

Ce TD et les suivants utilisent tous la solution que nous allons créer.

Pensez à bien enregistrer votre projet de manière à pouvoir le retrouver lors des prochaines séances.

1 Présentation

ASP.NET est un framework web gratuit permettant de créer des sites web et applications web en utilisant HTML, CSS et JavaScript. Vous pouvez également créer des API web et utiliser des technologies en temps réel telles que Web Sockets.

ASP.NET offre deux frameworks principaux pour la création d'applications Web : Web Forms et MVC. Les deux frameworks sont stables et matures, et vous pouvez créer des applications web avec n'importe lequel d'entre eux.

1.1 Web Forms

Avec ASP.NET Web Forms, vous pouvez créer des sites Web dynamiques à l'aide d'un modèle de glisser-déplacer familier, piloté par les événements. Une aire de conception et des centaines de contrôles et de composants vous permettent de créer rapidement des sites sophistiqués et puissants, gérés par interface utilisateur avec accès aux données.

Les avantages d'une application web basée sur Web Forms :

- Ils fournissent un modèle d'événements, qui préserve l'état entre deux requêtes, ce qui permet d'accélérer le développement d'applications Web d'entreprise. Les formulaires Web fournissent des dizaines d'événements qui sont pris en charge dans des centaines de contrôles serveur.
- Ils utilisent le ViewState et des pages basées sur les contrôles serveurs, ce qui peut rendre plus facile la gestion des informations.
- Ils fonctionnent bien pour de petites équipes de développeurs et designers qui veulent profiter du grand nombre de composants disponibles pour développer rapidement des applications.
- En général, ils sont moins complexes pour le développement d'applications, car les composants (la page, les contrôles, etc...) sont bien intégrés et nécessitent généralement moins de code que le modèle MVC.

Web Forms est plus facile à apprendre et permet de créer rapidement des écrans visuels grâce au drag & drop de composants. Le framework est intéressant pour débiter avec le Web quand on connaît déjà Win Forms ou WPF.

1.2 MVC

ASP.NET MVC est un outil puissant, basé sur les modèles, qui vous permet de générer des sites web en séparant de manière claire et précise tous les aspects associés, et propose un contrôle total sur le balisage, ce qui permet de bénéficier d'un déploiement flexible et optimal. ASP.NET MVC inclut de nombreuses fonctionnalités qui permettent de bénéficier d'un développement TDD rapide et convivial pour la création d'applications sophistiquées qui utilisent les normes web les plus récentes.

Les avantages d'une application web basée sur MVC :

- Elle rend plus facile à gérer la complexité en divisant l'application en un modèle, une vue et un contrôleur.

- Il n'utilise pas le ViewState, ou les contrôles serveurs. Cela rend le framework MVC idéal pour les développeurs qui veulent un contrôle total sur le comportement de l'application.
- Il utilise un pattern contrôleur frontal (Front Controller), qui traite les requêtes de l'application web par l'intermédiaire d'un contrôleur unique. Cela vous permet de concevoir une application qui prend en charge une infrastructure de routage riche. ASP.NET MVC ne considère pas une URL comme un point de terminaison vers un fichier d'une application. Une URL est considérée comme un moyen d'accéder à une ressource (logique) sur le serveur, mais pas nécessairement un fichier ASPX à exécuter. Nous aurons alors l'occasion de voir que les URLs possèdent un format particulier de type /contrôleur/action/id.
- Il offre un meilleur support pour le développement dirigé par les tests (Test Driven Development). Lors de la création d'un projet de type ASP .NET MVC, Microsoft propose de créer un projet de test, qui permettra de tester au fur et à mesure du développement de l'application.
- Il fonctionne bien pour des applications Web développées et soutenues par de grandes équipes de développeurs et de designers, qui ont besoin d'un degré élevé de contrôle sur le comportement des applications.

Le temps d'apprentissage du MVC est un peu plus long car il faut bien comprendre les concepts du framework. Une fois que vous aurez bien compris ces concepts, vous pourrez développer très rapidement des applications web modernes.

2 Introduction au MVC

2.1 Création du projet

Créer un nouveau projet Web que vous nommerez « *SondageSoiree* », la solution sera « *OrganisationSoiree* ».

Choisir comme type de projet : **Application Web ASP.NET (.NET Framework)** en **C#** :

The screenshot shows the 'Configure your new project' dialog box. At the top, it says 'Application web ASP.NET (.NET Framework)' with tabs for 'C#', 'Windows', 'Cloud', and 'Web'. The 'Web' tab is selected. Below this, there are fields for 'Nom du projet' (SondageSoiree), 'Emplacement' (D:\Projets\IUT\OS12 - ASP\), and 'Nom de la solution' (OrganisationSoiree). There is a checkbox for 'Placer la solution et le projet dans le même répertoire' which is unchecked. The 'Framework' dropdown is set to '.NET Framework 4.8'. At the bottom right, there are 'Retour' and 'Créer' buttons.

Sélectionner un modèle MVC, avec pour authentification « Comptes d'utilisateurs individuels » et ajouter les tests unitaires « *OrganisationSoiree.Tests* » :

The screenshot shows the 'Créer une application web ASP.NET' dialog box. On the left, there are five project templates: 'Vide', 'Web Forms', 'MVC' (which is highlighted in blue), 'API web', and 'Application avec une seule page'. On the right, there are two sections: 'Authentification' with 'Aucune authentification' selected and a 'Changer' link; and 'Ajouter des dossiers et des références de base' with checkboxes for 'Web Forms' (unchecked), 'MVC' (checked), and 'API web' (unchecked). Below this, the 'Avancé' section has checkboxes for 'Configurer pour HTTPS' (checked), 'Prise en charge de Docker' (unchecked, with a note '(Nécessite Docker Desktop)'), and 'Créer également un projet pour les tests unitaires' (checked). The 'OrganisationSoiree.Tests' project name is entered in the field below. At the bottom right, there are 'Retour' and 'Créer' buttons.

Notre application ASP.NET MVC possède une structure particulière, que nous allons devoir respecter. Elle est constituée :

- D'un répertoire Content, qui contiendra les éléments graphiques et de présentation de notre application : feuilles de styles CSS et XSLT, images...
- D'un répertoire Controllers, qui contiendra tous les contrôleurs de notre application.
- D'un répertoire Models, qui contiendra les classes qui constituent le modèle de l'application ASP.NET MVC.
- D'un répertoire Scripts, contenant les scripts JavaScript de l'application. Par défaut, ce répertoire contient les fichiers JavaScript composant les Frameworks JQuery et Ajax spécifiques pour ASP.NET MVC.
- Le fichier global.asax définit le routage par défaut de notre application. Dans les applications ASP.NET MVC, Microsoft l'utilise pour permettre d'exécuter une action d'un contrôleur à partir d'une URL. Il est aussi possible de personnaliser le routage en définissant ses propres règles. Par exemple, soit l'URL suivante : *http://www.monsite.fr/UnContrôleur/UneAction/UnParametre*. Une route correspondant à cette URL est définie dans la méthode RegisterRoutes() appelée dans le fichier Global.asax (la méthode est définie dans ~/App_Start/RouteConfig.cs).
- D'un répertoire Views, contenant toutes les vues. Lors de leur création nous auront l'occasion de définir les caractéristiques suivantes :
 - Le type de la vue : vues pages (pages ASP.NET à part entière), vues de contenu (pages s'exécutant au sein d'une Master Page) et vues partielles (aussi appelée vues contrôles utilisateurs Web).
 - S'il s'agit d'une vue typée ou non typée. Une vue typée permet de gérer un objet ou une collection d'objets d'un type particulier, par exemple un type du modèle de l'application.
 - Le mode de gestion, permettant de définir le comportement de la vue en fonction de son rôle : « List » pour afficher le contenu d'une collection d'objets, « Detail » pour consulter l'ensemble des propriétés d'un objet, « Edit » pour modifier l'ensemble des propriétés d'un objet, ...
 - Il contient un répertoire Shared, dont le rôle est particulier au sein de l'application. Comme son nom l'indique, il contient des vues qui sont partagées entre tous les contrôleurs de l'application. Lorsqu'un contrôleur souhaite exécuter une vue, le processus de recherche d'une vue est le suivant :
 - La vue (page ou contrôle utilisateur) est d'abord recherchée dans le répertoire ~/Views/<nomContrôleur>, où le nom du contrôleur est celui du contrôleur ayant demandé l'exécution de la vue.
 - Si cette vue n'est pas trouvée, alors elle est recherchée dans le répertoire ~/Views/Shared.

2.2 Le Routing

Lorsque vous construisez une application traditionnelle ASP.NET (WebForms), il y a une correspondance entre une URL et une page.

Si vous demandez la page `SomePage.aspx` au serveur, alors il doit forcément exister une page sur le disque qui soit nommée `SomePage.aspx`.

Si la page n'existe pas alors vous obtenez une erreur « 404 - Page Not Found ».

Au contraire, sur une application MVC, une URL correspond à une action contrôleur, il n'y a donc aucune correspondance entre l'URL, que vous saisissez dans votre navigateur Web et les fichiers que vous trouvez dans votre application.

Dans une application traditionnelle ASP.NET, les requêtes navigateurs sont mappées vers des pages. Ainsi, une application ASP.NET est dite « centrée sur le contenu » (content-centric) alors qu'une application MVC est dite « centrée logiquement » (logic centric).

Une requête navigateur est liée à une action contrôleur à travers une fonctionnalité du framework ASP.NET nommé le routage ASP.NET. Le routage ASP.NET est utilisé par le framework MVC pour router les requêtes vers les actions des contrôleurs.

Le routage ASP.NET dans le framework MVC, utilise une table de routage pour gérer les requêtes entrantes. La table de route est créée lorsque votre application démarre pour la première fois.

Cette table est configurée dans le fichier `~/App_Start/RouteConfig.cs`.

Lorsqu'une application MVC démarre pour la première fois, la méthode `Application_Start()` du fichier `Global.asax` est appelée. Cette méthode appelle alors la méthode `RegisterRoutes()` qui crée la table de routage.

2.2.1 Routage par défaut

La table de routage par défaut consiste en une seule et unique route.

Cette route par défaut découpe toutes les requêtes entrantes en trois segments (un segment URL est tout ce qui se trouve entre les slashes).

Le premier segment est mappé sur le nom du contrôleur, le second sur le nom de l'action et le segment final est mappé sur un paramètre passé à l'action et nommé `Id`.

Dans votre projet la route par défaut doit être :

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

La route `Default` inclut des valeurs par défaut pour les trois paramètres.

Si vous ne fournissez pas un contrôleur, alors la valeur par défaut sera prise à savoir `Home`. Pour l'action il s'agira d'`Index` et pour le paramètre `Id`, il s'agira d'une chaîne vide.

1. Exécuter l'application. Que remarquez-vous dans l'URL de la page d'accueil ?
2. Ajouter `/Home` à l'URL et recharger la page. Que se passe-t-il ?
3. Ajouter `/Index` à l'URL et recharger la page. Que se passe-t-il ?
4. Ajouter `/1` à l'URL et recharger la page. Que se passe-t-il ?
5. Modifier la signature de la méthode `Index()` en `Index(int id)` dans le fichier `Controllers/HomeController.cs` et exécuter l'application. Que remarquez-vous ?

Comme vous avez pu le voir, la route par défaut correspond à l'action `Index` dans le contrôleur `HomeController`. Tant que la méthode `Index()` n'attend pas de paramètre, l'identifiant dans l'URL suivante

est ignoré : /Home/Index/1. Si la méthode Index() attend un paramètre, il devient obligatoire de renseigner cette information dans l'url, à moins que le paramètre soit de type Nullable<int> ou int?.

2.2.2 Route personnalisée

Pour une application simple, la route par défaut devrait suffire. Pour une application plus complexe, vous pourriez découvrir que vous avez des besoins de routes spécialisées.

Dans le cas de création d'un blog, vous pouvez avoir besoin d'accéder directement à des url de ce type : /Photo/31-01-2016.

Pour prendre en charge ce type de requête, vous devez créer une route personnalisée :

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );

    routes.MapRoute(
        name: "Blog",
        url: "Photo/{date}",
        defaults: new { controller = "Photo", action = "AfficherResultat" }
    );
}
```

1. Créer la route ci-dessus.
2. Ajouter au répertoire Controllers le contrôleur PhotoController.cs et intégrer la méthode suivante :

```
public string AfficherResultat(string date)
{
    DateTime d = DateTime.ParseExact(date, "dd-MM-yyyy", CultureInfo.InvariantCulture);
    return "Il n'existe pas de photo à cette date " + d.ToString();
}
```

3. Exécuter l'application et ajouter à l'URL /Photo/31-01-2016 et recharger la page. Que constatez-vous ?
4. Dans la méthode RegisterRoutes(), placer la route *Blog* avant la route *Default*. Réaliser à nouveau l'étape précédente. Que constatez-vous ?
5. Modifie la date de l'URL par *test* et recharger la page. Que se passe-t-il ?

L'ordre des routes que vous ajoutez à la table des routes est important. Notre nouvelle route personnalisée est ajoutée avant la route par défaut. Si vous inversez l'ordre alors la route par défaut sera toujours appelée à la place de la route personnalisée.

La route personnalisée *Blog*, map les requêtes entrantes vers un contrôleur nommé *Photo* et invoque l'action AfficherResultat(). Lorsque la méthode AfficherResultat() est appelée, une date d'entrée date est attendue en tant que paramètre. Le paramètre attendu est de type DateTime. Le framework MVC est capable de convertir si c'est possible la valeur dans le bon type, sinon il retournera une erreur.

2.2.3 Contrainte de route

Dans la route personnalisée vue précédemment, nous avons pu remarquer que l'action appelée attend un paramètre de type `DateTime`. Si le paramètre dans l'url n'est pas de ce type, la page appelée retourne une erreur concernant un problème au niveau du type du paramètre. Une erreur qui n'est pas très belle à voir. Nous pouvons éviter ce genre d'erreur en utilisant des contraintes de route.

1. Modifier la route avec une expression régulière :

```
routes.MapRoute(  
    name: "Blog",  
    url: "Photo/{date}",  
    defaults: new { controller = "Photo", action = "AfficherResultat" },  
    constraints: new { date = @"(\d{1,2})-(\d{1,2})-(\d{4})" }  
);
```

2. Exécuter l'application et ajouter à l'URL /Photo/test et recharger la page. Que constatez-vous ?

3 Notre application MVC

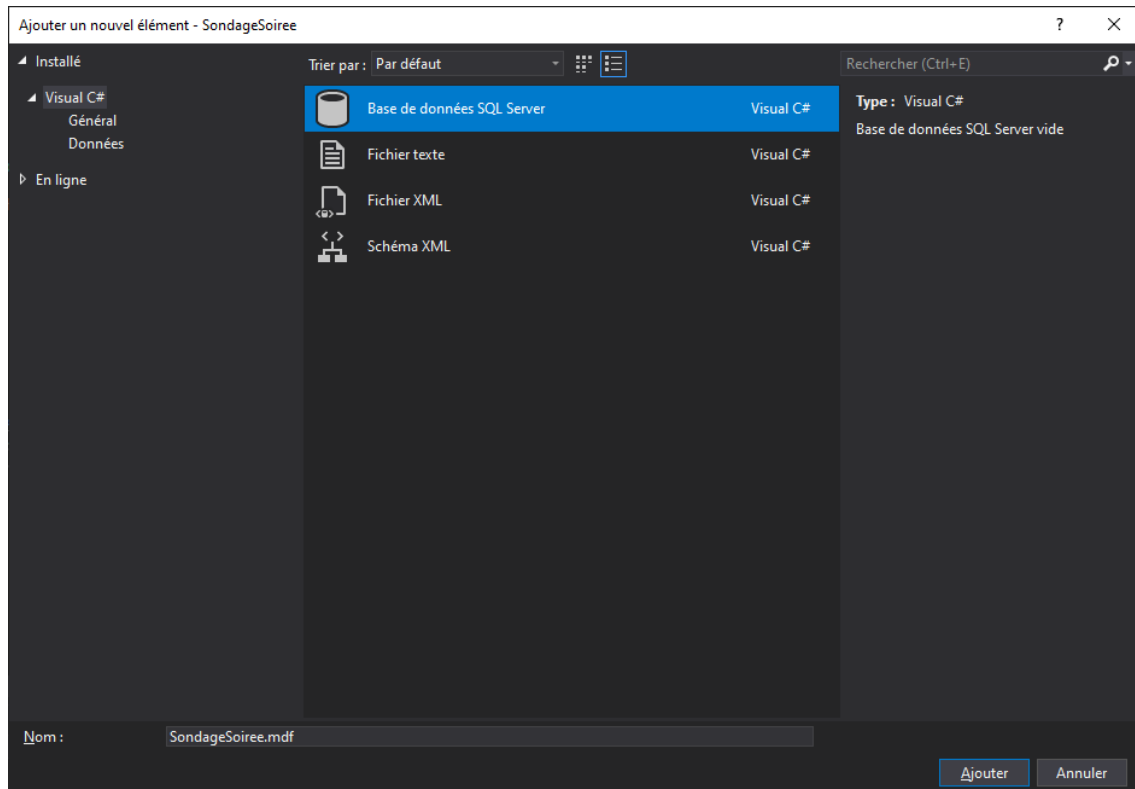
Nous allons créer une application permettant d'organiser un repas de fin d'année pour votre département. Cette application permettrait à tous les étudiants du département Informatique de voter pour des restaurants parmi une liste définie qui peut être enrichie par tous les utilisateurs.

La soirée se déroulera dans le restaurant qui récoltera le plus de vote. Le vote réalisé, il sera impossible de le modifier.

3.1 Définition du modèle

3.1.1 Génération du modèle

- Dans le dossier App_Data du projet, ajouter une BDD locale « SondageSoiree.mdf » :



- Exécuter le script SondageSoiree.sql dans votre base de données

Comme vu dans le TD1, nous allons générer un modèle de données Entity Framework. Créer un dossier nommé « Entity » dans le dossier Models. C'est dans ce dossier que nous allons générer le modèle.

- Ajouter un nouvel élément « ADO.NET Entity Data Model » avec pour nom « SoireeContext »
- Choisir « EF Designer à partir de la base de données »

- Choisir d'enregistrer les paramètres dans Web.Config en tant que : « SoireeContext » :

Assistant EDM

Choisir votre connexion de données

Quelle connexion de données votre application doit-elle utiliser pour établir une connexion à la base de données ?

SondageSoiree.mdf

Nouvelle connexion...

Cette chaîne de connexion semble contenir des données sensibles (par exemple, un mot de passe), lesquelles sont indispensables pour établir une connexion à la base de données. Le stockage des données sensibles dans la chaîne de connexion peut entraîner un risque de sécurité. Voulez-vous inclure les données sensibles dans la chaîne de connexion ?

☐ Non, exclure les données sensibles de la chaîne de connexion. Je définirai ces informations dans le code de mon application.

☐ Oui, inclure les données sensibles dans la chaîne de connexion.

Chaîne de connexion :

```
metadata=res://*/Models.Entity.SoireeContext.csdl|res://*/Models.Entity.SoireeContext.ssdl|
res://*/Models.Entity.SoireeContext.msl;provider=System.Data.SqlClient;provider connection
string="data source=(LocalDB)\MSSQLLocalDB;attachdbfilename=|DataDirectory|
\SondageSoiree.mdf;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"
```

☒ Enregistrer les paramètres de connexion dans Web.Config en tant que :

SoireeContext

< Précédent Suivant > Terminer Annuler

- Choisir les tables « Eleve », « Restaurant », « Sondage » et « Vote »
- Choisir l'option « Mettre au pluriel ou au singulier les noms d'objets générés »
- Mettre « *SondageSoiree.Models.Entity* » en espace de nom

Assistant EDM

Choisir vos paramètres et objets de base de données

Quels objets de base de données voulez-vous inclure dans votre modèle ?

☒ Tables

☒ dbo

☒ Eleve

☒ Restaurant

☒ Sondage

☒ Vote

☐ Vues

☐ Procédures et fonctions stockées

☒ Mettre au pluriel ou au singulier les noms d'objets générés

☒ Inclure les colonnes de clés étrangères dans le modèle

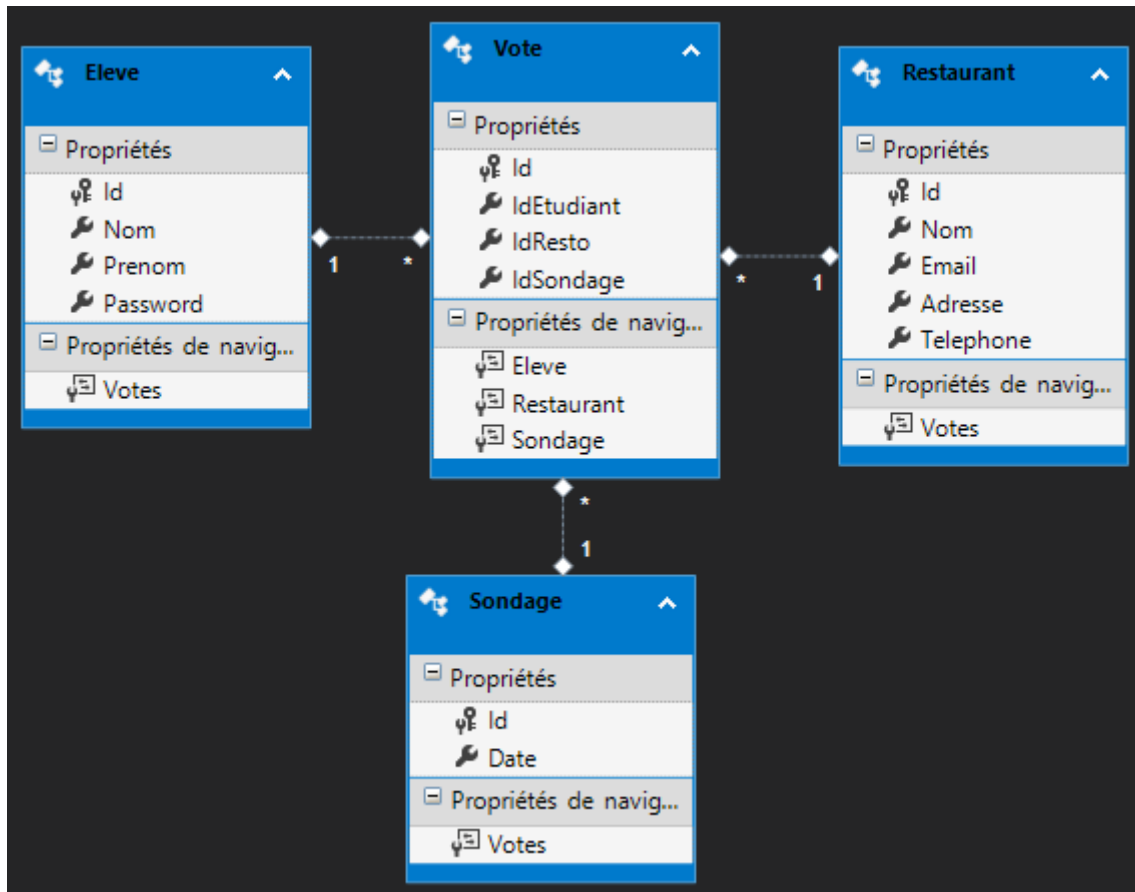
☐ Importer les fonctions et les procédures stockées sélectionnées dans le modèle d'entité

Espace de noms du modèle :

SondageSoiree.Models.Entity

< Précédent Suivant > Terminer Annuler

Le modèle généré devrait ressembler à :



Nous allons rajouter une classe *Resultat* (qui sera calculée).

- Créer une classe « *Resultat* » dans le dossier *Models* avec 2 propriétés :
 - Nom : string
 - NbVote : int

Le nom correspond au nom du restaurant.

3.1.2 Création de la couche d'accès aux données

Nous allons créer une couche d'accès aux données, appelée Data Access Layer, qui a pour but de faciliter l'accès aux données, de créer un point d'entrée unique, et éventuellement de changer le type d'accès aux données sans perturber le reste du programme. Notre point d'entrée sera la classe *SoireeContext*.

- Créer un dossier « *DAL* » dans le dossier *Models*
- Définir le contrat du Data Access Layer dans le répertoire *Models*, appelé « *IDa1* » : Interface contenant les différentes méthodes dont nous allons avoir besoin. Cette interface dérive de l'interface *IDisposable* (définit une méthode *Dispose()* pour libérer des ressources allouées). (Le code de l'interface se trouve dans le fichier *IDa1.cs*).
- Créer une classe « *Dal* » qui implémente l'interface *IDa1*. Cette classe est composée :
 - D'un constructeur (rappel création constructeur : taper *ctor* + 2 fois tabulation) qui instancie une variable privée de la classe *SoireeContext*. Cela permet d'accéder aux différents éléments de la base de données.
 - D'une méthode *Dispose()* qui utilise la méthode *Dispose()* de la classe *SoireeContext*. Cette méthode permet de supprimer notre objet *SoireeContext* dès que nous avons fini tous les traitements.
 - Dans toutes les méthodes du contrat *IDa1*, veuillez laisser pour le moment toutes ces méthodes vides.

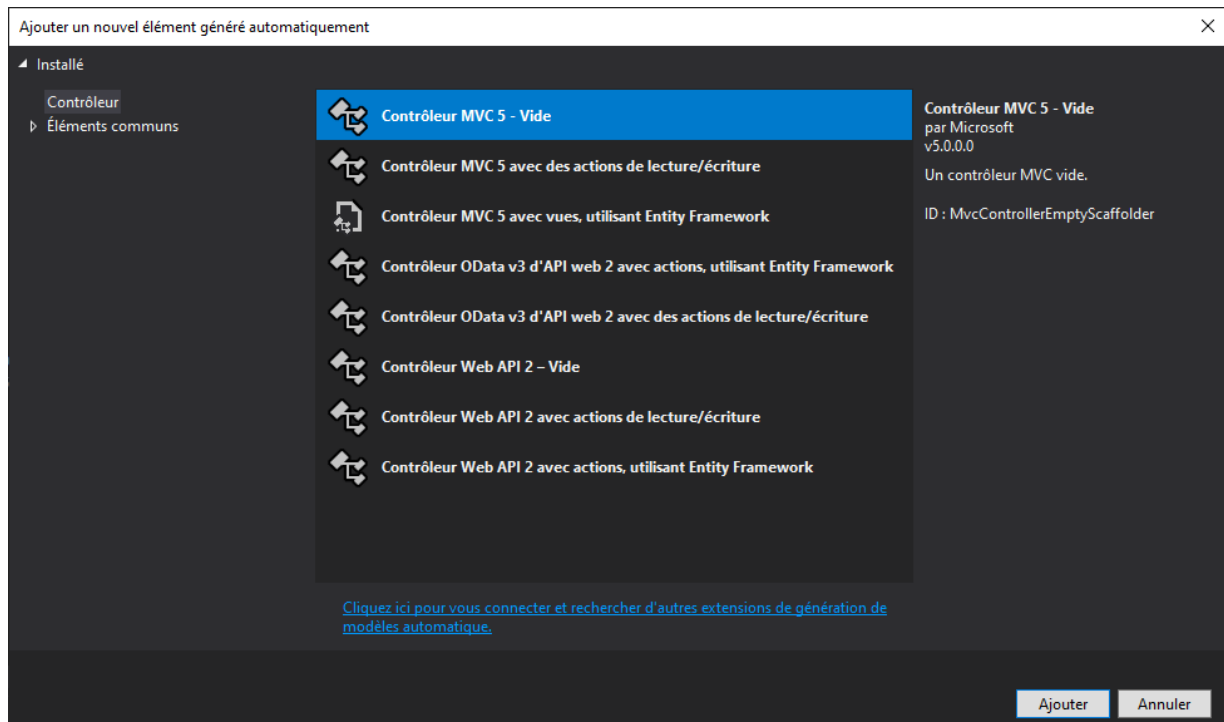
3.2 Définition des contrôleurs & vues

3.2.1 Contrôleur Restaurant

Nous allons créer nos premiers écrans.

Tout d'abord pour notre application, nous avons besoin de définir les lieux à proposer aux étudiants pour la soirée. Le contrôleur `RestaurantController` permettra d'accéder à des écrans de création de restaurant, de modification de restaurant et de la liste de tous les restaurants.

Créer un contrôleur MVC vide qui s'appelle « *RestaurantController* »



3.2.2 Créer un restaurant

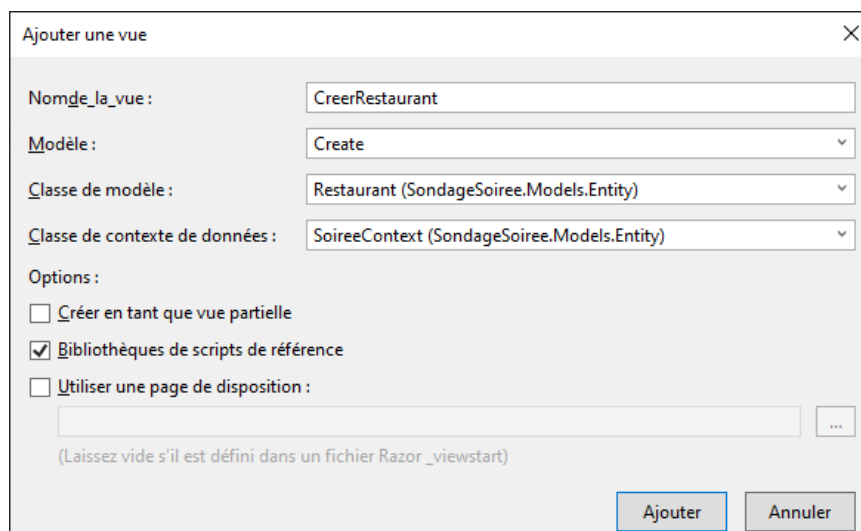
Dans ce contrôleur, ajouter la méthode « *CreerRestaurant()* » qui retourne une vue.

Ensuite il faut créer la vue :

- Faire une clic droit sur la méthode
- Choisir « Ajouter une vue... »

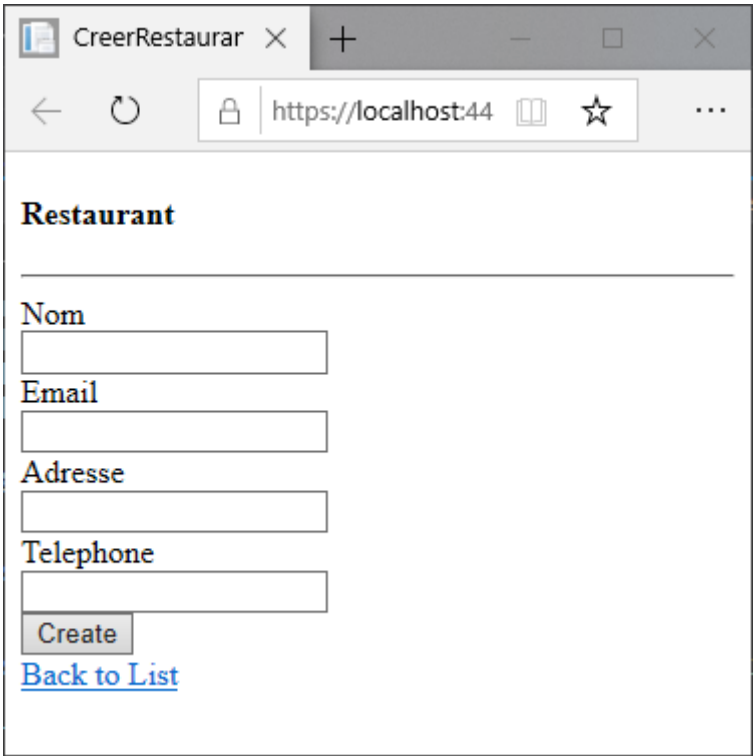
Cette vue est fortement typée (classe modèle `Restaurant`).

Elle prend pour modèle les écrans de créations « Create ».



NOTE : Il peut être nécessaire de régénérer la solution pour ajouter la première vue.

Tester la page, pour cela, modifier le routage par défaut afin de définir cette page comme page de démarrage.
C'est magique, votre page de création de restaurant a été automatiquement créée.



Le fichier de vue généré est au format cshtml. C'est un mélange de html et de balise Razor. On retrouve des similarités par rapport à l'ASP.NET classique :

ASP.NET WebForms (.aspx)	ASP.NET Razor (.cshtml)
<form runat="server">	@using (Html.BeginForm()) {
<asp:label Text="xxx" runat="server" />	@Html.LabelFor(model => model.Nom)
<asp:TextBox Text="xxx" runat="server" />	@Html.EditorFor(model => model.Nom)
<asp:RequiredFieldValidator etc... >	@Html.ValidationMessageFor(model => model.Email)

On peut noter quelques points nouveaux :

- En début du fichier on trouve « @model SondageSoiree.Models.Entity.Restaurant », cela indique que la vue est fortement typée. Cela implique que pour tous les éléments du style model => model.Nom, « model » fera référence à un « Restaurant ».
- @Scripts.Render("~/bundles/jquery") & @Scripts.Render("~/bundles/jqueryval") permettent d'inclure le javascript. En MVC, les fichiers JavaScript (et CSS) peuvent être regroupé en package (appelé bundles). Cela permet de grouper plusieurs fichier JS dans un seul bundle, de simplifier l'appel à ces fichiers, ainsi que de les minifier automatiquement au passage en production. (Cela permet aussi de ne gérer les versions de script JS qu'a un seul endroit). Les bundles sont définis dans le fichier « App_Start/BundleConfig.cs »
- @Html.ActionLink("Back to List", "Index") en MVC on ne définit plus de lien vers des fichier, mais des noms d'action. Exemple, le « Index » précédent représente l'action nommé « Index » du contrôleur courant. (On peut changer de contrôleur, ou ajouter des attributs, en rajoutant des paramètres).

La page n'a pas de style, rajouter le bundle CSS dans le header :

```
@Styles.Render("~/Content/css")
```

La page est tout de suite plus moderne et le design devient responsive.

3.2.3 Métadonnées et Validation des données

Quand l'on clique sur le bouton « Create », les données sont envoyées au serveur même si elle ne sont pas bonne. (Par exemple tous les champs sont obligatoires).

Nous allons enrichir notre modèle pour permettre de valider nos données. Pour cela nous allons utiliser les annotations (System.ComponentModel.DataAnnotations). En .NET il y a de très nombreuses annotations permettant, entre autres, de valider nos données. Par exemple :

- [Required]
- [StringLength(100)]
- [Phone] (Note : cette annotation ne fonctionne pas coté client)
- [EmailAddress]
- [Display(Name = xxx)] : pour définir le nom de l'attribut
- ...

L'utilisation des annotations est extrêmement simple.

Par exemple il suffit d'écrire :

```
[Required]
[EmailAddress]
[StringLength(150)]
[Display(Name = "Email")]
public string Email { get; set; }
```

Pour que la propriété "Email" du modèle soit obligatoire, avec une longueur max de 150...

Cependant dans notre cas nous ne pouvons pas rajouter ces annotations à notre modèle car le modèle est généré par Entity Framework, et nos annotations seraient perdu si on régénère le modèle.

Nous allons encore une fois utiliser les classes partielles et définir des métadonnées permettant de valider nos données. Les métadonnées sont définies dans une classe à part qui ressemble au modèle.

- Créer un répertoire « Metadata » dans le dossier « Models »
- Importer le fichier « RestaurantMetadata.cs » dans le répertoire, et regarder la structure du code
- Importer le fichier « Restaurant.Part.cs » dans le répertoire « Entity ». La classe partielle fait le lien entre la classe Entity Framework et la classe Metadata.
- Relancer l'application et retenter de cliquer sur « Create » avec des données non valides. Que constatez-vous ?

Des validateurs ont été définis automatiquement grâce aux annotations précisées sur les propriétés de la classe Restaurant.

Vous pouvez ajouter l'annotation [Display(Name="Téléphone")] afin de mettre les accents sur le libellé.

Dans les annotations [Required(ErrorMessage= ...)], vous pouvez préciser les messages d'erreur.

Si vous testez les validateurs, vous pouvez voir qu'ils se déclenchent côté client grâce aux scripts appelés dans votre page (script défini dans le modèle MVC) :

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

- Enlever ces scripts de la page et gérer le POST :
Ajouter une méthode CreerRestaurant(Restaurant poResto) qui retourne la vue courante avec comme paramètre l'objet Restaurant saisi (la méthode est appelée en POST, ajouter l'annotation [HttpPost]). Tester à nouveau les validateurs.
Que constatez-vous ?

Vous pouvez remettre les scripts de validation précédemment enlevés.

- Le champ Téléphone doit respecter l'expression régulière suivante @"^0[0-9]{9}\$". Utiliser l'annotation [RegularExpression] (L'annotation [Phone] est trop permissive).
- Dans la saisie d'un restaurant, il faut saisir obligatoirement un moyen de contact du restaurant, Téléphone ou Email. Il est possible de faire ce test côté client ou côté serveur. Côté serveur, il suffit de faire hériter la classe Restaurant de l'interface IValidatableObject (utiliser la classe partielle) qui impose l'implémentation d'un validateur. Voici le code :

```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
    var valid = new List<ValidationResult>();
    if (string.IsNullOrEmpty(Telephone) && string.IsNullOrEmpty(Email))
        valid.Add(new ValidationResult("Vous devez saisir au moins un moyen de contacter le restaurant", new[] { "Telephone", "Email" }));
    return valid;
}
```

- Modifier le post back de la création d'un restaurant afin de sauvegarde le restaurant en base (utiliser un objet Dal). Bien sûr uniquement si les données sont valides (cf. ModelState.IsValid). Si la donnée n'est pas bonne renvoyer le restaurant à la vue.
- Ajouter une validation qui teste l'existence du nom du restaurant. Si celui-ci existe, le restaurant n'est pas enregistré et un message d'erreur apparaît : « Ce restaurant existe déjà » (Note : utiliser ModelState.AddModelError pour ajouter un message d'erreur).

3.2.4 Injection de dépendances

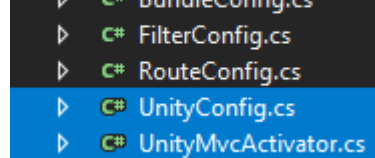
Lors de la sauvegarde nous avons utilisé un objet Dal. Cependant ce n'est pas très propre car nous avons lié notre contrôleur à une implémentation spécifique de IDal. (D'ailleurs actuellement, l'interface IDal ne sert à rien).

Nous allons utiliser un mécanisme, appelé « dependency injection » (ou injection de dépendances), c'est à dire que nous allons travailler uniquement avec des interfaces, et laisser *Unity* (l'outil de dependency injection), instancier la bonne implémentation.

- Préparer le contrôleur de restaurant en ajoutant le constructeur suivant :

```
private readonly IDal _dal;

public RestaurantController(IDal dal)
{
    _dal = dal;
}
```

- Installer *Unity* via NuGet : « *Unity.Mvc* »
 - Des nouveaux fichiers sont créés dans le dossier App_Start :
- 
- Essayer de relancer l'application, que ce passe-t-il ? (Une erreur apparaît car nous n'avons pas encore configuré Unity, et donc Unity n'est pas capable de fournir un IDal à notre Contrôleur).
 - Configurer Unity (fichier App_Start/UnityConfig.cs), rajouter la ligne (qui indique d'utiliser l'objet Dal lorsque l'on a besoin IDal) :

```
container.RegisterType<IDal, Dal>();
```

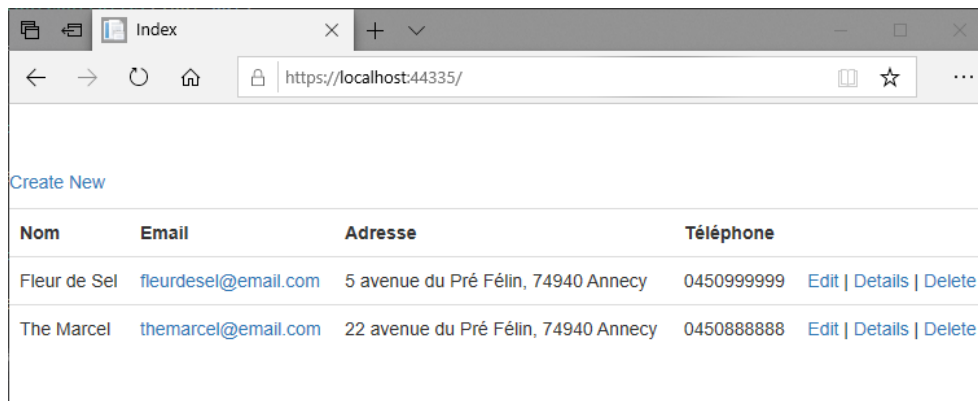
- Relancer l'application : Unity a fourni un objet Dal à notre contrôleur et tout fonctionne !

3.2.5 Liste restaurants

Dans la page de création, il existe un lien « Retour à la liste de restaurants ».

Ce lien permet d'aller dans l'écran qui liste les restaurants. Ce lien fait référence à l'action `Index()` du contrôleur `RestaurantController`.

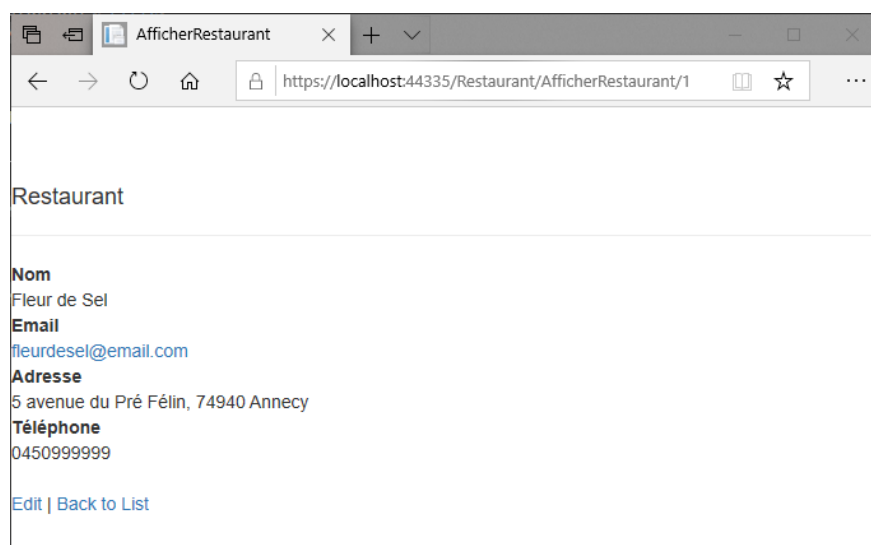
- Remplacer dans le routage par défaut, l'action `CreerRestaurant` par `Index` et créer la vue correspondante à cette action. Cette vue est fortement typée sur la classe modèle `Restaurant` et de classe de contexte `SoireeContext`. Cette vue prend pour modèle les écrans de création « List ».
- Passer l'objet le modèle à la view : `return View(_dal.RenvoieTousLesRestaurants());`
- Rajouter le style dans la page (comme vue précédemment, grâce aux `Bundle`)



- Référencer le lien « Crée » à l'action `CreerRestaurant` dans le contrôleur `RestaurantController`.
- Modifier l'action `CreerRestaurant`, si le modèle est valide, rediriger la page vers `Index` une fois l'insertion en base faite (Utiliser la méthode `RedirectToAction`).
- Supprimer les liens « Delete ».

3.2.6 Détail d'un restaurant

- Créer l'action `AfficherRestaurant` et sa vue dans le contrôleur `RestaurantController`. Cette vue est fortement typée sur la classe modèle `Restaurant` et de classe de contexte `SoireeContext`. Cette vue prend pour modèle les écrans de création « Details » :



- Référencer le lien « Détail » dans l'écran qui liste les restaurants, à cette action.
- Penser à gérer les actions sur les liens « Retour à la liste » des restaurants et « Modifier » (vers l'action que nous créerons par la suite).

3.2.7 Modifier un restaurant

- Créer l'action ModifierRestaurant et sa vue dans le contrôleur RestaurantController. Cette vue est fortement typée sur la classe modèle Restaurant et de classe de contexte SoireeContext. Cette vue prend pour modèle les écrans de création « Edit ».
Cet écran suit les mêmes règles que la création, sauf le test sur l'existence du nom du restaurant (Note : si vous voulez aller plus loin, rajouter la validation sur le nom (le nom ne doit pas exister en base avec un Id différent)).
- Référencer le lien « Modifier » dans l'écran qui liste les restaurants, à cette action.
- Une fois la modification effectuée, rediriger vers la page de détail de l'objet.

3.2.8 Intégration dans le Layout Global

L'ensemble des pages que nous avons créés sont indépendantes. (Chaque page définit son head, son body, etc...). Cela implique beaucoup de code (html) dupliqué.

ASP.NET MVC utilise un mécanisme de Layout. Si on ouvre nos fichier cshtml, on trouve ce code en début de fichier :

```
@{  
    Layout = null;  
}
```

Ce code désactive le Layout.

Si on le supprime, la page utilisera le Layout par défaut. (Le layout par défaut est défini dans le fichier View/_ViewStart.cshtml).

- Etudier le layout par défaut
- Supprimer le Layout = null et les balises redondantes (celles qui sont déjà définies dans le layout par défaut).
- Modifier le layout par défaut (supprimer les liens inutiles, et le personnaliser (nom de l'application, style...))
- Le layout global n'inclut pas le bundle *jqueryval*. Deux solutions :
 - Ajouter @Scripts.Render("~/bundles/jqueryval") au Layout global (sous le *jquery*)
→ Cette solution est simple mais pas très propre, on inclut le script *jqueryval* même dans les pages qui n'en ont pas besoin.
 - Utiliser l'instruction @section scripts { ... } dans les pages de création et de modification pour n'ajouter le script que dans les pages qui ont besoin de validation. (Note : le code entre { } sera placé au niveau de l'instruction @RenderSection("scripts", required: false) du layout.

Note : la prochaine fois que vous créez une vue, cochez la case « Utiliser une page de disposition » pour utiliser le Layout par défaut.