

MANUEL D'EXERCICES

ASP.NET MVC

Table des matières

Exercice 1.1 : Créer une application MVC simple.....	3
Exercice 2.1 : Créer le projet de l'étude de cas.....	12
Exercice 2.2 : Pages de disposition	20
Exercice 2.3 : Simplifier la vue et ajouter des styles	26
Exercice 3.1 : Ajouter Entity Framework à l'étude de cas	36
Exercice 3.2 : Programmer le référentiel	40
Exercice 3.3 : Ajouter le niveau de service	46
Exercice 3.4 : Ajouter le conteneur DI Ninject	51
Exercice 4.1 : Mettre en cache la sortie d'une action.....	57
Exercice 4.2 : Personnaliser des routes	60
Exercice facultatif 4.3 : Structurer une application avec des zones.....	66
Exercice 5.1 : Remplacer des éléments HTML par des aides HTML	70
Exercice 5.2 : Développer une aide HTML LabelTextBox	75
Exercice 5.3 : Utiliser les modèles d'édition et d'affichage	81
Exercice 5.4 : Ajouter la validation à l'étude de cas	88
Exercice 6.1 : Afficher un dialogue avec jQuery et Ajax.....	94
Exercice 6.2 : Adapter Todo pour les appareils mobiles.....	102
Exercice facultatif 6.3 : Internationalisation de l'application Todo	109
Exercice 7.1 : Ajouter des méthodes d'authentification et d'autorisation	113
Exercice 7.2 : Créer un contrôleur Web API.....	119
Exercice 8.1 : Déployer l'application.....	126

Exercice 1.1 : Créer une application MVC simple

Objectifs

Dans cet exercice, vous allez construire et explorer votre première application MVC. Vous allez créer une application MVC avec le modèle vide et ajouter un contrôleur et une vue. Vous allez également ajouter un lien à la vue et une deuxième vue pour afficher l'heure actuelle avec Razor.



Créer une application MVC avec le modèle de projet Empty

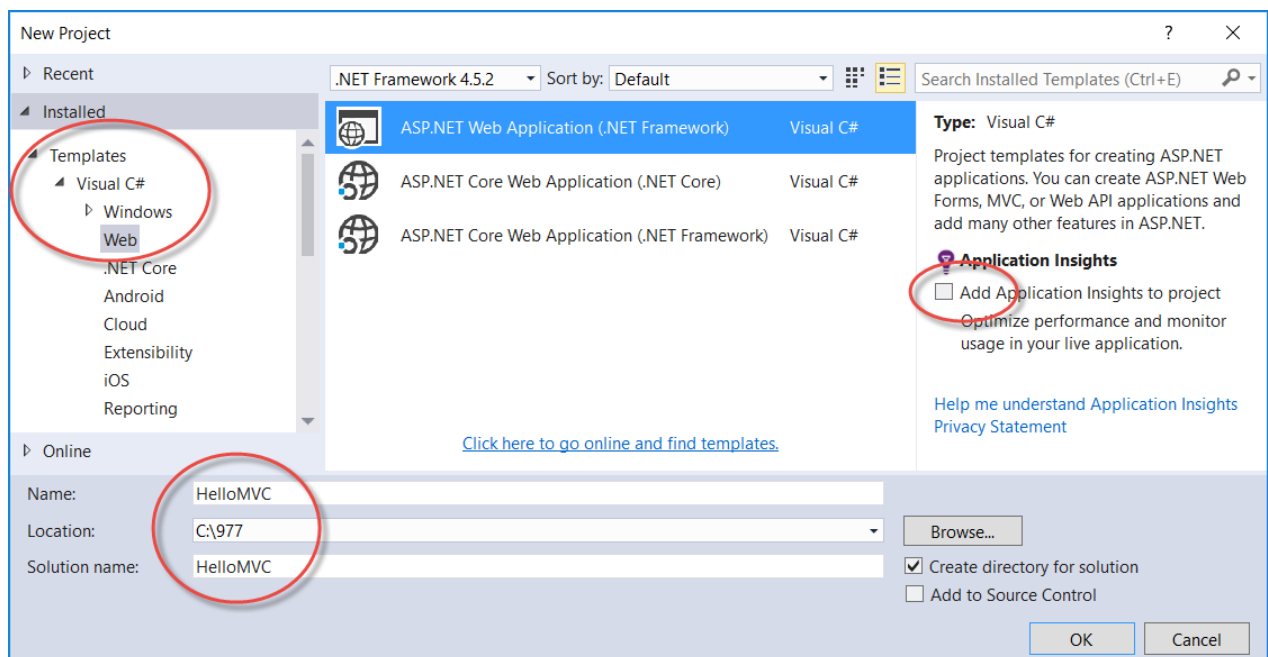
1. ☐ Démarrez **Visual Studio** en cliquant sur l'icône Visual Studio dans la barre des tâches.
2. ☐ Sélectionnez **New Project** dans la page de démarrage affichée lors de l'ouverture de Visual Studio.
3. ☐ Sélectionnez **Web** sous C# dans le volet de gauche et **ASP.NET Web Application** dans le volet central. Notez que .NET Framework 4.5.2 est sélectionné par défaut en haut du dialogue.



N'oubliez pas de sélectionner Web en choisissant C#.

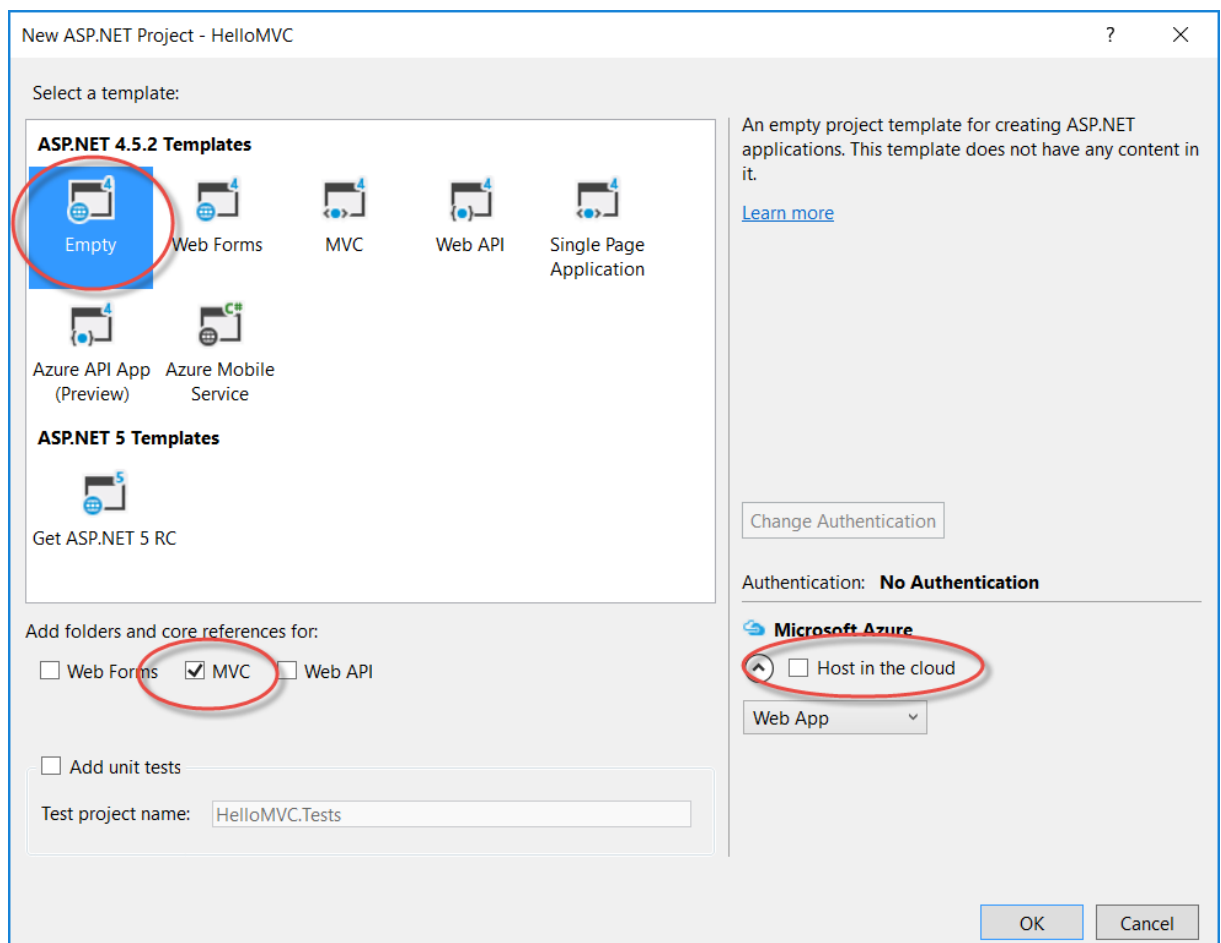
Exercice 1.1 : Créer une application MVC simple (suite)

4. □ Toujours dans le dialogue New Project :
- Saisissez **HelloMVC** comme nom d'application et vérifiez que le dossier C:\977 est sélectionné dans Location (emplacement)
 - Vérifiez que la case sous **Application Insights** est désélectionnée
 - Cliquez sur **OK**



Exercice 1.1 : Créer une application MVC simple (suite)

- 5.□ Dans le dialogue New ASP.NET MVC Project, sélectionnez le modèle **Empty** dans la section **ASP.NET 4.5.2 Templates**. Cochez la case **MVC** et vérifiez que la case **Host in the cloud** est désélectionnée. Ne cochez aucune autre case. Cliquez sur **OK**.



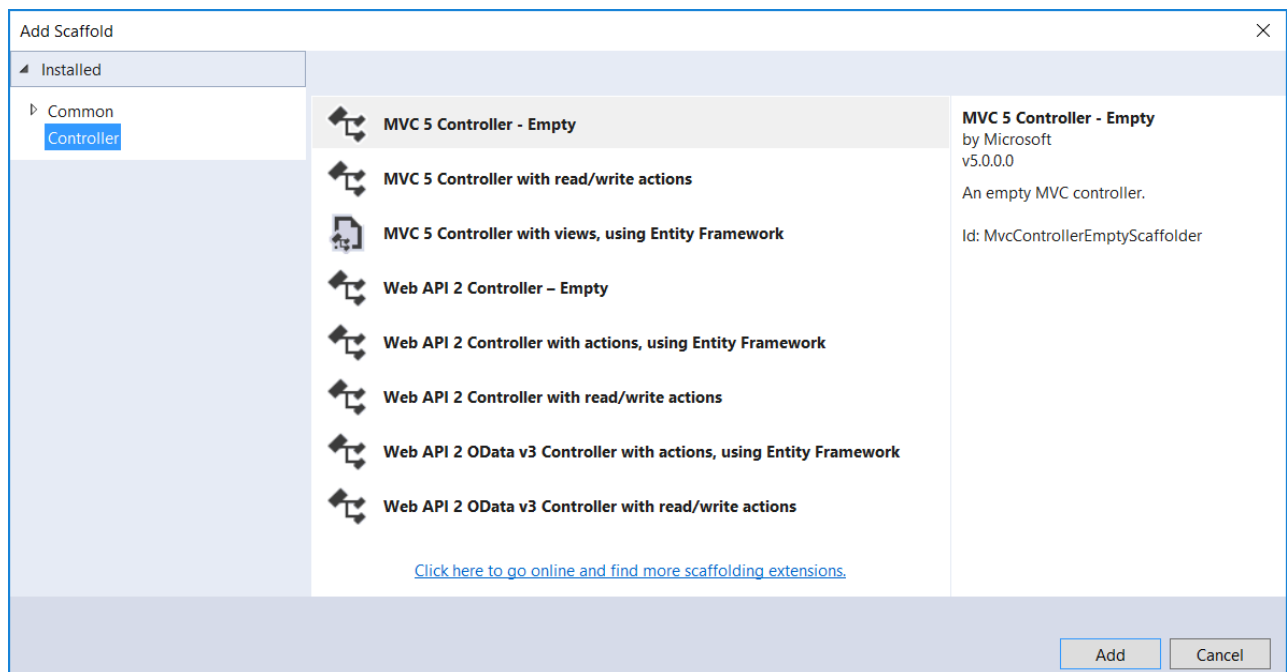
***i** Le modèle Empty crée une application qui ne fonctionne pas telle quelle. Si vous essayez de l'exécuter, vous obtiendrez une page d'erreur. Nous allons corriger cela en ajoutant un contrôleur et une vue au projet. En cochant la case MVC, on indique à Visual Studio de créer les dossiers attendus par le framework MVC.*

Exercice 1.1 : Créer une application MVC simple (suite)

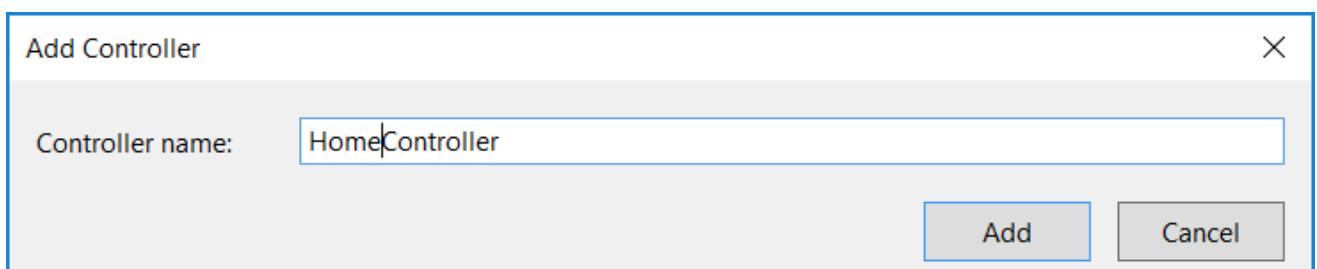


Afficher une page simple

6. ☐ Dans la fenêtre Solution Explorer, cliquez droit sur le dossier **Controllers** et sélectionnez **Add | Controller**.
7. ☐ Vérifiez que le modèle **MVC 5 Controller - Empty** est sélectionné. Cliquez sur **Add**.



8. ☐ Tapez **Home** pour nommer le contrôleur HomeController. Cliquez sur **Add**.



Exercice 1.1 : Créer une application MVC simple (suite)



Appelez le contrôleur HomeController, pas Home. Si vous vous trompez, vous pouvez supprimer le fichier du dossier Controllers et le recréer.



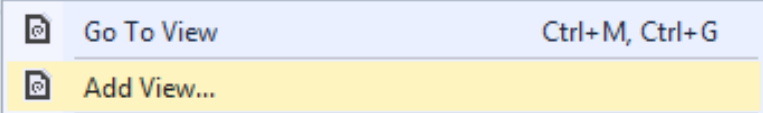
Une nouvelle classe avec une seule méthode appelée Index est ajoutée au projet.

- 9.□ Cliquez droit n'importe où *dans* la méthode Index et sélectionnez **Add View**.



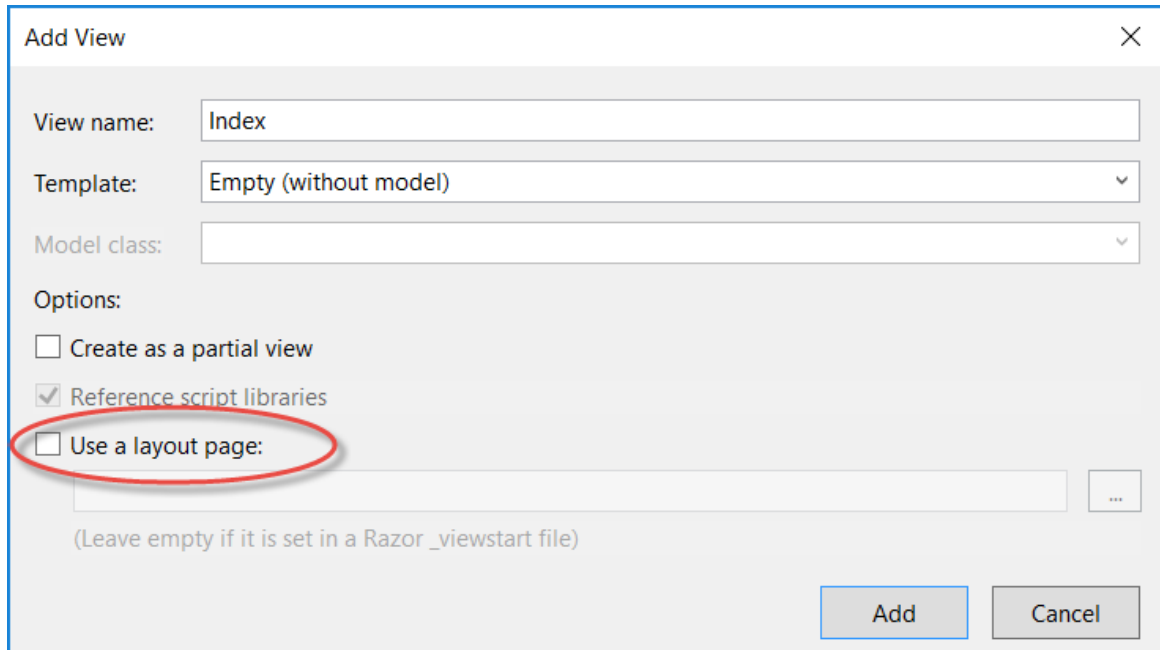
Voici la méthode Index :

```
public class HomeController : Controller
{
    //
    // GET: /Home/
    public ActionResult Index()
    {
        return View();
    }
}
```



Exercice 1.1 : Créer une application MVC simple (suite)

- 10.□ Désélectionnez la case **Use a layout** en bas du dialogue Add View. Gardez les valeurs par défaut pour toutes les autres options. Notez que la vue sera appelée Index. Cliquez sur **Add**.



***i** Un nouveau fichier cshtml est ajouté au projet, dans le dossier Views\Home.*

Notez que, en dehors des trois premières lignes, ce fichier est un fichier HTML standard, avec des sections head et body. Vous pouvez même retirer les trois premières lignes, car elles ne sont pas nécessaires dans cette version de l'application.

- 11.□ Tapez le texte que vous voulez dans le fichier Index.cshtml, entre les balises <div> ouvrante et fermante.

***i** Il s'agit d'un fichier HTML et, si vous êtes à l'aise avec le HTML, vous pouvez y placer des éléments HTML.*

Exercice 1.1 : Créer une application MVC simple (suite)

- 12.□ Exécutez l'application en tapant sur la touche <F5> ou en cliquant sur le bouton **Edge** de la barre d'outils.



Votre page devrait s'afficher dans le navigateur.



Quelle est l'URL de la page affichée dans la barre d'adresse du navigateur ?

- 13.□ Arrêtez l'application.



Pour arrêter l'application, fermez le navigateur et cliquez sur le carré rouge Stop Debugging dans la barre d'outils de Visual Studio.

Vous pouvez vérifier l'état de Visual Studio dans la barre de titre : si vous voyez (Running) ou (Debugging), cela signifie que l'application est démarrée.



Ajouter une deuxième vue afin d'afficher l'heure du serveur

- 14.□ Dans le fichier Index.cshtml, ajoutez un lien vers l'URL /Home/Time.



Votre code HTML va ressembler à ceci :

```
<a href="/Home/Time">Display current time</a>
```



IntelliSense vous aide quand vous tapez le HTML.


Exercice 1.1 : Créer une application MVC simple (suite)

15.□ Dans la classe HomeController, ajoutez une nouvelle méthode d'action. Vous pouvez copier-coller la méthode Index et la modifier, ou suivre les étapes suivantes :


- Placez le curseur après la méthode Index existante, mais avant la fin de la classe. Créez une ligne vide si nécessaire.
- Tapez mvc et appuyez deux fois sur la touche <Tab>.
- Tapez **Time** pour remplacer le nom par défaut de la méthode.

 Vous pouvez explorer et gérer les extraits de code existants avec le menu **Tools / Code Snippets Manager** de Visual Studio.

16.□ Ajoutez une vue appelée **Time** pour la nouvelle méthode d'action, toujours dans page de disposition.


 Vous l'avez déjà fait pour la méthode Index. Regardez les étapes 9 et 10 pour savoir comment faire, si nécessaire.

17.□ Dans la vue Time, ajoutez du texte entre les balises <div> ouvrante et fermante. On peut obtenir l'heure courante dans .NET avec la structure DateTime. Pour indiquer au moteur Razor que vous allez taper du code C# ou VB au milieu d'éléments HTML, il faut utiliser le caractère @.


 Votre code va ressembler à ceci :

The current time is: @DateTime.Now.ToLongTimeString()

Exercice 1.1 : Créer une application MVC simple (suite)

 *IntelliSense vous aide quand vous tapez du code C# dans la vue.*

- 18.□ Cliquez sur index.cshtml dans Solution Explorer pour le sélectionner et exécutez l'application. Cliquez sur le lien dans la page Index. La page Time doit s'afficher avec l'heure courante.

 *L'heure affichée est celle du serveur. Dans une application réelle, l'heure pourrait être différente pour l'utilisateur. Pour afficher l'heure locale de l'utilisateur, on peut utiliser du code JavaScript. Nous verrons comment ajouter du code JavaScript dans un autre chapitre.*

 *Quelle est l'URL de la page affichée dans la barre d'adresse du navigateur ?*



Félicitations ! Vous avez construit et exploré votre première application MVC. Vous avez créé une application MVC avec le modèle vide et ajouté un contrôleur et une vue. Vous avez également ajouté un lien à la vue et une deuxième vue pour afficher l'heure actuelle avec Razor.

Exercice 2.1 : Créer le projet de l'étude de cas

Objectifs

Dans cet exercice, vous allez créer le projet **Todo** et y ajouter une vue afin d'afficher une liste de catégories.



Créer le projet Todo

1. ☐ Ouvrez Visual Studio et créez un nouveau projet.
2. ☐ Sélectionnez **Web** dans votre langage dans la liste Templates, et sélectionnez le modèle **ASP.NET Web Application**. Vérifiez que **.NET Framework 4.5.2** est sélectionné dans la liste déroulante en haut du dialogue et que la case sous **Application Insights** n'est pas cochée.



Dans les étapes suivantes, les noms du projet et de la solution ne sont pas les mêmes.

3. ☐ Appelez le projet **Todo.Web** et la solution **Todo**. Dans Location, sélectionnez le dossier C:\977. Laissez la case **Create directory for solution** cochée. Cliquez sur **OK**.

Name:

Location:

Solution name:

☒ Create directory for solution

☐ Add to Source Control

4. ☐ Sélectionnez le modèle **Empty** dans la liste Project Template. Cochez la case **MVC**. Laissez toutes les autres cases non cochées. Cliquez sur **OK**.

Exercice 2.1 :
Créer le projet de l'étude de cas
(suite)

5. ☐ Ajoutez un contrôleur vide appelé **Category** au projet.



Vous l'avez déjà fait dans le premier exercice.

- a. Cliquez droit sur le dossier **Controllers** et sélectionnez **Add / Controller**. Gardez l'option par défaut **MVC 5 Controller - Empty** sélectionnez. Cliquez sur **Add**.
- b. Appelez le contrôleur **CategoryController**. Cliquez sur **Add**.

6. ☐ Appelez **CategoryList** la méthode Index de la classe **CategoryController**.

7. ☐ Ajoutez une vue appelée **CategoryList** sans page de disposition pour la méthode d'action **CategoryList** du contrôleur **Category**. Si la case **Use a layout page** est cochée, désélectionnez-la.



Vous avez également déjà fait ceci dans le premier exercice.

- a. Cliquez n'importe où dans la méthode **CategoryList** du contrôleur **Category** et sélectionnez **Add view**.
- b. Gardez le nom de vue par défaut **CategoryList**, et décochez la case **Use a layout**. Gardez la valeur par défaut de la liste **Template à Empty (without model)**, et laissez la case **Create as a partial view** non cochée.



Vous utiliserez des pages de disposition dans un autre exercice.

Exercice 2.1 : Créer le projet de l'étude de cas (suite)

- 8. ☐ Ajoutez du texte entre les balises <div> ouvrante et fermante de la vue CategoryList du contrôleur Category.
- 9. ☐ Ajoutez au projet un autre contrôleur vide appelé **Home**.



Reportez-vous aux étapes précédentes si nécessaire.

- 10. ☐ Ajoutez une vue appelée **Index** à la méthode d'action Index du contrôleur Home. **Désélectionnez** la case **Use a layout page** si elle est cochée.



Reportez-vous aux étapes précédentes si nécessaire.

- 11. ☐ Dans la vue Index que vous venez d'ajouter, utilisez la méthode d'aide ActionLink pour ajouter un lien vers la méthode d'action CategoryList du contrôleur Category, en utilisant. Appelez ce lien **Categories**.

Exercice 2.1 : Créer le projet de l'étude de cas (suite)



Votre code va ressembler à ceci :

Code C#/HTML de la vue Index, avec le nouveau code en gras :

```
@{
    Layout = null;
}
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @Html.ActionLink("Categories", "CategoryList", "Category")
    </div>
</body>
</html>
```



Visual Studio affiche la vue sélectionnée au démarrage de l'application. Pour éviter que la mauvaise vue soit affichée lors du débogage, nous allons définir la vue de démarrage dans les propriétés du projet.

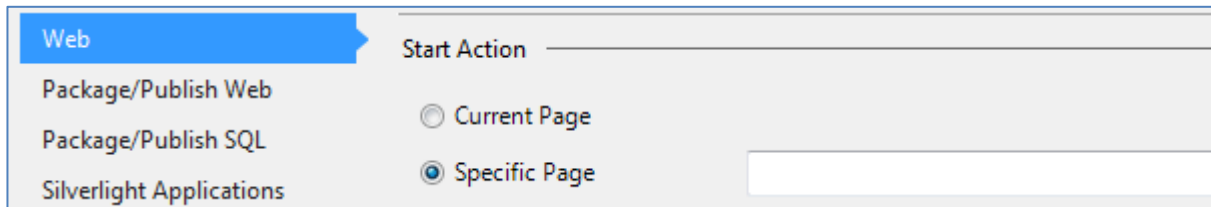
- 12.□ Cliquez droit sur le projet Todo.Web dans Solution Explorer et sélectionnez **Properties**.



Cliquez droit sur le projet, pas sur la solution.

Exercice 2.1 : Créer le projet de l'étude de cas (suite)

- 13.□ Cliquez sur l'onglet **Web** et sélectionnez **Specific Page** dans Start Action. Laissez la zone de texte vide.



- 14.□ Exécutez l'application et vérifiez que vous pouvez naviguer vers la page Listing.



Afficher la liste des catégories

***i** Une fois finalisée, la liste des catégories sera générée à partir d'une requête sur la base de données. Nous allons commencer par coder une liste de catégories en dur pour tester l'application.*

- 15.□ Fermez le navigateur et arrêtez l'application.
- 16.□ Dans le dossier Models du projet, ajoutez une nouvelle classe que vous appellerez **CategoryModel**.
- 17.□ Dans la classe CategoryModel, ajoutez deux propriétés auto-implémentées : **Id** de type int et **Title** de type string.

Exercice 2.1 : Créer le projet de l'étude de cas (suite)



Votre code va ressembler à ceci :

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

- 18.□ Dans la méthode d'action `CategoryList` de la classe `CategoryController`, avant l'instruction `return`, ajoutez le code ci-dessous pour créer une liste d'objets `CategoryModel`. Passez la variable `categories` à la méthode `View`.



Votre code va ressembler à ceci :



Ajoutez une instruction `using` pour `Todo.Web.Models`.

```
public ActionResult CategoryList()
{
    var categories = new List<CategoryModel>
    {
        new CategoryModel { Id = 1, Title = "Shopping" },
        new CategoryModel { Id = 2, Title = "DIY" },
        new CategoryModel { Id = 3, Title = "Children" },
    };
    return View(categories);
}
```

- 19.□ Dans le fichier `CategoryList.cshtml`, entre les balises `<div>` ouvrante et fermante, ajoutez un élément ``.

Exercice 2.1 : Créer le projet de l'étude de cas (suite)

- 20.□ Entre les balises ouvrante et fermante, ajoutez une boucle **foreach** en utilisant la syntaxe Razor. La boucle doit itérer sur la propriété Model. Pour chaque élément de category de la boucle, générez un élément dont le texte est la valeur de la propriété Title de l'élément.



Votre code va ressembler à ceci :

```
<div>
  List
  <ul>
    @foreach (var category in Model)
    {
      <li>@category.Title</li>
    }
  </ul>
</div>
```




N'hésitez pas à utiliser les extraits de code (snippets) de Visual Studio, ils fonctionnent également dans les vues cshtml. Saisissez @foreach et appuyez sur la touche <Tab> : le stub de la boucle foreach est généré automatiquement. Utilisez la touche <Tab> pour passer d'un champ modifiable à l'autre.



Quels sont les types de la propriété Model et de la variable category ?

Exercice 2.1 : Créer le projet de l'étude de cas (suite)

 *Le modèle n'est pas typé, vous pourrez corriger cela dans le bonus. Vous devez être attentif quand vous écrivez la propriété Title, car IntelliSense ne vous aide pas. En cas d'erreur de frappe, une erreur se produira lors de l'exécution.*

- 21.□ Exécutez l'application et allez jusqu'à la page Categories. La liste des catégories doit s'afficher.



Félicitations ! Vous avez créé le projet Todo auquel vous avez ajouté une vue afin d'afficher la liste des catégories.

Exercice 2.2 : Pages de disposition

Objectifs

Dans cet exercice, vous allez créer une vue de disposition et modifier les vues existantes afin d'utiliser la nouvelle disposition.



Factoriser le HTML de vues dans une vue de disposition

1. ☐ Ouvrez la solution du point de départ Ex 2.2-Starting Point.sln et régénérez la solution.

***i** La plupart des exercices ont un point de départ, généralement la solution terminée de l'exercice précédent, avec éventuellement des améliorations. Les points de départ sont dans le dossier C:\977\Exercices. Le titre de chaque dossier de solution renferme le numéro de l'exercice, par exemple Ex 2.2-Starting Point pour l'exercice 2.2.*


*Pour ouvrir une solution, vous pouvez aller au dossier correspondant avec Windows Explorer et double cliquer sur le fichier .sln, ou le rechercher à partir de **FILE | Open Project** dans Visual Studio.*

*Il faut générer chaque solution avant de l'éditer, que ce soit un point de départ ou un exercice terminé, afin de mettre à jour les packages utilisés par la solution. Pour générer une solution, vous pouvez sélectionner **BUILD | Build Solution** dans Visual Studio ou utiliser le raccourci clavier <Ctrl><Shift>. Il en va de même pour les À vous.*


Ce projet a été créé à partir du modèle Empty que vous avez déjà utilisé. Un contrôleur Home et une vue Index y ont été ajoutés.

Exercice 2.2 : Pages de disposition (suite)

- 2.□ Ouvrez la vue Index.cshtml dans le dossier Views\Home.

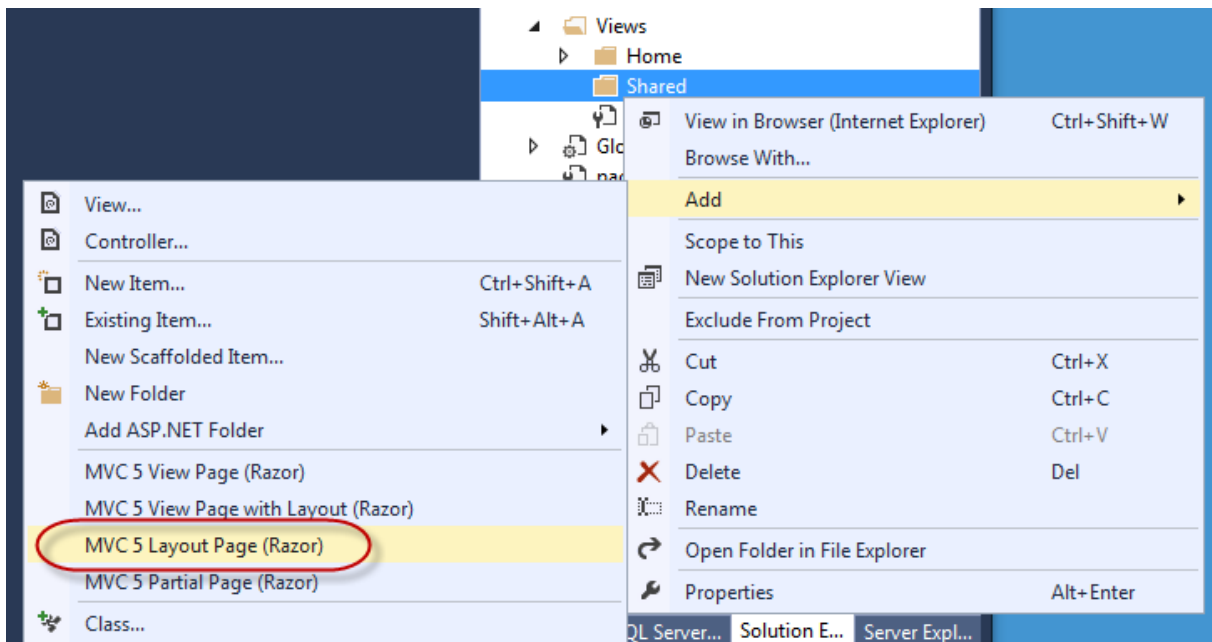
 *Il s'agit d'une simple page avec du contenu.*

- 3.□ Ajoutez un dossier appelé Shared au dossier Views.

 *Cliquez droit sur le dossier Views dans Solution Explorer et sélectionnez **Add / New Folder**.*


- 4.□ Ajoutez une nouvelle page de disposition appelée _Layout dans le dossier Shared.

 *Cliquez droit sur le dossier Shared et sélectionnez **Add / MVC 5 Layout Page (Razor)**.*




Exercice 2.2 : Pages de disposition (suite)

- 5.□ Dans la page `_Layout`, ajoutez du texte avant l'appel à `RenderBody`, comme par exemple **This is the layout page**.

 La vue `Index` n'utilise pas la page `_Layout`. Vous pourriez la modifier mais il est plus simple de demander à Visual Studio de générer automatiquement une nouvelle vue `Index`.

- 6.□ Cliquez droit sur `Index.cshtml` dans Solution Explorer et sélectionnez **Delete**. Cliquez sur **OK** dans la fenêtre qui s'affiche pour confirmer que vous voulez supprimer le fichier `Index.cshtml`.
- 7.□ Cliquez droit sur le dossier `Home` et sélectionnez **Add | MVC 5 View Page with Layout (Razor)**.
- 8.□ Appelez l'élément **Index** et cliquez sur **OK**.
- 9.□ Dans le dialogue `Select a Layout Page`, vérifiez que `_Layout.cshtml` est sélectionné, puis cliquez sur **OK**.
- 10.□ En bas de la vue `Index.cshtml`, ouvrez un élément `<div>` et saisissez du texte entre les balises `div` ouvrante et fermante, comme par exemple **This is the view**.
- 11.□ Exécutez votre projet. Cliquez droit n'importe où sur la page dans le navigateur et sélectionnez **View Source**.

 Le code source devrait être semblable à la vue avant que vous ne déplaciez son HTML dans la vue `_Layout`.

Exercice 2.2 : Pages de disposition (suite)



Ajouter une section dans la vue de disposition

- 12.□ Arrêtez l'application. Dans la vue _Layout, juste avant la balise fermante `</body>`, ajoutez un appel de méthode Razor pour générer une section facultative appelée footer.



Votre code Razor va ressembler à ceci :

```
@RenderSection("footer", false)
</body>
```

- 13.□ Dans la vue Index, définissez une section appelée footer et ajoutez-y du texte.



Votre code va ressembler à ceci :

```
@section footer
{
    <p>This is the footer</p>
}
```



Vous pouvez placer la section n'importe où dans la page.

- 14.□ Exécutez votre projet et vérifiez qu'il fonctionne comme prévu.

Exercice 2.2 : Pages de disposition (suite)



Déplacer la directive @Layout dans une vue _ViewStart

- 15.□ Ajoutez une vue partielle appelée _ViewStart dans le dossier Views.



Vous devez créer une vue partielle que vous placerez directement dans le dossier Views, pas dans un sous-dossier.



Nous étudierons les vues partielles dans la prochaine section de ce chapitre. Lorsque vous créez une vue partielle, celle-ci est vide. Vous auriez également pu créer une page view et supprimer tout son contenu.



*Cliquez droit sur le dossier Views et sélectionnez **Add / MVC 5 Partial Page (Razor)**.*

- 16.□ Coupez le bloc qui définit la propriété layout dans la vue Index, et collez-le dans la vue _ViewStart.



Code de la vue _ViewStart.cshtml :

```
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
}
```

- 17.□ Exécutez votre projet. Il devrait fonctionner comme précédemment.

Exercice 2.2 : Pages de disposition (suite)



Félicitations ! Vous avez créé une vue de disposition et modifié les vues existantes afin d'utiliser la nouvelle disposition.

Exercice 2.3 : Simplifier la vue et ajouter des styles

Objectifs

Dans cet exercice, vous allez utiliser une vue de disposition pour les vues de l'application. Celle-ci ajoutera un en-tête et un menu à toutes les vues. Vous allez également mieux organiser le code Razor avec une aide de vue.



Ajouter des styles pour améliorer l'apparence de l'application

1. ☐ Ouvrez la solution Ex 2.3-Starting Point.sln du point de départ de l'exercice et régénérez la solution.

***i** Il s'agit de la solution terminée de l'exercice 2.1, avec quelques modifications :*


- Deux nouvelles classes appelées *TodoItemModel* et *TodoListModel* ont été ajoutées au dossier *Models*.
- Les vues ont été modifiées afin d'utiliser la vue *_Layout*, comme vous l'avez fait dans l'exercice précédent.
- La vue *_Layout* a été modifiée.
- Des fichiers CSS ont été ajoutés au dossier *Content*.
- Des fichiers JavaScript ont été ajoutés au dossier *Scripts*.

2. ☐ Ouvrez la vue *Index.cshtml* dans le dossier *Views\Home*.

***?** Quelle vue de disposition utilise-t-elle ?*

Exercice 2.3 : Simplifier la vue et ajouter des styles (suite)

- 3.□ Examinez les modifications faites et démarrez l'application.

 *L'en-tête de page est différent maintenant. Il est défini dans la vue _Layout et utilise des styles définis dans le fichier Site.css situé dans le dossier Content du projet.*

- 4.□ Arrêtez l'application. Dans la vue Index qui se trouve dans le dossier Views\Home que vous avez ouvert à l'étape 2, coupez la ligne qui définit le lien d'action.

- 5.□ Ouvrez le fichier _Layout.cshtml dans le dossier Views\Shared. Cherchez la balise vers le milieu du fichier. Ajoutez un élément entre les balises ouvrante et fermante .

- 6.□ Collez la ligne qui définit le lien d'action entre les balises ouvrante et fermante.



Code HTML des deux étapes précédentes :

```
<ul class="nav navbar-nav navbar-right">  
  <li>@Html.ActionLink("Categories",  
    "CategoryList", "Category")</li>  
</ul>
```

- 7.□ Dupliquez la ligne que vous venez de coller (y compris les éléments) et modifiez la première ligne afin que le texte du lien soit Home, la méthode d'action Index et le contrôleur Home.



Pour dupliquer une ligne, placez le curseur sur la ligne à copier et appuyez sur <Ctrl><C>, puis sur <Ctrl><V> pour la coller.

Exercice 2.3 : Simplifier la vue et ajouter des styles (suite)



Votre nouveau code va ressembler à ceci :

```
<li>@Html.ActionLink("Home", "Index", "Home")</li>
```

- 8.□ Ouvrez la vue CategoryList.cshtml dans le dossier Views\Category. Ajoutez un attribut class dont la valeur est list-group à la balise , et un attribut class de valeur list-group-item à la balise .



Votre code va ressembler à ceci :

```
<ul class="list-group">
    @foreach (var category in Model)
    {
        <li class="list-group-item">
            @category.Title
        </li>
    }
</ul>
```

***i** Les classes sont définies dans Bootstrap, un package qui facilite la conception des interfaces utilisateur. Bootstrap fait partie des modèles de Visual Studio 2015 et peut être ajouté en tant que package NuGet dans les versions précédentes.*

Notez qu'IntelliSense vous aide quand vous tapez les noms de classe.

- 9.□ Exécutez l'application. Il y a maintenant un menu à droite de l'entête et la liste de la page CategoryList.cshtml est stylée.

Exercice 2.3 : Simplifier la vue et ajouter des styles (suite)



Modifier la liste des catégories afin d'afficher des icônes correspondant au type de catégorie sélectionné



Pour afficher les icônes, vous allez utiliser l'outil Font Awesome, qui convertit les icônes en polices (<http://fontawesome.io>). Cette bibliothèque a déjà été ajoutée au projet, dans les dossiers Content et fonts. Pour utiliser une icône de Font Awesome, vous devez ajouter la classe fa à un élément, ainsi qu'une classe correspondant à l'icône que vous avez choisie. Exemple : `class="fa fa-shoppingcart"`



Pour les deux prochaines étapes, vous pouvez saisir directement le code ou bien copier le code du fichier `CategoryClassName.txt` dans le dossier Copy-paste de la solution.

- 10.□ Toujours dans la vue `CategoryList.cshtml`, juste après l'accolade ouvrante de la boucle `foreach`, ajoutez du code pour définir une variable string dont la valeur dépend de celle de `category.id` :
- Si `category.id` est 1, la valeur de la variable string est **"fafa-shopping-cart"**
 - Si `category.id` est 2, la valeur de la variable string est **"fafa-wrench"**
 - Si `category.id` est 3, la valeur de la variable string est **"fafa-child"**
- 11.□ Modifiez le contenu de l'élément `li` en ajoutant un élément `span` avant `@category.Title`. En utilisant une syntaxe Razor, ajoutez à l'élément `span` un attribut `class` dont la valeur correspond à la variable string que vous venez de définir.

Exercice 2.3 : Simplifier la vue et ajouter des styles (suite)



Votre code va ressembler à ceci : (le nouveau code est en gras)

```
@foreach (var category in Model)
{
    var className = string.Empty;
    switch (category.Id)
    {
        case 1:
            className = "fa fa-shoppingcart";
            break;
        case 2:
            className = "fa fa-wrench";
            break;
        case 3:
            className = "fa fa-child";
            break;
    }
    <li class="list-group-item"><span
class="@className"/>@category.Title</li>
}
```

12. ☐ Démarrez l'application, ouvrez la page Categories et vérifiez que les icônes s'affichent comme prévu devant la catégorie.



Ajouter une vue pour afficher le contenu des catégories

13. ☐ Ouvrez le fichier TodoListController.txt dans le dossier Copy-paste de la solution. Sélectionnez tout le texte et copiez-le dans le presse-papiers.
14. ☐ Ajoutez au projet un nouveau contrôleur appelé TodoListController.

Exercice 2.3 : Simplifier la vue et ajouter des styles (suite)



*Vous l'avez déjà fait : Cliquez droit sur le dossier **Controllers** et sélectionnez **Add / Controller**.*

- 15.□ Renommez la méthode d'action Index en ItemList et ajoutez-y un paramètre appelé id.
- 16.□ Dans la méthode ItemList, collez le code que vous aviez copié à l'étape 13, avant l'instruction return. Utilisez la fonctionnalité de résolution de Visual Studio pour ajouter l'instruction using manquante.



Pour ajouter une instruction using manquante, cliquez sur le nom de la classe soulignée en rouge, puis sur l'ampoule qui s'affiche et sélectionnez la bonne option dans le menu contextuel. Vous pouvez aussi utiliser le raccourci clavier <Alt><Maj><F10>.

- 17.□ Passez la variable List en paramètre à la méthode View.

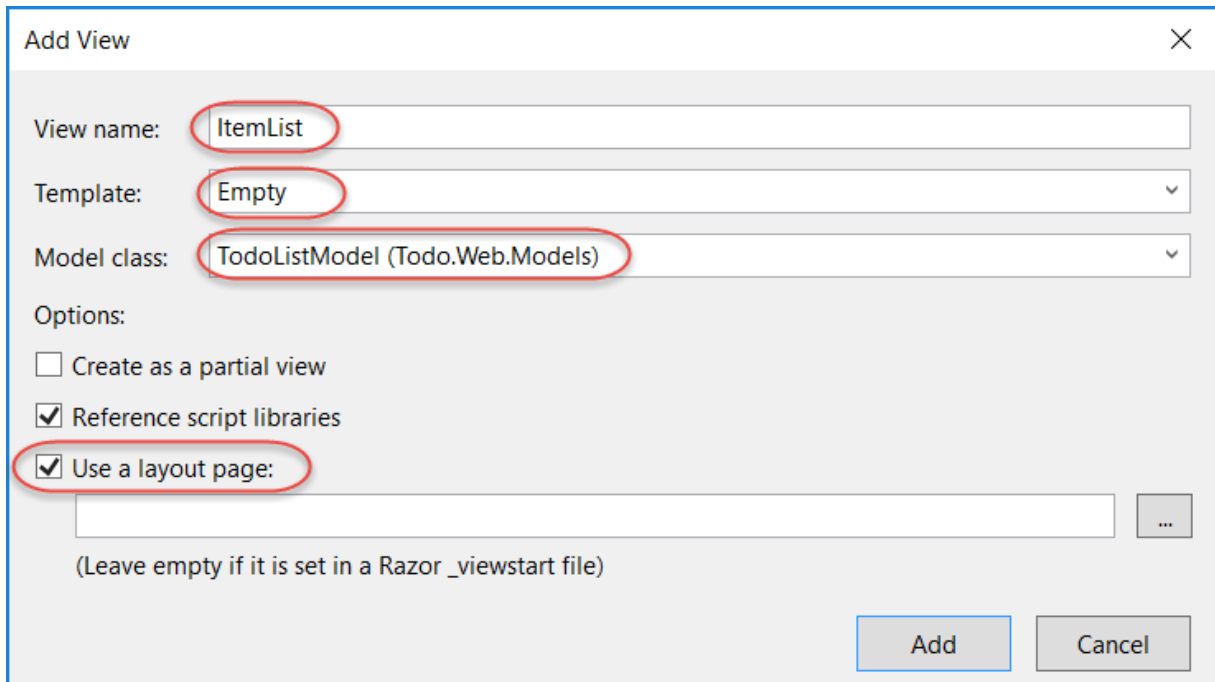


Votre code va ressembler à ceci :


```
public ActionResult ItemList(int id)
{
    var items = new List<TodoItemModel>
    {
        // Initialization code omitted
    }
    .Where(li => li.CategoryId == id).ToList();
    var list = new TodoListModel(items);
    return View(list);
}
```

Exercice 2.3 : Simplifier la vue et ajouter des styles (suite)

18. ☐ Ajoutez une vue appelée `ItemList` à la méthode d'action. Sélectionnez le modèle **Empty** et **TodoListModel** dans la liste déroulante **Model class**. Laissez la case **Use a layout page** cochée.




The screenshot shows the 'Add View' dialog box. The 'View name' field is 'ItemList'. The 'Template' dropdown is 'Empty'. The 'Model class' dropdown is 'TodoListModel (Todo.Web.Models)'. Under 'Options', 'Create as a partial view' is unchecked, 'Reference script libraries' is checked, and 'Use a layout page' is checked. There is an empty text field below 'Use a layout page' with a button to its right. At the bottom right are 'Add' and 'Cancel' buttons.

 **Sélectionnez le modèle Empty, pas Empty (without model). Dans la liste déroulante Model class, sélectionnez TodoListModel, pas TodoItemModel.**

 *Rappelez-vous : Cliquez droit n'importe où dans la méthode d'action et sélectionnez **Add view**.*

Exercice 2.3 : Simplifier la vue et ajouter des styles (suite)

- 19.□ En bas de la vue ItemList, ajoutez un élément `` avec un attribut `class` de valeur `list-group`. Dans l'élément ``, ajoutez une boucle `foreach` afin d'afficher la propriété `Title` de chaque item de la liste `Model.TODOItems`. Placez chacun dans un élément `` dont l'attribut `class` vaut `list-group-item`, comme pour la vue `List`.

 Vous pouvez taper le code vous-même, ou le copier depuis le fichier `ItemList.txt` dans le dossier `Copy-paste`.




Votre code va ressembler à ceci :

```
<ul class="list-group">
  @foreach (var item in Model.TODOItems)
  {
    <li class="list-group-item">@item.Title</li>
  }
</ul>
```



Changer les éléments de la vue `CategoryList` en liens vers des listes d'éléments

 Quand l'utilisateur clique sur un élément de la vue `CategoryList`, il devrait obtenir la liste des éléments du listing sélectionné. Changer les éléments `` en `<div>` et les `` en `<a>` indique à Bootstrap d'afficher chaque élément sous la forme d'un lien. Il faut aussi passer la valeur du `id` de l'élément sélectionné en tant que paramètre du lien.

- 20.□ De retour dans la vue `CategoryList`, changez la balise ouvrante `` en `<div>`. Notez que Visual Studio change la balise fermante pour vous.

Exercice 2.3 : Simplifier la vue et ajouter des styles (suite)


- 21.□ Dans la boucle foreach, changez la balise ouvrante en <a>. Notez à nouveau que Visual Studio change la balise fermante pour vous.
- 22.□ Dans la balise ouvrante <a>, ajoutez un attribut href dont la valeur est **/ToDoList/ItemList/@category.Id**.



Le code des trois dernières étapes va ressembler à ceci :

```
<div class="list-group">
  @foreach (var category in Model)
  {
    var className = string.Empty;
    switch (category.Id)
    {
      case 1:
        className = "fa fa-shopping-cart";
        break;
      case 2:
        className = "fa fa-wrench";
        break;
      case 3:
        className = "fa fa-child";
        break;
    }
    <a href="/ToDoList/ItemList/@category.Id"
      class="list-group-item"><span class="@className" />
      @category.Title</a>
  }
</div>
```

Exercice 2.3 : Simplifier la vue et ajouter des styles (suite)

 *Coder directement un lien avec un élément a n'est pas une bonne pratique en ASP.NET MVC, car cela ne tire pas parti du routage. Il faut normalement utiliser une méthode `ActionLink`, mais celle-ci ne peut pas inclure de balises HTML intégrées. Vous corrigerez cela dans un exercice ultérieur en créant votre propre aide HTML.*

- 23.□ Exécutez votre application. Allez dans la liste des catégories et cliquez sur l'une d'entre elles. Vous devriez voir la liste des éléments qu'elle contient.



Félicitations ! Vous avez utilisé une vue de disposition pour les vues de l'application. Celle-ci a ajouté un en-tête et un menu à toutes les vues. Vous avez également mieux organisé le code Razor avec une aide de vue.

Exercice 3.1 : Ajouter Entity Framework à l'étude de cas

Objectifs

Dans cet exercice, vous allez ajouter un modèle de données Entity Framework à un projet référentiel.



Ajouter un modèle de données entity pour la base de données SQL Server du projet


1. ☐ Ouvrez la solution point départ Ex 3.1-Starting Point.sln de l'exercice et régénérez-la.



Il s'agit de la solution terminée de l'exercice 2.3, à laquelle a été ajoutée une nouvelle bibliothèque de classes vide.

2. ☐ Cliquez droit sur le projet Todo.Repository et sélectionnez **Add | New Item**.
3. ☐ Dans le dialogue Add New Item, sélectionnez **Data** dans le volet de gauche, et **ADO.NET Entity Data Model** dans le volet central. Appellez le modèle TodoModel et cliquez sur **Add**.
4. ☐ Dans le dialogue de l'assistant qui est affiché, sélectionnez l'option **Code First from database** et cliquez sur **Next**.
5. ☐ Cliquez sur **New Connection** pour ouvrir le dialogue Connection Properties.
6. ☐ Dans le dialogue Choose Data Source qui s'affiche, sélectionnez **Microsoft SQL Server** et cliquez sur **Continue**.
7. ☐ Vérifiez que la source de données sélectionnée est **Microsoft SQL Server (SqlClient)**.


Exercice 3.1 : Ajouter Entity Framework à l'étude de cas (suite)

 Vous pouvez cliquer sur le bouton *Change* pour sélectionner une autre source de données.


8. ☐ Dans le dialogue *Connection Properties*, tapez le nom de serveur **.\sqlexpress** et laissez l'option par défaut **Use Windows Authentication** sélectionnée. Sélectionnez **ToDoDatabase** dans la liste des bases de données, puis cliquez sur **OK**.
9. ☐ De retour dans l'assistant *Entity Data Model*, cliquez sur **Next**.
10. ☐ Dans la liste *Database Objects*, développez les nœuds **Tables** et **dbo**, puis sélectionnez les trois tables *Category*, *TodoItem* et *LookAndFeel*. Vérifiez que la case **Pluralize or singularize generated object names** est cochée, puis cliquez sur **Finish**.



Examiner les fichiers générés


 L'assistant *Entity Framework* a généré quatre classes : *Category.cs*, *TodoItem.cs*, *Style.cs* et *ToDoModel.cs*.

11. ☐ Ouvrez les fichiers *Category.cs*, *TodoItem.cs* et *Style.cs* et examinez leur contenu.

 Ces fichiers renferment des classes *entity*, autrement dit qui n'ont que des propriétés. *Entity Framework* a généré ces classes à partir des tables sélectionnées dans la base de données mais vous auriez également pu les créer vous-même ou utiliser des classes déjà définies.

Certaines propriétés, que nous étudierons à la fin de la semaine, sont dites *virtuelles*. Elles définissent les relations existant au sein du modèle.


Exercice 3.1 : Ajouter Entity Framework à l'étude de cas (suite)

 *Quel est l'attribut associé à chaque nom de classe entity ?*



L'attribut Table définit le nom de la table à laquelle cette classe est mappée. Il n'est pas très utile ici car les tables sont par défaut mappées aux classes du même nom mais vous pouvez le modifier si les noms sont différents.

Il existe également un attribut Column que vous pouvez ajouter à chaque propriété afin de définir le nom de la colonne à laquelle est mappée la propriété (colonne et propriété par défaut).

 *Nous étudierons les autres attributs (Required, StringLength...) dans un autre chapitre.*


12.□ Ouvrez le fichier TodoModel.cs et examinez son contenu.

 *Combien de propriétés sont définies dans la classe TodoModel ? De quel type sont ces propriétés ?*




Chaque classe entity compte trois propriétés : Categories, TodoItems et Styles. Il s'agit de la classe DbSet, de type generic.

Exercice 3.1 : Ajouter Entity Framework à l'étude de cas (suite)

 Vous noterez les deux points suivants concernant la classe `TodoModel` :

- Le constructeur de classe appelle la classe de base avec un paramètre string indiquant le nom de la chaîne de connexion utilisée pour accéder à la base de données. La chaîne de connexion `TodoModel` est définie dans le fichier `App.config` du projet `TodoRepository`, dans la section `connectionStrings`. Vous pouvez remplacer cette chaîne dans le fichier `App.config` ou modifier le code du constructeur pour utiliser une autre base de données.
- La méthode `OnModelCreating` renferme du code généré pour mettre en œuvre les contraintes de la base de données dans le modèle Entity Framework. Vous pouvez également modifier ce code si nécessaire.

13.□ Générez votre solution et vérifiez qu'il n'y a pas d'erreur.

 Il n'est pas nécessaire d'exécuter l'application à ce point. Cet exercice est terminé si la génération se fait sans erreur. Vous allez utiliser le code généré dans le prochain exercice.



Félicitations ! Vous avez ajouté un modèle de données Entity Framework à un projet référentiel.

Exercice 3.2 : Programmer le référentiel

Objectifs

Dans cet exercice, vous allez développer une méthode dans la couche référentiel qui récupère les catégories dans la base de données. Vous allez aussi modifier le contrôleur afin d'accéder au référentiel à travers une interface.



Ajouter une classe pour récupérer les catégories

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 3.2-Starting Point.sln et régénérez-la.



Il s'agit de la solution de l'exercice 3.1 terminée avec les modifications suivantes :

- Deux nouveaux dossiers appelés *Repositories* et *Interfaces* ont été ajoutés au projet *Todo.Repository*.
- La chaîne de connexion à la base de données définie dans le fichier *app.config* du projet *Todo.Repository* a été copiée dans le fichier *Web.config* du projet *Todo.Web*.

2. ☐ Allez dans la classe *CategoryRepository* du dossier *Repositories* du projet *Todo.Repository*. Ajoutez-y une méthode publique appelée **List** qui ne prend pas de paramètre et renvoie une *List* de *Category*.
3. ☐ Dans la méthode *List*, créez un bloc *using* pour une variable de type *TodoModel* appelée *db*.



Tapez using et appuyez deux fois sur la touche <Tab>.

Exercice 3.2 : Programmer le référentiel (suite)

4. ☐ Dans le bloc using, renvoyez db.Categories en tant que liste en utilisant la méthode d'extension ToList.



Code C# pour les trois dernières étapes :

```
public List<Category> List()
{
    using (var db = new TodoModel())
    {
        return db.Categories.ToList();
    }
}
```



Tester le nouveau référentiel




Vous n'avez pas encore de couche de service. Vous allez donc directement référencer la couche référentiel depuis l'application Web. Ce n'est pas une bonne pratique, mais vous corrigerez cela dans un prochain exercice.

5. ☐ Cliquez droit sur le projet Todo.Web et sélectionnez **Add | Reference**.
6. ☐ Dans le dialogue **Reference Manager**, sélectionnez **Projects | Solution** dans le volet de gauche. Cochez la case près de **Todo.Repository** dans le volet central et cliquez sur **OK**.
7. ☐ Dans la classe CategoryController du projet Todo.Web, ajoutez un **using** pour Todo.Repository.Repositories.

Exercice 3.2 : Programmer le référentiel (suite)

- 8.□ Ajoutez une déclaration au niveau de la classe pour une variable appelée `_categoryRepository` de type `CategoryRepository`.

 *Le fait de mettre un souligné au début des noms de variables déclarées au niveau de la classe est une convention largement utilisée. Selon celle-ci, les paramètres et les variables des méthodes commencent par une lettre minuscule, et chaque membre public (méthode, propriété, classe...) commence par une lettre majuscule.*

- 9.□ Ajoutez un constructeur à la classe `CategoryController`.



Tapez `ctor` et appuyez deux fois sur `<Tab>`.

- 10.□ Dans le constructeur, initialisez la variable `_categoryRepository` avec un nouvel objet de classe `CategoryRepository`.




Votre code va ressembler à ceci :

```
public CategoryController()
{
    _categoryRepository = new CategoryRepository();
}
```

- 11.□ Toujours dans la classe `CategoryController`, supprimez toute la méthode `CategoryList`.
- 12.□ Copiez le code du fichier `CategoryList.txt` du dossier Copy-paste et collez-le dans la classe `CategoryController`.


Exercice 3.2 : Programmer le référentiel (suite)

 Ce code appelle la méthode `List` de `CategoryRepository` et crée un objet `CategoryModel` à partir d'elle. L'objet résultat est passé en tant que modèle à la vue `CategoryList`.




Votre code va ressembler à ceci :

```
public ActionResult CategoryList()
{
    var listing = _categoryRepository.List();
    var model = listing.Select(lst => new
        CategoryModel
        {
            Id = lst.Id,
            Title = lst.Title
        }).ToList();
    return View(model);
}
```

 Transformer la liste renvoyée par le référentiel en modèle nécessite de définir chaque propriété, ce qui peut devenir fastidieux. Cela représenterait beaucoup de code à écrire avec de nombreuses classes ayant beaucoup de propriétés. Vous automatiserez cette transformation avec `AutoMapper` dans un prochain exercice.

- 13.□ Exécutez l'application. Elle devrait fonctionner comme précédemment, avec la liste de catégories provenant cette fois de la base.

 La liste `TodoItem` est toujours codée en dur. Vous modifierez cela dans le prochain exercice.

Exercice 3.2 : Programmer le référentiel (suite)



Accéder au référentiel à travers une interface au lieu d'une classe



Le contrôleur accède au référentiel avec le nom de la classe. Vous allez minimiser le couplage entre les deux couches en remplaçant cela par une interface. Notez que la classe `CategoryRepository` a déjà été déclarée comme implémentant l'interface `ICategoryRepository`.

- 14.□ Dans la classe `CategoryRepository` du projet `Todo.Repository`, copiez la déclaration de la méthode `List` dans l'interface `ICategoryRepository`, sans le modificateur public ni le corps de la méthode. Ajoutez un point-virgule à la fin de l'instruction.



Votre code va ressembler à ceci :

```
public interface ICategoryRepository
{
    List<Category> List();
}
```



*Vous auriez pu utiliser les fonctionnalités de refactorisation de Visual Studio pour que l'interface soit générée à partir de la définition de la classe. Cliquez droit sur le nom de la classe dans l'éditeur de code et sélectionnez **Quick Actions and Refactorings / Extract Interface**. L'interface aurait été créée dans le même dossier que la classe.*

Exercice 3.2 : Programmer le référentiel (suite)

- 15.□ Dans la classe CategoryController du projet Todo.Web, changez le type de la déclaration de la variable `_categoryRepository`, de `CategoryRepository` en `ICategoryRepository`, en utilisant l'interface au lieu de la classe. Résolvez le using manquant.
- 16.□ Testez votre application. Elle devrait fonctionner comme avant.



Vous avez rendu les couches plus indépendantes. Le contrôleur utilise l'interface au lieu de l'implémentation (c'est-à-dire la classe). Vous les rendrez encore plus indépendantes dans un exercice ultérieur grâce à l'injection de dépendances.



Félicitations ! Vous avez développé une méthode dans la couche référentiel qui récupère les catégories dans la base de données. Vous avez aussi modifié le contrôleur afin d'accéder au référentiel à travers une interface.

Exercice 3.3 : Ajouter le niveau de service

Objectifs

Dans cet exercice, vous allez compléter la structure de l'application avec deux nouveaux projets : un projet modèle qui contient les classes du modèle, et un projet de service. Vous allez également modifier le projet Web afin d'appeler la couche service au lieu du référentiel.



Ajouter des interfaces au projet de service

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 3.3-Starting Point.sln et régénérez le projet.

***i** C'est la solution terminée de l'exercice 3.2 avec les bonus. Deux nouveaux projet y ont été ajoutés : Todo.Common, dans lequel les classes modèle ont été déplacées et Todo.Service dans lequel le package NuGet AutoMapper a été installé et configuré.*

2. ☐ Ouvrez le fichier I_TODO_Service.cs dans le dossier Interfaces du projet Todo.Service.

***i** Des méthodes effectuant des opérations sur la classe CategoryModel ont été définies dans l'interface.*

3. ☐ Ouvrez le fichier CategoryService.cs dans le dossier Services du projet Todo.Service. La classe TODO_Service doit implémenter l'interface I_Category_Service.



Votre code va ressembler à ceci :

```
public class CategoryService : I_Category_Service
```

Exercice 3.3 : Ajouter le niveau de service (suite)



Cliquez sur l'ampoule dans Visual Studio pour résoudre l'instruction using manquante.

- 4.□ Utilisez les fonctionnalités de refactorisation de Visual Studio pour implémenter toutes les méthodes de l'interface.



*Passez le curseur sur le texte `ICategoryService` dans la déclaration de la classe, cliquez sur l'ampoule et sélectionnez **Implement Interface**. Vous pouvez aussi utiliser la combinaison de touches `<Alt><Maj><F10>`.*



Implémenter la couche de service

- 5.□ Ajoutez une variable en lecture seule appelée `_categoryRepository` de type `ICategoryRepository` dans la classe `CategoryService`, au niveau de la classe.
- 6.□ Ajoutez un constructeur à la classe et donnez comme valeur à la variable `_categoryRepository` un nouvel objet `CategoryRepository`.



Code C# pour les deux dernières étapes :

```
ICategoryRepository _categoryRepository;  
public CategoryService()  
{  
    _categoryRepository = new CategoryRepository();  
}
```

Exercice 3.3 : Ajouter le niveau de service (suite)



Utilisez Visual Studio pour ajouter les instructions using qui manquent.



Les méthodes de `CategoryRepository` fonctionnent sur des entités `Category`, mais le contrôleur attend des objets `CategoryModel`. Pour automatiser la transformation d'objets `Category` en objets `CategoryModel`, vous allez utiliser un composant appelé `AutoMapper` qui est déjà configuré dans le projet.

- 7.□ Toujours dans la classe `CategoryService`, remplacez la ligne `throw` dans la méthode `List` par le code ci-dessous qui appelle le référentiel.



Votre code va ressembler à ceci :

```
public List<CategoryModel> List()
{
    return AutoMapper.Mapper.Map<List<CategoryModel>>(_categoryRepository.List());
}
```



La méthode `Map` de `AutoMapper` crée un objet du type passé en tant que paramètre générique, basé sur l'objet passé en paramètre à la méthode.

Dans le code précédent, la méthode `List` de la classe `CategoryRepository` renvoie une liste d'objets `Category`. `AutoMapper` crée une liste d'objets `CategoryModel` et donne comme valeur à chaque propriété de chaque objet `CategoryModel` de la liste celle de chaque propriété même nom dans l'objet `Category` correspondant.

Exercice 3.3 : Ajouter le niveau de service (suite)



Modifier le contrôleur afin d'utiliser le service au lieu du référentiel

8. ☐ Cliquez droit sur le projet Todo.Web et sélectionnez **Add | Reference**.
9. ☐ Décochez la référence Todo.Repository et cochez la référence Todo.Service dans l'onglet Solution.
10. ☐ Dans la classe CategoryController, appuyez sur <Ctrl><H> pour faire un remplacement. Saisissez **Repository** dans **Search item** et **Service** dans **Remplacement item**. Cochez le bouton **Match case** (« Aa ») et vérifiez que **Current Document** est sélectionné dans la liste déroulante. Cliquez sur le bouton **Replace All**, ou appuyez sur <Alt><A>. Corrigez l'instruction using qui indique une erreur en remplaçant Repositories par **Services**.
11. ☐ Toujours dans la classe CategoryController, modifiez la méthode CategoryList, en supprimant la déclaration du modèle et les lignes d'initialisation, et en renommant la variable category en **model**.



Votre code va ressembler à ceci :

```
public ActionResult CategoryList()
{
    var model = _categoryService.List();
    return View(model);
}
```

Exercice 3.3 : Ajouter le niveau de service (suite)

- 12.□ Pour finir, ouvrez le fichier Global.asax du projet Todo.Web et ajoutez la ligne suivante à la fin de la méthode Application_Start :



Votre code va ressembler à ceci :

```
Todo.Service.Configuration.AutoMapperConfig.Configure();
```



Cela fait que la méthode Configure sera appelée au démarrage de l'application pour initialiser AutoMapper.

- 13.□ Exécutez votre application. Elle devrait fonctionner comme avant.



Félicitations ! Vous avez complété la structure de l'application avec deux nouveaux projets : un projet modèle qui contient les classes du modèle, et un projet de service. Vous avez également modifié le projet Web afin d'appeler la couche service au lieu du référentiel.

Exercice 3.4 : Ajouter le conteneur DI Ninject

Objectifs

Dans cet exercice, vous allez ajouter de l'injection de dépendance (DI, Dependency Injection) aux projets Web et de service, avec Ninject. Vous configurerez l'injection de dépendance et modifierez le contrôleur et le service afin de l'utiliser.



Ajouter Ninject aux projets Web et de service


1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 3.4-Starting Point.sln et régénérez-la.



Il s'agit de la solution terminée de l'exercice 3.3.

2. ☐ Cliquez droit sur le projet Todo.Service dans Solution Explorer et sélectionnez **Manage NuGet Packages**.
3. ☐ Cliquez sur l'onglet **Browse** et sélectionnez **977 NuGet Packages** dans la liste déroulante Package source. Dans le champ de recherche en haut à gauche, tapez **Ninject** et appuyez sur <Entrée>. Cliquez sur **Ninject** et **Install** dans le volet droit, puis sur **OK** dans la fenêtre Review Changes. Si une fenêtre s'affiche à propos de la licence, cliquez sur **I Accept**. Fermez le dialogue NuGet.
4. ☐ Cliquez droit sur le projet Todo.Web dans Solution Explorer et sélectionnez **Manage NuGet Packages**.
5. ☐ Répétez l'étape 3 pour le projet Todo.Web.


Exercice 3.4 : Ajouter le conteneur DI Ninject (suite)

 *Les packages NuGet doivent être installés sur chaque projet. Vous utiliserez l'injection de dépendance sur le projet Web (pour obtenir des services) et sur le projet de service (pour obtenir les référentiels).*



Configurer Ninject

6. ☐ Dans le projet Todo.Web, ajoutez une classe appelée NinjectConfig au dossier App_Start. Supprimez la partie App_Start de l'espace de noms, en ne gardant que Todo.Web.


 *Les classes appelées lors du démarrage de l'application sont placées dans le dossier App_Start. Les appels sont faits depuis le fichier Global.asax.*

7. ☐ La classe NinjectConfig doit implémenter l'interface IDependencyResolver.




Votre code va ressembler à ceci :

```
public class NinjectConfig : IDependencyResolver
```

 *Ajoutez un using pour System.Web.Mvc.*

8. ☐ Copiez le contenu du fichier NinjectConfig.txt du dossier Copy-paste de la solution dans la classe. Résolvez les instructions using manquantes pour Ninject.

Exercice 3.4 : Ajouter le conteneur DI Ninject (suite)


 *Il s'agit de code que vous n'avez normalement pas à écrire. Essayez de le comprendre. Le constructeur initialise la variable de classe `_kernel`, puis appelle la méthode `AddBindings` actuellement vide. Les méthodes `GetService` et `GetServices` sont les implémentations de l'interface `IDependencyResolver`.*

- 9.□ Dans la méthode `AddBindings`, ajoutez une ligne qui lie l'interface `ICategoryService` à la classe `CategoryService`, à l'aide de la variable de classe `_kernel`. Résolvez les instructions `using` manquantes avec Visual Studio.



Votre code va ressembler à ceci :

```
_kernel.Bind<ICategoryService>().To<CategoryService>();
```

 *Un objet `CategoryService` va s'afficher pour chaque requête d'`ICategoryService`.*


- 10.□ Dans le fichier `Global.asax` du projet Web, ajoutez une ligne à la fin de la méthode `Application_Start` afin d'appeler la méthode `SetResolver` de la classe MVC `DependencyResolver`, en lui passant une nouvelle instance de la classe `NinjectConfig` définie dans les étapes précédentes.



Votre code va ressembler à ceci :

```
DependencyResolver.SetResolver(new NinjectConfig());
```

Exercice 3.4 : Ajouter le conteneur DI Ninject (suite)

 Vous avez configuré Ninject pour le projet Web. Vous allez maintenant le configurer pour le projet de service.

- 11.□ Ajoutez une nouvelle classe appelée NinjectConfig au dossier Configuration du projet Todo.Service. La classe doit être public.
- 12.□ Dans la classe NinjectConfig, ajoutez une méthode static appelée Config. La méthode doit recevoir un paramètre appelée kernel de type IKernel de l'espace de noms Ninject, et ne rien renvoyer. Résolvez l'instruction using manquante.
- 13.□ Dans la méthode Config, ajoutez une ligne liant l'interface ICategoryRepository à la classe CategoryRepository, à l'aide du paramètre kernel. Résolvez les instructions using manquantes.



Code C# pour les deux dernières étapes :

```
using Todo.Repository.Interfaces;
using Todo.Repository.Repositories;
namespace Todo.Service.Configuration
{
    public class NinjectConfig
    {
        public static void Config(IKernel kernel)
        {
            kernel.Bind<ICategoryRepository>().To<CategoryRepository>();
        }
    }
}
```

Exercice 3.4 : Ajouter le conteneur DI Ninject (suite)

- 14.□ Retournez à la classe NinjectConfig du projet Todo.Web et, dans la méthode AddBindings, appelez la méthode Config que vous venez de définir, en lui passant la variable locale `_kernel`.



Votre code doit ressembler à ceci :

```
Todo.Service.Configuration.NinjectConfig.Config(_kernel);
```

- 15.□ Générez la solution et corrigez les éventuelles erreurs.



Modifier le contrôleur et le service afin d'utiliser Ninject

- 16.□ Ouvrez la classe CategoryController du dossier Controllers du projet Todo.Web.
- 17.□ Ajoutez au constructeur un paramètre appelé **categoryService** de type ICategoryService.
- 18.□ Modifiez l'instruction du constructeur afin d'initialiser le `_categoryService` à l'aide du paramètre, au lieu d'instancier un nouvel objet CategoryService.



Votre code peut ressembler à ceci :

```
public CategoryController(ICategoryService  
    categoryService)  
{  
    _categoryService = categoryService;  
}
```

Exercice 3.4 : Ajouter le conteneur DI Ninject (suite)

- 19.□ Faites une modification semblable au constructeur de la classe CategoryService du projet Todo.Service, avec l'interface ICategoryRepository.



Votre code va ressembler à ceci :

```
public CategoryService(ICategoryRepository
    categoryRepository)
{
    _categoryRepository = categoryRepository;
}
```

- 20.□ Démarrez l'application. Elle devrait fonctionner comme précédemment.



Même si l'application fonctionne comme avant, son architecture a évolué et les différentes couches sont maintenant moins couplées. Les dépendances entre les couches sont définies dans des emplacements centralisés, dans les classes NinjectConfig des projets Todo.Web et Todo.Service. La plupart des infrastructures d'injection de dépendance permettent également de définir la configuration dans des fichiers XML, soit nativement, soit à travers des extensions.



Félicitations ! Vous avez ajouté de l'injection de dépendance aux projets Web et de service, avec Ninject. Vous avez configuré l'injection de dépendance et modifié le contrôleur et le service afin de l'utiliser.

Exercice 4.1 : Mettre en cache la sortie d'une action


Objectifs

Dans cet exercice, vous allez mettre la sortie d'une vue dans le cache de sortie. Vous allez également explorer d'autres propriétés de `OutputCache`.



Mettre la sortie d'une vue en cache

1. ☐ Créez un nouveau projet. Sélectionnez **Visual C# | Learning Tree** dans la liste de gauche et **ASP.NET MVC Basic Application** dans la liste centrale. Appelez le projet `OutputCaching`.

 *Ce modèle de projet a été créé spécifiquement pour le cours. Il est basé sur le modèle standard MVC sans authentification, et les contrôleurs et les vues ont été supprimés. Il comprend également un contrôleur `Home` et une vue `Index`. Vous trouverez ce modèle et des instructions sur comment l'installer dans le dossier `C:\977\Database\Project Templates`, ainsi que dans les fichiers d'exercice à télécharger.*

2. ☐ Générez la solution.
3. ☐ Ajoutez du code Razor dans la vue `Index` afin d'afficher l'heure, y compris les secondes, dans l'élément `<div>` existant.



Votre code Razor va ressembler à ceci :

```
Current time is: @DateTime.Now.ToLongTimeString()
```

Exercice 4.1 : Mettre en cache la sortie d'une action (suite)

- 4.❑ Exécutez l'application. Actualisez fréquemment la page. L'heure devrait être mise à jour à chaque seconde.
- 5.❑ Ajoutez un attribut `OutputCache` à la méthode `Index` du contrôleur `Home`, avec une durée de cinq secondes.



Votre code va ressembler à ceci :

```
[OutputCache(Duration = 5)]  
public ActionResult Index()
```

- 6.❑ Exécutez l'application. Actualisez fréquemment la page. L'heure devrait être mise à jour toutes les cinq secondes.



La sortie est le HTML envoyé au navigateur. Elle est mise dans le cache pendant cinq secondes.



Explorer l'effet de la propriété `VaryByParam` de l'attribut `OutputCache`

- 7.❑ Donnez une grande valeur à la propriété `Duration` de l'attribut `OutputCache` de la méthode `Index`, par exemple 100. Donnez `id` comme valeur à sa propriété `VaryByParam`.
- 8.❑ Ajoutez un paramètre nommé `id`, de type entier nullable, à la méthode `Index`.

Exercice 4.1 : Mettre en cache la sortie d'une action (suite)



Code des deux dernières étapes :

```
[OutputCache(Duration = 100, VaryByParam = "id")]  
public ActionResult Index(int? id)
```

- 9.□ Exécutez votre application. Actualisez la page régulièrement en appuyant sur la touche <F5>. L'heure ne devrait pas changer.



Le paramètre id ne change pas et la vue reste dans le cache pendant 100 secondes.

- 10.□ Modifiez l'URL dans la barre d'adresse du navigateur, en y ajoutant une valeur pour le paramètre id.



L'URL va ressembler à ceci :

`http://localhost:port/Home/Index/1`



Remplacez port par votre numéro de port.



Si vous actualisez la page, l'heure n'est pas modifiée. Un cache séparé est créé pour chaque valeur de id. Chaque fois que la valeur de id est modifiée, les données correspondantes sont extraites du cache, ou le code de la méthode d'action est exécuté si elles n'existent pas encore ou ont expiré.



Félicitations ! Vous avez mis la sortie d'une vue dans le cache de sortie. Vous avez également exploré d'autres propriétés de OutputCache.

Exercice 4.2 : Personnaliser des routes

Objectifs

Dans cet exercice, vous allez ajouter une nouvelle route afin que les utilisateurs aient un accès direct aux listes. Vous allez également utiliser le routage par attributs pour simplifier la configuration des routes.



Ajouter un nouvel élément dans la table de routage

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 4.2-Starting Point.sln et régénérez-la.



C'est la solution terminée de l'exercice 3.4, dans laquelle une nouvelle interface et son implémentation ont été définies dans les couches référentiel et de service.

2. ☐ Ouvrez le fichier RouteConfig.cs dans le dossier App_Start du projet Todo.Web.



Quels sont les noms des contrôleurs et méthodes d'action par défaut ?

-
3. ☐ Définissez une nouvelle route et ajoutez-la à la collection routes. La nouvelle route doit traiter des URL telles que `http://server:port/List/2`. Elle doit également appeler la méthode d'action `ItemList` du contrôleur `TodoList`, en lui passant le dernier segment de l'URL (2 dans l'exemple précédent) en tant que paramètre appelé `id`.

Exercice 4.2 : Personnaliser des routes (suite)



Placez la définition de la nouvelle route entre l'appel à `IgnoreRoute` et la définition de la route par défaut.



Votre code va ressembler à ceci :

```
routes.MapRoute("List", "List/{id}",  
new { controller = "ToDoList", action = "ItemList" });
```

4. ☐ Exécutez l'application. Ajoutez `List/2` à la fin de l'URL.



Les éléments des catégories correspondant au numéro de liste sélectionné s'affichent.

5. ☐ Essayez de modifier l'URL, en remplaçant le numéro de liste par un nom de liste tel que `Shopping`.



Que se passe-t-il ?



Remplacer l'élément de la table de routage par du routage par attributs



Personnaliser la table de routage dans la classe `RouteConfig` peut devenir complexe. Il est bien plus facile d'effectuer les personnalisations directement sur les méthodes d'action ou les classes contrôleur, avec le routage par attributs.

Exercice 4.2 : Personnaliser des routes (suite)

- 6.□ Arrêtez le débogage, puis supprimez ou commentez la route que vous avez ajoutée dans la classe RouteConfig dans la section précédente.



Gardez la définition de la route par défaut.



Le routage par attributs n'est pas opérationnel par défaut. Il faut l'activer explicitement dans l'application.

- 7.□ Ajoutez une ligne dans la méthode RegisterRoute afin d'activer le routage par attributs.



La ligne suivante doit être placée avant la route par défaut.



Votre code va ressembler à ceci :

```
routes.MapMvcAttributeRoutes();
```

- 8.□ Allez dans la classe TodoListController du projet Todo.Web. Ajoutez un attribut Route à la méthode ItemList, en passant à son constructeur la même définition de route utilisée dans la section précédente. La route doit toujours traiter des URL telles que `http://server:port/List/2`.



Votre code va ressembler à ceci :

```
[Route("List/{id}")]  
public ActionResult ItemList(int id)  
{
```

Exercice 4.2 : Personnaliser des routes (suite)

- 9.□ Testez votre application en ajoutant List/2 à la fin de l'URL.
- 10.□ Essayez de voir le détail d'une liste avec le menu Categories et en cliquant sur une liste.



La liste s'affiche-t-elle dans le navigateur ?

- 11.□ Ajoutez un second attribut Route à la méthode d'action ItemList, en utilisant le chemin complet depuis la racine de l'application, le nom de la méthode d'action et le paramètre id.



Votre code va ressembler à ceci :

```
[Route("List/{id}")]  
[Route("ToDoList/ItemList/{id}")]  
public ActionResult ItemList(int id)
```




Ajouter une nouvelle méthode d'action au contrôleur pour traiter les demandes de listes par nom



Vous allez maintenant ajouter une nouvelle méthode d'action à `ToDoListController`, avec un attribut de routage traitant les demandes comprenant le nom d'une liste. Les méthodes du service et du référentiel pour traiter ces demandes ont déjà été ajoutées.

Exercice 4.2 : Personnaliser des routes (suite)

- 12.□ Ajoutez une nouvelle méthode d'action appelée **ItemList** à `ToDoListController` qui reçoit un paramètre string appelé `name` et qui appelle la méthode `ListByName` de la variable `_todoItemService` pour récupérer et renvoyer la liste demandée.

 Vous pouvez taper le code vous-même ou le copier depuis le fichier `ListByName.txt` du dossier *Copy-paste*.



Votre code va ressembler à ceci :

```
[Route("List/{name}")]
public ActionResult ItemList(string name)
{
    var model = _todoItemService.ListByName(name);
    return View(model);
}
```

- 13.□ Exécutez votre application et saisissez une URL telle que `http://localhost:portnumber/List/Shopping`, en remplaçant *portnumber* par votre numéro de port.

 Une page d'erreur s'affiche. Essayez de comprendre pourquoi.

La requête est traitée par la première méthode `ItemList`, car le modèle de son attribut, `List/{id}`, correspond à l'URL. Mais le paramètre attendu est de type `integer`. Vous allez ajouter une contrainte à cette route pour résoudre le problème.

- 14.□ Ajoutez une contrainte à la première route de `ItemList`, afin qu'elle ne traite que les paramètres `id` entier.

Exercice 4.2 : Personnaliser des routes (suite)



Votre code va ressembler à ceci :

```
[Route("List/{id:int}")]
```

- 15.□ Exécutez votre application et vérifiez qu'elle fonctionne comme prévu : en navigant par le menu, ou en entrant directement une URL avec un numéro de liste ou un nom.



Félicitations ! Vous avez ajouté une nouvelle route afin que les utilisateurs aient un accès direct aux listes. Vous avez également utilisé le routage par attributs pour simplifier la configuration des routes.

Exercice facultatif 4.3 : Structurer une application avec des zones

Objectifs

Dans cet exercice, vous allez ajouter une zone d'administration à votre application.



Ajouter une zone au projet Web

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 4.3-Starting Point.sln et régénérez-la.



Il s'agit de la solution terminée de l'exercice 4.2.

2. ☐ Cliquez droit sur le projet Todo.Web et sélectionnez **Add | Area**. Appelez la zone Admin.



Un dossier Areas est ajouté au projet avec un sous-dossier Admin. Celui-ci a des dossiers Controllers, Models et Views, ainsi qu'une classe AdminAreaRegistration.

3. ☐ Ajoutez un contrôleur vide Users au dossier Controllers de la zone Admin.
4. ☐ Ajoutez une vue Index pour la méthode d'action Index. Remplacez le texte de l'en-tête et la valeur de ViewBag.Title par **Users Administration**.
5. ☐ Ouvrez la vue _Layout dans le dossier /Views/Shared et ajoutez un nouvel élément de menu à l'aide d'un ActionLink, conduisant à la méthode Index du contrôleur Users dans la zone Admin, après les ActionLink existant. Mettez le lien dans un élément .

Exercice facultatif 4.3 : Structurer une application avec des zones (suite)



Utilisez le `_Layout.cshtml` qui se trouve dans la zone principale, pas dans la zone Admin.



Votre code va ressembler à ceci :

```
<li>@Html.ActionLink("Administration",  
"Index", new { area = "Admin", controller =  
"Users" })</li>
```

- 6.□ Exécutez votre application et allez dans la page d'administration.



La page s'affiche-t-elle comme prévu ?



Visual Studio génère une page de disposition spécifique pour chaque zone, située dans le dossier `Views\Shared` de la zone. Vous pouvez personnaliser la page de disposition de la zone, ou réutiliser la page de disposition commune de la zone principale.



Comment réutiliser la vue `_Layout.cshtml` de la zone principale pour la page Index du contrôleur Users ?

Exercice facultatif 4.3 : Structurer une application avec des zones (suite)



*Vous pouvez changer la valeur de la propriété **Layout** dans le fichier `_ViewStart.cshtml` du dossier **Views** de la zone.*

- 7.□ Arrêtez le débogage et ouvrez le fichier `_ViewStart.cshtml` situé dans le dossier **Views** de la zone **Admin**. Supprimez la partie `Areas/Admin/` du chemin.



Votre code va ressembler à ceci :

```
Layout = "~/Views/Shared/_Layout.cshtml";
```

- 8.□ Exécutez votre application et vérifiez que la page d'administration utilise bien la disposition commune.
- 9.□ Dans la page d'administration, cliquez sur le lien **Home** ou **Categories**.



Les liens ne fonctionnent plus. Essayez de comprendre pourquoi et de corriger le problème.

- a. Dans la vue `_Layout.cshtml` de la zone principale, remplacez les noms des contrôleurs dans les liens **Home** et **Categories** par un objet anonyme qui a deux propriétés pour la zone (`area`) et le contrôleur. La valeur de la zone doit être une chaîne vide.

Exercice facultatif 4.3 : Structurer une application avec des zones (suite)



Votre code va ressembler à ceci :

```
<li>@Html.ActionLink("Home", "Index", new { area = "", controller = "Home"
})</li>
<li>@Html.ActionLink("Categories", "CategoryList",
new { area = "", controller = "Category" })</li>
```



**Félicitations ! Vous avez ajouté une zone d'administration
à votre application.**

Exercice 5.1 : Remplacer des éléments HTML par des aides HTML

Objectifs

Dans cet exercice, vous allez remplacer des éléments de saisie HTML par des aides HTML. Vous allez également ajouter et gérer une liste déroulante fortement typée.



Remplacer des éléments HTML par des aides HTML

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 5.1-Starting Point.sln et régénérez-la.

***i** C'est la solution terminée du À vous 5a dans laquelle des aides HTML sont utilisées dans la vue ItemEdit.cshtml du dossier Views\ToDoList, au lieu d'éléments HTML simples.*

2. ☐ Ouvrez la vue ItemEdit du dossier Views\ToDoList. Lisez le code afin de le comprendre.

***i** Le formulaire et ses éléments de saisie ont été remplacés par des aides HTML. Les éléments HTML précédents ont été mis en commentaires afin que vous puissiez comparer la version précédente à l'actuelle.*

3. ☐ Exécutez l'application. Elle doit fonctionner comme précédemment. Ne fermez pas encore le navigateur.

4. ☐ Allez jusqu'à la page Edit et cliquez droit sur la page dans le navigateur pour afficher son code source HTML.

***i** Le HTML est semblable à celui qui est en commentaire dans la vue, les attributs id et name ayant des valeurs correctes.*

Exercice 5.1 : Remplacer des éléments HTML par des aides HTML (suite)



Ajouter une liste déroulante fortement typée

***i** Une aide HTML `DropDownListFor` a besoin d'au moins deux propriétés : Un objet `SelectList` contenant les éléments à afficher dans la liste, et la valeur de l'élément sélectionné. Vous allez modifier le modèle `ToDoListModel` et ajouter ces deux propriétés.*

- 5.□ Allez dans la classe `ToDoListModel` du dossier `Models` du projet `ToDo.Common`. Ajoutez deux propriétés, une appelée **`CategorySelectList`** de type `SelectList`, et une autre appelée **`SelectedCategoryId`** de type entier.



Votre code va ressembler à ceci :


```
public class ToDoListModel
{
    public ToDoListModel(List<ToDoItemModel>
todoItems)
    {
        TodoItems = todoItems;
    }
    public List<ToDoItemModel> TodoItems { get; set; }
    public SelectList CategorySelectList { get; set; }
    public int SelectedCategoryId { get; set; }
}
```



Ajoutez l'instruction `using` manquante avec Visual Studio.

Exercice 5.1 : Remplacer des éléments HTML par des aides HTML (suite)

- 6.□ Allez dans `TodoListController` du projet `Todo.Web`. Dans la méthode d'action `ItemList` qui prend un paramètre entier, ajoutez du code pour :
- Charger la liste des catégories
 - Initialiser les propriétés `CategorySelectList` et `SelectedCategoryId` de la variable `model`.


 Tapez le code vous-même, ou collez-le depuis le fichier `ItemList.txt` du dossier *Copy-paste*.



Votre code va ressembler à ceci :

```
public ActionResult ItemList(int id)
{
    var model = _todoItemService.List(id);
    var categories = _categoryService.List();
    model.CategorySelectList = new
SelectList(categories, "Id", "Title");
    model.SelectedCategoryId = id;
    return View(model);
}
```

- 7.□ Ouvrez la vue `ItemList` du dossier `TodoList`. À la fin du code HTML, ajoutez une aide HTML `BeginForm` dans un bloc `using` afin de créer un formulaire. Passez-lui l'action `ChangeList`.

 `BeginForm` n'a pas de surcharge prenant simplement la méthode d'action. Vous pouvez spécifier la méthode d'action et le contrôleur, ou utiliser un objet `route` pour indiquer la méthode d'action, si elle est dans le contrôleur courant.

Exercice 5.1 : Remplacer des éléments HTML par des aides HTML (suite)

- 8.□ Dans le bloc using, créez une liste déroulante à l'aide de DropDownListFor. Passez-lui les paramètres suivants :
- Une expression lambda pour la propriété SelectedCategoryId
 - Et la propriété CategorySelectList de la variable Model.
- 9.□ Toujours dans le bloc using, ajoutez un bouton submit avec le texte Go.



Code des deux dernières étapes :

```
@using (Html.BeginForm(new { action = "ChangeList" }))  
{  
    @Html.DropDownListFor(m => m.SelectedCategoryId, Model.CategorySelectList)  
    <button type="submit">Go</button>  
}
```

- 10.□ Exécutez votre application. Allez dans une liste de listing. Une liste déroulante contenant la liste des listings doit s'afficher en bas, la liste courante étant sélectionnée.



Si vous cliquez sur le bouton Go, une page d'erreur s'affiche. Vous allez corriger cela dans les étapes suivantes.

- 11.□ Arrêtez le débogage. Dans TodoListController, ajoutez une méthode d'action appelée ChangeList qui reçoit un paramètre entier appelé SelectedCategoryId.

Exercice 5.1 : Remplacer des éléments HTML par des aides HTML (suite)

- 12.□ Dans la méthode `ChangeList`, redirigez vers la méthode d'action `ItemList`, en lui passant `SelectedCategoryId` en tant que valeur de route `id`.



Code C# des deux dernières étapes

```
public ActionResult ChangeList(int SelectedCategoryId)
{
    return RedirectToAction("ItemList",
        new { id = SelectedCategoryId });
}
```

- 13.□ Exécutez l'application, allez à une liste d'éléments, sélectionnez une catégorie dans la liste déroulante et cliquez sur le bouton **Go**.



Les éléments de la catégorie sélectionnée doivent s'afficher.




Félicitations ! Vous avez remplacé des éléments de saisie HTML par des aides HTML. Vous avez également ajouté et géré une liste déroulante fortement typée.

Exercice 5.2 : Développer une aide HTML LabelTextBox

Objectifs


Dans cet exercice, vous allez construire une méthode d'aide HTML personnalisée et l'utiliser dans une vue.

 Les formulaires d'édition ont souvent des ensembles couples label-zone de texte, les deux étant liés à la même propriété du modèle. Par exemple, une propriété *Title* du modèle sera utilisée avec un label pour afficher le titre, et avec une zone de texte pour en saisir la valeur. Vous allez développer une nouvelle aide HTML, *LabelTextBox*, qui regroupera un label et une zone de texte dans la même méthode d'aide. Dans la section bonus, vous allez également développer une méthode *LabelTextBoxFor* mettant en œuvre une liaison fortement typée.




Développer la méthode d'extension HTML *LabelTextBox*

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 5.2-Starting Point.sln et régénérez-la.


 C'est la solution terminée de l'exercice 5.1, incluant le dernier À vous terminé. Les classes *TodoItemRepository* et *TodoItemService* ont également été complétées afin d'inclure toutes les opérations CRUD (Create, Retrieve, Update, Delete).

2. ☐ Ouvrez le fichier *MyExtensions* dans le dossier *Extensions* du projet *Todo.Web*.

 Il n'y a qu'une méthode d'aide HTML, *ImageActionLink*, du dernier À vous.

Exercice 5.2 : Développer une aide HTML LabelTextBox (suite)


- 3.□ Ajoutez une nouvelle méthode appelée `LabelTextBox` et faites-en une méthode d'extension d'aide HTML. La méthode doit être static et avoir trois paramètres en plus du paramètre `html` : `name`, `text` et `value`. La valeur de retour doit être un `HtmlString`.

 *Visual Studio affichera une erreur, car la méthode ne renvoie pas encore de valeur. Vous la corrigerez dans une étape suivante.*



Votre code va ressembler à ceci :

```
public static HtmlString LabelTextBox(this HtmlHelper html, string name, string text,  
                                     string value)  
{  
}
```

 *Pour les trois prochaines étapes, vous pouvez taper le code vous-même, ou le copier depuis le fichier `LabelTextBox` dans le dossier `Copy-paste`.*

- 4.□ Créez un label dont le texte est la valeur du paramètre `text`, en utilisant la classe `TagBuilder`. Ajoutez-y un attribut `for`, dont la valeur est le paramètre `name`.



Vous pouvez vous inspirer de la méthode `ImageActionLink` existante.

Exercice 5.2 : Développer une aide HTML LabelTextBox (suite)



Votre code va ressembler à ceci :

```
var label = new TagBuilder("label");  
label.Attributes.Add("for", name);  
label.SetInnerText(text);
```



Pour ajouter des attributs à un TagBuilder, vous pouvez utiliser la méthode Add de sa propriété Attributes, ou appeler sa méthode MergeAttribute.

- 5.□ Après les trois lignes du label, créez une zone de texte. Pour cela, vous pouvez à nouveau utiliser la classe TagBuilder, en créant un élément input avec quatre attributs : type dont la valeur est text, id et name dont la valeur est le paramètre name, et value dont la valeur est égale au paramètre value. Ajoutez également une classe appelée form-control à la zone de texte.



Votre code va ressembler à ceci :

```
var textbox = new TagBuilder("input");  
textbox.Attributes.Add("type", "text");  
textbox.Attributes.Add("id", name);  
textbox.Attributes.Add("name", name);  
textbox.Attributes.Add("value", value);  
textbox.AddCssClass("form-control");
```

- 6.□ Renvoyez enfin la chaîne du label concaténée avec la chaîne de la zone de texte et encapsulée dans un objet HtmlString, en séparant les deux chaînes par un espace.

Exercice 5.2 : Développer une aide HTML LabelTextBox (suite)



Votre code va ressembler à ceci :

```
return new HtmlString(label.ToString() + " " + textbox.ToString());
```

- 7.□ Générez la solution et corrigez les éventuelles erreurs.



Tester l'aide LabelTextBox dans une vue

- 8.□ Ouvrez la vue ItemEdit dans le dossier Views\ToDoList. Ajoutez une instruction using pour Todo.Web.Extensions en haut du fichier.
- 9.□ Dans le premier élément <div>, juste au-dessus de la ligne LabelFor, ajoutez un appel à votre méthode d'aide LabelTextBox, en lui passant trois paramètres : La chaîne Title pour les deux premiers paramètres et la propriété Title du modèle pour les deux derniers paramètres. Ajoutez
 après l'appel à la méthode.





Votre code peut ressembler à ceci :


```
@Html.LabelTextBox("Title", "Title", Model.Title)  
<br />
```

- 10.□ Exécutez l'application et allez à la page d'édition.

Exercice 5.2 : Développer une aide HTML LabelTextBox (suite)


 *Quelle est la différence entre les deux lignes ?*


 *Le label personnalisé défini dans le modèle n'est pas affiché.*

 *Le texte du titre du label est codé en dur et n'utilise pas la valeur de l'attribut Display du modèle. Cela est corrigé dans la section bonus.*

 *L'attribut Display définit le texte à afficher pour une propriété. Il sera présenté ultérieurement dans le chapitre.*

11. ☐ Cliquez droit sur la page pour afficher le code source de la page.

 *Comparez les deux versions. Qu'est-ce qui est identique ? Quelles sont les différences ?*

 *Si vous sélectionnez Inspect à la place de View Source, le navigateur affiche le résultat calculé et parfois même une balise auto-fermante.*

12. ☐ De retour dans la classe MyExtensions, ajoutez un paramètre à l'appel de la méthode ToString sur la variable textbox, afin que l'élément généré soit auto-fermant.

Exercice 5.2 : Développer une aide HTML LabelTextBox (suite)



Votre code va ressembler à ceci :

```
return new HtmlString(label.ToString() + " " +  
    textbox.ToString(TagRenderMode.SelfClosing));
```

- 13.□ Exécutez à nouveau l'application et affichez le code source de la page d'édition. Vérifiez que l'élément input est maintenant bien formé.



Un des avantages de l'utilisation des aides HTML au lieu d'éléments HTML est d'imposer une mise en forme du HTML généré. Notez également que les éléments auto-fermants permettent de minimiser le HTML envoyé au navigateur.

- 14.□ Dans la vue Edit, supprimez les deux lignes qui appellent LabelFor et TextBoxFor dans le premier élément <div>.
- 15.□ Exécutez l'application et modifiez le titre d'un des éléments de listing. Vérifiez que la modification est affichée dans la liste.



Félicitations ! Vous avez construit une méthode d'aide HTML personnalisée et l'avez utilisée dans une vue.

Exercice 5.3 : Utiliser les modèles d'édition et d'affichage

Objectifs

Dans cet exercice, vous allez afficher et éditer un modèle avec les méthodes d'extension **Display** et **Editor**. Vous allez également ajouter des attributs au modèle afin de personnaliser l'affichage et l'édition des éléments du listing.



Ajouter une vue pour afficher le détail d'un élément de listing

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 5.3-Starting Point.sln et régénérez-la.



Il s'agit de la solution terminée de l'exercice 5.2 avec les ajouts suivants :

- Une classe *StyleModel* dans le projet *Todo.Common*
- Une propriété *Style* dans la classe *TodoItemModel*
- Des énumérations dans le projet *Todo.Common*
- Des mappages *AutoMapper* ajustés dans le projet *Todo.Service*
- Un référentiel *Style* dans le projet *Todo.Repository*
- Un service *Style* dans le projet *Todo.Service*



*Dans la version précédente de l'application, l'utilisateur allait dans la page d'édition d'un élément en cliquant dessus dans la vue des éléments de la liste. Une nouvelle vue *ItemDisplay* a été ajoutée au dossier *Views\TodoList* et liée aux éléments de la liste. Cette vue a un lien vers la vue *ItemEdit*.*

2. ☐ Exécutez votre application et allez dans une liste. Cliquez sur un élément de la liste.

Exercice 5.3 : Utiliser les modèles d'édition et d'affichage (suite)



Quelle vue est affichée ?



La vue ItemDisplay n'affiche pas encore les éléments de la liste. Vous allez corriger cela dans les étapes suivantes.

3. ☐ Cliquez sur le lien Edit.



La vue Edit est affichée.

4. ☐ Arrêtez l'application. Allez dans la vue ItemDisplay du dossier Views\ToDoList. Au-dessus de l'élément <div> existant, appelez la méthode DisplayForModel afin d'afficher les propriétés du modèle.



Votre code Razor va ressembler à ceci :

```
@Html.DisplayForModel()
```

5. ☐ Exécutez votre application et allez à un élément de la liste. L'élément doit être affiché quand vous cliquez dessus dans la liste. Un clic sur le lien Edit doit conduire à la page d'édition.



Appuyez sur <Ctrl><F5> pour vider le cache du navigateur et actualiser la page. L'aide DisplayForModel définit des classes CSS pour les éléments HTML générés. Ces classes sont définies à la fin du fichier site.css situé dans le dossier Contents du projet Todo.Web.

Exercice 5.3 : Utiliser les modèles d'édition et d'affichage (suite)



Personnaliser l'affichage avec des attributs



Certains champs tels que `id` ne devraient pas être affichés pour l'utilisateur. Vous allez les cacher avec des attributs dans le modèle. Vous allez aussi gérer l'affichage des valeurs nulles et des dates.

- 6.□ Dans la classe `TodoItemModel` du projet `Todo.Common`, ajoutez une instruction `using` pour les espaces de noms `System.Web.Mvc` et `System.ComponentModel.DataAnnotations`.
- 7.□ Toujours dans la classe `TodoItemModel`, ajoutez un attribut **`HiddenInput`** aux trois propriétés `id` : `Id`, `CategoryId` et `StyleId`. Ajoutez la propriété `false` à l'attribut `DisplayValue`.



Votre code peut ressembler à ceci :

```
[HiddenInput(DisplayValue = false)]
```

- 8.□ Ajoutez un attribut `DisplayFormat` aux trois propriétés nullable qui n'ont actuellement pas de valeur : `Quantity`, `Priority` et `DueDate`. Donnez comme valeur à la propriété `NullDisplayText` un texte à afficher quand la valeur est nulle, par exemple un tiret.



Votre code peut ressembler à ceci :

```
[DisplayFormat(NullDisplayText = "-")]
```

Exercice 5.3 : Utiliser les modèles d'édition et d'affichage (suite)

9. ☐ Ajoutez un attribut **DataType** à la propriété DueDate, en passant à son constructeur une valeur indiquant que seule la partie date de la propriété doit s'afficher.



Votre code va ressembler à ceci :

```
[DataType(DataType.Date)]
```

10. ☐ Ajoutez un attribut Display aux propriétés de votre choix pour modifier les labels affichés.



Voici quelques suggestions :

```
[Display(Name = "Item title")]
```

```
public string Title { get; set; }
```

```
[Display(Name = "Due date")]
```

```
public DateTime? DueDate { get; set; }
```

11. ☐ Exécutez votre application. L'affichage devrait être meilleur.



Remplacer des aides HTML spécifiques par EditorFor

12. ☐ Ouvrez la vue ItemEdit dans le dossier Views\TodoItem.
13. ☐ Supprimez tout le code Razor et les éléments HTML situés sous l'élément <h2>.
14. ☐ Remplacez-le par le contenu du fichier ItemEdit.txt du dossier Copy-paste de la solution.

Exercice 5.3 : Utiliser les modèles d'édition et d'affichage (suite)



Votre code va ressembler à ceci :

```
@using (Html.BeginForm())
{
    @Html.HiddenFor(m => m.Id)
    @Html.HiddenFor(m => m.CategoryId)
    @Html.HiddenFor(m => m.StyleId)
    <div class="form-group">
        @Html.LabelFor(m => m.Title)
        @Html.EditorFor(m => m.Title, new { htmlAttributes = new { @class = "form-
control" } })
    </div>
    <div class="checkbox">
        @Html.LabelFor(m => m.Done)
        @Html.EditorFor(m => m.Done)
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Quantity)
        @Html.EditorFor(m => m.Quantity, new { htmlAttributes =
new { @class = "form-control" } })
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Priority)
        @Html.EditorFor(m => m.Priority, new { htmlAttributes = new { @class = "form-
control" } })
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.DueDate)
        @Html.EditorFor(m => m.DueDate, new { htmlAttributes = new { @class = "form-
control" } })
    </div>
    <button type="submit" class="btn btn-default">OK</button>
}
```

Exercice 5.3 : Utiliser les modèles d'édition et d'affichage (suite)



Les aides HTML spécifiques ont été remplacées par l'aide générique EditorFor. Celle-ci utilise le type de données des propriétés du modèle pour déterminer l'interface utilisateur HTML à utiliser. On peut aussi fournir des indications complémentaires avec l'attribut DataType.

- 15.□ Exécutez votre application. Elle doit fonctionner comme précédemment. La page d'édition doit avoir plus de champs.



Félicitations ! Vous avez affiché et édité un modèle avec les méthodes d'extension Display et Editor. Vous avez également ajouté des attributs au modèle afin de personnaliser l'affichage et l'édition des éléments du listing.

Exercice 5.4 : Ajouter la validation à l'étude de cas

Objectifs

Dans cet exercice, vous allez valider le formulaire à l'aide d'attributs d'annotation de données, ceux qui sont intrinsèques, ou des attributs personnalisés. Vous allez également implémenter une règle métier dans le contrôleur.



Valider le formulaire avec des attributs d'annotation de données

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 5.4-Starting Point.sln et régénérez-la.



C'est la solution terminée de l'exercice 5.3, y compris les modèles du bonus.

2. ☐ Dans la classe TodoItemModel du projet Todo.Common, ajoutez des attributs aux propriétés pour respecter les contraintes suivantes :

- Le champ Title est obligatoire et doit avoir au moins 3 caractères et pas plus de 20.
- Le champ Quantity doit être un nombre positif inférieur à 10.
- Le champ DueDate doit être une date valide.



Gardez les attributs existants. Notez que certaines contraintes peuvent déjà être satisfaites par le code.

Exercice 5.4 : Ajouter la validation à l'étude de cas (suite)



Votre code va ressembler à ceci :

```
[Display(Name = "Item title")]
[Required]
[StringLength(20, MinimumLength = 3)]
public string Title { get; set; }
```

```
[DisplayFormat(NullDisplayText = "-")]
[Range(0, 10)]
public double? Quantity { get; set; }
```

```
[DisplayFormat(NullDisplayText = "-")]
[DataType(DataType.Date)]
[Display(Name = "Due date")]
public DateTime? DueDate { get; set; }
```

- 3.□ Dans la classe `TodoListController`, cherchez la méthode `ItemEdit` qui a un attribut `HttpPost` et ajoutez-y du code qui vérifie si le modèle est valide. Si ce n'est pas le cas, renvoyez la vue courante en lui passant le modèle.





Votre code va ressembler à ceci :

```
[HttpPost]
public ActionResult ItemEdit(ListingItemModel model)
{
    if (!ModelState.IsValid)
        return View(model);
    (code existant omis)
```

Exercice 5.4 : Ajouter la validation à l'étude de cas (suite)

- 4.□ Exécutez votre application et allez dans le formulaire d'édition. Essayez d'entrer des données invalides et soumettez le formulaire.

 *Les champs invalides doivent avoir une bordure rouge et le formulaire n'est pas enregistré. Si ce n'est pas le cas, appuyez sur <Ctrl><F5> pour actualiser le navigateur.*

 *Ce serait mieux pour l'utilisateur s'il savait pourquoi les données sont invalides. Vous allez ajouter des messages pour cela.*

- 5.□ Dans la vue ItemEdit.cshtml, ajoutez des appels à `ValidationMessageFor` pour les champs qui peuvent avoir des messages de validation : Title, Quantity, DueDate.



Code utilisé pour le premier champ :

```
@Html.EditorFor(m => m.Title)
@Html.ValidationMessageFor(m => m.Title)
```

- 6.□ Au début du formulaire, ajoutez un **ValidationSummary**, après les champs cachés.





Votre code Razor va ressembler à ceci :

```
@Html.ValidationSummary("Errors found:")
```

- 7.□ Exécutez votre application et vérifiez que les messages d'erreur s'affichent comme prévu.

Exercice 5.4 : Ajouter la validation à l'étude de cas (suite)

 *Edge et Chrome utilisent un sélecteur de date pour renseigner les champs de date et éviter les saisies au mauvais format. Utilisez Internet Explorer ou Firefox pour tester la validation des dates.*

 *Les messages d'erreur par défaut ne sont pas toujours appropriés pour les utilisateurs finaux. Ils peuvent être modifiés à l'aide d'attributs.*

- 8.□ Ajoutez des messages d'erreur aux attributs de validation de la classe `TodoItemModel` à l'aide des valeurs du tableau suivant. Exécutez et testez.

Propriété	Attribut	ErrorMessage
Title	StringLength	The title must have between 3 and 20 characters.
Quantity	Range	The quantity must be 10 or less.



Code pour la propriété Title

```
[Display(Name = "Item title")]
[Required]
[StringLength(20, MinimumLength = 3, ErrorMessage =
    "The title must have between 3 and 20 characters.")]
public string Title { get; set; }
```

Exercice 5.4 : Ajouter la validation à l'étude de cas (suite)



Code pour la propriété Quantity

```
[DisplayFormat(NullDisplayText = "-")]  
[Range(0, 10, ErrorMessage =  
    "The quantity must be 10 or less.")]  
public double? Quantity { get; set; }
```



Activer la validation côté client



Pour l'instant, la validation s'effectue sur le serveur lors de l'envoi du formulaire. Vous allez activer la validation côté client en JavaScript dans le navigateur.

Les modèles MVC de Visual Studio 2013 et des versions ultérieures ne contiennent pas le package jQuery Validation par défaut, contrairement aux versions précédentes. Les packages suivants ont été préinstallés dans le point de départ de l'exercice : jQuery Validation et Microsoft jQuery Unobtrusive Validation.

- 9.□ Ouvrez le fichier BundleConfig qui se trouve dans le dossier App_Start.



Comment s'appelle le groupe dans lequel se trouve la validation jQuery ?

- 10.□ Ajoutez le groupe jQuery Validation dans le fichier _layout.cshtml, sur la ligne qui suit l'ajout du groupe jQuery.

Exercice 5.4 : Ajouter la validation à l'étude de cas (suite)



Votre code Razor va ressembler à ceci :

```
@Scripts.Render("~/bundles/jqueryval")
```



Étant donné que jQuery Validation utilise jQuery, il doit être placé après.

- 11.□ Démarrez l'application et essayez de saisir des valeurs non valides dans le formulaire. Les messages d'erreur doivent s'afficher immédiatement.




Félicitations ! Vous avez validé le formulaire avec des attributs d'annotation de données prédéfinis. Vous pouvez maintenant passer aux bonus pour apprendre à développer un attribut personnalisé et à implémenter une règle métier dans le contrôleur.

Exercice 6.1 : Afficher un dialogue avec jQuery et Ajax

Objectifs


Dans cet exercice, vous allez utiliser jQuery avec Ajax pour afficher un dialogue permettant à l'utilisateur de choisir le style des éléments des listes.

 Vous allez permettre à l'utilisateur de personnaliser l'affichage des éléments des listes en le laissant choisir la police de caractères et la couleur utilisés. Des styles prédéfinis existent dans la table Style de la base de données. Dans le formulaire d'édition, l'utilisateur pourra sélectionner un style dans une liste et l'appliquer à l'élément. La sélection du style se fera dans un dialogue créé avec jQuery UI et alimenté par un appel Ajax au serveur.



Ajouter le balisage HTML et le code jQuery pour le traitement d'un clic sur le bouton Look and feel.


1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 6.1-Starting Point.sln et régénérez-la.

 C'est la solution Todo terminée, y compris avec le dernier À vous. Ninject a été configuré afin que les classes style puissent être utilisées avec l'injection de dépendance. La classe *TodoItemRepository* a également été modifiée : elle charge les styles pendant la lecture des éléments.

2. ☐ Ouvrez la vue ItemEdit dans le dossier View\TodoList du projet Web. Un bouton avec l'id btnStyle et un élément div avec l'id divStyle ont été ajoutés vers la fin du fichier.

Exercice 6.1 : Afficher un dialogue avec jQuery et Ajax (suite)


- 3.□ Ajoutez le code suivant dans le corps de la fonction, dans la section scripts, après l'appel à datepicker. Ce code intercepte l'événement click sur le bouton btnStyle.

 Vous pouvez copier le code depuis le fichier EditScript du dossier Copy-paste de la solution pour éviter de le taper.



Code jQuery :

```
$('#btnStyle').click(function () {  
    $.get(  
        '/ToDoList/EditStyle',  
        { id: $('#StyleId').val() },  
        function (data) {  
            $('#divStyle').html(data);  
        });  
});
```

 Quand l'utilisateur clique sur le bouton btnStyle, la méthode jQuery get est appelée pour générer une requête HTTP GET pour la méthode d'action EditStyle dans le contrôleur ToDoList, en lui passant le StyleId obtenu du champ input caché.



Coder la méthode d'action qui traite un clic sur le bouton Style

- 4.□ En haut de la classe ToDoListController, ajoutez du code permettant d'accéder à IStyleService à l'aide de l'injection de dépendances. Le code est semblable à celui qui existe déjà pour IToDoItemService et ICategoryService.

Exercice 6.1 : Afficher un dialogue avec jQuery et Ajax (suite)



Votre code va ressembler à ceci :

```
private ITodoItemService _todoItemService;
private ICategoryService _categoryService;
private IStyleService _styleService;

public TodoListController(ITodoItemService
    todoItemService, ICategoryService categoryService,
    IStyleService styleService)
{
    _todoItemService = todoItemService;
    _categoryService = categoryService;
    _styleService = styleService;
}
```

5.□ Ajoutez une nouvelle méthode d'action appelée *Style* qui reçoit un paramètre entier *id*. Avant l'instruction *return*, ajoutez du code qui :

- Obtient la liste des éléments *Style* depuis la couche de service.
- Construit une *SelectList*, en lui passant la liste des éléments *Style*, et en donnant de bonnes valeurs aux propriétés *value* et *text*. Utilisez également le paramètre passé à la méthode d'action comme valeur par défaut.
- Remplace l'instruction qui renvoie une *View* par une instruction qui renvoie une *PartialView*.



Essayez d'écrire le code vous-même ou utilisez le code suggéré ci-après.

Exercice 6.1 : Afficher un dialogue avec jQuery et Ajax (suite)



Votre code va ressembler à ceci :

```
public ActionResult EditStyle(int id)
{
    var list = _styleService.List();
    ViewBag.StyleList = new SelectList(list, "Id",
    "Name", id);
    return PartialView();
}
```



Ajouter une vue partielle pour afficher la liste des styles

- 6.□ Ajoutez une vue partielle appelée **Style** pour la méthode d'action, sans modèle, dans le dossier Views\ToDoList. Pour cela, cliquez-droit dans la méthode d'action, et sélectionnez **Add | View**.




N'oubliez pas de cocher Create as a partial view Si vous avez oublié de cocher l'option, vous pouvez supprimer le fichier et le recréer, ou bien supprimer toutes les lignes dans le fichier.

Vous devez appeler la vue partielle EditStyle. Visual Studio peut renommer le fichier quand vous cochez la case Create as a partial view.

- 7.□ Dans la vue EditStyle, ajoutez un texte d'invite pour l'utilisateur et affichez la liste des objets Style stockée dans le ViewBag, avec l'aide HTML DropDownList.

Exercice 6.1 : Afficher un dialogue avec jQuery et Ajax (suite)


 *La méthode fortement typée `DropDownListFor` ne présenterait pas d'avantages ici, car la liste sera traitée par du code jQuery dans le navigateur.*



Votre code Razor va ressembler à ceci :

Please select the style:

```
@Html.DropDownList("StyleList")
```

 *Le paramètre passé à la méthode `DropDownList` (`StyleList`) doit être le nom de l'élément de `ViewBag` contenant `SelectList`, comme indiqué dans la méthode d'action `EditStyle` de la classe `ToDoListController`.*

8. ☐ Exécutez votre application. Allez dans la vue `ItemEdit`. Cliquez sur le bouton **Style**.



La vue partielle doit s'afficher sur la page, entre les deux boutons. Pas formidable... Vous allez maintenant afficher la vue partielle dans un dialogue, grâce à jQueryUI.


9. ☐ Dans la section scripts de la vue `ItemEdit`, ajoutez un appel à la méthode jQuery `dialog`, puis exécutez à nouveau l'application.




Votre code jQuery va ressembler à ceci :

```
$('#divStyle').html(data).dialog();
```

Exercice 6.1 : Afficher un dialogue avec jQuery et Ajax (suite)

 *Un dialogue doit maintenant afficher le texte d'invite ainsi qu'une liste des styles présents dans la base de données.*

Il n'est pas encore possible d'enregistrer l'option sélectionnée dans le dialogue. Vous allez ajouter cette fonctionnalité dans les étapes suivantes. Par ailleurs, le dialogue n'est pas modal : Quand il est ouvert, l'utilisateur peut continuer à accéder aux éléments de la page.

 *Pour l'étape suivante, vous pouvez copier le code du fichier Dialog qui se trouve dans le dossier Copy-paste ou bien le taper vous-même.*

Si vous tapez le code, voici un conseil pour l'écriture de code JavaScript : Dès que vous tapez une parenthèse ou une accolade ouvrante, tapez la fermante, puis saisissez votre code entre les deux. Vous pouvez aussi utiliser le raccourci clavier <Ctrl><K>+<Ctrl><D> pour mettre le texte en forme.

- 10.□ Entre les parenthèses de l'appel à dialog, tapez une paire d'accolades. Ajoutez le code suivant pour que le dialogue soit modal, et définir deux boutons OK et Cancel. Les deux boutons doivent fermer le dialogue, et le bouton OK doit enregistrer la valeur de la liste déroulante dans le champ caché StyleId, afin qu'il soit posté au serveur lors de la soumission de la page.

Exercice 6.1 : Afficher un dialogue avec jQuery et Ajax (suite)



Code jQuery :

```
$.get(
    '/ToDoList/EditStyle',
    { id: $('#StyleId').val() },
    function (data) {
        $('#divStyle').html(data).dialog({
            modal: true,
            width: 500,
            buttons: {
                Ok: function () {
                    var id = $('#StyleList').val();
                    $('#StyleId').val(id);
                    $(this).dialog('close');
                },
                Cancel: function () {
                    $(this).dialog('close');
                }
            }
        });
    });
```

- 11.□ Ouvrez la vue ItemDisplay.cshtml. Supprimez la ligne qui définit un titre h2 et celle qui appelle DisplayForModel. Dé-commentez le bloc de lignes suivant.

***i** Les propriétés des styles sont appliquées en tant qu'attributs de style au titre et aux champs du modèle.*

- 12.□ Démarrez l'application. Elle devrait fonctionner comme précédemment, y compris l'enregistrement des éléments avec la valeur du style.



Félicitations ! Vous avez utilisé jQuery avec Ajax pour afficher un dialogue permettant à l'utilisateur de choisir le style des éléments des listes.

Exercice 6.2 : Adapter Todo pour les appareils mobiles

Objectifs

Dans cet exercice, vous allez modifier votre application afin qu'elle soit compatible avec les matériels mobiles. Vous allez créer des styles spécifiques pour les mobiles, ajouter des vues spécifiques et personnaliser une vue pour l'affichage sur un périphérique mobile.



Explorer les paramètres du viewport et définir des règles CSS spécifiques pour les matériels mobiles.

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 6.2-Starting Point.sln et régénérez-la.

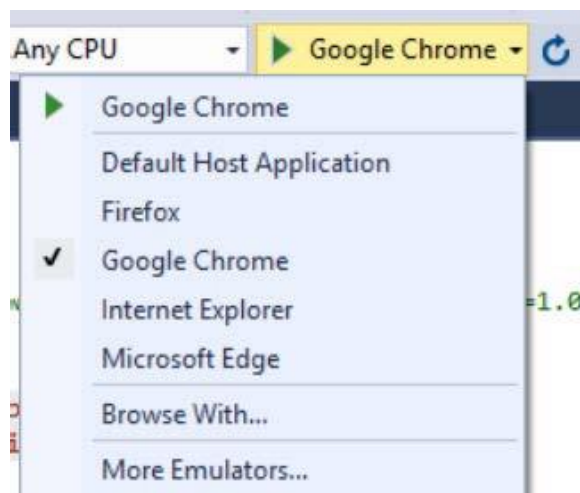


C'est la solution terminée de l'exercice 6.1.




Pour cet exercice, vous allez exécuter l'application Todo dans Chrome car ce dernier est capable d'émuler facilement un appareil mobile.

2. ☐ Sélectionnez **Google Chrome** dans la barre d'outils de Visual Studio pour démarrer l'application dans Chrome.



Exercice 6.2 : Adapter Todo pour les appareils mobiles (suite)

- 3.□ Appuyez sur <F12> dans Chrome pour ouvrir la fenêtre des outils de développement.
- 4.□ En haut à gauche de la fenêtre des outils de développement, cliquez sur l'icône **Toggle device toolbar**.

 L'émulateur de Chrome vous montre à quoi ressemblerait votre application sur un appareil mobile si vous ne donnez aucune instruction au navigateur pour modifier l'aspect des pages. C'est ce que vous allez faire maintenant.

- 5.□ Dans la vue de disposition _Layout.cshtml, supprimez les commentaires de la ligne meta qui définit le viewport, près du haut du fichier.
- 6.□ Enregistrez le fichier _Layout.cshtml et actualisez le navigateur si vous ne l'aviez pas fermé ou bien redémarrez l'application.

 Son apparence est-elle meilleure cette fois-ci ? Quelles différences voyez-vous par rapport au navigateur de bureau ?

- 7.□ Agrandissez la fenêtre de l'émulateur en faisant glisser les poignées qui se trouvent sur le côté, puis examinez le menu.

 Quand la fenêtre est suffisamment large, le menu doit s'afficher normalement.

Exercice 6.2 : Adapter Todo pour les appareils mobiles (suite)



Ce comportement provient de Bootstrap et des classes navbar- qui sont définies dans _Layout.cshtml. Bootstrap utilise beaucoup de sections dans ses fichiers CSS pour ajuster l'apparence des pages aux dimensions du navigateur. Vous allez essayer une section media personnalisée dans les étapes suivantes.*

- 8.□ Ouvrez le fichier Site.css dans le dossier Content du projet Web.



Vous pouvez copier le code des deux étapes suivantes depuis le fichier media du dossier Copy-paste de la solution.

- 9.□ À la fin du fichier, ajoutez une section media pour l'écran et une largeur maximale de 768 pixels, en vous aidant de vos notes de cours.
- 10.□ Dans le bloc media, ajoutez une règle CSS pour afficher le nom de l'application, Todo, en rouge. Cherchez le nom de la classe dans le fichier _Layout.cshtml.



Code CSS :


```
@media only screen and (max-width: 768px) {  
    .navbar-brand {  
        color: red !important;  
    }  
}
```



Notez l'utilisation du mot-clé !important pour redéfinir le comportement de la classe navbar-brand défini par Bootstrap.

Exercice 6.2 : Adapter Todo pour les appareils mobiles (suite)


- 11.□ Actualisez le navigateur ou redémarrez l'application dans Chrome. Appuyez sur <Ctrl><F5> dans le navigateur pour actualiser le cache. Vérifiez que les modifications ont été appliquées.

 *L'en-tête Todo doit être en rouge quand la fenêtre est suffisamment petite.*

- 12.□ Appuyez sur <Maj><F5> dans Visual Studio pour arrêter le débogage.




Créer une vue spécifique pour les mobiles


 *Vous allez maintenant créer une vue spécifiques pour les mobiles afin d'afficher un élément. Le dialogue Add View ne considère pas que ItemDisplay.mobile est un nom valide pour une vue. Vous allez donc créer une vue avec un autre nom, puis la renommer.*

- 13.□ Cliquez droit sur TodoList dans le dossier Views du projet Todo.Web et sélectionnez **Add | View**.
- 14.□ Gardez le nom View par défaut (vous allez le renommer). Pour Template, sélectionnez **Details**, et pour la classe Model, sélectionnez **TodoItemModel**. La seule option de vue sélectionnée devrait être **Use a layout page**. Cliquez sur **Add**.
- 15.□ Dans Solution Explorer, renommez View en **ItemDisplay.mobile.cshtml**.

Exercice 6.2 : Adapter Todo pour les appareils mobiles (suite)

 Vous pouvez taper le code vous-même pour les trois prochaines étapes, ou le copier depuis *ItemDisplayMobile* dans le dossier *Copy-paste*. Si vous choisissez le copier-coller, remplacez tout le contenu du fichier *ItemDisplay.mobile.cshtml* par le contenu copié.

- 16.□ Dans le fichier *ItemDisplay.mobile.cshtml*, supprimez les éléments `<dt>...</dt><dd>...</dd>` pour *CategoryId*, en haut, et pour *StyleId* en bas.
- 17.□ Ouvrez *ItemDisplay.cshtml* (le fichier non-mobile) et copiez le dernier div – celui qui a des appels à *ActionLink* – dans le presse-papiers y compris les balises `<div>` et `</div>`.
- 18.□ De retour dans *ItemDisplay.mobile.cshtml*, supprimez l'élément `<p>...</p>` à la fin du fichier et collez à la place le code copié à l'étape précédente. Remplacez le texte de l'élément `<h4>` par une description (**Mobile** par exemple).
- 19.□ Démarrez l'application dans Chrome. Si besoin, appuyez sur `<F12>` pour afficher la fenêtre des outils de développement. Ouvrez la vue d'affichage d'un élément.

 La vue mobile doit s'afficher. Vous pouvez retourner sur le bureau en cliquant sur le bouton **Toggle Device Toolbar** dans la fenêtre des outils de développement. N'oubliez pas d'appuyer sur `<F5>` pour actualiser le navigateur lorsque vous changez de mode d'affichage.

Exercice 6.2 : Adapter Todo pour les appareils mobiles (suite)



Personnaliser une vue pour les appareils mobiles



Vous allez retirer le bouton Style quand l'application est utilisée sur un appareil mobile.

- 20.□ Dans le fichier ItemEdit.cshtml, cherchez le `<div>...</div>` qui a le bouton Style. Entourez tout l'élément `<div>` d'une instruction if testant si le navigateur n'est pas un mobile.



Votre code va ressembler à ceci :

```
if (!HttpContext.GetOverriddenBrowser().IsMobileDevice)
{
    <div class="form-group">
        <button type="button" id="btnStyle" class="btn btn-info">Style</button>
        <div id="divStyle"></div>
    </div>
}
```

- 21.□ Testez votre application sur un émulateur d'appareil mobile et non mobile.



N'oubliez pas d'actualiser la page lorsque vous changez de mode d'affichage.



Le bouton Style ne doit apparaître que sur un navigateur non mobile.

Exercice 6.2 :
Adapter Todo pour les appareils mobiles
(suite)



Félicitations ! Vous avez modifié votre application afin qu'elle soit compatible avec les matériels mobiles. Vous avez créé des styles spécifiques pour les mobiles, ajouté des vues spécifiques et personnalisé une vue pour l'affichage sur un périphérique mobile.

Exercice facultatif 6.3 : Internationalisation de l'application Todo

Objectifs

Dans cet exercice, vous allez rendre votre application internationale. Pour cela, vous allez déplacer toutes les chaînes de textes dans des ressources et utiliser les ressources traduites quand nécessaire.



Remplacer les chaînes codées en dur par des chaînes définies dans des ressources.


1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 6.3-Starting Point.sln et régénérez-la.



C'est la solution terminée de l'exercice 6.2.

2. ☐ Ajoutez un nouveau projet à la solution. Sélectionnez **Windows** dans Visual C# dans le volet de gauche, et le modèle **Class Library**. Appelez le projet **Todo.Resources**.
3. ☐ Supprimez la Class1 ajoutée par défaut au projet. Ajoutez un nouveau dossier appelé Shared au projet Todo.Resources.
4. ☐ Cliquez droit sur le dossier Shared dans le projet Todo.Resources et sélectionnez **Add | New Item**. Choisissez **General** sous votre langage dans le volet de gauche et **Resource File** comme modèle. Appelez la nouvelle ressource Layout.
5. ☐ Dans la fenêtre de l'éditeur de Layout.resx, sélectionnez **Public** dans la liste déroulante Access Modifier située en haut de la fenêtre.

Exercice facultatif 6.3 : Internationalisation de l'application Todo (suite)

 *La classe générée doit être public pour permettre à d'autres projets d'accéder aux ressources.*

6. ☐ Ajoutez les chaînes suivantes au fichier de ressources :

Nom	Valeur
Todo	To-do
Home	Home
Categories	Categories
Admin	Administration

7. ☐ Cliquez droit sur le dossier References dans le projet Todo.Web et sélectionnez **Add Reference**. Sélectionnez **Solution** dans le volet de gauche et cochez la case Todo.Resources. Cliquez sur **OK**.
8. ☐ Allez dans le fichier _Layout.cshtml dans le dossier Views\Shared. Remplacez le premier paramètre du premier action link affichant le titre To-do par une référence à la propriété de ressource Todo correspondante.



Votre code va ressembler à ceci :

```
@Html.ActionLink(Todo.Resources.Shared.Layout.TODO,
    "Index", "Home", null, new { @class = "navbarbrand"
})
```

9. ☐ Répétez l'étape précédente pour les trois autres action link qui définissent le menu, en utilisant les propriétés correspondantes à chaque fois.

Exercice facultatif 6.3 : Internationalisation de l'application Todo (suite)

10. ☐ Exécutez votre application. Le titre et le menu doivent s'afficher comme précédemment.



Créer une version dans une autre langue

11. ☐ Dupliquez le fichier Layout.resx dans le dossier Shared du projet Todo.Resources et renommez la copie Layout.fr.resx.
12. ☐ Dans le designer de Layout.fr.resx, sélectionnez No Code generation dans la liste déroulante Access Modifier. Modifiez les valeurs de chaque élément en utilisant les traductions suivantes :

Nom	Valeur
Todo	À faire
Home	Accueil
Categories	Catégories
Admin	Administration

13. ☐ Ouvrez le fichier Web.config du projet Todo.Web.




Ouvrez le fichier Web.config situé à la racine du projet, pas celui qui est dans le dossier Views.

14. ☐ Ajoutez la ligne suivante juste sous l'élément <system.web> :

```
<globalization culture="fr" uiCulture="fr" />
```

Exercice facultatif 6.3 : Internationalisation de l'application Todo (suite)

 Cette ligne indique au serveur d'utiliser la culture française. Vous auriez également pu remplacer les deux occurrences de `fr` par `auto` pour indiquer au navigateur d'adopter la culture du système utilisé. Si c'est la culture française qui est configurée sur le serveur, le système utilisera alors les ressources du français.

15. ☐ Démarrez l'application.

 Le menu doit maintenant être en français.



Félicitations ! Vous avez rendu votre application internationale. Pour cela, vous avez déplacé toutes les chaînes de textes dans des ressources et utilisé les ressources traduites quand nécessaire.

Exercice 7.1 : Ajouter des méthodes d'authentification et d'autorisation

Objectifs

Dans cet exercice, vous allez restreindre l'accès à la zone Admin aux utilisateurs authentifiés et personnaliser les données stockées pour chaque utilisateur.



Restreindre l'accès au contrôleur UsersController aux utilisateurs authentifiés

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 7.1-Starting Point.sln et régénérez-la.



C'est la solution terminée de l'exercice 6.2 avant que l'étude de cas ne soit adaptée à l'internationalisation.

Plusieurs classes et vues ont été ajoutées, celles qui font partie du modèle MVC standard quand l'option Individual User Accounts est sélectionnée. Des menus Register et Login ont aussi été ajoutés dans la vue de disposition.

2. ☐ Ouvrez UsersController dans la zone Admin.
3. ☐ Modifiez le contrôleur afin que seuls les utilisateurs connectés aient accès à ses méthodes d'action.



Ajoutez l'attribut Authorize à la classe UsersController.



Code C# :

```
[Authorize]  
public class UsersController : Controller
```

Exercice 7.1 :
Ajouter des méthodes d'authentification et d'autorisation
(suite)



À la prochaine étape, *ne cliquez pas* sur Register, ni dans le menu ni sur la page de connexion, car cela créerait la base de données identity. Vous allez d'abord personnaliser les données à stocker pour chaque utilisateur avant de créer cette base.

- 4.□ Démarrez l'application et essayez d'ouvrir la page d'administration.



Vous devriez arriver sur la page de connexion.



Quelle est l'URL de la page de connexion ?



Notez la valeur du paramètre returnUrl (%2F est la valeur du caractère \ encodé).



Comment MVC sait-il quelle méthode d'action appeler pour la page de connexion ?



Regardez le code du fichier Startup.Auth.cs dans le dossier App_Start.

Exercice 7.1 : Ajouter des méthodes d'authentification et d'autorisation (suite)



Ajouter des données utilisateur personnalisées

***i** Les utilisateurs doivent saisir un nom et un mot de passe pour se connecter mais vous pouvez personnaliser cette étape. Vous allez ajouter un nouveau champ, **HomeCity**, qui sera enregistré avec l'identité de l'utilisateur.*

- 5.□ Ouvrez le fichier IdentityModels.cs dans le dossier Models du projet Todo.Web.

***i** Ce fichier renferme la définition de deux classes que vous pouvez personnaliser. Vous pouvez enregistrer des données personnalisées pour chaque utilisateur en ajoutant de nouvelles propriétés à la classe **ApplicationUser**.*

- 6.□ Ajoutez une propriété appelée **HomeCity** de type string à la classe **ApplicationUser**.



Votre code va ressembler à ceci :

```
public class ApplicationUser : IdentityUser
{
    public string HomeCity { get; set; }
    Code omitted...
}
```

- 7.□ Ajoutez la même propriété dans classe **RegisterViewModel** du fichier **AccountViewModel.cs** dans le dossier Models.

Exercice 7.1 : Ajouter des méthodes d'authentification et d'autorisation (suite)

- 8.□ Dans le fichier AccountController.cs, cherchez la méthode Register qui a un attribut HttpPost. Initialisez la propriété HomeCity dans l'initialiseur d'objet de la classe ApplicationUser, en lui donnant une valeur à partir du model.



Votre code va ressembler à ceci :

```
var user = new ApplicationUser { UserName =  
    model.Email, Email = model.Email, HomeCity =  
    model.HomeCity };
```

- 9.□ Dans la vue Register.cshtml, située dans le dossier Views\Account, dupliquez le <div> form-group de Email. Dans le <div> dupliqué, remplacez Email par **HomeCity** deux fois.




Votre code va ressembler à ceci :


```
<div class="form-group">  
    @Html.LabelFor(m => m.HomeCity, new { @class = "col-md-2 control-label" })  
<div class="col-md-10">  
    @Html.TextBoxFor(m => m.HomeCity, new { @class = "form-control" })  
    </div>  
</div>
```

Exercice 7.1 : Ajouter des méthodes d'authentification et d'autorisation (suite)

- 10.□ Démarrez l'application, inscrivez-vous et créez un compte.

 *Vous devez maintenant indiquer votre ville natale en plus du nom et du mot de passe.
Patientez quelques instants pendant la création de la base de données, puis vérifiez que vous êtes connecté.*

- 11.□ Cliquez sur votre adresse e-mail dans le menu.

 *Vous pouvez changer le mot de passe.*


- 12.□ Appuyez sur <Shift><F5> dans Visual Studio pour arrêter le débogage.




Explorer la base de données utilisée pour l'authentification

- 13.□ Dans Visual Studio, cliquez sur **View | Server Explorer**.
- 14.□ Dans la fenêtre Server Explorer, cliquez droit sur le nœud Data Connections et sélectionnez **Add Connection**.
- 15.□ Dans le dialogue Add Connection, saisissez **.\SQLExpress** dans Server name et sélectionnez **DefaultConnection** dans la liste déroulante Connect to a database. Cliquez sur **OK**.
- 16.□ Dans Server Explorer, développez la nouvelle collection et le dossier **Tables**. Cliquez droit sur la table **AspNetUsers** et sélectionnez **Show Table Data**.
- 17.□ Cliquez sur le bouton **Show all files** dans la barre d'outils de Solution Explorer.

Exercice 7.1 :
Ajouter des méthodes d'authentification et d'autorisation
(suite)

 *Vous devriez voir une ligne contenant l'adresse e-mail et la ville natale que vous venez de saisir.*

 *Comment fait MVC pour savoir quel nom donner à la base de données ?*



Félicitations ! Vous avez restreint l'accès à la zone Admin aux utilisateurs authentifiés et personnalisé les données stockées pour chaque utilisateur.

Exercice 7.2 : Créer un contrôleur Web API

Objectifs

Dans cet exercice, vous allez ajouter un contrôleur Web API à votre application. Vous allez appeler des services depuis le navigateur, depuis Fiddler et depuis une application cliente WPF.



Créer le contrôleur Web API et y ajouter des méthodes d'action.

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 7.2-Starting Point.sln et régénérez-la.

***i** C'est la solution terminée de l'exercice 7.1 (avant l'ajout du projet de test), légèrement modifiée. Les modifications permettent l'utilisation de l'injection de dépendances avec Ninject dans un contrôleur Web API.*

Les modifications faites ne sont pas dans les objectifs de ce cours. Si vous avez besoin de les répliquer, regardez la classe NinjectResolver dans le dossier App_Start du projet Web. La classe NinjectConfig a également été légèrement modifiée afin d'interagir avec la nouvelle classe.

Un nouveau dossier ApiControllers a aussi été ajouté au projet Todo.Web.

2. ☐ Cliquez droit sur le dossier ApiControllers du projet Todo.Web dans Solution Explorer et sélectionnez **Add | Controller** pour ajouter un nouveau contrôleur.
3. ☐ Sélectionnez **Web API 2 Controller – Empty**, cliquez sur **Add** et appelez le contrôleur CategoryApiController.

Exercice 7.2 : Créer un contrôleur Web API (suite)



Notez que le contrôleur dérive de la classe ApiController.

- 4.□ Déclarez une variable de type `ICategoryService` appelée `-categoryService` au niveau de la classe.



N'oubliez pas d'ajouter l'instruction `using` en cliquant sur l'ampoule dans Visual Studio.

- 5.□ Ajoutez un constructeur à la classe et initialisez la variable que vous venez de définir avec un paramètre de même type passé au constructeur.



Il s'agit du constructeur avec injection de dépendance standard qui a été ajouté au projet dans le chapitre 3. Vous pouvez copier le code depuis `CategoryController` si vous le souhaitez.




Le code des deux dernières étapes va ressembler à ceci :

```
public class CategoryApiController : ApiController
{
    private ICategoryService _categoryService;
    public CategoryApiController(ICategoryService
categoryService)
    {
        _categoryService = categoryService;
    }
}
```


Exercice 7.2 : Créer un contrôleur Web API (suite)


- 6.□ Ajoutez une méthode d'action appelée **GetList** à la classe **CategoryApiController**. La méthode ne reçoit pas de paramètre et renvoie une liste d'objets **CategoryModel**. Résolvez à nouveau le using manquant.

 *Obtenez la liste grâce au service.*


 *Votre code va ressembler à ceci :*

```
public List<CategoryModel> GetList()
{
    return _categoryService.List();
}
```

- 7.□ Ouvrez l'application dans Edge, Firefox ou Chrome.

 *Internet Explorer n'affiche pas les données renvoyées au format XML ou JSON, comme le font Edge, Firefox ou Chrome à l'étape suivante. Il propose à la place de les enregistrer dans un fichier, ce qui n'est pas aussi pratique quand on développe.*

- 8.□ Ajoutez **/api/CategoryApi** à l'URL dans la barre d'adresse afin d'obtenir `http://localhost:5555/api/CategoryAPI`.

 *Les données renvoyées doivent s'afficher en XML ou JSON dans le navigateur. Celui-ci n'accepte que ces formats. Vous allez modifier cela plus tard.*

Exercice 7.2 : Créer un contrôleur Web API (suite)

- 9.□ Arrêtez le débogage et ajoutez à la classe `CategoryApiController` une deuxième méthode appelée **GetOne** avec un seul paramètre entier appelé `id`. Écrivez le code dans la classe pour obtenir le `CategoryModel` correspondant à la valeur de `id`.



Votre code va ressembler à ceci :

```
public CategoryModel GetOne(int id)
{
    return _categoryService.Read(id);
}
```

- 10.□ Générez votre application et testez la méthode dans le navigateur en ajoutant un nombre à l'URL, par exemple `http://localhost:5555/api/CategoryApi/2`.



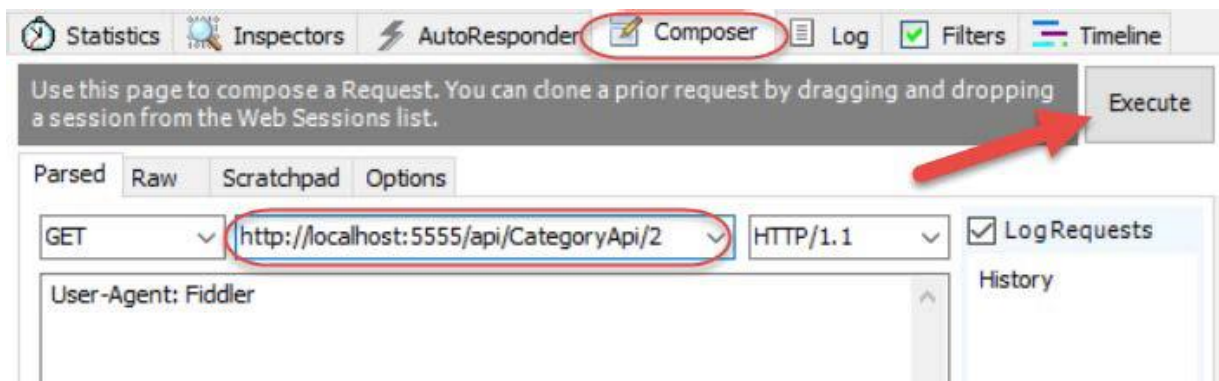
Changer le format des données renvoyées avec Fiddler



Laissez l'application ouverte pour les prochaines étapes ou redémarrez-la si vous l'aviez fermée.

Exercice 7.2 : Créer un contrôleur Web API (suite)

11. □ Démarrez Fiddler. Cliquez sur l'onglet **Composer** dans le volet de droite. Copiez l'URL dans la barre d'adresse du navigateur et collez-la dans la zone de texte en haut du volet Composer. Cliquez sur **Execute**.



12. □ Double cliquez sur la demande dans le volet de gauche.

? *Quel est le format des données renvoyées ? _____*

13. □ De retour dans l'onglet Composer, ajoutez la ligne suivante à l'entête de la requête :

Accept: application/xml

14. □ Cliquez sur **Execute** à nouveau puis double cliquez sur la demande dans le volet de gauche.

? *Quel est le format des données renvoyées ? _____*

Exercice 7.2 : Créer un contrôleur Web API (suite)



Accéder au service Web API depuis une application cliente



Vous allez appeler votre service Web API depuis une application cliente Windows. Celle que vous utiliserez a été écrite avec WPF, mais le même code pourrait s'appliquer à une application console, une application Windows Forms ou une application Windows Store.

- 15.□ Ouvrez la solution Ex 7.2-Starting Point-WpfClient située dans C:\977\cs\Exercices\Ex 7.2-Starting Point\WpfClient, dans une nouvelle instance de Visual Studio et régénérez-la.



La solution WpfClient doit être ouverte dans une nouvelle instance de Visual Studio et la solution Todo dans Visual Studio.

- 16.□ Double cliquez sur MainWindow.xaml dans Solution Explorer. Cliquez droit sur la surface du designer et sélectionnez **View Code**.



Observez l'appel au service Web API et l'URL utilisée.



Ne fermez surtout pas la solution Todo pour l'étape suivante.

- 17.□ Exécutez l'application et cliquez sur le bouton.



La liste des catégories s'affiche.

Exercice 7.2 : Créer un contrôleur Web API (suite)



Cette application utilise la bibliothèque Json.NET installée avec NuGet. Elle offre de nombreuses fonctionnalités pour traiter des données JSON dans une application .NET.



Félicitations ! Vous avez ajouté un contrôleur Web API à votre application. Vous avez appelé des services depuis le navigateur, depuis Fiddler et depuis une application cliente WPF.

Exercice 8.1 : Déployer l'application

Objectifs

Dans cet exercice, vous allez préparer le déploiement de votre application et la déployer dans votre serveur Web IIS local.



Publier l'application dans un package Web Deploy.

***i** On peut déployer une application directement depuis Visual Studio vers un serveur Web IIS. Mais cela requiert des droits d'administration et n'est pas la configuration habituelle dans les entreprises.*

Dans la plupart des entreprises, le développeur Web crée un package de déploiement et le transmet à un administrateur IIS, qui l'importe dans le serveur Web IIS. Vous allez effectuer ces deux étapes ici.

1. ☐ Ouvrez la solution du point de départ de l'exercice Ex 8.1-Starting Point.sln et régénérez-la.

***i** C'est la solution terminée de l'exercice 7.2.*

2. ☐ Développez le nœud Web.config du projet Todo.Web dans Solution Explorer et ouvrez le fichier Web.Release.config. Lisez les commentaires.

***i** C'est un fichier de transformations utilisé pendant le processus de génération. Le fichier par défaut ne fait que retirer l'attribut debug du fichier Web.config. Vous pourriez y définir les chaînes de connexion à utiliser en production.*

3. ☐ Cliquez droit sur le projet Todo.Web et sélectionnez **Publish**.

Exercice 8.1 : Déployer l'application (suite)

- 4. ☐ Dans l'onglet Profile, sélectionnez **Custom** dans la liste déroulante. Appelez le nouveau profile **Prod**. Cliquez sur **OK** pour aller dans l'onglet Connection.
- 5. ☐ Sélectionnez **Web Deploy Package** dans la liste Publish method.



Vous devez sélectionner Web Deploy Package et non l'élément Web Deploy sélectionné par défaut.

- 6. ☐ Sélectionnez **C:\977\Deploy\Todo.Web.zip** dans Package location et saisissez **Todo** dans Site name. Cliquez sur **Next**.
- 7. ☐ Ne modifiez aucune valeur dans le volet Settings. La configuration doit déjà être Release.



La section Databases liste les chaînes de connexion trouvées dans le fichier Web.config. Vous pourriez modifier les chaînes de connexion utilisées dans un environnement de production, ou même déployer une base de données avec le package.

- 8. ☐ Cliquez sur **Next**, puis sur **Publish**.



Un package de déploiement est créé dans C:\977\Deploy.

Exercice 8.1 : Déployer l'application (suite)



Importer le package de déploiement dans IIS

- 9.□ Cliquez droit sur l'icône de l'invite de commande dans la barre des tâches, puis sur l'option de menu **Developer Command Prompt for VS...** Sélectionnez **Run as administrator** et cliquez sur **Yes** dans le dialogue User Account Control.
- 10.□ Dans l'invite de commande, saisissez **cd c:\977\deploy** et appuyez sur <Entrée>.
- 11.□ Saisissez **toto** et appuyez sur la touche <Tab>. La commande suivante doit s'afficher : **Todo.Web.deploy.cmd**. Ajoutez **/Y** à la fin de la commande : **Todo.Web.deploy.cmd /Y**. Appuyez sur <Entrée>.
- 12.□ Ouvrez un navigateur, saisissez **localhost** dans la barre d'adresse et appuyez sur <Entrée>. Vous devriez être dirigé vers l'application Todo ouverte dans IIS.



Supprimez le numéro de port dans l'adresse si elle a été complétée par saisie automatique.



Votre application est maintenant déployée sur le serveur Web IIS.



Félicitations ! Vous avez préparé le déploiement de votre application et l'avez déployée dans votre serveur Web IIS local.