Laboratoire: autorisations

# Objectif(s)

* MS Identity
* Autorisation

# Créer la branche de la fonctionnalité

Clonage du repository

1. Dans Gitkraken, clonez le repository
   * Soit via URL
   * Soit via GitHub.com
2. Ouvrez le repository (OK)

Création d’une branche pour la fonctionnalité

1. Créez une branche appelée **FCT \_Autorisations**
2. Assurez-vous d'être dans la branche que vous avez créée dans LOCAL

Dans ce laboratoire, vous aurez à décider vous-mêmes quand faire les *Comment, commit, push.*

# Configurer Les rôles

## Créer des roles

Définir des variables statiques pour les rôles afin d’éviter les « chaînes magiques ».

1. Ouvrez la classe **Utility/AppConstants**

// Rôles

public const string AdminRole = "Admin";

public const string HunterRole = "Hunter";

public const string PlayerRole = "Player";

## Modifier la configuration utilisateur et rôle

1. Ouvrez la classe **ZombiePartyDbContext**

public class ZombiePartyDbContext: IdentityDbContext<IdentityUser>

{

## Enlever l’obligation de l’envoie de courriel

Nous n’avons pas les ressources configurées pour faire de l’envoie de courriels, nous allons donc retirer le code qui fait l’envoie de courriel pour la confirmation de l’enregistrement.

1. Ouvrez le fichier **/Areas/Identity/Pages/Account/Register.chtml.cs**
2. Retirez la variable \_emailSender et toutes les références au **IEmailSender** (IEmailSender ne doit plus être utilisé dans ce fichier, sinon vous allez avoir une erreur car il n’y a pas de servie d’enregistré (DI) pour IEmailSender)
3. Retirez les lignes d’envoie le courriel de confirmation (environ lignes 131 à 147)

\_logger.LogInformation("User created a new account with password.");

var userId = await \_userManager.GetUserIdAsync(user);

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 = userId, code = code, returnUrl = returnUrl },

protocol: Request.Scheme

);

await \_emailSender.SendEmailAsync(Input.Email, "Confirm your email",

$"Please confirm your account by <a href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

if (\_userManager.Options.SignIn.RequireConfirmedAccount)

{

return RedirectToPage("RegisterConfirmation", new { email = Input.Email, returnUrl = returnUrl });

}

else

{

await \_signInManager.SignInAsync(user, isPersistent: false);

return LocalRedirect(returnUrl);

}

## Configurer Initializer

Créer l’interface et la classe DbInitialer permettant de générer des données de départ pour les utilisateurs et les rôles. Les utilisateurs ne peuvent pas s’enregistrer en tant qu’administrateur par eux-mêmes via *Register!* Vous verrez la gestion des utilisateurs, rôles et revendications dans un cours subséquent.

1. Créez un dossier **DbInitializer**
2. Ajoutez une interface IdbInitializer
3. Contenant la méthode **Initialize**

public interface IDbInitializer

{

void Initialize();

}

1. Créez une classe **DbInitializer** implémentant l’interface
2. Injectez les éléments suivants dans la classe pour :
   * Le DbContext
   * Le user manager
   * Le rôle manager

public class DbInitializer : IDbInitializer {

private readonly UserManager<IdentityUser> \_userManager;

private readonly RoleManager<IdentityRole> \_roleManager;

private readonly ZombiePartyDbContext \_db;

public DbInitializer(

UserManager<IdentityUser> userManager,

RoleManager<IdentityRole> roleManager,

ZombiePartyDbContext db)

{

\_roleManager = roleManager;

\_userManager = userManager;

\_db = db;

}

}

1. Implémentez la méthode **Initialize**

Modifiez VotreNom en bleu par votre nom :)

public void Initialize() {

// Exécuter les migrations sont effectuées

try {

if (\_db.Database.GetPendingMigrations().Count() > 0) {

\_db.Database.Migrate();

}

}

catch(Exception ex) { }

// Créer les rôles suivants si aucun rôle ne figure dans la bd

if (!\_roleManager.RoleExistsAsync(AppConstants.AdminRole).GetAwaiter().GetResult())

{

\_roleManager.CreateAsync(new IdentityRole(AppConstants.AdminRole))

.GetAwaiter().GetResult();

\_roleManager.CreateAsync(new IdentityRole(AppConstants.HunterRole))

.GetAwaiter().GetResult();

\_roleManager.CreateAsync(new IdentityRole(AppConstants.PlayerRole))

.GetAwaiter().GetResult();

// Créer un User pour le rôle Admin

\_userManager.CreateAsync(new ApplicationUser {

UserName = "VotreNom@ZombieParty.com",

Email = "VotreNom@ZombieParty.com",

NickName = "Votre surnom",

PhoneNumber = "1111111111",

EmailConfirmed = true

}, "Admin123\*").GetAwaiter().GetResult();

ApplicationUser user = \_db.ApplicationUsers.FirstOrDefault(u => u.Email == "VotreNom@ZombieParty.com");

\_userManager.AddToRoleAsync(user, AppConstants.AdminRole)

.GetAwaiter().GetResult();

1. Ajoutez quatre autres utilisateurs, deux chasseurs et deux joueurs

// Créer deux Users pour le rôle Chasseur

\_userManager.CreateAsync(new ApplicationUser

{

UserName = "TopChasseur@ZombieParty.com",

Email = "TopChasseur@ZombieParty.com",

NickName = "Top Chasseur",

PhoneNumber = "1111111111",

EmailConfirmed = true,

}, "Chasseur123\*").GetAwaiter().GetResult();

ApplicationUser user2 = \_db.ApplicationUsers.FirstOrDefault(u => u.Email == "TopChasseur@ZombieParty.com");

\_userManager.AddToRoleAsync(user2, AppConstants.HunterRole)

.GetAwaiter().GetResult();

\_userManager.CreateAsync(new ApplicationUser

{

UserName = "CrazyChasseur@ZombieParty.com",

Email = "CrazyChasseur@ZombieParty.com",

NickName = "Crazy Chasseur",

PhoneNumber = "1111111111",

EmailConfirmed = true,

}, "Chasseur123\*").GetAwaiter().GetResult();

ApplicationUser user3 = \_db.ApplicationUsers.FirstOrDefault(u => u.Email == "CrazyChasseur@ZombieParty.com");

\_userManager.AddToRoleAsync(user3, AppConstants.HunterRole)

.GetAwaiter().GetResult();

// Créer deux Users pour le rôle Joueur

\_userManager.CreateAsync(new ApplicationUser

{

UserName = "SuperJoueur@ZombieParty.com",

Email = "SuperJoueur@ZombieParty.com",

NickName = "Super Joueur",

PhoneNumber = "1111111111",

EmailConfirmed = true,

}, "Joueur123\*").GetAwaiter().GetResult();

ApplicationUser user4 = \_db.ApplicationUsers.FirstOrDefault(u => u.Email == "SuperJoueur@ZombieParty.com");

\_userManager.AddToRoleAsync(user4, AppConstants.PlayerRole)

.GetAwaiter().GetResult();

\_userManager.CreateAsync(new ApplicationUser

{

UserName = "JoueurPourri@ZombieParty.com",

Email = "JoueurPourri@ZombieParty.com",

NickName = "Joueur Pourri",

PhoneNumber = "1111111111",

EmailConfirmed = true,

}, "Joueur123\*").GetAwaiter().GetResult();

ApplicationUser user5 = \_db.ApplicationUsers.FirstOrDefault(u => u.Email == "JoueurPourri@ZombieParty.com");

\_userManager.AddToRoleAsync(user5, AppConstants.PlayerRole)

.GetAwaiter().GetResult();

}

}

## Modifier les injections afin d’inclure le rôle

1. Ouvez le fichier *Program.cs*

builder.Services.AddDefaultIdentity<IdentityUser, IdentityRole>()

.AddEntityFrameworkStores<ZombiePartyDbContext>();

builder.Services.AddScoped<IDbInitializer, DbInitializer>();

1. Injectez le **IDbInitialiser** dans *Program.cs*

builder.Services.AddScoped<IDbInitializer, DbInitializer>();

builder.Services.AddRazorPages();

1. Ajoutez le Seed de base de données

void SeedDatabase()

{

using (var scope = app.Services.CreateScope())

{

var dbInitializer = scope.ServiceProvider.GetRequiredService<IDbInitializer>();

dbInitializer.Initialize();

}

}

app.UseAuthentication();

app.UseAuthorization();

SeedDatabase();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

app.MapRazorPages();

app.Run();

1. Insérez la gestion du **Cookie**

builder.Services.AddScoped<IZombieService, ZombieService>();

#endregion

builder.Services.ConfigureApplicationCookie(options => {

options.LoginPath = $"/Identity/Account/Login";

options.LogoutPath = $"/Identity/Account/Logout";

options.AccessDeniedPath = $"/Identity/Account/AccessDenied";

});

var app = builder.Build();

# Ajouter des restrictions par autorisations

## Vues Index et Privacy

1. Ouvrez le contrôleur **Home**
2. Ajoutez des autorisations pour les actions **Index** et **Privacy**
   * Index : Accessible à tous
   * Privacy : doit être authentifié (« loggin »)

Essayez par vous-même

[AllowAnonymous]

public IActionResult Index()

{

[Authorize]

public IActionResult Privacy()

{

1. Testez en vous connectant à différents rôles

## Vues pour Weapons

1. Ouvrez le contrôleur **Weapon**
2. Ajoutez des autorisations pour le contrôleur au complet Admin seulement

[Authorize(Roles = AppConstants.AdminRole)]

1. Testez en vous connectant à différents rôles

## Vues pour HuntingLog

1. Ouvrez le contrôleur **HuntingLog**
2. Ajoutez des autorisations pour les actions en utilisant les constantes de rôles créées précédemment :
   * Index : Accessible à tous
   * Upsert : Chasseurs et admin seulement

[Authorize(Roles = AppConstants.HunterRole + "," + AppConstants.AdminRole)]

1. Testez en vous connectant à différents rôles
2. Dans la vue **HuntingLog/Index**, ajoutez le code qui permet de n’afficher le bouton **Create New** que si l’utilisateur a le rôle admin ou chasseur.

@if(this.User.IsInRole(AppConstants.AdminRole) || this.User.IsInRole(AppConstants.HunterRole))

{

<div class="col-4 text-right">

<**a** class="btn btn-success" **asp-controller**="HuntingLog" **asp-action**="Upsert">

<i class="fas fa-plus-square"></i>&nbsp;@Locals["New"]

</**a**>

</div>

}

## Commentaires et validation (Commit) des changements dans le code

1. Faites un dernier **Commit**
2. "Poussez" **Push** les modifications sur le repo en ligne **Remote**: cliquez sur 
3. Merge la branche **FCT \_ Autorisations** dans la branche **Main** et faites un **dernier push.**

**Fin du laboratoire!**