dernier contient également d’autres éléments partagés que nous verrons plus loin.

Ainsi, si un contrôleur nommé EmployeesController contient une action nommée Create avec le code vu plus haut, MVC recherchera par défaut la vue Views/Employees/Create.cshtml, puis s’il ne la trouve pas, la vue Views/Shared/Create.cshtml.

**Spécification explicite de la vue**

La recherche de la vue basée sur les conventions peut toutefois être contournée si besoin. La méthode View possède en effet plusieurs surcharges, dont l’une d’elles permet de spécifier explicitement la vue à utiliser, de façon relative ou absolue :

// Spécification de la vue de façon relative

return View("Details");

// Spécification de la vue de façon absolue

return View("~/Views/Details.cshtml");

## Passage des données du contrôleur à la vue

Dans le chapitre consacré aux contrôleurs, nous avons vu comment les informations fournies par l’utilisateur peuvent être transmises aux actions des contrôleurs par le mécanisme de Model Binding.

Nous allons voir maintenant le mécanisme inverse, c’est à dire comment passer des données d’une action du contrôleur à sa vue, pour qu’elle les affiche. Ces données peuvent être fournies par un contexte de données, ou en dur dans l’action du contrôleur.

NB/ Pour maintenir une architecture propre, les vues doivent afficher/saisir les données, mais ne pas contenir de règles de gestion métier.

### Par la méthode View

Examinons le code généré automatiquement par Visual Studio lors de la création du contrôleur EmployeeController, pour la récupération et l’affichage du détail d’un employé :

public async Task<IActionResult> Details(int? id)

{

    if (id == null) return NotFound();

    var employee = await \_context.Employee

         .SingleOrDefaultAsync(m => m.EmployeeId == id);

    if (employee == null) return NotFound();

    return View(employee);

}

Ce code récupère une instance de type Employee via un DbContext, et la passe en paramètre à la méthode View(). Cette instance est ainsi transmise à la vue Details, et lui sert de source de données.

Le paramètre de la méthode View est type Object, ce qui permet de passer à la vue n’importe quel type d’objet. Il existe également une autre surcharge qui prend en paramètre à la fois le chemin de la vue, et l’objet qui lui servira de source de données.

Dans la vue, pour accéder à l’objet passé par le contrôleur, il faut tout d’abord déclarer son type, c’est-à-dire le nom complet de sa classe. On utilise pour cela la directive **@model**.

L’objet en lui-même est accessible via la propriété **Model** de la vue.

**Exemple 1 : objet simple**

@model Northwind2.Models.Employee

<h2>Details</h2>

<div>

    <h4>Employee</h4>

    <hr />

    <dl class="dl-horizontal">

        <dt>Nom</dt>

        <dd>@Model.LastName</dd>

        <dt>Prénom</dt>

        <dd>@Model.FirstName</dd>

</dl>  
</div>  
<div>

    <a asp-action="Edit" asp-route-id="@Model.EmployeeId">Editer</a> |

    <a asp-action="Index">Retour à la liste</a>

</div>

Cette vue affiche les nom et prénom de l’employé que lui a passé le contrôleur.

La première ligne est la directive qui déclare le type d’objet utilisé comme source de données de la vue.

On utilise ensuite la propriété Model de la vue pour accéder à l’employé et à ses propriétés.

**Exemple 2 : collection d’objets**

Dans le code ci-dessous, la source de données de la vue est cette fois une collection d’employés. On affiche les employés dans un tableau en parcourant la collection avec une boucle foreach, et en construisant le tableau ligne à ligne.

@model IEnumerable<Northwind2.Models.Employee>

<h2>Employés</h2>

<table class="table">

    <thead>

        <!-- Ligne d'en-têtes -->

        <tr>

            <th>Nom</th>

            <th>Prénom</th>

        </tr>

    </thead>

    <tbody>

        <!-- Corps du tableau -->

        @foreach (var item in Model)

        {

            <tr>

                <td>@item.LastName</td>

                <td>@item.FirstName</td>

            </tr>

        }

    </tbody>

</table>

### Par les propriétés ViewData et ViewBag

ViewData et ViewBag sont deux propriétés de la classe Controller qui permettent de passer des données du contrôleur à la vue, sans passer par la méthode View.

**ViewData** est de type ViewDataDictionary, qui est un dictionnaire <string, object>. Elle permet donc de stocker de multiples valeurs sous forme d’objets identifiés par des chaînes de caractères.

/!\ Les valeurs stockées dans ces propriétés ne perdurent pas entre deux requêtes http successives, puisqu’un nouveau contrôleur est instancié à chaque requête.

L’exemple suivant illustre l’utilisation de ViewData pour gérer l’ordre de tri d’un tableau (colonne et sens). L’objectif est qu’à chaque clic sur un lien « Trier par nom », on inverse le sens de tri de la colonne LastName.

public async Task<IActionResult> Index(string sortOrder)

{

   // Récupération et tri des données selon l’ordre passé en paramètre

...

   // On renvoie à la vue l’ordre de tri inverse

   ViewData["TriParNom"] = sortOrder == "LastName" ? "LastName\_desc" : "LastName";

   // On renvoie une vue triée

   return View(await employés.AsNoTracking().ToListAsync());

}

Cette méthode de contrôleur reçoit en paramètre un ordre de tri. Elle trie les données et renvoie à la vue par l’intermédiaire de ViewData, une chaîne représentant l’ordre de tri inverse.

La vue contient le lien hypertexte suivant :

<a asp-action="Index" asp-route-sortOrder="@ViewData["TriParNom"]">  
Trier par nom</a>

Un clic sur ce lien génère une requête http de type Get avec un paramètre sortOrder, dont la valeur provient du dictionnaire ViewData.

Remarquons que les données stockées dans ViewData sont de type Object, ce qui peut nécessiter de faire du transtypage dans la vue. De plus, il n’y a pas de vérification de l’existence des clés utilisées dans les vues au moment de la compilation.

**ViewBag** ne fait qu’encapsuler ViewData sous une forme différente. Elle tire parti des propriétés dynamiques, introduites dans C# 4.0. Il s’agit de propriétés ajoutées sur un objet au moment de l’exécution, et qui n’ont pas besoin d’être définies au préalable. Ainsi, en reprenant l’exemple précédent, on pourrait écrire :

<a asp-action="Index" asp-route-sortOrder="@ViewBag.TriParNom">Trier par nom</a>

ViewBag peut être utilisé pour accéder aux valeurs de ViewData, et inversement, puisque les deux partagent la même collection de données.

NB/ Le dictionnaire ViewData peut aussi être utilisé pour passer des informations entre une vue et une page de disposition, comme nous le verrons plus loin.

## Formats d’affichage

On peut décorer les propriétés d’un modèle avec des attributs, de façon à donner des indications à la vue sur la façon d’afficher les données. Ils sont exploités par les Html Helpers et les Tag Helpers que nous verrons plus loin.

Ces attributs font partie de l’espace de noms [System.ComponentModel.DataAnnotations](https://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations(v=vs.110).aspx).

### Nom d’affichage et type de donnée

Exemple :

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

namespace Northwind2.Models

{

   public partial class Employee

   {

      ...

      [Display(Name = "Nom")]

      public string LastName { get; set; }

      [Display(Name = "Prénom")]

      public string FirstName { get; set; }

[Display(Name = "Date de naissance")]

[DataType(DataType.Date)]

public DateTime? BirthDate { get; set; }

[DataType(DataType.MultilineText)]

public string Notes { get; set; }

...

L’attribut Display permet de spécifier le libellé associé à un champ

L’attribut DataType permet d’adapter l’affichage du champ au type de la donnée. Exemples :

* DataType.Date permet d’afficher une date sans la partie heure (ce qui est le cas le plus courant)
* DataType.Url permet d’afficher une url sous forme de lien hypertexte cliquable
* DataType.Password permet d’afficher un mot de passe sous la forme « \*\*\*\*\*\* »
* DataType.MultilineText permet d’afficher la zone de saisie avec plusieurs lignes (si utilisation d’un HTML Helper uniquement)
* …

Il n’est nécessaire que lorsque qu’il ne peut pas être déduit automatiquement à partir du type de la propriété. Par exemple, il est nécessaire si on souhaite afficher une date sans la partie heure.

Ci-dessous des extraits de code html générés par la vue pour un propriété DateTime, avec et sans spécification du DataType sur l’entité :

<!-- Code html généré si on précise un attribut DataType de type Date -->

<input type="date" ... />

<!-- Code html généré si on ne précise aucun atribut DataType -->

<input type="datetime-local" ... />

Dans le second cas, la partie heure sera affichée.

### Chaînes de formats

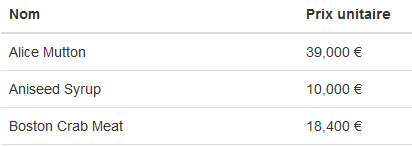
Les formats de certaines données, comme les date et prix peuvent être précisés au moyen d’un attribut DisplayFormat. Exemple :

[Display(Name = "Prix unitaire")]

[DisplayFormat(DataFormatString = "{0:C3}")]

public decimal UnitPrice { get; set; }

Le format appliqué ici permet d’afficher la valeur de UnitPrice en tant que prix, c’est à dire avec le symbole de la devise de la culture courante, et avec 3 chiffres après la virgules :



Par défaut, le format ne s’applique que sur les éléments non éditables (comme le tableau ci-dessus par exemple). Si on veut l’appliquer aussi sur des éléments éditables, comme les zones de saisies d’un formulaire, il faut affecter l’attribut ApplyFormatInEditMode à True :

[Display(Name = "Prix unitaire")]

[DisplayFormat(DataFormatString = "{0:C3}", ApplyFormatInEditMode = true)]

public decimal UnitPrice { get; set; }

/!\ Le fait d’appliquer le format également en édition peut poser des problèmes pour la validation de la saisie !

## Moteur de vue et syntaxe Razor

Le moteur de vue est le composant qui génère les réponses au format html à partir des vues.

Le code d’une vue est un mélange de code html et de code C# exécuté par le serveur. L’incorporation du C# au html est faite au moyen de la **syntaxe Razor**. Cette syntaxe est décrite en détails sur [cette page](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor) de docs.microsoft.com. Nous allons en voir ici les éléments essentiels.

### Transition html / C#

La transition entre le html et le C# est faite par le symbole arobase (@) :

<p>Date du jour : @DateTime.Today.ToLongDateString()</p>

On peut utiliser plusieurs expressions sur une même ligne :

<address>

    @Model.Address.Street<br />

    @Model.Address.City, @Model.Address.Country @Model.Address.PostalCode<br />

    <abbr title="Phone">P:</abbr>@Model.Address.Phone

</address>

NB/ Pour rappel, Model permet d’accéder à l’objet source de données de la vue.

### Blocs de code

Pour créer un bloc de code destiné à faire du traitement sans affichage, on utilise des accolades :

@{

   int? age = null;

   TimeSpan? ts = (DateTime.Today - Model.BirthDate);

   if (ts.HasValue)

   {

      age = (int)(ts.Value.Days / 365.25);

   }

}

<p><strong>Age : </strong>@age</p>

Les variables définies à l’intérieur d’un bloc de code sont utilisables à l’extérieur, comme la variable age dans cet exemple.

Le bloc de code C# peut lui-même inclure du code html, qui inclut des expressions Razor :

@{

   TimeSpan? ts = (DateTime.Today - Model.BirthDate);

   if (ts.HasValue)

   {

      int age = (int)(ts.Value.Days / 365.25);

      <p><strong>Age : </strong>@age</p>

   }

}

Si on veut afficher une valeur sans utiliser d’élément html spécifique, on peut utiliser la balise Razor <text> :

@{

   TimeSpan? ts = (DateTime.Today - Model.BirthDate);

   if (ts.HasValue)

   {

      int age = (int)(ts.Value.Days / 365.25);

      <text>Age : @age</text>

   }

}

On peut également utiliser la syntaxe « @ : », qui permet d’afficher le reste de la ligne comme du html :

@{

   TimeSpan? ts = (DateTime.Today - Model.BirthDate);

   if (ts.HasValue)

   {

      int age = (int)(ts.Value.Days / 365.25);

      @:Age : @age

   }

}

### Structures de contrôle

Les structures de contrôle (if…else, switch, for, try…catch, using…etc.) se comportent comme des blocs de code :

@if (age <= 25)

{

   <span>Moins de 25 ans</span>

}

else

{

   <span>Plus de 25 ans</span>

}

Notez qu’il n’est pas nécessaire de remettre un @ devant le else.

### Commentaires

Razor accepte les commentaires de type C# et html, et ajoute en plus la syntaxe @\*...\*@

@\* Calcul de l'age \*@

@{

   int? @age = null;

   TimeSpan? ts = (DateTime.Today - Model.BirthDate);

   if (ts.HasValue)

   {

      // Approximation ne tenant pas compte de la date réelle d'anniversaire

      age = (int)(ts.Value.Days / 365.25);

      <!-- Affichage de l'âge -->

      <span><strong>Age : </strong>@age</span>

   }

}

Dans le code html de la page finale, seuls les commentaires de type html sont présents.

### Directives

Une directive est représentée par un mot réservé après l’@. Quelques directives très utilisées :

**@model** permet de déclarer le type de l’objet qui sert de source de données à la vue

**@using** permet d’utiliser un espace de noms

**@section** est utilisée pour définir des sections dans la vue (exemple : une section de titre, une section de scripts…), qui sont référencées dans le fichier de mise en page \_Layout.cshtml (cf. plus loin)

## Les fichiers multimédias

### Affichage

**Image statique**

Pour afficher une image statique, c’est-à-dire stockée dans un sous-répertoire de wwwroot, on utilise l’élément <img> en affectant son attribut « src » avec le chemin relatif de l’image par rapport à wwwroot.

<img src="images/Northwind.jpg" alt="NorthWind" class="img-responsive" />

**Image issue de la base de données**

Une image stockée en base de données est représentée dans le modèle par une propriété binaire de type tableau de bytes :

public byte[] Photo { get; set; }

Pour afficher cette image, dans la vue, il faut convertir la propriété binaire en chaîne de caractères encodée en base 64 :

<img src="data:image;base64,@System.Convert.ToBase64String(item.Photo)"  
 width="80" height="80" />

**Vidéo**

Pour incorporer une vidéo dans la page, on utilise les éléments video et source :

<video width="600" controls>

   <source src="videos/Northwind.mp4" type="video/mp4">  
   <source src="videos/Northwind.webm" type="video/webm">

      Il est temps de mettre à jour votre navigateur !

</video>

A l’intérieure de <video>, on peut placer plusieurs éléments <source> pour référencer plusieurs formats de fichiers différents. Si un navigateur ne prend pas en charge le premier format, il utilise le second, et ainsi de suite. S’il ne prend en charge aucun des formats, le texte spécifié à la fin s’affiche.

### Téléchargement

Pour afficher un lien permettant à l’utilisateur de télécharger un fichier statique, on procède de la façon suivante :

On crée une action spécifique dans le contrôleur, ressemblant à ceci :

public FileResult Download()  
{  
 string fileName = "Northwind.mp4";  
 byte[] fileBytes = System.IO.File.ReadAllBytes($"wwwroot/videos/{fileName}");  
 return File(fileBytes, System.Net.Mime.MediaTypeNames.Application.Octet, fileName);  
}

Le type MediaTypeNames.Application.Octet est un type binaire qui n'est pas interprété, ce qui force le téléchargement plutôt que l’affichage dans le navigateur.

Dans la vue, on affiche un lien hypertexte pour le téléchargement :

<a href="Home/Download">Télécharger</a>

# HtmlHelper et TagHelper

## Les HTML Helper

Nous avons vu que la syntaxe Razor permet d’incorporer du code C# dans la vue pour construire le code html final. Les HTML Helpers permettent de simplifier grandement le code C#.

Les HTML Helpers sont des fonctions qui relient les propriétés de l’objet source de données aux éléments html dans les deux sens : pour afficher les valeurs des propriétés dans les éléments html, et inversement, pour affecter les propriétés à partir des valeurs des éléments html, lors de la soumission d’un formulaire.

**Exemple1** : extrait de vue qui affiche un tableau d’employés

<table class="table">

   <thead>  
 <!-- Ligne d'en-têtes -->

      <tr>

         <th>

            @Html.DisplayNameFor(model => model.LastName)

         </th>

         <th>

            @Html.DisplayNameFor(model => model.FirstName)

         </th>

         <th>

            @Html.DisplayNameFor(model => model.BirthDate)

         </th>

      </tr>

   </thead>

   <tbody>  
 <!-- Corps du tableau -->

      @foreach (var item in Model)

      {

         <tr>

            <td>

               @Html.DisplayFor(modelItem => item.LastName)

            </td>

            <td>

               @Html.DisplayFor(modelItem => item.FirstName)

            </td>

            <td>

               @Html.DisplayFor(modelItem => item.BirthDate)

            </td>

         </tr>

      }

   </tbody>

</table>

La fonction DisplayNameFor permet de récupérer la valeur de l’attribut [Display] associé à une propriété de l’objet source. Sans ce helper, il faudrait utiliser un gros bloc de code C# pour obtenir un résultat équivalent.

La fonction DisplayFor permet de récupérer la valeur d’une propriété de l’objet source. On pourrait a priori se passer de cette fonction en utilisant plus simplement la syntaxe suivante : @Model.LastName

L'intérêt de DisplayFor est qu'elle exploite automatiquement les formats d'affichage définis sur le modèle. Par exemple, si on définit le format suivant sur la date de naissance :

[DisplayFormat(DataFormatString = "{0:ddd dd MMM yyyy}")]

… DisplayFor affichera la date comme ceci : « dim. 31 déc. 2017 »

… alors que @Model.BirthDate l’affichera selon le format par défaut : « 31/12/17 00:00:00 »

Certains HTML Helpers génèrent le contenu d’éléments html (comme l’exemple précédent) ; d’autres génèrent les éléments html eux-mêmes. C’est le cas notamment pour des éléments de formulaires.

**Exemple 2** : formulaire de création d’un employé

<form asp-action="Create">

   <div class="form-horizontal">

      <h4>Personne</h4>

      <hr />

      @Html.ValidationSummary(true)

      <div class="form-group">

         @Html.LabelFor(m => m.LastName, "Nom",  
 new { @class="col-md-2 control-label" })

         <div class="col-md-10">

            @Html.EditorFor(model => model.LastName,  
 new { @class = "form-control" })

            @Html.ValidationMessageFor(model => model.LastName)

         </div>

      </div>

      <div class="form-group">

         @Html.LabelFor(m => m.BirthDate, "Date de naissance",  
 new { @class = "col-md-2 control-label" })

         <div class="col-md-10">

            @Html.EditorFor(model => model.BirthDate,  
 new { @class = "form-control" })

            @Html.ValidationMessageFor(model => model.BirthDate)

         </div>

      </div>

...

      <div class="form-group">

         <div class="col-md-offset-2 col-md-10">

            <input type="submit" value="Create" class="btn btn-default" />

         </div>

      </div>

   </div>

</form>

La méthode « LabelFor » génère un élément <label> associé à un élément <input>. On précise en paramètres :

* Le nom de la propriété du modèle qui sera saisie dans l’élément <input> associé au <label>
* Le texte du libellé
* Un tableau de chaînes représentant les attributs de l’élément. En l’occurrence, on l’utilise pour spécifier la classe CSS. Comme le mot « class » est un mot réservé du langage C#, il faut le faire précéder d’un @ pour qu’il soit interprété comme du html.

La méthode « EditorFor » génère un élément <input type=”text”>.

La méthode « ValidationMessage » génère un élément <span> permettant d’afficher le message d’erreur associé à la zone de saisie.

**La classe HtmlHelper et les classes d’extension**

Dans la syntaxe « @Html.DisplayFor » :

* Html est une propriété de la classe [WebViewPage](https://msdn.microsoft.com/en-us/library/system.web.mvc.webviewpage(v=vs.118).aspx), qui représente la vue. Cette propriété est de type [HtmlHelper](https://msdn.microsoft.com/en-us/library/system.web.mvc.htmlhelper(v=vs.118).aspx), qui est la classe qui prend en charge le rendu visuel de la vue.
* DisplayFor est une méthode d’extension de la classe HtmlHelper.

Toutes les méthodes helper sont en fait des méthodes d’extension de la classe HtmlHelper, fournies par des classes de l’espace de noms [Microsoft.AspNetCore.Mvc.Rendering](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.rendering?view=aspnetcore-2.0)

Beaucoup de méthodes helper prennent en paramètre une expression lambda pour désigner une propriété de l’objet source qu’elles utilisent.

## Les Tag Helpers

Les [Tags Helpers](https://docs.microsoft.com/fr-fr/aspnet/core/mvc/views/tag-helpers/intro) ont été introduits par ASP.NET Core. Ils jouent le même rôle que les HTML Helpers, avec un domaine d’utilisation plus restreint, mais une syntaxe beaucoup plus proche du html que du C#.

Supposons qu’on dispose d’un modèle possédant la propriété suivante :

[Display(Name = "Prénom")]

public string FirstName { get; set; }

Dans un formulaire, on souhaite pouvoir saisir la valeur de cette propriété dans une interface du type :



Cette interface est composée d’un libellé auquel est appliqué un style, et d’une zone de saisie. Le code html qui doit être généré à partir de la vue pour le libellé est le suivant :

<label for="FirstName" class="caption">Prénom</label>

Avec un HTML Helper, ceci peut être obtenu de la façon suivante :

@Html.LabelFor(m => m.FirstName, new { @class = "col-md-2 control-label" })

Avec un Tag Helper, on écrira plus simplement :

<label asp-for="FirstName" class="caption"></label>

On voit que le Tag Helper label ressemble fortement à l’élément html <label>, ce qui rend son écriture bien plus intuitive que celle du Html Helper. Le Tag Helper est cependant bien interprété côté serveur.

**Références aux propriétés de l’objet source**

L’attribut asp-for permet de spécifier le nom d’une propriété de l’objet source de données. Il n’est ainsi pas nécessaire d’utiliser une expression lambda. Lorsque l’attribut est interprété, sa valeur est en fait automatiquement traduite en expression lambda.

NB/ La valeur de l’attribut asp-for peut être spécifiée au moyen d’une expression Razor (précédée de @)

La valeur de l’attribut Display associé à la propriété spécifiée par asp-for est automatiquement récupérée pour générer le texte du libellé.

**Préfixe**

La ressemblance forte entre les Tag Helpers et le HTML rend plus difficile la distinction entre le code exécuté côté serveur (les Tag Helpers), et le code client (le html). Mais il est possible de rendre les choses plus claires en définissant un préfixe pour les Tag Helpers au début de la vue, comme le montre l’exemple suivant :

@tagHelperPrefix "hlp:"

...

<hlp:label asp-for="FirstName" class="caption"></hlp:label>

### Différences avec les HTML Helpers

Outre la syntaxe, la différence majeure entre les HTML Helpers et les Tag Helpers est que les premiers sont appelés comme des méthodes, ce qui permet de les utiliser sans générer d’élément html. C’est le cas par exemple de @Html.DisplayFor et @Html.DisplayNameFor, que nous avons rencontrés plus haut.

Les Tags Helpers n’offrent pas cette possibilité. C’est pourquoi seuls les HTML Helpers qui génèrent des éléments html ont des équivalents en Tag Helpers.

### Implémentation et référencement

Les Tag Helpers sont implémentés par des classes xxxTagHelper, où xxx représente généralement le nom de l’élément html généré. La liste de toutes les classes de Tag Helpers fournis nativement par Microsoft peut être consultée sur [cette page](https://docs.microsoft.com/en-us/aspnet/core/api/microsoft.aspnetcore.mvc.taghelpers).

Pour la mise en œuvre des Tag Helpers utilisés dans les formulaires, on pourra se référer à [cette page](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms).

Les Tag Helpers sont référencés dans le fichier \_ViewImports.chtml, dont le contenu sera importé dans toutes les vues. Par défaut, lors de la création d’une nouvelle application MVC, ce fichier contient la ligne suivante, qui permet de référencer les Tag Helpers fournis par Microsoft :

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

On peut ajouter des références vers d’autres assemblies issues de packages NuGet ou de notre propre solution.

### Quelques exemples

**Lien hypertexte**

L’exemple ci-dessous montre comment générer un lien avec le Tag Helper Anchor :

<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>

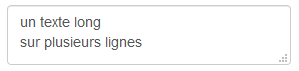
Le code html généré est le suivant :

<a href="/Employees/Edit/4">Edit</a>

Edit est le nom d’une action du contrôleur Employees. L’id de l’item courant lui est passé en paramètre grâce à l’attribut asp-route-id. De façon générale, asp-route-\* permet de passer une valeur en tant qu’élément de route.

**Texte multilignes**

Un formulaire peut contenir des zones de saisies multilignes, afin de faciliter la saisie de textes longs (typiquement des commentaires, messages, notes…) :



Ce type de zone de saisie correspond à l’élément html <textarea>, qui peut être généré par le Tag Helper de même nom :

<**textarea** **asp-for**="Notes" class="form-control"></**textarea**>

Pour que les retours à la ligne soient bien pris en compte dans la propriété du modèle, celle-ci doit être décorée d’un attribut indiquant qu'il s'agit d'un texte multilignes :

[DataType(DataType.MultilineText)]  
public string Notes { get; set; }

/!\ Lorsqu’on génère la vue automatiquement avec Visual Studio, c’est un Tag Helper Input qui est généré à la place du Textarea, même si le type MultilineText est spécifié sur l’entité. C’est un bug d’ASP.Net MVC…

Pour que le texte s’affiche également sur plusieurs lignes dans les vues en lecture seule (ex : vue de détail), il faut utiliser une classe de style CSS. Pour cela :

* Dans le fichier wwwroot\css\site.css (qui est référencé dans la page de disposition par défaut), ajouter la classe de style suivante :

/\* Pour afficher du texte sur plusieurs lignes \*/

.multiline {

    white-space: pre-wrap;

}

* Dans la vue, utiliser le code suivant :

<dd>

    <div class="multiline">

        @Html.DisplayFor(model => model.Notes)

    </div>

</dd>

**Eléments multimédia**

<!-- Image -->

<hlp:img src="~/images/Northwind.jpg" alt="North Wind" class="img-responsive" />

<!-- Vidéo -->

<!-- Le Tag Helper ajoute l'attribut poster qui permet d’afficher une image

à la place de la vidéo tant qu’elle n’est pas lancée -->

<hlp:video poster="~/images/Northwind.jpg" width="600" controls>

    <hlp:source src="~/videos/Northwind.mp4" type="video/mp4">

        Il est temps de mettre à jour votre navigateur !

</hlp:video>

/!\ Les chemins d’accès aux fichiers doivent utilisent le ~ pour faire référence au dossier wwwroot.

**Formulaire**

L’exemple suivant reprend le formulaire vu dans le paragraphe sur les HTML Helpers, en utilisant des Tag Helpers :

<form asp-action="Create">

    <div class="form-horizontal">

        <h4>Personne</h4>

        <hr />

        <div asp-validation-summary="ModelOnly" class="text-danger"></div>

        <div class="form-group">

            <label asp-for="LastName" class="col-md-2 control-label"></label>

            <div class="col-md-10">

                <input asp-for="LastName" class="form-control" />

                <span asp-validation-for="LastName" class="text-danger"></span>

            </div>

        </div>

        <div class="form-group">

            <label asp-for="BirthDate" class="col-md-2 control-label"></label>

            <div class="col-md-10">

                <input asp-for="BirthDate" class="form-control" />

                <span asp-validation-for="BirthDate" class="text-danger"></span>

            </div>

        </div>

...

        <div class="form-group">

            <div class="col-md-offset-2 col-md-10">

                <input type="submit" value="Create" class="btn btn-default" />

            </div>

        </div>

    </div>

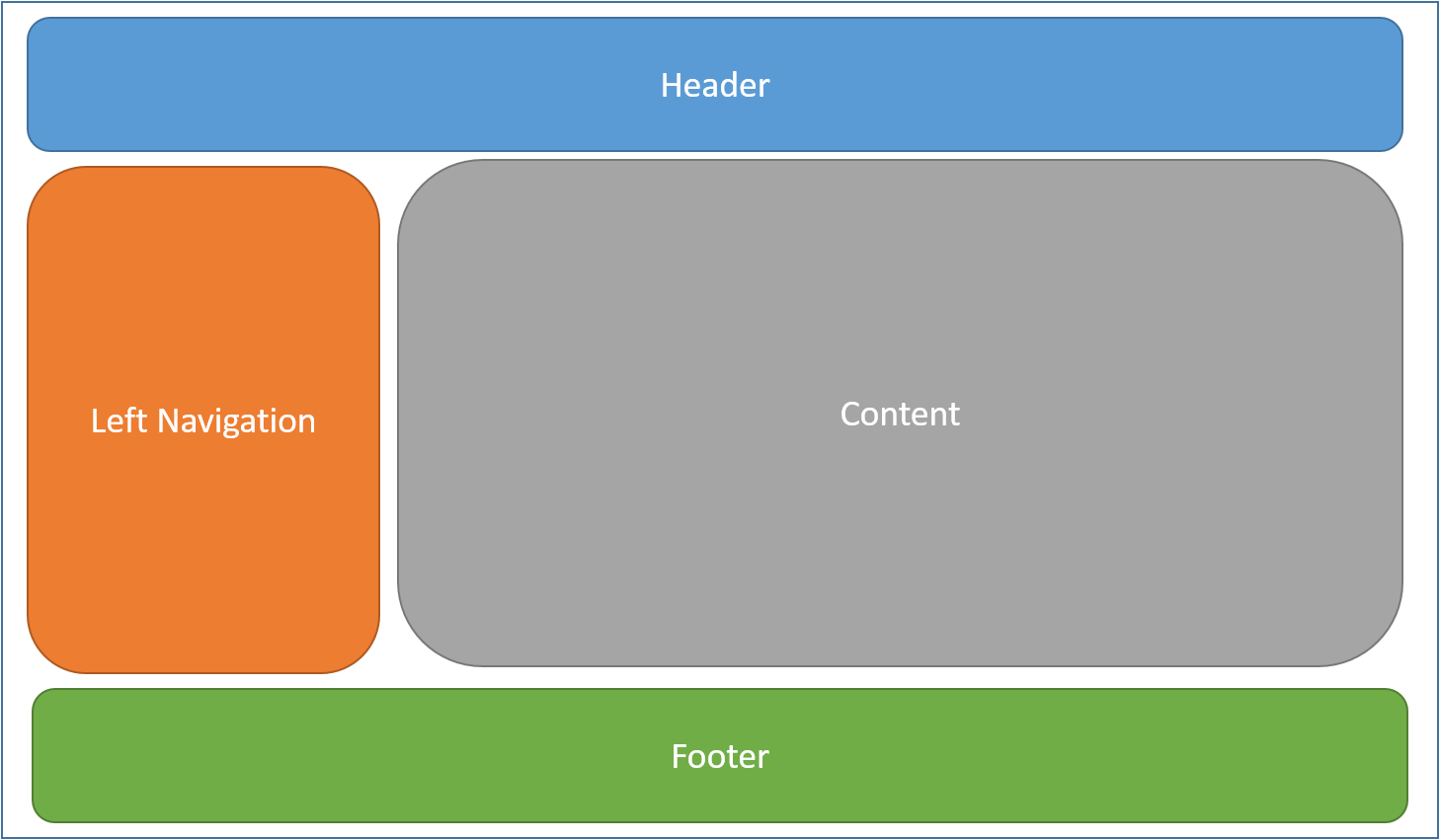
</form>

<div asp-validation-summary et <span asp-validation-for sont des Tag Helpers qui affichent les erreurs de validation.

<input asp-for produit bien entendu une zone de saisie avec l’élément <input>

# Les pages de disposition

Les pages d’une application web ont généralement une structure visuelle commune, afin que l’application ait une présentation homogène. Voici un exemple de structure :



On désigne cette structure de page par les mots **disposition** ou **mise en page** ou encore **layout** en anglais. Elle est décrite par une ou plusieurs **pages de disposition**, qui sont des fichiers cshtml.

Pour générer la réponse html à une requête http, le moteur de vue Razor combine le code d’une vue et d’une page de disposition.

## Contenu

Lorsqu’on crée une nouvelle application web MVC, elle contient par défaut la page de disposition Views/Shared/\_Layout.cshtml. Cette page contient des éléments utilisés dans toute l’application :

* Une référence au titre de la vue courante
* Les références aux feuilles de style Bootstrap
* Une barre de navigation
* Un emplacement réservé pour le corps de la vue
* Un pied de page par défaut
* Les références aux bibliothèques de scripts Bootstrap et jQuery
* Un emplacement réservé pour des scripts fournis par la vue

Les emplacements réservés font référence à des sections ou au corps de la vue courante. Ils sont implémentés par les méthodes RenderBody et RenderSection.

Exemple :

<!DOCTYPE html>

<html>

<head>

    <meta name="viewport" content="width=device-width" />

    <title>@ViewData.Title</title>

</head>

<body>

    <div class="container body-content">

        @RenderBody()

    </div>

    @RenderSection("scripts", required: false)

</body>

</html>

Lorsque le moteur de vue Razor génère le code html d’une réponse à partir d’une vue et de la page de disposition, il remplace @RenderBody par le corps de la vue, et @RenderSection("nom") par le contenu de la section de nom spécifié en paramètre.

La page de disposition peut contenir des emplacements réservés pour plusieurs sections, certaines étant facultatives, et d’autres obligatoires. Si une vue ne contient pas les sections obligatoires, cela provoque une erreur à l’exécution.

Dans notre exemple, la page de disposition contient un emplacement réservé pour une section nommée « scripts » facultative. La vue pourra ainsi contenir une section nommée « scripts » fournissant des scripts spécifiques :

@section scripts  
{

    <script type="text/javascript" src="/scripts/main.js"></script>

}

Le titre de la page est également défini dans la vue et récupéré dans la page de disposition grâce au dictionnaire ViewData. En effet, comme il s'agit d'un simple texte, inutile d'utiliser une section pour cela.

**Le framework Bootstrap**

La page de disposition par défaut d’une application ASP.Net Core utilise le framework Bootstrap pour la mise en page de la vue et l’ajout de styles CSS. Ce framework est le plus populaire pour la création d’applications responsives. Il est composé d’un ensemble de fichiers CSS et JavaScript et facilite la création d’une mise en page qui s’adapte aux différents types de terminaux (smartphone, tablettes, PC).

La description de ce framework sort du cadre de ce cours. On se référera aux pages suivantes :

* [Présentation synthétique](https://docs.microsoft.com/en-us/aspnet/core/client-side/bootstrap) sur docs.microsoft.com
* [Page de démo](https://getbootstrap.com/docs/3.3/examples/theme/) des composants Bootstrap (regarder le code source de cette page)
* [Page de doc](https://getbootstrap.com/docs/3.3/components/) sur les composants et les icônes
* [Page sur les scripts](https://getbootstrap.com/docs/3.3/javascript/) permettant d'ajouter du dynamisme (transitions, modales, onglets…)

## Référencement

**Page par défaut**

La page de disposition par défaut Views/Shared/\_Layout.cshtml, créée en même temps que l’application, est référencée à l’intérieur du fichier Views/\_ViewStart.cshtml, de la façon suivante :

@{

    Layout = "\_Layout"; // Référence relative au fichier partagé \_Layout.cshtml

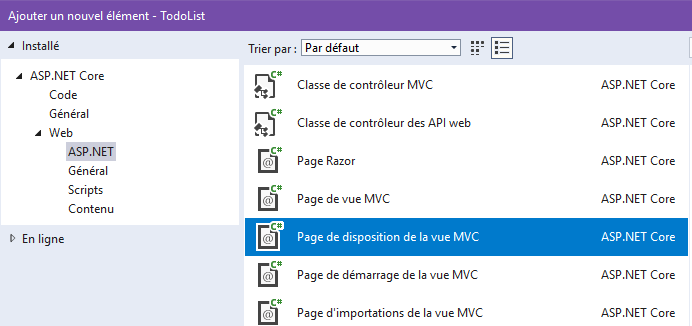
}

Le fichier \_ViewStart est exécuté par le moteur de vue avant chaque vue (sauf pour les vues partielles), de sorte que la disposition est bien appliquée à chaque vue.

**Pages spécifiques**

Certaines parties de l’application sont spécifiques et ont besoin d’une disposition différente de celle définie dans la page de disposition par défaut.

On peut ajouter d’autres pages de disposition à l’application, au moyen de la boîte de dialogue d’ajout d’un nouvel élément, accessible via le menu contextuel « Ajouter \ Nouvel élément… » :



Une page de disposition doit être placée dans un sous-dossier du dossier Views. Il peut s’agir du dossier Shared contenant les éléments partagés entre toutes les vues, ou bien d’un dossier de vues particulier si la page est destinée uniquement aux vues de ce dossier.

**Références relatives et absolues**

Une vue peut faire référence à une page de disposition de façon relative ou absolue. L’exemple suivant montre une référence absolue à la page de disposition \_myLayoutPage, placée dans le dossier Shared :

@{

    ViewData["Title"] = "Home Page";

    Layout = "~/Views/Shared/\_myLayoutPage.cshtml";

}

Une référence relative spécifie uniquement son nom (comme dans le fichier \_ViewStart). Dans ce cas, MVC cherche la page en premier dans le dossier contenant la vue, puis, s’il ne la trouve pas, dans le dossier Shared.

C’est toujours la première page trouvée qui est utilisée. On ne peut pas combiner plusieurs pages de disposition.

La référence à la page de disposition peut être précisée au début du code de la vue, mais aussi dans le code d’une action du contrôleur, au moment du rendu de la page :

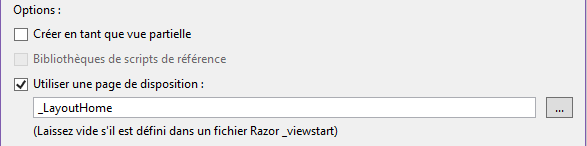
public IActionResult Index()

{

    return View("Index", "myLayoutPage");

}

Comme nous l’avons vu plus haut, la boîte de dialogue de création de vue permet de spécifier une page de disposition à utiliser pour la vue. La référence, absolue ou relative, est ajoutée dans le code de la vue.



# Les formulaires

Les formulaires sont les éléments d’interaction essentiels de l’utilisateur avec l’application. Ils permettent à l’utilisateur de saisir des données, puis de les envoyer au serveur par clic sur un bouton de soumission.

## Le Tag Helper Form

Dans sa forme la plus simple, un formulaire ressemble à ceci :

<form asp-action="Edit">

   <div>

...

      <input type="submit" value="Valider" />

   </div>

</form>

Par défaut, les données sont traitées côté serveur par le contrôleur dont le nom correspond au répertoire dans lequel se trouve la vue. Mais on peut spécifier explicitement le nom d’un autre contrôleur au moyen de l’attribut « **asp-controller** ».

Le nom de l’action du contrôleur qui traite les données est spécifié par l’attribut « **asp-action** ».

Le formulaire suivant :

<form asp-controller="Employees" asp-action="Edit">  
...  
</form>

… génère le code html suivant :

<form action="/Employees/Edit/5" method="post">  
 ...

<input name="\_\_RequestVerificationToken" type="hidden" value="..." />

</form>

Les attributs asp-controller et asp-action du Tag Helper Form permettent de générer la valeur de l’attribut « action » de l’élément html <form>.

Le champ caché nommé « \_\_RequestVerificationToken » est utilisé pour empêcher les attaques Cross-Site Requet Forgery (CSRF). Ce type d’attaque consiste à faire exécuter une requête http falsifiée à un utilisateur authentifié, en profitant de ses droits et à son insu.

## Attributs de l’élément <form>

[Cette page](https://developer.mozilla.org/fr/docs/Web/HTML/Element/Form) fournit la liste complète des attributs de l’élément html <form>. En voici deux qui sont couramment utilisés :

**Method** : définit la méthode http pour envoyer les données au serveur. Il peut avoir les valeurs « get » ou « post ». Si on ne spécifie rien dans le template, le code html est généré avec la valeur post, ce qui signifie que les données sont transportées dans le corps de la requête. Avec la valeur get, les données sont passées en tant que paramètres dans l’url de la requête.

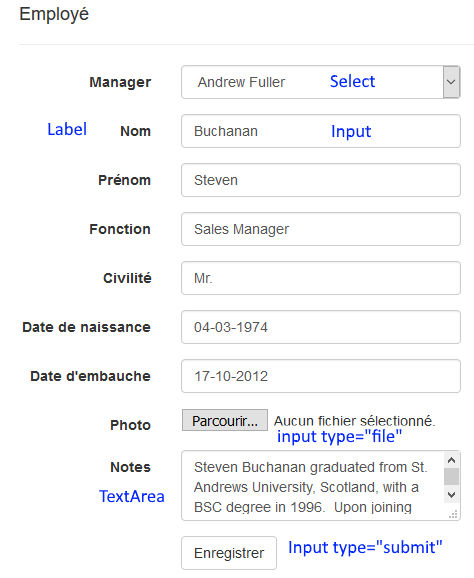
**Enctype** : définit le format d’encodage (appelé type MIME) des données envoyées au serveur via la méthode post. Il peut prendre les valeurs suivantes :

* application/x-www-form-urlencoded (valeur par défaut) : format semblable à celui utilisé pour encoder les url
* multipart/form-data : à utiliser si le formulaire contient un élément <input type="file"> pour envoyer un fichier
* text/plain : utilisé uniquement à des fins de débogage, pour faciliter la lecture des données, mais pas adapté pour transmettre les données de façon fiable

## Tag Helpers de formulaires

Les principaux Tag Helpers utilisés dans les formulaires sont présentés sur [cette page](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms).

La capture ci-dessous présente un formulaire utilisant ces Tag Helpers :



Le code de la vue représentant ce formulaire est le suivant :

<**form** **asp-action**="Edit" enctype="multipart/form-data">

   <div class="form-horizontal">

      <h4>Employé</h4>

      <hr />

      <**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>  
       <!-- L'id ne doit pas être modifiable, mais est nécessaire pour identifier

l'enregistrement à mettre à jour dans la base -->

      <**input** **type**="hidden" **asp-for**="EmployeeId" />

      <div class="form-group">

         <**label** **asp-for**="ReportsTo" class="control-label col-md-2"></**label**>

         <div class="col-md-10">

            <!-- Liste déroulante alimentée par le ViewBag -->

            <**select** **asp-for**="ReportsTo" class="form-control"  
 **asp-items**="ViewBag.ReportsTo"></**select**>

            <**span** **asp-validation-for**="ReportsTo" class="text-danger"></**span**>

         </div>

      </div>

      <div class="form-group">

         <!-- Libellé reprenant la valeur de l'attribut Display

            associé à la propriété LastName -->

         <**label** **asp-for**="LastName" class="col-md-2 control-label"></**label**>

         <div class="col-md-10">

            <!-- Zone de saisie de la valeur de la propriété LastName -->

            <**input** **asp-for**="LastName" class="form-control" />

            <!-- Génère les données nécessaires à la validation  
  de la saisie côté client -->

            <**span** **asp-validation-for**="LastName" class="text-danger"></**span**>

         </div>

      </div>

...

      <div class="form-group">

         <**label** **asp-for**="HireDate" class="col-md-2 control-label"></**label**>

         <div class="col-md-10">

            <**input** **asp-for**="HireDate" class="form-control" />

            <**span** **asp-validation-for**="HireDate" class="text-danger"></**span**>

         </div>

      </div>

      <div class="form-group">

         <**label** **asp-for**="Photo" class="col-md-2 control-label"></**label**>

         <div class="col-md-10">

            <!-- Bouton pour sélectionner et uploader un fichier de type image,

                et affichage du nom du fichier sélectionné -->

            <input type="file" name="photo" accept="image/\*" />

         </div>

      </div>

      <div class="form-group">

         <**label** **asp-for**="Notes" class="col-md-2 control-label"></**label**>

         <div class="col-md-10">

            <!-- Zone de texte redimensionnable avec barre de défilement -->

            <**textarea** **asp-for**="Notes" class="form-control"></**textarea**>

            <**span** **asp-validation-for**="Notes" class="text-danger"></**span**>

         </div>

      </div>

      <div class="form-group">

         <div class="col-md-offset-2 col-md-10">

            <!-- Bouton de soumission du formulaire -->

            <input type="submit" value="Enregistrer" class="btn btn-default" />

         </div>

      </div>

   </div>

</**form**>

Analysons ce code :

Les éléments <div> permettent de positionner les autres éléments dans la page. Les Tag Helpers <label>, <input> et <select> possèdent un attribut asp-for précisant à quelle propriété de la source de données ils sont reliés.

**Gestion de l’identifiant**

L’identifiant de l’employé ne doit pas être modifiable par l’utilisateur et n’a pas lieu d’être affiché. Cependant, il est nécessaire au contrôleur pour identifier l’enregistrement à mettre à jour dans la base de de données. On le met donc dans un champ caché.

**Gestion de la liste déroulante Manager**

Le contenu de la liste déroulante est défini par l’attribut asp-items. Il est ici affecté par la valeur de la propriété dynamique ReportsTo du ViewBag. Celle-ci doit contenir une collection de type SelectList.

**Gestion de l’upload de la photo**

L’élément <input> de type file utilisé pour sélectionner le fichier de la photo, comprend un attribut « accept » qui permet de spécifier le type de fichier attendu. La valeur « image/\* » désigne tous les fichiers de type image (jpeg, bmp, gif…).

Pour que le fichier soit correctement transmis, le formulaire doit avoir l’attribut enctype="multipart/form-data"

Pour récupérer le fichier sélectionné, la méthode du contrôleur qui traite les données du formulaire doit comporter un paramètre de type IFormFile, et de même nom que l’élément <input>.

Voici le code de la méthode du contrôleur qui traite les données du formulaire :

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(int id, IFormFile photo,

   [Bind("EmployeeId,ReportsTo,LastName,FirstName,Title,TitleOfCourtesy,BirthDate,HireDate,Photo,Notes")] Employee employee)

{

   if (id != employee.EmployeeId)

   {

      return NotFound();

   }

   if (ModelState.IsValid)

   {

      try

      {

         if (photo != null)

         {

            // Transforme la photo en tableau de bytes

            Stream s = photo.OpenReadStream();

            using (MemoryStream ms = new MemoryStream())

            {

               s.CopyTo(ms);

               employee.Photo = ms.ToArray();

            }

         }

         // Ceci marque toutes les propriétés de l'entité comme modifiées

         \_context.Update(employee);

         // On marque la propriété photo comme non modifiée si aucun

         // fichier n'a été uploadé

         var entry = \_context.ChangeTracker.Entries<Employee>().FirstOrDefault();

         if (photo == null)

            entry.Property(e => e.Photo).IsModified = false;

         // De même, on marque la propriété AddressId inchangée car le formulaire

         // n'affiche pas cette donnée

         entry.Property(e => e.AddressId).IsModified = false;

         // Enregistre les modifications dans la base

         await \_context.SaveChangesAsync();

      }

      catch (DbUpdateConcurrencyException)

      {

         if (!EmployeeExists(employee.EmployeeId))

            return NotFound();

         else

            throw;

      }

      return RedirectToAction("Index");

   }

   // Création d'une liste pour alimenter la liste déroulante  
 // de sélection du manager

   var employés = \_context.Employee.OrderBy(e => e.FullName);

   ViewData["ReportsTo"] = new SelectList(employés,

         "EmployeeId", "FullName", employee.ReportsTo);

   return View(employee);

}

Si le fichier reçu dans le paramètre « photo » n’est pas null, il est transformé en tableau de bytes pour pouvoir être enregistré dans le champ correspondant de la base de données.

Avant enregistrement en base, on marque les propriétés de l’entité employee comme modifiées, en appelant la méthode Update du DbContext. Ceci permet de mettre à jour les champs correspondants dans la base de données, lorsqu’on appelle SaveChangeAsync. Cependant, dans le cas particulier de la propriété Photo, on ne souhaite pas supprimer la photo existante en base si aucun fichier n’a été uploadé (choix fonctionnel). Pour cela, on passe explicitement son statut IsModified à false, de façon à ce qu’elle ne soit pas affectée par SaveChangeAsync.

Pour alimenter la liste déroulante de sélection du manager, on crée une SelectList, dont le constructeur prend en paramètres :

* Une collection d’items de type IEnumerable. Elle est ici obtenue au moyen d’une requête sur le DbContext
* La propriété de la classe Employee qui définit la valeur de l’élément sélectionné
* La propriété de la classe Employee qui définit le texte de l’élément sélectionné
* L’instance d’Employee sélectionnée (ReportsTo est bien de type Employee)

# La validation

La validation consiste à vérifier que les données saisies se conforment à certaines règles, avant leur enregistrement en base. La validation sert des objectifs de sécurité et de cohérence des données.

Les règles de validation peuvent être jouées à la fois côté client (par du code JavaScript), et côté serveur. Les jouer côté client n’est pas obligatoire, mais permet une meilleure expérience utilisateur, puisque les erreurs peuvent être affichées immédiatement, sans avoir à poster la page. Côté serveur, elles sont jouées automatiquement avant l’appel de la méthode d’action du contrôleur.

La validation côté serveur est toujours nécessaire, car il se peut que le JavaScript soit désactivé sur le navigateur. De plus, certaines règles de validation complexes ne peuvent pas être vérifiées côté client.

## Validation côté serveur

ASP.Net fournit un moyen très commode pour définir des règles de validation de façon centralisée : les attributs de validation.

Comme les attributs de format d’affichage, des attributs de validation sont définis dans l’espace de noms System.ComponentModel.DataAnnotations. Nous verrons également comment définir nos propres attributs personnalisés.

Les attributs sont appliqués sur les propriétés des modèles, ce qui permet de centraliser les règles de validation, et de les appliquer à toutes les saisies utilisant ces modèles.

### Attributs prédéfinis

[Cette page MSDN](https://msdn.microsoft.com/library/system.componentmodel.dataannotations(v=vs.110).aspx) fournit la liste complète des attributs prédéfinis (pour la validation, et l’affichage). Voici les attributs de validation les plus communs :

|  |  |
| --- | --- |
| **Attribut** | **Fonction** |
| Required | Rend la saisie du champ obligatoire |
| MinimumLength | Définit une longueur minimale |
| StringLength | Définit la longueur maximale (et éventuellement minimale) d’un champ texte |
| RegularExpression | Expression régulière qui définit le format de la valeur saisie |
| Range | Définit une plage de valeurs pour la saisie |
| EmailAddress | La valeur saisie doit respecter le format d’une adresse e-mail |
| Phone | La valeur saisie doit respecter le format d’un N° de téléphone |
| Compare | Compare la valeur du champ avec un autre champ (ex : pour confirmer une adresse mail) |

NB/ Les attributs EmailAddress et Phone sont en fait des expressions régulières prédéfinies.

L’exemple ci-dessous montre une utilisation concrète de quelques attributs de validation sur une classe de modèle décrivant un film :

public class Movie

{

   public int ID { get; set; }

   [StringLength(60, MinimumLength =3)]

   [Required]

   public string Title { get; set; }

   [Display(Name ="Release Date")]  
   [DisplayFormat(DataFormatString = "{0:dd-MM-yyyy}",  
 ApplyFormatInEditMode = true)]

   public DateTime ReleaseDate { get; set; }

   [RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]\*$")]

   [StringLength(30)]

   [Required]

   public string Genre { get; set; }

   [Range(1, 100)]

   [DataType(DataType.Currency)]

   public decimal Price { get; set; }

   [RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]\*$")]

   [StringLength(5)]

   [Required]

   public string Rating { get; set; }

}

Les champs de types numériques ou date sont nativement obligatoires. Il n’est pas nécessaire d’ajouter l’attribut [Required].

**Remarques :**

* Ne pas confondre les attributs de validation et les attributs de format d’affichage (DataType, Display, DisplayFormat). Ces derniers n’ont pas d’incidence sur la saisie, mais uniquement sur l’affichage.
* Une même propriété peut être décorée de plusieurs attributs. On peut les combiner sur une même ligne pour plus de concision. Exemple :

[Range(1, 100), DataType(DataType.Currency)]

public decimal Price { get; set; }

Voici un exemple du visuel qu’on peut obtenir pour un formulaire se saisie d’un film, avec les règles de validation décrites ci-dessus :



**Personnalisation des messages d’erreurs**

Les messages d’erreurs sont ceux fournis par défaut par MVC, mais il est possible de définir nos propres messages d’erreurs, comme le montre l’exemple suivant :

[RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]\*$", ErrorMessage = "Le genre doit  
 comporter des lettres, tirets ou espaces, et commencer par une majuscule")]

[StringLength(30), Required]

public string Genre { get; set; }

### Attributs personnalisés

Certaines règles de validation ne peuvent pas être implémentées au moyen des attributs prédéfinis, notamment lorsqu’elles portent sur plusieurs champs à la fois. Dans ce cas, on peut définir nos propres attributs de validation, ou bien faire implémenter l’interface IValidatableObject à la classe de modèle.

Supposons par exemple qu’on souhaite interdire de saisir le genre « Classique » sur un film sorti après 1960. On peut implémenter cette règle au moyen d’un attribut personnalisé qu’on appellera ClassicMovie, et qui est décrit par la classe suivante :

using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models

{

   public class ClassicMovieAttribute : ValidationAttribute

   {

      private int \_year;

      public ClassicMovieAttribute(int Year)

      {

         \_year = Year;

      }

      protected override ValidationResult IsValid(object value, ValidationContext validationContext)

      {

         Movie movie = (Movie)validationContext.ObjectInstance;

         if (movie.Genre.ToUpper() == "CLASSIQUE" &&  
 movie.ReleaseDate.Year > \_year)

         {

            return new ValidationResult("Le genre Classique ne peut pas être attribué à un film sorti après " + \_year);

         }

         return ValidationResult.Success;

      }

   }

}

La classe dérive de ValidationAttribute, et on redéfinit sa méthode IsValid. Cette méthode possède deux paramètres :

* value représente la valeur saisie à vérifier
* validationContext décrit le contexte dans lequel le contrôle de validation est exécuté.

La propriété Objectinstance permet d’obtenir l’objet en cours de validation (un film dans cet exemple). On a ainsi accès à l’ensemble des valeurs saisies, ce qui permet de faire porter la validation sur plusieurs champs.

Nous avons ici passé l’année en paramètre du constructeur, de façon à pouvoir spécifier sa valeur au moment de l’utilisation de l’attribut.

Cet attribut peut être mis en œuvre de la façon suivante :

[ClassicMovie(1960)]

public string Genre { get; set; }

### Méthode de validation

La règle de validation précédente peut également être implémentée dans la méthode Validate définie par l’interface IValidatableObject. C’est ce qui est fait dans l’exemple ci-dessous, dans lequel on a rajouté également un second contrôle portant sur le genre et le prix du film :

public class Movie : IValidatableObject

{

   public int ID { get; set; }

   [StringLength(60, MinimumLength =3), Required]

   [DataType(DataType.MultilineText)]

   [Display(Name ="Titre")]

   public string Title { get; set; }

   [Display(Name ="Date de sortie")]

   [DisplayFormat(DataFormatString = "{0:dd-MM-yyyy}"]

   public DateTime ReleaseDate { get; set; }

   [RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]\*$")]

   [StringLength(30), Required]

   public string Genre { get; set; }

   [Range(1, 100), DataType(DataType.Currency)]

   public decimal Price { get; set; }

   public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)

   {

      Movie movie = (Movie)validationContext.ObjectInstance;

      if (movie.Genre.ToUpper() == "CLASSIQUE" && movie.ReleaseDate.Year > 1960)

      {

         yield return new ValidationResult("Le genre Classique ne peut pas être attribué à un film sorti après 1960", new string[] { "Genre" });

      }

      if (movie.Genre.ToUpper() == "CLASSIQUE" && movie.Price > 10)

      {

         yield return new ValidationResult("Les films classiques ne doivent pas couter plus de 10 €", new string[] { "Price" });

      }

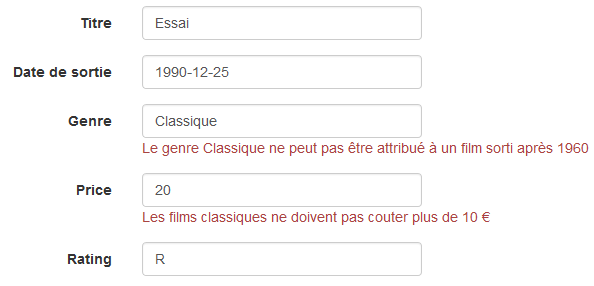
   }

}

La classe Movie implémente l’interface IValidatableObject en fournissant une méthode Validate.

Le contexte dans lequel la validation a lieu est passé en paramètre de cette méthode (comme pour les attributs personnalisés). La méthode peut effectuer plusieurs contrôles de validation, et renvoyer leurs résultats un par un au moyen de l’instruction « yield return ». On voit en effet que le retour de la méthode est énumérable.

Le résultat de chaque contrôle est représenté par un objet de type ValidationResult, dont le constructeur prend en paramètre le message d’erreur à afficher, et un tableau des noms des propriétés sur lesquelles porte l’erreur. Ce second paramètre n’est pas obligatoire, mais il permet d’afficher le message à côté de la zone de saisie concernée (sinon, le message est affiché au-dessus du formulaire) :



/!\ Lorsqu’on définit de règles de validation par attribut et par une méthode Validate, ASP.Net vérifie tout d’abord les règles définies par les attributs. Si et seulement si aucune erreur n’est remontée, il exécute ensuite la méthode Validate du modèle.

### Recommandations

Les attributs personnalisés sont bien adaptés à des contrôles portant sur un seul champ, et applicables sur plusieurs modèles. Pour des contrôles spécifiques à un modèle donné, ou portant sur plusieurs champs, on préférera implémenter IValidatableObject.

Lorsqu’un contrôle porte sur un seul champ, il est bon d’afficher le message d’erreur à proximité de ce champ. En revanche, lorsqu’il porte sur plusieurs champs, il est plus clair d’afficher le message d’erreur au-dessus du formulaire. On peut éventuellement ajouter des astérisques à côté des champs incriminés, comme ceci :

Movie movie = (Movie)validationContext.ObjectInstance;

if (movie.Genre.ToUpper() == "CLASSIQUE" && movie.ReleaseDate.Year > 1960)

{

   yield return new ValidationResult("Le genre Classique ne peut pas être attribué à un film sorti après 1960");

   yield return new ValidationResult("\*", new string[] { "Genre","ReleaseDate" });

}

En vérifiant le caractère obligatoire des champs par des attributs, on n’a pas besoin de tester la valeur null dans la méthode Validate, ce qui simplifie le code.

### Vérification du résultat de la validation

La validation des données est faite automatiquement par ASP.net MVC avant l’exécution de la méthode d’action du contrôleur. Dans cette méthode, on peut vérifier le résultat de la validation de la façon suivante :

[HttpPost]

public async Task<IActionResult> Create(Movie movie)

{

   if (ModelState.IsValid)

   {

      \_context.Add(movie);

      await \_context.SaveChangesAsync();

      return RedirectToAction("Index");

   }

   return View(movie);

}

ModelState est une propriété de la classe ControllerBase, qui représente l’état de la liaison de modèle. Elle inclut les informations de validation.

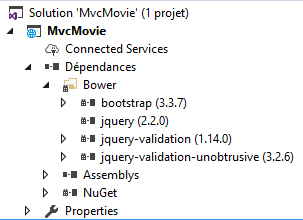
Dans cet exemple, si les données sont valides, on les enregistre en base, et on affiche la page d’index (qui affiche par exemple la liste des films). Si les données ne sont pas valides, on réaffiche la vue de saisie, afin d‘afficher les messages d’erreurs et de permettre à l’utilisateur de corriger les erreurs.

NB/ Il est possible de lancer manuellement la validation en faisant :

TryValidateModel(movie);

## Validation côté client

La validation côté client fait gagner du temps à l’utilisateur, car elle ne nécessite pas de poster la page. Elle est en effet réalisée en JavaScript, et utilise trois bibliothèques jQuery, comme le montre la capture ci-dessous :



Le principe de la validation côté client est le suivant :

* Les HTML Helpers ou Tag Helpers génèrent des attributs data-\* sur les éléments html, à partir des attributs de validation et des attributs DataType ajoutés sur les propriétés du modèle
* Lorsqu’on clique sur le bouton de soumission du formulaire, le code JavaScript contrôle les valeurs saisies, en se basant sur les attributs data-\*
* Si des erreurs sont relevées, le code JavaScript affiche des messages d’erreurs, et empêche la soumission du formulaire.

Exemple : reprenons la propriété Genre définie sur le modèle Movie :

[RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]\*$")]

[StringLength(30), Required]

public string Genre { get; set; }

Dans le template de la vue qui permet de saisir un film, nous avons ceci :

<div class="form-group">

    <label asp-for="Genre" class="col-md-2 control-label"></label>

    <div class="col-md-10">

        <input asp-for="Genre" class="form-control" />

        <span asp-validation-for="Genre" class="text-danger"></span>

    </div>

</div>

Les Tag Helpers input et span génèrent le code html suivant :

<div class="form-group">

<label class="col-md-2 control-label" for="Genre">Genre</label>

<div class="col-md-10">

<input class="form-control" type="text" data-val="true"

data-val-length="Le champ Genre doit &#xEA;tre une cha&#xEE;ne dont la longueur maximale est de 30."

data-val-length-max="30"

data-val-regex="Le champ Genre doit correspondre &#xE0; l&#x27;expression r&#xE9;guli&#xE8;re &#x27;^[A-Z]&#x2B;[a-zA-Z&#x27;&#x27;-&#x27;\s]\*$&#x27;."

data-val-regex-pattern="^[A-Z]&#x2B;[a-zA-Z&#x27;&#x27;-&#x27;\s]\*$"

data-val-required="Le champ Genre est requis."

id="Genre" name="Genre" value="" />

<span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg-replace="true"></span>

</div>

</div>

On voit que des attributs data-val-\* ont été générés pour chacune des règles de validation. Ces attributs sont utilisés par le code JavaScript pour contrôler la saisie et afficher les erreurs éventuelles.

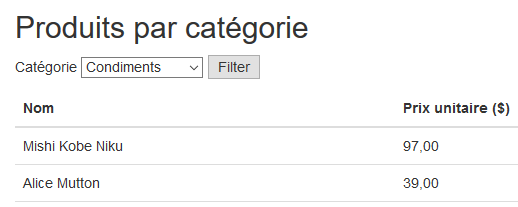
# Les ViewModels

## Principe

Les ViewModels, qu’on peut traduire en « modèles pour vues » ou « entités visuelles » sont des entités dédiées à l’affichage et la saisie d’informations dans les vues.

Jusqu’ici nous avons utilisé les entités du modèle pour alimenter les vues. Ceci fonctionne bien tant que chaque vue n’a besoin que d’un seul type d’entité modèle. Cependant, une vue plus élaborée peut avoir besoin de données issues de plusieurs types de modèles, et de propriétés non stockées dans le modèle.

Exemple : on souhaite créer une vue qui affiche les produits par catégorie :



Les données manipulées par cette vue sont :

* La liste des catégories qui aliment la liste déroulante
* L’Id de la catégorie sélectionnée, qui sera envoyée au contrôleur pour le filtrage des produits
* La liste des produits de la catégorie sélectionnée, affichée dans le tableau

Les produits sont modélisés par l’entité Product, et les catégories par l’entité Category. La vue utilise donc deux types d’entités. (NB/ Product contient l’id de la catégorie, mais pas son libellé)

De plus, l’Id de la catégorie sélectionnée n’a pas à être stocké dans le modèle, car c’est une propriété qui ne sert qu’à la vue.

On a donc besoin d’une entité visuelle, qui agrège les données nécessaires à cette vue.

## Mise en œuvre

Voici l’entité visuelle nécessaire à la vue précédente :

namespace Northwind2.Models

{

   public class ProductsViewModel

   {

      // Liste des catégories

      public List<Category> Categories { get; set; }

      // Catégorie sélectionnée

      public Guid CategorieSelec { get; set; }

      // Produits de la catégorie sélectionnée

      public List<Product> Produits { get; set; }

   }

}

La propriété CategorieSelec permet de stocker le code de la catégorie sélectionnée par l’utilisateur, afin de l’envoyer dans la requête http.

Voici le code de l’action Index du contrôleur ProductController, qui renvoie cette vue :

public async Task<IActionResult> Index(Guid IdCate)

{

   // Création d'une entité visuelle fournissant la liste des catégories

   // et les produits de la catégorie demandée

   var vmProduit = new ProductsViewModel();

   // Récupération des produits de la catégorie demandée

   IQueryable<Product> produits = \_context.Product;

   if (IdCate != null && IdCate != Guid.Empty)

   {

      vmProduit.CategorieSelec = IdCate;

      produits = produits.Where(p => p.CategoryId == IdCate);

   }

   vmProduit.Produits = await produits.OrderBy(p => p.Name)

                        .AsNoTracking().ToListAsync();

   vmProduit.Categories = await \_context.Category.OrderBy(c => c.Name)

                           .AsNoTracking().ToListAsync();

   return View(vmProduit);

}

Voici un extrait du code de la vue montrant comment utiliser l’entité visuelle et appeler l’action ci-dessus :

@model Northwind2.Models.ProductsViewModel

@{

    ViewData["Title"] = "Produits";

}

<h2>Produits par catégorie</h2>

<form asp-controller="Products" asp-action="Index" method="get">

   <p>

      <span>Catégorie</span>

      <select asp-for="IdCate"

              asp-items="@(new SelectList(Model.Categories,"CategoryId","Name"))">

         <option value="">Toutes</option>

      </select>

      <input type="submit" value="Filtrer" />

   </p>

</form>

<table class="table">

    <thead>

        <tr>

            <th>

                @Html.DisplayNameFor(model => model.Produits[0].Name)

            </th>

            <th>

                @Html.DisplayNameFor(model => model.Produits[0].UnitPrice)

            </th>

...

        </tr>

    </thead>

    <tbody>

@foreach (var item in Model.Produits) {

        <tr>

            <td>

                @Html.DisplayFor(modelItem => item.Name)

            </td>

            <td>

                @Html.DisplayFor(modelItem => item.UnitPrice)

            </td>

...  
        </tr>

}

    </tbody>

</table>

Le modèle utilisé par la vue est de type Northwind2.Models.ProductsViewModel.  
La propriété qui contient la liste des produits est nommée Produits (surlignée en vert).  
Les libellés des en-têtes de colonne du tableau sont récupérés sur le premier produit de la liste, au moyen de « model.Produits[0].xxx ».

Le formulaire contient un élément <select> pour la liste déroulante, et un bouton de soumission. L’essentiel se trouve dans les attributs des Tag Helpers form et select :

Les attributs du Tag Helper form précisent le contrôleur et l’action visés, ainsi que la méthode d’envoi des données. Comme on ne modifie pas les données de la base, on utilise la méthode get et non post.

Sur le Tag Helper select :

* asp-for précise la propriété de l’entité ProductViewModel qui sera affectée par l’élément sélectionné dans la liste.
* asp-items fournit le contenu de la liste déroulante. Il s’agit d’une collection de type SelectList, dont le constructeur permet de préciser :
  + La propriété du modèle fournissant la liste d’entités
  + Le nom de la propriété de ces entités qui représente la valeur sélectionnée
  + Le nom de la propriété de ces entités qui représente la valeur affichée dans la liste

**Remarque :**

L’exemple que nous venons de voir peut être géré plus simplement sans entité visuelle, en remplissant la liste déroulante des catégories au moyen de ViewBag. Dans ce cas, le modèle utilisé par la vue est une simple liste de produits, et la liste déroulante peut être générée par un HTML Helper de la façon suivante :

@Html.DropDownList("IdCate", new SelectList(ViewBag.categories, "CategoryId", "Name"))

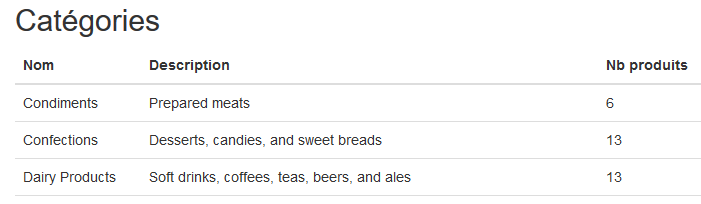
Bien que plus simple, cette solution a toutefois l’inconvénient d’utiliser ViewBag, c’est-à-dire une collection d’Object, et non des données fortement typées. Elle est déconseillée par Microsoft. De plus, si une entité visuelle est nécessaire pour d’autres données, autant l’utiliser aussi pour la liste déroulante.

**Mappage entre entités du modèle et entité visuelle**

Dans l’exemple précédent, l’entité visuelle est utilisée uniquement pour afficher des données, et non pour en saisir. Elle n’est pas nécessaire pour la saisie d’un produit, car l’Id de la catégorie fait partie de l’entité Product. Toutefois, dans certains cas, il peut être nécessaire d’utiliser une entité visuelle pour la saisie. Dans ce cas, dans le contrôleur, il faut copier les valeurs de l’entité visuelle dans les entités de modèle avant enregistrement en base.

## Agrégation

Considérons l’exemple ci-dessous dans lequel on affiche la liste des catégories et le nombre de produits qu’elle contiennent :



Un moyen simple de réaliser cette page est de créer une vue utilisant une collection d’entités Category, fournie par la méthode de contrôleur suivante :

public async Task<IActionResult> Index()

{

   var categories =  \_context.Category.Include(c => c.Products)  
 .OrderBy(c => c.Name);

   return View(await categories.AsNoTracking().ToListAsync());

}

Chaque catégorie a une propriété Products qui fournit la liste des produits de la collection. Dans la vue, le nombre de produits de chaque catégorie est récupéré par la propriété Products.Count. Pour que cela fonctionne, il faut charger la liste des produits de chaque catégorie en utilisant la méthode Include().

Cette méthode à l’avantage d’être très simple à mettre en œuvre. Cependant, si le nombre de produits et de catégories est élevé, cela peut poser des problèmes de performances à cause du chargement d’un volume d données important en mémoire. On aura dans ce cas intérêt à utiliser la méthode suivante, plus optimisée :

**Utilisation d’une entité visuelle et d’une requête SQL**

Au lieu de récupérer tous les produits de chaque catégorie, nous pouvons les compter au moyen d’une requête SQL avec clause GROUP BY. Mais pour récupérer à la fois les caractéristiques de chaque catégorie et son nombre de produits, nous avons besoin de l’entité visuelle suivante :

public class CategorieViewModel

{

   public Category Categorie { get; set; }

   // Nombre de produits de la catégorie

   public int NbProduits { get; set; }

}

Le contrôleur remplit une liste de ces entités à partir de la requête SQL de la façon suivante :

public async Task<IActionResult> OptimizedList()

{

   // Requête SQL optimisée : on ramène uniquement les infos nécessaires

   string req = @"select C.CategoryId, C.Name, C.Description,  
  count(P.ProductId) ProductsCount

         from Category C

         left outer join Product P on C.CategoryId = P.CategoryId

         group by C.CategoryId, C.Name, C.Description

         order by C.Name";

   var categories = new List<CategorieViewModel>();

   using (var conn = (SqlConnection)\_context.Database.GetDbConnection())

   {

      var cmd = new SqlCommand(req, conn);

      await conn.OpenAsync();

      using (var sdr = await cmd.ExecuteReaderAsync())

      {

         while (sdr.Read())

         {

            var c = new CategorieViewModel();

            c.Categorie = new Category();

            c.Categorie.CategoryId = (Guid)sdr["CategoryId"];

            c.Categorie.Name = (string)sdr["Name"];

            c.Categorie.Description = (string)sdr["Description"];

            c.NbProduits = (int)sdr["ProductsCount"];

            categories.Add(c);

         }

      }

   }

   return View(categories);

}

Lorsqu’on utilise EF, on peut récupérer une connexion à la base depuis le contexte au moyen de la méthode : \_context.Database.GetDbConnection()

On peut ensuite utiliser un objet SqlCommand classique utilisant cette connexion pour exécuter la requête SQL. Notez que celle-ci utilise une jointure externe entre les tables Category et Product, pour ramener toutes les catégories, y compris celle qui n’ont aucun produit.

Dans la vue, le modèle utilisé est le suivant :

@model IEnumerable<Northwind2.Models.CategorieViewModel>

… et on peut afficher le résultat de la façon suivante :

@foreach (var item in Model)

{

   <tr>

      <td>

         @Html.DisplayFor(modelItem => item.Categorie.Name)

      </td>

      <td>

         @Html.DisplayFor(modelItem => item.Categorie.Description)

      </td>

      <td>

         @Html.DisplayFor(modelItem => item.NbProduits)

      </td>

   </tr>

}

# Tri, pagination, filtrage

## Tri

Lorsque des données sont affichées en tableau, on peut proposer un tri selon les différentes colonnes du tableau. Il peut être appliqué par exemple par clic sur les en-têtes de colonnes.

La façon la plus simple d’appliquer le tri, est de le faire au moment de la récupération des données, dans la requête SQL ou la requête Linq To Entities.

Dans l’exemple qui suit, nous allons permettre à l’utilisateur de trier le tableau d’employés selon les colonnes Nom ou Date d’embauche.

A chaque clic sur l’en-tête de l’une de ces colonnes, on inverse le sens du tri courant.

Voici le code de la vue, dans lequel nous avons ajouté des liens hypertexte sur les en-têtes de colonnes, afin d’appliquer le tri :

<table class="table">

   <thead>

      <tr>

         <th>

            <a asp-action="Index" asp-route-sortOrder="@ViewData["TriParNom"]">

               @Html.DisplayNameFor(model => model.LastName)

            </a>

         </th>

...

         <th>

            <a asp-action="Index" asp-route-sortOrder="@ViewData["TriParDate"]">

               @Html.DisplayNameFor(model => model.HireDate)

            </a>

         </th>

L’attribut **asp-route-sortOrder** permet de passer le critère et le sens de tri demandés en paramètre à la méthode du contrôleur. L’information est passée en tant que paramètre d’url, sous la forme suivante :

// Tri ascendant selon le nom  
http://localhost:52902/Employees?sortOrder=LastName

// Tri descendant selon le nom

http://localhost:52902/Employees?sortOrder=LastName\_desc

Après application du tri demandé, le sens de tri inverse doit être mémorisé par le contrôleur, afin qu’il puisse l’appliquer lors d’un prochain clic-sur l’en-tête de colonne. Il utilise pour cela le dictionnaire ViewData.

Exemple : supposons que le contrôleur trie au départ la liste par nom ascendant. Alors il doit également affecter à l’entrée ViewData["TriParNom"] la valeur « LastName\_desc ». De cette façon, si l’utilisateur clique sur l’en-tête de la colonne Nom, cela génèrera une requête avec l’url :

http://localhost:52902/Employees?sortOrder=LastName\_desc

En résumé, lors d’un clic sur un en-tête de colonne :

* La vue envoie une requête en lui passant le sens de tri en paramètre grâce à asp-route-sortOrder. Sur la vue, le sens de tri est obtenu par le dictionnaire ViewData.
* La méthode du contrôleur trie les données selon le critère et le sens reçus
* Elle renvoie à la vue un sens de tri inverse pour ce critère grâce au dictionnaire ViewData

Voici le code de la méthode Index du contrôleur :

public async Task<IActionResult> Index(string sortOrder)

{

   IQueryable<Employee> employés = \_context.Employee

            .Include(e => e.Address)

            .Include(e => e.ReportsToNavigation);

   // si la chaîne de tri reçue en paramère est vide,  
 // on définit un critère de tri par défaut

   if (string.IsNullOrEmpty(sortOrder)) sortOrder = "LastName";

   // Si le tri demandé est descendant

   if (sortOrder.EndsWith("\_desc"))

   {

      // On extrait le critère de tri

      string prop = sortOrder.Substring(0, sortOrder.Length - 5);

      employés = employés.OrderByDescending(e => EF.Property<object>(e, prop));

   }

   else

      employés = employés.OrderBy(e => EF.Property<object>(e, sortOrder));

   // On envoie le sens du tri inverse à la vue pour le prochain appel

   ViewData["TriParNom"] = sortOrder == "LastName" ? "LastName\_desc" : "LastName";

   ViewData["TriParDate"] = sortOrder == "HireDate" ? "HireDate\_desc" : "HireDate";

   // On renvoie une vue triée

   return View(await employés.AsNoTracking().ToListAsync());

}

Notons les éléments suivants :

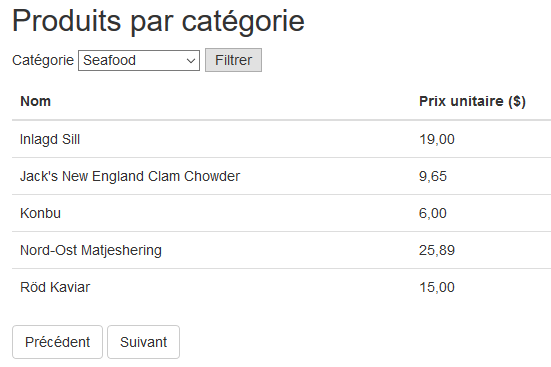
Au premier appel de la fonction, le paramètre sortOrder est vide. On définit donc un tri par défaut ascendant selon le nom.

On applique le tri sur la requête Linq To Entities au moyen des méthodes OrderBy (tri ascendant) et OrderByDescending (tri descendant).

Le nom de la propriété sur laquelle le DbSet est trié, est défini dynamiquement grâce à la méthode statique EF.Property

## Pagination

La pagination consiste à afficher les données d’un tableau en plusieurs pages successives, de façon à réduire la quantité d’information à charger en une seule fois. Il faut définir un nombre d’éléments par page et ajouter des boutons de navigations entre les pages, comme le montre la capture suivante :



Pour implémenter la pagination, nous avons besoin de :

* Filtrer les éléments d’une page donnée, identifiée par son indice
* Gérer l’état des boutons Précédent et Suivant, qui doivent être désactivés quand on est respectivement sur la première et sur la dernière page.

Ceci peut être implémenté par une classe générique, utilisable sur n’importe quelle liste à paginer. Voici son code :

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

namespace Northwind2.Models

{

   /// <summary>

   /// Modélise une page d'éléments

   /// </summary>

   /// <typeparam name="T"></typeparam>

   public class PageItems<T> : List<T>

   {

      // Indice de la page courante

      public int PageIndex { get; private set; }

      // Nombre total de pages

      public int TotalPages { get; private set; }

      /// <summary>

      /// Crée une page d'éléments à partir d'une liste

      /// </summary>

      /// <param name="items">Liste des éléments de la page</param>

      /// <param name="count">Nombre total d'éléments de la source dont est extraite la liste</param>

      /// <param name="pageIndex">Indice de la page</param>

      /// <param name="pageSize">Nombre d'éléments par page</param>

      public PageItems(List<T> items, int count, int pageIndex, int pageSize)

      {

         // Mémorise l'indice de la page crée

         PageIndex = pageIndex;

         // Calcule le nombre total de pages

         TotalPages = (int)Math.Ceiling(count / (double)pageSize);

         // Ajoute les éléments de la page à la liste interne

         this.AddRange(items);

      }

      // Renvoie Vrai s'il existe une page avant la page courante

      public bool HasPreviousPage

      {

         get { return (PageIndex > 1); }

      }

      // Renvoie Vrai s'il existe une page après la page courante

      public bool HasNextPage

      {

         get { return (PageIndex < TotalPages); }

      }

      /// <summary>

      /// Renvoie une page d'éléments tirés d'une liste source

      /// </summary>

      /// <param name="source">liste d'éléments source</param>

      /// <param name="pageIndex">Indice de la page</param>

      /// <param name="pageSize">Taille de la page (nombre d'éléments)</param>

      /// <returns></returns>

      public static async Task<PageItems<T>> CreateAsync(IQueryable<T> source,  
  int pageIndex, int pageSize)

      {

         // récupère le nombre d'éléments de la source

         var count = await source.CountAsync();

         // Extrait de la source un nombre d'éléments correpondant à pageSize

         // en commençant à partir de l'élément d'indice  
 // (pageIndex - 1) \* pageSize + 1

         var items = await source.Skip((pageIndex - 1) \* pageSize)  
 .Take(pageSize).ToListAsync();

         // Crée et renvoie une page constituée de ces éléments

         return new PageItems<T>(items, count, pageIndex, pageSize);

      }

   }

}

Nous allons maintenant mettre en œuvre la pagination sur l’exemple précédent, qui concerne l’affichage des produits d’une catégorie.

La vue soit désormais afficher une page de produits. Il faut donc modifier le type de la propriété Produits de l’entité visuelle :

public class ProductsViewModel

{

   // Liste des catégories

   public List<Category> Categories { get; set; }

   // Id de la catégorie sélectionnée

   public Guid IdCate { get; set; }

   // Produits de la catégorie sélectionnée

   public PageItems<Product> Produits { get; set; }

}

Dans le contrôleur, il faut également modifier le code de chargement des produits :

/// <summary>

/// Liste paginée des produits d'une catégorie donnée

/// </summary>

/// <param name="IdCate">Id de la catégorie</param>

/// <param name="page">N° de la page</param>

/// <returns></returns>

public async Task<IActionResult> Index(Guid IdCate, int page=1)

{

   // Création d'une entité visuelle fournissant la liste des catégories

   // et les produits de la catégorie demandée

   var vmProduit = new ProductsViewModel();

   // Récupération des produits de la catégorie demandée

   IQueryable<Product> produits = \_context.Product;

   if (IdCate != null && IdCate != Guid.Empty)

   {

      vmProduit.IdCate = IdCate;

      produits = produits.Where(p => p.CategoryId == IdCate);

   }

   // On ne charge qu'une page de 5 produits à la fois

   vmProduit.Produits = await PageItems<Product>.CreateAsync(

         produits.OrderBy(p => p.Name).AsNoTracking(), page, 5);

   vmProduit.Categories = await \_context.Category.OrderBy(c => c.Name)

                           .AsNoTracking().ToListAsync();

   return View(vmProduit);

}

Nous ne chargeons plus tous les produits de la catégorie sélectionnée, mais uniquement une page de 5 produits. L’indice de la page à afficher est reçu en paramètre. Sa valeur est 1 par défaut lors du premier appel à la méthode Index.

Cette méthode peut être exploitée dans la vue de la façon suivante :

<form asp-controller="Products" asp-action="Index" method="get">

   <p>

      <span>Catégorie </span>

      <select asp-for="IdCate"

              asp-items="@(new SelectList(Model.Categories,"CategoryId","Name"))">

         <option value="">Toutes</option>

      </select>

      <input type="submit" value="Filtrer" />

   </p>

</form>

<table class="table">

...

</table>

@{

   var etatPrecedent = !Model.Produits.HasPreviousPage ? "disabled" : "";

   var etatsuivant = !Model.Produits.HasNextPage ? "disabled" : "";

}

<a asp-action="Index"

   asp-route-idcate="@Model.IdCate"

   asp-route-page="@(Model.Produits.PageIndex - 1)"

   class="btn btn-default @etatPrecedent">

   Précédent

</a>

<a asp-action="Index"

   asp-route-idcate="@Model.IdCate"

   asp-route-page="@(Model.Produits.PageIndex + 1)"

   class="btn btn-default @etatsuivant">

   Suivant

</a>

Le formulaire contient une liste déroulante identique à celle que nous avons déjà utilisée. L’attribut asp-for= « IdCate » permet de renvoyer l’Id de la catégorie sélectionnée à la méthode Index du contrôleur.

Les boutons de navigation entre les pages de données (Précédent et Suivant) sont modélisés par des Tag Helpers Anchor ayant les attributs suivants :

* asp-route-idcate : représente l’id de la catégorie sélectionnée
* asp-route-page : représente l’index de la page demandée.

Ces deux informations sont renvoyées à la méthode Index du contrôleur, lorsqu’on clique sur les boutons, et elles lui permettent de filtrer les données à renvoyer.

Il est important de noter que asp-for= « IdCate » et asp-route-idcate affectent le même paramètre de la méthode du contrôleur.

L’état des boutons de navigation est géré par des classes css, représentées par les variables etatPrecedent et etatSuivant, qui sont liées aux propriétés HasPreviousPage et HasNextPage du modèle

## Filtrage

Le filtrage consiste à n’afficher que les éléments d’une collection qui satisfont un certain nombre de conditions.

Exemple : les produits de la catégorie XX et de prix unitaire supérieur à YY.

Visuellement, les critères de filtre peuvent être saisis au moyen de divers contrôles (boutons radio, case à cocher, lites déroulantes, liens, zones de saisies libres…). Le filtrage peut être lancé dès la modification d’un des critères de filtres, ou bien au clic sur un bouton unique, afin d’appliquer tous les critères de filtres d’un coup. La seconde technique a l’avantage de limiter le nombre de requêtes http exécutées, ce qui optimise les performances. Elle peut toutefois être jugée moins ergonomique que la première…

Lors de la mise en œuvre du filtrage, on est confronté à la question suivante : comment transmettre les valeurs de filtre sélectionnées par l’utilisateur à l’action du contrôleur ?

Les deux choix possibles sont :

**Par le modèle**

Souvent, l’entité utilisée comme modèle pour la vue n’est pas adaptée, car elle ne possède pas les propriétés nécessaires pour stocker les valeurs de filtre. Il faut dans ce cas utiliser une entité visuelle (View Model), comme nous l’avons vu dans le chapitre précédent

**Par l’url**

Chaque valeur de filtre sera passée en paramètre dans la partie requête de l’url (après le point d’interrogation), et récupérée dans un paramètre de l’action du contrôleur.

Cette technique est adaptée si :

* Les critères de filtre ne sont pas trop nombreux (pour limiter le nombre de paramètres dans l’action et la longueur de l’url)
* Les valeurs de filtre ne sont pas confidentielles (car elles apparaissent dans l’url)
* On ne dispose pas déjà d’une entité visuelle associée à l’entité de modèle

L’exemple ci-dessous montre comment implémenter le filtrage d’une liste de personnes selon deux critères : un morceau du nom et l’âge.

Voici le formulaire utilisé dans la vue :

<**form** **asp-controller**="Employees" **asp-action**="Index" method="get">

    <p>

        <label>Nom</label>

        <input type="text" name="texteNom" value="@ViewBag.texteNom">

        <label>Age</label>

        <**select** name="age" **asp-items**="@{new SelectList(ViewBag.Ages,  
  "Key", "Value", ViewBag.AgeSelec)}">

        </**select**>

        <input type="submit" value="Filter" />

    </p>

</**form**>

Le formulaire utilise la méthode get, ce qui signifie que les valeurs de filtres sont transmises à l’action par l’url de la requête.

Chaque valeur de filtre est transmise à l’action par un paramètre d'url nommé comme le contrôle qui lui fournit sa valeur, en l'occurrence « texteNom » pour le filtre sur le nom et « age » pour le filtre sur l’âge.

Les items de la liste déroulante sont créés au moyen d’un objet SelectList dont le constructeur prend en paramètre une collection d’objets et les noms des propriétés qui définissent la valeur et le texte des items. Ici, la collection utilisée est un simple dictionnaire transmis par le ViewBag, et on utilise les propriétés Key et Value pour définir la valeur et le texte des items.

Après soumission du formulaire par le bouton « Filtrer » et obtention de la réponse, on souhaite que les valeurs de filtres saisies restent affichées. Pour cela, l’action du contrôleur devra les mémoriser dans le ViewBag. Il faut alors restaurer ces valeurs dans la vue ; c’est ce qui est fait en affectant la propriété « value » de la zone de saisie, et en passant la valeur ViewBag.AgeSelec en dernier paramètre du constructeur de SelectList.

Le code de l’action du contrôleur est le suivant :

public async Task<IActionResult> Index(string texteNom, int age)

{

    // Valeurs pour la liste déroulante Age

    ViewBag.Ages = new Dictionary<int, string>()

            {

                { 0, "" },

                { 20, "< 20 ans" },

                { 30, "< 30 ans" },

                { 40, "< 40 ans" },

                { 1000, "> 40 ans" },

            };

    // Mémorisation des valeurs de filtres saisies

    ViewBag.AgeSelec = age;

    ViewBag.texteNom = texteNom;

    IQueryable<Employee> employés = \_context.Employee;

    // Si le critère de filtre sur le nom est spécifié, on l'applique

    if (!string.IsNullOrEmpty(texteNom))

    {

        employés = employés.Where(e => e.LastName.Contains(texteNom));

    }

    // Si le critère de filtre sur l'âge est spécifié, on l'applique

    if (age > 0)

    {

        DateTime dateFiltrage = DateTime.Today.AddYears(-age);

        employés = employés.Where(e => e.BirthDate > dateFiltrage);

    }

return View(await employés.AsNoTracking().ToListAsync());

}

On commence par remplir le dictionnaire qui sert de source de données à la liste déroulante. Puis on mémorise les valeurs de filtre saisies par l’utilisateur, afin de pouvoir les restaurer au chargement de la vue.

Enfin, on applique les filtres sur le DbSet qui sert de modèle à la vue.

**Remarques :**

On peut aussi remplir la liste déroulante des tranches d’âges directement dans la vue, comme ceci :

<**form** **asp-controller**="Employees" **asp-action**="Index" method="get">

    <p>

        <label>Nom</label><input type="text" name="texteNom">

        <label>Age</label>

        <select name="age">

            <**option** **value**="0"></**option**>

            <**option** **value**="20">&lt; 20 ans</**option**>

            <**option** **value**="30">&lt; 30 ans</**option**>

            <**option** **value**="40">&lt; 40 ans</**option**>

            <**option** **value**="1000">&gt;= 40 ans</**option**>

        </select>

        <input type="submit" value="Filter" />

    </p>

</**form**>

Par contre, il est dans ce cas plus difficile de gérer la valeur sélectionnée par défaut dans la liste.

A titre d’information, voici également la syntaxe avec un Html Helper :

@Html.DropDownList("age",  
 new SelectList(ViewBag.Ages, "Key", "Value", ViewBag.AgeSelec))

# Le routage

ASP.Net Core MVC utilise un middleware de routage pour reconnaître les url des requêtes entrantes et les mapper sur les actions. Les routes sont aussi utilisées pour générer des url de liens hypertextes envoyés dans les réponses.

Les routes sont définies de façon conventionnelle ou par des attributs.

## Routage conventionnel

Comme nous l’avons vu dans le paragraphe consacré au Model Binding, lorsqu’on crée un nouveau projet ASP.Net Core, un modèle de route par défaut est défini de la façon suivante dans la classe Startup :

app.UseMvc(routes =>

{

    routes.MapRoute(

        name: "default",

        template: "{controller=Home}/{action=Index}/{id?}");

});

MapRoute est utilisée pour créer un modèle de route.

Les valeurs spécifiées pour le contrôleur et l’action sont des valeurs par défaut si aucune valeur n’est fournie pour ces segments dans la requête http.

Le point d’interrogation définit un segment facultatif

Lorsque le serveur reçoit une requête, il utilise ce modèle pour extraire les informations des segments d’url de la requête. Ainsi, à partir de la requête dont l’url est :

http://mondomaine/Products/Details/5

…les informations suivantes sont automatiquement extraites :

controller = ProductsController, action = Details, id = 5

La requête peut être réduite au minimum à l’url « http://mondomaine », qui est équivalente à « http://mondomaine/Home/Index »

Le modèle de route établit donc une convention pour les url. Il est possible de définir plusieurs modèles de routes avec des noms différents. Dans ce cas, ils sont utilisés dans l’ordre dans lequel ils ont été définis. Les url reçues sont comparées successivement aux différents modèles, et le premier qui correspond est utilisé.

Un même contrôleur peut posséder plusieurs surcharges d’actions, susceptibles de traiter une même requête. Exemple :

public async Task<IActionResult> Edit(int? id)

public async Task<IActionResult> Edit(int id, Product product)

Ces deux actions peuvent a priori répondre à une requête dont l’url est /Products/Edit/8.  
Pour les différentier, il faut décorer l’une d’elles d’un attribut :

[HttpPost]

public async Task<IActionResult> Edit(int id, Product product)

De cette façon, l’action ci-dessus traite les requêtes de type POST, et l’autre surcharge traite les autres verbes http.

On peut spécifier plusieurs routes en utilisant Routes.MapRoute. Elles doivent avoir des noms uniques, et sont évaluées dans leur ordre d’ajout. C’est-à-dire que la première route correspondant à l’url de la requête http à traiter est utilisée pour identifier le contrôleur et l’action à exécuter.

## Routage par attributs

Si on ne définit pas de routage conventionnel en paramètre de app.UseMvc, on peut définir un routage au moyen d’attributs placés sur les actions de contrôleurs. L’exemple ci-dessous montre comment définir un routage identique au routage conventionnel vu précédemment, mais en utilisant des attributs :

public class MyDemoController : Controller

{

    [Route("")]

    [Route("Home")]

    [Route("Home/Index")]

    public IActionResult MyIndex()

    {

        return View("Index");

    }

    [Route("Home/About")]

    public IActionResult MyAbout()

    {

        return View("About");

    }

    [Route("Home/Contact")]

    public IActionResult MyContact()

    {

        return View("Contact");

    }

}

Le code du contrôleur est donc plus verbeux, mais les attributs permettent un meilleur contrôle, puisqu’on peut définir un routage spécifique à chaque action indépendamment des noms des contrôleurs et des actions.

On peut aussi combiner la route avec la méthode http, de la façon suivante (c’est ce qui est recommandé pour les API web) :

[HttpGet("Employes/Details/{id}")]

public IActionResult Details(int id)

{  
 ...

}

Ici, la route spécifie en plus un paramètre « id » obligatoire à passer dans l’url.

La route complète peut être spécifiée en deux morceaux, placés sur le contrôleur et sur l’action :

[Route("Employees")]

public class EmployeesController : Controller

{

[HttpGet("Details/{id}")]

public IActionResult Details(int id)

{  
 ...

}

}

On peut utiliser les jetons [Controller] et [Action] à l’intérieur des attributs pour désigner les contrôleurs et actions de façon générique :

[Route("[Controller]/[Action]")]

public class EmployeesController : Controller

{

[HttpGet("{id}")]

public IActionResult Details(int id)

{  
 ...

}

}

Pour une description complète de toutes les possibilités de routage, consulter [cette page de doc](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing) Microsoft.

# Les middlewares

Les middlewares sont des traitements qui gèrent les requêtes et les réponses. Ils sont distribués par packages NuGet, et on peut créer nos propres middlewares.

Un middleware exécute une action asynchrone (un délégué) sur un HttpContext, et choisit ou non de passer ensuite la requête au middleware suivant. Il peut exécuter des actions avant et après le middleware suivant. Cet enchaînement d'actions constitue le **pipeline** de l'application. Il est schématisé par la figure ci-dessous :



Les différents middlewares sont chaînés dans la méthode Startup.Configure, en appelant la méthode d'extension Use de IApplicationBuilder.

ASP.Net Core fournit par défaut 8 middlewares : authentification, CORS, cache de réponse, compression de la réponse, routage, session, fichiers statiques, réécriture d'url.

L'ordre dans lequel sont ajoutés les middlewares dans la méthode Configure est celui dans lequel ils sont appliqués sur la requête http, et l'inverse de l'ordre dans lequel ils sont appliqués sur la réponse.

Les middlewares sont exécutés avec les méthodes Map, Run et Use.

Map permet de créer des branches de pipeline, et chaque branche est exécutée en fonction de l'url de la requête.

# La sécurité

## Identification, authentification et autorisation

**Identification**

Une application peut exiger que ses utilisateurs soient identifiés pour qu’ils puissent l’utiliser. L’identification se traduit par la création d’un compte, c’est-à-dire la mémorisation d’informations sur l’identité de l’utilisateur, telles que nom et prénom, identifiants, mot de passe, adresse postale, numéro de téléphone, numéro de sécurité sociale, numéro SIREN, empreinte digitale…etc.

**Authentification**

L'authentification est le processus qui permet de vérifier si un utilisateur est autorisé à accéder à l’application ou à une partie de l’application.

Pour cela, l’utilisateur doit donc déjà être connu du système, c’est-à-dire posséder un compte. Pour s’authentifier, il doit fournir la preuve que c’est bien lui qui tente de se connecter, en fournissant certaines informations qui seront comparées à celles détenues par le système. Cela se traduit généralement par la saisie d’un login et d’un mot de passe, ou par la présentation à un système biométrique tel qu’un lecteur d’empreinte digitale.

**Autorisation**

Les autorisations déterminent ce qu’un utilisateur a le droit de faire dans l’application. Par exemple, un administrateur pourrait accéder à certains paramètres de l’application, qui seraient inaccessibles pour les autres utilisateurs.

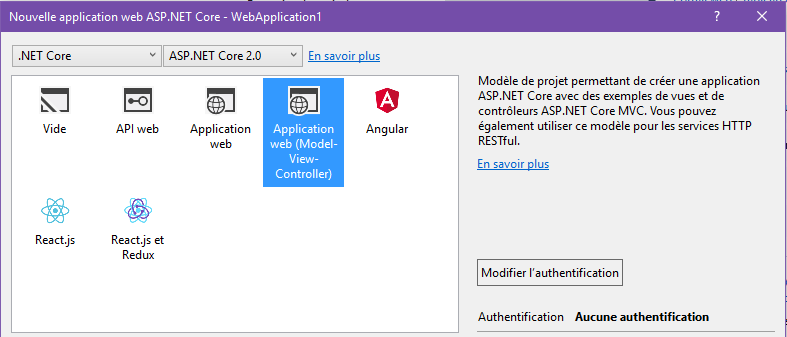
Les autorisations sont modélisées par des rôles et des droits. Pour autoriser ou non l’accès à une fonctionnalité, l’application doit comparer les droits requis par cette fonctionnalité à ceux détenus par l’utilisateur.

## ASP.NET Core Identity

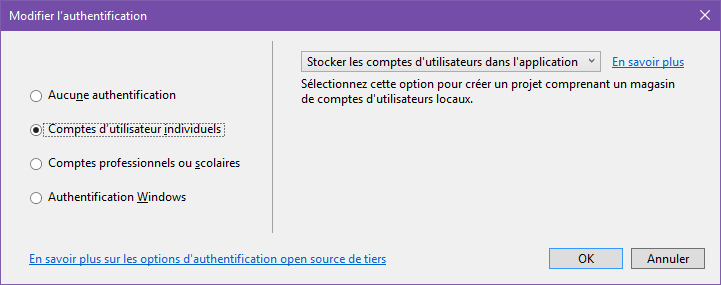
Nous allons voir maintenant comment implémenter l’identification et l’authentification dans une application web avec ASP.NET Core Identity, un système intégré à .Net Core.

### Choix du système d’authentification

La fenêtre de création d’une nouvelle application ou API web contient un bouton « Modifier l’authentification » :



Un clic sur ce bouton ouvre la fenêtre suivante :



Cette fenêtre permet de sélectionner un système d’authentification existant en fonction du type d’utilisateurs visé par l’application (comptes individuels, professionnels, ou utilisateurs Windows).

Les systèmes proposés sont un magasin local (tel qu’une base de données), Active Directory, une application Azure, Office 365 ou l’authentification Windows. Mais il est également possible d’utiliser les fournisseurs d’identité Microsoft, Facebook, Twitter ou Google. Il faut dans ce cas sélectionner les options présentées dans la capture ci-dessus, et il sera ensuite nécessaire de faire un paramétrage dans une interface de gestion sur le site du fournisseur choisi.

Dans la suite de ce cours, nous présenterons comment gérer des comptes individuels dans une base de données SQL Server locale. Les options à sélectionner sont dans ce cas également celles présentées dans la capture ci-dessus.

### Génération de l’architecture de l’application

Lorsqu’on valide notre choix dans la fenêtre précédente, Visual Studio génère automatiquement une architecture de code pour l’authentification, comme le montre la capture d’écran ci-dessous :

|  |  |
| --- | --- |
|  | Contrôleurs pour gérer la création de compte, la connexion, la déconnexion, la modification ou l’envoi d’un nouveau mot de passe.  Code de migration de base de données pour créer les tables nécessaires à la gestion de l’authentification  ViewModels utilisés par les contrôleurs décrits plus haut  Vues associées aux contrôleurs décrits plus haut  NB/ Les vues générées contiennent par défaut des textes en anglais qu’il faudra modifier ou rendre traductibles. |

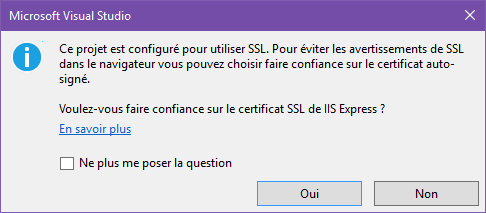
Entity Framework Core est utilisé pour interagir avec la base de données qui contiendra les tables nécessaires à la gestion de l’authentification.

Dans le fichier de configuration appsettings.json, une chaîne de connexion a été créée automatiquement pour la connexion à cette base. On peut bien sûr modifier le nom de la base, et éventuellement préciser le nom d’une base déjà existante.

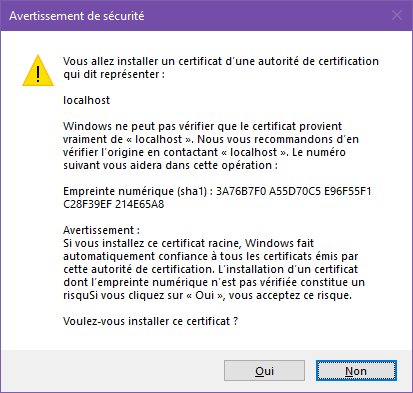
Pour créer les tables dans la base, il faut utiliser les classes de migration générées dans le dossier Data\Migrations. On exécute pour cela la commande « **update -database** » dans la console du gestionnaire de packages. On peut ensuite vérifier la création des tables dans SSMS :

|  |  |
| --- | --- |
|  | Les tables générées commencent toutes par « AspNet », de façon à ne pas entrer en conflit avec les noms des tables éventuellement déjà existantes.  Toutes ces tables sont vides. |

Lorsqu’on lance l’application, la boîte de dialogue suivante s’affiche :

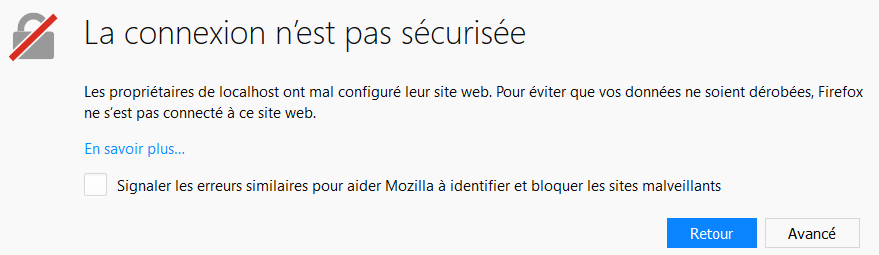


En cliquant sur Oui, le message d’avertissement suivant s’affiche du fait que le certificat est auto-signé :



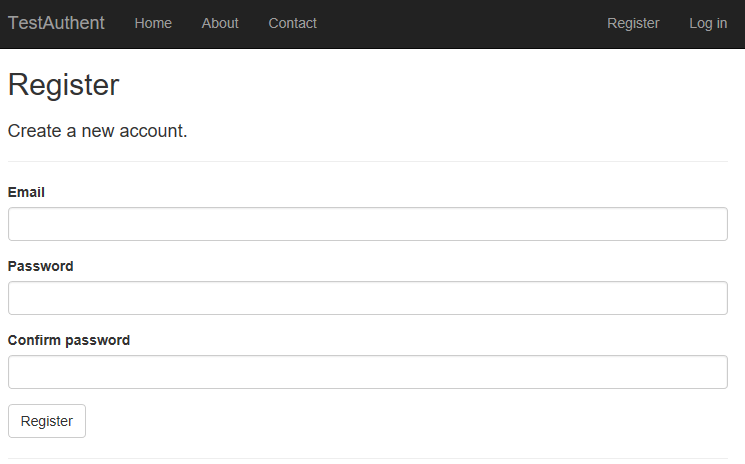
Cliquer sur oui

Le navigateur peut éventuellement afficher un message d’avertissement supplémentaire :



… auquel cas, il faut ajouter une exception de sécurité en cliquant sur Avancé pour pouvoir exécuter l’application.

L’application contient par défaut des entrées de menus et des pages pour la création d’un compte et pour l’authentification :



### Règles de gestion sur les mots de passes

Il est possible de définir des restrictions qui s’appliquent sur le mot de passe lors de la création d’un compte. Pour cela, dans la méthode ConfigureServices de la classe Startup, il faut configurer des paramètres d’identification, comme le montre l’extrait de code suivant :

public void ConfigureServices(IServiceCollection services)

{

    services.AddDbContext<ApplicationDbContext>(options =>

       options.UseSqlServer(Configuration.GetConnectionString("DefaultConnect")));

    services.AddIdentity<ApplicationUser, IdentityRole>()

        .AddEntityFrameworkStores<ApplicationDbContext>()

        .AddDefaultTokenProviders();

    services.Configure<IdentityOptions>(options =>

    {

        // Password settings

        options.Password.RequireDigit = true;

        options.Password.RequiredLength = 8;

        options.Password.RequireNonAlphanumeric = true;

        options.Password.RequireUppercase = true;

        options.Password.RequireLowercase = false;

        options.Password.RequiredUniqueChars = 6;

        // Lockout settings

        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);

        options.Lockout.MaxFailedAccessAttempts = 10;

        options.Lockout.AllowedForNewUsers = true;

        // User settings

        options.User.RequireUniqueEmail = true;

    });

Ce code définit les règles suivantes sur le mot de passe :

* Il doit contenir des chiffres
* Il doit faire au moins 8 caractères
* Il doit contenir au moins un caractère non alphanumérique
* Il doit contenir au moins une majuscule
* Il doit avoir au moins 6 caractères différents

NB/ Les propriétés suivantes ont la valeur « True » par défaut :  
RequireDigit , RequireNonAlphanumeric, RequireUppercase.   
Donc, si on ne modifie pas les paramètres, les restrictions correspondantes sont appliquées par défaut.

En-dessous figurent des règles concernant le blocage de l’utilisateur : l’utilisateur sera bloqué pendant 30 minutes si son authentification a échoué 10 fois de suite.

### Demande d’authentification

La gestion des autorisations la plus basique consiste à donner accès à l’application complète uniquement aux utilisateurs authentifiés. Ceci peut être géré de façon simple par le code suivant dans la méthode Startup.ConfigureServices :

services.AddMvc(config =>

{

    var policy = new AuthorizationPolicyBuilder()

        .RequireAuthenticatedUser()

        .Build();

    config.Filters.Add(new AuthorizeFilter(policy));

});

Ce code consiste à utiliser un **filtre d’autorisation**, c’est-à-dire une méthode qui sera exécutée automatiquement avant chaque action de contrôleur, et qui pourra empêcher l’exécution de ces dernières si l’utilisateur n’est pas authentifié.

Une gestion un peu plus fine des autorisations consiste à préciser quelles actions de contrôleurs requièrent que l’utilisateur soit authentifié. Elle est mise en œuvre de façon simple à l’aide d’attributs sur les contrôleurs et leurs actions, comme le montre l’exemple ci-dessous :

[Authorize]

public class HomeController : Controller

{

    [AllowAnonymous]

    public IActionResult Index()

    {

        return View();

    }

    [AllowAnonymous]

    public IActionResult About()

    {

        ViewData["Message"] = "Your application description page.";

        return View();

    }

    public IActionResult Contact()

    {

        ViewData["Message"] = "Your contact page.";

        return View();

    }

    [AllowAnonymous]

    public IActionResult Error()

    {

        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });

    }

}

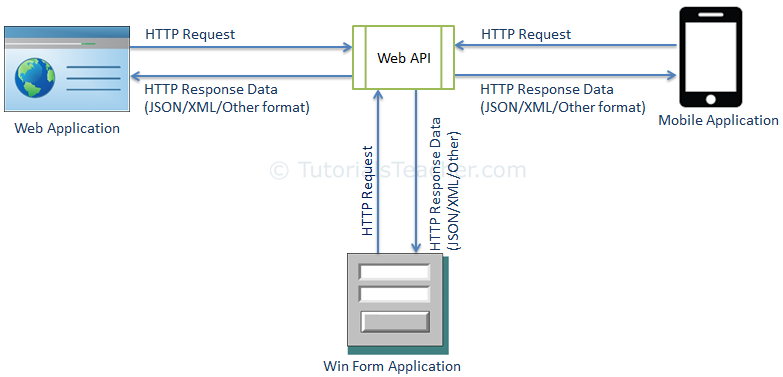
En mettant l’attribut Authorize sur la classe du contrôleur, on requiert que l’utilisateur soit authentifié pour exécuter n’importe quelle action de ce contrôleur. On peut cependant ajouter l’attribut AllowAnonymous sur certaines actions pour autoriser l’accès à ces dernières aux utilisateurs non authentifiés. Finalement, dans le code précédent, seule l’action Contact requerra que l’utilisateur soit authentifié. On pourrait donc implémenter ceci en mettant un unique attribut Authorize sur cette action.

# API Web

Une API Web est un ensemble de méthodes accessibles par le protocole http. Elle peut être créée avec différentes technologies, telles que Java, .Net, PHP…

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.

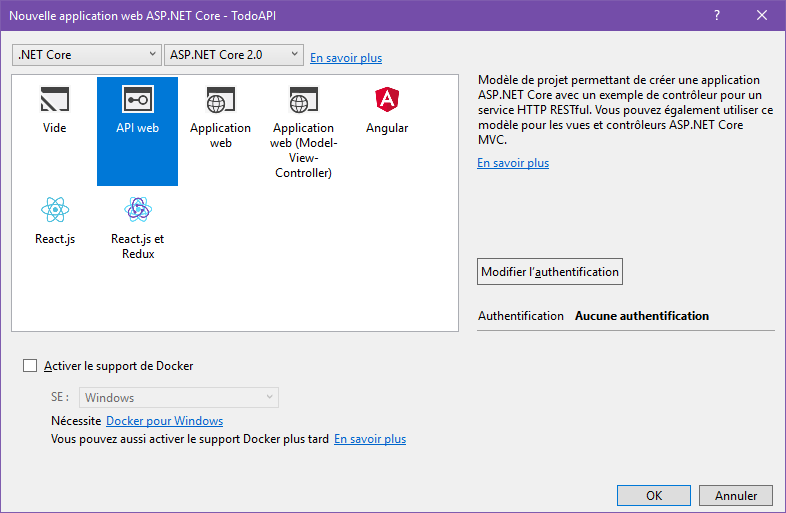
L’API peut être exploitée par des applications de bureau classiques, web ou mobiles, comme illustré par le schéma ci-dessous.



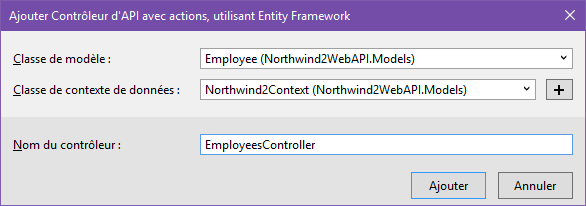
ASP.Net Web API est une technologie de création d’API Web ayant les caractéristiques suivantes :

* Basée sur ASP.Net, et prenant en charge le pipeline de ce type d’appli
* Prend en charge les formats de réponses JSON, XML et BSON
* Fournit la classe HttpClient pour communiquer avec le serveur.

## Création d’une API Web



Ajout d’un contrôleur



La fenêtre est simplifiée par rapport à la création d’un contrôleur d’une appli web, puisqu’il n’y a pas la partie génération des vues.

## Actions

Hormis les méthodes spéciales telles que le constructeur, des surcharges d’opérateurs…etc., toutes les méthodes publiques de la classe sont considérées comme des actions.

Pour spécifier à quel verbe http une action répond, on peut :

* Simplement nommer la méthode comme le verbe : Get, Post, Put…
* La décorer d’un attribut HttpXXX ou AcceptVerbs

Sinon, les méthodes répondent par défaut au verbe Post.

/!\ Si plusieurs actions peuvent répondre à la même requête, leurs appels échoueront avec une erreur 500

## Réponse http

Le résultat de l’action est un objet dérivé de [StatusCodeResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.statuscoderesult?view=aspnetcore-2.0). Il est automatiquement sérialisé en JSON et renvoyé dans le corps de la réponse http.

Une valeur numérique est également placée dans l’en-tête pour indiquer la nature du résultat. Elle est fournie par la propriété StatusCode de la classe StatusCodeResult. Il s’agit d’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.

## Test de l’API avec PostMan

Les requêtes http de type Get peuvent être testées dans un navigateur car les données à passer éventuellement à la requête sont saisies dans l’url. Les requêtes http de type Post et PUT nécessitent en revanche de spécifier un corps de requête, ce qui ne peut pas être fait directement dans un navigateur.

PostMan est un logiciel gratuit permettant de tester tous les types de requêtes.

**Requête POST**

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

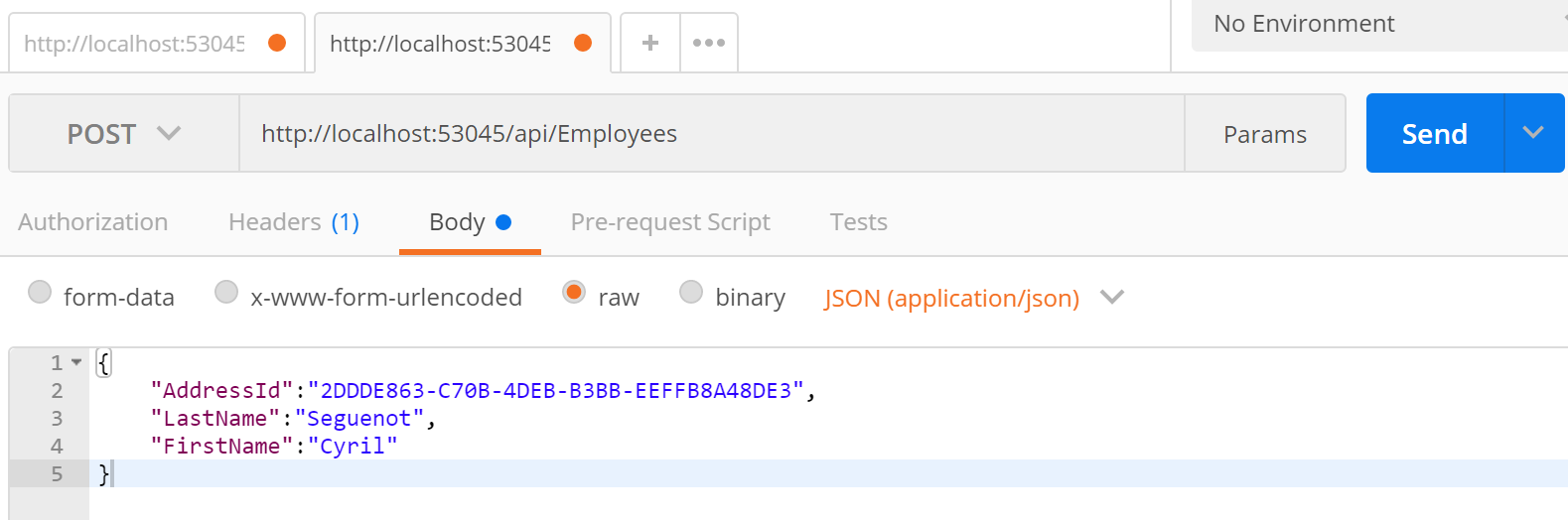
// POST: api/Employees

[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 :



Le corps de la requête est créé dans l’onglet Body. On spécifie le format JSON pour les données qui seront envoyées, et on sélectionne le bouton radio « raw » (brut) de façon à saisir les données sous forme d’un flux JSON brut.

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.

**Requête PUT**

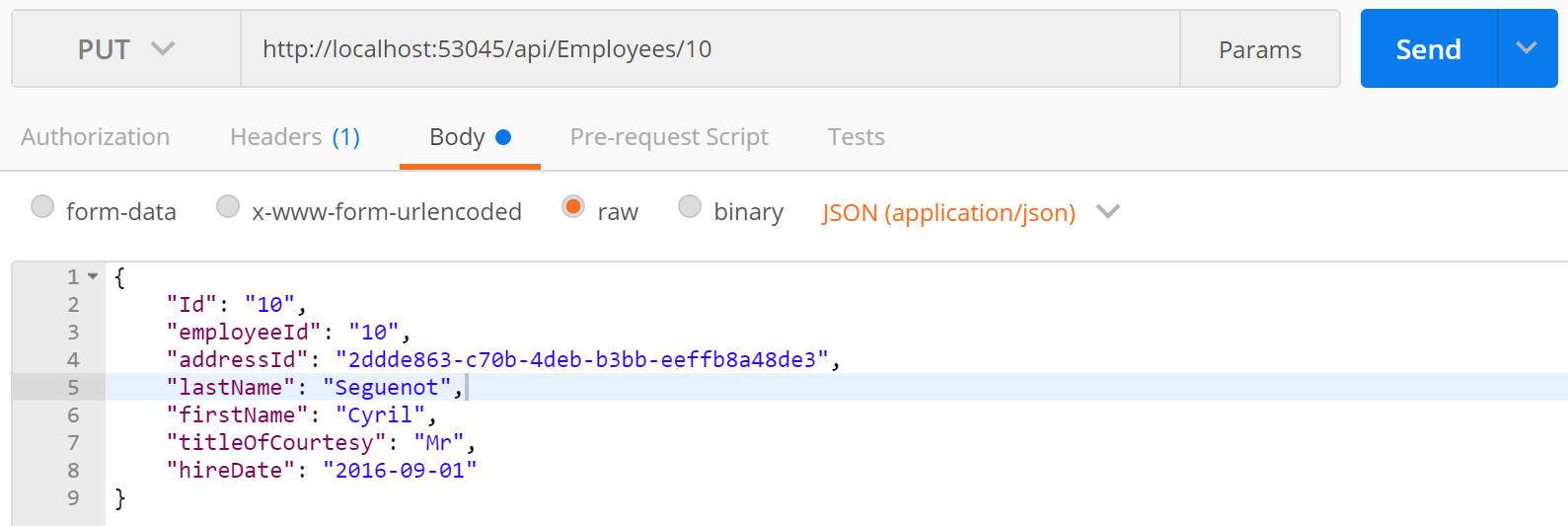
Pour tester la mise à jour d’un employé via l’action suivante :

// PUT: api/Employees/5

[HttpPut("{id}")]

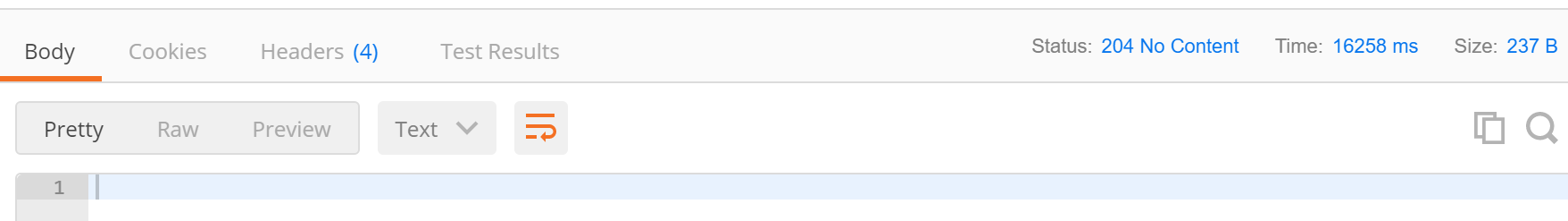
public async Task<IActionResult> PutEmployee([FromRoute] int id, [FromBody] Employee employee)

… 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.

**Requête DELETE**

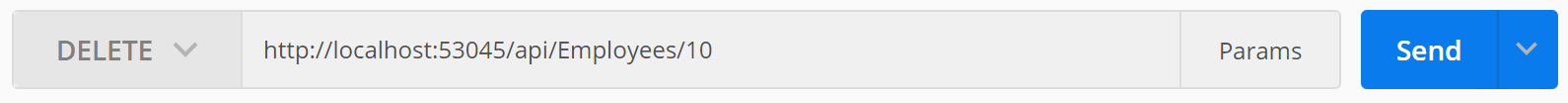
Pour tester la suppression d’un employé via l’action suivante :

// DELETE: api/Employees/5

[HttpDelete("{id}")]

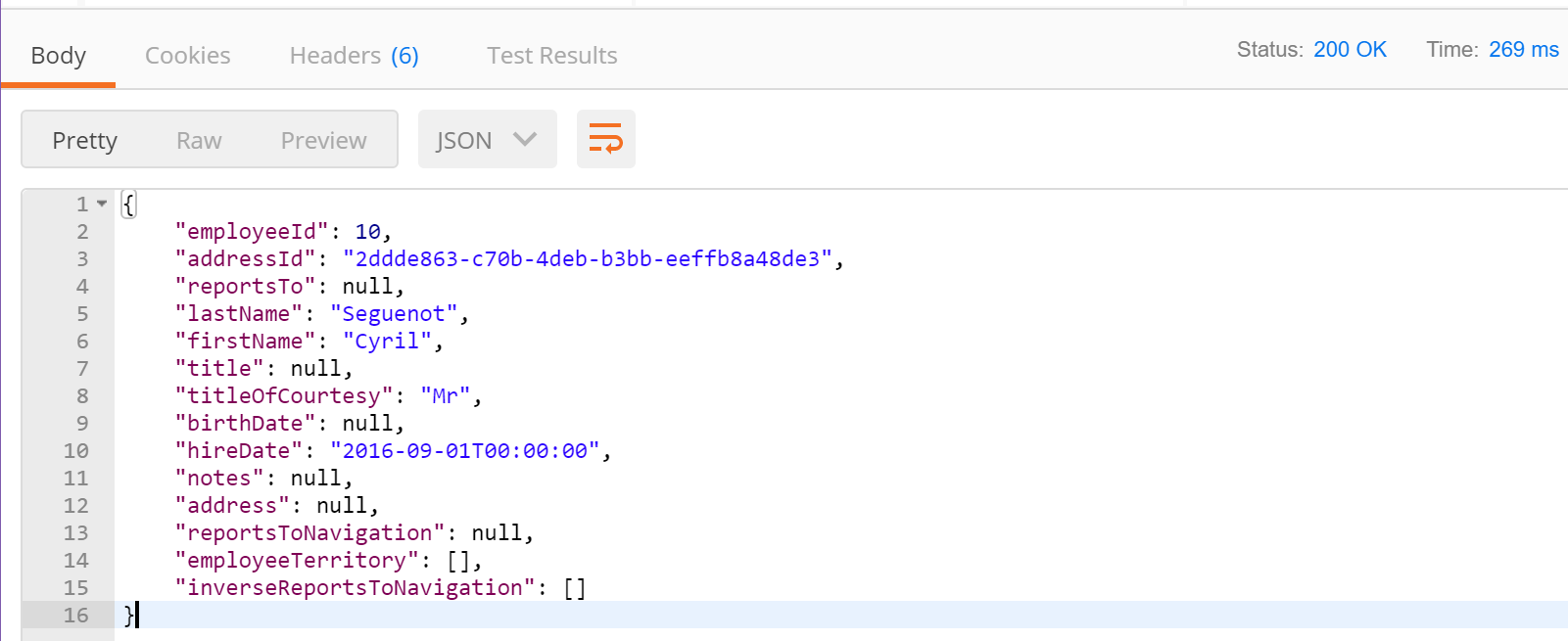
public async Task<IActionResult> DeleteEmployee([FromRoute] int id)

… 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.

## Création d’une application cliente

Pour créer une application console cliente de l’API Web :

* Ajouter le package Nuget suivant au projet avec la console du gestionnaire de packages :  
  Install-Package Microsoft.AspNet.WebApi.Client
* Ajouter des références vers system.net et System.Net.Http
* 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/)

[Liste des Tag Helpers](https://docs.microsoft.com/en-us/aspnet/core/api/microsoft.aspnetcore.mvc.taghelpers)

[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)

[Tutoriel ASP.Net MVC sur TutorialsTeacher.com](http://www.tutorialsteacher.com/mvc/asp.net-mvc-tutorials)

[Documentation de référence Bootstrap 4](https://getbootstrap.com/docs/4.0/getting-started/introduction/)

[Tutoriel W3Schools sur Bootstrap 4](https://www.w3schools.com/bootstrap4/)