Ce TD est la suite directe du TD2 et utilise l'application créée précédemment.
Pensez à bien enregistrer votre projet de manière à pouvoir le retrouver lors des prochaines séances.

1 Authentification

1.1 ASP.NET Identity

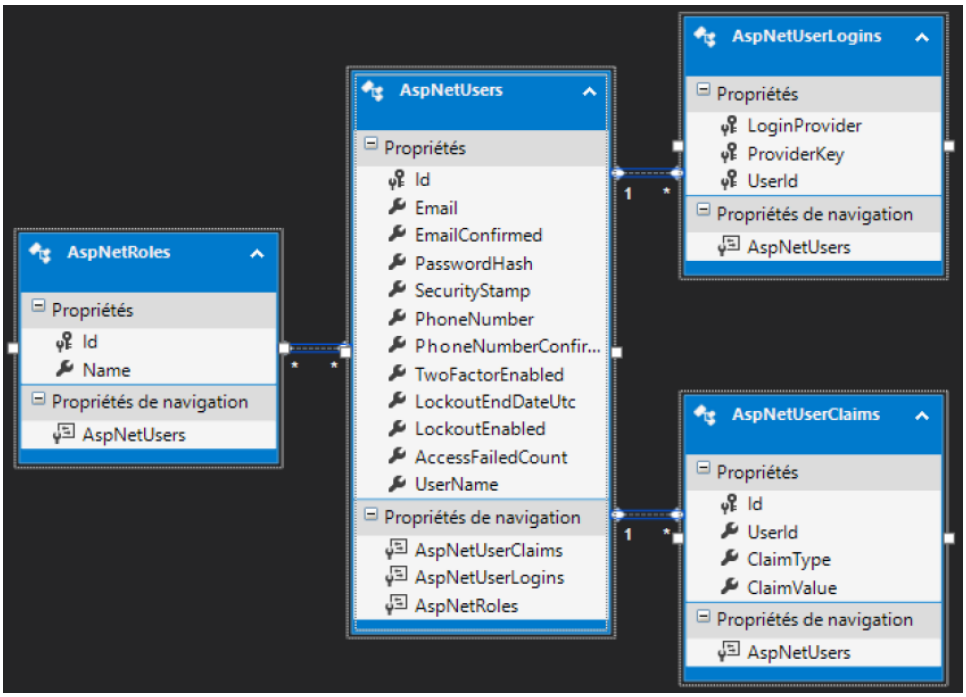
1.1.1 Comptes d'utilisateurs individuels

L'authentification et la gestion des comptes utilisateurs (création de compte, modification de mot de passe...) sont des actions classiques que l'on retrouve sur les sites web.
Microsoft fournit directement cette gestion. Le framework de gestions des comptes utilisateur s'appelle ASP.NET Identity (http://www.asp.net/identity), il permet facilement de :

- Créer des comptes
- Modifier des comptes
- Gérer des authentifications externes (Google, Facebook…)
- Gérer de l'authentification à deux facteurs (SMS / Email)

Lors de la création d'un projet web ASP.NET il est possible de choisir un mode d'authentification préinstallé, comme les « Comptes d'utilisateurs individuels ».

Avec cette option un schéma de données est ajouté au projet :



Dans ce modèle un peu compliqué on retrouve :

- L'entité principale **AspNetUsers** qui continent les informations de base de l'utilisateur.
- L'entité **AspNetUserLogins** qui représente les différents identifiants alternatifs que l'utilisateur peut avoir (token d'authentification Google, Facebook…)
- L'entité **AspNetRoles** contient les autorisations basées sur les rôles (Un utilisateur qui a le rôle « Admin » a accès à certaines pages).

- L'entité **AspNetUserClaims** conserve un ensemble de clés / valeurs permettant d'authentifier un utilisateur sous différents systèmes, de manière générique.
  (Par exemple un système A va demander les claims Email et Date de Naissance pour identifier un utilisateur, un système B utilisera seulement un login…)

Les entités sont définies dans le package `Microsoft.AspNet.Identity.EntityFramework`, ce sont des implémentations des interfaces définies dans `Microsof.AspNet.Identity`. ASP.Net Identity fonctionne de manière Entity First (Génération de la base de données à partir des entités), donc nous ne pourrons malheureusement pas l'utiliser (problème de droit insuffisant sur `srv-jupiter`).

Il est possible d'utiliser ASP.Net Identity de manière Database First, mais beaucoup de code sera nécessaire (Il faudra redéfinir un certain nombre d'interface de `Microsof.AspNet.Identity`). Nous allons donc redéfinir manuellement une authentification « basique » (uniquement la gestion de création de compte / connexion / déconnexion) basé sur la classe `ClaimsIdentity`.

1.1.2 Autres authentifications

Lors de la création d'un projet MVC nous avons la possibilité d'utiliser d'autres authentifications :

- Pas d'authentification
- Comptes d'utilisateurs individuels (vu précédemment)
- Comptes d'organisation (pour les applications Cloud / Microsoft Azure)
- Authentification Windows (pour les applications intranet, authentifié par Active Directory)

1.2 Création de l'authentification « Maison »

1.2.1 Initialisation

Lors de la création de votre projet, si vous avez choisis l'authentification « Compte d'utilisateurs individuels » vous possédez déjà les dépendances nécessaires.

Sinon il faut ajouter les packages NuGet suivants :

- `Microsoft.AspNet.Identity.Owin`
- `Microsoft.Owin.Host.SystemWeb`

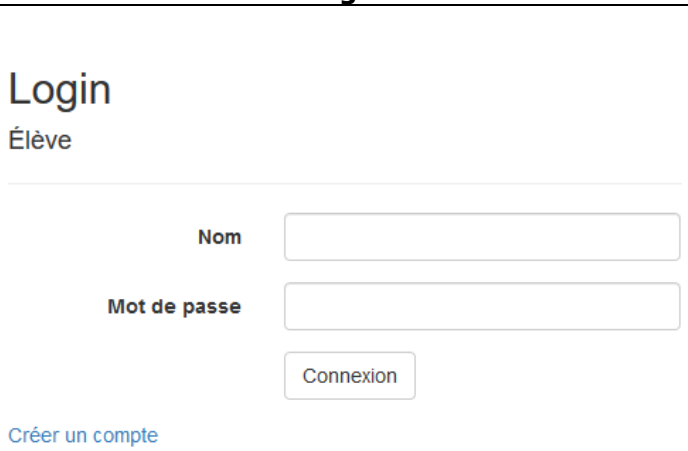
Afin de configurer l'authentification, placer le fichier `Startup.cs` à la racine du projet (remplacer celui existant si nécessaire).

1.2.2 Création du module

Vous savez créer des contrôleurs et des vues, vous pourrez donc générer celles nécessaires avec l'entité Elève.

Voici quelques contraintes pour la gestion de l'authentification :

- Le contrôleur devra s'appeler `CompteController`
- Utiliser les méthodes créées dans la classe `Dal` pour créer, récupérer des étudiants et leur authentification
- Les utilisateurs devront pouvoir se connecter (nom de l'action : `Login`)
- Les utilisateurs devront pouvoir se déconnecter
- Les utilisateurs devront pouvoir créer un compte
- Le nom doit être unique (il sert à s'authentifier) : il faudra rajouter une méthode `EleveExist` dans `IDal`
- Le mot de passe doit faire au moins 6 caractères
- Le mot de passe de doit **JAMAIS** être stocké en clair :
  - Utiliser la méthode `Crypto.HashPassword()` pour hacher le mot de passe
  - Utiliser la méthode `Crypto.VerifyHashedPassword()` pour vérifier que le mot de passe haché correspond au mot de passe envoyé
- Les pages de login et création de compte devront ressembler à :

Login	Création de compte
	

Aide :

- Utiliser le tag @Html.PasswordFor() pour générer un <input type="password" />
- Utiliser les méthodes d'authentifications suivantes (à rajouter dans le contrôleur) :

```
private void IdentitySignin(Eleve eleve)
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.NameIdentifier, eleve.Id.ToString()),
        new Claim(ClaimTypes.Name, eleve.Nom)
    };

    var identity = new ClaimsIdentity(claims, DefaultAuthenticationTypes.ApplicationCookie);

    HttpContext.GetOwinContext().Authentication.SignIn(new AuthenticationProperties()
    {
        AllowRefresh = true,
        IsPersistent = true,
        ExpiresUtc = DateTime.UtcNow.AddDays(7)
    }, identity);
}

private void IdentitySignout()
{
    HttpContext.GetOwinContext().Authentication.SignOut(
        DefaultAuthenticationTypes.ApplicationCookie,
        DefaultAuthenticationTypes.ExternalCookie
    );
}
```

1.2.3 Gérer les autorisations des pages

Les utilisateurs peuvent maintenant s'authentifier. Il faut maintenant restreindre les accès au site :

- L'ensemble des actions de `RestaurantController` doivent être restreintes aux utilisateurs authentifiés. (Utiliser l'annotation `[Authorize]`) sur la classe.
- Seules les actions de connexion et de création de compte doivent être accessibles à tous (utiliser `[Authorize]` et/ou `[AllowAnonymous]`).

Tester maintenant d'accéder à la liste des restaurants sans être authentifier. L'application devrait rediriger vers la page de connexion.

Ajouter le fichier `_LoginPartial.cshtml` fourni dans le dossier `/Views/Shared/`
Modifier le layout global en appelant cette vue partielle juste après les éléments du menu :
`@Html.Partial("_LoginPartial")`

1.2.4 Gérer les autorisations avancées

Nous avons maintenant un système d'authentification qui divise le site en deux parties :

- La partie publique, accessible sans authentification
- La partie privée, accessible uniquement après connexion

Ce n'est cependant pas suffisant, nous souhaitons que tous les élèves puissent voter, mais pas que n'importe quelle élève puisse modifier les restaurants et administrer les sondages !
Nous avons donc besoin d'utiliser les Rôles.

- Rajouter une colonne `Role` dans la table `Eleve` (`varchar(50)` autorisant les valeurs `null`)

- Modifier votre compte dans la base en mettant « *Admin* » comme rôle.

- Mettre à jour le modèle Entity Framework (Clic droit sur le modèle → Mettre à jour le modèle à partir de la base de données).

- Modifier la fonction `IdentitySignin()` et rajouter à la liste des claims :

```
if (eleve.Role != null)
    claims.Add(new Claim(ClaimTypes.Role, eleve.Role));
```

- Rajouter des `[Authorize(Roles = "Admin")]` sur les actions du `RestaurantController` sur les méthodes qui ne doivent être accessibles qu'aux admins (Creer / Modifier)

- Cacher les liens qui ne sont plus accessibles (par exemple le lien d'édition dans la liste des restaurants)
    - Utiliser l'instruction `@if (User.IsInRole("Admin"))`

- Vérifier que tout fonctionne :
    - Se connecter avec un compte admin et vérifier que l'on a accès à tout
    - Se connecter avec un compte standard et vérifier que l'on a accès juste aux détails

2 Gestion des sondages

La gestion des sondages est similaire à la gestion des restaurants, à vous de jouer !
Il doit être possible de voir la liste des sondages (pour tout utilisateur authentifié) et d'en créer de nouveaux (administrateur seulement).

Créer la classe partielle, les métadonnées de validation, les contrôleurs, les vues, gérer les autorisations…

Note :

- La date doit être >= la date du jour
- Modifier la route par défaut du site pour diriger vers la liste des sondages
- Rajouter les informations suivantes dans les métadonnées pour obtenir un calendrier :

```
[DataType(DataType.Date)]
[DisplayFormat(DataFormatString = "{0:dd/MM/yyyy}", ApplyFormatInEditMode = true)]
```

3 Gestion des Votes / Résultats

La page de vote sera un peu plus compliquée à réaliser (même si au niveau de l'interface graphique elle reste très simple) :

Contrairement aux autres pages de type Création, la page vote ne correspond pas directement à un Model. Or nos vues sont typées, nous devons donc créer un modèle « transformé » qui fournira l'ensemble des données à la vue.
Cela s'appelle un « ViewModel ». Ce concept est très souvent utilisé car en général notre modèle ne correspond que rarement à ce que nous voulons exactement afficher.

- Créer un nouveau répertoire « *ViewModels* » à la racine du projet
- Rajouter le fichier `VoteViewModel` dedans. Vous remarquerez que :
  - Il y a 2 view models, un pour les éléments de la liste (`ChoixRestoViewModel`) et un pour la liste (`VoteViewModel`)
  - On peut directement mettre les annotations sur le view model (pas besoin de metadata, car le fichier n'est pas généré).

Créer maintenant le reste du code nécessaire au vote et a l'affichage des résultats. Rappel : un élève ne peut voter qu'une seule fois par sondage :

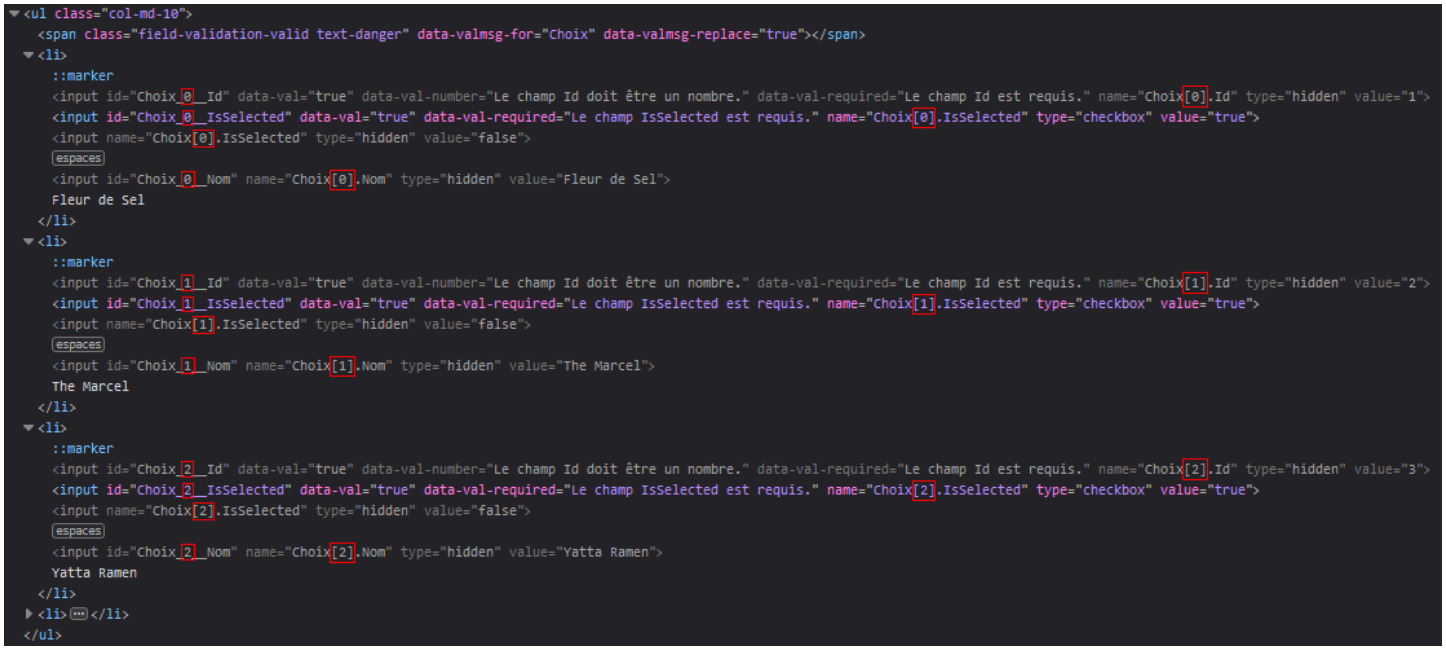
- Si un élève n'a pas voté et qu'il essaye d'accéder au résultat : le rediriger vers la page de vote
- Inversement : si un élève qui a déjà voté essaye d'accéder à la page des votes, le rediriger vers les résultats

Aide 1

Lors ce que vous générer la vue pour les votes (type de vue « Create »), vous remarquez que la liste n'est pas affichée. Il faudra l'écrire manuellement de la manière suivante :

```
@Html.ValidationMessageFor(model => model.Choix, "", new { @class = "text-danger" })
@for (int i = 0; i < Model.Choix.Count; i++)
{
    <li>
        @Html.HiddenFor(m => m.Choix[i].Id)
        @Html.CheckBoxFor(m => m.Choix[i].IsSelected)
        @Html.HiddenFor(m => m.Choix[i].Nom)
        @Html.DisplayFor(m => m.Choix[i].Nom)
    </li>
}
```

Les éléments de type *Hidden* permettrons de régénère la liste lors du POST. L'utilisation d'un `for (int i …)` est indispensable, ASP utilise l'index `i` pour nommer les champs. Si on observe dans le navigateur le code HTML généré de la page on retrouve notre index :



Si on utilise un `foreach`, il n'y aura pas d'index, et ASP nommera les éléments de manière identique, on ne pourra donc pas récupérer les données dans le POST.

Aide 2

Pour récupérer l'identifiant de l'utilisateur connecté, on utilisera le code suivant :

```
int userId = int.Parse(
    ((ClaimsIdentity)User.Identity).Claims.First(s => s.Type == ClaimTypes.NameIdentifier).Value
);
```

Pour comprendre un peu la raison de ce code regarder plus en détail la fonction de connexion (dans le `CompteController`) :

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.NameIdentifier, eleve.Id.ToString()),
    new Claim(ClaimTypes.Name, eleve.Nom)
};
```

4 Sécurité

Les principales failles de sécurité sur internet sont le Cross-Site Scripting (XSS), les injections SQL, le Cross Site Request Forgery (CSRF). Notre application ne présente normalement pas de faille XSS ni de possibilité d'injection SQL, mais nous allons devoir un peu modifier notre code pour être résistant au CSRF.

4.1 XSS

Le XSS est une faille assez célèbre. La plupart des Framework proposent par défaut des mécanismes de protection.

Rappel : le XSS consiste à injecter du code sur un serveur qui sera exécuté par les navigateurs des utilisateurs.
Par exemple si sur un formulaire on rentre la chaine `<script>alert("bonjour")</script>` et que le serveur accepte cette requête et stock la valeur sans l'échapper alors il y a une grosse faille de sécurité XSS (tous les utilisateurs consultant la page verront une popup « bonjour » s'afficher, ou beaucoup plus grave, suivant le code…).

En .NET il y a une protection contre ce genre d'attaque (même dans les anciennes versions d'ASP.NET WebForms), le serveur rejette systématiquement les requêtes qui peuvent être potentiellement dangereuse.

Vous pouvez essayer par exemple de créer une utilisateur avec comme nom `<script>alert('toto')</script>` cela ne marchera pas, la requête sera arrêtée avant d'atteindre notre fonction POST.

4.2 Injection SQL

L'injection SQL consiste à tenter de détourner une requête SQL.
Par exemple supposons que la requête SQL d'authentification soit `"SELECT * FROM User WHERE login='"+ login + "' AND password = '"+ pwd + "' ;"`. login et pwd sont récupéré du formulaire d'authentification.
Que se passe-t-il si on rentre pour login « toto » et pour password « `xx' OR '1'='1` » ?

- ➔ La requête deviendra : `SELECT * FROM USER where login='toto' AND password = 'xx' OR '1'='1'`
- ➔ L'utilisateur sera toujours authentifié, peu importe le mot de passe !

Deux solutions à ce problème :

- Si l'on utilise du SQL ne jamais faire de concaténation mais utiliser tout le temps des requêtes paramétrée. (Ex : « `SELECT * FROM USER where login=@login AND password = @pwd` »).
- Si on utilise un ORM (comme c'est notre cas), il n'y a pas de risque. (A moins d'une faille dans l'ORM mais c'est plutôt rare !)

4.3 Cross-Site Request Forgery

Une faille plus discrète mais pas moins dangereuse est le CSRF. Surtout que peu de Framework propose une protection par défaut. Le principe est là encore très simple, il consiste à envoyer au serveur un formulaire qu'il n'a pas créé.

```
<body onload="document.getElementById('fm1').submit()">
    <form id="fm1" action="http://yoursite.com/UserProfile/SubmitUpdate" method="post">
        <input name="email" value="hacker@somewhere.evil" />
        <input name="hobby" value="Defacing websites" />
    </form>
</body>
```

Hébergeons-le sur un site. Un *utilisateur A*, connecté à *yoursite.com*, se rends sur notre site.

Que se passe-t-il ?
Au chargement de la page (onload), le navigateur de l'*utilisateur A* enverra le formulaire à *yoursite.com* avec les données choisies par le pirate. *A* est authentifié sur *yoursite.com*, pas de soucis, le serveur acceptera la requête, et enregistrera les données du pirate !

Pour cela il existe un moyen assez simple de se protéger : rajouter un jeton aléatoire dans le formulaire. Au moment du POST vérifier que le jeton est bien celui que l'on a généré. En ASP MVC, le jeton est généré dans tous les formulaires permettant l'édition. Vous avez peut-être remarqué le code `@Html.AntiForgeryToken()` dans les fichiers `cshtml` juste après `@using (Html.BeginForm())`

C'est la génération du jeton aléatoire.
Dans le code HTML généré on peut voir un `<input type="hidden" />` correspondant :

```
<form action="/Restaurant/CreerRestaurant" method="post" novalidate="novalidate">
  <input name="__RequestVerificationToken" type="hidden"
  value="Of5I7twUVfln55nqp9_L1RwnYLTT4ODD2klqS-c5H_GIuUWSMFEjF9UBZ-
  TS_4V1WJHXP0SqBRCvwK72wYMDn3kRrqS6IHKOR0em94ClBU9qou_ch3JilBqA2">
```

Vous pouvez essayer de recharger la page, il change à chaque fois. Cependant notre code coté serveur ne valide pas encore ce jeton ! Il est indispensable de rajouter l'attribut `[ValidateAntiForgeryTokenAttribute]` à chaque fonction POST. Prenez l'habitude de toujours utiliser cette annotation quand vous utilisez une fonction `[HttpPost]`.

Si vous voulez vérifier la protection fonctionne, vous pouvez faire le test suivant :

- Se déconnecter
- Aller sur la page de login (un jeton T1 est généré)
- Sauvegarder la page de login (le fichier HTML, via le navigateur), elle contient le jeton T1
- Se connecter en utilisant la page que l'on a chargée, on envoie T1 au serveur, la requête est valide
- Ouvrir le fichier HTML dans le navigateur
- Tenter de se connecter avec cette page, on ré-envoie T1 au serveur, qui a déjà été utilisé, la requête est considérée comme frauduleuse

Plus de détails : https://docs.microsoft.com/en-us/aspnet/web-api/overview/security/preventing-cross-site-request-forgery-csrf-attacks

De plus comme bonne pratique de sécurité, il est recommandé de toujours préciser la méthode HTTP acceptée par une action d'un contrôleur, en ajoutant `[HttpGet]` aux actions sans attribut par exemple.

Si l'on souhaite accepter plusieurs méthodes HTTP sur une même action, on pourra utiliser l'attribut `[AcceptVerbs(HttpVerbs.Get | HttpVerbs.Post)]`

5 Pour aller plus loin

Nous avons maintenant un site complet permettant de gérer des sondages simples.
Cependant il est loin d'être parfait :

- Vérifier qu'il n'y a pas de lien mort sur le site
- Gérer la suppression des Restaurants, Sondages, Elèves… (Attention aux contraintes d'intégrité référentielles !)
- Valider le mot de passe lors de la création (en ajoutant un 2ème champ mot de passe qui devra être identique au premier)
    - Utiliser un ViewModel
    - Utiliser l'annotation [Compare]
- Rajouter une partie de gestion des utilisateurs
    - Rajouter un rôle « root »
    - Le root pourra modifier le rôle des autres utilisateurs
    - Le root aura tous les droits de l'admin