Arborescence des fichiers de la solution eClassRoom

```
eClassRoom/
│
├── .github/
│   └── copilot-instructions.md
│
├── Client/
│   ├── Pages/
│   │   ├── Clients.razor
│   │   ├── Clients.razor.cs
│   │   ├── Utilisateurs.razor
│   │   ├── Utilisateurs.razor.cs
│   │   └── ... (autres pages Razor)
│   └── ... (autres dossiers/fichiers client)
│
├── Docs/
│   ├── Arborescence-Fichiers.md
│   ├── Clients.md
│   └── ... (autres docs techniques)
│
├── EFModel/
│   ├── EClassRoomDbContext.cs
│   └── Models/
│       ├── Client.cs
│       ├── Utilisateur.cs
│       ├── SalleDeFormation.cs
│       ├── MachineVirtuelle.cs
│       ├── Facture.cs
│       └── ProvisionningVM.cs
│
├── Server/
│   ├── Controllers/
│   │   ├── AuthController.cs
│   │   ├── ClientsController.cs
│   │   ├── UtilisateursController.cs
│   │   ├── SallesDeFormationController.cs
│   │   └── ... (autres contrôleurs)
│   ├── Services/
│   │   ├── ClientService.cs
│   │   ├── UtilisateurService.cs
│   │   ├── SalleDeFormationService.cs
│   │   └── ... (autres services)
│   ├── Program.cs
│   ├── appsettings.json
│   └── ... (autres fichiers serveur)
│
├── Shared/
│   └── Dtos/
│       ├── ClientDto.cs
```

```
│       ├── UtilisateurDto.cs
│       ├── SalleDeFormationDto.cs
│       ├── MachineVirtuelleDto.cs
│       └── ... (autres DTO)
│
├── Test.CLI/
│   └── Program.cs
│
└── ... (autres dossiers/fichiers racine)
```

Remarques :

- Les dossiers principaux correspondent à chaque couche de l'architecture (Client, Server, EFModel, Shared).
- Les fichiers `.razor` et `.razor.cs` sont dans `Client/Pages`.
- Les modèles EF sont dans `EFModel/Models`.
- Les DTO sont dans `Shared/Dtos`.
- Les contrôleurs et services sont dans `Server/Controllers` et `Server/Services`.
- Les fichiers de documentation sont dans `Docs/`.

Infrastructure Azure : création modulaire avec Azure SDK .NET 2.x

Vue d'ensemble

Ce service permet de créer des ressources Azure (Resource Group, VM, etc.) de façon modulaire, en utilisant le SDK Azure .NET 2.x (non Fluent).
Chaque étape (création du groupe, de la VM, etc.) est encapsulée dans une méthode réutilisable.

Fonctionnement

- **Authentification** : Utilise `DefaultAzureCredential` pour s'authentifier auprès d'Azure.
- **Création d'un Resource Group** : La méthode `CreateResourceGroupAsync` crée ou met à jour un groupe de ressources dans la région souhaitée.
- **Création d'une VM** : La méthode `CreateVirtualMachineAsync` crée une VM Windows 10 avec les paramètres fournis (nom, admin, NIC, etc.).
- **Extensibilité** : Vous pouvez ajouter d'autres méthodes pour créer les ressources réseau nécessaires (VNet, Subnet, IP publique, NIC).

Bonnes pratiques

- Les identifiants et secrets ne doivent jamais être stockés en clair dans le code.
- Utilisez des identités managées ou Azure Key Vault pour la gestion des secrets.
- Toutes les opérations sont asynchrones (`async/await`).
- Chaque étape est factorisée pour la réutilisabilité et la sécurité.

Références de code

- **AzureInfrastructureService.cs** : Service C# pour la gestion modulaire de l'infrastructure Azure.
- Documentation Azure SDK .NET
- Exemples Azure.ResourceManager

Infrastructure Azure : création modulaire avec Azure SDK .NET 2.x

Objectif

Automatiser la création d'objets Azure (Resource Groups, Machines Virtuelles, etc.) depuis l'application, en utilisant le SDK Azure .NET (version 2.x, non Fluent).

Prérequis

- Installer les packages NuGet suivants dans le projet concerné (exemple pour un projet Console ou Service) :
  - `Azure.Identity`
  - `Azure.ResourceManager`
  - `Azure.ResourceManager.Compute`
  - `Azure.ResourceManager.Resources`
- Disposer d'un compte Azure avec les droits nécessaires.
- Authentification recommandée : DefaultAzureCredential (gère Azure CLI, Managed Identity, etc.).

Étapes modulaires

1. Authentification et initialisation du client

```
using Azure.Identity;
using Azure.ResourceManager;
using Azure.ResourceManager.Resources;

var credential = new DefaultAzureCredential();
var armClient = new ArmClient(credential, "<subscription-id>");
```

2. Création d'un Resource Group

```
var rgCollection = armClient.GetDefaultSubscription().GetResourceGroups();
string rgName = "myResourceGroup";
string location = "westeurope";
var rgData = new ResourceGroupData(location);
var rgLro = await rgCollection.CreateOrUpdateAsync(WaitUntil.Completed, rgName,
rgData);
var resourceGroup = rgLro.Value;
```

3. Création d'une machine virtuelle

```
using Azure.ResourceManager.Compute;
using Azure.ResourceManager.Compute.Models;
using Azure.ResourceManager.Network;
using Azure.ResourceManager.Network.Models;

// Prérequis : créer un VNet, un subnet, une IP publique, une NIC (voir doc Azure)
// Exemple simplifié pour la VM :
var vmCollection = resourceGroup.GetVirtualMachines();
var vmData = new VirtualMachineData(location)
{
    HardwareProfile = new HardwareProfile { VmSize =
VirtualMachineSizeType.StandardD2V3 },
    StorageProfile = new StorageProfile
    {
        OSDisk = new OSDisk(DiskCreateOptionType.FromImage)
        {
            Name = "osdisk",
            Caching = CachingType.ReadWrite,
            ManagedDisk = new ManagedDiskParameters { StorageAccountType =
StorageAccountType.StandardLrs }
        },
        ImageReference = new ImageReference
        {
            Publisher = "MicrosoftWindowsDesktop",
            Offer = "Windows-10",
            Sku = "win10-21h2-pro",
            Version = "latest"
        }
    },
    OSProfile = new OSProfile
    {
        ComputerName = "vm-demo",
        AdminUsername = "azureuser",
        AdminPassword = "MotDePasseSecurise123!"
    },
    NetworkProfile = new NetworkProfile
    {
        NetworkInterfaces = {
            new NetworkInterfaceReference { Id = "<nic-resource-id>", Primary =
true }
        }
    }
};
var vmLro = await vmCollection.CreateOrUpdateAsync(WaitUntil.Completed, "vm-demo",
vmData);
var vm = vmLro.Value;
```

4. Modularité

- Chaque étape (création du groupe, du réseau, de la VM) doit être encapsulée dans une méthode/service réutilisable.
- Utilisez des paramètres pour rendre les méthodes génériques (nom, région, taille, image, etc.).
- Gérez les exceptions et les statuts d'opération (LRO).

Bonnes pratiques

- Ne stockez jamais de secrets ou mots de passe en clair dans le code.
- Utilisez des identités managées ou Azure Key Vault pour la gestion des secrets.
- Privilégiez l'asynchrone (`async/await`) pour toutes les opérations Azure.
- Logguez les opérations et surveillez les statuts de déploiement.

Références

- Documentation officielle Azure SDK .NET
- Exemples Azure.ResourceManager

Résumé:
La création d'objets Azure (Resource Group, VM, etc.) se fait de façon modulaire via le SDK Azure .NET 2.x, sans Fluent API, en utilisant des clients typés, des modèles de données et des opérations asynchrones. Chaque étape doit être factorisée pour la réutilisabilité et la sécurité.# Provisionnement métier d'une salle de formation Azure

Objectif métier

Permettre à chaque stagiaire d'une salle de formation d'obtenir une machine virtuelle dédiée dans Azure, avec une adresse IP publique pour se connecter à distance.

Processus

- Lors de la création d'une salle de formation, le système provisionne automatiquement une VM pour chaque stagiaire.
- Chaque VM est accessible via une IP publique, transmise au stagiaire pour sa session.
- Toutes les informations de provisionnement sont historisées pour traçabilité et support.

Bénéfices

- Automatisation du déploiement des environnements de formation.
- Attribution claire des ressources à chaque stagiaire.
- Suivi et audit des accès et des ressources provisionnées.

Données stockées

- Pour chaque stagiaire : salle, VM, IP publique, date de provisionnement.
- Ces données sont utilisées pour l'affichage, la gestion et la facturation.

Résumé :
Ce processus garantit que chaque stagiaire dispose d'un environnement isolé, traçable et prêt à l'emploi pour

la formation, tout en permettant à l'organisation de piloter et d'auditer l'usage des ressources Azure.

Provisionnement Azure d'une salle de formation – Documentation technique

Fonctionnement

- Le service `AzureSalleDeFormationService` orchestre la création d'une salle de formation Azure.
- Pour chaque stagiaire, il provisionne une VM, une IP publique, une NIC, et stocke l'adresse IP publique en base.
- Les informations sont persistées dans la table `ProvisionningVM` (EF).
- La méthode `ProvisionnerSalleAsync` retourne un objet `ProvisionningResultDto` contenant la salle et la liste des IPs publiques par stagiaire.

Flux technique

1. Création du Resource Group, VNet, Subnet pour la salle.
2. Pour chaque stagiaire :
    - Création d'une IP publique, d'une NIC, puis d'une VM.
    - Récupération de l'adresse IP publique.
    - Ajout d'un enregistrement dans la table `ProvisionningVM`.
3. Retour d'un DTO avec la salle et la liste des IPs publiques.

Modèle EF

- **ProvisionningVM** : table de liaison entre salle, stagiaire, VM et IP publique.
- Ajout du DbSet dans `EClassRoomDbContext` et configuration des relations.

DTO

- **ProvisionningResultDto** : contient la salle et la liste des stagiaires avec leur IP publique.

Sécurité

- L'appel à Azure se fait avec un compte technique sécurisé.
- Les informations sensibles (IP, VM) sont stockées côté serveur et transmises uniquement aux utilisateurs autorisés.

Références de code :

- `AzureSalleDeFormationService.cs`
- `AzureInfrastructureService.cs`
- `ProvisionningVM.cs` (EF)
- `ProvisionningResultDto.cs` (DTO)

Documentation technique : Gestion des clients

Vue d'ensemble

La gestion des clients repose sur l'architecture multi-couches :

- **Client** (Blazor WebAssembly) : Affichage, création et gestion des clients via une page Razor et un fichier code-behind.
- **Server** : Expose des endpoints REST sécurisés pour la gestion des clients, en s'appuyant sur la couche service.
- **Shared** : Utilisation de DTO (`ClientDto`) pour échanger les données de façon sécurisée.
- **EFModel** : Entité `Client` mappée sur la base PostgreSQL.

Fonctionnalités côté client

- La page `Clients.razor` affiche la liste des clients dans un tableau BootstrapBlazor.
- Un bouton "Nouveau Client" ouvre un formulaire modal pour la création d'un client.
- Toute la logique métier (chargement, création, gestion du modal) est déportée dans le fichier code-behind `Clients.razor.cs`.
- Les appels HTTP sont réalisés via `HttpClient` et consomment l'API REST du serveur.
- Après création d'un client, la liste est automatiquement rafraîchie.

Fonctionnalités côté serveur

- Le contrôleur REST reçoit les requêtes, les valide et délègue à la couche service (`ClientService`).
- Le service effectue la conversion entité <-> DTO et gère la persistance via Entity Framework.
- Les mots de passe administrateurs sont traités de façon sécurisée (jamais exposés côté client).

Sécurité

- Les endpoints sont sécurisés par authentification JWT.
- Les DTO permettent de maîtriser les données exposées au client.

Références de code

- `Client/Pages/Clients.razor` : UI et modal de création.
- `Client/Pages/Clients.razor.cs` : Code-behind, logique métier et appels API.
- `Shared/Dtos/ClientDto.cs` : Définition du DTO.
- `Server/Services/ClientService.cs` : Service métier côté serveur.

Exemple d'usage

1. L'utilisateur clique sur "Nouveau Client", saisit les informations et valide.
2. Le client Blazor envoie la requête POST à l'API.
3. Le serveur crée le client, retourne le DTO.
4. Le client recharge la liste et affiche le nouveau client.

Cette documentation est générée automatiquement à chaque évolution du code lié à la gestion des clients.

- Le contrôleur `ClientsController` reçoit les requêtes du client, valide les données, puis délègue la logique à la couche de services (`ClientService`).

- La couche de services utilise Entity Framework (`EFModel`) pour interagir avec la base PostgreSQL (lecture, création, modification, suppression d'entités `Client`).
- Les entités EF ne sont jamais exposées directement au client : seules les données des DTO transitent.

Résumé du flux

1. L'utilisateur interagit avec la page `Clients.razor` (affichage, ajout, édition, suppression).
2. Le code-behind effectue des appels HTTP vers l'API REST du serveur.
3. Le serveur traite les requêtes via le contrôleur et la couche de services, qui utilise EF pour manipuler la base de données.
4. Les réponses sont renvoyées sous forme de DTO et affichées côté client.

Références de code

- **Clients.razor** : UI et composants BootstrapBlazor.
- **Clients.razor.cs** : Logique métier côté client, appels API, gestion des états.
- **ClientsController.cs** (Server) : Endpoints REST pour la gestion des clients.
- **Services/ClientService.cs** (Server) : Logique métier et accès aux données via EF.
- **EFModel/Models/Client.cs** : Entité EF représentant un client.
- **Shared/ClientDto.cs** : DTO utilisé pour le transport des données client.

Documentation technique : Gestion des clients

Service côté serveur (`ClientService`)

Le service `ClientService` centralise la logique métier liée aux clients :

- Fournit des méthodes asynchrones pour lister, ajouter, modifier et supprimer des clients.
- Utilise les DTO (`ClientDto`) pour échanger les données entre le serveur et le client, évitant d'exposer directement les entités EF.
- Garantit que les mots de passe administrateurs ne sont jamais exposés côté client.

Page Razor côté client (`Clients.razor`)

- Affiche la liste des clients dans un tableau BootstrapBlazor.
- Récupère les données via un appel HTTP GET à l'API REST (`api/clients`).
- Utilise les DTO pour le binding et l'affichage.

Sécurité et architecture

- Les opérations CRUD sont sécurisées côté serveur (authentification requise).
- L'utilisation de DTO permet de maîtriser les données exposées au client.
- L'architecture en couches facilite la maintenance et l'évolution de l'application.

Références

- `Server/Services/ClientService.cs`
- `Client/Pages/Clients.razor`

Création de l'architecture multi-couches avec la CLI .NET

Voici les commandes pour générer la structure de l'application :

1. **Créer la solution principale**

```
dotnet new sln -n eClassRoom
```

2. **Créer le projet Blazor WebAssembly (Client)**

```
dotnet new blazorwasm -n Client -o Client --no-https
```

3. **Créer le projet Web API (Server)**

```
dotnet new webapi -n Server -o Server
```

4. **Créer la bibliothèque partagée pour les DTO (Shared)**

```
dotnet new classlib -n Shared -o Shared
```

5. **Créer le projet EFModel pour les entités et le DbContext**

```
dotnet new classlib -n EFModel -o EFModel
```

6. **Ajouter les projets à la solution**

```
dotnet sln add .\Client\Client.csproj
dotnet sln add .\Server\Server.csproj
dotnet sln add .\Shared\Shared.csproj
dotnet sln add .\EFModel\EFModel.csproj
```

7. **Ajouter les références de projet**

   - Le serveur doit référencer `Shared` et `EFModel` :

```
dotnet add .\Server\Server.csproj reference .\Shared\Shared.csproj
dotnet add .\Server\Server.csproj reference .\EFModel\EFModel.csproj
```

- Le client doit référencer `Shared`:

```
dotnet add .\Client\Client.csproj reference .\Shared\Shared.csproj
```

8. **Installer les packages nécessaires**

- Pour Entity Framework Core et PostgreSQL dans `EFModel` et `Server`:

```
dotnet add .\EFModel\EFModel.csproj package
Microsoft.EntityFrameworkCore
dotnet add .\EFModel\EFModel.csproj package
Npgsql.EntityFrameworkCore.PostgreSQL
dotnet add .\Server\Server.csproj package
Microsoft.EntityFrameworkCore.Design
dotnet add .\Server\Server.csproj package
Microsoft.AspNetCore.Authentication.JwtBearer
```

Résumé

- **Client**: Blazor WebAssembly pour l'UI.
- **Server**: API REST .NET.
- **Shared**: DTO partagés.
- **EFModel**: Entités EF Core et DbContext.

Cette structure respecte l'architecture multi-couches décrite dans la documentation du projet.

Arborescence complète des fichiers de la solution eClassRoom

```
eClassRoom/
│
├── .gitignore
├── eClassRoom.sln
├── files.txt
├── README.md
│
├── .github/
│   ├── copilot-instructions.md
│   └── workflows/
│
├── Client/
│   ├── App.razor
│   ├── Client.csproj
│   ├── Program.cs
```

```
│   ├── _Imports.razor
│   ├── Pages/
│   │   ├── Clients.razor
│   │   ├── Clients.razor.cs
│   │   ├── Counter.razor
│   │   ├── Index.razor
│   │   ├── Login.razor
│   │   ├── Login.razor.cs
│   │   ├── MachinesVirtuelles.razor
│   │   ├── MachinesVirtuelles.razor.cs
│   │   ├── SalleDeFormation.razor
│   │   ├── SallesDeFormation.razor
│   │   ├── SallesDeFormation.razor.cs
│   │   ├── Utilisateurs.razor
│   │   ├── Utilisateurs.razor.cs
│   │   └── Weather.razor
│   ├── Layout/
│   │   ├── MainLayout.razor
│   │   ├── MainLayout.razor.css
│   │   ├── NavMenu.razor
│   │   └── NavMenu.razor.css
│   ├── Properties/
│   │   └── launchSettings.json
│   ├── wwwroot/
│   │   ├── favicon.png
│   │   ├── icon-192.png
│   │   ├── index.html
│   │   ├── css/
│   │   │   └── app.css
│   │   ├── lib/
│   │   │   └── bootstrap/
│   │   │       └── dist/
│   │   │           ├── css/
│   │   │           └── js/
│   │   └── sample-data/
│   │       └── weather.json
│   └── ... bin/, obj/ (build system)
│
├── Docs/
│   ├── Arborescence-Fichiers.md
│   ├── AzureInfrastructure.md
│   ├── AzureSalleDeFormation_metier.md
│   ├── AzureSalleDeFormation_sequence.puml
│   ├── AzureSalleDeFormation_technique.md
│   ├── Clients.md
│   ├── ClientsService.md
│   ├── dotnet-cli-creation.md
│   ├── EFModel.jpg
│   ├── EFModel.puml
│   ├── Files.md
│   ├── GestionJWT.md
│   ├── introduction m�tier.puml
│   ├── Login.md
│   ├── MachinesVirtuelles.md
```

```
│   ├── MachinesVirtuelles_Metier.md
│   ├── MyProject.md
│   ├── MyProject.pdf
│   ├── m�tier.png
│   ├── Project.md
│   ├── Project.pdf
│   ├── Provisionner_SalleDeFormation.png
│   ├── SallesDeFormation.md
│   ├── SallesDeFormation_Metier.md
│   ├── scenario.png
│   ├── scenario.puml
│   ├── Specifications.md
│   ├── Utilisateurs.md
│   ├── Utilisateurs_Metier.md
│   └── VSCode_Chat_CopilotForGitHub.png
│
├── EFMigrator/
│   ├── appsettings.json
│   ├── EFMigrator.csproj
│   ├── Program.cs
│   └── ... bin/, obj/ (build system)
│
├── EFModel/
│   ├── appsettings.json
│   ├── Class1.cs
│   ├── EClassRoomDbContext.cs
│   ├── EClassRoomDbContextFactory.cs
│   ├── EFModel.csproj
│   ├── Migrations/
│   │   ├── 20250918143152_InitialCreate.cs
│   │   ├── 20250918143152_InitialCreate.Designer.cs
│   │   ├── 20250918161858_Update1.cs
│   │   ├── 20250918161858_Update1.Designer.cs
│   │   ├── 20250918182208_Update2.cs
│   │   ├── 20250918182208_Update2.Designer.cs
│   │   ├── 20250918182812_Update3.cs
│   │   ├── 20250918182812_Update3.Designer.cs
│   │   ├── 20250918185926_Update4.cs
│   │   ├── 20250918185926_Update4.Designer.cs
│   │   ├── 20250918191931_Update5.cs
│   │   ├── 20250918191931_Update5.Designer.cs
│   │   └── EClassRoomDbContextModelSnapshot.cs
│   ├── Models/
│   │   ├── Client.cs
│   │   ├── Facture.cs
│   │   ├── MachineVirtuelle.cs
│   │   ├── ProvisionningVM.cs
│   │   ├── SalleDeFormation.cs
│   │   └── Utilisateur.cs
│   └── ... bin/, obj/ (build system)
│
├── Server/
│   ├── appsettings.Development.json
│   ├── appsettings.json
```

```
│   ├── Program.cs
│   ├── Server.csproj
│   ├── Server.csproj.user
│   ├── Server.http
│   ├── Startup.cs
│   ├── Controllers/
│   │   ├── AuthController.cs
│   │   ├── ClientController.cs
│   │   ├── Controllers2/
│   │   ├── MachinesController.cs
│   │   ├── SallesController.cs
│   │   └── UsersController.cs
│   ├── Properties/
│   │   └── launchSettings.json
│   ├── Services/
│   │   ├── AuthService.cs
│   │   ├── AzureInfrastructureService.cs
│   │   ├── AzureSalleDeFormationService.cs
│   │   ├── ClientService.cs
│   │   ├── MachineVirtuelleService.cs
│   │   ├── SalleDeFormationService.cs
│   │   └── UtilisateurService.cs
│   ├── WeatherForecast.cs
│   └── ... bin/, obj/ (build system)
│
├── Shared/
│   ├── Class1.cs
│   ├── Shared.csproj
│   ├── Dtos/
│   │   ├── ClientDto.cs
│   │   ├── LoginDto.cs
│   │   ├── MachineVirtuelleDto.cs
│   │   ├── ProvisionningResultDto.cs
│   │   ├── SalleDeFormationDto.cs
│   │   └── UtilisateurDto.cs
│   └── ... bin/, obj/ (build system)
│
├── Test.CLI/
│   ├── appsettings.json
│   ├── Program.cs
│   ├── Test.CLI.csproj
│   ├── Properties/
│   │   └── launchSettings.json
│   └── ... bin/, obj/ (build system)
```

Gestion de l’authentification JWT (JSON Web Token)

Vue d’ensemble

L’application utilise l’authentification basée sur des tokens JWT pour sécuriser l’accès à l’API REST. Cette approche permet de vérifier l’identité des utilisateurs et de protéger les endpoints sensibles.

Architecture concernée

- **Server** : Génère et valide les tokens JWT. Les contrôleurs REST sont protégés par l'attribut `[Authorize]`.
- **Client** : Récupère le token lors de la connexion et l'ajoute dans l'en-tête `Authorization` des requêtes HTTP.
- **Shared** : Utilise des DTO pour transporter les données d'authentification.
- **EFModel** : Utilisé pour valider les identifiants côté serveur (vérification du hash du mot de passe).

Étapes principales

1. Configuration du serveur

- Ajout du package NuGet `Microsoft.AspNetCore.Authentication.JwtBearer`.
- Configuration de l'authentification JWT dans `Program.cs` : définition du schéma, des paramètres de validation, et du secret partagé (stocké dans `appsettings.json`).
- Ajout de l'appel à `app.UseAuthentication()` et `app.UseAuthorization()` dans le pipeline.

2. Endpoint d'authentification

- Création d'un contrôleur `AuthController` avec une méthode `Login` qui :
  - Valide les identifiants reçus (vérification du hash du mot de passe).
  - Génère un token JWT signé si l'authentification réussit.
  - Retourne le token au client dans la réponse.

3. Sécurisation des endpoints

- Ajout de l'attribut `[Authorize]` sur les contrôleurs ou actions à protéger.
- Seuls les clients présentant un JWT valide peuvent accéder à ces endpoints.

4. Côté client (Blazor WebAssembly)

- Lors de la connexion, le client envoie les identifiants à l'API `/api/auth/login`.
- Si la connexion réussit, le token JWT est stocké dans le `localStorage` du navigateur.
- Un `DelegatingHandler` personnalisé ajoute automatiquement le token dans l'en-tête `Authorization` (`Bearer {token}`) de chaque requête HTTP vers l'API.

Points de sécurité

- Les mots de passe sont stockés de façon sécurisée (hashés) côté serveur.
- Le secret JWT doit être complexe et stocké de façon sécurisée (jamais dans le code source).
- Les tokens ont une durée de vie limitée (ex : 2h).
- Les données sensibles ne transitent jamais en clair.

Avantages de cette approche

- Séparation claire des responsabilités entre les couches.
- Sécurité renforcée grâce à l'utilisation de JWT et de DTO.
- Facilité d'extension pour la gestion des rôles, des permissions, ou du rafraîchissement de token.

Références de code

- **Program.cs** : Configuration JWT.
- **appsettings.json** : Paramètres du token.
- **AuthController.cs** : Génération et validation du JWT.
- **ClientsController.cs** : Exemple de protection d'un endpoint.
- **Login.razor.cs** : Récupération et stockage du token côté client.
- **Program.cs (Client)** : Injection automatique du token dans les requêtes HTTP.

Documentation de la page Login.razor

Rôle de la page

La page `Login.razor` permet à l'utilisateur de s'authentifier dans l'application. Elle présente un formulaire de connexion (nom d'utilisateur et mot de passe) et gère l'envoi des identifiants au serveur pour obtenir un token JWT.

Dépendances principales

- **HttpClient** : Utilisé pour envoyer la requête POST d'authentification à l'API REST du serveur (`/api/auth/login`).
- **IJSRuntime** : Permet de stocker le token JWT dans le `localStorage` du navigateur côté client.
- **DTO LoginDto** (projet Shared) : Sert à transporter les identifiants de connexion du client vers le serveur.
- **TokenResponse** : Objet utilisé pour désérialiser la réponse contenant le token JWT.

Fonctionnement et interactions

1. **Affichage du formulaire**
   La page Razor affiche un formulaire BootstrapBlazor pour saisir le nom d'utilisateur et le mot de passe.

2. **Soumission du formulaire**
   Lors de la soumission, la méthode `HandleLogin()` du code-behind est appelée.

3. **Appel à l'API d'authentification**

   - `HttpClient.PostAsJsonAsync` envoie les identifiants au serveur.
   - Le serveur (contrôleur `AuthController`) reçoit la requête, valide les identifiants via la couche de services (qui utilise EF pour accéder à la table des utilisateurs et vérifier le hash du mot de passe).
   - Si la validation réussit, le serveur génère un JWT et le retourne dans la réponse.

4. **Stockage du token côté client**

   - Le code-behind récupère le token JWT de la réponse.
   - Il utilise `IJSRuntime` pour stocker ce token dans le `localStorage` du navigateur.

5. **Utilisation du token pour les requêtes suivantes**

   - Un `DelegatingHandler` personnalisé injecte automatiquement le token JWT dans l'en-tête `Authorization` de chaque requête HTTP vers l'API.

    - Cela permet d'accéder aux endpoints protégés par `[Authorize]` côté serveur.

Interactions avec les services et Entity Framework

- **Côté serveur** :
    - Le contrôleur `AuthController` délègue la validation des identifiants à un service d'authentification (dans le dossier `Services`).
    - Ce service utilise Entity Framework (`EFModel`) pour accéder à la base PostgreSQL, rechercher l'utilisateur et comparer le hash du mot de passe.
    - Si la validation est positive, le service génère un JWT.
- **Côté client** :
    - La page `Login.razor` n'accède pas directement à la base de données ni à EF : elle interagit uniquement avec l'API via `HttpClient` et gère le stockage du token.

Sécurité

- Les mots de passe ne transitent jamais en clair : seul le hash est comparé côté serveur.
- Le token JWT est stocké côté client et transmis dans l'en-tête `Authorization` pour chaque requête protégée.

Résumé du flux

1. L'utilisateur saisit ses identifiants sur `Login.razor`.
2. Le code-behind envoie ces identifiants à l'API d'authentification.
3. Le serveur valide via EF et, si OK, retourne un JWT.
4. Le client stocke le JWT et l'utilise pour les requêtes futures.

Références de code

- **Login.razor** : Formulaire et UI.
- **Login.razor.cs** : Logique de soumission, appel API, gestion du token.
- **AuthController.cs** (Server) : Endpoint d'authentification.
- **Services/AuthService.cs** (Server) : Validation des identifiants via EF.
- **EFModel/Models/Utilisateur.cs** : Entité utilisateur pour la validation.

Documentation de la page MachinesVirtuelles.razor

Rôle de la page

La page `MachinesVirtuelles.razor` permet de gérer les machines virtuelles (affichage, création, modification, suppression) via une interface utilisateur Blazor.

Dépendances principales

- **HttpClient** : Pour les appels HTTP (GET, POST, PUT, DELETE) vers l'API REST `/machines`.

- **MachineVirtuelleDto** (projet Shared) : Sert à transporter les données des machines virtuelles entre le client et le serveur.
- **BootstrapBlazor** : Pour l'UI (tableaux, formulaires).
- **IJSRuntime** : Peut être utilisé pour des interactions avancées côté client.

Fonctionnement et interactions

1. **Chargement des machines virtuelles**
    - Appel GET `/machines` pour récupérer la liste.
    - Affichage dans un tableau.
2. **Création**
    - Bouton "Nouvelle machine virtuelle" ouvre le formulaire.
    - POST `/machines` avec le DTO pour créer une VM.
3. **Modification**
    - Bouton "Modifier" ouvre le formulaire prérempli.
    - PUT `/machines/{id}` avec le DTO pour modifier.
4. **Suppression**
    - Bouton "Supprimer" appelle DELETE `/machines/{id}`.

Interactions Services et EF

- **Côté client** :
    - Utilise uniquement les DTO et l'API REST.
- **Côté serveur** :
    - Le contrôleur `/machines` reçoit les requêtes, valide et délègue à la couche de services (`MachineVirtuelleService`).
    - La couche de services utilise Entity Framework pour manipuler les entités `MachineVirtuelle` dans la base PostgreSQL.
    - Les entités EF ne sont jamais exposées directement : seules les données des DTO transitent.

Architecture côté serveur

- **MachinesController** : Contrôleur Web API qui expose les endpoints REST pour la gestion des machines virtuelles. Il reçoit les requêtes du client, les valide et appelle la couche de service.
- **MachineVirtuelleService** : Service métier qui centralise la logique de gestion des machines virtuelles. Il utilise le `EClassRoomDbContext` (Entity Framework) pour accéder à la base de données et effectuer les opérations CRUD sur les entités `MachineVirtuelle`.

Résumé du flux

1. L'utilisateur interagit avec la page (affichage, ajout, édition, suppression).
2. Le code-behind effectue des appels HTTP vers l'API REST.
3. Le serveur traite via le contrôleur et la couche de services, qui utilise EF pour la base de données.

4. Les réponses sont renvoyées sous forme de DTO et affichées côté client.

Références de code

- **MachinesVirtuelles.razor** : UI et composants.
- **MachinesVirtuelles.razor.cs** : Logique métier côté client, appels API.
- **MachinesController.cs** (Server) : Endpoints REST pour la gestion des machines virtuelles.
- **Services/MachineVirtuelleService.cs** (Server) : Logique métier et accès aux données via EF.
- **EFModel/Models/MachineVirtuelle.cs** : Entité EF.
- **Shared/MachineVirtuelleDto.cs** : DTO utilisé pour le transport des données.

Documentation métier : Gestion des machines virtuelles

1. Rôle métier des machines virtuelles

Dans eClassRoom, une **machine virtuelle** (VM) représente un environnement informatique individuel provisionné pour un stagiaire ou un formateur dans le cadre d'une salle de formation. Chaque VM est associée à une session de formation et permet d'accéder à un poste de travail distant (Windows ou Linux) avec une configuration adaptée (type, OS, disque, etc.).

Les machines virtuelles sont créées, modifiées ou supprimées par les administrateurs via l'interface web. Elles sont provisionnées dans Azure et associées à des utilisateurs et des salles de formation.

2. Attributs métier d'une machine virtuelle

Les principaux attributs d'une VM sont :

- **Id** : Identifiant unique.
- **Nom** : Nom de la machine virtuelle.
- **TypeOs** : Système d'exploitation (Windows, Linux).
- **TypeVm** : Type de VM Azure (ex : Standard_D2s_v3).
- **Sku** : SKU Azure.
- **Offer** : Offre Azure.
- **Version** : Version de l'image.
- **DiskIso** : ISO ou disque utilisé pour l'initialisation.
- **NomMarketing** : Nom commercial ou marketing de la VM.

3. DTO (Data Transfer Object)

Le **MachineVirtuelleDto** (dans le projet Shared) permet de transporter les données d'une VM entre le client et le serveur sans exposer directement l'entité EF. Il reprend les attributs métier nécessaires à l'UI et aux échanges API.

4. Service métier

Le **MachineVirtuelleService** (dans le dossier Services du Server) centralise la logique métier :

- Récupération de la liste des VMs.
- Création d'une nouvelle VM à partir d'un DTO.
- Modification et suppression d'une VM existante.
- Utilisation du DbContext EF pour toutes les opérations sur la base.

Ce service garantit la cohérence métier et isole la logique d'accès aux données.

5. Couche Entity Framework (EF)

L'entité **MachineVirtuelle** (dans EFModel/Models) correspond à la table des machines virtuelles en base PostgreSQL.
Le **EClassRoomDbContext** expose un DbSet pour permettre les opérations CRUD via EF Core.

Les relations éventuelles (ex: association à une salle de formation ou à un utilisateur) sont gérées dans le modèle EF.

6. Flux métier

1. L'utilisateur (admin) crée/modifie/supprime une VM via l'UI.
2. Le client Blazor envoie un DTO au serveur via l'API REST.
3. Le contrôleur appelle le service, qui applique la logique métier et utilise EF pour persister les changements.
4. Les données sont échangées sous forme de DTO, jamais d'entités EF.

7. Sécurité et bonnes pratiques

- Les endpoints sont protégés par JWT et `[Authorize]`.
- Seuls les administrateurs ou formateurs autorisés peuvent gérer les VMs.
- Les données sensibles (ex: configuration, accès) ne sont jamais exposées directement.

Résumé:
La gestion des machines virtuelles dans eClassRoom repose sur une séparation stricte entre la couche métier (service), la couche d'accès aux données (EF), et la couche de présentation (DTO/UI), garantissant sécurité, évolutivité et clarté du modèle métier.

Présentation de mon projet

Chapitre 1 - Gestion des salles de réunion 1.1 Description métier de l'application Un client de Aurera passe un contrat pour gérer des salles de formation sur la plateforme eClassRoom. Cette plateforme est hébergée sur Azure. Le client va utiliser un site Web pour renseigner ses administrateurs, ses formateurs et ses stagiaires et créer des salles de formation. POur créer une salle, le client et son représentant, identifié par un email, renseigne le formateur et les stagiaires, choisi son type de machine virtuelle (VM) LInux ou Windows, modèle de VM, OS avec détails (sku, offer, version) et le modèle de disk ISO préconfiguré. L'appuie sur le bouton Provisionner permet la création des machines virtuelles dans Azure et on met à disposition dans un portail l'accès aux fichiers unitaires RDP pour que chaque stagiaire puisse se connecter à sa VM.

1.2 Structure de projet initiale Dans le répertoire SRC, je veux construire une arborescence pour un projet Visual Studio 2019 de type WebAssembly Blazor avec modèle Entity Framework pour PostgreSQL et une

couche de services et des End Points REST en Web API .NET C#. Tu peux m'aider ? [Create Workspace...]

1.3 Le modèle Entity Framework (EF) Voici un e description simple du modèle de données. La société Aurera possèdes clients pour la solution eClassRoom. Un client possède les attributs suivants : nom de société, adresse, code postal, ville, pays, email de l'administrateur, mobile optionnel. On fournit un mot de passe à l'administrateur d'un client. cela implique qu'il existe une table des utilisateurs. Chaque utilisateur possède un email, nom, prénom, mot de passe et un role (admin, formateur, stagiaire). UN client peut créer des salles de formations. Un salle de formation a les attributs suivants: un nom de formation, un formateur, des stagiaires, une date de début, une date de fin, les caractéristiques des machines virtuelles (type Windows ou Linux, type de VM Azure, sku, offer, version, disk ISO spécifique, nom marketing de la VM). Chaque stagiaire possède une machine virtuelle et via le site web, il peut accèder au fichier RDP de connexion. Les VM sont supervisées et leurs utilisations est tracées pour permettre de créer des factures menseulles pour les clients. Tu peux me générer le modèle dans une sytnthèse en français puis le code entity framework en C# dans un projet VS EFModel ? Tu peux créer le DBContext ?

Arborescence à créer :

SRC └── EFModel ├── EFModel.csproj ├── EClassRoomDbContext.cs └── Models ├── Client.cs ├── Utilisateur.cs ├── SalleDeFormation.cs ├── MachineVirtuelle.cs └── Facture.cs

OK pour lier les stagiaires à une salle

Tu peux me créer les fichiers src\Client\Client.csproj et src\Server\Server.csproj avec .NET 8

1.4 Page de Login Tu pourrais me coder la page Login.razor et son code Login.razor.cs qui permet de saisir email et mot de passe. Sur l'appuie du bouton "Connecter", la page fait un appel httpClient vers le end point Web API /login. Le controller IIS va utiliser une couche de Service (que tu vas créer) qui utilise la couche Entity Framework et qui va exécuter la requête de vérifcation qu'un utilisateur existe et que le mot de passe est correct.

Pour installer le package NuGet `Npgsql.EntityFrameworkCore.PostgreSQL`, exécute cette commande dans le terminal à la racine de ton projet Server :

```
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
```

Ou via Visual Studio :

1. Clique droit sur le projet **Server** > **Gérer les packages NuGet**.
2. Recherche `Npgsql.EntityFrameworkCore.PostgreSQL`.
3. Clique sur **Installer**.

Après installation, tu pourras utiliser `options.UseNpgsql(...)` dans ta configuration du DbContext.

Tu peux me créer le fichier appsettings.json avec des valeurs par defaut ?

On continue avec le projet Client. Je veux que tu me code la page de gestion des clients avec les fichiers razor et razor.cs . Je veux pouvoir créer, modifier ou supprimer des clients. La page razor fait des appels httpclient vers un controller IIS /users qui fait ensuite appel à une couche de services qui fait les accès à EFModel. Tu mets les DTO dans un projet séparé nommé Shared.

Voici la structure recommandée :

Les DTO sont placés dans le projet Shared. La page Blazor (Clients.razor et Clients.razor.cs) utilise ces DTO et fait des appels HTTP vers users. Le code ci-dessous est prêt à l'emploi.

Je voudrais que tu me documentes le code généré sur répertoire SRC. Que peux tu écrire en français dans le texte pour faire la documentation d'architecture technique ?

Voici un exemple de texte pour documenter l'architecture technique du code généré dans le répertoire `SRC` :

---

2. Architecture technique de la solution

La solution eClassRoom est structurée en plusieurs projets distincts dans le dossier `SRC`, chacun ayant un rôle précis dans l'architecture en couches :

2.1. Présentation générale

L'application repose sur une architecture **multi-couches** :

- **Client** : Application Blazor WebAssembly pour l'interface utilisateur.
- **Server** : API REST .NET (Web API) pour l'accès aux données et la logique métier.
- **EFModel** : Modèles de données Entity Framework Core et accès à la base PostgreSQL.
- **Shared** : Objets de transfert de données (DTO) partagés entre le client et le serveur.

2.2. Détail des projets

2.2.1. Client

Le projet `Client` contient l'application Blazor WebAssembly :

- Affichage des pages (login, gestion des clients, etc.).
- Appels HTTP vers l'API REST du serveur via `HttpClient`.
- Utilisation de composants BootstrapBlazor pour l'UI.
- Les pages Razor sont organisées dans le dossier `Pages`.

2.2.2. Server

Le projet `Server` expose des endpoints REST (contrôleurs Web API) :

- Chaque contrôleur reçoit les requêtes du client, les valide et appelle la couche de services.
- La couche de services (dossier `Services`) centralise la logique métier et les accès aux données via Entity Framework.
- La configuration du serveur (connexion PostgreSQL, CORS, Swagger, etc.) est réalisée dans `Program.cs` et `appsettings.json`.

2.2.3. EFModel

Le projet `EFModel` regroupe :

- Les entités du modèle de données (dossier `Models`) : `Client`, `Utilisateur`, `SalleDeFormation`, `MachineVirtuelle`, `Facture`.
- Le contexte de base de données `EClassRoomDbContext` configuré pour PostgreSQL.
- Ce projet est référencé par le serveur pour l'accès aux données.

2.2.4. Shared

Le projet `Shared` contient les **DTO** (Data Transfer Objects) :

- Les DTO servent à échanger des données entre le client et le serveur de façon sécurisée et adaptée.
- Ils évitent d'exposer directement les entités EF côté client.

2.3. Communication et flux de données

- Le **client** envoie des requêtes HTTP (GET, POST, PUT, DELETE) vers les endpoints du **serveur**.
- Le **serveur** traite les requêtes via ses contrôleurs, qui délèguent la logique à la couche de services.
- La couche de services utilise **Entity Framework** pour interagir avec la base PostgreSQL.
- Les données sont échangées sous forme de **DTO** définis dans le projet `Shared`.

2.4. Sécurité et bonnes pratiques

- Les mots de passe doivent être stockés de façon sécurisée (hashage côté serveur).
- Les accès à l'API sont à sécuriser (authentification, autorisation).
- L'utilisation de DTO permet de maîtriser les données exposées au client.

2.5. Extensibilité

- L'architecture permet d'ajouter facilement de nouvelles entités, pages ou services.
- Les projets sont découplés pour faciliter la maintenance et les évolutions.

Ce texte peut être adapté et enrichi selon les besoins spécifiques de ton projet et les choix techniques retenus.

Je voudrais que tu reprennes le code et me génère la documentation dans le folder Docs tel qu'indiquer dans mes instructions.

Documente moi la page Login.razor et son code-behind. Explique moi les dépendances et les intéractions Services et EF. Fais le dans le folder de haut niveau Docs.

Documente moi la page Clients.razor et son code-behind. Explique moi les dépendances et les intéractions Services et EF. Fais le dans le folder de haut niveau Docs. Merci mon ami !

On continue avec le projet Client. Je veux que tu me code la page de gestion des machines virtuelles avec les fichiers razor et razor.cs . Je veux pouvoir créer, modifier ou supprimer des VMs. Utilise le modèle EF pour connaitre les attributs. La page razor fait des appels httpclient vers un controller IIS /users qui fait ensuite appel à une couche de services qui fait les accès à EFModel. Tu mets les DTO dans un projet séparé nommé Shared. N'oublie pas de créer la documentation de page et les dépendances et les intéractions Services et EF. Fais le dans le folder de haut niveau Docs.

On continue avec le projet Client. Je veux que tu me code la page de gestion des salles de formation avec les fichiers razor et razor.cs . Je veux pouvoir créer, modifier ou supprimer des salles. Utilise le modèle EF pour

connaitre les attributs. La page razor fait des appels httpclient vers un controller IIS /users qui fait ensuite appel à une couche de services qui fait les accès à EFModel. Tu mets les DTO dans un projet séparé nommé Shared. TU créé aussi la couche services. N'oublie pas de créer la documentation de page et les dépendances et les intéractions Services et EF. Fais aussi la documentation métier. Faisle dans le folder de haut niveau Docs.

On continue avec le projet Client. Je veux que tu me code la page de gestion des utilisateurs avec les fichiers razor et razor.cs . Je veux pouvoir créer, modifier ou supprimer des utilisateurs. Utilise le modèle EF pour connaitre les attributs. La page razor fait des appels httpclient vers un controller IIS /users qui fait ensuite appel à une couche de services qui fait les accès à EFModel. Tu mets les DTO dans un projet séparé nommé Shared. Tu créé aussi la couche services. N'oublie pas de créer la documentation de page et les dépendances et les intéractions Services et EF. Fais aussi la documentation métier. Faisle dans le folder de haut niveau Docs.

Infrastructure Azure

Chapitre 2 - Azure On passe au service technique Azure. Je veux créer des objets dans Azure : des resource groups, des VMs. Tu saurais faire cela de manière modulaire pour chaque étape et le tout avec le SDK Azure SDK version 2.0. Je ne veut pas utiliser les API Fluent mais la nouvelle version. Tu peux me coder cela ? Créée la documentation Azure dans le Folders 'Docs'.

Tu peux me coder cette partie Azure Infrastructure ?

Ajoute ici d'autres méthodes pour créer VNet, Subnet, IP publique, NIC, etc.

Ja'i besoin de créer un Service Azure Métier de plus haut-niveau. Je veux le code pour pouvoir créer une salle de formation: il faut donc provisionner autant de machines virtuelles qu'il y a de stagiaires. Il faut donc faire les appels aux bonnes fonctions du service AzureInfrastructure et passer les paralètres user name password etc. Tu peux me coder ce service métier ?

La méthode ProvisionnerSalleAsync doit retourner un objet complexe qui contient la SalleDeFormationsDto et pour chaque stagiaire, la liste des adresses ip public ; ainsi on pourra donner aux stagiaires cette information pour qu'ils se connectent. Je veux stocker ces informations via le modèle Entity Framework dans une Table ProvisionningVM. Il faut donc que tu améliores le modèle EF et ajoute dans la couche service ces ajouts en bases de données. Tu saurais me coder cela et me le documenter en technique et en métier dans le folder 'Docs' ?

Tu peux me recréer un diagramme de séquence au format PlantUML à partir de la doc et du code des deux services ?

Documentation de la page SallesDeFormation.razor

Rôle de la page

La page `SallesDeFormation.razor` permet de gérer les salles de formation (affichage, création, modification, suppression) via une interface utilisateur Blazor.

Dépendances principales

- **HttpClient** : Pour les appels HTTP (GET, POST, PUT, DELETE) vers l'API REST `/salles`.

- **SalleDeFormationDto** (projet Shared) : Sert à transporter les données des salles entre le client et le serveur.
- **BootstrapBlazor** : Pour l'UI (tableaux, formulaires).
- **IJSRuntime** : Peut être utilisé pour des interactions avancées côté client.

Fonctionnement et interactions

1. **Chargement des salles**
    - Appel GET `/salles` pour récupérer la liste.
    - Affichage dans un tableau.
2. **Création**
    - Bouton "Nouvelle salle de formation" ouvre le formulaire.
    - POST `/salles` avec le DTO pour créer une salle.
3. **Modification**
    - Bouton "Modifier" ouvre le formulaire prérempli.
    - PUT `/salles/{id}` avec le DTO pour modifier.
4. **Suppression**
    - Bouton "Supprimer" appelle DELETE `/salles/{id}`.

Interactions Services et EF

- **Côté client** :
    - Utilise uniquement les DTO et l'API REST.
- **Côté serveur** :
    - Le contrôleur `/salles` reçoit les requêtes, valide et délègue à la couche de services (`SalleDeFormationService`).
    - La couche de services utilise Entity Framework pour manipuler les entités `SalleDeFormation` dans la base PostgreSQL.
    - Les entités EF ne sont jamais exposées directement : seules les données des DTO transitent.

Architecture côté serveur

- **SallesController** : Contrôleur Web API qui expose les endpoints REST pour la gestion des salles de formation. Il reçoit les requêtes du client, les valide et appelle la couche de service.
- **SalleDeFormationService** : Service métier qui centralise la logique de gestion des salles. Il utilise le `EClassRoomDbContext` (Entity Framework) pour accéder à la base de données et effectuer les opérations CRUD sur les entités `SalleDeFormation`.

Résumé du flux

1. L'utilisateur interagit avec la page (affichage, ajout, édition, suppression).
2. Le code-behind effectue des appels HTTP vers l'API REST.
3. Le serveur traite via le contrôleur et la couche de services, qui utilise EF pour la base de données.

4. Les réponses sont renvoyées sous forme de DTO et affichées côté client.

Références de code

- **SallesDeFormation.razor** : UI et composants.
- **SallesDeFormation.razor.cs** : Logique métier côté client, appels API.
- **SallesController.cs** (Server) : Endpoints REST pour la gestion des salles.
- **Services/SalleDeFormationService.cs** (Server) : Logique métier et accès aux données via EF.
- **EFModel/Models/SalleDeFormation.cs** : Entité EF.
- **Shared/SalleDeFormationDto.cs** : DTO utilisé pour le transport des données.

Documentation métier : Gestion des salles de formation

1. Rôle métier des salles de formation

Dans eClassRoom, une **salle de formation** représente un espace virtuel où se déroule une session de formation. Elle regroupe un formateur, des stagiaires, une période (date de début/fin) et les ressources nécessaires (machines virtuelles, etc.).

Les salles sont créées et gérées par les administrateurs ou formateurs via l'interface web.

2. Attributs métier d'une salle de formation

Les principaux attributs d'une salle sont :

- **Id** : Identifiant unique.
- **Nom** : Nom de la salle ou de la formation.
- **Formateur** : Nom ou identifiant du formateur responsable.
- **DateDebut** : Date de début de la session.
- **DateFin** : Date de fin de la session.

3. DTO (Data Transfer Object)

Le **SalleDeFormationDto** (dans le projet Shared) permet de transporter les données d'une salle entre le client et le serveur sans exposer directement l'entité EF. Il reprend les attributs métier nécessaires à l'UI et aux échanges API.

4. Service métier

Le **SalleDeFormationService** (dans le dossier Services du Server) centralise la logique métier :

- Récupération de la liste des salles.
- Création d'une nouvelle salle à partir d'un DTO.
- Modification et suppression d'une salle existante.
- Utilisation du DbContext EF pour toutes les opérations sur la base.

Ce service garantit la cohérence métier et isole la logique d'accès aux données.

5. Couche Entity Framework (EF)

L'entité **SalleDeFormation** (dans EFModel/Models) correspond à la table des salles de formation en base PostgreSQL.
Le **EClassRoomDbContext** expose un DbSet pour permettre les opérations CRUD via EF Core.

Les relations éventuelles (ex : association à un formateur, à des stagiaires, à des machines virtuelles) sont gérées dans le modèle EF.

6. Flux métier

1. L'utilisateur (admin ou formateur) crée/modifie/supprime une salle via l'UI.
2. Le client Blazor envoie un DTO au serveur via l'API REST.
3. Le contrôleur appelle le service, qui applique la logique métier et utilise EF pour persister les changements.
4. Les données sont échangées sous forme de DTO, jamais d'entités EF.

7. Sécurité et bonnes pratiques

- Les endpoints sont protégés par JWT et `[Authorize]`.
- Seuls les utilisateurs autorisés peuvent gérer les salles.
- Les données sensibles ne sont jamais exposées directement.

Résumé :
La gestion des salles de formation dans eClassRoom repose sur une séparation stricte entre la couche métier (service), la couche d'accès aux données (EF), et la couche de présentation (DTO/UI), garantissant sécurité, évolutivité et clarté du modèle métier.

Spécifications du modèle EF, des API et des Services

1. Entités du modèle EF

Client

- Id : int
- NomSociete : string
- Adresse : string
- CodePostal : string
- Ville : string
- Pays : string
- EmailAdministrateur : string
- Mobile : string
- MotDePasseAdministrateur : string
- Utilisateurs : ICollection
- Salles : ICollection

Utilisateur

- Id : int
- Email : string
- Nom : string
- Prenom : string
- MotDePasse : string
- Role : RoleUtilisateur
- ClientId : int
- Client : Client

RoleUtilisateur (Enum)

- Administrateur
- Formateur
- Stagiaire

SalleDeFormation

- Id : int
- NomFormation : string
- FormateurId : int
- Formateur : Utilisateur
- Stagiaires : ICollection
- DateDebut : DateTime
- DateFin : DateTime
- ClientId : int
- Client : Client
- Machines : ICollection

MachineVirtuelle

- Id : int
- TypeOS : string
- TypeVM : string
- Sku : string
- Offer : string
- Version : string
- DiskISO : string
- NomMarketingVM : string
- FichierRDP : string
- Supervision : string
- StagiaireId : int
- Stagiaire : Utilisateur
- SalleDeFormationId : int
- Salle : SalleDeFormation

Facture

- Id : int

- ClientId : int
- Client : Client
- Mois : DateTime
- Montant : decimal
- Details : string

ProvisionningVM

- Id : int
- SalleDeFormationId : int
- SalleDeFormation : SalleDeFormation
- StagiaireId : int
- Stagiaire : Utilisateur
- VmName : string
- PublicIp : string
- DateProvisionning : DateTime

---

2. Liste des End Points Web API

UsersController

- GET /users : Récupère tous les utilisateurs
- GET /users/{id} : Récupère un utilisateur par son id
- POST /users : Crée un nouvel utilisateur
- PUT /users/{id} : Met à jour un utilisateur
- DELETE /users/{id} : Supprime un utilisateur

SallesController

- GET /salles : Récupère toutes les salles de formation
- GET /salles/{id} : Récupère une salle par son id
- POST /salles : Crée une nouvelle salle
- PUT /salles/{id} : Met à jour une salle
- DELETE /salles/{id} : Supprime une salle

MachinesController

- GET /machines : Récupère toutes les machines virtuelles
- GET /machines/{id} : Récupère une machine par son id
- POST /machines : Crée une nouvelle machine
- PUT /machines/{id} : Met à jour une machine
- DELETE /machines/{id} : Supprime une machine

AuthController

- POST /api/auth/login : Authentifie un utilisateur et retourne un token JWT

---

3. Liste des Services

UtilisateurService

- GetAllAsync() : Liste des utilisateurs
- GetByIdAsync(id) : Détail d'un utilisateur
- AddAsync(dto) : Ajout d'un utilisateur
- UpdateAsync(id, dto) : Mise à jour d'un utilisateur
- DeleteAsync(id) : Suppression d'un utilisateur

SalleDeFormationService

- GetAllAsync() : Liste des salles de formation
- GetByIdAsync(id) : Détail d'une salle
- AddAsync(dto) : Ajout d'une salle
- UpdateAsync(id, dto) : Mise à jour d'une salle
- DeleteAsync(id) : Suppression d'une salle

MachineVirtuelleService

- GetAllAsync() : Liste des machines virtuelles
- GetByIdAsync(id) : Détail d'une machine
- AddAsync(dto) : Ajout d'une machine
- UpdateAsync(id, dto) : Mise à jour d'une machine
- DeleteAsync(id) : Suppression d'une machine

AuthService

- ValidateUserAsync(username, password) : Valide un utilisateur
- GenerateJwtToken(user) : Génère un token JWT

Sommaire Documentation eClassRoom

Introduction & Architecture

Fonctionnalités Métier

Services & Techniques

- ClientsService
- Gestion JWT
- Login
- Azure Infrastructure
- Azure Salle de Formation - Métier
- Azure Salle de Formation - Technique
- Provisionner Salle de Formation

Spécifications & Annexes

- Spécifications
- dotnet CLI création
- MyProject
- MyProject (PDF)
- Project (PDF)
- VSCode Chat Copilot For GitHub

Ce sommaire référence les principaux fichiers de documentation présents dans le dossier `Docs`.

Synthèse Architecture eClassRoom

Vue d'ensemble

L'application eClassRoom repose sur une architecture **multi-couches** moderne, sécurisée et évolutive, adaptée à la gestion de salles de formation virtuelles.

Couches principales

- **Client** : Application web Blazor WebAssembly (SPA) pour l'interface utilisateur.
    - Affichage, saisie, interactions utilisateurs.
    - Appels HTTP sécurisés vers l'API via JWT.
- **Server** : API REST .NET (Web API) pour la logique métier et l'accès aux données.
    - Contrôleurs REST sécurisés par JWT.
    - Gestion des droits, logique métier, orchestration des services.
- **EFModel** : Couche d'accès aux données basée sur Entity Framework Core.
    - Modèles de données (Clients, Utilisateurs, Salles, VMs, Factures…).
    - Connexion à la base PostgreSQL.
- **Shared** : Objets de transfert de données (DTO) partagés entre client et serveur.
    - Garantit la sécurité et la cohérence des échanges.

Sécurité

- Authentification basée sur des tokens JWT.
- Les endpoints sensibles sont protégés par `[Authorize]`.
- Les mots de passe sont stockés hashés côté serveur.
- CORS configuré pour permettre les échanges entre le client et l'API.

Points forts

- **Séparation claire des responsabilités** : chaque couche a un rôle précis.
- **Extensible** : ajout de nouvelles fonctionnalités ou entités facilité.
- **Sécurisé** : gestion des droits, authentification forte, données sensibles protégées.
- **Interopérable** : API REST standard, compatible avec d'autres clients ou outils.

Schéma de fonctionnement

1. L'utilisateur se connecte via le portail web (Blazor).
2. Le client envoie ses requêtes à l'API REST, en incluant le token JWT.
3. Le serveur valide, traite la demande, et interagit avec la base de données via EF Core.
4. Les données sont échangées sous forme de DTO, jamais d'entités directes.
5. Les réponses sont renvoyées au client pour affichage ou action.

En résumé :
eClassRoom est une solution web robuste, sécurisée et moderne, pensée pour la gestion efficace de salles de formation virtuelles, avec une architecture claire et évolutive.

Documentation de la page Utilisateurs.razor

Rôle de la page

La page `Utilisateurs.razor` permet de gérer les utilisateurs (affichage, création, modification, suppression) via une interface utilisateur Blazor.

Dépendances principales

- **HttpClient** : Pour les appels HTTP (GET, POST, PUT, DELETE) vers l'API REST `/users` et `/clients`.
- **UtilisateurDto** (projet Shared) : Sert à transporter les données des utilisateurs entre le client et le serveur, incluant l'identifiant de la société (ClientId).
- **ClientDto** (projet Shared) : Sert à afficher la liste des sociétés dans la combo box.
- **BootstrapBlazor** : Pour l'UI (tableaux, formulaires, combo box).
- **IJSRuntime** : Peut être utilisé pour des interactions avancées côté client.

Fonctionnement et interactions

1. **Chargement des utilisateurs et des sociétés**
    - Appel GET `/clients` pour récupérer la liste des sociétés.
    - Appel GET `/users` pour récupérer la liste des utilisateurs.

    - Affichage dans un tableau.

2. **Création et modification**
    - Bouton "Nouvel utilisateur" ou "Modifier" ouvre le formulaire.
    - Le formulaire contient une combo box pour sélectionner la société (ClientId).
    - POST `/users` ou PUT `/users/{id}` avec le DTO incluant le ClientId.

3. **Suppression**
    - Bouton "Supprimer" appelle DELETE `/users/{id}`.

Interactions Services et EF

- **Côté client** :
    - Utilise uniquement les DTO et l'API REST.
    - La combo box est alimentée par la liste des clients récupérée via `/clients`.
- **Côté serveur** :
    - Le contrôleur `/users` reçoit les requêtes, valide et délègue à la couche de services (`UtilisateurService`).
    - La couche de services utilise Entity Framework pour manipuler les entités `Utilisateur` dans la base PostgreSQL, en tenant compte de la relation avec `Client`.
    - Les entités EF ne sont jamais exposées directement : seules les données des DTO transitent.

Architecture côté serveur

- **UsersController** : Contrôleur Web API qui expose les endpoints REST pour la gestion des utilisateurs. Il reçoit les requêtes du client, les valide et appelle la couche de service.
- **UtilisateurService** : Service métier qui centralise la logique de gestion des utilisateurs. Il utilise le `EClassRoomDbContext` (Entity Framework) pour accéder à la base de données et effectuer les opérations CRUD sur les entités `Utilisateur`, en gérant la relation avec la société.

Résumé du flux

1. L'utilisateur interagit avec la page (affichage, ajout, édition, suppression).
2. Le code-behind effectue des appels HTTP vers l'API REST.
3. Le serveur traite via le contrôleur et la couche de services, qui utilise EF pour la base de données.
4. Les réponses sont renvoyées sous forme de DTO et affichées côté client.

Particularité

- Lors de la création ou modification d'un utilisateur, une combo box permet de sélectionner la société à laquelle il appartient (ClientId).

Références de code

- **Utilisateurs.razor** : UI et composants, combo box pour la société.
- **Utilisateurs.razor.cs** : Logique métier côté client, appels API, gestion de la liste des sociétés.
- **UsersController.cs** (Server) : Endpoints REST pour la gestion des utilisateurs.
- **Services/UtilisateurService.cs** (Server) : Logique métier et accès aux données via EF.

- **EFModel/Models/Utilisateur.cs** : Entité EF (avec clé étrangère vers Client).
- **Shared/UtilisateurDto.cs** : DTO utilisé pour le transport des données, incluant ClientId.
- **Shared/ClientDto.cs** : DTO pour la liste des sociétés.

Documentation métier : Gestion des utilisateurs

1. Rôle métier des utilisateurs

Dans eClassRoom, un **utilisateur** représente toute personne ayant accès à la plateforme : administrateur, formateur ou stagiaire. Chaque utilisateur possède un profil, un rôle et des accès adaptés à ses fonctions.

Les utilisateurs sont créés et gérés par les administrateurs via l'interface web.

2. Attributs métier d'un utilisateur

Les principaux attributs d'un utilisateur sont :

- **Id** : Identifiant unique.
- **Email** : Adresse email (identifiant de connexion).
- **Nom** : Nom de famille.
- **Prénom** : Prénom.
- **MotDePasse** : Mot de passe (stocké hashé côté serveur).
- **Role** : Rôle de l'utilisateur (admin, formateur, stagiaire).

3. DTO (Data Transfer Object)

Le **UtilisateurDto** (dans le projet Shared) permet de transporter les données d'un utilisateur entre le client et le serveur sans exposer directement l'entité EF. Il reprend les attributs métier nécessaires à l'UI et aux échanges API.

4. Service métier

Le **UtilisateurService** (dans le dossier Services du Server) centralise la logique métier :

- Récupération de la liste des utilisateurs.
- Création d'un nouvel utilisateur à partir d'un DTO.
- Modification et suppression d'un utilisateur existant.
- Utilisation du DbContext EF pour toutes les opérations sur la base.

Ce service garantit la cohérence métier et isole la logique d'accès aux données.

5. Couche Entity Framework (EF)

L'entité **Utilisateur** (dans EFModel/Models) correspond à la table des utilisateurs en base PostgreSQL.
Le **EClassRoomDbContext** expose un DbSet pour permettre les opérations CRUD via EF Core.

Les relations éventuelles (ex : association à un client, à une salle, à des machines virtuelles) sont gérées dans le modèle EF.

6. Flux métier

1. L'administrateur crée/modifie/supprime un utilisateur via l'UI.
2. Le client Blazor envoie un DTO au serveur via l'API REST.
3. Le contrôleur appelle le service, qui applique la logique métier et utilise EF pour persister les changements.
4. Les données sont échangées sous forme de DTO, jamais d'entités EF.

7. Sécurité et bonnes pratiques

- Les endpoints sont protégés par JWT et `[Authorize]`.
- Seuls les administrateurs peuvent gérer les utilisateurs.
- Les mots de passe sont stockés de façon sécurisée (hashés).
- Les données sensibles ne sont jamais exposées directement.

Résumé :
La gestion des utilisateurs dans eClassRoom repose sur une séparation stricte entre la couche métier (service), la couche d'accès aux données (EF), et la couche de présentation (DTO/UI), garantissant sécurité, évolutivité et clarté du modèle métier. □