**ASP.NET MVC CORE 3.1**

**Documentation du projet  SportsStore**

**2020-2021**

Nom et prénom de l’étudiant : Barman Rayan

# Configuration

La classe « **Startup** » est responsable de la configuration de l’application. Elle a deux méthodes :

* ConfigureServices () : cette méthode est utilisée pour configurer les « services » de l’application. Un service est un composant réutilisable qui fournit des fonctionnalités d’application. Les « services » sont utilisées dans l’application grâce à l’injection de dépendance.
* Configure () : Cette méthode permet de configurer les middlewares. Les composants middleware permettent de traiter les requêtes http et même de produire la réponse à la requête. L’application .NET reçois des requêtes http qui les passes ensuite à un « Request pipeline » qui contient tous les composant middleware de l’application.

(parler plus en détails des middleware et des services grâce au chapitre 12)

**Razor** est responsable du traitement des fichiers «.cshtml », il permet de générer du code HTML qui est compris par le navigateur. Au sein des fichiers Razor, il y a du code html et du code C# qui contient un peu de logique.

**L’Entity Framework Core** est un ORM qui permet de faire du mapping avec les données dans la base de données et les modèles de l’application. Il fournit un accès à la base de données grâce à une classe « context » qui représentera les données dans la base de données.

On configure aussi dans la classe Startup la connexion à la base de données grâce à l’objet de type IConfiguration, nous avons accès au fichier json dans laquelle nous pouvons récupérer le nom de la base de données.

La création de la base de données se fait sur base des modèles grâce à la fonctionnalité migration. Un fichier sera créé avec le code Sql qui permet de créer la base de données.

Quand ASP.NET Core a besoin de créer une nouvelle instance du HomeController pour traiter les requêtes http, il inspectera le constructeur et verra qu’il a besoin d’un objet de type IStoreRepository. Pour déterminer l’implémentation de l’objet, ASP.NET Core consulte la configuration dans la classe Startup, qui lui donne l’implémentation à utiliser. Ainsi chaque requête créera une nouvelle instance de EFStoreRepository. On utilise ici le mécanisme de l’injection de constructeur car on utilise l’interface et pas l’implémentation directement. Comme une interface ne doit jamais, on aura aucun changement si on change d’implémentation.

Une « **view model class** » est une classe spécifique pour passer des données entre un contrôler et une vue. Dans notre projet la classe qui fait ce travail est : **ProductsListViewModel**. Elle contient une collection de produits, …

Pour permettre au client de naviguer à travers les pages d’une liste de produits sans devoir mettre dans l’Url la page qu’on souhaite, nous mettons en places un **Tag Helper** qui permettra de naviguer de pages en pages grâce à un système de « lien » avec la balise <a>. Les Tag helpers sont très utilisé en ASP.NET Core car elle permet de générer du code C# et du code HTML. Ça permet d’éviter la redondance de code. On peut voir ça comme une méthode dans lequel on donne des paramètres.

L’application à différents endpoints qui permet d’accéder à une ressource que l’on souhaite. Pour chaque endpoint il faut définir son URL ainsi que le contrôler qui est rattacher à ce endpoint et la méthode du contrôler. C’est cette méthode qui s’occupera de traiter la requête http venant sur ce endpoint. Le client envoie une requette http en spécifiant le endpoint grâce à ….

Un **view components** est un composant qui peut être réutiliser plusieurs fois. Comme nous voulons que le menu de navigation apparaisse sur plusieurs pages nous le mettons dans un composant réutilisable.

**Razor Pages** est un Framework alternatif au Framework MVC. Tout d’abord il faut configurer Razor Pages. Pour cela il faut aller dans la classe startup et rajouter Razor Pages aux services et le rajouter comme endpoint pour qu’on puisse accéder à la page comme celle-ci ne sera lier à aucun contrôleur. C’est elle-même qui traite les requêtes. Razor pages à ses propres fichiers de configuration dont son propre layout. Un Razor page se distingue par une mention « @page » tout au-dessus. La page razor à une « page model class » qui lui est associé, elle lui permet de gérer toutes les requêtes http.

Activer une session quand on sur l’application pour ne pas perde ses données : Pour activer une session dans l’application il faut se rendre dans le fichier Startup et y ajouter dans la méthode ConfigureServices AddDisrtibutedMemoryCache() qui permet de mettre en place le stockage des données en mémoire. La méthode AddSession() permet d’accéder aux données de la session. UseSession() permet d’associer les requêtes du client à une seul session.

Dans le chapitre 9, nous enlevons de la responsabilité à la page Razor Cart, ce n’est plus lui qui s’occupe de la session car il aurait fallu dupliquer du code pour faire la même chose dans une autre page Razor. La classe qui va stocker le panier est une classe qui hérite du modèle de base.

Les « **view component** » sont des fragments d’HTML qui sont inclut dans le layout. Les view components sont similiare au partial view car elles sont aussi intégrées dans un code HTML déjà existant. Elles sont puissantes car quand une modification est faite alors elles seront mises à jour automatiquement.

**Blazor** est un framework qui permet de créer des interface web dynamiquement en utilisant du C# et de l’HTML. Blazor utilise les Razor Page pour fournir le contenu au navigateur, le contenu contient le JS qui est rattaché au server et fait un rendu du Blazor HTML. Les pages Razor sont mise à jour sans faire de requête http.

«endpoints.MapFallbackToPage("/admin/{\*catchall}", "/Admin/Index"); » permet à ce que chaque url préfixer de « /admin » est rediriger vers /Admin/Index qui est la page par défaut de l’Admin. Dans la page « Index.cshtml » nous faisons appel à un composant Razor qui va permettre de voir si l’url que nous avons écrit est bon, dans le cas où c’est bon il va utiliser l’url courante pour localiser un composant Razor et ensuite nous l’afficher. Dans le cas où ce n’est pas bon, un message d’erreur sera affiché. Tout passe par index.cshtml. Tous les composant razor contiennent l’extension «.razor» et sont appelé grâce à la directive « @page » qui correspond à l’url. Pour changer de composant Razor, donc pour afficher autre chose on fait appel à des NavLink qui permettent d’appliquer un autre composant Razor correspondant au nouvel URL sans faire de nouvelle requête http.

Dans le **composant Razor** OrderTable, il y a 4 propriétés dans une balise « @code » qui contiennent toutes les 4 l’attribut [Parameter]. Cela veut dire que les valeurs leurs seront transmise à l’exécution par le composant parent qui est « Orders.razor ».

La force des Composant Razor c’est qu’ils peuvent vivre longtemps et traiter plusieurs interactions avec l’utilisateur sans faire de nouvelle requête http. Dans un Blazor Component nous ne pouvons pas envoyer des données avec la requête http POST.

Chapitre 11 :

Pour avoir un système d’authentification on va utiliser l’ASP.NET Core Identity framework. Pour cela, il faut créer une seconde base de données.

L’attribut [**Authorize**] permet de restreindre l’accès ; tout le monde n’aura donc pas accès à la page des admins. Quand un utilisateur essaye d’accéder à l’URL « /admin » il est redirigé vers l’url « Account/login » où on lui demande de lui fournir un login et mot de passe. (Comment l’application sait qu’il faut redirigé vers cette url ? et pq pas mettre l’url dans startup ? on ne met dans startup que les url dynamiques ? comment la méthode login reçoit returnUrl ? )

[ValidateAntiForgeryToken] permet de se prémunir contre les attaques

AMELIORATION A FAIRE :

* Ne pas se diriger directement sur la page cart après avoir introduit qq chose dans le panier. Mettre une notification flottante en haute à droite qui disparait après 5 sec