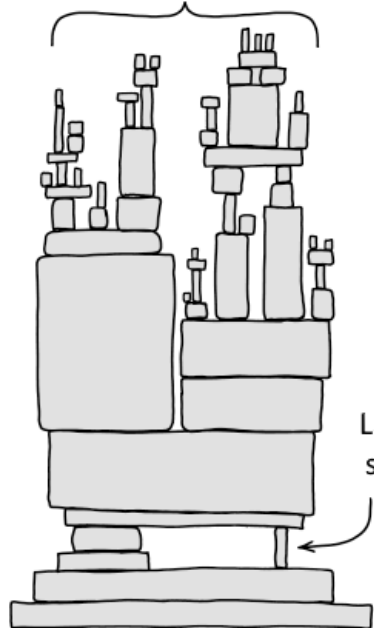


# Semaine 09 Partie 2

## Gestion des utilisateurs

Bases de données et programmation Web

15 projets qui utilisent  
une même BD



La BD qu'on  
s'apprête à  
modifier



- ❖ Gestion des utilisateurs
- ❖ Validation du ModelState personnalisée



## ❖ Gestion des utilisateurs

◆ En programmation Web, vous avez principalement utilisé Entity Framework **Identity** pour la gestion des utilisateurs. Dans ce cours, nous allons créer la table d'utilisateurs *from scratch*. C'est une opportunité de pratiquer certains concepts qui visent à protéger les données :

- Le **hachage** 
- Le **salage** 
- Le **chiffrement** 



- ◆ Le **hachage** et le **salage**: Ces deux techniques sont utilisées pour encrypter de façon sécuritaire des mots de passe.
- ◆ Le **chiffrement**: Cette technique est utilisée pour les informations importantes qu'on veut sécuriser mais qui ne sont pas appelées à changer.



## ❖ Table d'utilisateurs

### ◆ Voici un exemple de table d'utilisateurs

```
CREATE TABLE Utilisateurs.Utilisateur(  
    UtilisateurID int IDENTITY(1,1),  
    Pseudonyme nvarchar(50) NOT NULL,  
    → MotDePasseHache varbinary(32) NOT NULL,  
    → MdpSel varbinary(16) NOT NULL,  
    Email nvarchar(256) NOT NULL,  
    → NAS varbinary(max) NOT NULL  
    CONSTRAINT PK_Utilisateur_UtilisateurID PRIMARY KEY (UtilisateurID)  
);  
GO
```

- Ces trois colonnes ne sont pas comme les autres : elles ont un *secret obscur*.
  - Le **mot de passe** est stocké de manière hachée avec du **sel** et ne sera jamais enregistré déchiffré.
  - Le **NAS** est stocké de manière chiffrée et pourra être déchiffré par l'application, au besoin.



## ❖ La gestion du mot de passe

- ◆ Le **hachage** et le **salage**: Ces deux techniques sont utilisées pour encrypter de façon sécuritaire des mots de passe.
- ◆ Dans la table utilisateur, nous avons deux champs reliés au mot de passe:
  - Le mot de passe haché, de type varbinary(32)
  - Le sel qui sera de type varbinary(16)

```
| → MotDePasseHache varbinary(32) NOT NULL,  
| → MdpSel varbinary(16) NOT NULL,
```



### ❖ La gestion du mot de passe

- ◆ L'inscription: L'utilisateur va s'inscrire avec un mot de passe dans un formulaire. Quand on va traiter le formulaire, on va créer un sel et encrypter son mot de passe haché qu'on va enregistrer dans la table Utilisateur de la BD.
- ◆ La connexion: Puis l'utilisateur va se connecter dans un formulaire. Quand on va traiter le formulaire, on va utiliser le sel enregistré pour de nouveau encrypter le mot de passe fourni et on pourra vérifier que le résultat obtenu est le même mot de passe haché que celui qu'on avait enregistré auparavant dans la BD.




## ❖ La gestion du mot de passe

- ◆ L'inscription: L'utilisateur va s'inscrire avec un mot de passe et on va créer un sel et encrypter son mot de passe haché qu'on va enregistrer dans la table Utilisateur de la BD.
- ◆ Mot de passe haché
  - **L'utilisateur s'inscrit** : On prend son mot de passe « Patate1234 », on lui ajoute un sel aléatoire : « Patate1234AB8201F039E91C7900AAB2... », on hache le mot de passe avec un algorithme jugé sécuritaire (SHA-256, SHA-3, BLAKE2, Argon2, bcrypt, etc.) : « 3810CBAA7200FFD83810A... » et on stocke cette valeur hachée dans la BD.





### ❖ La gestion du mot de passe

- ◆ La connexion: Puis l'utilisateur va se connecter et on va utiliser le sel enregistré pour de nouveau encrypter le mot de passe fourni et on pourra vérifier que le résultat obtenu est le même mot de passe haché que celui qu'on avait enregistré auparavant dans la BD (lors de l'inscription)
- ◆ **L'utilisateur se connecte** : On prend le mot de passe qu'il tente « Patate1234 », on lui ajoute le même sel (qu'on a gardé en mémoire lors de l'inscription) : « Patate1234AB8201F039E91C7900AAB2... » puis on hache le mot de passe avec le même algorithme que lors de l'inscription. Est-ce que le résultat obtenu est identique à la valeur hachée stockée dans la BD ? **Oui ?** -> Authentification réussie ! 



### ❖ Table d'utilisateurs

```
MotDePasseHache varbinary(32) NOT NULL,  
MdpSel varbinary(16) NOT NULL,
```

- ◆ Le **mot de passe** ne sera **jamais stocké en clair** et même les administrateurs de la BD ne peuvent pas connaître le mot de passe facilement.
- ◆ Le **sel** est **stocké en clair**. Un sel différent a été obtenu de manière **aléatoire** pour chaque utilisateur, mais il est **fixe** pour l'utilisateur une fois choisi.



```
MotDePasseHache varbinary(32) NOT NULL,  
MdpSel varbinary(16) NOT NULL,
```

## ❖ Table d'utilisateurs

### ◆ Mot de passe haché

- À quoi sert le sel ?
  - Une grande portion d'utilisateurs ont tendance à utiliser **un ensemble très populaire de mots de passe faibles**. (**password123**, **password**, **123456**, **qwerty**, etc.) Ça veut dire que sans sel, les hachages de ces mots de passe très populaires deviennent **méga fréquents**. Si un attaquant a accès aux mots de passe hachés, il pourrait deviner les mots de passe très populaires car leur hachage est « célèbre ». Le **sel**, qui est pseudo-aléatoire et surtout **unique à chaque utilisateur**, mitige ce problème et élimine les hachages « célèbres ». (L'attaque décrite s'appelle **Rainbow table attack**. L'attaquant utilise une table de milliers de **hachages populaires** pour les comparer à ceux dans la BD)
- Pourquoi le sel est stocké en clair ?
  - Le sel **n'est pas secret**. Son objectif est seulement **d'ajouter de l'aléatoire dans les hachages**. (Éviter de le dévoiler inutilement est quand même logique)
- Pourquoi le mot de passe est VARBINARY(32) et le sel est VARBINARY(16) ?
  - **32** : Les algorithmes de hachage produisent un output de taille fixe. (Ex : SHA-256 -> 32 bytes)
  - **16** : Un sel de 16 bytes est suffisant pour créer amplement d'aléatoire.



## ❖ Insérer un utilisateur dans la table

### ◆ Exemple pour le hachage :

```
SELECT CRYPT_GEN_RANDOM(16)  
AS 'Sel aléatoire de 16 bytes';
```

Sel aléatoire de 16 bytes

0x4A4B74260D69B4171A4AF186150A52A5

```
SELECT HASHBYTES('SHA2_256', '209841098ACCEFFF035123561AAFFEEEB2')  
AS 'Donnée hachée avec SHA 256'
```

Donnée hachée avec SHA 256

0xAD60973DA8A47F0FE491003D18A83D52E1526B4D9D9234...

- Exemple de **procédure** qui **INSERT** un nouvel utilisateur avec son **mot de passe haché** et son **sel aléatoire**.

- Nous n'avons pas encore **chiffré le NAS**. On fait ça dans les prochaines diapos.

```
CREATE TABLE Utilisateurs.Utilisateur(  
    UtilisateurID int IDENTITY(1,1),  
    Pseudonyme nvarchar(50) NOT NULL,  
    MotDePasseHache varbinary(32) NOT NULL,  
    MdpSel varbinary(16) NOT NULL,  
    Email nvarchar(256) NOT NULL,  
    NAS varbinary(max) NOT NULL,
```

```
CREATE PROCEDURE Utilisateurs.USB_CreerUtilisateur  
    @Pseudonyme nvarchar(50),  
    @MotDePasse nvarchar(100),  
    @NAS char(9),  
    @Email nvarchar(256)  
AS  
BEGIN  
    -- Sel aléatoire  
    DECLARE @MdpSel varbinary(16) = CRYPT_GEN_RANDOM(16);  
  
    -- Concaténation mdp + sel  
    DECLARE @MdpEtSel nvarchar(116) = CONCAT(@MotDePasse, @MdpSel);  
  
    -- Hachage du mot de passe  
    DECLARE @MdpHache varbinary(32) = HASHBYTES('SHA2_256', @MdpEtSel);  
  
    -- Insertion  
    INSERT INTO Utilisateurs.Utilisateur (Pseudonyme, MotDePasseHache, MdpSel, Email, NAS)  
    VALUES  
    (@Pseudonyme, @MdpHache, @MdpSel, @Email, 'NAS pas géré encore');  
END
```



- ◆ Le **chiffrement**: Cette technique est utilisée pour les informations importantes qu'on veut sécuriser mais qui ne sont pas appelées à changer.
- ◆ Pour la table Utilisateurs, nous allons utiliser la technique du chiffrement pour le NAS.



## ❖ Table d'utilisateurs

- ◆ NAS (ou autre donnée sensible) chiffré NAS varbinary(max) NOT NULL,
  - **L'utilisateur fournit une information sensible** : « 442120982 » puis on chiffre la donnée à l'aide d'un algorithme de chiffrement symétrique (AES, Twofish, ChaCha20, etc.) et d'une clé secrète : « 12098ABC00FF826181CAA393... » et on stocke cette donnée chiffrée. Pas nécessaire d'utiliser du sel car l'algorithme de chiffrement ET la clé secrète apportent déjà assez d'aléatoire.
  - **Le système a éventuellement besoin de cette information** : On déchiffre la donnée stockée avec la même clé secrète que pour le chiffrement. C'est tout : on a récupéré la donnée sensible initiale.
  - Pourquoi varbinary(max) ? L'output d'un algorithme de chiffrement qui travaille « par bloc » comme AES n'est pas fixe. (Contrairement à une fonction de hachage) Pour ne pas se mettre à calculer la taille de l'output, utiliserons simplement varbinary(max). De toute façon, le type varbinary s'adaptera à la taille occupée par la donnée.



NAS varbinary(max) NOT NULL,

## ❖ Table d'utilisateurs

- ◆ NAS (ou autre donnée sensible) chiffré
  - Mieux un algorithme de chiffrement **symétrique** ou **asymétrique** ? Pour cette situation, un **algorithme symétrique est mieux**, car beaucoup plus performant. Les deux sont **sécuritaires** cela dit. **Asymétrique** est généralement **plus sécuritaire** et est nécessaire dans certaines situations. (Communication, signatures digitales, etc.)
  - **La clé secrète** ... ? Elle sort d'où ? La même clé secrète est utilisée pour **chiffrer** et **déchiffrer**, pour **tous les utilisateurs**. (Donc la même clé pour TOUS, contrairement à un **sel**) Elle est donc ultra-méga-secrète et doit être conservée en sécurité à tout prix !
  - Où stocke-t-on la clé secrète ... ?
    - **Gros budget, grosse sécurité** : « **EKS** : Encrypted Key Store » ou « **HSM** : Hardware Security Module ». Ce sont des systèmes fait exprès pour stocker des clés.
    - **Budget modeste, sécurité modeste** : Dans un fichier chiffré avec un mot de passe, dans une table de BD avec un mot de passe.
  - Les applications qui ont besoin de la clé secrète symétrique peuvent y accéder dans le **EKS** / **HSM** / le **fichier** / la **table** à l'aide de permissions ou d'accès adaptés.



### ❖ Insérer un utilisateur dans la table

- ◆ Pour le chiffrement du NAS, il y a quelques préparatifs ...
  - Créer une **clé MASTER** pour la BD.
  - Créer un **certificat auto-signé**.
  - Créer une **clé symétrique** pour le chiffrement / déchiffrement.
- ◆ Après ça nous pourrons **chiffrer** et **déchiffrer** des données librement.





## ❖ Préparatifs pour créer une clé symétrique

### ◆ Créer une clé MASTER :



```
USE Ma_BD;  
GO
```

- La **master key** est nécessaire pour pouvoir chiffrer et protéger les autres clés. (Dont la clé symétrique qu'on s'apprête à créer)

```
-- Créer une clé avec un mot de passe  
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'P4ssw0rd!';  
GO  
  
-- Vérifier l'existence de la clé (Facultatif)  
SELECT * FROM sys.symmetric_keys;
```

## ATTENTION:

ici au cégep, les PASSWORD doivent suivre les règles du cégep..

**Ce mot de passe est trop court! TROP COURT!**  
**Allongez-le....**



### ❖ Préparatifs pour créer une clé symétrique

- ◆ Une fois que la clé **MASTER** a été créée, il faut ensuite créer un certificat auto-signé.

#### ◆ Créer un **certificat auto-signé** :

```
-- Créer un certificat auto-signé
CREATE CERTIFICATE MonCertificat WITH SUBJECT = 'ChiffrementNAS';
GO

-- Vérifier l'existence du certificat (Facultatif)
SELECT * FROM sys.certificates;
```

- Le **certificat** est nécessaire pour authentifier le **détenteur d'une clé**. Nous en aurons besoin pour que la base de données sache que c'est bel et bien nous qui sommes en train d'essayer de **déchiffrer les données**.
- On aurait d'ailleurs pu spécifier une **date d'expiration** et un **mot de passe** pour le certificat, mais gardons ça simple.



## ❖ Préparatifs pour créer une clé symétrique

### ◆ Créer la clé symétrique

```
-- Créer une clé symétrique
CREATE SYMMETRIC KEY MaSuperCle WITH ALGORITHM = AES_256 ENCRYPTION BY CERTIFICATE MonCertificat;

-- Vérifier la clé symétrique (Facultatif)
SELECT * FROM sys.symmetric_keys;
```

- Voilà, nous allons être prêts à chiffrer des données. En général, ça ressemblera à ça :

```
-- Chiffrement de donnée
OPEN SYMMETRIC KEY MaSuperCle
    DECRYPTION BY CERTIFICATE MonCertificat;

DECLARE @ValeurChiffree varbinary(max) = EncryptByKey(KEY_GUID('MaSuperCle'), 'valeur top secrète');

CLOSE SYMMETRIC KEY MaSuperCle;
GO
```

Champ ou texte à chiffrer



## ❖ Insérer un utilisateur dans la table

◆ Le traitement du mot de passe avec le **chiffrement du NAS** cette fois !

```
EXEC Utilisateurs.USB_CreerUtilisateur
    @Pseudonyme = 'max',
    @MotDePasse = 'Salut1!',
    @NAS = '111222333',
    @Email = 'm@m.m'
GO
```

```
CREATE PROCEDURE Utilisateurs.USB_CreerUtilisateur
    @Pseudonyme nvarchar(50),
    @MotDePasse nvarchar(100),
    @NAS char(9),
    @Email nvarchar(256)
AS
BEGIN
    -- Sels aléatoires
    DECLARE @MdpSel varbinary(16) = CRYPT_GEN_RANDOM(16);

    -- Concaténation de données et sel
    DECLARE @MdpEtSel nvarchar(116) = CONCAT(@MotDePasse, @MdpSel);

    -- Hachage du mot de passe
    DECLARE @MdpHachage varbinary(32) = HASHBYTES('SHA2_256', @MdpEtSel);

    -- Chiffrement du NAS
    OPEN SYMMETRIC KEY MaSuperCle
    DECRYPTION BY CERTIFICATE MonCertificat;

    DECLARE @NasChiffre varbinary(max) = EncryptByKey(KEY_GUID('MaSuperCle'), @NAS);

    CLOSE SYMMETRIC KEY MaSuperCle;

    -- Insertion
    INSERT INTO Utilisateurs.Utilisateur (Pseudonyme, MotDePasseHache, MdpSel, Email, NAS)
    VALUES
        (@Pseudonyme, @MdpHachage, @MdpSel, @Email, @NasChiffre);
END
GO
```



### ❖ La gestion du mot de passe

- ◆ **Nous avons terminé la partie de l'inscription:** L'utilisateur va s'inscrire avec un mot de passe et on va créer un sel et encrypter son mot de passe haché qu'on va enregistrer dans la table Utilisateur de la BD.
- ◆ **Nous abordons maintenant la partie de la connexion:** Puis l'utilisateur va se connecter et on va utiliser le sel enregistré pour de nouveau encrypter le mot de passe fourni et on pourra vérifier que le résultat obtenu est le même mot de passe haché que celui qu'on avait enregistré auparavant dans la BD (lors de l'inscription).



## ❖ Déchiffrer le NAS d'un utilisateur dans la table

◆ Hmmm, c'est bel et bien chiffré :

```
SELECT Pseudonyme, NAS FROM Utilisateurs.Utilisateur;
```

Pseudonyme	NAS
max	0x007F676BDBB83A42B10B7013A01A4F1E02000000A1333C1...

◆ Il faut faire une **procédure** pour déchiffrer le NAS. On peut déchiffrer ainsi :

```
OPEN SYMMETRIC KEY MaSuperCle  
    DECRYPTION BY CERTIFICATE MonCertificat;  
  
SELECT Pseudonyme, CONVERT(char(9), DecryptByKey(NAS)) AS NAS FROM Utilisateurs.Utilisateur  
  
CLOSE SYMMETRIC KEY MaSuperCle;  
GO
```

Pseudonyme	NAS
max	111222333



### ❖ Déchiffrer le NAS d'un utilisateur dans la table

- ◆ Dans le code précédent, la procédure va retourner deux champs: Pseudonyme et NAS.
- ◆ Nous aurons besoin de créer une table pour avoir un modele pour recevoir le résultat de l'exécution de la procédure avec `.FromSQLRaw`

Si vous avez créé une table `RetourUser` qui comprendra le champ `Pseudonyme nvarchar(20)` ET surtout le champ `NAS char(9)`. Nous aurons le modèle `RetourUsers` pour exécuter la procédure.

```
RetourUsers ? retourUser = (await _context.RetourUsers.FromSqlRaw(query, parameters.ToArray()).ToListAsync()).FirstOrDefault();
```



### ❖ Une **nouvelle clé** symétrique, à chaque fois !

```
-- Créer une clé symétrique  
CREATE SYMMETRIC KEY MaSuperCle WITH ALGORITHM = AES_256 ENCRYPTION BY CERTIFICATE MonCertificat;
```

- ◆ Dans vos TPs, comme vous changerez fréquemment de machine (donc de BD, physiquement) pour avancer votre travail, gardez à l'esprit qu'à chaque fois que vos migrations créeront une **clé symétrique**, **elle sera différente**.
  - C'est donc inutile de conserver des données chiffrées de votre session de travail précédente. (À moins de faire un **backup de la BD**, mais ce n'est pas le but : avec **Evolve**, pendant le développement de l'appli, c'est facile de recréer la BD)
  - Contentez-vous de réexécuter toutes vos **migrations** chaque fois que vous changez de machine pour utiliser le même ensemble initial de données de test. (et donc re-chiffrer, avec la nouvelle clé)





## ❖ Authentifier un utilisateur (Exemple avec une procédure)

- On prend le **mot de passe** *tenté* par l'utilisateur, on lui **concatène** le **sel** qui avait été utilisé pour **hacher** le mot de passe existant.. et on hache !
- Si le **hachage** obtenu est **identique** au mot de passe haché dans la BD : le mot de passe fourni est le bon !

```
-- Procédure authentification

CREATE PROCEDURE Utilisateurs.USP_AuthUtilisateur
    @Pseudo nvarchar(50),
    @MotDePasse nvarchar(50)
AS
BEGIN

    DECLARE @Sel varbinary(16);
    DECLARE @MdpHache varbinary(32);
    SELECT @Sel = Sel, @MdpHache = MotDePasseHache
    FROM Utilisateurs.Utilisateur
    WHERE Pseudo = @Pseudo;

    IF HASHBYTES('SHA2_256', CONCAT(@MotDePasse, @Sel)) = @MdpHache
    BEGIN
        SELECT * FROM Utilisateurs.Utilisateur WHERE Pseudo = @Pseudo;
    END
    ELSE
    BEGIN
        SELECT TOP 0 * FROM Utilisateurs.Utilisateur;
    END

END
GO
```

```
EXEC Utilisateurs.USP_AuthUtilisateur
@Pseudonyme = 'max', @MotDePasse = 'Salut1!'
```



Les infos de  
l'utilisateur



- ❖ Gestion des utilisateurs dans le projet Web
  - ◆ Configuration de l'authentification dans **Program.cs**

```
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme).AddCookie(options =>  
{  
    options.LoginPath = "/Utilisateurs/Connexion";  
    options.LogoutPath = "/Utilisateurs/Deconnexion";  
});
```

} /Controller/Action pour Connexion et Déconnexion

Juste après .AddDbContext... dans Program.cs

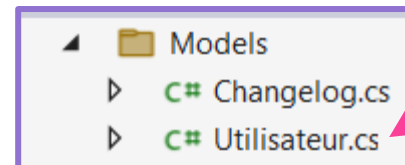
```
app.UseRouting();
```

```
app.UseAuthentication();
```

```
app.UseAuthorization();
```

N'oubliez pas `UseAuthentication()` un peu plus bas.

Notre table `Utilisateur` a été **scaffold** en Model



... C'est tout pour la configuration ! 🙌



## ❖ Gestion des utilisateurs dans le projet Web

- ◆ Contrôleur pour **inscription, connexion et déconnexion**
  - Créez un contrôleur vide au lieu de l'auto-générer. Les actions seront **atypiques**.

```
public class UtilisateursController : Controller
{
    readonly S09_LaboContext _context;
    public UtilisateursController(S09_LaboContext context)
    {
        _context = context;
    }
}
```

- Les exemples d'**actions** des prochaines diapos sont adaptées à cette **table d'utilisateurs**. Rien ne vous empêche d'ajouter / retirer des propriétés.

```
CREATE TABLE Utilisateurs.Utilisateur (
    UtilisateurID int IDENTITY(1,1),
    Pseudonyme nvarchar(50) NOT NULL,
    MotDePasseHache varbinary(32) NOT NULL,
    MdpSel varbinary(16) NOT NULL,
    Email nvarchar(256) NOT NULL,
    NAS varbinary(max) NOT NULL,
```



## ❖ Gestion des utilisateurs dans le projet Web

- ◆ **Inscription** : **ViewModel** pour envoyer les informations d'inscription de la vue au contrôleur

- Ce **ViewModel** ne fait référence à aucune table dans la BD et sert juste à **encapsuler** les données du formulaire d'inscription pour les envoyer au contrôleur.

- On doit se *gâter* sur les **[DataAnnotations]** pour **offrir le meilleur feedback possible** lors de la validation du formulaire d'inscription.

```
public class InscriptionViewModel
{
    [Required(ErrorMessage = "Un nom d'utilisateur est requis.")]
    5 references
    public string Pseudonyme { get; set; } = null!;

    [Required (ErrorMessage = "Un mot de passe est requis.")]
    [StringLength(50, MinimumLength = 6, ErrorMessage = "Le mot de passe doit avoir entre 6 et 50 caractères.")]
    [DataType(DataType.Password)]
    5 references
    public string MotDePasse { get; set; } = null!;

    [Required (ErrorMessage = "Veuillez confirmer le mot de passe.")]
    [DataType(DataType.Password)]
    [Compare(nameof(MotDePasse), ErrorMessage = "Les deux mots de passe sont différents.")]
    3 references
    public string MotDePasseConfirmation { get; set; } = null!;

    [Required (ErrorMessage = "Une adresse courriel est requise.")]
    [DataType(DataType.EmailAddress)]
    4 references
    public string Email { get; set; } = null!;

    [Required (ErrorMessage = "Le NAS est obligatoire.")]
    [StringLength(9, ErrorMessage = "Un NAS doit contenir 9 chiffres, sans espaces.")]
    [RegularExpression(@"\d+", ErrorMessage = "Un NAS doit contenir 9 chiffres, sans espaces.")]
    4 references
    public string NAS { get; set; } = null!;
}
```



## ❖ Gestion des utilisateurs dans le projet Web

### ◆ Inscription : Action

- En résumé :
  - > On vérifie si le **pseudonyme** est **déjà utilisé**.
  - > On **INSERT** le nouvel utilisateur à l'aide d'une **procédure stockée**. (Montrée dans les diapos précédentes)
  - > Si **exception SQL** quelconque, on revient dans la **vue d'inscription**. Ça ne devrait pas arriver si on a bien fait nos **DataAnnotations** dans le **ViewModel** et que la BD est fonctionnelle.
  - > Sinon, tout est beau et on peut changer de vue Razor.

```
[HttpPost]
0 references
public async Task<IActionResult> Inscription(InscriptionViewModel ivm)
{
    bool existeDeja = await _context.Utilisateurs.AnyAsync(x => x.Pseudonyme == ivm.Pseudonyme);
    if (existeDeja)
    {
        ModelState.AddModelError("Pseudonyme", "Ce pseudonyme est déjà pris.");
        return View(ivm);
    }
    string query = "EXEC Utilisateurs.USB_CreerUtilisateur @Pseudonyme, @MotDePasse, @NAS, @Email";
    List<SqlParameter> parameters = new List<SqlParameter>
    {
        new SqlParameter{ParameterName = "@Pseudonyme", Value = ivm.Pseudonyme},
        new SqlParameter{ParameterName = "@MotDePasse", Value = ivm.MotDePasse},
        new SqlParameter{ParameterName = "@NAS", Value = ivm.NAS},
        new SqlParameter{ParameterName = "@Email", Value = ivm.Email}
    };
    try
    {
        await _context.Database.ExecuteSqlRawAsync(query, parameters.ToArray());
    }
    catch(Exception)
    {
        ModelState.AddModelError("", "Une erreur est survenue. Veuillez réessayer.");
        return View(ivm);
    }
    return RedirectToAction("Connexion", "Utilisateurs");
}
```



## ❖ Gestion des utilisateurs dans le projet Web

- ◆ **Connexion** : **ViewModel** pour envoyer les informations de connexion de la vue au contrôleur
  - Vous pourriez décider d'utiliser le courriel au lieu du pseudonyme.

```
4 references
public class ConnexionViewModel
{
    [Required (ErrorMessage = "Veuillez préciser un nom d'utilisateur.")]
    4 references
    public string Pseudonyme { get; set; } = null!;

    [Required (ErrorMessage = "Veuillez entrer un mot de passe.")]
    4 references
    public string MotDePasse { get; set; } = null!;
}
```



## ❖ Gestion des utilisateurs dans le projet Web

### ◆ Connexion : Action

- En résumé :
- > On utilise une **procédure stockée** qui retourne l'utilisateur **seulement si le mot de passe fourni est valide** ! (Montrée dans la prochaine diapo)
- > Si les identifiants fournis sont **valides**, on crée un **cookie** 🍪 qui identifie l'utilisateur, on lui envoie en le stockant dans son navigateur et on redirige vers une autre page.
- > Si les identifiants sont **invalides**, on reste sur la même page et on affiche un message d'erreur.

```
[HttpPost]
0 references
public async Task<IActionResult> Connexion(ConnexionViewModel cvm)
{
    string query = "EXEC Utilisateurs.USB_AuthUtilisateur @Pseudonyme, @MotDePasse";
    List<SqlParameter> parameters = new List<SqlParameter>
    {
        new SqlParameter{ParameterName = "@Pseudonyme", Value = cvm.Pseudonyme},
        new SqlParameter{ParameterName = "@MotDePasse", Value = cvm.MotDePasse}
    };
    Utilisateur? utilisateur = (await _context.Utilisateurs.FromSqlRaw(query, parameters.ToArray()).ToListAsync()).FirstOrDefault();
    if(utilisateur == null)
    {
        // Premier paramètre vide car erreur pas associée à une propriété spécifique
        ModelState.AddModelError("", "Nom d'utilisateur ou mot de passe invalide");
        return View(cvm);
    }

    List<Claim> claims = new List<Claim>
    {
        new Claim(ClaimTypes.NameIdentifier, utilisateur.UtilisateurId.ToString()),
        new Claim(ClaimTypes.Name, utilisateur.Pseudonyme)
    };

    ClaimsIdentity identite = new ClaimsIdentity(claims, CookieAuthenticationDefaults.AuthenticationScheme);
    ClaimsPrincipal principal = new ClaimsPrincipal(identite);
    await HttpContext.SignInAsync(principal);

    return RedirectToAction("Index", "Utilisateurs");
}
```





## ❖ Gestion des utilisateurs dans le projet Web

- ◆ **Connexion : Procédure stockée** qui retourne l'utilisateur **SI** les identifiants fournis sont **valides**. (Similaire à celle montrée plus tôt)

```
CREATE PROCEDURE Utilisateurs.USB_AuthUtilisateur
    @Pseudonyme nvarchar(50),
    @MotDePasse nvarchar(100)
AS
BEGIN
    DECLARE @Sel varbinary(16);
    DECLARE @MdpHache varbinary(32);
    SELECT @Sel = MdpSel, @MdpHache = MotDePasseHache
    FROM Utilisateurs.Utilisateur
    WHERE Pseudonyme = @Pseudonyme; -- Si les pseudos sont uniques !

    IF HASHBYTES('SHA2_256', CONCAT(@MotDePasse, @Sel)) = @MdpHache
    BEGIN
        -- On retourne l'utilisateur si le mot de passe est valide
        SELECT * FROM Utilisateurs.Utilisateur WHERE Pseudonyme = @Pseudonyme;
    END
    ELSE
    BEGIN
        SELECT TOP 0 * FROM Utilisateurs.Utilisateur; -- On retourne rien si mot de passe invalide
    END
END
GO
```

↖ Ici, on ne peut pas juste **SELECT NULL**. Rappelez-vous qu'avec **.FromSqlRaw**, on doit retourner un objet dont la structure correspond à une des tables ou des vues de la BD !





## ❖ Gestion des utilisateurs dans le projet Web

### ◆ Déconnexion : Action (Presqu'aussi simple qu'avec Angular !)



On coupe dans le sucre

- En résumé :

- > Avec `.SignOutAsync()`, on retire le cookie du navigateur.
- > L'action est de type `Get` car dans ce cas, on a juste fait un « bouton » de déconnexion, sans vue Razor particulière.
- > On peut ensuite rediriger vers l'action de notre choix.

```
<a asp-action="Deconnexion">Deconnexion</a>
```

```
[HttpGet]
```

```
0 references
```

```
public async Task<IActionResult> Deconnexion()  
{  
    await HttpContext.SignOutAsync();  
    return RedirectToAction("Index", "Utilisateurs");  
}
```



## ❖ Gestion des utilisateurs dans le projet Web

◆ Dans n'importe quelle action, grâce au **cookie** 🍪, on peut récupérer l'utilisateur qui vient d'appeler l'action.

- Avec les if, aucune erreur ne sera générée si l'action est utilisée par un **visiteur**, bien entendu.

Différences

```
public async Task<IActionResult> Index()
{
    ViewData["utilisateur"] = "Visiteur";
    IIdentity? identite = HttpContext.User.Identity;    (C'est écrit Identity, mais ça n'a rien à voir avec la librairie Identity)
    if (identite != null && identite.IsAuthenticated)
    {
        string pseudo = HttpContext.User.FindFirstValue(ClaimTypes.Name);
        Utilisateur? utilisateur = await _context.Utilisateurs.FirstOrDefaultAsync(x => x.Pseudonyme == pseudo);
        if(utilisateur != null)
        {
            ViewData["utilisateur"] = utilisateur.Pseudonyme;
        }
    }
    return View();
}
```

Bonjour max

<p>Bonjour @ViewData["utilisateur"]</p>



## ❖ Gestion des utilisateurs dans le projet Web

- ◆ Rappel : Pour qu'une action soit seulement utilisable par un utilisateur **authentifié**, (donc avec un **cookie**) on ajoute **[Authorize]** au-dessus.
  - Alternativement, on peut mettre **[Authorize]** au-dessus d'un contrôleur pour que TOUTES ses actions soient exclusivement utilisables par un utilisateur authentifié.

```
[Authorize]  
0 references  
public async Task<IActionResult> Deconnexion()  
{
```



## ❖ Usage de ModelState quand on crée / update

- ◆ Lorsqu'on crée ou update une entité qui possède une ou des références vers une autre entité, **ModelState.IsValid** peut nous agacer :
  - Au moment où l'**action** reçoit l'entité créée dans le formulaire d'une vue Razor, la référence vers l'entité (Ex : **Employee** Employee, ci-dessous) pourrait être **vide**.

```
Artiste.Employee = artisteEmployee.Employee;
```

- Même si on **comble la référence** (comme ci-dessus) dans l'action avant d'utiliser **ModelState.IsValid**, **ModelState** prend seulement compte de l'état du **Model** tel qu'il était lorsqu'il a été reçu par l'action. (**Autrement dit, les changements qu'on lui fait dans l'action ne sont pas pris en compte**)
- Dans cette situation, ça signifie que **ModelState.IsValid** ne pourra jamais être **true**... dans ce cas, comment on peut vérifier la validité de notre donnée en tenant compte des modifications qu'on lui fait dans l'action ? (Prochaine diapo)

```
[Table("Artiste", Schema = "Employes")]
16 references
public partial class Artiste
{
    [Key]
    [Column("ArtisteID")]
    11 references
    public int ArtisteId { get; set; }
    [StringLength(20)]
    19 references
    public string Specialite { get; set; } = null!;
    [Column("EmployeeID")]
    3 references
    public int EmployeeId { get; set; }

    [ForeignKey("EmployeeId")]
    [InverseProperty("Artistes")]
    13 references
    public virtual Employee Employee { get; set; } = null!;
}
```



## ❖ Usage de ModelState quand on crée / update

◆ On peut effectuer une autre validation mieux adaptée dans l'action.

- La première ligne est une modification de la donnée reçue en paramètre. (**artisteEmploye** est la donnée qui a été reçue par l'action)
- Ensuite, on peut utiliser un validateur qui tiendra compte de l'état actuel de notre donnée.
- Dans le constructeur de **ValidationContext**, on glisse le paramètre qui contient la donnée, comme ça c'est sa classe (et donc ses **[DataAnnotations]**) qui sont utilisées.

```
artisteEmploye.Artiste.Employe = artisteEmploye.Employe;  
bool isValid = Validator.TryValidateObject(artisteEmploye,  
    new ValidationContext(artisteEmploye),  
    new List<ValidationResult>(), true);  
  
if (isValid)  
{  
    try
```

(En gros, ce bout de code remplacerait ModelState.IsValid)