Frameworks de développement

TP 5 User Identity with ASP.NET CORE

ASP.NET Core Identity:

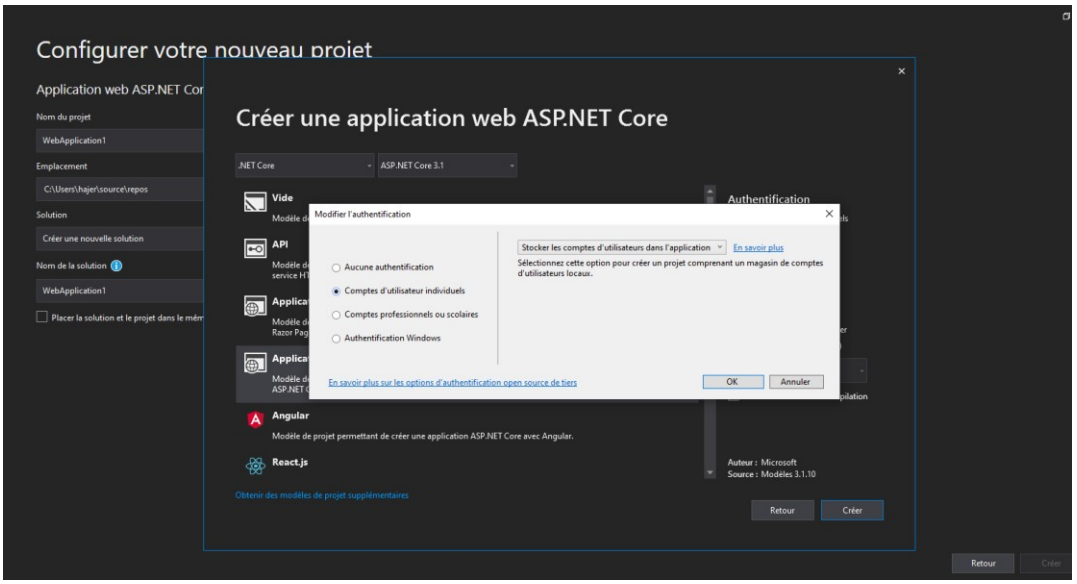
- Est une API qui prend en charge la fonctionnalité de connexion de l'interface utilisateur.
- Gère les utilisateurs, les mots de passe, les données de profil, les rôles, les revendications, les jetons, la confirmation par e-mail et bien plus encore.

Les utilisateurs peuvent créer un compte avec les informations de connexion stockées dans Identity ou ils peuvent utiliser un fournisseur de connexion externe. Les fournisseurs de connexion externes pris en charge incluent Facebook, Google, Microsoft Account et Twitter.

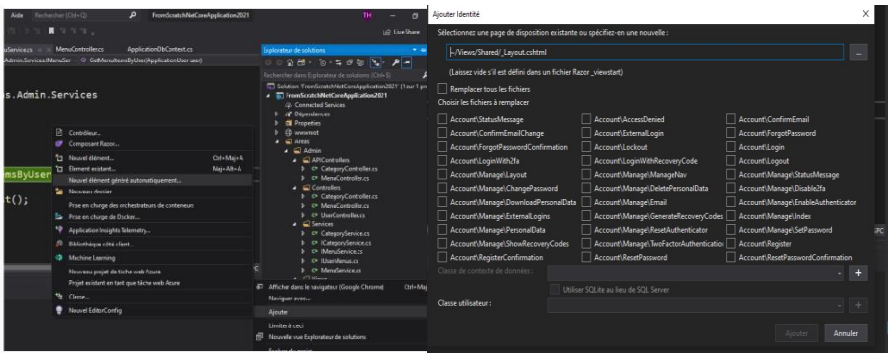
Identity est généralement configurée à l'aide d'une base de données SQL Server pour stocker les noms d'utilisateur, les mots de passe et les données de profil. Vous pouvez également utiliser un autre magasin persistant, par exemple, le stockage table Azure.

Nous pouvons créer le projet via la CLI : `dotnet new MVC --auth Individual -o WebApp1.`

On peut aussi travailler avec Visual Studio pour la prise en charge de l'API Identity.



Idéalement serait de faire la génération automatique des composants Identity au fur et à mesure en fonction du besoin.



- Pour générer les tables de l'Identity, il faut commencer par récupérer les dépendances à partir de l'interface de gestionnaire des packages (faites attention aux versions):

```
<PackageReference Include="Microsoft.AspNetCore.Identity" Version="2.2.0" />
     <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore"
 Version="6.0.15" />
```

- Il faut ajouter le service Identity dans program.cs :

```
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<AppDbContext>();
```

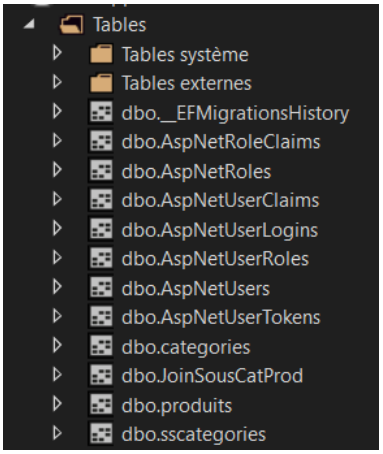
- On peut le configurer via le code suivant, Un exemple de configuration des options du service Identity

```
builder.Services.Configure<IdentityOptions>(options =>
{
    // Password settings.
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;

    // Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.AllowedForNewUsers = true;

    // User settings.
    options.User.AllowedUserNameCharacters =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;
});
```

- Il faut modifier la classe de contexte comme suit :

```
public class AppDbContext : IdentityDbContext
    {
        public AppDbContext(DbContextOptions
           <AppDbContext> options) : base(options)
        {


        }
```

- On peut ajouter une migration et mettre à jour la base de données. Les tables de l'Identity vont être crées.



Le code précédent configure Identity avec les valeurs d'option par défaut. Les services sont mis à la disposition de l'application via l' injection de dépendances.
Identity est activé en appelant UseAuthentication. **UseAuthentication** Ajoute l'intergiciel (middleware

) d'authentification au pipeline de requête.

L'application générée par un modèle n'utilise pas d'autorisation.
app.UseAuthorization est inclus pours'assurer qu'il est ajouté dans le bon ordre si l'application ajoute une autorisation. UseRouting, UseAuthentication , UseAuthorization et UseEndpoints doivent être appelés dans l'ordre.

- Générer les vues associées à Identity (Register, Login, Logout) via **ajouter nouvel élément généré automatiquement (scaffold Identity)**

- Etant donné que nous allons travailler avec les Razor pages de génération de vue Identity (projet Razor Pages), il faut activer les razor pages via

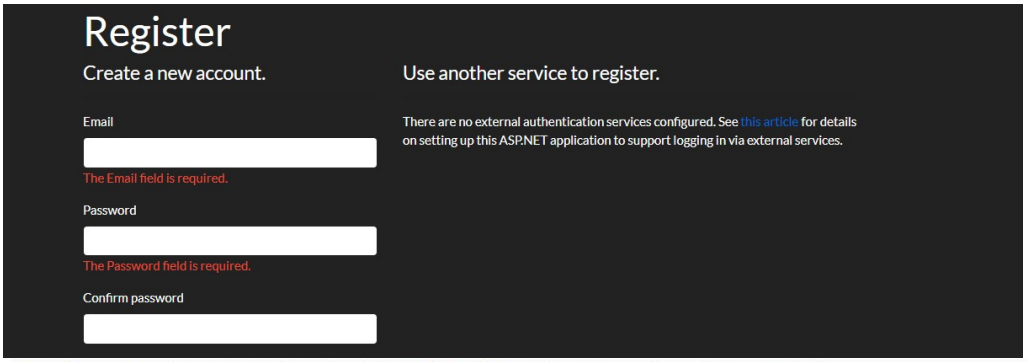
```
builder.Services.AddRazorPages();
app.MapRazorPages();
```

- Examiner les liens d'authentification dans la vue partielle LoginPartial

```
@using Microsoft.AspNetCore.Identity

@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
@if (SignInManager.IsSignedIn(User))
{
    <li class="nav-item">
        <a id="manage" class="nav-link text-dark" asp-area="Identity" asp-page="/
Account/Manage/Index" title="Manage">Hello @UserManager.GetUserName(User)!</a>
    </li>
    <li class="nav-item">
        <form id="logoutForm" class="form-inline" asp-area="Identity" asp-page="/
Account/Logout" asp-route-returnUrl="@Url.Action("Index", "Home", new { area =
"" })">
            <button id="logout" type="submit" class="nav-link btn btn-link text-
dark">Logout</button>
        </form>
    </li>
}
else
{
    <li class="nav-item">
        <a class="nav-link text-dark" id="register" asp-area="Identity" asp-page="/
Account/Register">Register</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" id="login" asp-area="Identity" asp-page="/
Account/Login">Login</a>
    </li>
}
</ul>
```

1. Examiner le « Register »

Quand un utilisateur clique sur le bouton Register de la Register page, l' RegisterModel.OnPostAsync action est appelée. L'utilisateur est créé par CreateAsync sur l' objet userManager.




```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync())
                                          .ToList();
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = Input.Email, Email = Input.Email };
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with password.");

            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
            var callbackUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { area = "Identity", userId = user.Id, code = code },
                protocol: Request.Scheme);
```

Question 1 : Lister tous les utilisateurs de votre application ASP.Net Core MVC

Commencer par ajouter des utilisateurs via UI
Pour lister les utilisateurs ajoutés via UI, il faut :

- Ajouter un ActionResult sous AccountController

```
private readonly UserManager<IdentityUser> _userManager;
0 références
public AccountController(UserManager<IdentityUser> _userManager)
{
    this._userManager = _userManager;
}
0 références
public IActionResult GetAllUsers()
{
    var users = _userManager.Users;
    return View(users);
}
```

- Ajouter une vue contenant les Emails de la liste des utilisateurs

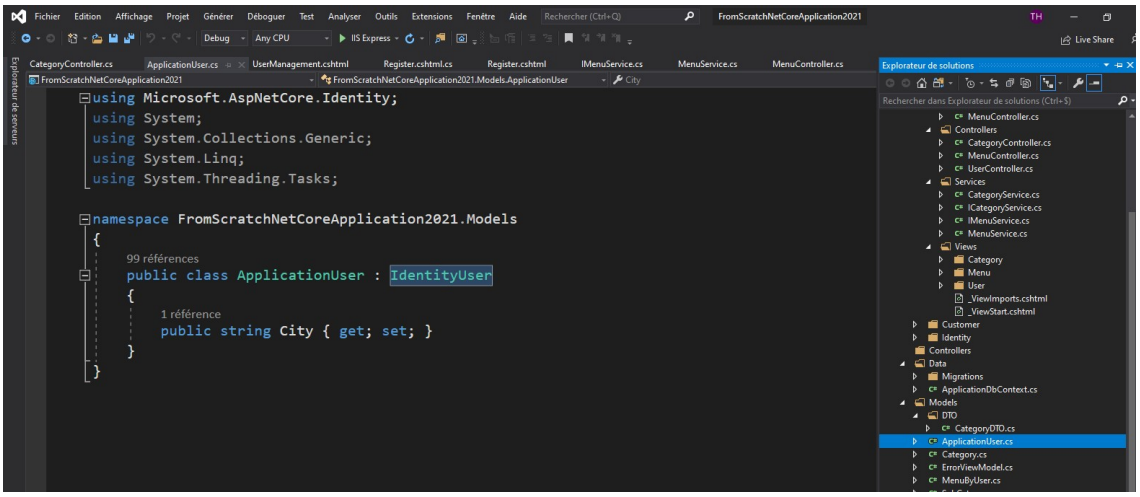
```
@model IEnumerable<Microsoft.AspNetCore.Identity.IdentityUser>
```

```
@{ ViewData["Title"] = "UserManagement";
   Layout = "~/Views/Shared/_Layout.cshtml"; }
<h1>UserManagement</h1>

@foreach (var item in Model)
{
<h4>@item.Email</h4>}
```

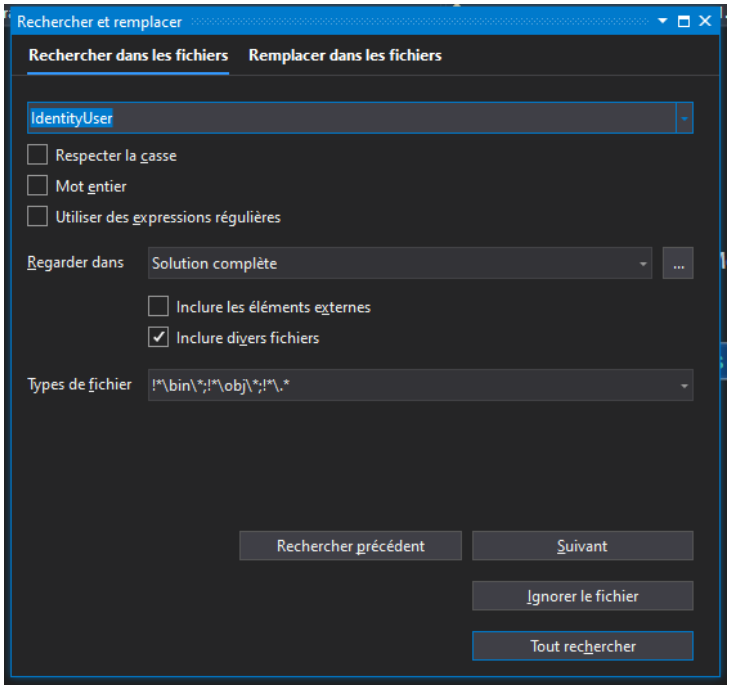
- Tester

Question 2 : Etendre IdentityUser pour plus de propriétés en ajoutant string City.

- Créer une classe ApplicationUser qui hérite de IdentityUSer et qui contient les propriétés que vous voulez ajouter.



A présent, toutes les références de IdentityUser doivent changer en ApplicationUser.



!!!Pensez à forcer IdentityDbContext à IdentityDbContext<ApplicationUser>

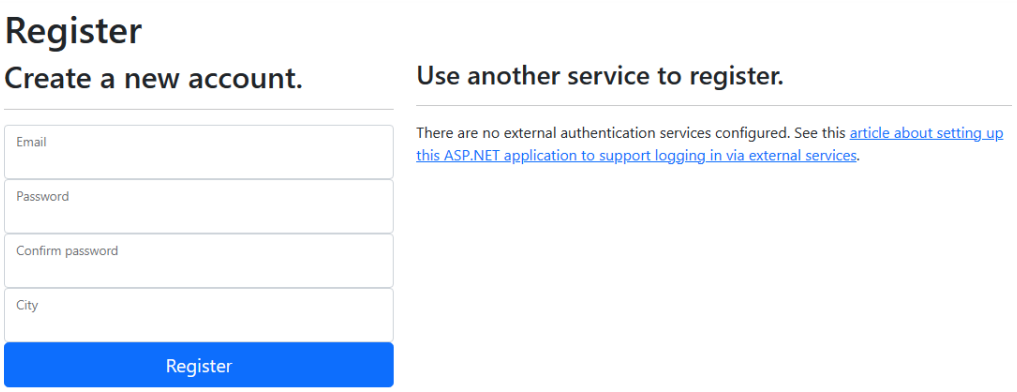
```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
```

- Faites les modifications nécessaires (directives manquantes)
- Ajouter les migrations pour que la propriété s'ajoutent à la base de données.
- Mettez à jour la base de données
- Modifier Register Model du Register.cshtml pour ajouter la propriété City

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

- Réaliser les modifications nécessaires pour que la vue Register s'affiche ainsi:

(La propriété City s'ajoute pour la création d'un nouveau compte utilisateur)



- Ajouter à la vue Register.cshtml

```
<div class="form-group">
    <label asp-for="Input.City"></label>
    <input asp-for="Input.City" class="form-control" />
    <span asp-validation-for="Input.City" class="text-danger"></span>
</div>
```

- Dans OnPostAsync du RegisterModel, veuillez ajouter : `user.City = Input.City;`
- Tester

Question 3 : L'objectif de cette partie consiste à lister tous les produits relatifs à un utilisateur particulier.

$ Ajouter un modèle Produit avec les propriétés de votre choix

$ Ajouter un modèle PanierParUser comme suit:

```
public class PanierParUser
    {
        public Guid Id { get; set; }
        public string UserID { get; set; }
        public Guid ProduitId { get; set; }
        public List<Produit> produits { get; set; }
    }
```

- Ajouter DbSet à ApplicationDbContext.
- Faire les migrations et mettez à jour la base de données
- Peupler la table PanierParUser
- Ecrire un actionResult pemettant de récupérer les produits par utilisateur (ASP.Net Core MVC).

```
public IActionResult PanierParUSer()
        {
            var currentuser = _userManager.GetUserId(User);
            var pro = _context.paniers
                .Where(c=>c.UserID==currentuser)
                .ToList();
            return View(pro);
        }
```

- Ajouter une vue fortement typée et tester.