



Documentation technique

présenté par

Jonathan SAENGER

et réalisé dans le cadre du Bachelor

Concepteur Développeur d'Application C#

Session Hiver 2024

Studi
DIGITAL EDUCATION

Table des matières

Table des matières	2
Partie 1. SPECIFICATION TECHNIQUES.....	4
1. Environnement de travail.....	4
2. Outils de conception, de gestion et développement	4
a. Choix de l'IDE : Visual Studio 2022	4
b. Maquettage : FIGMA	5
c. Modèle Conceptuel de Données (MCD) : Looping	5
d. Outil de versionning : Git et Github	6
e. Gestion du projet : Trello	6
3. Technologies pour la partie front-end	7
4. Technologie pour la partie back-end.....	7
Partie 2. CONCEPTION ET ARCHITECTURE DU SYSTEME.....	9
1. Diagramme de cas d'utilisation.....	9
2. Diagramme de séquence du processus de vente	10
3. Modèle Conceptuel de Données (MCD)	10
4. Architecture MVC (Modèles, Vues, Contrôleurs).....	13
Partie 3. MESURES DE SECURITE	15
1. Sécurité des bases de données.....	15
a. Utilisation des chaînes de connexion sans mot de passe.....	15
b. Configuration des règles de pare-feu au niveau du serveur	16
c. Création d'un administrateur Microsoft Entra.....	17
2. Utilisation de l'ORM Entity Framework pour sécuriser ses données	17
a. Protection contre les injections SQL	18
b. Protection contre les attaques XSS (Cross-Site Scripting)	19
3. Sécurité des comptes d'utilisateurs	20
a. Validation restrictive des mots de passe	20
b. Confirmation par Email lors de l'inscription	22
c. Hachage des mots de passe	23
Mise en place de la clé unique associé à chaque utilisateur	23
d. Définition des rôles dans l'application	24
e. Protection des attributions du rôle Admin	27
4. Protection contre les attaques CSRF	27
5. Utilisation du protocole HTTPS lors du déploiement	28
Partie 4. DEPLOIEMENT DE L'APPLICATION	31
1. Configuration de l'environnement de production	31

2. Publier l'application en ligne depuis l'interface de Visual Studio	32
Partie 5. EVOLUTION FUTURE DE L'APPLICATION.....	34
1. L'application directement à la fin des Jeux Olympiques	34
2. Points d'attentions juridiques	34
a. Protection <i>a posteriori</i> des données des utilisateurs.....	34
b. Conservation de l'historique de vente.....	34

Partie 1. SPECIFICATION TECHNIQUES

1. Environnement de travail

L'ensemble de l'application a été réalisé sur un ordinateur utilisant le système d'exploitation Windows dans ses versions 10 et 11.

Pour tout développeur travaillant dans un environnement Linux ou MacOS, il est à fait possible de cloner le projet depuis le dépôt distant sur Github et de travailler dessus. Aucune incompatibilité particulière n'est à mentionner.

Pour ce qui est de l'outil et l'interface de développement, veuillez-vous rendre au premier chapitre à propos de l'environnement de travail.

2. Outils de conception, de gestion et développement

a. Choix de l'IDE : Visual Studio 2022

L'application a été développée avec Visual Studio. Cet IDE complet est parfaitement adapté pour les projets en C# .net. Chaque étapes du développement, de la conception jusqu'au déploiement, en passant par les tests et les collaborations, sont assurées par un ensemble d'outils et de fonctionnalités facilement accessible pour le développeur.

Pour les développeurs sur Linux et MacOS, il est tout à fait possible de télécharger l'IDE. A défaut d'utiliser Visual Studio, il est tout à fait possible de travailler sur l'application après l'avoir cloné avec Visual Studio Code.

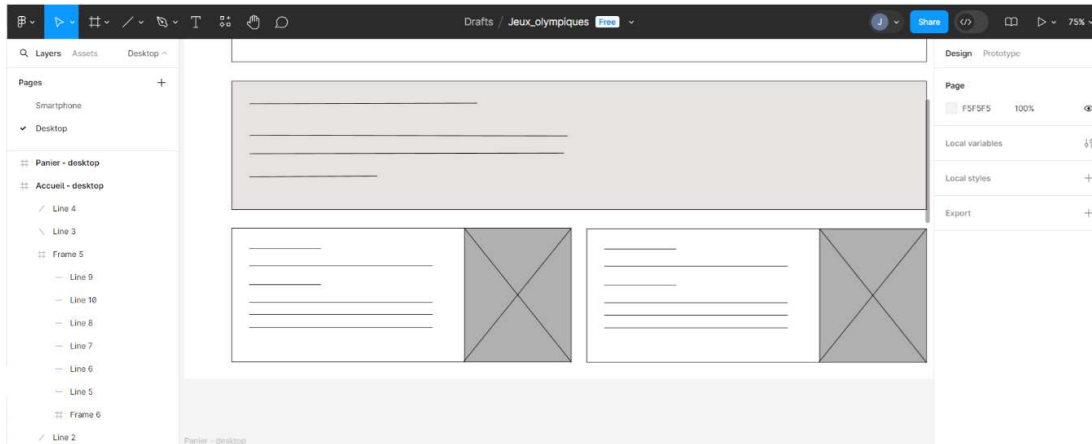
La documentation accompagne la prise en main des outils : <https://visualstudio.microsoft.com/fr/downloads/>



Extrait de la documentation concernant Visual Studio et VS Code

b. Maquettage : FIGMA

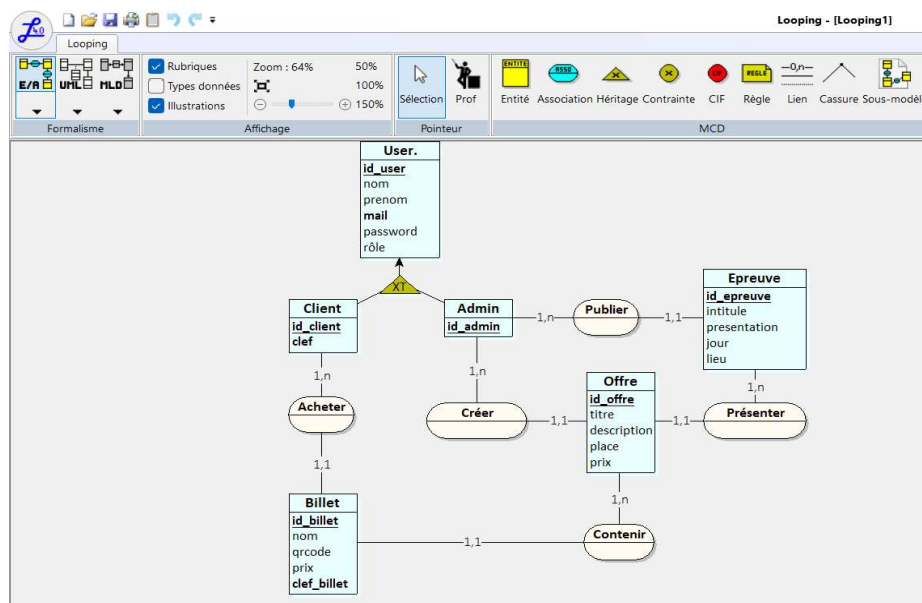
Le logiciel en ligne FIGMA a été utilisé pour la réalisation des Wireframes. Cet outil est facilement accessible depuis n'importe quel poste de travail et se prend facilement en main. Le prototypage interactif qu'il propose permet des visualisations réalistes des projets et une bonne collaboration entre les équipes et le client.



Extrait de l'interface de FIGMA

c. Modèle Conceptuel de Données (MCD) : Looping

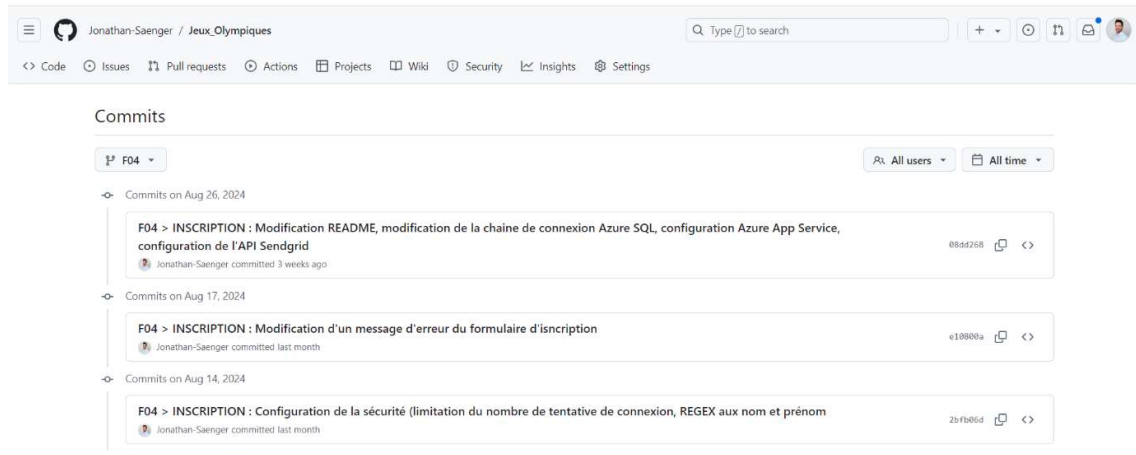
Le MCD a été conçu sur le logiciel Looping. Optimisé pour la méthode MERISE, il est gratuit et facilement accessible. Le logiciel est téléchargeable sur le lien : <https://www.looping-mcd.fr/>



Extrait de l'interface du logiciel Looping

d. Outil de versionning : Git et Github

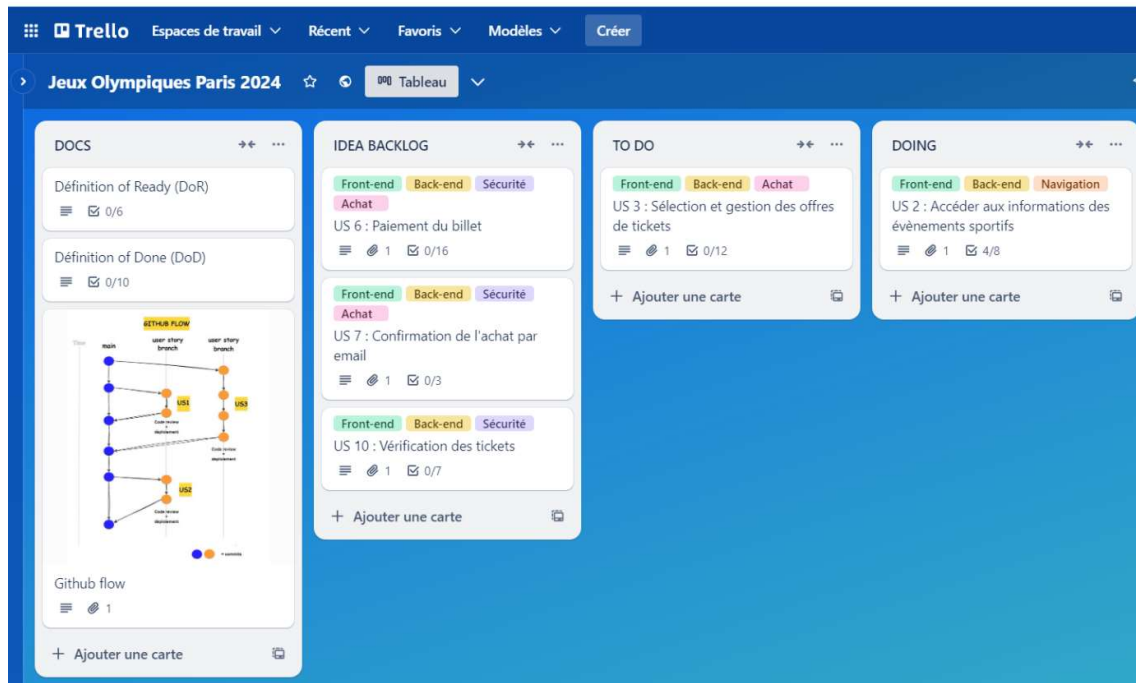
Pour sauvegarder les différentes étapes et versions du projet, Git a été utilisé et le dépôt installé sur Github



Extrait de l'interface Github

e. Gestion du projet : Trello

Trello est un excellent outil de gestion de projet et d'organisation des tâches. Il permet, à travers une interface visuelle intuitive et une flexibilité, de collaborer avec ses équipes en temps réel. Ceci grâce à des tableaux et des cartes personnalisables. Il est aussi idéal pour échanger avec le client qui souhaite suivre en temps réel l'avancement du projet.



Extrait de l'interface Trello

3. Technologies pour la partie front-end

Pour le développement de la partie front-end, les technologies suivantes ont été utilisées :

- **HTML 5** : en portant une attention particulière sur les balises sémantiques pour le référencement. En effet, le site de vente des billets pour les jeux olympiques doit être la référence en sa matière et être appuyé vers le haut des suggestions dans les moteurs de recherches.
- **CSS 5 / Bootstrap (5.3.3)** : il est fondamental dans ce genre d'application de rendre l'expérience utilisateur fluide et intuitive. Le framework Bootstrap permet l'utilisation optimisée de composants prêts à l'emploi et responsive. Ce dernier point est important car il est possible que la grande majorité des utilisateurs naviguent dans l'application via leur smartphone.
- **Javascript / JQuery (3.6.0)** : pour dynamiser les pages et apporter des fonctionnalités qui optimisent davantage l'expérience de l'utilisateur. JQuery permet l'implémentation du filtre de recherche avec l'ajax.
- **Razor page** : utilisé avec ASP.NET CORE, c'est un moteur de vue qui facilite grandement l'affichage des données par le biais de composants et la communication avec le back-end grâce aux injections de dépendances

4. Technologie pour la partie back-end

Pour le développement de la partie back-end, les technologies suivantes ont été utilisées :

- Le langage de programmation **C#** (12) : orienté objet, fortement typé, sécurisé et riche en fonctionnalité, il offre un niveau de performance élevé ce qui en fait un choix important pour ce genre de projet.
- Le Framework **ASP.NET CORE (8.0.8)** : propose tout un écosystème pour le développement d'application. Il a été choisi dans cette application car il propose une structure MVC (Modèle, Vues, Contrôleur) qui permet de bien structurer son code. Ce Framework met à disposition un nombre important de bibliothèques qui répondent à la quasi-totalité des besoins que nous avons dans le cadre de ce projet suite aux exigences du client.
- L'ORM (Object-Relational Mapping) **Entity Framework (8.0.8)** : facilite et accélère les interactions entre l'application et la base de données. Il permet une manipulation fluide des objets tout en assurant l'intégrité et la sécurité des données.
- Base de données **Azure SQL Database** : hautement disponible, les sauvegardes sont assurées automatiquement, elle propose des services de sécurité avancés

(chiffrement des données et gestion des accès sur lesquels nous reviendrons dans cette documentation).

L'ensemble des choix techniques dans le cadre de la réalisation de cette application sont détaillés au fur et à mesure de cette présente documentation.

Partie 2. CONCEPTION ET ARCHITECTURE DU SYSTEME

1. Diagramme de cas d'utilisation

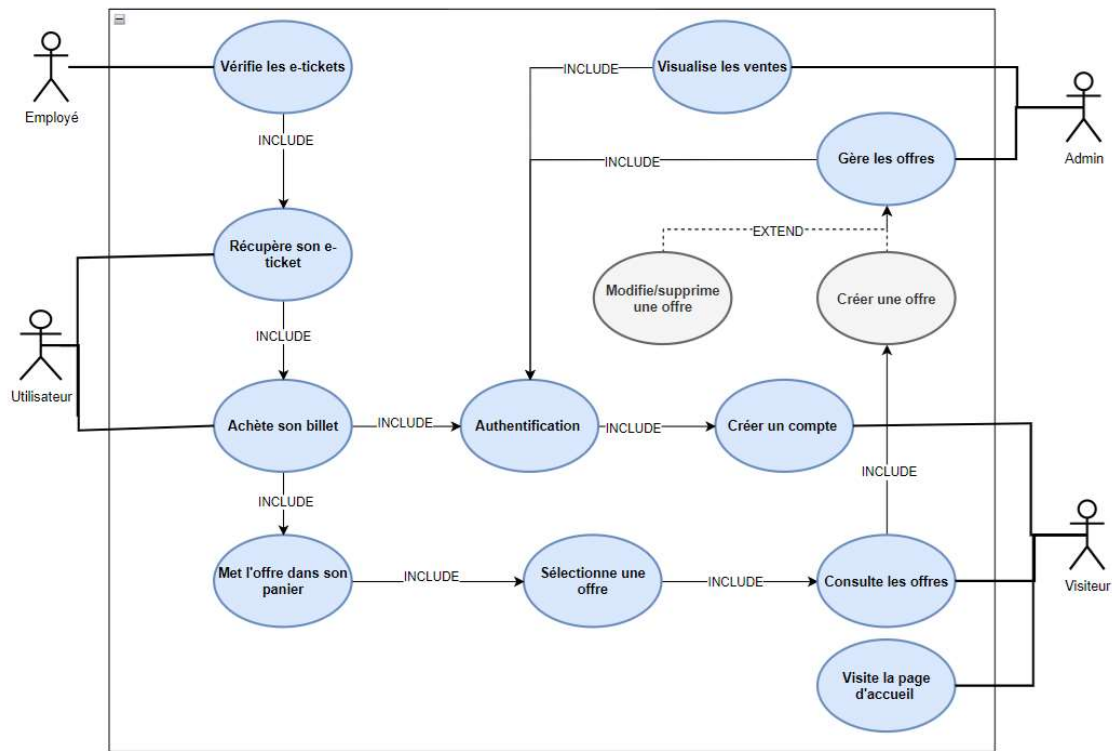


Diagramme de cas d'utilisation

2. Diagramme de séquence du processus de vente

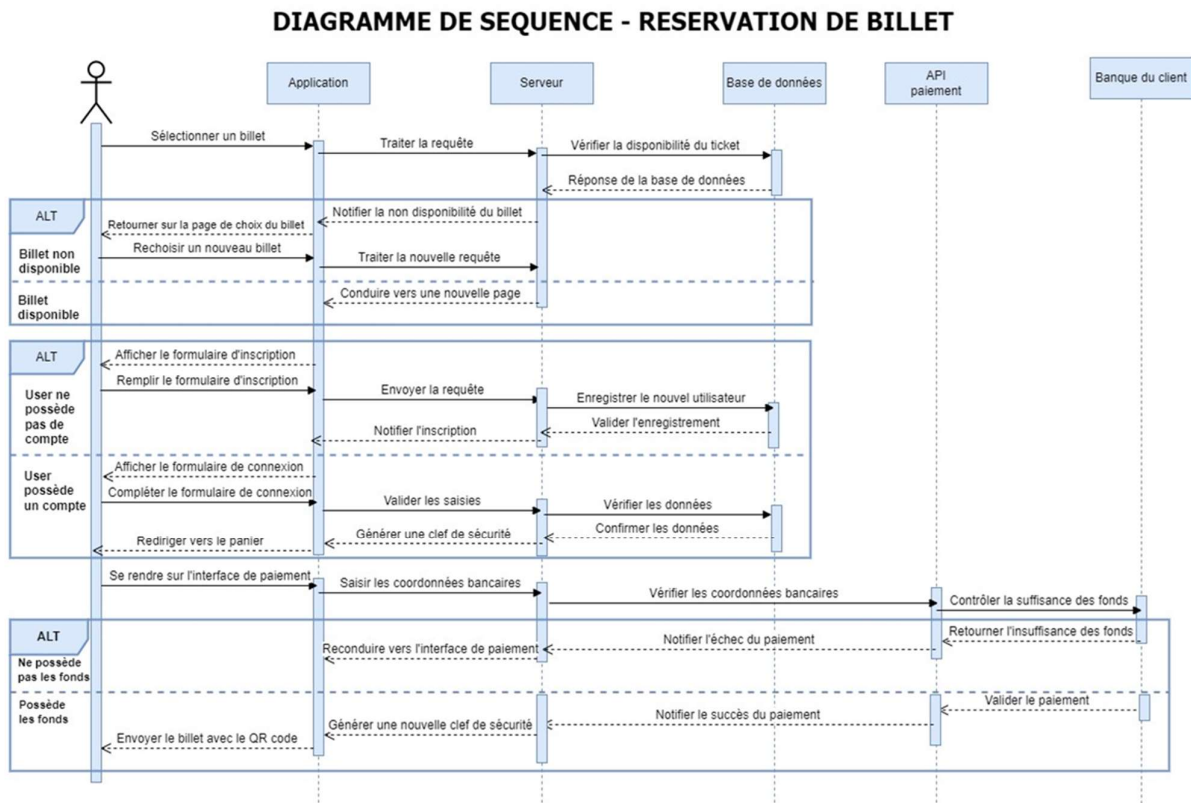


Diagramme de séquence détaillant le processus de vente d'un billet

3. Modèle Conceptuel de Données (MCD)

De nombreux utilisateurs/clients vont utiliser l'application. L'entité représentant le User fera l'objet d'une attention particulière. Nous utilisons ainsi l'API Identity dont le schéma est fourni par .NET, que nous reprenons en le complétant avec les besoins de l'application. Nous obtenons le Modèle Conceptuel de Données suivant :

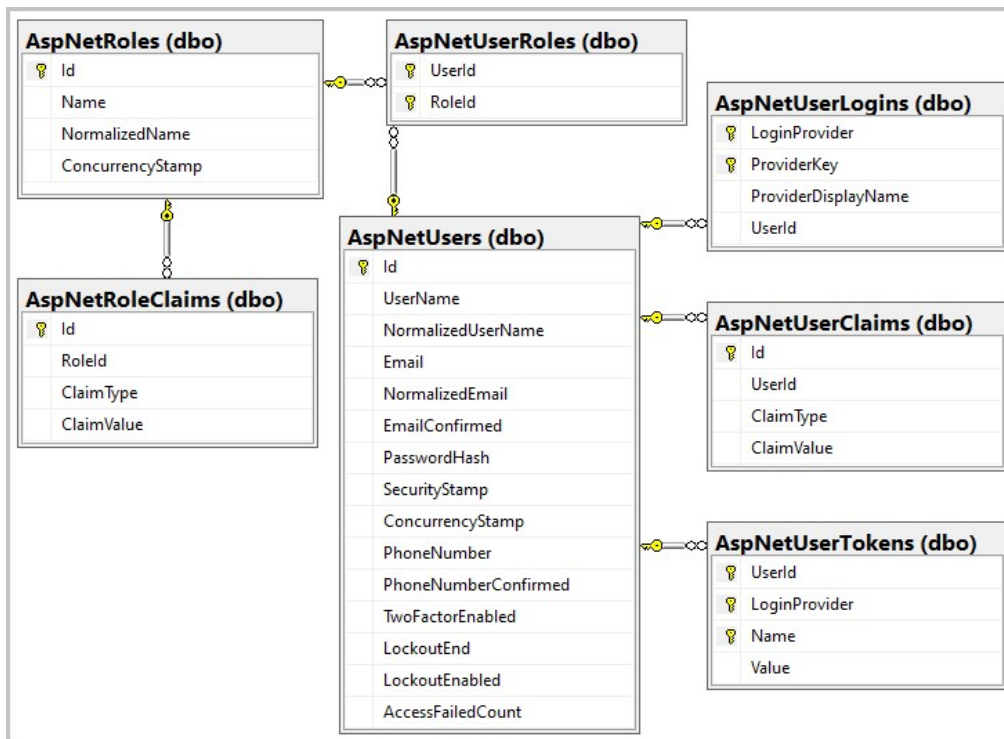
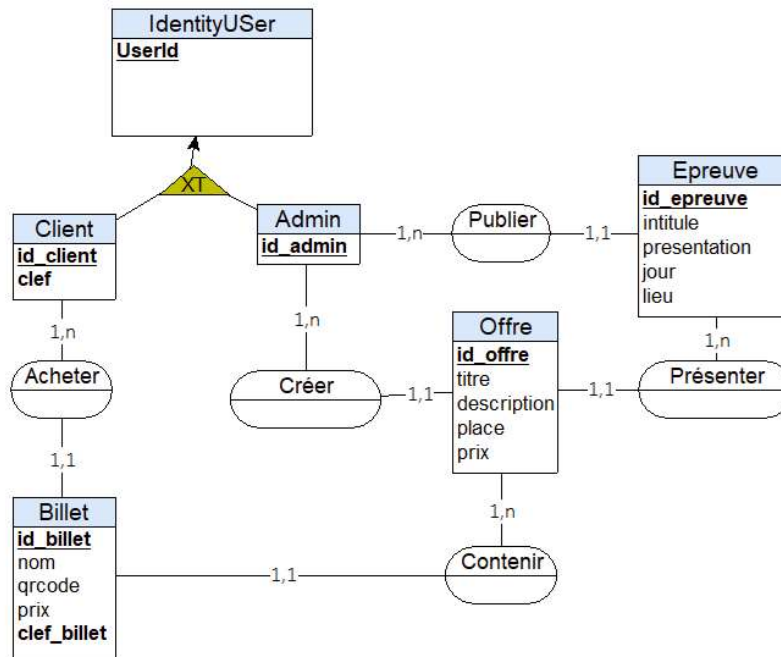


Diagramme fournit par Microsoft de la fonctionnalité Identity

Explication du diagramme :

AspNetUser représente l'utilisateur. Cette entité est centrale dans notre application. **AspNetRole** représente un rôle. Nous allons pouvoir y intégrer le rôle *Admin* qui bénéficiera de certains privilège dans la gestion de l'application. **AspNetUserClaims** représente une revendication d'un utilisateur possède. **AspNetUserToken** représente un jeton d'utilisation pour un utilisateur. Cette entité va nous servir à gérer les sessions des utilisateurs de l'application, plus précisément l'*Admin* et les utilisateurs qui souhaitent acheter leur billet. **AspNetUserLogin** associe un utilisateur à une connexion. **AspNetRoleClaim** représente une revendication octroyée à tous les utilisateurs au sein d'un rôle. **AspNetUserRole** est une entité de jointure qui associe des utilisateurs et des rôles.

Concrètement, chaque **User** peut avoir plusieurs **UserClaims**, **UserLogins**, **UserTokens** et **RoleClaims**. **User** et **Rôle** sont liés par une relation *Many To Many* qui créé l'entité d'association **UserRole**.

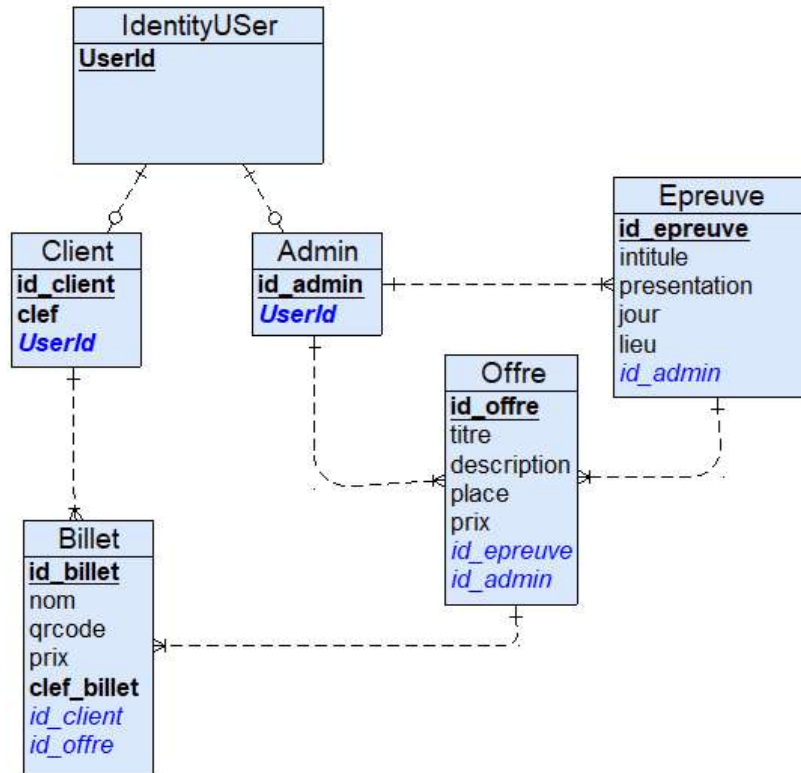


Modèle Conceptuel de Données schématisant les entités et leurs relations

Explications des relation et cardinalités à partir des deux diagrammes :

- Les entités *Client* et *Admin* sont héritèrent de l'*Identity User* .
- Chaque *User* peut avoir plusieurs *UserClaims*, plusieurs, *UserLogins*, *UserTokens*.
- Chaque *Role* peut avoir plusieurs *RoleClaims* associées.
- Chaque *User* peut avoir plusieurs *Roles* associés , et chaque *Role* peut être associé à plusieurs *Users*. Il s'agit d'une relation plusieurs-à-plusieurs qui nécessite une table de jointure dans la base de données. La table de jointure est représentée par l'entité *UserRole*.
- L'*Admin* publie une à plusieurs *Epreuve(s)*. Chaque *Epreuve* n'est publié que par un seul *Admin*.
- L'*Admin* créé une ou plusieurs *Offre(s)*. Chaque *Offre* n'est créée que par un seul *Admin*.
- Chaque *Epreuve* présente une ou plusieurs *Offre(s)*. Chaque *Offre* est présenté dans une seule *Epreuve*.
- Chaque *Offre* est contenu dans un ou plusieurs *billet(s)*. Chaque *Billet* ne contient qu'une seule offre.

Du MCD, nous en déduisons le Modèle Logique de Données (MLD) avec l'ajout, dans chaque entité, des clés étrangères :



Modèle Logique de données (MLD)

Clés primaires et clés étrangères :

- *Userid*, est une clé étrangère dans les tables *Client* et *Admin* qui fait référence à la clé primaire *Userid* dans *IdentityUser*. Rappelons que ce dernier est déjà inclus dans ASP.NET CORE et qu'il nous incombe de différencier le rôle Admin.
- *id_admin* est une clé étrangère dans les tables *Offres* et *Epreuve* qui fait référence à la clé primaire *id_admin* dans la table *Admin*
- *id_offre* est une clé étrangère dans la table *Epreuve* qui fait référence à la clé primaire *id_offre* dans la table *Offre*.
- *id_client* et *id_offre* sont des clés étrangères dans la table *Billet* qui font références à la clé primaire *id_client* et *id_offre* dans les tables *Client* et *Offre*.

4. Architecture MVC (Modèles, Vues, Contrôleurs)

L'architecture MVC permet de séparer les différentes logiques de l'application : logique métier, les données et le design. Une application conçue selon ce principe présente une grande modularité qui est nécessaire dans le contexte d'un projet comme celui des jeux olympiques. Ainsi, l'application est mieux organisée, plus facile à gérer et déboguer.

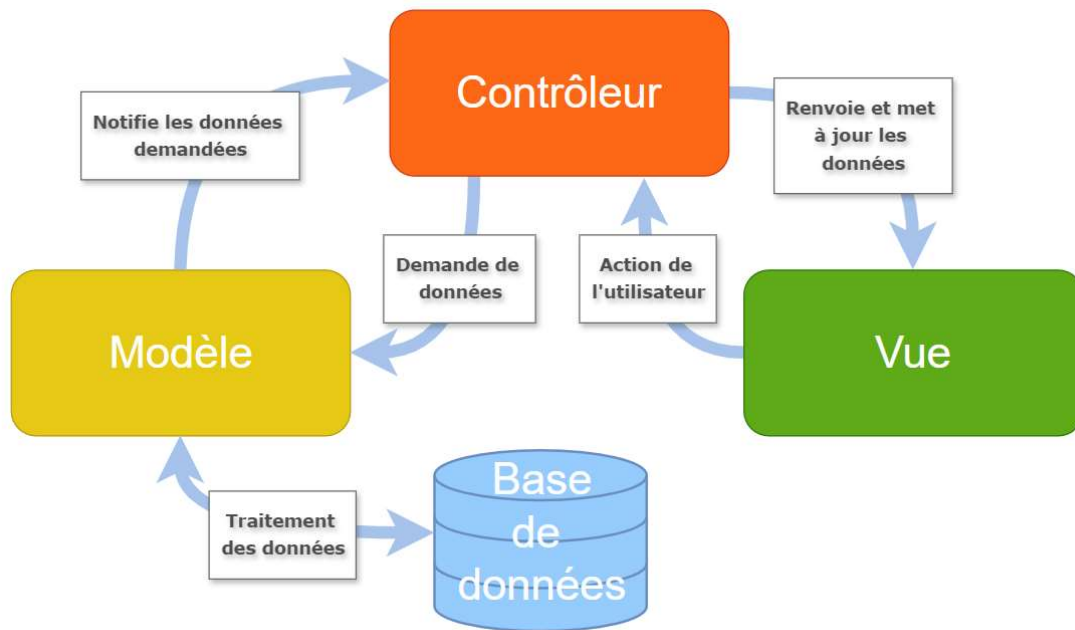


Schéma explicatif de l'architecture MVC

La vue gère l'affichage des informations et les interactions utilisateur dans l'application. La particularité avec le Framework ASP.NET Core et l'incorporation du balisage Razor qui est du code C# interagissant avec du HTML.

Le contrôleur est utilisé pour définir et regrouper un ensemble d'actions, c'est-à-dire des méthodes qui gèrent les demandes. Il fait le lien entre le modèle et la vue en renvoyant les réponses du modèle à la vue.

Le modèle permet de gérer et structurer les données en lien avec le contrôleur. La logique métier, l'ensemble de la gestion et la sauvegarde de l'état de l'application, est centralisée dans le modèle.

Partie 3. MESURES DE SECURITE

1. Sécurité des bases de données

Nous utilisons SQL Server pour la base de données en local et Azure SQL Database pour la base de données en ligne connectée à l'application. La chaîne de connexion de la base de données locale est configurée dans le fichier *appsettings.Development.json* et celle de la base de données en ligne est configurée dans le fichier *appsettings.json*. L'utilisation de deux bases de données distinctes est une bonne pratique recommandée dans la documentation officielle de Microsoft.

Les deux bases de données obéissent aux mêmes règles de sécurité.

a. Utilisation des chaînes de connexion sans mot de passe

Une fois la base de données créée, nous avons installé l'ORM Entity Framework. Nous reviendrons sur l'intérêt d'utiliser un ORM en matière de sécurité dans la section suivante.

La documentation explique consciencieusement comment configurer de manière sécurisée la base de données (<https://learn.microsoft.com/fr-fr/azure/azure-sql/database/azure-sql-dotnet-entity-framework-core-quickstart?view=azuresql&tabs=dotnet-cli%2Cservice-connector%2Cportal>).

Pour commencer, nous avons configuré la connexion à notre base de données, Azure SQL Database, à l'aide d'Entity Framework dans le fichier *appsetting.Development.json* :

```
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft.AspNetCore": "Warning"
6      }
7    },
8    "ConnectionStrings": {
9      "AZURE_SQL_CONNECTIONSTRING": "Data Source=tcp:jeux-olympiquesdbserver.database.windows.net,1433;Initial Catalog=Jeux_Olympiques_db;Authentication=Active Directory Default;Encrypt=True;"
10   }
```

Extrait du fichier *appsetting.Development.json* avec la chaîne de connexion

Comme nous pouvons le constater, nous nous sommes connecté, sans mettre en évidence l'identifiant et le mot de passe. Il s'agit d'une chaîne de connexion sans mot de passe qui est disponible dans l'onglet « Chaînes de connexion » au sein des paramètres de la base de données dans le portail Azure.

Il est très important, même dans le contexte d'une configuration de base de données locale, de ne pas mettre en évidence nos identifiants pour des raisons de sécurité. Pour assurer cela, Entity Framework s'appuie sur les bibliothèques *Microsoft.Data.SqlClient* et *Azure.Identity*.

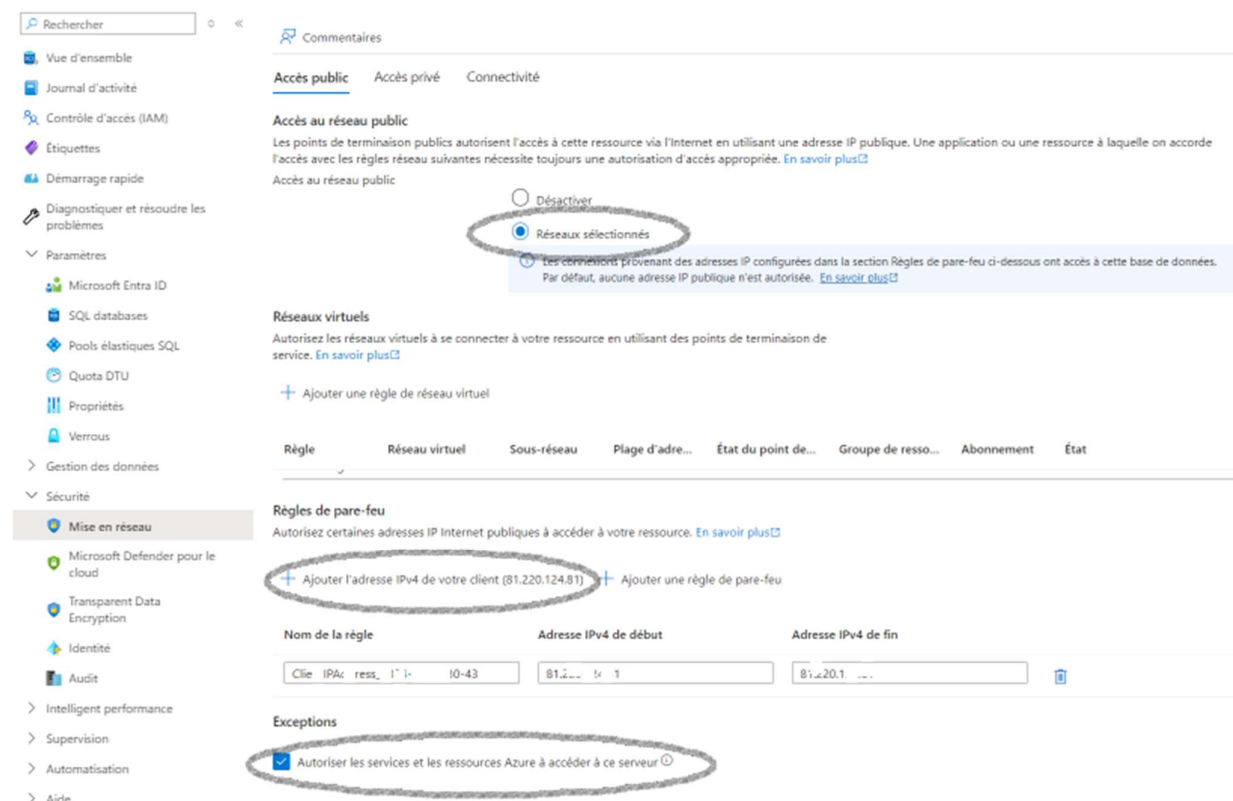
Pour renforcer la sécurité, nous nous assurons que le fichier *appsettings.Development.json* est ignoré dans notre contrôle de version en l'ajoutant à *.gitignore* pour éviter de pousser ces informations sensibles sur notre dépôt.

Pour la base de données en production, deux options s'offrent à nous :

- L'utilisation d'Azure Key Vault pour stocker la chaîne de connexion de manière sécurisée, plutôt que de la laisser en clair dans *appsettings.json*
- Lors du déploiement sur Azure, nous pouvons configurer la chaîne de connexion directement dans les paramètres de l'application sur notre interface Services Azure pour remplacer ce qui est dans *appsettings.json*.

b. Configuration des règles de pare-feu au niveau du serveur

Cette manipulation s'effectue directement dans le portail Azure. Dans le menu de gauche, nous sélectionnons « Base de données SQL », nous nous rendons dans l'onglet « Sécurité », puis « Mise en Réseau ». Nous avons coché, dans « Accès public » l'option « Réseaux sélectionnés ». Dans les « Règles de pare-feu », nous avons autorisé notre adresse IP Internet à accéder à notre base de données.



Création des règles de pare-feu dans le portail Azure

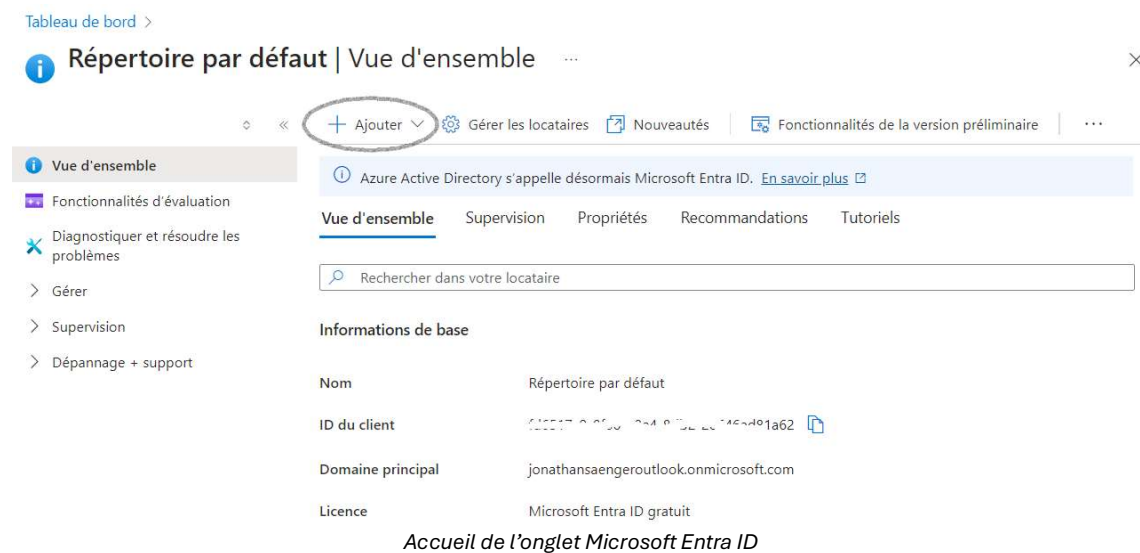
Grâce à ce modèle, l'accès à la base de données est très limitée car est autorisée pour seulement une adresse IP. Autrement dit, un utilisateur essaye de se connecter à partir d'un autre réseau, il sera bloqué par le pare-feu.

c. Création d'un administrateur Microsoft Entra

Cette fonction permet d'activer une authentification spéciale pour accéder au serveur. Autrement dit, l'accès sera uniquement réservés à des identités précises définies par l'administrateur.

Nous configurons, sur ce serveur, un administrateur qui n'est autre que le compte que nous utilisons pour nous connecter localement aux services Microsoft (Visual Studio dans notre cas).

Dans notre portail Microsoft Azure, nous nous rendons dans l'onglet « Microsoft Entra ID » avant de sélectionner « Définir un administrateur » et de se définir comme administrateur en cochant la case devant son adresse email.



2. Utilisation de l'ORM Entity Framework pour sécuriser ses données

L'ORM Entity Framework s'installe facilement et rapidement par le biais de la ligne de commande :

```
dotnet add package Microsoft.EntityFrameworkCore  
dotnet add package Microsoft.EntityFrameworkCore.SqlServer  
dotnet add package Microsoft.EntityFrameworkCore.Design
```

Extrait de la ligne de commande pour installer Entity Framework

En installant Entity Framework Core avec le provider SQL Server, nous avons les outils nécessaires pour travailler avec SQL Server en local. C'est parfait pour notre environnement de développement.

En parallèle, Entity Framework dispose d'un éventail de fonctionnalités qui permettent de renforcer la sécurité de l'application. Parmi celles-ci, nous pouvons citer :

- Protection contre les injections SQL grâce à l'utilisation des requêtes paramétrées
- Protection contre les attaques XSS grâce à un mécanisme de validation, auquel nous avons ajoutés d'autres couches, et d'encodage des entrées.

a. Protection contre les injections SQL

Une injection SQL est une attaque dans laquelle un utilisateur malveillant injecte une requête SQL dans un champ de saisie utilisateur d'une application dans le but, par exemple, d'usurper l'identité d'un/plusieurs utilisateurs de l'application, d'entrée par effraction dans une base de données pour y saisir des données confidentielle ou encore d'endommager ou supprimer la base de données.

Une des premières mesure contre ce genre d'attaque est, comme configuré automatiquement grâce à Entity Framework (EF), de valider toutes les entrées des utilisateurs en testant leur type, leur longueur ou encore leur format.

EF génère automatiquement des requêtes paramétrées pour prévenir les injections SQL.

Les caractères spéciaux, souvent utilisés dans les requêtes SQL sont considérés comme potentiellement dangereux et sont donc automatiquement échappés.

La documentation officielle, dans sa partie sur EF, encourage l'utilisation d'opérateurs LINQ. EF traite le code SQL comme une sous-requête et compose dessus dans la base de données. Par exemple, il est possible de commencer une requête LINQ en utilisant la méthode *FromSql* qui est protégée contre l'injection de code SQL et intègre toujours les données de paramètre sous forme de paramètre SQL distinct.

b. Protection contre les attaques XSS (Cross-Site Scripting)

La documentation nous fournit une définition très précise de ce genre d'attaque : « *Cross-Site Scripting (XSS) est une vulnérabilité de sécurité qui permet à un attaquant de placer des scripts côté client (généralement JavaScript) dans des pages Web. Lorsque d'autres utilisateurs chargent des pages affectées, les scripts de l'attaquant s'exécutent, permettant à l'attaquant de voler des cookies et des jetons de session, de modifier le contenu de la page Web par manipulation DOM ou de rediriger le navigateur vers une autre page. Les vulnérabilités XSS se produisent généralement lorsqu'une application prend une entrée utilisateur et la sort sur une page sans la valider, l'encoder ou l'échapper.* ».

On s'en rend compte à la lecture de la définition, les dégâts sur l'application et l'expérience de l'utilisateur peuvent être dramatiques. Pour empêcher les attaques de ce genre, les champs de saisies de l'utilisateur doivent implémenter une validation stricte.

Des règles de validations sont déjà implémentées par Entity Framework et l'Identity, notamment avec l'utilisation des annotations de données comme [Required] ou encore [StringLength] qui permettent de limiter les saisies malveillantes.

Pour renforcer la sécurité face à ce genre d'attaque, nous avons ajoutés **des expressions régulières (REGEX)** qui limitent l'utilisation de certains caractères, notamment ceux utilisés, le plus souvent, pour écrire du script.

```
public class InputModel
{
    //Ajouts personnalisés
    [Required]
    [DataType(DataType.Text)]
    [RegularExpression(@"^[A-ZÀ-ÿ]+[a-zA-ZÀ-ÿ\s]*$", ErrorMessage = "Veuillez utiliser uniquement des lettres")]
    [StringLength(100, ErrorMessage = "Le {0} doit comporter au moins {2} caractères et au maximum {1} caractères.", MinimumLength = 1)]
    [Display(Name = "Prénom")]
    4 références
    public string FirstName { get; set; }

    [Required]
    [DataType(DataType.Text)]
    [RegularExpression(@"^[A-ZÀ-ÿ]+[a-zA-ZÀ-ÿ\s]*$", ErrorMessage = "Veuillez utiliser uniquement des lettres")]
    [StringLength(100, ErrorMessage = "Le {0} doit comporter au moins {2} caractères et au maximum {1} caractères.", MinimumLength = 1)]
    [Display(Name = "Nom")]
    4 références
    public string LastName { get; set; }

    /// <summary>
    /// This API supports the ASP.NET Core Identity default UI infrastructure and is not intended to be used
    /// directly from your code. This API may change or be removed in future releases.
    /// </summary>
    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    7 références
    public string Email { get; set; }
```

Extrait du fichier « register.cshtml.cs » avec l'écriture de REGEX pour imposer uniquement l'utilisation des lettres dans les champs de saisies relatifs au nom et prénom

Deuxièmement, l'encodage de sortie doit garantir que toutes les données renvoyées au serveur soient correctement filtrées afin qu'elles ne puissent pas être exécutées en tant que code par le navigateur de l'utilisateur. C'est le cas avec l'**encodage HTML grâce à Razor**.

L'architecture .NET CORE MVC avec Identity et Entity Framework inclut l'utilisation du moteur Razor dans les Vues. Ce dernier encode automatiquement les caractères

spéciaux en entités HTML grâce à sa syntaxe « @ { } ». Par exemple, les caractères < et > (considérés comme des caractères dangereux) sont convertis en < et >, empêchant ainsi l'injection de scripts malveillants. Par ailleurs, Razor fournit des méthodes qui permettent un encodage personnalisé en fonction des besoins.

```

1  @page
2  @model LoginModel
3
4  @{
5      ViewData["Title"] = "Log in";
6  }
7
8  <h1>@ViewData["Title"]</h1>
9  <div class="row">
10     <div class="col-md-4">
11         <section>
12             <form id="account" method="post">
13                 <h2>Use a local account to log in.</h2>
14                 <hr />
15                 <div asp-validation-summary="ModelOnly" class="text-danger" role="alert"></div>
16                 <div class="form-floating mb-3">
17                     <input asp-for="Input.Email" class="form-control" autocomplete="username" aria-required="true" placeholder="name@example.com" />
18                     <label asp-for="Input.Email" class="form-label">Email</label>
19                     <span asp-validation-for="Input.Email" class="text-danger"></span>
20                 </div>
21                 <div class="form-floating mb-3">
22                     <input asp-for="Input.Password" class="form-control" autocomplete="current-password" aria-required="true" placeholder="password" />
23                     <label asp-for="Input.Password" class="form-label">Password</label>
24                     <span asp-validation-for="Input.Password" class="text-danger"></span>
25                 </div>
26                 <div class="checkbox mb-3">
27                     <label asp-for="Input.RememberMe" class="form-label">
28                         <input class="form-check-input" asp-for="Input.RememberMe" />
29                         @Html.DisplayNameFor(m => m.Input.RememberMe)
30                     </label>

```

Extrait de la page login.cs utilisant le moteur Razor

Plusieurs commentaires sur le code ci-dessus :

- Nous pouvons lire l'encodage automatique avec l'utilisation de `@ViewData["Title"]`, ou encore `@Html.DisplayNameFor(m => m.Input.RememberMe)`, Ces valeurs seront automatiquement encodées pour prévenir les attaques XSS.
- Nous pouvons également voir l'utilisation de Tag Helpers comme `asp-for`, `asp-validation-for` par exemple qui garantissent que les valeurs sont correctement encodées dans le contexte approprié (attributs HTML, URL, etc.).
- Nous voyons aussi la présence de `<div asp-validation-summary="ModelOnly">` qui indique l'utilisation de la validation du modèle tel que nous l'avons définis précédemment. Cela se traduit, en cas de mauvaise saisie, par un affichage instantané de l'erreur de saisie à corriger.

3. Sécurité des comptes d'utilisateurs

a. Validation restrictive des mots de passe

La première mesure de sécurité pour protéger les mots de passe des utilisateurs est de suggérer que ces derniers soit fort. Un minimum de caractère, une lettre majuscule, un caractère spécial et au moins un chiffre sont exigés. Cette exigence est imposé d'emblée par *Identity* et elle répond parfaitement à notre besoin. Nous la conservons. On s'assure

alors que l'utilisateur s'équipe d'un mot de passe suffisamment robuste face à une éventuelle **attaque par force brute**. Par le biais de cette attaque, l'utilisateur mal intentionné va répéter des tentatives en essayant plusieurs mots de passe à la suite.

En parallèle d'un mot de passe difficile à deviner, nous allons plus loin en incrémentant dans notre code la méthode *PasswordSignInAsync*. Cette méthode va nous permettre de verrouiller la page de connexion durant un certain délai après 3 tentatives de connexion infructueuse. Cette fonctionnalité démotive l'utilisateur malveillant à retenter l'opération ensuite. La méthode *PasswordSignInAsync* est configurée dans le fichier *Login.cs* :

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");

    ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync()).ToList();

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.Email, Input.Password, Input.RememberMe, lockoutOnFailure: true);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new { ReturnUrl = returnUrl, RememberMe = Input.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return Page();
        }
    }
}
```

Extrait du fichier *Login.cs*

Pour l'activer, nous avons ensuite à ajouter les options de verrouillage dans notre fichier *Program.cs* :

```
30 builder.Services.Configure<IdentityOptions>(options =>
31 {
32     // Default Lockout settings.
33     options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
34     options.Lockout.MaxFailedAccessAttempts = 5;
35     options.Lockout.AllowedForNewUsers = true;
36 });
37
```

Extrait du fichier *Program.cs* avec la configuration du blocage de l'authentification

Dans cet extrait de code, nous avons définis la classe *IdentityOptions* et configuré *LockoutOptions* de manière à ce que l'authentification soit bloquée après 5 tentatives infructueuses.

Identification bloquée

Suite à un nombre trop important de tentatives de connexions, votre compte est momentanément bloqué.

Veuillez tenter de vous reconnecter ultérieurement ou contacter l'administrateur.

Message affiché lorsque le compte est bloqué.

b. Confirmation par Email lors de l'inscription

Soumettre la création d'un compte utilisateur à la validation via un lien envoyé par Email permet de valider l'identité de la personne qui s'inscrit. Dans le Framework ASP.NET CORE MVC, avec l'implémentation d'*Identity*, une fonctionnalité de validation d'email est déjà implémentée avec *IMailerSender*. Cette API permet, via *Identity*, d'envoyer des emails de confirmation et de réinitialisation de mot de passe. Nous utilisons les deux fonctionnalités.

Dans le cadre de notre application, nous l'avons donc configurée pour la rendre utilisable. A ce titre, nous utilisons Sendgrid qui est une plateforme d'email offrant un service d'envoi d'emails de manière fiable et évolutive.

Une règle de sécurité importante est à prendre en considération : ne jamais laisser la clé API apparaitre ou que ce soit dans l'application. Pour cela, Microsoft met à disposition *Secret Manager* qui permet un stockage sécurisé des secrets d'application en phase de développement. Nous le configurons simplement avec la ligne de commande :

CLI .NET

```
dotnet user-secrets set SendGridKey <key>
```

```
Successfully saved SendGridKey to the secret store.
```

Extrait de la ligne de commande permettant de configurer secrètement la clé API de Sendgrid

Pour la phase de production, nous utilisons Azure Key Vault que nous configurons directement depuis notre espace de gestion App Services Microsoft Azure.

Premièrement, cette sécurité permet de s'assurer que l'adresse email fournie appartient bien à l'utilisateur qui s'inscrit. On évite ainsi l'usurpation d'identité. Ce point est crucial car le paiement des tickets est soumis à l'inscription. Sans cette vérification, quelqu'un pourrait utiliser une adresse email qui ne lui appartient pas et procéder à des achats à l'insu d'une autre personne.

Ensuite, cette mesure permet de se prémunir contre les inscriptions massives d'un utilisateur malveillant ou un bot autrement dit, ça permet d'éviter les SPAMS.

Enfin, la vérification du mail permet de s'assurer que l'utilisateur est réellement intéressé par l'inscription et qu'il a accès à l'email fourni. Cela évite les inscriptions accidentelles ou non désirées.

Conformément à la demande du client, le compte de l'administrateur est créé en amont. Ce dernier n'a donc pas besoin de s'inscrire sur l'application.

Pour les utilisateurs, la fonctionnalité de changement de mot de passe en cas d'oubli a été implémentée.

c. Hachage des mots de passe

ASP.NET Core MVC avec Identity inclut d'office la fonction de hachage des mots de passe. Ce point est important car elle permet de ne laisser aucune visibilité sur le mot de passe personnel des utilisateurs.

PasswordHash	FirstName	LastName
AQAAAAIAAYagAAAAELHYafDOzDRX2Ly4MdG48CsSejTgiUEY8DFLhcHXnK449mGSMXGIlz11PO18CGZ0vw==	Jonathan	Saenger
NULL	NULL	NULL

Extraite de la base de données SQL Server avec un aperçu du hachage du mot de passe

Mise en place de la clé unique associé à chaque utilisateur

Demandé par le client, il s'agit d'un point important car il constitue un facteur de confirmation l'identité de l'utilisateur unique et un élément de vérification de l'authenticité d'un ticket.

Pour configurer cette fonctionnalité, il a fallu tout d'abord créer des nouveaux attributs pour l'utilisateur (en créant une classe dérivée qui hérite de IdentityUser) de notre application, puis écrire un code qui permet de générer cette clé :

```

public class Jeux_OlympiquesUser : IdentityUser
{
    [PersonalData]
    7 références | 1/1 ayant réussi
    public string? FirstName { get; set; }

    [PersonalData]
    7 références | 1/1 ayant réussi
    public string? LastName { get; set; }

    [PersonalData]
    7 références | 1/1 ayant réussi
    public string? AccountKey { get; private set; }

    // Méthode pour générer la clé unique
    3 références | 1/1 ayant réussi
    public void GenerateAccountKey()
    {
        using (var sha256 = SHA256.Create())
        {
            var combinedValue = $"{Id}{FirstName}{LastName}{Email}";
            var hashBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(combinedValue));
            AccountKey = BitConverter.ToString(hashBytes).Replace("-", "").ToLower();
        }
    }
}

```

Extrait de la classe Jeux_OlympiqueUser avec le code qui génère la clé unique pour chaque utilisateur

Nous avons défini une méthode, *GénérerAccountKey* qui crée une clé unique pour chaque utilisateur et le stock dans la base de données afin qu'il soit utilisable dans l'application. Pour cela, nous avons créé une instance de l'algorithme de hachage SHA256. Fourni par le framework .net, cet algorithme a l'avantage d'être quasiment impossible à lire. Nous avons ensuite créé une variable sous la forme d'une interpolation de chaîne de caractères qui reprend l'id, le nom, le prénom et l'email de l'utilisateur.

L'intérêt de prendre autant d'éléments est d'assurer l'unicité de chaque clé. Nous avons ensuite créé une autre variable qui va hacher la chaîne avec l'algorithme SHA256. Enfin, nous récupérons le résultat pour le convertir en chaîne hexadécimale en minuscule dans laquelle nous supprimons les tirets. Voici le résultat obtenu :

AccountKey	FirstName	LastName
3485ada96a05c3e3f2c0448c43876e3f86be2faf53c048db1cc4f687de59	Jonathan	Saenger
NULL	NULL	NULL

Extrait de la base de données SQL Server avec la clé unique d'un utilisateur

d. Définition des rôles dans l'application

Définir les rôles au sein de l'application consiste à définir des autorisations en fonction des rôles. Cela va permettre de sécuriser les espaces réservés notamment. A ce titre, nous distinguons deux rôles au sein de notre application avec chacun ses autorisations :

- **Rôle User** : il s'agit du rôle standard de chaque visiteur devenu utilisateur après l'inscription. Ce rôle lui permettra d'effectuer un achat de ticket directement depuis l'application. Chaque inscrit hérite automatiquement de ce rôle.
- **Rôle Admin** : ce rôle confèrera à son détenteur le privilège d'accéder à un espace de gestion depuis lequel il pourra gérer les annonces et les utilisateurs. Il est impossible d'hériter de ce rôle depuis l'application, c'est pourquoi nous allons le créer et l'attribuer manuellement.

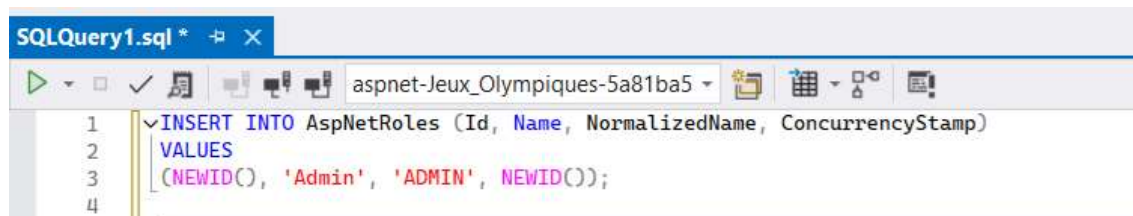
Puisque nous définissons des autorisations selon les rôles, nous ajoutons des services de rôle à Identity dans notre fichier *Program.cs* :

```
builder.Services.AddDefaultIdentity<Jeux_OlympiquesUser>(options => options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddRoles<IdentityRole>();
builder.Services.AddControllersWithViews();
```

Extrait du fichier Program.cs avec l'ajout du service Rôle à Identity

Ensuite, nous créons plusieurs requêtes SQL dans notre base de données :

- dans notre table Rôle (*AspNetRoles*), nous insérons le rôle *Admin*.



Requête SQL d'ajout du rôle Admin dans la table AspNetRoles

The screenshot shows the 'dbo.AspNetRoles' table in the 'aspnet-Jeux_Olympiques-5a8...' database. The table has four columns: Id, Name, NormalizedName, and ConcurrencyStamp. There are two rows: the first row contains the 'Admin' role with a GUID for Id and a GUID for ConcurrencyStamp; the second row is a placeholder with NULL values.

	Id	Name	NormalizedName	ConcurrencyStamp
1	59C78292-150C-4218-B08...	Admin	ADMIN	F92315DD-48DD-4EA1-A4C3-C532D81...
2	NULL	NULL	NULL	NULL

Extrait de la table AspNetRoles de la base de données avec le succès de la création du rôle Admin

- dans notre table User (*AspNetUsers*), nous créons une requête afin d'insérer manuellement les identifiants du compte de l'administrateur conformément à la demande du client qui souhaitait un compte impossible à créer depuis l'application :

```

SQLQuery4.sql *
aspnet-Jeux_Olympiques-5a81ba5
1  INSERT INTO AspNetUsers (Id, UserName, NormalizedUserName, Email, NormalizedEmail, EmailConfirmed, PasswordHash,
2  SecurityStamp, ConcurrencyStamp, PhoneNumber, PhoneNumberConfirmed, TwoFactorEnabled, LockoutEnd, LockoutEnabled,
3  AccessFailedCount, AccountKey, FirstName, LastName)
4  VALUES
5  (NEWID(), 'admin@jeuxolympiques.com', 'ADMIN@JEUXOLYMPIQUES.COM', 'admin@jeuxolympiques.com', 'ADMIN@JEUXOLYMPIQUES.COM', 1,
6  'AQAAAAIAAYagAAAE1swvJSYrab4nmV8D11AaNZIz/Owlf98691Z7X0Vxr3DCd/XzSutmpTNTMddrNjA==',
7  '65K4MQIYK40SUVYHUY46HRXZ3EUBN06R', '35c830fa-3d2d-44cb-b87d-580672737c2', NULL, 0, 0, NULL, 1, 0,
8  '546cb87093b63ed1acbc40747c99677eaf32b34d12a9c613295c586a98a7478e', 'AdminPrenom', 'AdminNom');

```

Requête SQL de création du compte Admin dans la table AspNetUsers

	Id	UserName	NormalizedUser...	Email	NormalizedEmail	EmailConfirmed	PasswordHash	SecurityStamp
1	1D6FDC68-4D11-...	admin@jeuxolympiques.com	ADMIN@JEUXOLYH...	admin@jeuxolym...	ADMIN@JEUXOLYH...	1	AQAAAAIAAYagAA...	65K4MQIYK40SUV...
2	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Extrait de la table AspNetUsers de la base de données avec le succès de la création du compte Admin

- la dernière de nos requêtes consiste à associer notre User à son rôle Admin. Les tables AspNetUsers et AspNetRoles sont en relation Many to Many, une table d'association AspNetUserRoles est créée. C'est dans cette dernière que nous ajoutons notre Admin.

```

SQLQuery6.sql *
dbo.AspNetRoles.data *
dbo.AspNetUsers.data *
dbo.AspNetRoles [Données]
dbo.AspNetUserRoles.sql
aspnet-Jeux_Olympiques-5a81ba5
1  DECLARE @userId nvarchar(450) = (SELECT Id FROM AspNetUsers WHERE Email = 'admin@jeuxolympiques.com')
2  DECLARE @roleId nvarchar(450) = (SELECT Id FROM AspNetRoles WHERE Name = 'Admin')
3  INSERT INTO AspNetUserRoles (UserId, RoleId)
4  VALUES (@userId, @roleId);

```

Requête SQL de création du compte Admin avec le rôle Admin dans la table de jonction AsNetUserRoles

Comme l'a souhaité le client, l'identifiant de l'administrateur de l'application est créé, il reste désormais à configurer les droits dans l'application. Nous avons modifié le menu de navigation du site pour y insérer un lien Dashboard uniquement accessible pour l'Administrateur authentifié.

```

@if (SignInManager.IsSignedIn(User) && (User.IsInRole("Admin")))
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Admin" asp-action="Dashboard">Dashboard</a>
    </li>
}


```

Extrait du fichier _LoginPartial.cshtml avec une boucle qui ouvre l'accès au Dashboard uniquement pour l'Admin.

e. Protection des attributions du rôle Admin

Le rôle *Admin* dispose de certains privilège qu'il faut protéger, en particulier celui de gérer l'application. *L'Admin* peut voir les statistiques de vente, gérer les utilisateurs et créer du contenu (actualités et billets à vendre).

Dans le cas d'un CRUD (Create, Read, Update, Delete), chaque tâche de création ou de suppression est soumise à une autorisation dont bénéficie uniquement l'*Admin* pour gérer le contenu de l'application.



```
52
53 // POST: Events/Create
54 // To protect from overposting attacks, enable the specific properties you want to bind to.
55 // For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
56 [HttpPost]
57 [ValidateAntiForgeryToken]
58 [Authorize(Roles = "Admin")]
59 public async Task<IActionResult> Create([Bind("Id,Entitled,Presentation,Date,Site")] Event @event)
60 {
61     if (ModelState.IsValid)
62     {
63         _context.Add(@event);
64         await _context.SaveChangesAsync();
65         return RedirectToAction(nameof(Index));
66     }
67     return View(@event);
68 }
```

Extrait de l'EventController.cs avec l'autorisation réservée à l'Admin

Ainsi, un utilisateur autre que l'*admin* se verra refuser l'accès à une fonctionnalité s'il venait à tenter d'y entrer par le biais de l'URL. Il sera renvoyer vers un formulaire de connexion. Une deuxième tentative le mènera vers une page affichant un message d'erreur mentionnant le refus d'accès.

Jeux_Olympiques Accueil Actualité Billeterie Confidentialité

Access denied

You do not have access to this resource.

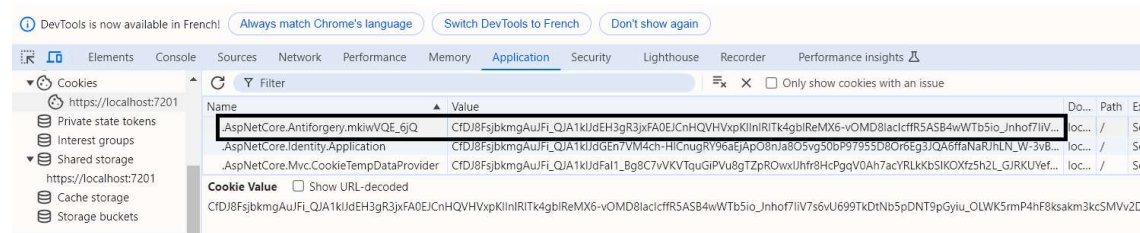
Message d'erreur s'affichant lors de la tentative d'accès à une page réservé à l'Admin

4. Protection contre les attaques CSRF

L'attaque CSRF (Cross-Site Request Forgery) consiste à faire exécuter à un utilisateur authentifié, une action malveillante sur un site web dans lequel il a confiance mais sans son consentement et à son insu. L'attaquant profite de celle-ci et de l'acceptation des cookies de session de l'utilisateur pour valider les requêtes.

Par exemple :

Pour empêcher les attaques CSRF, ASP.NET MVC utilise, par défaut, des tokens anti-falsification, également appelés « *request verification tokens* » avec ASP.NET Core Data Protection.



Exemple d'un cookie automatiquement généré lors de l'authentification d'un utilisateur

Conformément aux recommandations OWASP (organisation travaillant sur la sécurité des applications web qui propose des recommandations, des outils et des solutions de sécurisation), ce token obéit à plusieurs règles :

- Unique par session d'utilisateur
- Secret
- Imprévisible (grande valeur aléatoire générée par une méthode sécurisée).

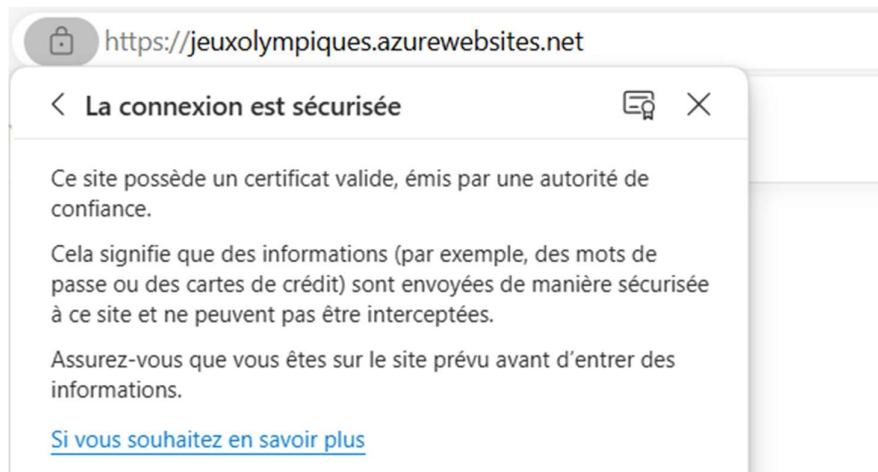
Même si la protection contre les attaques CSRF est incluse automatiquement, la documentation Microsoft fournit tous les détails pour éventuellement créer manuellement un Token.

5. Utilisation du protocole HTTPS lors du déploiement

L'échange des données par le protocole http (HyperText Transfer Protocol) présente plusieurs failles dont les plus dangereuses sont le transit limpide des données pouvant être lues par un tiers et un tiers pouvant se faire passer pour le serveur et réceptionner toutes les données qui sont transmises par le client.

Dans le cadre de notre application, il est impossible de courir un tel risque et le protocole HTTPS (HyperText Transfer Protocol Secure) est obligatoire.

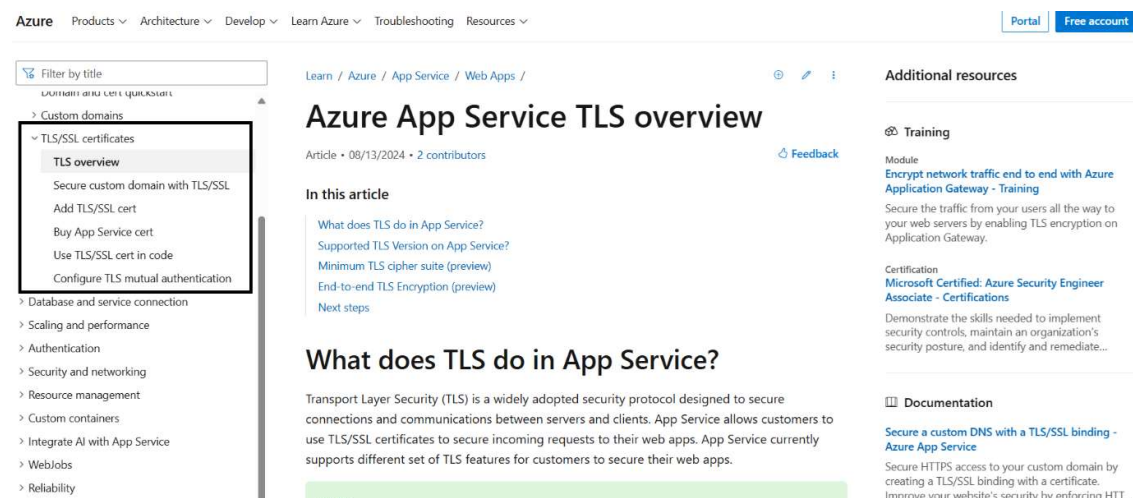
App Service de Microsoft Azure propose d'intégrer automatiquement un certificat managé TLS/SSL. Notre domaine étant un domaine par défaut (*azurewebsites.net*), il est automatiquement pris en charge par Azure. Cette option est à la fois simple, gratuite et répond à nos besoins dans le cadre de cette application. C'est pour cette raison que nous l'avons choisi.



Adresse du site employant le HTTPS avec l'affichage du certificat de sécurité

Les certificats TLS/SSL, Transport Layer Security et Secure Sockets Layer, assurent l'utilisation d'un protocole de sécurité qui protège les connexions et les communications entre les serveurs et les utilisateurs. Les requêtes et données entrées par les utilisateurs (identifiants, les mots de passe, les informations de paiement par exemple) à destination du serveur sont sécurisés grâce à un chiffrement et une confidentialité rendant illisible les données si elles sont saisies par une personne malveillante.

A défaut d'un certificat inclut par Azure App Service, il est tout à fait possible de se laisser guider par la documentation de Microsoft qui explique comment ajouter manuellement un certificat.



Extrait de la documentation (<https://learn.microsoft.com/en-us/azure/app-service/overview-tls>)

Après avoir consulté la documentation, on peut se rendre dans son espace afin de choisir le certificat le plus adapté selon ses besoins et les exigences de l'application :

Microsoft Azure **Mettre à niveau** Rechercher dans les ressources, services et documents (G+)

Accueil > Toutes les ressources > JeuxOlympiques | Certificats >

Sélectionner l'offre de tarification App Service

☒ Affichage matériel ☐ Affichage des fonctionnalités

Affichage de 16 offres de tarification App Service

Nom	ACU/processeur virtuel	Processeur virtuel	Mémoire (Go)	Stockage à distance (Go)	Mise à l'échelle (instance)	Contrat SLA	Coût horaire (instance)	Coût par mois (instance)
Dev/Test (Pour les charges de travail moins exigeantes)								
<input checked="" type="checkbox"/> De base B1	100	1	1,75	10	3	99,95 %	0,075 USD	54,75 USD
De base B2	100	2	3,5	10	3	99,95 %	0,15 USD	109,50 USD
De base B3	100	4	7	10	3	99,95 %	0,30 USD	219,00 USD
Production (Pour la plupart des charges de travail de production)								
Premium v3 P0V3	195*	1	4	250	30	99,95 %	0,169 USD	123,37 USD
Premium v3 P1V3	195	2	8	250	30	99,95 %	0,338 USD	246,74 USD
Premium v3 P2V3	195	4	16	250	30	99,95 %	0,676 USD	493,48 USD
Premium v3 P3V3	195	8	32	250	30	99,95 %	1,352 USD	986,96 USD
Hérité								
Standard S1	100	1	1,75	50	10	99,95 %	0,10 USD	73,00 USD
Standard S2	100	2	3,5	50	10	99,95 %	0,20 USD	146,00 USD
Standard S3	100	4	7	50	10	99,95 %	0,40 USD	292,00 USD
Premium P1	100	1	1,75	250	20	99,95 %	0,30 USD	219,00 USD

Extrait de la page de gestion des certificats depuis l'espace de gestion App Service Azure.

Dans notre fichier *program.cs*, *UseHttpsRedirection()* est une méthode qui ajoute un intergiciel pour rediriger les requêtes HTTP vers HTTPS. Cette méthode figure par défaut.

```

50
51
52 app.UseHttpsRedirection();
53 app.UseStaticFiles();
54
55 app.UseRouting();
56 app.UseAuthentication();
57 app.UseAuthorization();
58
59 app.MapControllerRoute(
60     name: "default",
61     pattern: "{controller=Home}/{action=Index}/{id?}");
62 app.MapRazorPages();
63
64 app.Run();
65

```

Extrait du fichier Program.cs


Partie 4. DEPLOIEMENT DE L'APPLICATION

L'application est déployée sur la plateforme cloud de Microsoft Azure. Depuis notre espace sur Azure Service App, nous pouvons superviser l'infrastructure.

1. Configuration de l'environnement de production

Pour des raisons de sécurité, il est important de passer d'un environnement de développement à un environnement de production avant le déploiement. En effet, l'environnement de développement peut activer des fonctionnalités qui ne doivent pas être exposées en production.

.NET fournit directement le fichier qui configure l'environnement de développement :



```
1 {
2   "$schema": "http://json.schemastore.org/launchsettings.json",
3   "iisSettings": {
4     "windowsAuthentication": false,
5     "anonymousAuthentication": true,
6     "iisExpress": {
7       "applicationUrl": "http://localhost:53980",
8       "sslPort": 44307
9     }
10  },
11  "profiles": {
12    "http": {
13      "commandName": "Project",
14      "dotnetRunMessages": true,
15      "launchBrowser": true,
16      "applicationUrl": "http://localhost:5148",
17      "environmentVariables": {
18        "ASPNETCORE_ENVIRONMENT": "Development"
19      }
20    },
21    "https": {
22      "commandName": "Project",
23      "dotnetRunMessages": true,
24      "launchBrowser": true,
25      "applicationUrl": "https://localhost:7201;http://localhost:5148",
26      "environmentVariables": {
27        "ASPNETCORE_ENVIRONMENT": "Development"
28      }
29    },
30    "IIS Express": {
31      "commandName": "IISExpress",
32      "launchBrowser": true,
33      "environmentVariables": {
34        "ASPNETCORE_ENVIRONMENT": "Development"
35      }
36    }
37  }
38 }
```

Extrait du fichier launchSettings.json

Comme nous pouvons le voir, .NET définit l'ensemble des variables d'environnement `ASPNETCORE_ENVIRONMENT` sur « Development ». Ce document est utilisé uniquement pour le développement en local, n'est pas déployé et contient les paramètres du profil. La documentation recommande de « *configurer l'environnement de production pour optimiser la sécurité, les performances et la robustesse de l'application.* ».

En cas de déploiement sans changement de la variable d'environnement, le message suivant apparaît lorsque l'on navigue sur l'application :

Error.

An error occurred while processing your request.

Request ID: 00-5a4d7bc79bad8b1febcb62b005a052f5-2092cd9d1dd4f5ea-00

Development Mode

Swapping to **Development** environment will display more detailed information about the error that occurred.

The **Development** environment shouldn't be enabled for deployed applications. It can result in displaying sensitive information from exceptions to end users. For local debugging, enable the **Development** environment by setting the **ASPNETCORE_ENVIRONMENT** environment variable to **Development** and restarting the app.

Message d'erreur apparaissant dans le navigateur

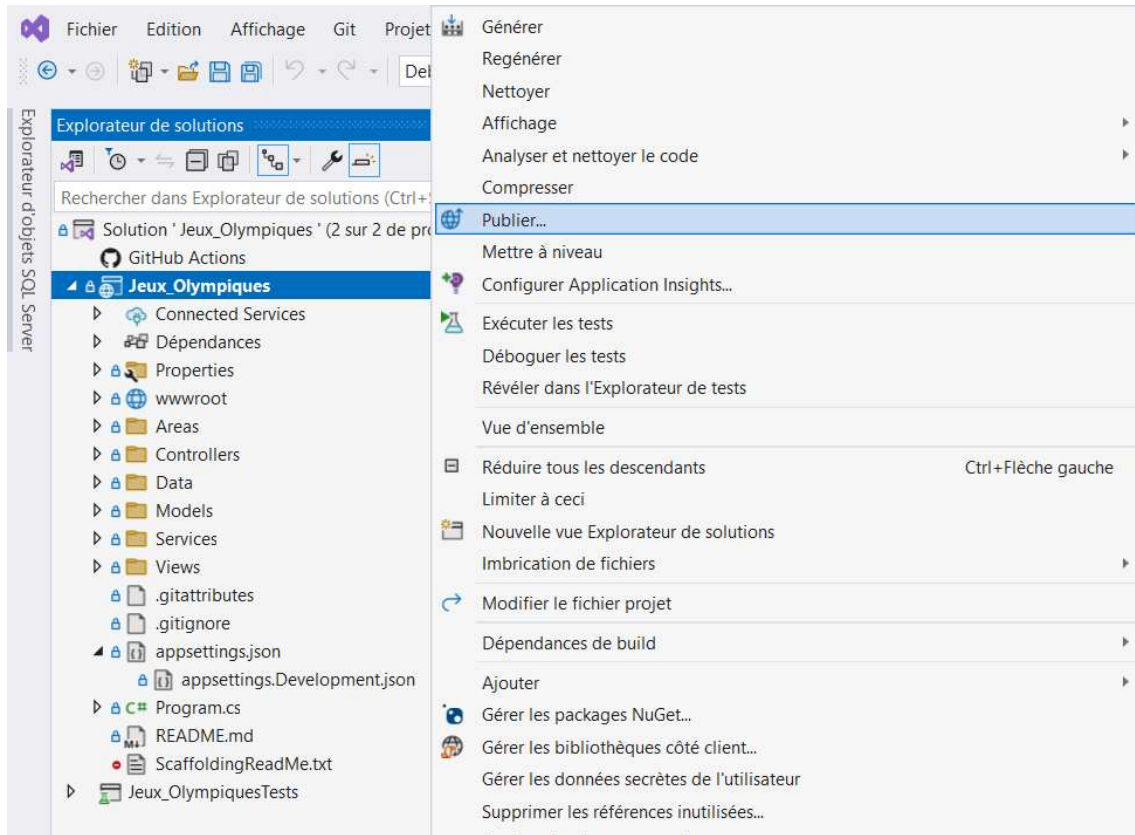
Cette page est directement configuré dans le fichier *Error.cshtml* :

```
1  @model ErrorViewModel
2  @{
3      ViewData["Title"] = "Error";
4  }
5
6  <h1 class="text-danger">Error.</h1>
7  <h2 class="text-danger">An error occurred while processing your request.</h2>
8
9  @if (Model.ShowRequestId)
10 {
11     <p>
12         <strong>Request ID:</strong> <code>@Model.RequestId</code>
13     </p>
14 }
15
16 <h3>Development Mode</h3>
17 <p>
18     Swapping to <strong>Development</strong> environment will display more detailed information about the error that occurred.
19 </p>
20 <p>
21     <strong>The Development environment shouldn't be enabled for deployed applications.</strong>
22     It can result in displaying sensitive information from exceptions to end users.
23     For local debugging, enable the <strong>Development</strong> environment by setting the <strong>ASPNETCORE_ENVIRONMENT</strong> environment variable
24     and restarting the app.
25 </p>
```

Extrait du fichier Error.cshtml qui configure le message d'erreur lors d'un déploiement

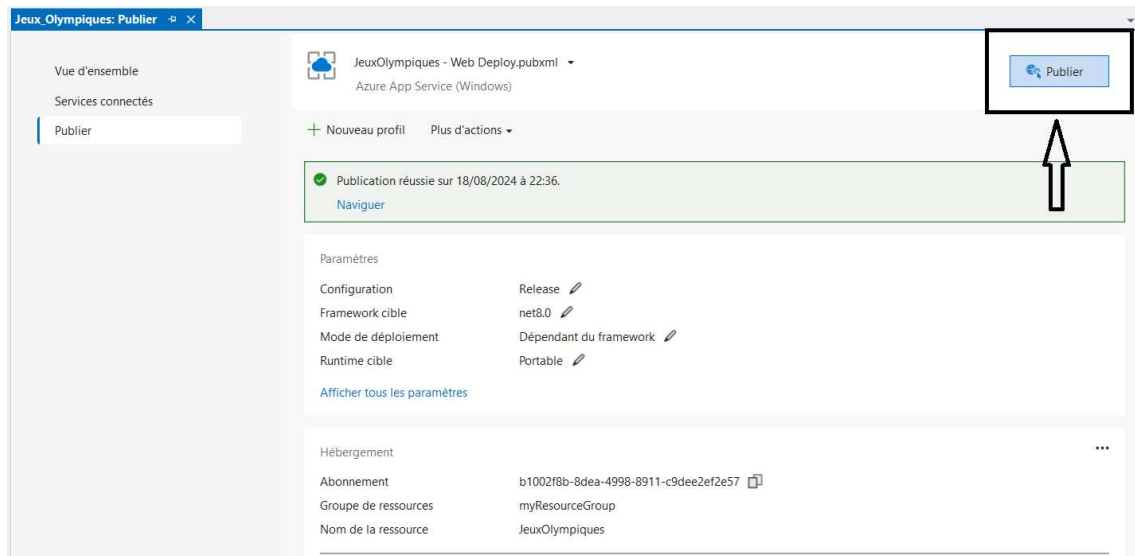
2. Publier l'application en ligne depuis l'interface de Visual Studio

Le déploiement de l'application peut se faire directement depuis Visual Studio en faisant un « clique droit » sur le répertoire principal de l'application et en sélectionnant *Publier* :



Extrait de l'interface de Visual Studio

Puis à l'apparition de la nouvelle page, nous cliquons sur *Publier* :



Extrait de l'interface de Visual Studio

Partie 5. EVOLUTION FUTURE DE L'APPLICATION

Les Jeux Olympiques est un évènement très ponctuel qui est organisé tous les 4 ans et le client nous a sollicité pour un besoin qui est immédiat. Cependant, des éléments sont à prendre en considération pour anticiper l'évolution de l'application. Un point d'attention doit être apporté sur la veille juridique tout au long du cycle de vie de l'application. Il est important d'anticiper les évolutions en matière de sécurité des données et de conformité réglementaire (RGPD, etc.).

1. L'application directement à la fin des Jeux Olympiques

Adaptabilité et réutilisabilité :

Bien que les Jeux Olympiques soient un événement ponctuel, l'application a été conçue de manière modulaire. Elle peut ainsi être adaptée facilement à d'autres grands événements sportifs.

2. Points d'attentions juridiques

a. Protection *a posteriori* des données des utilisateurs

Un nombre important d'utilisateur sont attendus. Leurs données personnelles sont conservées dans une base de données et elles doivent être, même après la compétition, soigneusement protégées.

Un audit de sécurité et de vérification de protection des données peut être effectué par un organisme officiel. Une bonne gestion des données permettra de fournir aux organismes assez facilement les informations demandées.

b. Conservation de l'historique de vente

De nombreuses ventes vont être effectuées depuis l'application pendant les Jeux Olympiques. Des contrôles peuvent être effectués par les organismes fiscaux pendant ou après la compétition. Il est donc important de conserver soigneusement l'historique de vente. Ce dernier est consultable depuis l'espace de gestion de l'administrateur de l'application.