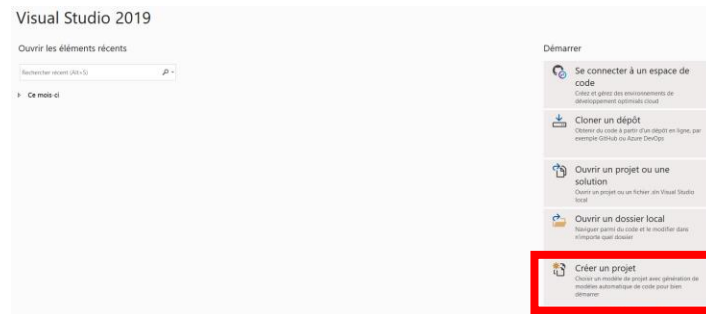


ASP.NET Core MVC

Créer une application web

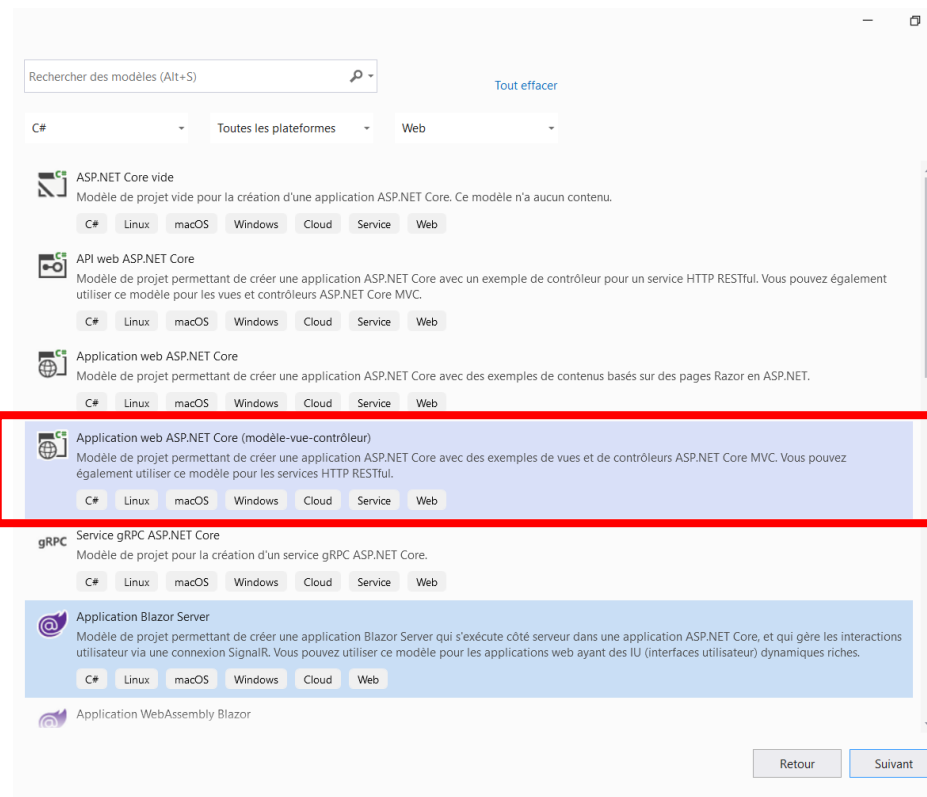
Dans Visual Studio, sélectionnez **Créer un Projet**.



Pour sélectionner le modèle de notre projet, nous utiliserons le filtre des modèles :

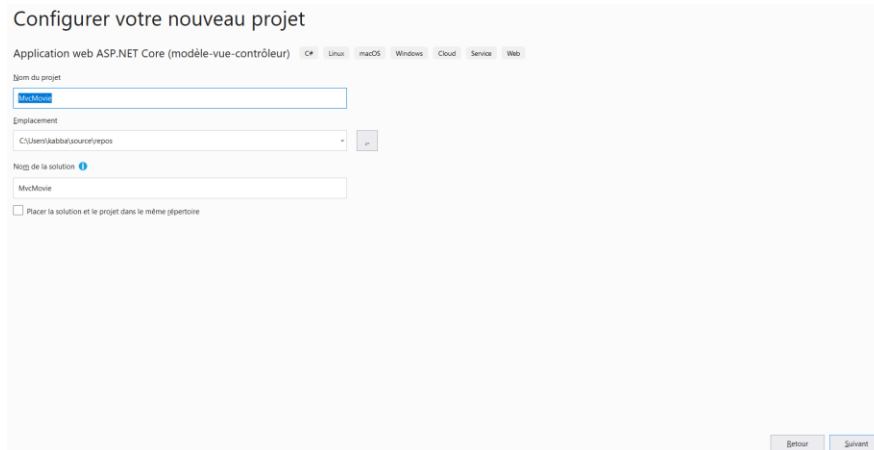
Sélectionnez **C#** comme le langage de programmation et **Web** comme type de projet.

Sélectionnez le modèle **Application web ASP.NET Core (modèle-vue-contrôleur)**



Renseignez la boîte de dialogue **Configurer un nouveau projet** :

- Nommez le projet « **MvcMovie** » (ceci est important pour que l'espace de noms corresponde quand vous copierez le code).



Configurer votre nouveau projet

Application web ASP.NET Core (modèle-vue-contrôleur) C# Linux macOS Windows Cloud Service Web

Nom du projet
MvcMovie

Emplacement
C:\Users\abba\source\repos

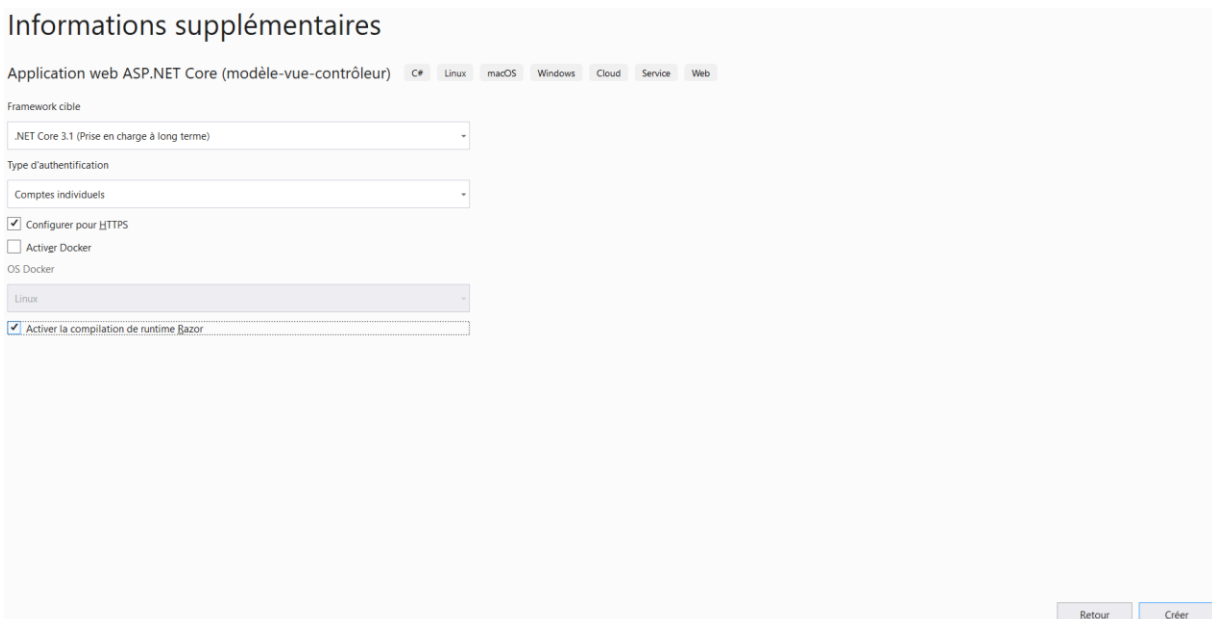
Nom de la solution
MvcMovie

☐ Placer la solution et le projet dans le même répertoire

Retour Suivant

Renseignez la boîte de dialogue **Informations supplémentaires** :

- Sélectionnez **.NET Core 3.1** comme Framework cible (si vous n'avez pas la version 3 installée sélectionnez alors la version 2)
- Comme Type d'authentification, sélectionnez **Comptes individuels**.
- Sélectionnez les deux options :
 - Configurer pour HTTPS
 - Activer la compilation de runtime Razor
- Sélectionnez **Créer**



Informations supplémentaires

Application web ASP.NET Core (modèle-vue-contrôleur) C# Linux macOS Windows Cloud Service Web

Framework cible
.NET Core 3.1 (Prise en charge à long terme)

Type d'authentification
Comptes individuels

☒ Configurer pour HTTPS

☐ Activer Docker

OS Docker
Linux

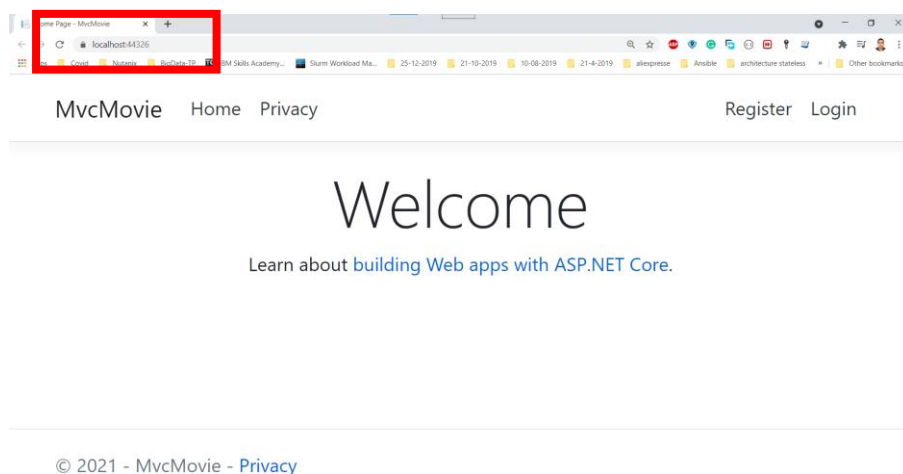
☒ Activer la compilation de runtime Razor

Retour Créer

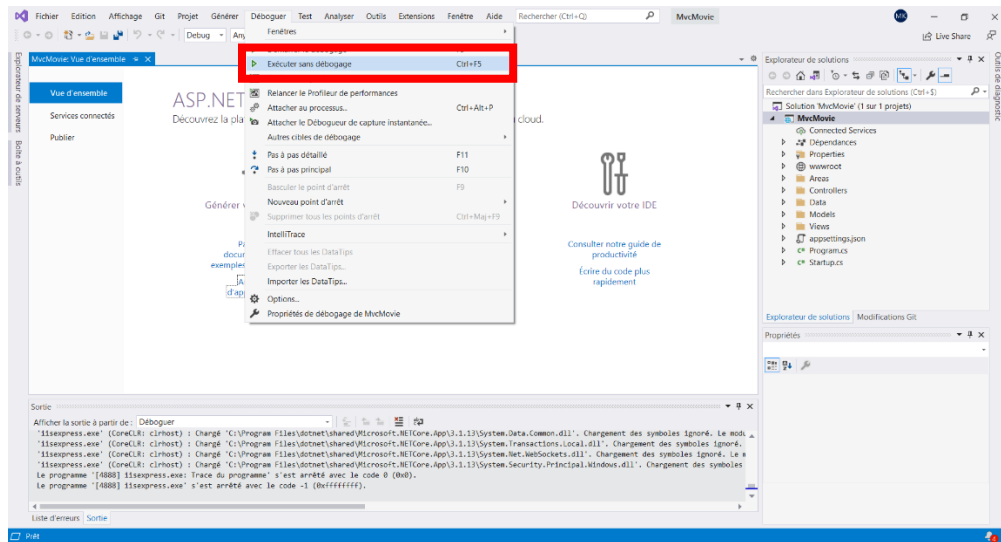
Visual Studio a utilisé un modèle par défaut pour le projet MVC que vous venez de créer. Vous disposez maintenant d'une application fonctionnelle en entrant un nom de projet et en sélectionnant quelques options. Il s'agit d'un projet de démarrage de base qui constitue un bon point de départ.

Sélectionnez **Ctrl-F5** pour exécuter l'application en mode non-débogage. **Faite confiance** à la certification SSL de IIS Express.

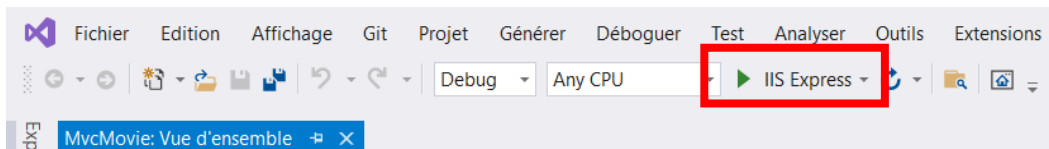
- Visual Studio démarre IIS Express et exécute votre application. Notez que la barre d'adresse affiche `localhost:port#`, et non quelque chose comme `example.com`. C'est parce que `localhost` est le nom d'hôte standard de votre ordinateur local. Quand Visual Studio crée un projet web, un port aléatoire est utilisé pour le serveur web. Dans l'image ci-dessus, le numéro de port est 44326. L'URL dans le navigateur affiche `localhost:44326`. Quand vous exécutez l'application, vous voyez un autre numéro de port.



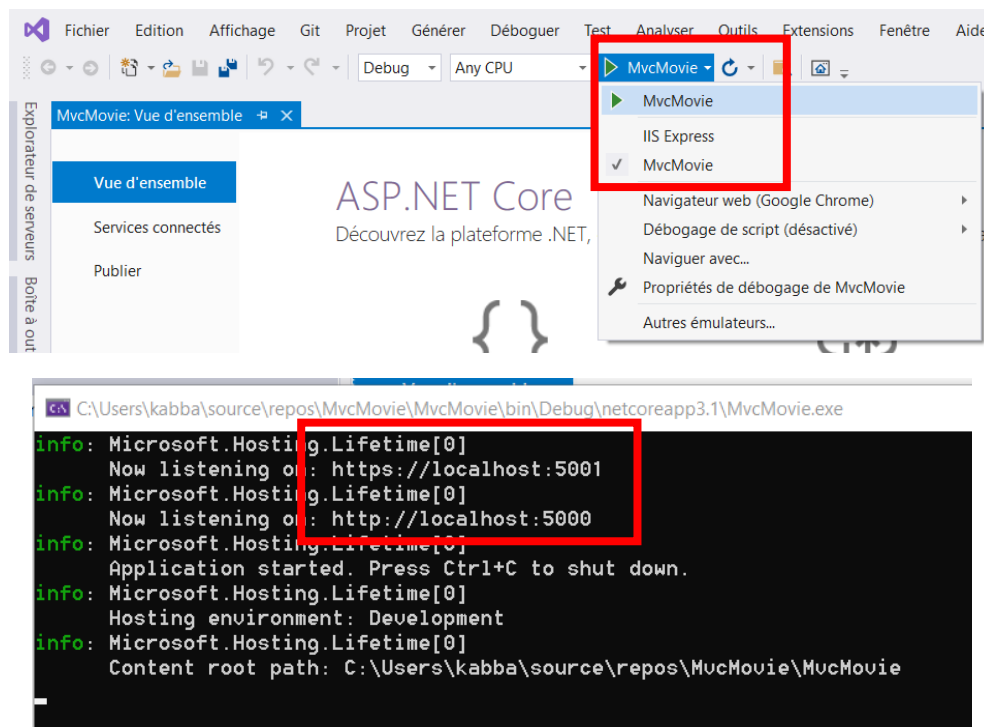
- Si vous lancez l'application avec **Ctrl+F5** (mode sans débogage), vous pouvez **effectuer des modifications du code, enregistrer le fichier, actualiser le navigateur et afficher les modifications du code**. De nombreux développeurs préfèrent utiliser le mode sans débogage pour lancer rapidement l'application et voir les modifications.
- Vous pouvez lancer l'application en mode débogage ou sans débogage à partir de l'élément de menu **Déboguer** :



- Vous pouvez déboguer l'application en sélectionnant le bouton **IIS Express**



- Vous pouvez déboguer l'application en utilisant un serveur web embarqué sélectionnant l'option **MvcMovie** puis cliquez sur bouton **MvcMovie** (l'application dans ce cas vas tournée sur les ports **5000** pour **http**, et **5001** pour **https**)



Ajouter un contrôleur à une application ASP.NET Core MVC

Le modèle d'architecture MVC (Model-View-Controller) sépare une application en trois composants principaux : **M**odèle, **V**ue et **C**ontrôleur. Le modèle MVC vous permet de créer des applications qui sont plus faciles à tester et à mettre à jour que les applications monolithiques traditionnelles. Les applications basées sur MVC contiennent :

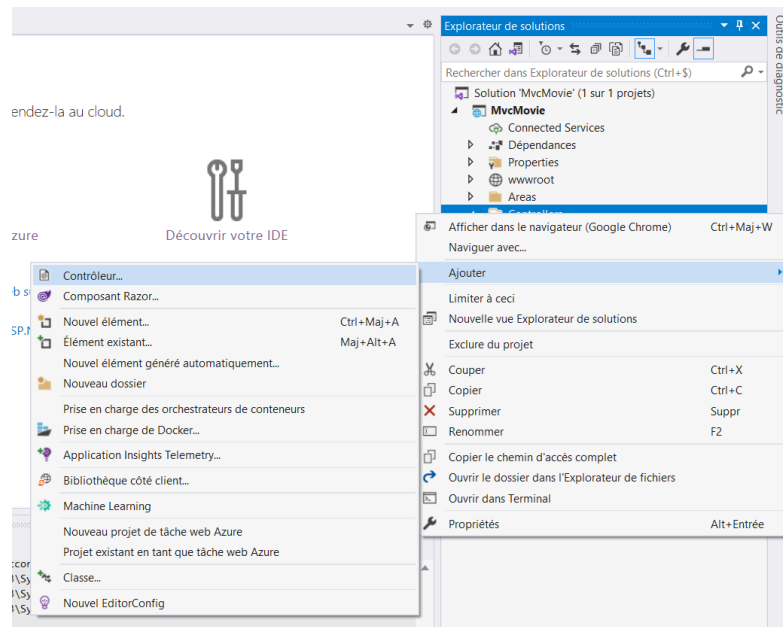
- **Modèles** : des classes qui représentent les données de l'application. Les classes du modèle utilisent une logique de validation pour appliquer des règles d'entreprise à ces données. En règle générale, les objets du modèle récupèrent et stockent l'état du modèle dans une base de données. Dans ce lab, un modèle `Movie` récupère les données des films dans une base de données, les fournit à la vue ou les met à jour. Les données mises à jour sont écrites dans une base de données.
- **Vue** : Les vues sont les composants qui affichent l'interface utilisateur de l'application. En règle générale, cette interface utilisateur affiche les données du modèle.
- **Contrôleurs** : les classes qui gèrent les demandes du navigateur. Ils récupèrent les données du modèle et appellent les modèles de vue qui retournent une réponse. Dans une application MVC, la vue affiche seulement les informations ; le contrôleur gère et répond aux entrées et aux interactions de l'utilisateur. Par exemple, le contrôleur gère les valeurs des données de routage et des chaînes de requête, et passe ces valeurs au modèle. Le modèle peut utiliser ces valeurs pour interroger la base de données. Par exemple, `https://localhost:1234/Home/About` a comme données de routage `Home` (le contrôleur) et `About` (la méthode d'action à appeler sur le contrôleur Home). `https://localhost:1234/Movies/Edit/5` est une demande de modification du film avec l'ID 5 à l'aide du contrôleur movie. Les données de routage sont expliquées plus loin dans le lab.

Le modèle MVC vous permet de créer des applications qui séparent les différents aspects de l'application (logique d'entrée, logique métier et logique de l'interface utilisateur), tout en assurant un couplage faible entre ces éléments. Le modèle spécifie l'emplacement de chaque type de logique dans l'application. La logique de l'interface utilisateur appartient à la vue. La logique d'entrée appartient au contrôleur. La logique métier appartient au modèle. Cette séparation vous aide à gérer la complexité quand vous créez une application, car elle vous permet de travailler sur un aspect de l'implémentation à la fois, sans impacter le code d'un autre aspect. Par exemple, vous pouvez travailler sur le code des vues de façon indépendante du code de la logique métier.

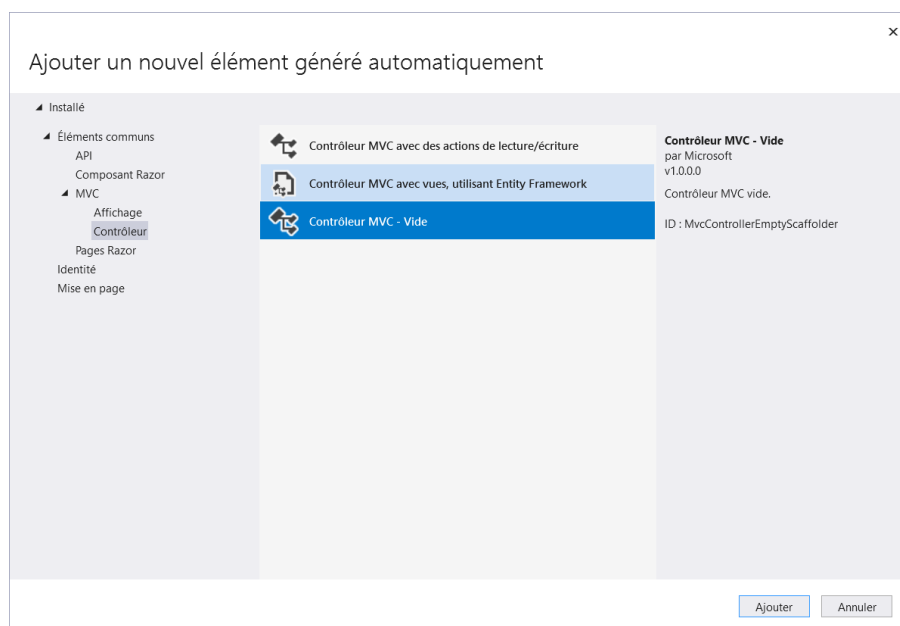
Nous présentons ces concepts dans ce lab et nous vous montrons comment les utiliser pour créer une application de gestion de films. Le projet MVC contient des dossiers pour les *contrôleurs* et pour les *vues*.

Ajouter un contrôleur

Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le **Contrôleurs** > **Ajouter** > **Contrôleur**



- Dans la boîte de dialogue **Ajouter un nouvel élément généré automatiquement**, sélectionnez **Contrôleur MVC – vide**



Dans la **boîte de dialogue Ajouter un contrôleur MVC vide**, entrez **HelloWorldController** et sélectionnez **AJOUTER**.

Remplacez le contenu de *Controllers/HelloWorldController.cs* par ceci :

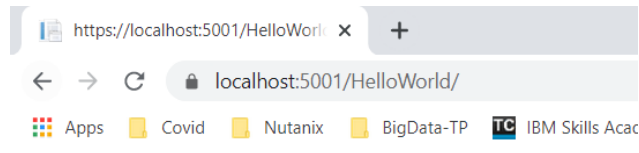
```
1 using Microsoft.AspNetCore.Mvc;
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Threading.Tasks;
6
7 namespace MvcMovie.Controllers
8 {
9     public class HelloWorldController : Controller
10    {
11        // GET: /HelloWorld/
12        public string Index()
13        {
14            return "c'est mon action par défaut";
15        }
16        // GET: /HelloWorld/Welcome/
17        public string Welcome()
18        {
19            return "c'est la méthode de l'action Welcome";
20        }
21    }
22 }
23
```

Chaque méthode `public` d'un contrôleur peut être appelée en tant que point de terminaison HTTP. Dans l'exemple ci-dessus, les deux méthodes retournent une chaîne de caractères. Notez les commentaires qui précèdent chaque méthode.

Un point de terminaison HTTP est une URL qui peut être ciblée dans l'application web, comme `https://localhost:5001/HelloWorld`, et qui combine le protocole utilisé : `HTTPS`, l'emplacement réseau du serveur web (avec le port TCP) : `localhost:5001` et l'URI cible `HelloWorld`.

Le premier commentaire indique qu'il s'agit d'une méthode HTTP GET qui est appelée en ajoutant `/HelloWorld/` à l'URL de base. Le deuxième commentaire indique une méthode HTTP GET qui est appelée en ajoutant `/HelloWorld/Welcome/` à l'URL. Plus loin dans ce lab, le moteur de génération de modèles automatique est utilisé pour générer des méthodes `HTTP POST` qui mettent à jour des données.

Exécutez l'application en mode de sans débogage et ajoutez « HelloWorld » au chemin dans la barre d'adresse. La méthode `Index` retourne une chaîne.



c'est mon action par défaut

MVC appelle les classes du contrôleur (et les méthodes d'action au sein de celles-ci) en fonction de l'URL entrante. La logique de routage d'URL par défaut utilisée par le modèle MVC utilise un format comme celui-ci pour déterminer le code à appeler :

```
/[Controller]/[ActionName]/[Parameters]
```

Le format de routage est défini dans la méthode `Configure` du fichier *Startup.cs*.

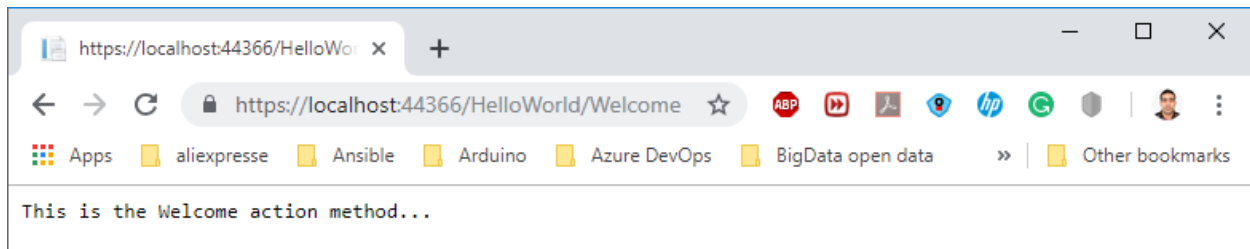
```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Quand vous naviguez jusqu'à l'application et que vous ne fournissez aucun segment d'URL, sa valeur par défaut est le contrôleur « Home » et la méthode « Index » spécifiée dans la ligne du modèle mise en surbrillance ci-dessus.

Le premier segment d'URL détermine la classe du contrôleur à exécuter. Ainsi, `localhost:xxxx/HelloWorld` est mappé à la classe `HelloWorldController`. La seconde partie du segment d'URL détermine la méthode d'action sur la classe. Ainsi, `localhost:xxxx/HelloWorld/Index` provoque l'exécution de la méthode `Index` de la classe `HelloWorldController`. Notez que vous n'avez eu qu'à accéder à `localhost:xxxx/HelloWorld` pour que la méthode `Index` soit appelée par défaut. La raison en est que `Index` est la méthode par défaut qui est appelée sur un contrôleur si un nom de méthode n'est pas explicitement spécifié. La troisième partie du segment d'URL (`id`) concerne les données de routage. Les données de routage sont expliquées plus loin dans le tutoriel.

Accédez à `https://localhost:xxxx/HelloWorld/Welcome`. La méthode `Welcome` s'exécute et retourne la chaîne `c'est la méthode de l'action Welcome`. Pour cette URL,

le contrôleur est `HelloWorld`, et `Welcome` est la méthode d'action. Vous n'avez pas encore utilisé la partie `[Parameters]` de l'URL.



Modifiez le code pour passer des informations sur les paramètres de l'URL au contrôleur. Par exemple, `/HelloWorld/Welcome?name=kabbaj&numtimes=3`. Modifiez la méthode `Welcome` en y incluant les deux paramètres, comme indiqué dans le code suivant.

```
17 // GET: /HelloWorld/Welcome/
18 0 références
19 public string Welcome(string name, int numtimes = 1)
20 {
21     return HtmlEncoder.Default.Encode($"Salut {name}, Numtimes est : {numtimes}");
22 }
```

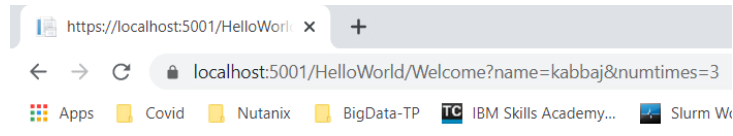
Le code précédent :

- Utilise la fonctionnalité de paramètre facultatif de C# pour indiquer que le paramètre `numTimes` a 1 comme valeur par défaut si aucune valeur n'est passée pour ce paramètre.
- Utilise `HtmlEncoder.Default.Encode` pour protéger l'application des entrées malveillantes (à savoir JavaScript).
- Utilise des chaînes interpolées dans `$"Salut {name}, Numtimes est : {numtimes}"`.

Exécutez l'application et accédez à :

```
https://localhost:xxxx/HelloWorld/Welcome?name=kabbaj&numtimes=4
```

(Remplacez xxxx par votre numéro de port.) Vous pouvez essayer différentes valeurs pour `name` et `numtimes` dans l'URL. Le système de liaison de données du modèle MVC mappe automatiquement les paramètres nommés provenant de la chaîne de requête dans la barre d'adresse aux paramètres de votre méthode.



Salut kabbaj, Numtimes est : 3

Dans l'image ci-dessus, le segment d'URL (`Parameters`) n'est pas utilisé, les paramètres `name` et `numTimes` sont passés en tant que chaînes de requête.

Remplacez la méthode `Welcome` par le code suivant :

```
18 // GET: /HelloWorld/Welcome/
19 0 références
20 public string Welcome(string name, int ID = 1)
21 {
22     return HtmlEncoder.Default.Encode($"Salut {name}, Numtimes est : {ID}");
23 }
```

Exécutez l'application et entrez l'URL suivante :
`https://localhost:xxx/HelloWorld/Welcome/3?name=kabbaj`



Salut kabbaj, ID est : 3

Cette fois, le troisième segment de l'URL correspondait au paramètre de routage `id`. La méthode `Welcome` contient un paramètre `id` qui correspondait au modèle d'URL de la méthode `MapRoute`. Le `?` de fin (dans `id?`) indique que le paramètre `id` est facultatif.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Dans ces exemples, le contrôleur a fait la partie « VC » du modèle MVC, autrement dit, la vue et le contrôleur fonctionnent. Le contrôleur retourne directement du HTML. En règle

générale, vous ne souhaitez pas que les contrôleurs retournent directement du HTML, car le codage et la maintenance deviennent dans ce cas très laborieux. Au lieu de cela, vous utilisez généralement un fichier de modèle de vue Razor distinct pour faciliter la génération de la réponse HTML.

Ajouter une vue à une application ASP.NET Core MVC

Dans cette section, vous modifiez la classe `HelloWorldController` de façon à utiliser les fichiers de vue Razor pour encapsuler proprement le processus de génération des réponses HTML à un client.

Vous créez un fichier de modèle de vue à l'aide de Razor. Les modèles de vue Razor ont l'extension de fichier `.cshtml`. Ils offrent un moyen élégant pour créer une sortie HTML avec C#.

Actuellement, la méthode `Index` retourne une chaîne avec un message qui est codé en dur dans la classe du contrôleur. Dans la classe `HelloWorldController`, remplacez la méthode `Index` par le code suivant :

```
public IActionResult Index()
{
    return View();
}
```

Le code précédent appelle la méthode `View` du contrôleur. Il utilise un modèle de vue pour générer une réponse HTML. Les méthodes du contrôleur (également appelées *méthodes d'action*), comme la méthode `Index` ci-dessus, retournent généralement un `IActionResult` (ou une classe dérivée de `ActionResult`), et non un type comme `string`.

Ajouter une vue

- Cliquez avec le bouton droit sur le dossier *Views*, cliquez sur **Ajouter > Nouveau dossier**, puis nommez le dossier *HelloWorld*.
- Cliquez avec le bouton droit sur le dossier *Vues/HelloWorld*, puis cliquez sur **Ajouter > Nouvel élément**.
- Dans la boîte de dialogue **Ajouter un nouvel élément généré automatiquement**
 - Sélectionnez **Vue Razor**
 - Nommez la valeur de la zone **Nom de la vue**, *Index*.
 - Sélectionnez **Ajouter**

Ajouter Vue Razor

Nom de la vue

Modèle

Empty (sans modèle)

Classe de modèle

Classe de contexte de données

Options

☐ Créer en tant que vue partielle

☐ Bibliothèques de scripts de référence

☒ Utiliser une page de disposition

(Laissez vide s'il est défini dans un fichier Razor _viewstart)

Ajouter

Annuler

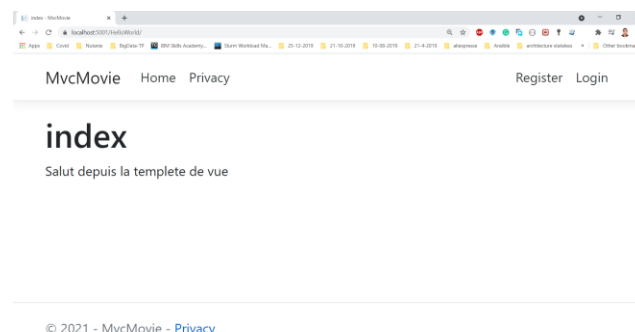
Remplacez le contenu du fichier vue Razor *Views/HelloWorld/Index.cshtml* par le code suivant :

```

1
2      @{
3          ViewData["Title"] = "index";
4      }
5
6      <h1>index</h1>
7
8      <p>Salut depuis la templete de vue</p>

```

Accédez à `https://localhost:xxxx/HelloWorld`. La méthode `Index` dans `HelloWorldController` n'a pas accompli beaucoup d'actions. Elle a exécuté l'instruction `return View();`, laquelle spécifiait que la méthode doit utiliser un fichier de modèle de vue pour restituer une réponse au navigateur. Étant donné que vous n'avez pas explicitement spécifié le nom du fichier de modèle de vue, MVC a utilisé par défaut le fichier vue *Index.cshtml* présent dans le dossier */Views/HelloWorld*. L'image ci-dessous montre la chaîne « Hello from our View Template! » codée en dur dans la vue.



Changer les vues et les pages de disposition

Sélectionnez les liens du menu (**MvcMovie**, **Home** et **Privacy**). Chaque page affiche la même disposition de menu. La disposition du menu est implémentée dans le fichier `Views/Shared/_Layout.cshtml`. Ouvrez le fichier `Views/Shared/_Layout.cshtml`.

Les modèles de disposition vous permettent de spécifier la disposition du conteneur HTML de votre site dans un emplacement unique, puis de l'appliquer sur plusieurs pages de votre site. Recherchez la ligne `@RenderBody()`. `RenderBody` est un espace réservé dans lequel toutes les pages spécifiques aux vues que vous créez s'affichent, *encapsulées* dans la page de disposition. Par exemple, si vous sélectionnez le lien **Privacy**, la vue `Views/Home/Privacy.cshtml` est restituée dans la méthode `RenderBody`.

Changer le lien de titre, de pied de page et de menu dans le fichier de disposition

- Dans les éléments de titre et de pied de page, remplacez `MvcMovie` par `Movie App`.
- Modifier l'élément d'ancrage `MvcMovie` par `Movie App`.

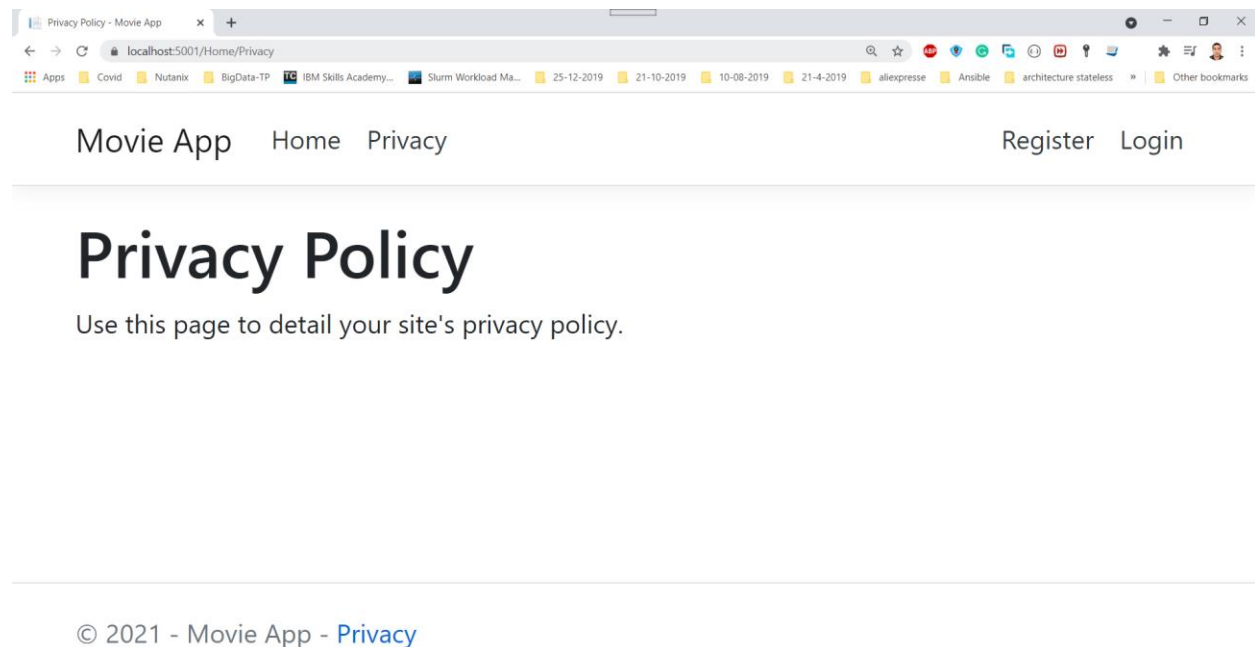
Les lignes en surbrillance en vert suivant illustre les changements :

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="utf-8" />
5 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6 <title>@ViewData["Title"] - Movie App</title>
7 <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.min.css" />
8 <link rel="stylesheet" href="/css/site.css" />
9 </head>
10 <body>
11 <div>
12 <div class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
13 <div class="container">
14 <a class="navbar-brand" asp-area="" asp-controller="Movies" asp-action="Index">Movie App</a>
15 <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-collapse" aria-controls="navbarSupportedContent"
16 <div class="navbar-collapse">
17 <span class="navbar-toggler-icon"></span>
18 <div>
19 <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
20 <partial name="_LoginPartial" />
21 <ul class="navbar-nav flex-grow-1">
22 <li class="nav-item">
23 <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
24 </li>
25 <li class="nav-item">
26 <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
27 </li>
28 </ul>
29 </div>
30 </div>
31 </div>
32 </div>
33 <div class="container">
34 <div class="main" class="pb-3">
35 @RenderBody()
36 </div>
37 </div>
38 <div class="border-top footer text-muted">
39 <div class="container">
40 &copy; 2021 - Movie App - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
41 </div>
42 </div>
43 </body>
44 </html>
```

Dans le balisage précédent, l'attribut Tag Helper Ancre `asp-area` a été omis, car cette application n'utilise pas de zones.

Remarque : Le contrôleur `Movies` n'a pas encore été implémenté. À ce stade, le lien `Movie App` ne fonctionne pas.

Enregistrer vos modifications et sélectionnez le lien **Privacy**. Notez comment le titre sur l'onglet du navigateur affiche **Privacy Policy - Movie App** au lieu de **Privacy Policy - Mvc Movie** :



Sélectionnez le lien **Home** et notez que le titre et le texte d'ancrage affichent également **Movie App**. Nous avons pu effectuer ce changement une fois dans le modèle de disposition et avoir le nouveau texte de lien et le nouveau titre reflétés sur toutes les pages du site.

Examinez le fichier `Views/_ViewStart.cshtml`

```
@{
    Layout = "_Layout";
}
```

Le fichier `Views/_ViewStart.cshtml` introduit le fichier `Views/Shared/_Layout.cshtml` dans chaque vue. La propriété `Layout` peut être utilisée pour définir un mode de disposition différent ou lui affecter la valeur `null`. Aucun fichier de disposition n'est donc utilisé.

Modifiez le titre et l'élément `<h1>` du fichier de vue *Views/HelloWorld/Index.cshtml* :

```
_Layout.cshtml | index.cshtml* | HelloWorldController.cs
1
2   @{
3       ViewData["Title"] = "Movie List";
4   }
5
6   <h1>Ma Liste des Films</h1>
7
8   <p>Salut depuis la templete de vue</p>
```

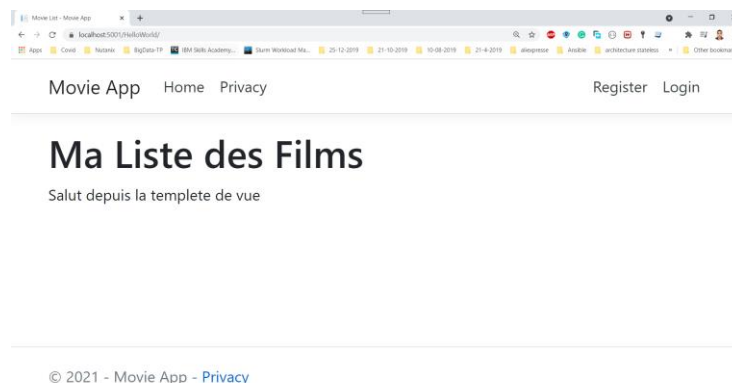
Le titre et l'élément `<h1>` sont légèrement différents afin que vous puissiez voir quel morceau du code modifie l'affichage.

Dans le code ci-dessus, `ViewData["Title"] = "Movie List";` définit la propriété `Title` du dictionnaire `ViewData` sur « Movie List ». La propriété `Title` est utilisée dans l'élément HTML `<title>` dans la page de disposition :

```
<title>@ViewData["Title"] - Movie App</title>
```

Enregistrez la modification et accédez à `https://localhost:xxxx/HelloWorld`. Notez que le titre du navigateur, l'en-tête principal et les en-têtes secondaires ont changé. (Si vous ne voyez pas les changements dans le navigateur, vous voyez peut-être le contenu mis en cache. Appuyez sur Ctrl+F5 dans votre navigateur pour forcer le chargement de la réponse du serveur.) Le titre du navigateur est créé avec la valeur `ViewData["Title"]` que nous avons définie dans le modèle de vue *Index.cshtml* et la chaîne « - Movie App » ajoutée dans le fichier de disposition.

Notez également comment le contenu du modèle de vue *Index.cshtml* a été fusionné avec le modèle de vue *Views/Shared/_Layout.cshtml* et qu'une seule réponse HTML a été envoyée au navigateur. Les modèles de disposition permettent d'apporter facilement des modifications qui s'appliquent à toutes les pages de votre application.



Nos quelques « données » (dans le cas présent, le message « Salut depuis la templete de vue ») sont toutefois codées en dur. L'application MVC a une vue (« V ») et vous avez un contrôleur (« C »), mais pas encore de modèle (« M »).

Passage de données du contrôleur vers la vue

Les actions du contrôleur sont appelées en réponse à une demande d'URL entrante. Une classe de contrôleur est l'endroit où le code est écrit et qui gère les demandes du navigateur entrantes. Le contrôleur récupère les données d'une source de données et détermine le type de réponse à envoyer au navigateur. Il est possible d'utiliser des modèles de vue à partir d'un contrôleur pour générer et mettre en forme une réponse HTML au navigateur.

Les contrôleurs sont chargés de fournir les données nécessaires pour qu'un **modèle de vue** restitue une réponse. Une meilleure pratique : N'oubliez pas que les modèles de vue ne doivent **pas** exécuter de logique métier ou interagir directement avec une base de données. Au lieu de cela, un modèle de vue doit fonctionner uniquement avec les données que le contrôleur lui fournit. Préserver cette « séparation des intérêts » permet de maintenir le code clair, testable et facile à gérer.

Actuellement, le `Welcome` méthode présente dans la classe `HelloWorldController` prend un paramètre `name` et un paramètre `ID`, puis sort les valeurs directement dans le navigateur. Au lieu que le contrôleur restitue cette réponse sous forme de chaîne, changez le contrôleur pour qu'il utilise un modèle de vue à la place. Comme le modèle de vue génère une réponse dynamique, les bits de données appropriés doivent être passés du contrôleur à la vue pour générer la réponse. Pour cela, le contrôleur doit placer les données dynamiques (paramètres) dont le modèle de vue a besoin dans un dictionnaire `ViewData` auquel le modèle de vue peut ensuite accéder.

Dans `HelloWorldController.cs`, modifiez la méthode `Welcome` pour ajouter une valeur `Message` et `NumTimes` au dictionnaire `ViewData`. Le dictionnaire `ViewData` est un objet dynamique, ce qui signifie que n'importe quel type peut être utilisé, l'objet `ViewData` ne possède aucune propriété définie tant que vous ne placez pas d'élément dedans. Le système de liaison de données MVC mappe automatiquement les paramètres nommés (`name` et `numTimes`) provenant de la chaîne de requête dans la barre d'adresse aux paramètres de votre méthode. Le fichier `HelloWorldController.cs` complet ressemble à ceci :


```

10 public class HelloWorldController : Controller
11 {
12     // GET: /HelloWorld/
13     // 0 références
14     public IActionResult Index()
15     {
16         return View();
17     }
18     // GET: /HelloWorld/Welcome/
19     // 0 références
20     public IActionResult Welcome(string name, int numTimes = 1)
21     {
22         ViewData["Message"] = "Salut " + name;
23         ViewData["NumTimes"] = numTimes;
24         return View();
25     }
26 }

```

L'objet dictionnaire `ViewData` contient des données qui seront passées à la vue.

Créez un modèle de vue Welcome nommé *Views/HelloWorld/Welcome.cshtml*.

Vous allez créer une boucle dans le modèle de vue *Welcome.cshtml* qui affiche « Hello » `NumTimes`. Remplacez le contenu de *Views/HelloWorld/Welcome.cshtml* avec le code suivant :

```

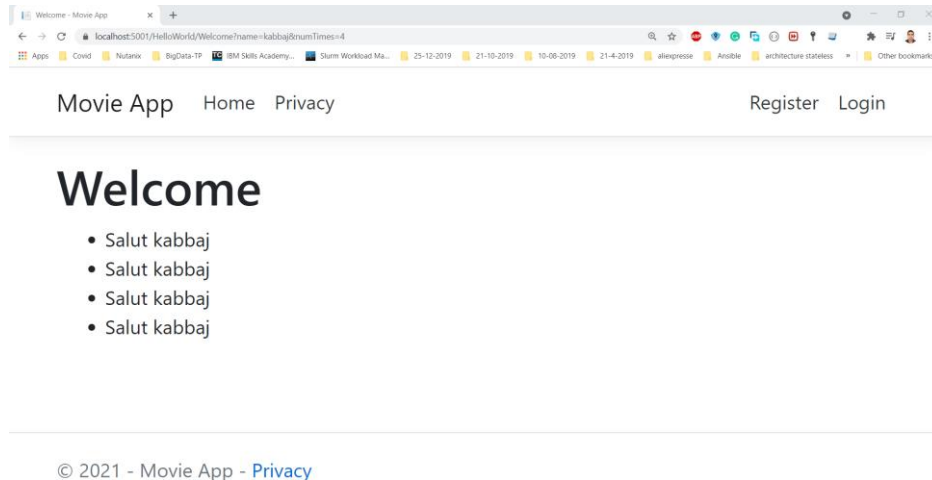
Welcome.cshtml | index.cshtml | HelloWorldController.cs | MvcMov
1
2 @{
3     ViewData["Title"] = "Welcome";
4 }
5
6 <h1>Welcome</h1>
7
8 <ul>
9     @for(int i=0; i<(int)ViewData["numTimes"]; i++)
10     {
11         <li>@ViewData["Message"]</li>
12     }
13 </ul>

```

Enregistrez vos modifications et accédez à l'URL suivante :

```
https://localhost:xxxx/HelloWorld/Welcome?name=kabbaj&numtimes=4
```

Les données sont extraites de l'URL et passées au contrôleur à l'aide du classeur de modèles MVC. Le contrôleur regroupe les données dans un dictionnaire `ViewData` et passe cet objet à la vue. La vue restitue ensuite les données au format HTML dans le navigateur.



Dans l'exemple ci-dessus, le dictionnaire `ViewData` a été utilisé pour passer des données du contrôleur à une vue. Plus loin dans ce lab, un modèle de vue est utilisé pour passer les données d'un contrôleur à une vue. L'approche basée sur le modèle de vue pour passer des données est généralement préférée à l'approche basée sur le dictionnaire `ViewData`.

Ajouter un modèle dans une application ASP.NET Core MVC

Dans cette section, vous allez ajouter des classes pour la gestion des films dans une base de données. Ces classes constituent la partie « **Modèle** » de l'application **MVC**.

Vous utilisez ces classes avec Entity Framework Core (EF Core) pour travailler avec une base de données. EF Core est un framework de mappage relationnel d'objets qui simplifie le code d'accès aux données à écrire.

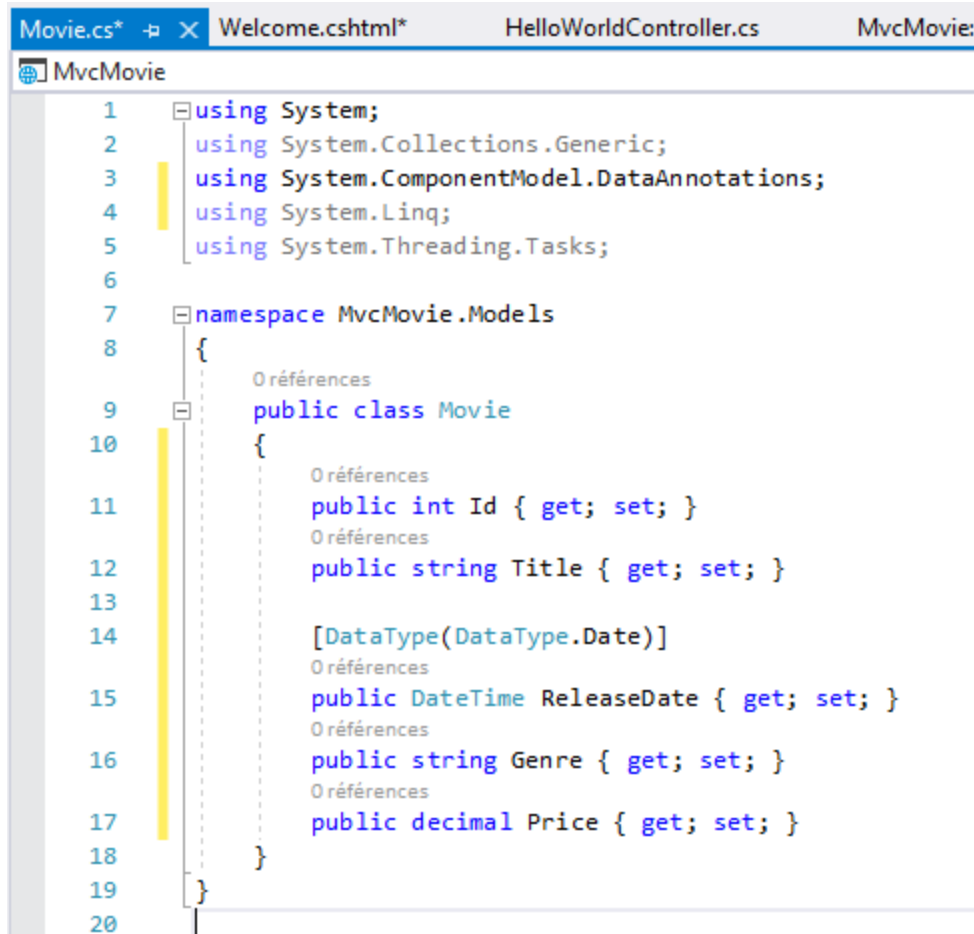
Les classes de modèle que vous créez portent le nom de classes OCT (« **O**bjets **CLR** **T**raditionnel »), car elles n'ont pas de dépendances envers EF Core. Elles définissent simplement les propriétés des données stockées dans la base de données.

Dans ce lab, vous écrivez d'abord les classes du modèle, puis EF Core crée la base de données. Une autre approche consiste à générer les classes de modèle à partir d'une base de données existante.

Ajouter une classe de modèle de données

Cliquez avec le bouton droit sur le dossier *Models* > **Ajouter** > **Classe**. Nommez la classe **Movie**.

Ajoutez les propriétés suivantes à la classe Movie :



```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel.DataAnnotations;
4 using System.Linq;
5 using System.Threading.Tasks;
6
7 namespace MvcMovie.Models
8 {
9     public class Movie
10    {
11        public int Id { get; set; }
12        public string Title { get; set; }
13
14        [DataType(DataType.Date)]
15        public DateTime ReleaseDate { get; set; }
16        public string Genre { get; set; }
17        public decimal Price { get; set; }
18    }
19 }
20
```

La classe Movie contient :

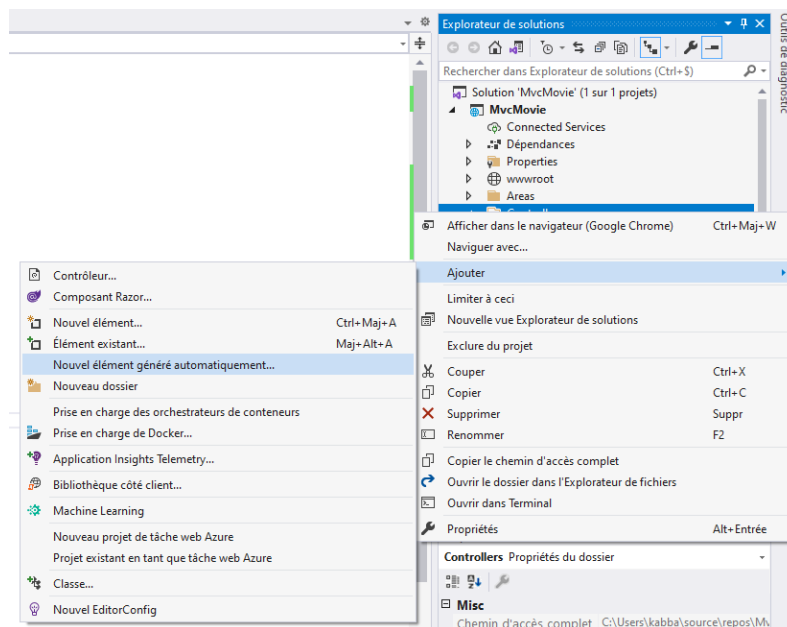
- Champ ID requis par la base de données pour la clé primaire.
- [DataType (DataType. date)]: l'attribut DataType spécifie le type des données (date). Avec cet attribut :
 - L'utilisateur n'est pas obligé d'entrer des informations de temps dans le champ de date.
 - Seule la date est affichée, pas les informations de temps.

DataAnnotations sont abordées dans ce lab ultérieur.

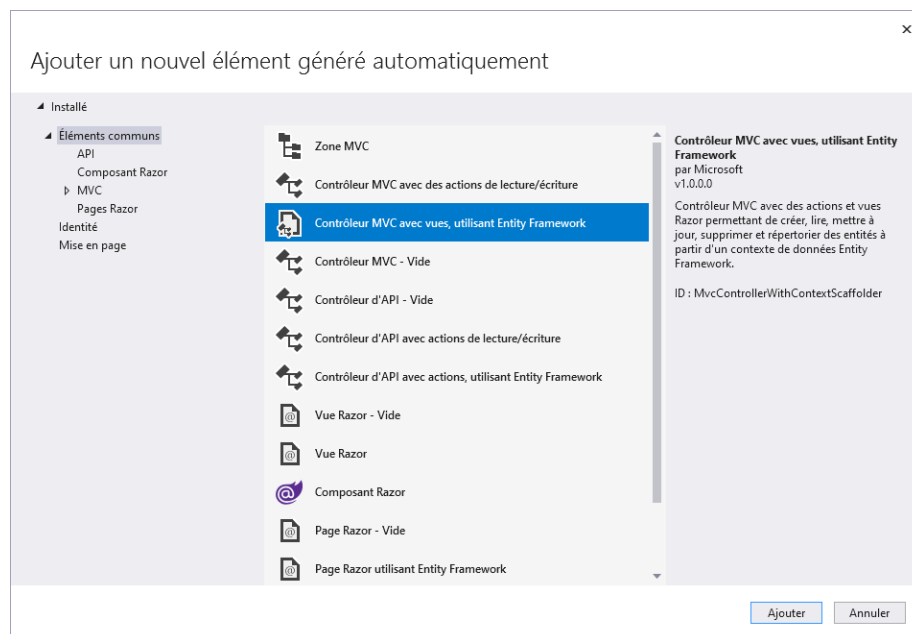
Générer automatiquement le modèle de film

Dans cette section, le modèle de film est généré automatiquement. Autrement dit, l'outil de génération de modèles automatique génère des pages pour les opérations de création, de lecture, de mise à jour et de suppression (CRUD) pour le modèle de film.

Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le dossier *Contrôleurs*, puis choisissez **Ajouter > Nouvel élément généré automatiquement**.



Dans la boîte de dialogue **Ajouter un modèle automatique**, sélectionnez **Contrôleur MVC avec vues, utilisant Entity Framework > Ajouter**.



Renseignez la boîte de dialogue **Ajouter un contrôleur** :

- **Classe du modèle** : *Movie* (*MvcMovie.Models*)

- **Classe de contexte de données** : sélectionnez l'icône + et ajoutez le **MvcMovie.Models.MvcMovieContext** par défaut

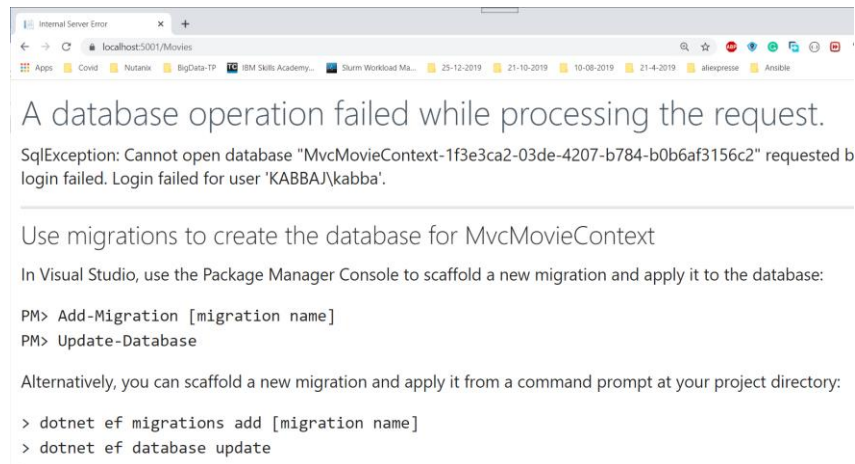
- **Affichages** : conservez la valeur par défaut de chaque option activée
- **Nom du contrôleur** : conservez la valeur par défaut *MoviesController*
- Sélectionnez **Ajouter**

Visual Studio crée :

- Une classe de contexte de base de données Entity Framework Core (*Data/MvcMovieContext.cs*)
- Un contrôleur de films (*Controllers/MoviesController.cs*)
- Des fichiers de vues Razor pour les pages Create, Delete, Details, Edit et Index (*Views/Movies/*.cshtml*)

La création automatique du contexte de base de données et de méthodes d'action et de vues CRUD (créer, lire, mettre à jour et supprimer) porte le nom de *génération de modèles automatique*.

Si vous exécutez l'application et que vous cliquez sur le lien **Mvc Movie**, vous recevez une erreur semblable à la suivante :

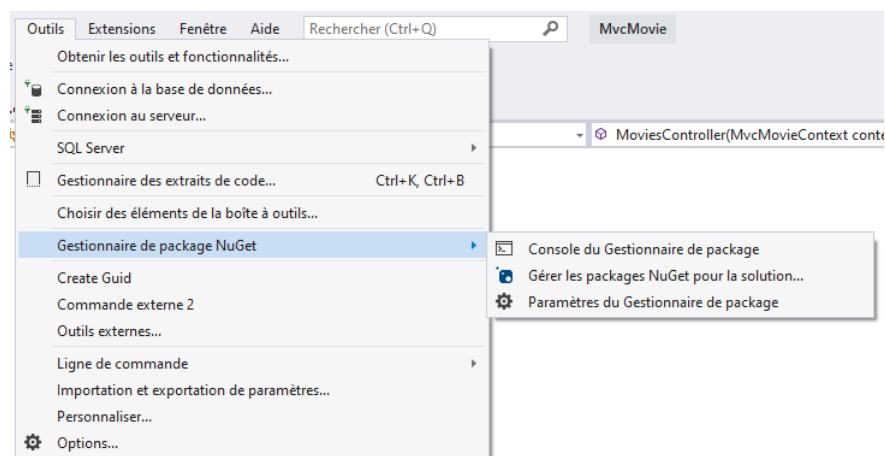


Vous devez créer la base de données, et vous utilisez pour cela la fonctionnalité Migrations d'EF Core. Les migrations permettent de créer une base de données qui correspond à votre modèle de données, et de mettre à jour le schéma de base de données quand votre modèle de données change.

Migration initiale

Dans cette section, vous devez effectuer les tâches suivantes :

- Ajouter une migration initiale
 - Mettez à jour la base de données avec la migration initiale.
1. Dans le menu **Outils**, sélectionnez **Gestionnaire de package NuGet > Console du gestionnaire de package (PMC)**.



2. Dans la console du Gestionnaire de package, entrez les commandes suivantes :

```
PM> Add-Migration Initial -Context MvcMovieContext
```

```
PM> Update-Database -Context MvcMovieContext
```

La commande `Add-Migration` génère le code nécessaire à la création du schéma de base de données initial.

Le schéma de base de données est basé sur le modèle spécifié dans la classe `MvcMovieContext` (dans `Data/MvcMovieContext.cs`). L'argument `Initial` est le nom de la migration. Vous pouvez utiliser n'importe quel nom, mais par convention, un nom décrivant la migration est utilisé.

La commande `Update-Database` exécute la méthode `Up` dans le fichier `Migrations/{horodatage}_InitialCreate.cs`, ce qui entraîne la création de la base de données. Les commandes précédentes génèrent l'avertissement suivant :

```
PM> Add-Migration Initial
Build started...
Build succeeded.
More than one DbContext was found. Specify which one to use. Use the '-Context' parameter for PowerShell commands and the '--context' parameter for dotnet commands.
PM> Add-Migration Initial -Context MvcMovieContext
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
  No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision that can accommodate all the values using 'HasColumnType()'.
  To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
More than one DbContext was found. Specify which one to use. Use the '-Context' parameter for PowerShell commands and the '--context' parameter for dotnet commands.
PM> Update-Database -Context MvcMovieContext
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
  No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision that can accommodate all the values using 'HasColumnType()'.
Done.
PM>
```

Vous pouvez ignorer cet avertissement. Il sera résolu dans un prochain tutoriel.

Examiner le contexte inscrit avec l'injection de dépendances

ASP.NET Core comprend l'injection de dépendances (DI). Des services (tels que le contexte de base de données EF Core) sont inscrits avec l'injection de dépendances au démarrage de l'application. Ces services sont affectés aux composants qui les nécessitent (par exemple les Pages Razor) par le biais de paramètres de constructeur. Le code du constructeur qui obtient une instance de contexte de base de données est indiqué plus loin dans le tutoriel.

L'outil de génération de modèles automatique a créé automatiquement un contexte de base de données et l'a inscrit dans le conteneur d'injection de dépendances.

Examinez la méthode `Startup.ConfigureServices` suivante. La ligne en surbrillance a été ajoutée par l'outil de génération de modèles automatique :

```

28 public void ConfigureServices(IServiceCollection services)
29 {
30     services.AddDbContext<ApplicationDbContext>(options =>
31         options.UseSqlServer(
32             Configuration.GetConnectionString("DefaultConnection")));
33     services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
34         .AddEntityFrameworkStores<ApplicationDbContext>();
35     services.AddControllersWithViews();
36     services.AddRazorPages();
37
38     services.AddDbContext<MvcMovieContext>(options =>
39         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
40 }

```

`MvcMovieContext` coordonne les fonctionnalités d'EF Core (Create, Read, Update, Delete, etc.) pour le modèle `Movie`. Le contexte de données (`MvcMovieContext`) est dérivé de `Microsoft.EntityFrameworkCore.DbContext`. Il spécifie les entités qui sont incluses dans le modèle de données :

```

8 namespace MvcMovie.Data
9 {
10     7 références
11     public class MvcMovieContext : DbContext
12     {
13         0 références
14         public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
15             : base(options)
16         {
17         }
18         7 références
19         public DbSet<MvcMovie.Models.Movie> Movie { get; set; }
20     }

```

Le code précédent crée une propriété `DbSet<Movie>` pour le jeu d'entités. Dans la terminologie Entity Framework, un jeu d'entités correspond généralement à une table de base de données. Une entité correspond à une ligne dans la table.

Le nom de la chaîne de connexion est transmis au contexte en appelant une méthode sur un objet `DbContextOptions`. Pour le développement local, le système de configuration ASP.NET Core lit la chaîne de connexion à partir du fichier `appsettings.json`.

Tester l'application

- Exécutez l'application et ajoutez `/Movies` à l'URL dans le navigateur (`http://localhost:port/movies`).

Si vous obtenez une exception de base de données similaire à ce qui suit :


```
SqlException: Cannot open database "MvcMovieContext-GUID" requested by the login. The login failed for user 'User-name'.
```

Vous avez manqué l'étape des migrations.

- Testez le lien **Créer**.
- Testez les liens **Modifier**, **Détails** et **Supprimer**.

Examiner la classe `Startup` :

```
28 public void ConfigureServices(IServiceCollection services)
29 {
30     services.AddDbContext<ApplicationDbContext>(options =>
31         options.UseSqlServer(
32             Configuration.GetConnectionString("DefaultConnection")));
33     services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
34         .AddEntityFrameworkStores<ApplicationDbContext>();
35     services.AddControllersWithViews();
36     services.AddRazorPages();
37
38     services.AddDbContext<MvcMovieContext>(options =>
39         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
40 }
```

Le code précédent mis en cadre montre le contexte de la base de données des films qui est ajouté au conteneur d'injection de dépendance :

- `services.AddDbContext<MvcMovieContext>(options =>` spécifie la base de données à utiliser et la chaîne de connexion.
- `=>` est un opérateur lambda

Ouvrez le fichier `Controllers/MoviesController.cs` et examinez le constructeur :

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
}
```

Le constructeur utilise une injection de dépendance pour injecter le contexte de base de données (`MvcMovieContext`) dans le contrôleur. Le contexte de base de données est utilisé dans chacune des méthodes la CRUD du contrôleur.

Modèles fortement typés et mot clé @model

Plus tôt dans ce lab, vous avez vu comment un contrôleur peut passer des données ou des objets à une vue en utilisant le dictionnaire `ViewData`. Le dictionnaire `ViewData` est un objet dynamique qui fournit un moyen pratique d'effectuer une liaison tardive pour passer des informations à une vue.

Le modèle MVC fournit également la possibilité de passer des objets de modèle fortement typés à une vue. Cette approche fortement typée permet une meilleure vérification de votre code au moment de la compilation. Le mécanisme de génération de modèles automatique a utilisé cette approche (c'est-à-dire passer un modèle fortement typé) avec la classe `MoviesController` et les vues quand il a créé les méthodes et les vues.

Examinez la méthode `Details` générée dans le fichier `Controllers/MoviesController.cs` :

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

Le paramètre `id` est généralement passé en tant que données de routage. Par exemple, `https://localhost:5001/movies/details/1` définit :

- Le contrôleur sur le contrôleur `movies` (le premier segment de l'URL).
- L'action sur `details` (le deuxième segment de l'URL).
- L'ID sur 1 (le dernier segment de l'URL).

Vous pouvez aussi passer `id` avec une requête de chaîne, comme suit :

`https://localhost:5001/movies/details?id=1`

Le paramètre `id` est défini comme type nullable (`int?`) au cas où la valeur d'ID n'est pas fournie.

Une expression lambda est passée à `FirstOrDefaultAsync` pour sélectionner les entités de film qui correspondent aux données de routage ou à la valeur de la chaîne de requête.

```
var movie = await _context.Movie
    .FirstOrDefaultAsync(m => m.Id == id);
```

Si un film est trouvé, une instance du modèle `Movie` est passée à la vue `Details` :

```
return View(movie);
```

Examinez le contenu du fichier *Views/Movies/Details.cshtml*:

```

@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>

```

En incluant une instruction `@model` en haut du fichier de la vue, vous pouvez spécifier le type d'objet attendu par la vue. Quand vous avez créé le contrôleur pour les films, l'instruction `@model` suivante a été incluse automatiquement en haut du fichier *Details.cshtml* :

```
@model MvcMovie.Models.Movie
```

Cette directive `@model` vous permet d'accéder au film que le contrôleur a passé à la vue en utilisant un objet `Model` qui est fortement typé. Par exemple, dans la vue *Details.cshtml*, le code passe chaque champ du film aux Helpers HTML `DisplayNameFor` et `DisplayFor` avec l'objet `Model` fortement typé. Les méthodes et les vues `Create` et `Edit` passent aussi un objet du modèle `Movie`.

Examinez la vue *Index.cshtml* et la méthode `Index` dans le contrôleur *Movies*. Notez comment le code crée un objet `List` quand il appelle la méthode `View`. Le code passe cette liste `Movies` de la méthode d'action `Index` à la vue :

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

Quand vous avez créé le contrôleur pour les films, la génération de modèles automatique a inclus automatiquement l'instruction `@model` suivante en haut du fichier *Index.cshtml* :

```
@model IEnumerable<MvcMovie.Models.Movie>
```

La directive `@model` vous permet d'accéder à la liste des films que le contrôleur a passé à la vue en utilisant un objet `Model` qui est fortement typé. Par exemple, dans la vue *Index.cshtml*, le code boucle dans les films avec une instruction `foreach` sur l'objet `Model` fortement typé :

```

@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                    <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

Comme l'objet `Model` est fortement typé (en tant qu'objet `IEnumerable<Movie>`), chaque élément de la boucle est typé en tant que `Movie`. Entre autres avantages, cela signifie que votre code est vérifié au moment de la compilation :

Utiliser SQL dans ASP.NET Core

L'objet `MvcMovieContext` gère la tâche de connexion à la base de données et de mappage d'objets `Movie` à des enregistrements de la base de données. Le contexte de base de données est inscrit auprès du conteneur Injection de dépendances dans la méthode `ConfigureServices` du fichier *Startup.cs* :

```
28 public void ConfigureServices(IServiceCollection services)
29 {
30     services.AddDbContext<ApplicationDbContext>(options =>
31         options.UseSqlServer(
32             Configuration.GetConnectionString("DefaultConnection")));
33     services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
34         .AddEntityFrameworkStores<ApplicationDbContext>();
35     services.AddControllersWithViews();
36     services.AddRazorPages();
37
38     services.AddDbContext<MvcMovieContext>(options =>
39         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
40 }
```

Le système de configuration d'ASP.NET Core lit `ConnectionString`. Pour un développement local, il obtient la chaîne de connexion à partir du fichier *appsettings.json* :

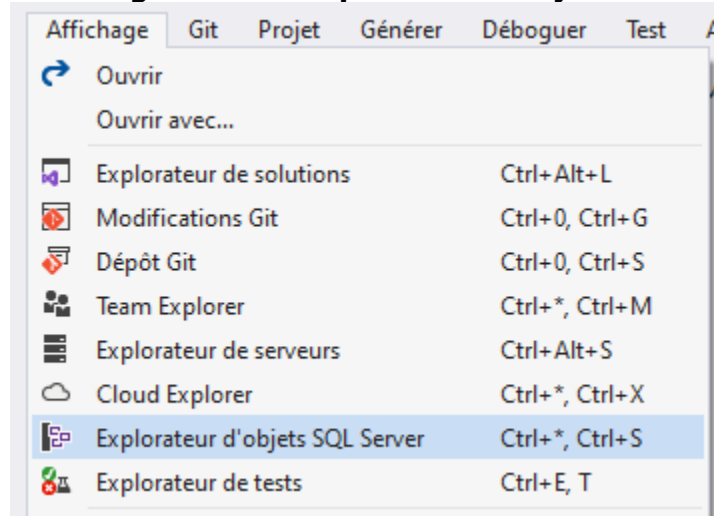
```
"ConnectionStrings": {
  "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-2;Truste
}
```

Quand vous déployez l'application sur un serveur de test ou de production, vous pouvez utiliser une variable d'environnement ou une autre approche pour définir un serveur SQL Server réel comme chaîne de connexion.

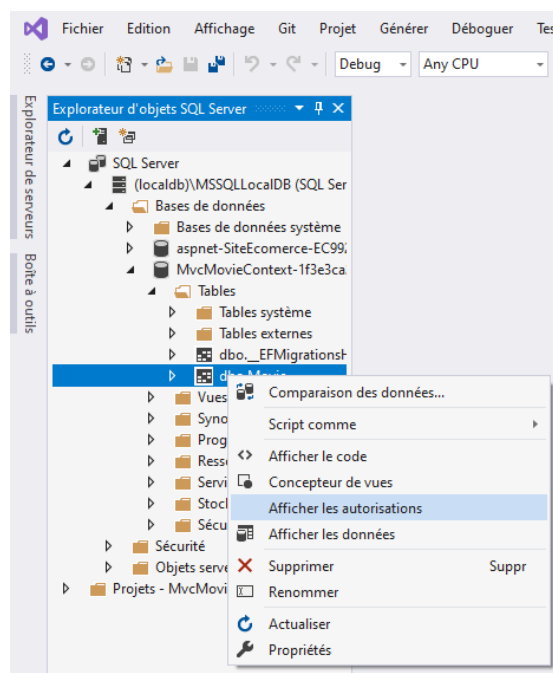
SQL Server Express LocalDB

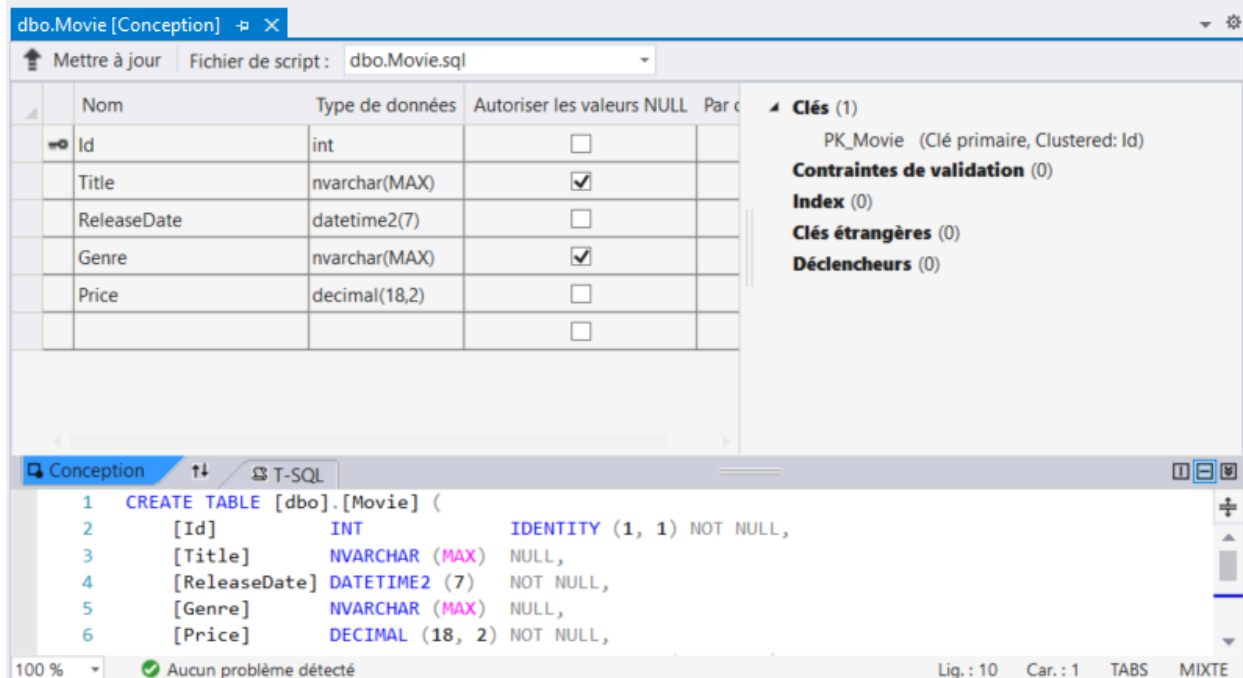
LocalDB est une version allégée du moteur de base de données SQL Server Express qui est ciblée pour le développement de programmes. LocalDB démarre à la demande et s'exécute en mode utilisateur, ce qui n'implique aucune configuration complexe. Par défaut, la base de données LocalDB crée des fichiers *.mdf* dans le répertoire *C:/Users/{utilisateur}*.

- Dans le menu **Affichage**, ouvrez l'**Explorateur d'objets SQL Server** (SSOX).



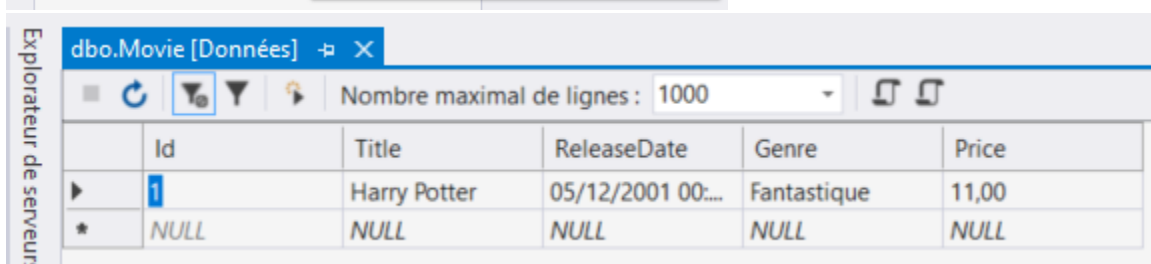
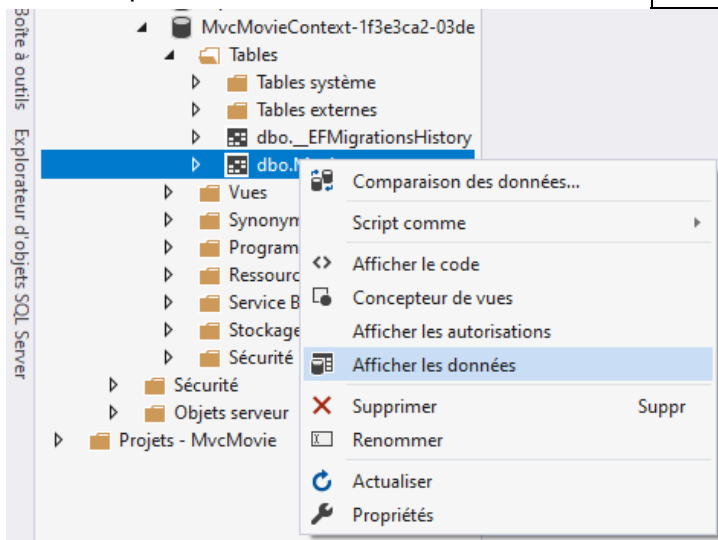
- Cliquez avec le bouton droit sur la table **Movie** > **Concepteur de vues**.





Notez l'icône de clé en regard de **ID**. Par défaut, EF fait d'une propriété nommée **ID** la clé primaire.

- Cliquez avec le bouton droit sur la table **Movie** > **Afficher les données**.



Amorcer la base de données

Créez une classe nommée `SeedData` dans l'espace de noms *Modèles*. Remplacez le code généré par ce qui suit :

```

using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        {
            public static void Initialize(IServiceProvider serviceProvider)
            {
                using (var context = new MvcMovieContext(
                    serviceProvider.GetRequiredService<
                        DbContextOptions<MvcMovieContext>>()))
                {
                    // Look for any movies.
                    if (context.Movie.Any())
                    {
                        return; // DB has been seeded
                    }

                    context.Movie.AddRange(
                        new Movie
                        {
                            Title = "When Harry Met Sally",
                            ReleaseDate = DateTime.Parse("1989-2-12"),
                            Genre = "Romantic Comedy",
                            Price = 7.99M
                        },

                        new Movie
                        {
                            Title = "Ghostbusters ",
                            ReleaseDate = DateTime.Parse("1984-3-13"),
                            Genre = "Comedy",
                            Price = 8.99M
                        },

                        new Movie
                        {
                            Title = "Ghostbusters 2",
                            ReleaseDate = DateTime.Parse("1986-2-23"),
                            Genre = "Comedy",
                            Price = 9.99M
                        },

                        new Movie
                        {
                            Title = "Rio Bravo",
                            ReleaseDate = DateTime.Parse("1959-4-15"),
                            Genre = "Western",
                            Price = 3.99M
                        }
                    );
                    context.SaveChanges();
                }
            }
        }
    }
}

```

Si la base de données contient des films, l'initialiseur de valeur initiale retourne une valeur et aucun film n'est ajouté.

```
if (context.Movie.Any())
{
    return; // DB has been seeded.
}
```

Ajouter l'initialiseur de valeur initiale

Remplacez le contenu de *Program.cs* par le code suivant :

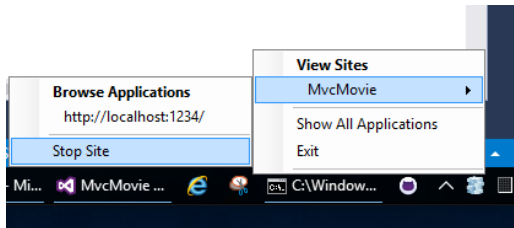
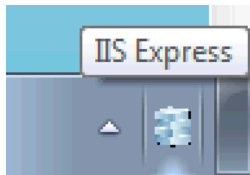


```
16 public class Program
17 {
18     // Références
19     public static void Main(string[] args)
20     {
21         var host = CreateHostBuilder(args).Build();
22
23         using (var scope = host.Services.CreateScope())
24         {
25             var services = scope.ServiceProvider;
26             try
27             {
28                 var context = services.GetRequiredService<MvcMovieContext>();
29                 context.Database.Migrate();
30                 SeedData.Initialize(services);
31             }
32             catch (Exception ex)
33             {
34                 var logger = services.GetRequiredService<ILogger<Program>>();
35                 logger.LogError(ex, "An error occurred seeding the DB.");
36                 throw;
37             }
38         }
39
40         host.Run();
41     }
42 }
```

Tester l'application

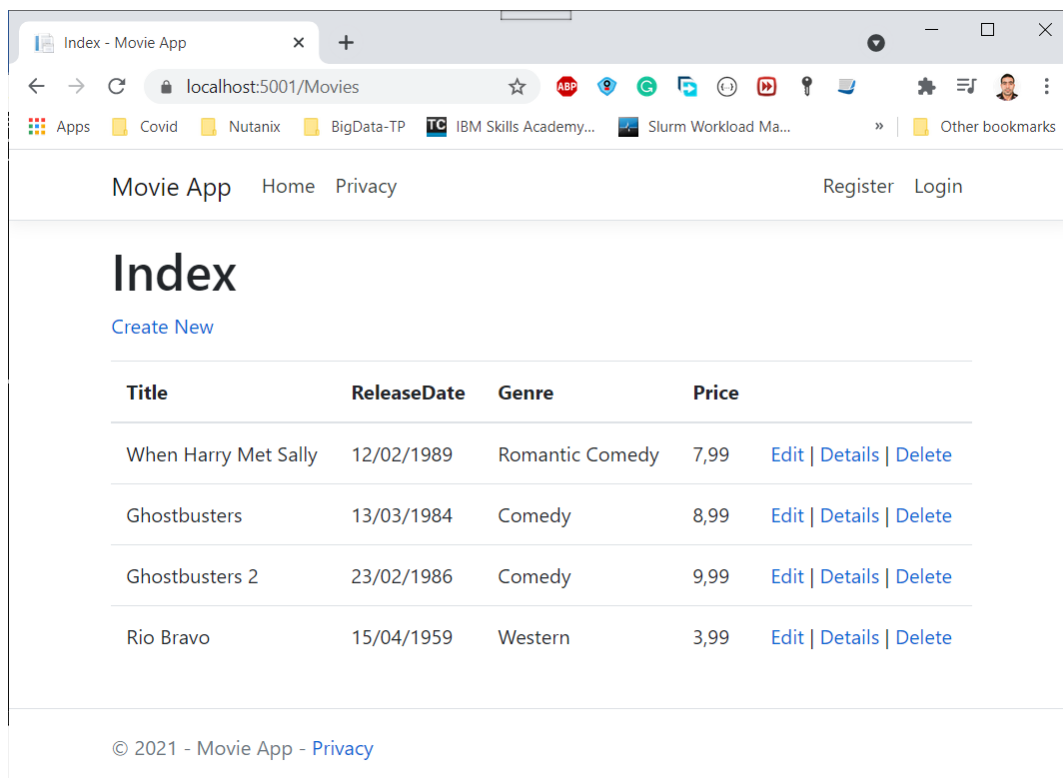
- **Supprimez** tous les enregistrements de la base de données. Pour ce faire, utilisez les liens de suppression disponibles dans le navigateur ou à partir de SSOX.
- Forcez l'application à s'initialiser (appelez les méthodes de la classe `Startup`) pour que la méthode seed s'exécute. Pour forcer l'initialisation, IIS Express doit être arrêté et redémarré. Pour cela, adoptez l'une des approches suivantes :

- Cliquez avec le bouton droit sur l'icône de barre d'état système IIS Express dans la zone de notification, puis appuyez sur **Quitter** ou sur **Arrêter le site**.



- Si vous exécutez Visual Studio en mode sans débogage, appuyez sur F5 pour l'exécuter en mode débogage.
- Si vous exécutez Visual Studio en mode débogage, arrêtez le débogueur et appuyez sur F5.

L'application affiche les données de départ.



Méthodes et vues de contrôleur dans ASP.NET Core

Nous avons une bonne ébauche de l'application de films, mais sa présentation n'est pas idéale, par exemple, **ReleaseDate** devrait être écrit en deux mots.

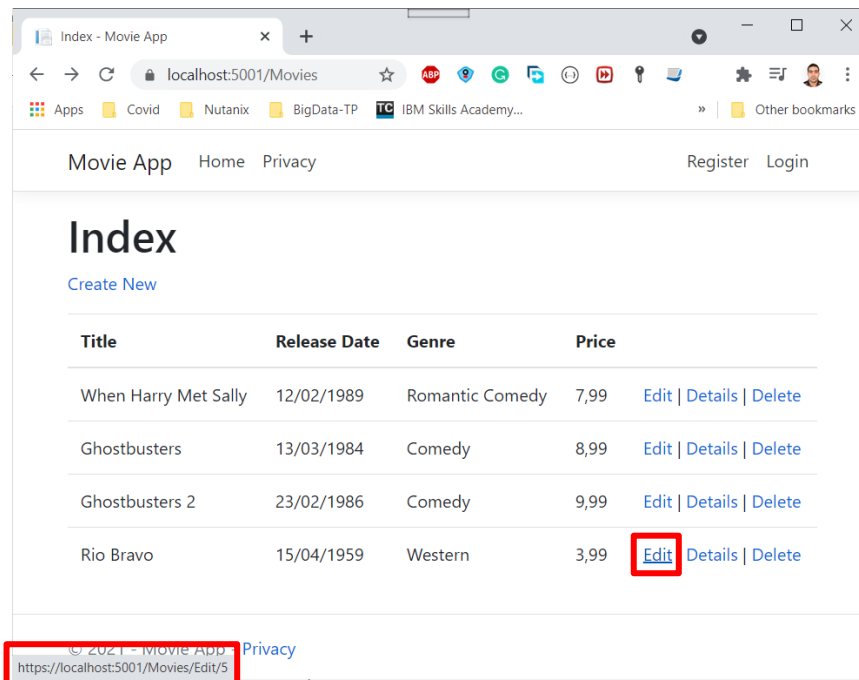
Ouvrez le fichier *Models/Movie.cs*, puis ajoutez les lignes en surbrillance ci-dessous :

```
8 namespace MvcMovie.Models
9 {
10     public class Movie
11     {
12         public int Id { get; set; }
13         public string Title { get; set; }
14
15         [Display(Name = "Release Date")]
16         [DataType(DataType.Date)]
17         public DateTime ReleaseDate { get; set; }
18         public string Genre { get; set; }
19         [Column(TypeName = "decimal(18, 2)")]
20         public decimal Price { get; set; }
21     }
22 }
```

Nous abordons DataAnnotations dans la suite de ce tutoriel. L'attribut `Display` spécifie les éléments à afficher pour le nom d'un champ (dans le cas présent, « Release Date » au lieu de « ReleaseDate »). L'attribut `DataType` spécifie le type des données (Date). Les informations d'heures stockées dans le champ ne s'affichent donc pas.

L'annotation de données `[Column(TypeName = "decimal(18, 2)")]` est nécessaire pour qu'Entity Framework Core puisse correctement mapper `Price` en devise dans la base de données.

Accédez au contrôleur `Movies` et maintenez le pointeur de la souris sur un lien **Edit** pour afficher l'URL cible.



Les liens **Edit**, **Details** et **Delete** sont générés par le Tag Helper Anchor Core MVC dans le fichier *Views/Movies/Index.cshtml*.

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
<a asp-action="Details" asp-route-id="@item.ID">Details</a> |
<a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
</td>
</tr>
```

Les Tag Helpers permettent au code côté serveur de participer à la création et au rendu des éléments HTML dans les fichiers Razor. Dans le code ci-dessus, le `AnchorTagHelper` génère dynamiquement la valeur d'attribut `href` HTML à partir de l'ID d'itinéraire et de la méthode d'action de contrôleur. Utilisez **Afficher la Source** dans votre navigateur favori ou les outils de développement pour examiner le balisage généré. Une partie du code HTML généré est affichée ci-dessous :

```
<td>
  <a href="/Movies/Edit/4"> Edit </a> |
  <a href="/Movies/Details/4"> Details </a> |
  <a href="/Movies/Delete/4"> Delete </a>
</td>
```

Rappelez-vous le format du routage défini dans le fichier *Startup.cs* :

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

ASP.NET Core traduit `https://localhost:5001/Movies/Edit/4` en une requête à la méthode d'action `Edit` du contrôleur `Movies` avec un paramètre `Id` de 4. (Les méthodes de contrôleur sont également appelées méthodes d'action.)

Les Tag Helpers sont l'une des nouvelles fonctionnalités les plus populaires dans ASP.NET Core.

Ouvrez le contrôleur `Movies` et examinez les deux méthodes d'action `Edit`. Le code suivant montre la méthode `HTTP GET Edit`, qui extrait le film et renseigne le formulaire de modification généré par le fichier Razor `Edit.cshtml`.

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

Le code suivant montre la méthode `HTTP POST Edit`, qui traite les valeurs de film publiées :


```

// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] M
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}

```

L'attribut `[Bind]` est l'un des moyens qui permettent d'assurer une protection contre la sur-publication (over-posting). Vous devez inclure dans l'attribut `[Bind]` uniquement les propriétés que vous souhaitez modifier. Les ViewModels fournissent une alternative pour empêcher la sur-publication.

Notez que la deuxième méthode d'action `Edit` est précédée de l'attribut `[HttpPost]`.

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] M
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}

```

L'attribut `[HttpPost]` indique que cette méthode `Edit` peut être appelée *uniquement* pour les requêtes `POST`. Vous pouvez appliquer l'attribut `[HttpGet]` à la première méthode `Edit`, mais cela n'est pas nécessaire car `[HttpGet]` est la valeur par défaut.

L'attribut `[ValidateAntiForgeryToken]` est utilisé pour lutter contre la falsification de requête. Il est associé à un jeton anti-contrefaçon généré dans le fichier de la vue `Edit` (`Views/Movies/Edit.cshtml`). Le fichier de la vue `Edit` génère le jeton anti-contrefaçon avec le Tag Helper `Form`.

```
<form asp-action="Edit">
```

Le Tag Helper Form génère un jeton anti-contrefaçon masqué qui doit correspondre au jeton anti-contrefaçon généré par `[ValidateAntiForgeryToken]` dans la méthode `Edit` du contrôleur `Movies`.

La méthode `HttpGet Edit` prend le paramètre `ID` du film, recherche le film à l'aide de la méthode Entity Framework `FindAsync`, et retourne le film sélectionné à la vue `Edit`. Si un film est introuvable, l'erreur `NotFound` (HTTP 404) est retournée.

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

Quand le système de génération de modèles automatique a créé la vue `Edit`, il a examiné la classe `Movie` et a créé le code pour restituer les éléments `<label>` et `<input>` de chaque propriété de la classe. L'exemple suivant montre la vue `Edit` qui a été générée par le système de génération de modèles automatique de Visual Studio :

```

@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Id" />
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ReleaseDate" class="control-label"></label>
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Genre" class="control-label"></label>
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Notez que le modèle de vue comporte une instruction `@model MvcMovie.Models.Movie` en haut du fichier. `@model MvcMovie.Models.Movie` indique que la vue s'attend à ce que le modèle pour le modèle de vue soit de type `Movie`.

Le code de génération de modèles automatique utilise plusieurs méthodes Tag Helper afin de rationaliser le balisage HTML. Le Tag Helper Label affiche le nom du champ (« Title », « ReleaseDate », « Genre » ou « Price »). Le Tag Helper Input affiche l'élément `<input>` HTML. Le Tag Helper Validation affiche les messages de validation associés à cette propriété.

Exécutez l'application et accédez à l'URL `/Movies`. Cliquez sur un lien **Edit**. Dans le navigateur, affichez la source de la page. Le code HTML généré pour l'élément `<form>` est indiqué ci-dessous.

```
<form action="/Movies/Edit/7" method="post">
  <div class="form-horizontal">
    <h4>Movie</h4>
    <hr />
    <div class="text-danger" />
    <input type="hidden" data-val="true" data-val-required="The ID field is required."
    <div class="form-group">
      <label class="control-label col-md-2" for="Genre" />
      <div class="col-md-10">
        <input class="form-control" type="text" id="Genre" name="Genre" value="Wes
        <span class="text-danger field-validation-valid" data-valmsg-for="Genre" d
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-md-2" for="Price" />
      <div class="col-md-10">
        <input class="form-control" type="text" data-val="true" data-val-number="T
        <span class="text-danger field-validation-valid" data-valmsg-for="Price" d
      </div>
    </div>
    <!-- Markup removed for brevity -->
    <div class="form-group">
      <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Save" class="btn btn-default" />
      </div>
    </div>
    <div>
      <input name="__RequestVerificationToken" type="hidden" value="CfDJ8Inyxgp63fRFqUePGvuI
    </div>
  </div>
</form>
```

Les éléments `<input>` sont dans un élément `HTML <form>` dont l'attribut `action` est défini de façon à publier à l'URL `/Movies/Edit/id`. Les données du formulaire sont publiées au serveur en cas de clic sur le bouton `Save`. La dernière ligne avant l'élément `</form>` de fermeture montre le jeton XSRF masqué généré par le Tag Helper Form.

Traitement de la requête POST

Le code suivant montre la version `[HttpPost]` de la méthode d'action `Edit`.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] M
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

L'attribut `[ValidateAntiForgeryToken]` valide le jeton XSRF masqué généré par le générateur de jetons anti-contrefaçon dans le Tag Helper Form

Le système de liaison de modèle prend les valeurs de formulaire publiées et crée un objet `Movie` qui est passé en tant que paramètre `movie`. La méthode `ModelState.IsValid` vérifie que les données envoyées dans le formulaire peuvent être utilisées pour changer (modifier ou mettre à jour) un objet `Movie`. Si les données sont valides, elles sont enregistrées. Les données de film mises à jour (modifiées) sont enregistrées dans la base de données en appelant la méthode `SaveChangesAsync` du contexte de base de données. Après avoir enregistré les données, le code redirige l'utilisateur vers la méthode d'action `Index` de la classe `MoviesController`, qui affiche la collection de films, avec notamment les modifications qui viennent d'être apportées.

Avant que le formulaire soit publié sur le serveur, la validation côté client vérifie les règles de validation sur les champs. En cas d'erreur de validation, un message d'erreur s'affiche et le formulaire n'est pas publié. Si JavaScript est désactivé, aucune validation côté client n'est effectuée, mais le serveur détecte les valeurs publiées qui ne sont pas valides, et les valeurs de formulaire sont réaffichées avec des messages d'erreur. Plus loin dans ce lab, nous examinerons la Validation du modèle plus en détail. Le Tag Helper Validation dans le modèle de vue `Views/Movies/Edit.cshtml` se charge de l'affichage des messages d'erreur appropriés.

The screenshot shows a web browser window with the address bar at `localhost:5001/Movie...`. The page title is "Movie App". The main heading is "Edit Movie". The form contains the following fields:

- Title:
- Release Date: with a calendar icon. Below it, a red error message says "The Release Date field is required."
- Genre:
- Price:

Below the form, there is a blue "Save" button and a blue link "Back to List". The footer of the page reads "© 2021 - Movie App - Privacy".

Toutes les méthodes `HttpGet` du contrôleur `Movies` suivent un modèle similaire. Elles reçoivent un objet de film (ou une liste d'objets, dans le cas de `Index`) et passent l'objet (modèle) à la vue. La méthode `Create` passe un objet de film vide à la vue `Create`. Toutes les méthodes qui créent, modifient, suppriment ou changent d'une quelconque manière des données le font dans la surcharge `[HttpPost]` de la méthode. Modifier des données dans une méthode `HTTP GET` présente un risque pour la sécurité. La modification des données dans une méthode `HTTP GET` enfreint également les bonnes pratiques HTTP et le modèle architectural REST, qui spécifie que les requêtes GET ne doivent pas changer l'état de votre application. En d'autres termes, une opération GET doit être sûre, ne doit avoir aucun effet secondaire et ne doit pas modifier vos données persistantes.

Ajouter une fonction de recherche à une application ASP.NET Core MVC

Dans cette section, vous ajoutez une fonctionnalité de recherche à la méthode d'action `Index` qui vous permet de rechercher des films par *genre* ou par *nom*.

Mettez à jour la méthode `Index` avec le code suivant :

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

La première ligne de la méthode d'action `Index` crée une requête LINQ pour sélectionner les films :

```
var movies = from m in _context.Movie
              select m;
```

La requête est *seulement* définie à ce stade, elle n'a **pas** été exécutée sur la base de données.

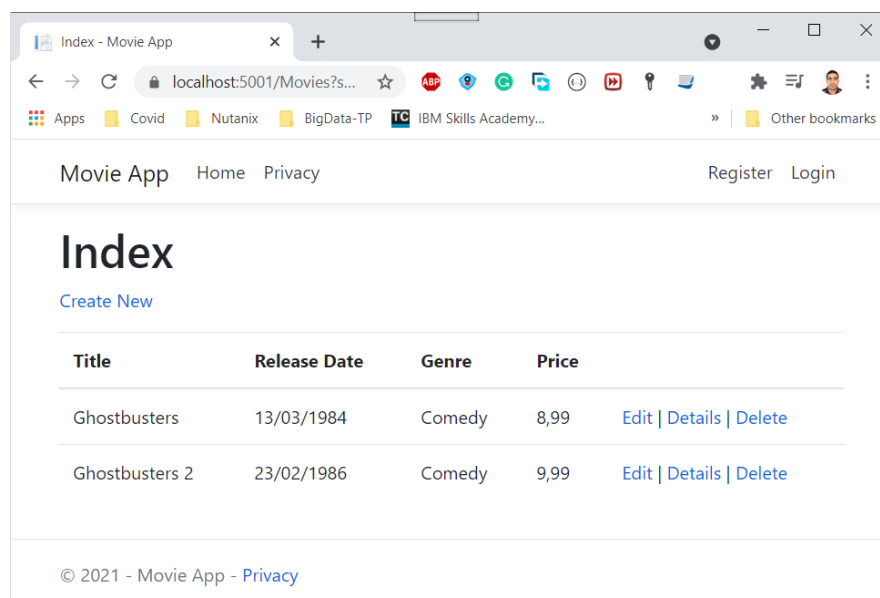
Si le paramètre `searchString` contient une chaîne, la requête de films est modifiée de façon à filtrer sur la valeur de la chaîne de recherche :

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

Le code `s => s.Title.Contains()` ci-dessus est une expression lambda. Les expressions lambda sont utilisées dans les requêtes LINQ basées sur une méthode en tant qu'arguments pour les méthodes d'opérateur de requête standard, comme la méthode `Where` ou `Contains` (utilisée dans le code ci-dessus). Les requêtes LINQ ne sont pas exécutées quand elles sont définies ou quand elles sont modifiées en appelant une méthode, comme `Where`, `Contains` ou `OrderBy`. Au lieu de cela, l'exécution de la requête est différée. Cela signifie que l'évaluation d'une expression est retardée jusqu'à ce que sa valeur réalisée fasse l'objet d'une itération réelle ou que la méthode `ToListAsync` soit appelée.

Remarque : La méthode `Contains` est exécutée sur la base de données, et non pas dans le code C# ci-dessus. Le respect de la casse pour la requête dépend de la base de données et du classement. Sur SQL Server, `Contains` est mappé à SQL LIKE, qui ne respecte pas la casse. Dans SQLite, avec le classement par défaut, elle respecte la casse.

Accédez à `/Movies/Index`. Ajoutez une chaîne de requête comme `?searchString=Ghost` à l'URL. Les films filtrés sont affichés.



Index			
Create New			
Title	Release Date	Genre	Price
Ghostbusters	13/03/1984	Comedy	8,99
Ghostbusters 2	23/02/1986	Comedy	9,99

Si vous changez la signature de la méthode `Index` pour y inclure un paramètre nommé `id`, le paramètre `id` correspondra à l'espace réservé facultatif `{id}` pour les routes par défaut définies dans *Startup.cs*.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Remplacez le paramètre par `id` et toutes les occurrences de `searchString` par `id`.

La méthode `Index` précédente :

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                  select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

La méthode `Index` mise à jour avec le paramètre `id` :

```

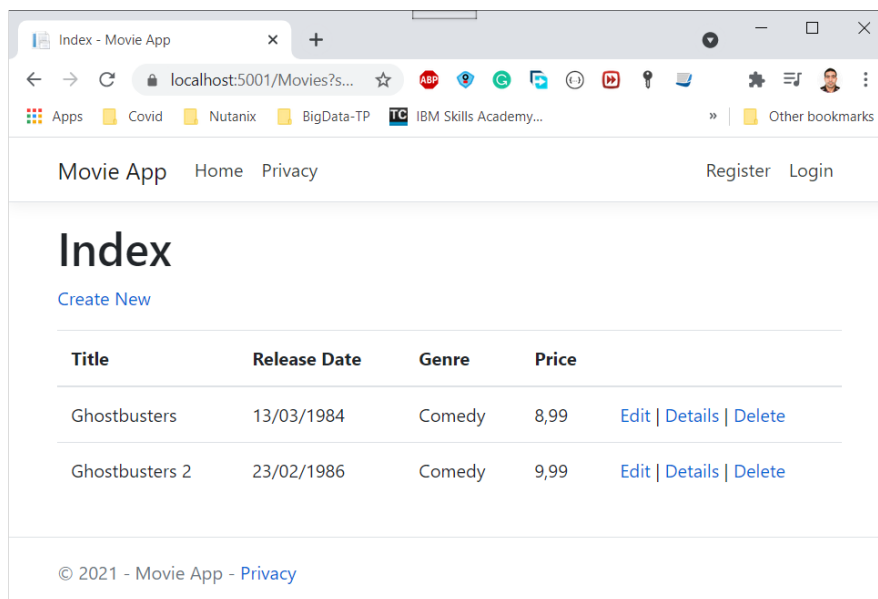
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                  select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(await movies.ToListAsync());
}

```

Vous pouvez maintenant passer le titre de la recherche en tant que données de routage (un segment de l'URL) et non pas en tant que valeur de chaîne de requête.



Cependant, vous ne pouvez pas attendre des utilisateurs qu'ils modifient l'URL à chaque fois qu'ils veulent rechercher un film. Vous allez donc maintenant ajouter des éléments d'interface utilisateur pour les aider à filtrer les films. Si vous avez changé la signature de la méthode `Index` pour tester comment passer le paramètre `ID` lié à une route, rétablissez-la de façon à ce qu'elle prenne un paramètre nommé `searchString`:

```

public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                  select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}

```

Ouvrez le fichier *Views/Movies/Index.cshtml* et ajoutez le balisage `<form>` mis en surbrillance ci-dessous :

```

    ViewData["Title"] = "Index";
}

<h2>Index</h2>

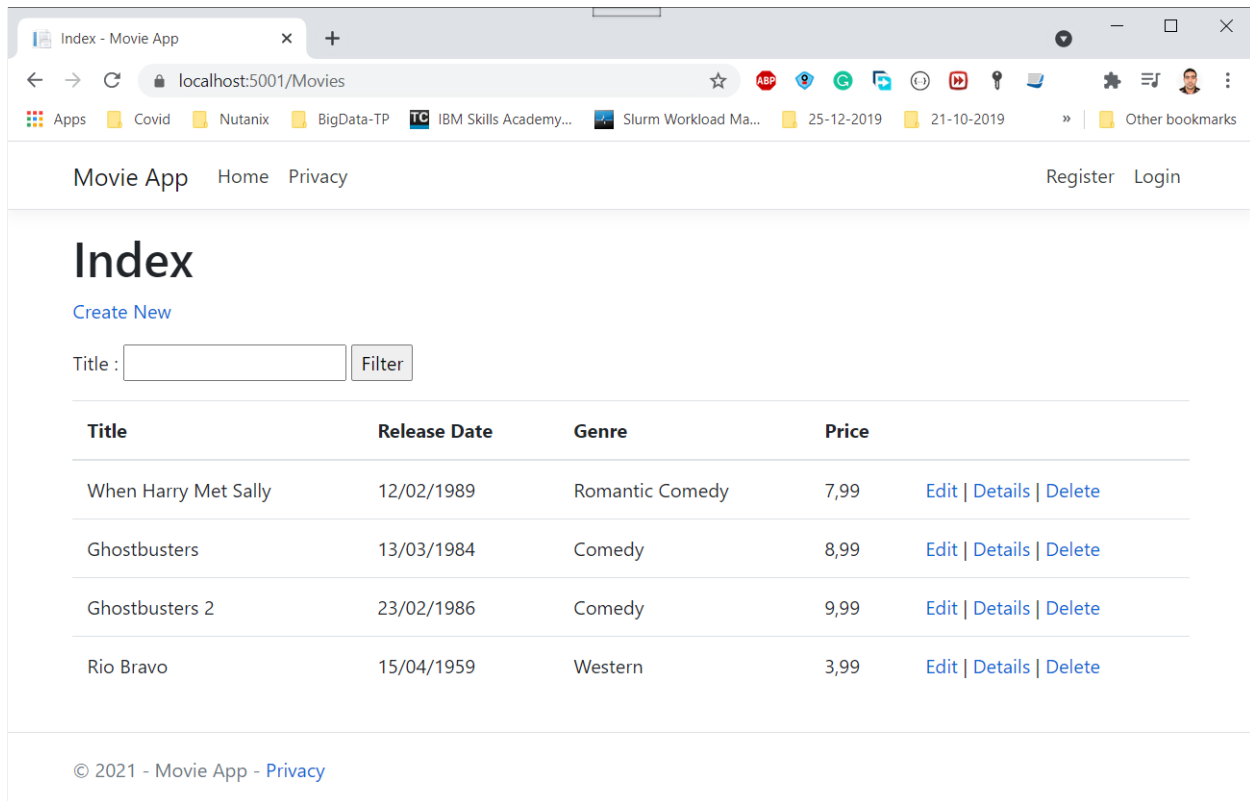
<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>

```

La balise HTML `<form>` utilise le Tag Helper de formulaire, de façon que quand vous envoyez le formulaire, la chaîne de filtrage soit envoyée à l'action `Index` du contrôleur de films. Enregistrez vos modifications puis testez le filtre.



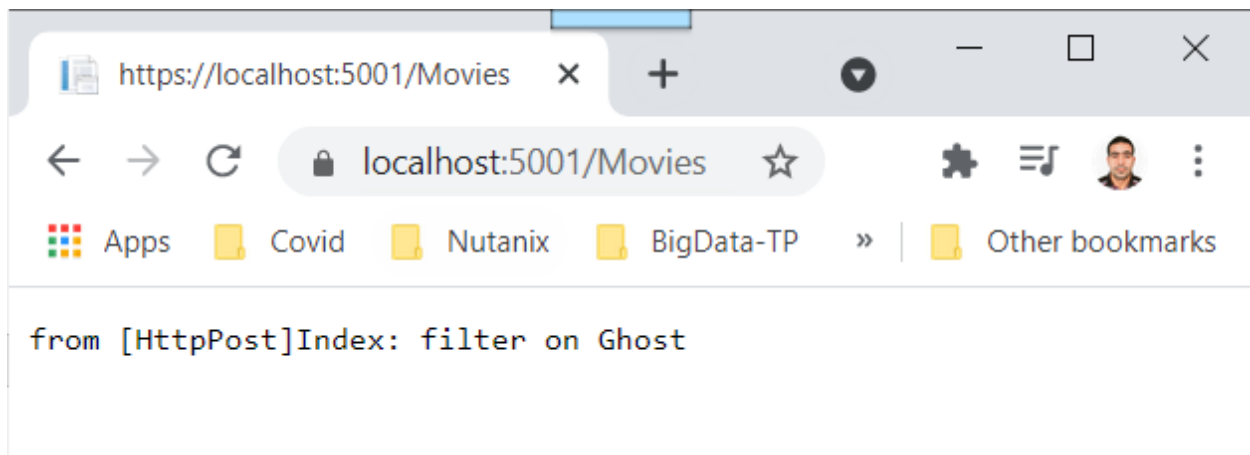
Contrairement à ce que vous pourriez penser, une surcharge de `[HttpPost]` dans la méthode `Index` n'est pas nécessaire. Vous n'en avez pas besoin, car la méthode ne change pas l'état de l'application, elle filtre seulement les données.

Vous pourriez ajouter la méthode `[HttpPost] Index` suivante.

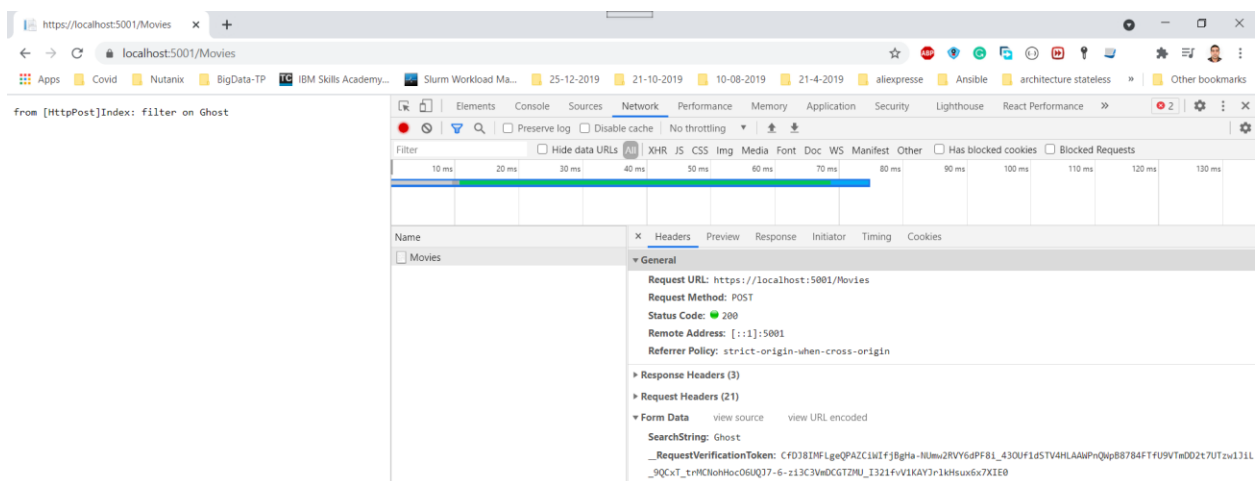
```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

Le paramètre `notUsed` est utilisé pour créer une surcharge pour la méthode `Index`. Nous parlons de ceci plus loin dans le lab.

Si vous ajoutez cette méthode, le demandeur de l'action correspondrait à la méthode `[HttpPost] Index` et la méthode `[HttpPost] Index` s'exécuterait comme indiqué dans l'image ci-dessous.



Cependant, même si vous ajoutez cette version `[HttpPost]` de la méthode `Index`, il existe une limitation dans la façon dont tout ceci a été implémenté. Imaginez que vous voulez insérer un signet pour une recherche spécifique, ou que vous voulez envoyer un lien à vos amis sur lequel ils peuvent cliquer pour afficher la même liste filtrée de films. Notez que l'URL de la requête HTTP POST est identique à l'URL de la requête GET (`localhost:xxxxx/Movies/Index`) : il n'existe aucune information de recherche dans l'URL. Les informations de la chaîne de recherche sont envoyées au serveur en tant que valeur d'un champ de formulaire. Vous pouvez vérifier ceci avec les outils de développement du navigateur ou avec l'excellent outil Fiddler. L'illustration ci-dessous montre les outils de développement du navigateur Chrome :



Vous pouvez voir le paramètre de recherche et le jeton XSRF dans le corps de la demande. Notez que, comme indiqué dans le tutoriel précédent, le Tag Helper de formulaire génère un jeton XSRF anti-contrefaçon. Nous ne modifions pas les données : nous n'avons donc pas besoin de valider le jeton dans la méthode du contrôleur.

Comme le paramètre de recherche se trouve dans le corps de la demande et pas dans l'URL, vous ne pouvez pas capturer ces informations de recherche pour les insérer dans un signet ou les partager avec d'autres personnes. Résolvez ce problème en spécifiant que la requête doit être `HTTP GET` :

```
@model IEnumerable<MvcMovie.Models.Movie>

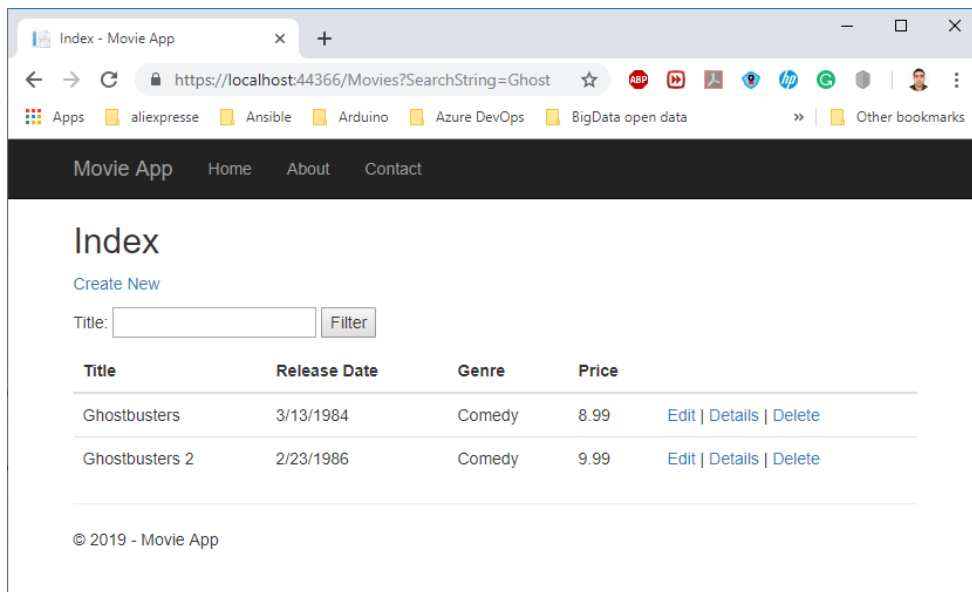
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
        </tr>
    </thead>
</table>
```

Maintenant, quand vous soumettez une recherche, l'URL contient la chaîne de requête de la recherche. La recherche accède également à la méthode d'action `HttpGet Index`, même si vous avez une méthode `HttpPost Index`.



La mise en forme suivante montre la modification apportée à la balise `form` :

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

Ajouter la recherche par genre

Ajoutez la classe `MovieGenreViewModel` suivante au dossier *Models* :

```

9      public class MovieGenreViewModel
10     {
11         public List<Movie> Movies;
12         public SelectList Genres;
13         Oréférences
14         public string MovieGenre { get; set; }
15         Oréférences
16         public string SearchString { get; set; }
17     }
18 
```

Le modèle de vue `MovieGenreViewModel` contiendra :

- Une liste de films.
- Une `SelectList` contenant la liste des genres. Cela permet à l'utilisateur de sélectionner un genre dans la liste.
- `MovieGenre`, qui contient le genre sélectionné.

- `SearchString`, qui contient le texte que les utilisateurs entrent dans la zone de texte de recherche.

Remplacez la méthode `Index` dans `MoviesController.cs` par le code suivant :

```
// GET: Movies
public async Task<IActionResult> Index(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel
    {
        Genres = new SelectList(await genreQuery.Distinct().ToListAsync()),
        Movies = await movies.ToListAsync()
    };

    return View(movieGenreVM);
}
```

Le code suivant est une requête `LINQ` qui récupère tous les genres de la base de données.

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

La `SelectList` de genres est créée en projetant les genres distincts (nous ne voulons pas que notre liste de sélection ait des genres en doublon).

Quand l'utilisateur recherche l'élément, la valeur de recherche est conservée dans la zone de recherche.

Ajouter la recherche par genre à la vue Index

Mettez à jour `Index.cshtml` comme suit :

```
<th>
    @Html.DisplayNameFor(model => model.Movies[0].Genre)
</th>
<th>
    @Html.DisplayNameFor(model => model.Movies[0].Price)
</th>
</tr>
</thead>
<tbody>
    @foreach (var item in Model.Movies)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
    }
</tbody>
</table>

@model MvcMovie.Models.MovieGenreViewModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>

        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

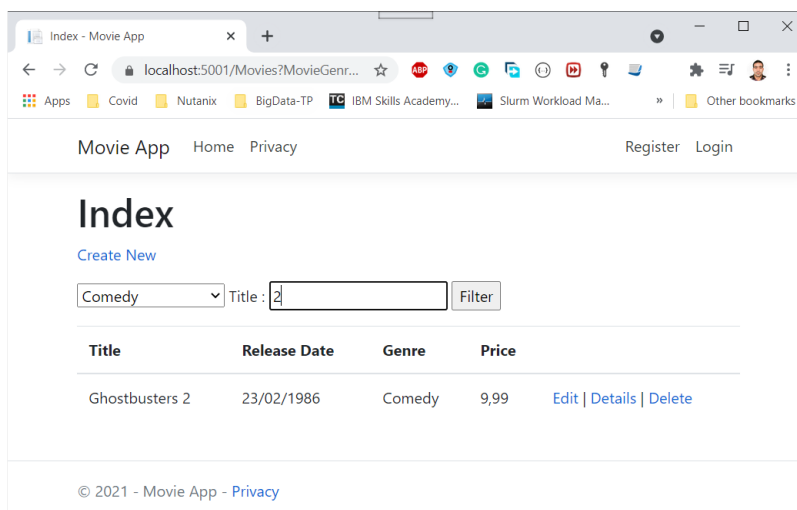
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
            </th>
```

Examinez l'expression lambda utilisée dans le Helper HTML suivant :

```
@Html.DisplayNameFor(model => model.Movies[0].Title)
```

Dans le code précédent, le Helper HTML `DisplayNameFor` inspecte la propriété `Title` référencée dans l'expression lambda pour déterminer le nom d'affichage. Comme l'expression lambda est inspectée et non pas évaluée, vous ne recevez pas de violation d'accès quand `model`, `model.Movies` ou `model.Movies[0]` sont `null` ou vides. Quand l'expression lambda est évaluée (par exemple `@Html.DisplayFor(modelItem => item.Title)`), les valeurs des propriétés du modèle sont évaluées.

Testez l'application en effectuant une recherche par genre, par titre de film et selon ces deux critères :



Ajouter un nouveau champ à une application ASP.NET Core MVC

Dans cette section, Migrations Entity Framework Code First est utilisé pour :

- Ajouter un nouveau champ au modèle.
- Migrer le nouveau champ vers la base de données.

Quand EF Code First est utilisé pour créer automatiquement une base de données, Code First :

- Ajoute une table à la base de données pour en suivre le schéma.

- Vérifie que la base de données est synchronisée avec les classes de modèle à partir desquelles elle a été générée. S'ils ne sont pas synchronisés, EF lève une exception. Cela facilite la recherche de problèmes d'incohérence de code/de bases de données.

Ajouter une propriété Rating au modèle Movie

Ajouter une propriété `Rating` à *Models/Movie.cs* :

```
public class Movie
{
    public int Id { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Générez l'application (Ctrl+Maj+B).

Comme vous avez ajouté un nouveau champ à la classe `Movie`, vous devez mettre à jour la liste verte des liaisons pour y inclure cette nouvelle propriété. Dans *MoviesController.cs*, mettez à jour l'attribut `[Bind]` des méthodes d'action `Create` et `Edit` pour y inclure la propriété `Rating` :

```
[Bind("ID,Title,ReleaseDate,Genre,Price,Rating")]
```

Mettez à jour les modèles de vue pour afficher, créer et modifier la nouvelle propriété `Rating` dans la vue du navigateur.

Ouvrez le fichier */Views/Movies/Index.cshtml* et ajoutez un champ `Rating` :

```

<thead>
  <tr>
    <th>
      @Html.DisplayNameFor(model => model.Movies[0].Title)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Movies[0].Genre)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Movies[0].Price)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Movies[0].Rating)
    </th>
  </tr>
</thead>
<tbody>
  @foreach (var item in Model.Movies)
  {
    <tr>
      <td>
        @Html.DisplayFor(modelItem => item.Title)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.ReleaseDate)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Genre)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Price)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Rating)
      </td>
      <td>
        <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |

```


Mettez à jour le fichier `/Views/Movies/Create.cshtml` avec un champ `Rating`.

Vous pouvez copier/coller le « groupe de formulaire » précédent et laisser IntelliSense vous aider à mettre à jour les champs. IntelliSense fonctionne avec des Tag Helpers.

```

</div>
<div class="form-group">
  <label asp-for="Title" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    <input asp-for="Title" class="form-control" />
    <span asp-validation-for="Title" class="text-danger" />
  </div>
</div>
<div class="form-group">
  <label asp-for="R" class="col-md-2 control-label"></label>
  <div class="col-
    <input asp-f
    <span asp-va
  </div>
</div>
<div class="form-gro
  <div class="col-
    <input type=
  </div>
</div>
</div>
</form>

```



Mettez à jour la classe `SeedData` pour qu'elle fournisse une valeur pour la nouvelle colonne. Vous pouvez voir un exemple de modification ci-dessous, mais elle doit être appliquée à chaque `new Movie`.

```

new Movie
{
  Title = "When Harry Met Sally",
  ReleaseDate = DateTime.Parse("1989-1-11"),
  Genre = "Romantic Comedy",
  Rating = "R",
  Price = 7.99M
},

```

L'application ne fonctionne pas tant que la base de données n'est pas mise à jour pour inclure le nouveau champ. Si elle est exécutée maintenant, l'erreur `SqlException` est levée :

```
SqlException: Invalid column name 'Rating'.
```

Cette erreur survient car la classe du modèle `Movie` mise à jour est différente du schéma de la table `Movie` de la base de données existante. (Il n'existe pas de colonne `Rating` dans la table de base de données.)

Plusieurs approches sont possibles pour résoudre l'erreur :

1. Laissez Entity Framework supprimer et recréer automatiquement la base de données sur la base du nouveau schéma de classe de modèle. Cette approche est très utile au début du cycle de développement quand vous effectuez un développement actif sur une base de données de test. Elle permet de faire évoluer rapidement le schéma de modèle et de base de données ensemble. L'inconvénient, cependant, est que vous perdez les données existantes dans la base de données. Par conséquent, n'utilisez pas cette approche sur une base de données de production. L'utilisation d'un initialiseur pour amorcer automatiquement une base de données avec des données de test est souvent un moyen efficace pour développer une application. Il s'agit d'une bonne approche pour le développement initial et lors de l'utilisation de SQLite.
2. Modifier explicitement le schéma de la base de données existante pour le faire correspondre aux classes du modèle. L'avantage de cette approche est que vous conservez vos données. Vous pouvez apporter cette modification manuellement ou en créant un script de modification de la base de données.
3. Utilisez les migrations Code First pour mettre à jour le schéma de base de données.

Pour ce tutoriel, les migrations Code First sont utilisées.

Dans le menu **Outils**, sélectionnez **Gestionnaire de package NuGet > Console du Gestionnaire de package**.

Dans la console du Gestionnaire de package, entrez les commandes suivantes :

```
PM> Add-Migration Rating -Context MvcMovieContext
```

```
PM> Update-Database -Context MvcMovieContext
```

La commande `Add-Migration` indique au framework de migration d'examiner le modèle `Movie` actuel avec le schéma de la base de données `Movie` actuel et de créer le code nécessaire pour migrer la base de données vers le nouveau modèle.

Le nom « Rating » est arbitraire et est utilisé pour nommer le fichier de migration. Il est utile d'utiliser un nom explicite pour le fichier de migration.

Si tous les enregistrements de la base de données sont supprimés, la méthode d'initialisation amorce la base de données et inclut le champ `Rating`.

Exécutez l'application et vérifiez que vous pouvez créer/modifier/afficher des films avec un champ `Rating`. Vous devez ajouter le **champ `Rating` aux modèles de vue `Edit`, `Details` et `Delete`**.



Ajouter une validation à une application ASP.NET Core MVC

Dans cette section :

- Une logique de validation est ajoutée au modèle `Movie`.
- Vous vous assurez que les règles de validation sont appliquées chaque fois qu'un utilisateur crée ou modifie un film.

Ne vous répétez pas

L'un des principes de conception de MVC est « Ne vous répétez pas » (désigné par l'acronyme DRY, Don't Repeat Yourself). ASP.NET Core MVC vous encourage à spécifier les fonctionnalités ou les comportements une seule fois, puis à utiliser la réflexion partout dans une application. Cela réduit la quantité de code à écrire, et rend le code que vous écrivez moins susceptible aux erreurs et plus facile à tester et à gérer.

La prise en charge de la validation fournie par MVC et Entity Framework Core Code First est un bon exemple du principe DRY en action. Vous pouvez spécifier de façon déclarative des règles de validation à un seul emplacement (dans la classe de modèle), et les règles sont appliquées partout dans l'application.

Ajout de règles de validation au modèle de film

Ouvrez le fichier *Movie.cs*. DataAnnotations fournit un ensemble intégré d'attributs de validation que vous appliquez de manière déclarative à n'importe quelle classe ou propriété. (Il contient également des attributs de mise en forme comme `DataType` qui aident à effectuer la mise en forme et ne fournissent aucune validation.)

Mettez à jour la classe `Movie` pour tirer parti des attributs de validation intégrés `Required`, `StringLength`, `RegularExpression` et `Range`.


```

10 public class Movie
11 {
12     11 références
13     public int Id { get; set; }
14
15     [StringLength(60, MinimumLength = 3)]
16     [Required]
17     17 références
18     public string Title { get; set; }
19
20     [Display(Name = "Release Date")]
21     [DataType(DataType.Date)]
22     16 références
23     public DateTime ReleaseDate { get; set; }
24
25     [RegularExpression(@"^[A-Z]+[a-zA-Z0-9'\s-]*$")]
26     [Required]
27     [StringLength(30)]
28     19 références
29     public string Genre { get; set; }
30
31     [Range(1, 100)]
32     [DataType(DataType.Currency)]
33     [Column(TypeName = "decimal(18, 2)")]
34     16 références
35     public decimal Price { get; set; }
36
37     [RegularExpression(@"^[A-Z]+[a-zA-Z0-9'\s-]*$")]
38     [Required]
39     [StringLength(5)]
40     14 références
41     public string Rating { get; set; }
42 }

```

Les attributs de validation spécifient le comportement que vous souhaitez appliquer sur les propriétés du modèle sur lesquels ils sont appliqués :

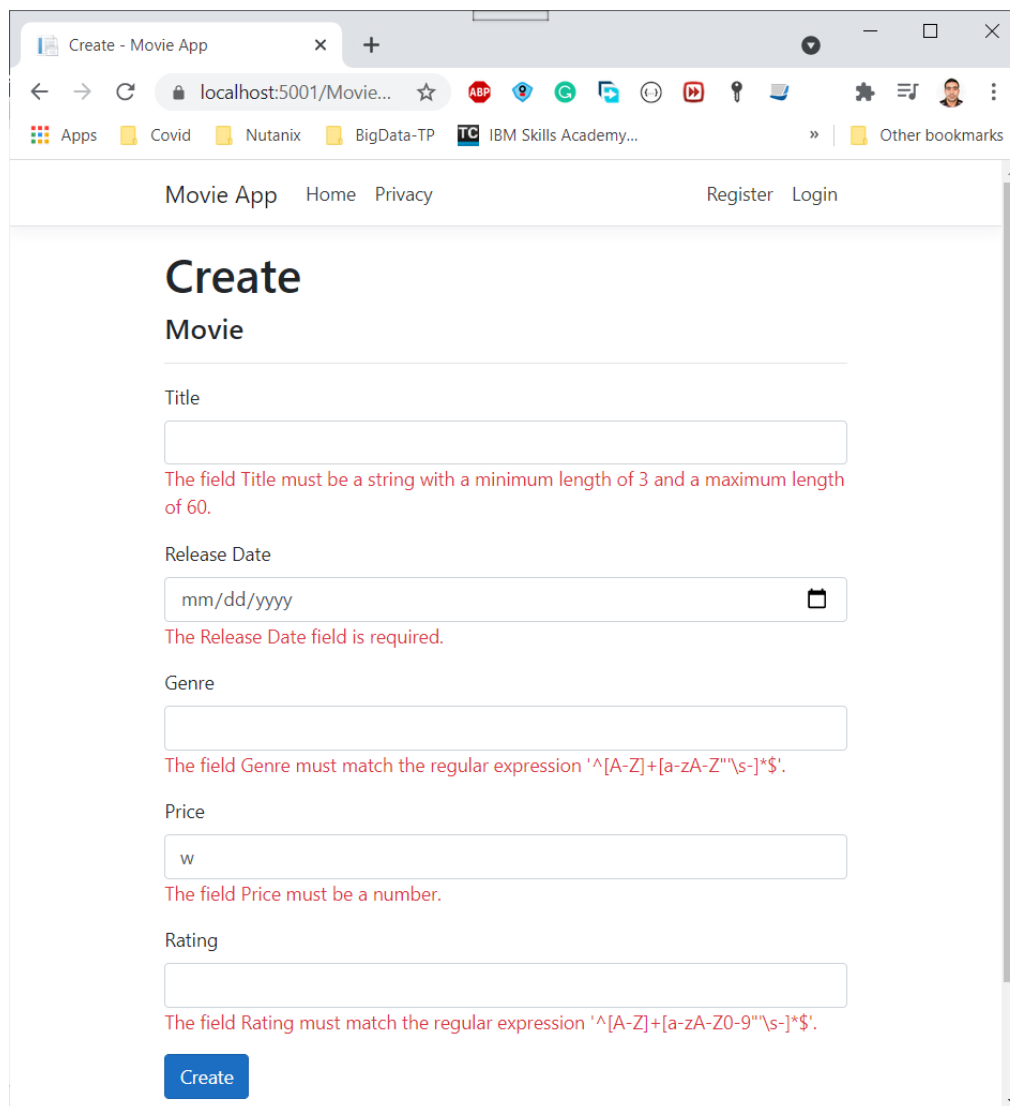
- Les attributs `Required` et `MinimumLength` indiquent qu'une propriété doit avoir une valeur, mais rien n'empêche un utilisateur d'entrer un espace blanc pour satisfaire cette validation.
- L'attribut `RegularExpression` sert à limiter les caractères pouvant être entrés. Dans le code ci-dessus, `Genre` et `Rating` doivent utiliser uniquement des lettres (la première lettre en majuscule, les espaces blancs, les chiffres et les caractères spéciaux ne sont pas autorisés).
- L'attribut `Range` contraint une valeur à une plage spécifiée.
- L'attribut `StringLength` vous permet de définir la longueur maximale d'une propriété de chaîne, et éventuellement sa longueur minimale.
- Les types valeur (tels que `decimal`, `int`, `float` et `DateTime`) sont obligatoires par nature et n'ont pas besoin de l'attribut `[Required]`.

L'application automatique des règles de validation par ASP.NET Core permet d'accroître la fiabilité de votre application. Cela garantit également que vous n'oublierez pas de valider un élément et que vous n'autoriserez pas par inadvertance l'insertion de données incorrectes dans la base de données.

Interface utilisateur d'erreur de validation dans MVC

Exécutez l'application et accédez au contrôleur Movies.

Appuyez sur le lien **Create New** pour ajouter un nouveau film. Remplissez le formulaire avec des valeurs non valides. Dès que la validation côté client jQuery détecte l'erreur, elle affiche un message d'erreur.



The screenshot shows a web browser window with the address bar displaying 'localhost:5001/Movie...'. The page title is 'Create - Movie App'. The navigation bar includes 'Movie App', 'Home', 'Privacy', 'Register', and 'Login'. The main content area is titled 'Create Movie' and contains a form with the following fields and validation messages:

- Title:** A text input field. Below it, a red message reads: 'The field Title must be a string with a minimum length of 3 and a maximum length of 60.'
- Release Date:** A date input field with a calendar icon. Below it, a red message reads: 'The Release Date field is required.'
- Genre:** A text input field. Below it, a red message reads: 'The field Genre must match the regular expression '^[A-Z]+[a-zA-Z0-9]*\$'.'
- Price:** A text input field containing the value 'w'. Below it, a red message reads: 'The field Price must be a number.'
- Rating:** A text input field. Below it, a red message reads: 'The field Rating must match the regular expression '^[A-Z]+[a-zA-Z0-9]*\$'.'

At the bottom of the form is a blue button labeled 'Create'.

Notez que le formulaire a affiché automatiquement un message d'erreur de validation approprié dans chaque champ contenant une valeur non valide. Les erreurs sont appliquées à la fois côté client (à l'aide de JavaScript et jQuery) et côté serveur (au cas où un utilisateur aurait désactivé JavaScript).

L'un des principaux avantages est que vous n'avez pas eu à changer une seule ligne de code dans la classe `MoviesController` ou dans la vue `Create.cshtml` pour activer cette interface utilisateur de validation. Le contrôleur et les vues créées précédemment dans ce tutoriel ont détecté les règles de validation que vous avez spécifiées à l'aide des attributs de validation sur les propriétés de la classe de modèle `Movie`. Testez la validation à l'aide de la méthode d'action `Edit`. La même validation est appliquée.

Les données de formulaire ne sont pas envoyées au serveur tant qu'il y a des erreurs de validation côté client. Vous pouvez vérifier cela en plaçant un point d'arrêt dans la méthode `HTTP Post`, en utilisant l'outil Fiddler ou à l'aide des Outils de développement F12.

Fonctionnement de la validation

Vous vous demandez peut-être comment l'interface utilisateur de validation a été générée sans aucune mise à jour du code dans le contrôleur ou dans les vues. Le code suivant montre les deux méthodes `Create`.

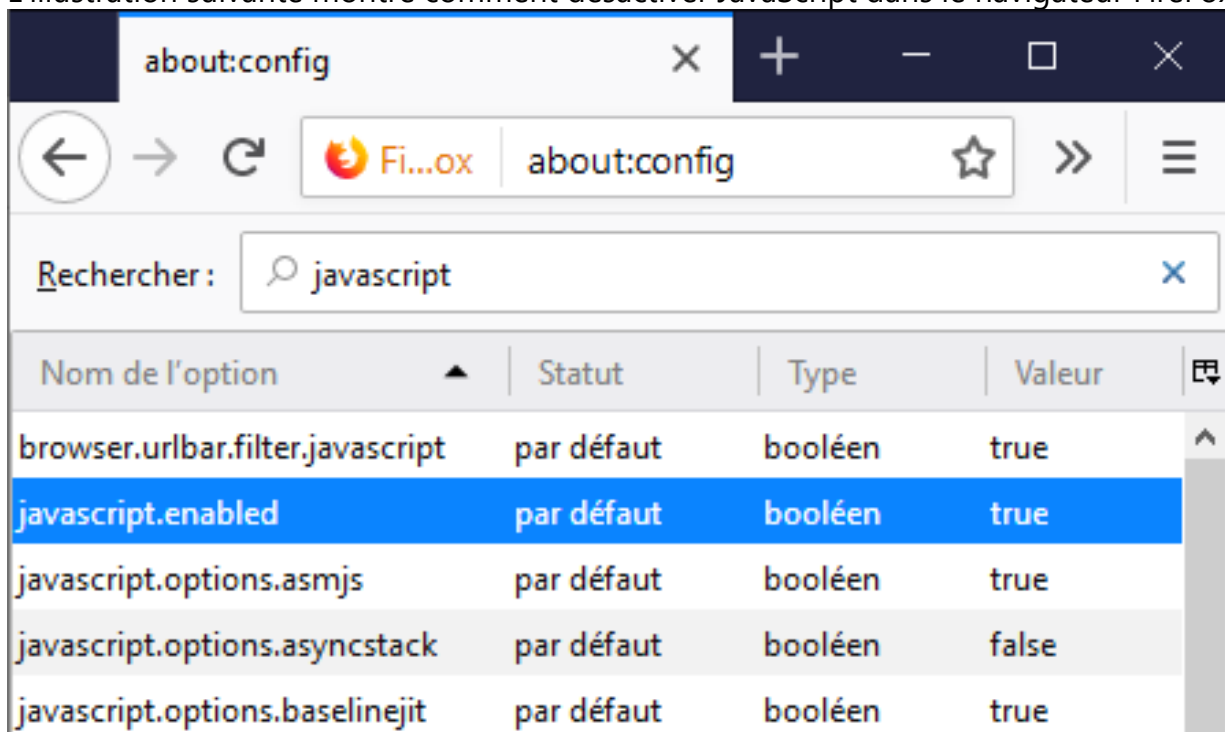
```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

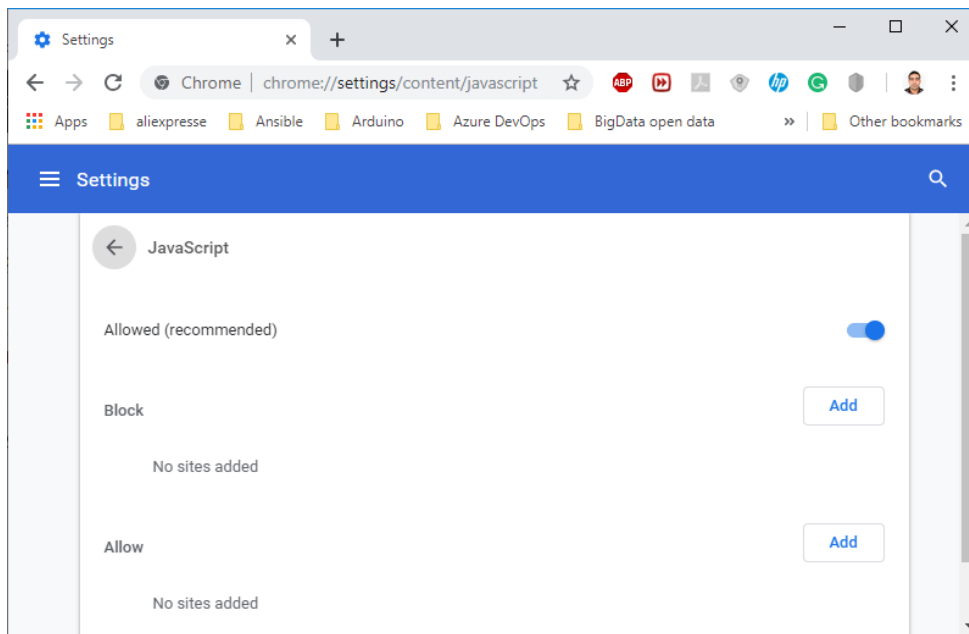
La première méthode d'action (HTTP GET) `Create` affiche le formulaire de création initial. La deuxième version (`[HttpPost]`) gère la publication de formulaire. La seconde méthode `Create` (la version `[HttpPost]`) appelle `ModelState.IsValid` pour vérifier si le film a des erreurs de validation. L'appel de cette méthode évalue tous les attributs de validation qui ont été appliqués à l'objet. Si l'objet comporte des erreurs de validation, la méthode `Create` réaffiche le formulaire. S'il n'y a pas d'erreur, la méthode enregistre le nouveau film dans la base de données. Dans notre exemple de film, le formulaire n'est pas publié sur le serveur quand des erreurs de validation sont détectées côté client ; la seconde méthode `Create` n'est jamais appelée quand il y a des erreurs de validation côté client. Si vous désactivez JavaScript dans votre navigateur, la validation client est désactivée et vous pouvez tester la méthode `Create` HTTP POST `ModelState.IsValid` pour détecter les erreurs de validation.

Vous pouvez définir un point d'arrêt dans la méthode `[HttpPost] Create` et vérifier que la méthode n'est jamais appelée. La validation côté client n'enverra pas les données du formulaire quand des erreurs de validation seront détectées. Si vous désactivez JavaScript dans votre navigateur et que vous envoyez ensuite le formulaire avec des erreurs, le point d'arrêt sera atteint. Vous bénéficiez toujours d'une validation complète sans JavaScript.

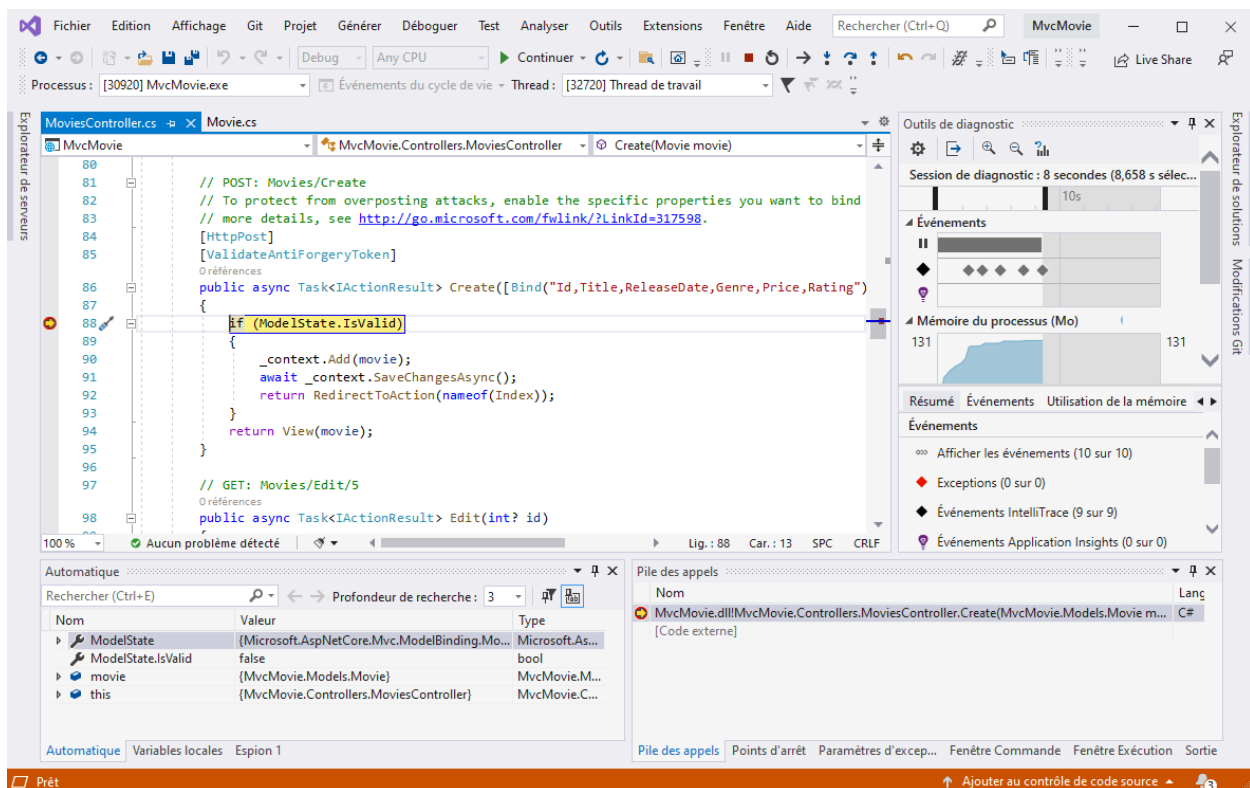
L'illustration suivante montre comment désactiver JavaScript dans le navigateur FireFox.



L'illustration suivante montre comment désactiver JavaScript dans le navigateur Chrome.



Après la désactivation de JavaScript, publiez les données non valides et parcourez le débogueur.



La partie du modèle d'affichage *Create.cshtml* est indiqué dans le balisage suivant :

```

<h4>Movie</h4>
<hr />
<div class="row">
  <div class="col-md-4">
    <form asp-action="Create">
      <div asp-validation-summary="ModelOnly" class="text-danger"></div>
      <div class="form-group">
        <label asp-for="Title" class="control-label"></label>
        <input asp-for="Title" class="form-control" />
        <span asp-validation-for="Title" class="text-danger"></span>
      </div>

      @*Markup removed for brevity.*@

```

Le balisage précédent est utilisé par les méthodes d'action pour afficher le formulaire initial et pour le réafficher en cas d'erreur.

Le Tag Helper Input utilise les attributs DataAnnotations et produit les attributs HTML nécessaires à la validation jQuery côté client. Le Tag Helper Validation affiche les erreurs de validation.

Le grand avantage de cette approche est que ni le contrôleur ni le modèle de vue `Create` ne savent rien des règles de validation appliquées ou des messages d'erreur affichés. Les règles de validation et les chaînes d'erreur sont spécifiées uniquement dans la classe `Movie`. Ces mêmes règles de validation sont automatiquement appliquées à la vue `Edit` et à tous les autres modèles de vues que vous pouvez créer et qui modifient votre modèle.

Quand vous devez changer la logique de validation, vous pouvez le faire à un seul endroit en ajoutant des attributs de validation au modèle (dans cet exemple, la classe `Movie`). Vous n'aurez pas à vous soucier des différentes parties de l'application qui pourraient être incohérentes avec la façon dont les règles sont appliquées. Toute la logique de validation sera définie à un seul emplacement et utilisée partout. Le code est ainsi très propre, facile à mettre à jour et évolutif. Et cela signifie que vous respecterez entièrement le principe DRY.

Utilisation d'attributs DataType

Ouvrez le fichier `Movie.cs` et examinez la classe `Movie`. L'espace de noms `System.ComponentModel.DataAnnotations` fournit des attributs de mise en forme en plus de l'ensemble intégré d'attributs de validation. Nous avons déjà appliqué une valeur

d'énumération `DataType` aux champs de date de sortie et de prix. Le code suivant illustre les propriétés `ReleaseDate` et `Price` avec l'attribut `DataType` approprié.

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

Les attributs `DataType` fournissent uniquement des indices permettant au moteur de vue de mettre en forme les données (et fournissent des éléments/attributs tels que `<a>` pour les URL et `` pour l'e-mail). Vous pouvez utiliser l'attribut `RegularExpression` pour valider le format des données. L'attribut `DataType` sert à spécifier un type de données qui est plus spécifique que le type intrinsèque de la base de données ; il ne s'agit pas d'un attribut de validation. Dans le cas présent, nous voulons uniquement effectuer le suivi de la date, et non de l'heure. L'énumération `DataType` fournit de nombreux types de données, telles que `Date`, `Time`, `PhoneNumber`, `Currency` ou `EmailAddress`. L'attribut `DataType` peut également permettre à l'application de fournir automatiquement des fonctionnalités propres au type. Par exemple, vous pouvez créer un lien `mailto:` pour `DataType.EmailAddress`, et vous pouvez fournir un sélecteur de date pour `DataType.Date` dans les navigateurs qui prennent en charge HTML5. Les attributs `DataType` émettent des attributs HTML 5 `data-` compréhensibles par les navigateurs HTML 5. Les attributs `DataType` ne fournissent **aucune** validation.

`DataType.Date` ne spécifie pas le format de la date qui s'affiche. Par défaut, le champ de données est affiché conformément aux formats par défaut basés sur le `CultureInfo` du serveur.

L'attribut `DisplayFormat` est utilisé pour spécifier explicitement le format de date :

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

Le paramètre `ApplyFormatInEditMode` indique que la mise en forme doit également être appliquée quand la valeur est affichée dans une zone de texte à des fins de modification. (Ceci peut ne pas être souhaitable pour certains champs ; par exemple, pour les valeurs

monétaires, vous ne souhaitez sans doute pas que le symbole monétaire figure dans la zone de texte.)

Vous pouvez utiliser l'attribut `DisplayFormat` par lui-même, mais il est généralement préférable d'utiliser l'attribut `DataType`. L'attribut `DataType` donne la sémantique des données, plutôt que de décrire comment effectuer le rendu sur un écran, et il offre les avantages suivants dont vous ne bénéficiez pas avec `DisplayFormat` :

- Le navigateur peut activer des fonctionnalités HTML5 (par exemple pour afficher un contrôle de calendrier, le symbole monétaire correspondant aux paramètres régionaux, des liens de messagerie, etc.).
- Par défaut, le navigateur affiche les données à l'aide du format correspondant à vos paramètres régionaux.
- L'attribut `DataType` peut permettre à MVC de choisir le modèle de champ correct pour afficher les données (`DisplayFormat`, utilisé par lui-même, utilise le modèle de chaîne).

Notes

- La validation jQuery ne fonctionne pas avec l'attribut `Range` et `DateTime`. Par exemple, le code suivant affiche toujours une erreur de validation côté client, même quand la date se trouve dans la plage spécifiée :
- ```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```
- Vous devez désactiver la validation de date jQuery pour utiliser l'attribut `Range` avec `DateTime`. Il n'est généralement pas recommandé de compiler des dates dures dans vos modèles. Par conséquent, l'utilisation de l'attribut `Range` et de `DateTime` est déconseillée.
- Le code suivant illustre la combinaison d'attributs sur une seule ligne :

Vous devez désactiver la validation de date jQuery pour utiliser l'attribut `Range` avec `DateTime`. Il n'est généralement pas recommandé de compiler des dates dures dans vos modèles. Par conséquent, l'utilisation de l'attribut `Range` et de `DateTime` est déconseillée.

Le code suivant illustre la combinaison d'attributs sur une seule ligne :



```

public class Movie
{
 public int Id { get; set; }

 [StringLength(60, MinimumLength = 3)]
 public string Title { get; set; }

 [Display(Name = "Release Date"), DataType(DataType.Date)]
 public DateTime ReleaseDate { get; set; }

 [RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$"), Required, StringLength(30)]
 public string Genre { get; set; }

 [Range(1, 100), DataType(DataType.Currency)]
 [Column(TypeName = "decimal(18, 2)")]
 public decimal Price { get; set; }

 [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
 public string Rating { get; set; }
}

```

Dans la partie suivante de la série, nous examinons l'application et nous apportons des améliorations aux méthodes `Details` et `Delete` générées automatiquement.

## Examiner les méthodes Details et Delete d'une application ASP.NET Core

Ouvrez le contrôleur vidéo et examinez la méthode `Details`:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
 if (id == null)
 {
 return NotFound();
 }

 var movie = await _context.Movie
 .FirstOrDefaultAsync(m => m.Id == id);
 if (movie == null)
 {
 return NotFound();
 }

 return View(movie);
}
```

Le moteur de génération de modèles automatique MVC qui a créé cette méthode d'action ajoute un commentaire présentant une requête HTTP qui appelle la méthode. Dans le cas présent, il s'agit d'une requête GET avec trois segments d'URL : le contrôleur `Movies`, la méthode `Details` et une valeur `id`. N'oubliez pas que ces segments sont définis dans *Startup.cs*.

```
app.UseMvc(routes =>
{
 routes.MapRoute(
 name: "default",
 template: "{controller=Home}/{action=Index}/{id?}");
});
```

EF facilite la recherche de données à l'aide de la méthode `FirstOrDefaultAsync`. Une fonctionnalité de sécurité importante intégrée à la méthode réside dans le fait que le code vérifie que la méthode de recherche a trouvé un film avant de tenter toute opération que ce soit avec lui. Par exemple, un pirate informatique pourrait induire des erreurs dans le site en modifiant l'URL créée par les liens, en remplaçant `http://localhost:xxxx/Movies/Details/1` par quelque chose comme `http://localhost:xxxx/Movies/Details/12345` (ou une autre valeur qui ne représente pas un film réel). Si vous avez recherché un film null, l'application lève une exception.

Examinez les méthodes `Delete` et `DeleteConfirmed`.

```

// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
 if (id == null)
 {
 return NotFound();
 }

 var movie = await _context.Movie
 .FirstOrDefaultAsync(m => m.Id == id);
 if (movie == null)
 {
 return NotFound();
 }

 return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
 var movie = await _context.Movie.FindAsync(id);
 _context.Movie.Remove(movie);
 await _context.SaveChangesAsync();
 return RedirectToAction(nameof(Index));
}

```

Notez que la méthode `[HTTP GET Delete]` ne supprime pas le film spécifié, mais retourne une vue du film où vous pouvez soumettre (HttpPost) la suppression. L'exécution d'une opération de suppression en réponse à une requête GET (ou encore l'exécution d'une opération de modification, d'une opération de création ou de toute autre opération qui modifie des données) génère une faille de sécurité.

La méthode `[HttpPost]` qui supprime les données est nommée `DeleteConfirmed` pour donner à la méthode HTTP POST une signature ou un nom unique. Les signatures des deux méthodes sont illustrées ci-dessous :

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
```

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

Le Common Language Runtime (CLR) nécessite des méthodes surchargées pour avoir une signature à paramètre unique (même nom de méthode, mais liste de paramètres différentes). Toutefois, vous avez ici besoin de deux méthodes `Delete` (une pour GET et une pour POST) ayant toutes les deux la même signature de paramètre. (Elles doivent toutes les deux accepter un entier unique comme paramètre.)

Il existe deux approches pour ce problème. L'une consiste à attribuer aux méthodes des noms différents. C'est ce qu'a fait le mécanisme de génération de modèles automatique dans l'exemple précédent. Toutefois, elle présente un petit problème : ASP.NET mappe des segments d'une URL à des méthodes d'action par nom. Si vous renommez une méthode, il est probable que le routage ne pourra pas trouver cette méthode. La solution consiste à faire ce que vous voyez dans l'exemple, c'est-à-dire à ajouter l'attribut `ActionName("Delete")` à la méthode `DeleteConfirmed`. Cet attribut effectue un mappage du système de routage afin qu'une URL qui inclut `/Delete/` pour une requête POST trouve la méthode `DeleteConfirmed`.

Pour contourner le problème des méthodes qui ont des noms et des signatures identiques, vous pouvez également modifier artificiellement la signature de la méthode POST pour inclure un paramètre supplémentaire (inutilisé). C'est ce que nous avons fait dans une publication précédente quand nous avons ajouté le paramètre `notUsed`. Vous pouvez faire de même ici pour la méthode `[HttpPost] Delete` :

```
// POST: Movies/Delete/6
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```