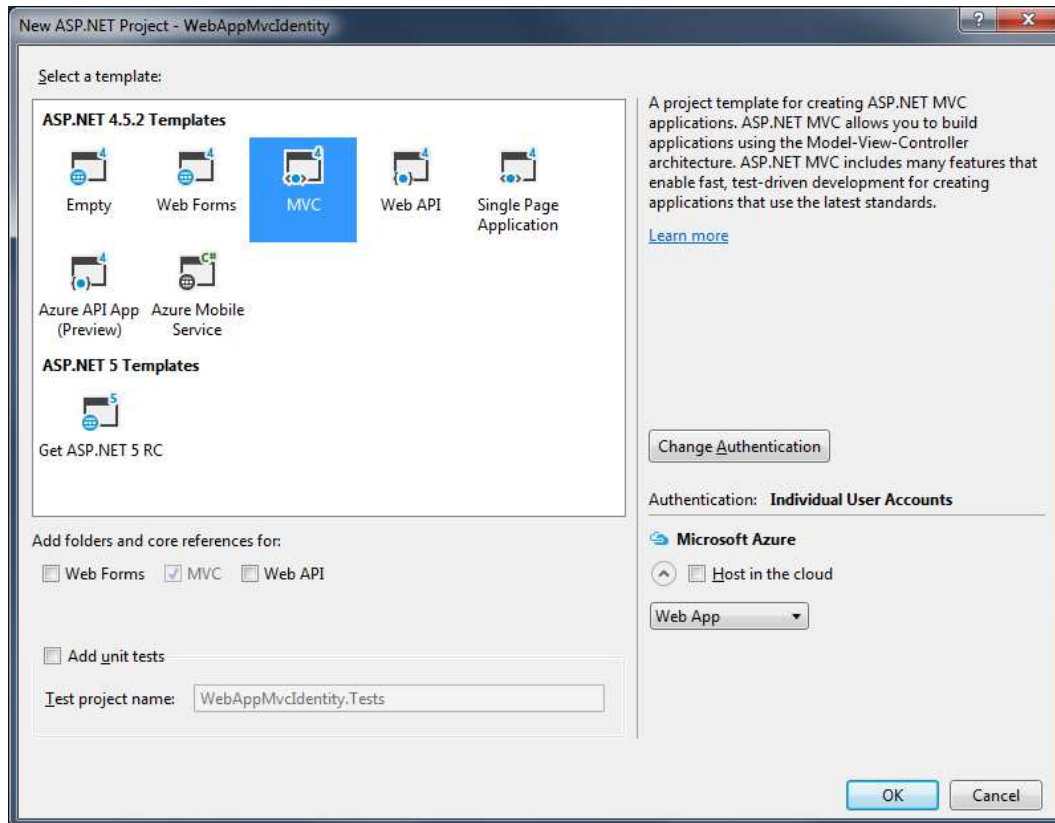


Authentification et autorisation d'une application web ASP.NET MVC 5 avec ASP.NET Identity 2.0

Créer une solution sur base du template de base asp mvc 5 en laissant l'authentification « Individual User Accounts » :



1. Via code first créer les tables

AspNet.Identity utilise EntityFramework pour créer les tables et les utiliser.

En démarrant l'application web une première fois, et en enregistrant un nouvel utilisateur, la structure des tables nécessaires sera créée automatiquement.

Avant cela, nous devons modifier la connectionString du web.config qui est utilisée par AspNet.Identity pour utiliser notre base de données.

La connectionString utilisée est visible dans le constructeur de la classe `ApplicationDbContext` du fichier `IdentityModels.cs` (dossier `Models`) :

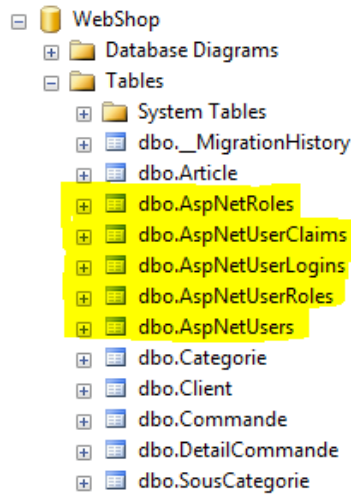
```
public ApplicationDbContext()
{
    : base("DefaultConnection", throwIfV1Schema: false)
}
```

Et naturellement on la retrouve dans le web.config :

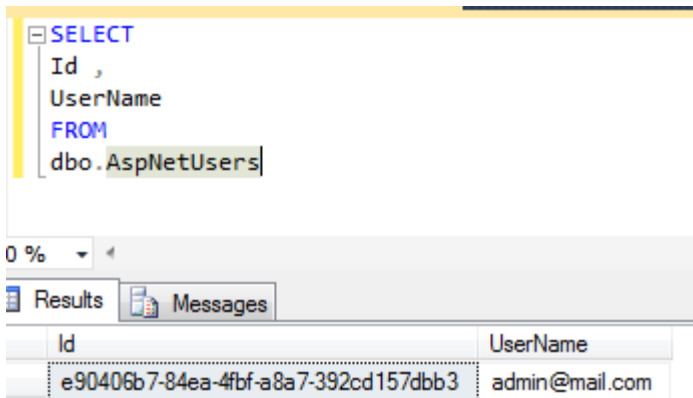
```
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data Source=localhost;Initial
Catalog=WebShop;Integrated Security=True" . . .
</connectionStrings>
```

NB : on peut modifier ce nom, il est possible également d'avoir plusieurs connectionString dans le web.config.

Je lance l'application et crée un premier user « admin@mail.com » via le lien « Register », et comme attendu, les tables ont été créées :



Je retrouve bien mon utilisateur dans la table dbo.AspNetUsers.



Remarques

Il est possible de redéfinir le nom des tables et/ou du schéma utilisé par la DLL

Microsoft.AspNet.Identity.

Pour modifier la chose, on écrira dans IdentityModels.cs et dans la classe `ApplicationDbContext` :

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    // redéfinir le schéma
    modelBuilder.HasDefaultSchema("tb");
    // et le nom des tables si besoin
    //modelBuilder.Entity<ApplicationUser>().ToTable("UserTableName");
    //modelBuilder.Entity<IdentityRole>().ToTable("RoleTableName");
    //modelBuilder.Entity<IdentityUserRole>().ToTable("UserRoleTableName");
    //modelBuilder.Entity<IdentityUserClaim>().ToTable("UserClaimTableName");
    //modelBuilder.Entity<IdentityUserLogin>().ToTable("UserLoginTableName");
}
```

Par défaut le mot de passe doit vérifier une certaine complexité, on peut modifier cela via le fichier

IdentityConfig.cs du dossier App_Start, précisément dans la méthode Create :

```
manager.PasswordValidator = new PasswordValidator
{
    RequiredLength = 6,
    RequireNonLetterOrDigit = true,
    RequireDigit = true,
    RequireLowercase = true,
    RequireUppercase = true,
};
...
}
```

Dans cette même méthode du fichier IdentityConfig.cs on peut aussi modifier les contraintes relatives au login de connexion :

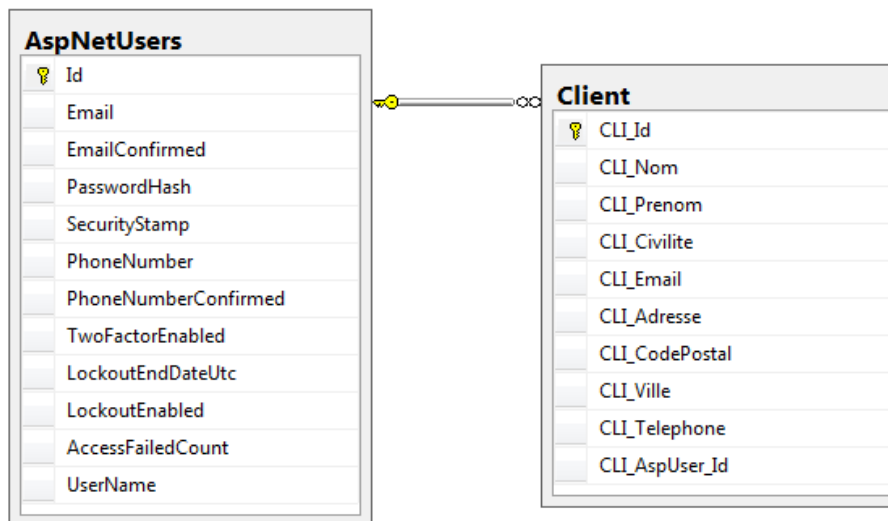
```

    ...
    manager.UserValidator = new UserValidator<ApplicationUser>(manager)
    {
        AllowOnlyAlphanumericUserNames = false,
        RequireUniqueEmail = true
    };
    ...

```

2. Lier les users ASP à mon business

Dans mon application, les clients doivent créer un compte pour pouvoir acheter un article. Je vais donc ajouter dans ma table Client une foreign key vers la table AspNetUsers.



Dans mon application, je vais utiliser la vue Register.cshtml pour créer un compte client, il y aura donc en réalité création d'un user ASP et d'un enregistrement dans la table client.

Je modifie la classe RegisterViewModel en conséquence en y ajoutant les propriétés du client :

```

public class RegisterViewModel
{
    ...

    [Required]
    public string Nom { get; set; }
    public string Prenom { get; set; }
    [Required]
    public string Civilite { get; set; }
    [Required]
    public string Adresse { get; set; }
    [Required]
    public string CodePostal { get; set; }
    [Required]
    public string Ville { get; set; }
    [Required]
    public string Telephone { get; set; }
}

```

Et j'ajoute ces propriétés dans la vue.

Create a new account.

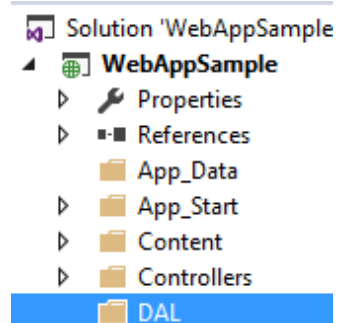
Civilite	<input type="text"/>
Nom	<input type="text"/>
Prenom	<input type="text"/>
Email	<input type="text"/>
Adresse	<input type="text"/>
CodePostal	<input type="text"/>
Ville	<input type="text"/>
Telephone	<input type="text"/>
Password	<input type="password"/>
Confirm password	<input type="password"/>

Register

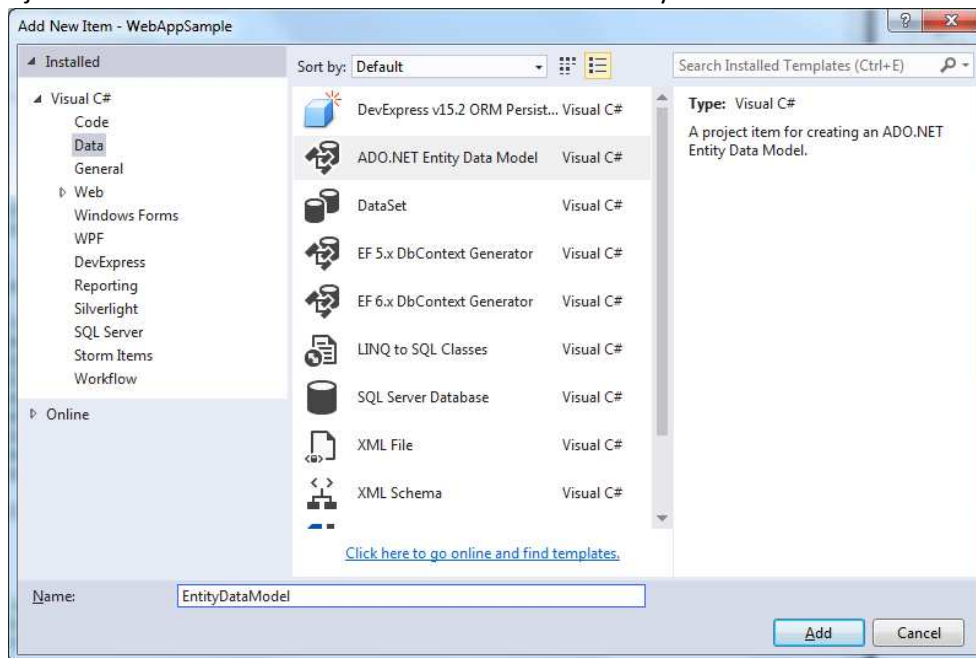
Modifions l'action POST Register de l'AccountController pour y intégrer la création du client.

Pour créer un nouveau client, utilisons pour changer EntityFramework.

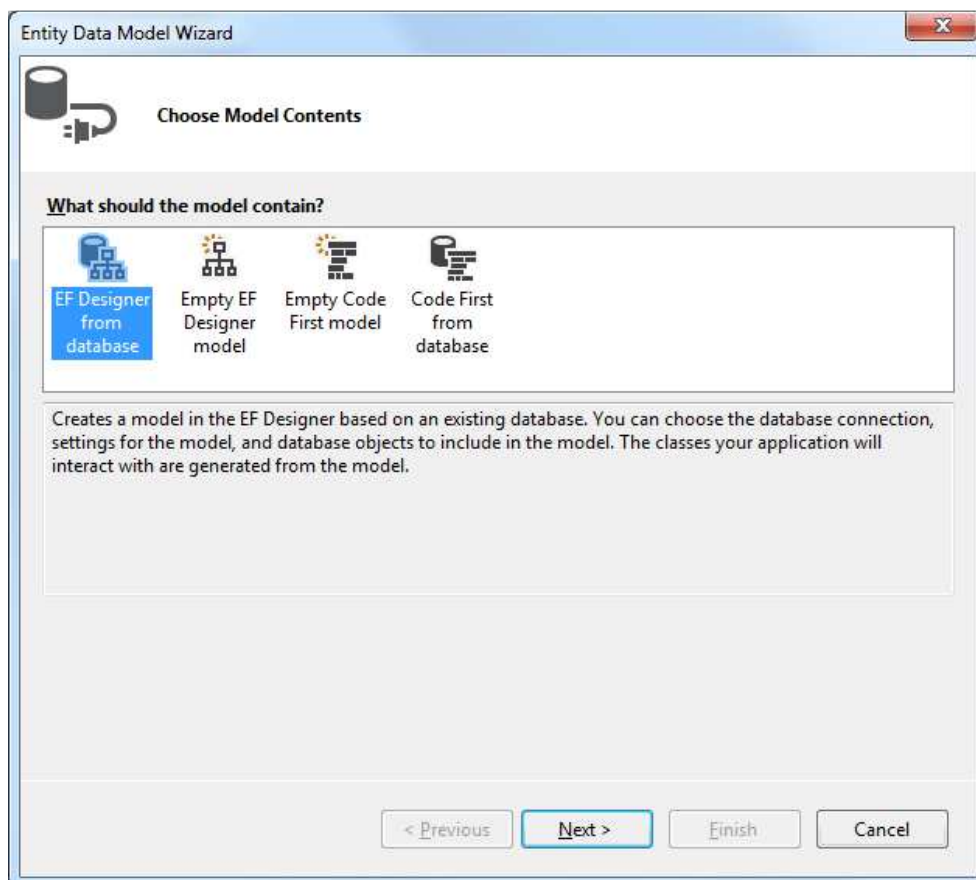
Pour être relativement propre, créons un dossier « DAL » (pour Data Access Layer) dans le projet :



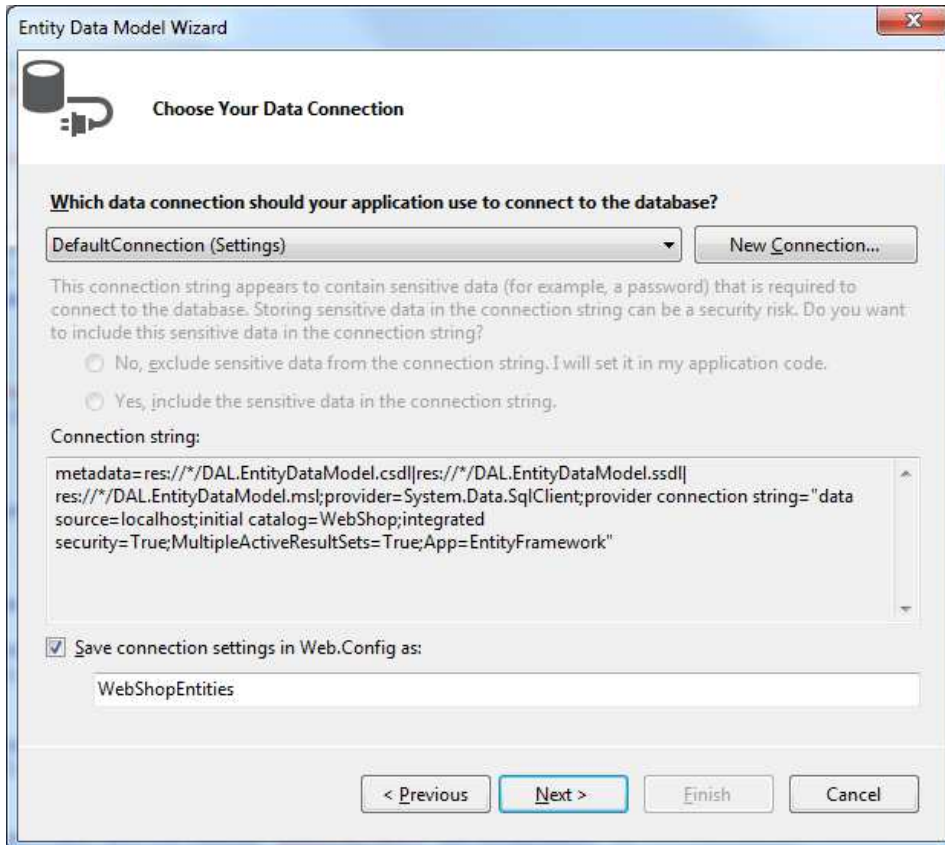
Ajoutons à ce dossier « DAL » un item « ADO.NET Entity Data Model » :



Sélectionner « EF Designer from database » pour générer nos classes de l'EntityFramework à partir de la base de données :



Sélectionner la connexion à la base de données :



The screenshot shows the 'Entity Data Model Wizard' window, specifically the 'Choose Your Data Connection' step. The window has a title bar with 'Entity Data Model Wizard' and a close button. Below the title bar is a header area with a database icon and the text 'Choose Your Data Connection'. The main content area is titled 'Which data connection should your application use to connect to the database?'. It features a dropdown menu set to 'DefaultConnection (Settings)' and a 'New Connection...' button. Below this, a text box explains that the connection string might contain sensitive data and asks if the user wants to include it. Two radio buttons are present: 'No, exclude sensitive data from the connection string. I will set it in my application code.' (selected) and 'Yes, include the sensitive data in the connection string.'. A text area labeled 'Connection string:' contains the following text: `metadata=res://*/DAL.EntityDataModel.csdl|res://*/DAL.EntityDataModel.ssdl|res://*/DAL.EntityDataModel.msl;provider=System.Data.SqlClient;provider connection string="data source=localhost;initial catalog=WebShop;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"`. Below the text area is a checkbox 'Save connection settings in Web.Config as:' which is checked, followed by a text box containing 'WebShopEntities'. At the bottom are four buttons: '< Previous', 'Next >', 'Finish', and 'Cancel'.

Entity Data Model Wizard

Choose Your Data Connection

Which data connection should your application use to connect to the database?

DefaultConnection (Settings) New Connection...

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☐ No, exclude sensitive data from the connection string. I will set it in my application code.

☐ Yes, include the sensitive data in the connection string.

Connection string:

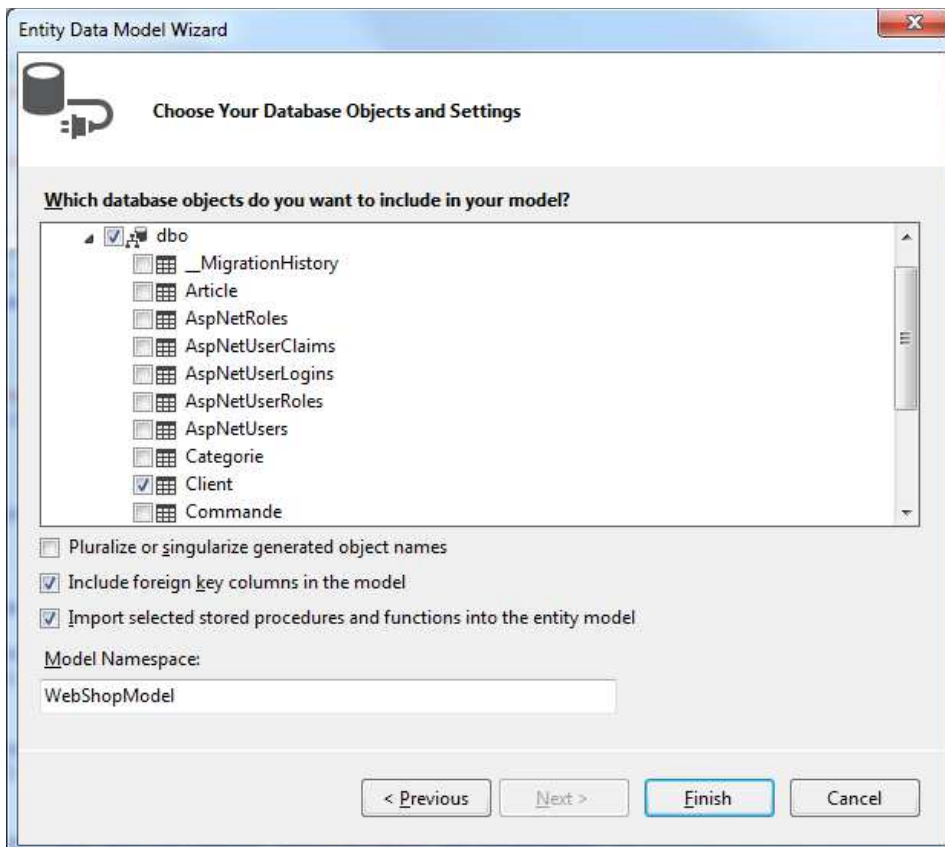
```
metadata=res://*/DAL.EntityDataModel.csdl|res://*/DAL.EntityDataModel.ssdl|
res://*/DAL.EntityDataModel.msl;provider=System.Data.SqlClient;provider connection string="data
source=localhost;initial catalog=WebShop;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
```

☒ Save connection settings in Web.Config as:

WebShopEntities

< Previous Next > Finish Cancel

Et ensuite la table client :



The screenshot shows the 'Entity Data Model Wizard' window, specifically the 'Choose Your Database Objects and Settings' step. The window has a title bar with 'Entity Data Model Wizard' and a close button. Below the title bar is a header area with a database icon and the text 'Choose Your Database Objects and Settings'. The main content area is titled 'Which database objects do you want to include in your model?'. It features a tree view with a folder 'dbo' expanded, showing a list of database objects: '_MigrationHistory', 'Article', 'AspNetRoles', 'AspNetUserClaims', 'AspNetUserLogins', 'AspNetUserRoles', 'AspNetUsers', 'Categorie', 'Client' (checked), and 'Commande'. Below the tree view are three checkboxes: 'Pluralize or singularize generated object names' (unchecked), 'Include foreign key columns in the model' (checked), and 'Import selected stored procedures and functions into the entity model' (checked). A text box labeled 'Model Namespace:' contains 'WebShopModel'. At the bottom are four buttons: '< Previous', 'Next >', 'Finish', and 'Cancel'.

Entity Data Model Wizard

Choose Your Database Objects and Settings

Which database objects do you want to include in your model?

dbo

- ☐ _MigrationHistory
- ☐ Article
- ☐ AspNetRoles
- ☐ AspNetUserClaims
- ☐ AspNetUserLogins
- ☐ AspNetUserRoles
- ☐ AspNetUsers
- ☐ Categorie
- ☒ Client
- ☐ Commande

☐ Pluralize or singularize generated object names

☒ Include foreign key columns in the model

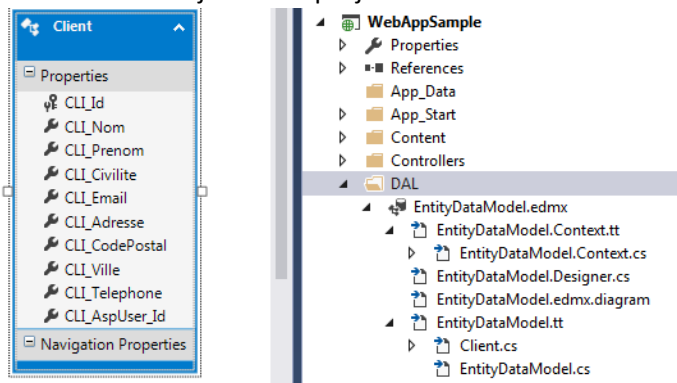
☒ Import selected stored procedures and functions into the entity model

Model Namespace:

WebShopModel

< Previous Next > Finish Cancel

Des fichiers ont été ajoutés au projet dans le dossier « DAL » :



Une nouvelle `ConnectionString` a également été ajoutée dans le `web.config` :

```
<add name="WebShopEntities"
connectionString="metadata=res://*/DAL.EntityDataModel.csdl|res://*/DAL.EntityDataModel.ssdl|res://*/DAL.EntityDataModel.msl;provider=System.Data.SqlClient;provider connection string='data source=localhost;initial catalog=WebShop;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework';"
providerName="System.Data.EntityClient" />
```

A ce stage, le plus intéressant pour nous est le fichier « .edmx » qui affiche en design les classes de l'EntityFramework qui sont mappées avec la base de données et le fichier « .Context » :

```
namespace WebAppSample.DAL
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    public partial class WebShopEntities : DbContext
    {
        public WebShopEntities()
            : base("name=WebShopEntities")
        {
        }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        public virtual DbSet<Client> Client { get; set; }
    }
}
```

Pour utiliser EntityFramework, il faudra simplement instancier la classe `WebShopEntities`.

Par exemple pour récupérer tous les clients on écrira :

```
using (DAL.WebShopEntities context = new DAL.WebShopEntities() )
{
    var lstClient = context.Client.ToList();
}
```

Et pour ajouter un client on écrira :

```
using (DAL.WebShopEntities context = new DAL.WebShopEntities() )
{
    context.Client.Add(
        new DAL.Client
        {
            CLI_Nom = "Hecquet",
            CLI_Prenom = "Jean-Pierre",
            CLI_Civilite = "Monsier",
            CLI_Email = "jp.hecquet@ephec.be",
            CLI_Adresse = "rue des Pommes, 11",
        }
    );
}
```

```

        CLI_CodePostal = "1070",
        CLI_Ville = "ANDERLECHT",
        CLI_Telephone = "02 599 99 99",
        CLI_AspUser_Id = "xxxxxx"
    });
    context.SaveChanges();
}

```

Revenons à nos moutons, et modifions l'action POST Register du controller AccountController pour ajouter le nouveau user créé dans la table client.

Par défaut l'action POST Register est implémentée comme suit :

```

public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);
            return RedirectToAction("Index", "Home");
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}

```

Le nouveau compte est créé si « result.Succeeded » est true.

C'est dans ce bloc « if » qu'il faut ajouter l'ajout du user dans la table client :

```

if (result.Succeeded)
{
    try
    {
        // ajout du user dans la table client
        using (DAL.WebShopEntities dal = new DAL.WebShopEntities())
        {
            dal.Client.Add(
                new DAL.Client
                {
                    CLI_Nom = model.Nom,
                    CLI_Prenom = model.Prenom,
                    CLI_Civilite = model.Civilite,
                    CLI_Email = model.Email,
                    CLI_Adresse = model.Adresse,
                    CLI_CodePostal = model.CodePostal,
                    CLI_Ville = model.Ville,
                    CLI_Telephone = model.Telephone,
                    CLI_AspUser_Id = user.Id
                });
            dal.SaveChanges();
        }
    }
    catch (Exception)
    {
        // client n'est pas ajouté delete l'asp user créé
        await UserManager.DeleteAsync(user);
        ModelState.AddModelError("", "Echec création client, veuillez réessayer");
        return View(model);
    }
    // on affecte le rôle client
    UserManager.AddToRole(user.Id, "client");

    await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);

    return RedirectToAction("Index", "Home");
}

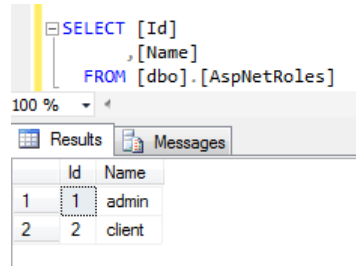
```


Notez que l'ajout du client est dans un bloc « try ... catch », en cas d'erreur le user est supprimé

```
catch (Exception)
{
    // client n'est pas ajouté delete l'asp user créé
    UserManager.Delete(user);
    ModelState.AddModelError("", "Echec création client, veuillez réessayer");
    return View(model);
}
```

Notez aussi l'affectation du rôle « client » au user : `UserManager.AddToRole(user.Id, "client");`

Attention, pour attribuer ce rôle, il faut l'avoir ajouté dans la table `dbo.AspNetRoles` :



Id	Name
1	admin
2	client

Lorsqu'un user est connecté (authentifié), on récupère facilement son Id de la table `AspNetUsers` :

```
using Microsoft.AspNet.Identity ;
User.Identity.GetUserId() ;
```

Si le user n'est pas authentifié, la fonction renvoie null.

On peut préalablement vérifier l'authentification via la propriété `User.Identity.IsAuthenticated`.

3. Autorisation

Le client connecté pourra acheter des articles, consulter ses commandes, ...

Les controllers qui implémentent ces fonctionnalités seront protégés par l'attribut `Authorize` défini au niveau du controller :

```
[Authorize]
public class MonController : Controller
{
    ...
}
```

Si une action d'un controller protégé doit être néanmoins accessible pour tous on utilisera l'attribut `AllowAnonymous` pour l'action concernée :

```
[AllowAnonymous]
public ActionResult MonAction()
{
    ...
}
```

A l'inverse, on peut décider de protéger une action dans un controller non protégé :

```
public class MonController : Controller
{
    [Authorize]
    public ActionResult MonAction()
    {
        ...
    }
    ...
}
```

4. Rôle

Comme vu ci-avant, il est possible d'attribuer un rôle à un user asp.

On ajoutera les rôles directement dans la base de données dans la table `AspNetRoles`, et le rôle joué par un user dans la table `AspNetUserRoles`.

En se connectant à l'application avec un user qui a le rôle « admin », on constate que son rôle est bien renvoyé :

```
public ActionResult Contact()
{
    ViewBag.Message = "Your contact page.";
    if (User.Identity.IsAuthenticated && User.IsInRole("Admin"))
    {
        ViewBag.Message += " And I am an Admin !!!";
    }

    return View();
}
```

Il suffit donc d'appeler la méthode `IsInRole(string role)` du namespace `System.Security.Principal`. Attention, la méthode est sensible à la casse.

On peut dès lors protéger un controller et/ou une action selon le rôle :

```
[Authorize(Roles = "Admin")]
public ActionResult ActionForAdmin()
{
    ...
}
```

Essayons en tant que « Client » d'atteindre la page contact qui est réservée aux « admin ».

Nous sommes redirigés vers la page de login, ce qui n'est pas très élégant puisque nous sommes déjà connectés.

Si on « augmente » l'accès à la page contact, plus de problème :

```
[Authorize(Roles = "Admin, Client")]
```

Pour rediriger vers une page spécifique lorsqu'on essaie d'atteindre une page inaccessible pour notre rôle on peut surcharger la classe `AuthorizeAttribute` :

```
public class AuthorizeCustomAttribute : AuthorizeAttribute
{
    public AuthorizeCustomAttribute()
    {
        View = "AuthorizeFailed";
    }

    public string View { get; set; }

    /// <summary>
    /// Check for Authorization
    /// </summary>
    /// <param name="filterContext"></param>
    public override void OnAuthorization(AuthorizationContext filterContext)
    {
        base.OnAuthorization(filterContext);
        IsUserAuthorized(filterContext);
    }

    /// <summary>
    /// Method to check if the user is Authorized or not
    /// if yes continue to perform the action else redirect to error page
    /// </summary>
    /// <param name="filterContext"></param>
    private void IsUserAuthorized(AuthorizationContext filterContext)
```

```

{
    // If the Result returns null then the user is Authorized
    if (filterContext.Result == null)
        return;

    //If the user is Un-Authorized then Navigate to Auth Failed View
    if (filterContext.HttpContext.User.Identity.IsAuthenticated)
    {
        var vr = new ViewResult();
        vr.ViewName = View;
        ViewDataDictionary dict = new ViewDataDictionary();
        dict.Add("Message", "Sorry you are not Authorized to Perform this Action");
        vr.ViewData = dict;
        var result = vr;
        filterContext.Result = result;
    }
}
}

```

Il faut également créer une view AuthorizeFailed.cshtml dans le dossier Shared :

```

@{
    ViewBag.Title = "AuthorizeFailed";
}
<h2>AuthorizeFailed</h2>
@ViewData["Message"]

```

Ensuite, on utilisera l'attribut custom pour protéger un controller ou une action :

```

[AuthorizeCustom(Roles = "Admin")]
public ActionResult Contact()
{
    ...
}

```

Remarque :

On peut également protéger un controller ou une action en spécifiant l'utilisateur :

```

[Authorize(Users = "User1, User2, ...")]

```

Divers (utile pour la réalisation de votre projet)

1. Poster une liste d'objet

Une vue liste des cours

```

@using (Html.BeginForm())
{
    for (int i = 0; i < Model.Count(); i++)
    {
        <tr>
            <td>
                @Html.TextBoxFor(x => Model.ToList()[i].COUR_Id)
            </td>
            <td>
                @Html.TextBoxFor(x => Model.ToList()[i].COUR_Libelle)
            </td>
            <td>
                @Html.TextBoxFor(x => Model.ToList()[i].COUR_Ponderation)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) |
                @Html.ActionLink("Details", "Details", new { /* id=item.PrimaryKey */ }) |
                @Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })
            </td>
        </tr>
    }
    <tr>
        <td colspan="4"><input type="submit" value="Save" /></td>
    </tr>
}

```

Et l'action POST associée. Vérifier en DEBUG que le model est bien complété

```
[HttpPost]
public ActionResult Index(List<CoursModels> model)
{
    return View(BL.CoursBL.GetAllCours());
}
```

2. Session

Pour stocker des informations dans la session :

```
Session["ItemSession"] = _lst;
```

Et pour récupérer l'information stockée dans la session :

```
(List<ObjetModel>) Session["ItemSession"]
```