MS861 – L'accès aux données en C# sous Visual Studio – Livret Ateliers

Module 1 – Architecture des applications liées aux données

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- choisir une architecture d'accès aux données ;
- choisir une technologie d'accès aux données ;

Exercice 1
Etude de cas

Dans cet exercice, vous allez devoir choisir quelle solution peut être envisagée (architecture et/ou technologie) en fonction des informations qui vous seront fournies.

- **Scénario d'une application de gestion de clients**

Votre entreprise souhaite concevoir une nouvelle application de gestion clientèle de type WPF qui attaquera la base de données clients. Cette base est ancienne et sa structure est mise à jour environ deux fois par an. Même si les clients ont un droit de regard sur les informations qui les concernent, il n'est nullement envisageable d'exposer ces données en dehors de l'entreprise. Par contre, il n'est pas impossible que de futures applications puissent avoir besoin d'accéder également à ces informations.

- **Scénario d'une application de gestion de commandes**

Votre entreprise souhaite offrir à ses vendeurs une solution de gestion de commandes plus simple que celle actuellement employée. En effet, il est aujourd'hui nécessaire de connecter le portable des vendeurs au réseau de l'entreprise à l'aide d'un VPN, de sorte à ce qu'ils puissent utiliser l'application qui leur a été fourni. Lorsque les vendeurs ne peuvent se connecter à Internet, ils doivent alors gérer leurs commandes à l'aide de classeurs Excel que le Service Informatique doit ensuite intégrer à la base.

- **Scénario d'une application de gestion de livraisons**

Les sociétés de livraison avec lesquelles travaille votre entreprise, ont décidé d'utiliser un nouveau format d'échange de données standardisé (type JSON). Jusqu'à présent, ils utilisaient un portail Web dédié pour mettre à jour l'état des livraisons dont ils avaient la responsabilité. Il vous est donc demandé de trouver une solution pour intégrer simplement et efficacement ces données qui proviendront de différentes applications externes.

Module 2 – ADO.NET

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- mettre en œuvre une approche connectée à l'aide d'ADO.NET ;
- mettre en œuvre une approche déconnectée à l'aide d'ADO.NET ;

Exercice 1
Approche connectée

Dans le cadre des futurs développements dans votre entreprise, il vous a été demandé de créer une première application qui sera utilisée par vos collègues développeurs. En effet, les administrateurs système et base de données ne souhaitent pas que vous disposiez de SQL Server Management Studio sur vos postes. Il vous faut donc concevoir une petite application console permettant de lister l'ensemble des tables de la base de travail, de récupérer la structure d'une table en particulier ou encore de pouvoir directement soumettre la requête de son choix.

Tâche 1

L'application sera une simple application Console nommée **QuickSQL** et proposant 4 choix : Liste des Tables, Structure d'une table, Requête et Quitter. Il doit être possible de changer le nom du serveur de base de données sans toutefois devoir recompiler l'application. Pour cela, vous avez décidé d'utiliser le fichier de configuration et plus précisément la section **AppSettings** (le reste des informations de la chaîne de connexion seront fixes, à savoir le nom de la base **TSQL2012** et le mode d'authentification qui sera le mode intégré).

1. Créer le projet application Console

2. Ajouter une référence à **System.Configuration**, ce qui vous permettra d'utiliser la classe **ConfigurationManager** et plus précisément son indexeur **AppSettings**

3. Ajouter la clé **ServerName** à la section **AppSettings** du fichier de configuration, avec pour valeur .

4. Ajouter le code nécessaire au **static void Main** pour stocker la chaîne de connexion complète dans une variable statique de la classe **Program** que vous appellerez **connectionString**

5. Ajouter le code nécessaire à la mise en place du menu de l'application. Votre code devrait ressembler à ceci :

```
            int choix=0;
            do
            {
                DisplayMenu();
                choix = GetValeur("Veuillez faire votre choix:");
                switch (choix)
                {
                    // Demande de la liste des tables
                    case 1:
```

```
                        GetListTable();
                        break;
                    // Demande de la structure d'une table
                    case 2:
                        GetStructTable();
                        break;
                    // Exécution d'une requête personnalisée
                    case 3:
                        ExecQuery();
                        break;
                    default:
                        if(choix!=0)
                            Console.WriteLine("Valeur incorrecte");
                        break;
                }
            } while (choix != 4);
            Console.WriteLine("Fermeture du programme");
```

```
private static void DisplayMenu()
        {
            Console.WriteLine("Menu:");
            Console.WriteLine("1. Liste des tables");
            Console.WriteLine("2. Structure d'une table");
            Console.WriteLine("3. Requête");
            Console.WriteLine("4. Quitter");
        }
```

Tâche 2

Afin de connaître la liste des tables disponibles dans la base, il est possible de procéder de plusieurs façons. Par habitude et afin de faciliter les potentiels changements de moteur de base de données, vous allez utiliser une requête s'appuyant sur les vues d'**INFORMATION_SCHEMA** de la norme SQL.

1. Dans la méthode **GetListTable**, définissez et ouvrez un objet connexion de type **SqlConnection**

2. Définissez un objet **SqlCommand** à partir de la connexion précédente et avec pour requête :

```
select TABLE_SCHEMA + '.' + TABLE_NAME from INFORMATION_SCHEMA.TABLES
where TABLE_TYPE='BASE TABLE'
order by TABLE_SCHEMA,TABLE_NAME
```

3. A l'aide d'un objet **SqlDataReader**, parcourrez le résultat et afficher chaque valeur sur une ligne séparée

Tâche 3

Afin de connaître la structure d'une table, il vous faudra demander dans un premier temps le nom de la table à l'utilisateur. Ensuite, vous émettrez une requête de type **SELECT TOP 1 * from Table**, afin de récupérer par l'objet **SqlDataReader** les informations nécessaires. Vous mettrez également en place une gestion d'erreur globale affichant directement le message d'exception. Notez que vous pouvez affichez ce message dans cette application car elle est à destination d'un public qui sera à même d'interpréter ces erreurs.

1. Dans la méthode **GetStructTable**, définissez et ouvrez un objet connexion de type **SqlConnection**

2. Définissez un objet **SqlCommand** à partir de la connexion précédente et avec la requête indiquée

3. A l'aide des méthodes **GetName**, **GetFieldType** et **GetDataTypeName** et de la propriété **FieldCount** de la classe **SqlDataReader**, afficher la structure de la table sous la forme :

NOM_CHAMP [Type_Dot_Net <-> Type_SQL]

4. Ajouter une gestion d'erreur de type **try/catch** pour afficher toute exception à l'écran

Tâche 4

La dernière option proposée est potentiellement dangereuse car si rien n'est prévu, l'utilisateur pourrait endommager la base. Toutefois, dans notre cas, l'application sera utilisée uniquement en interne par des développeurs, et logiquement, uniquement sur un serveur de test et/ou développement. De plus, en utilisant l'authentification intégrée de SQL Server, l'administrateur de base de données peut s'assurer de ne pas donner trop de droits aux personnes concernées et également de mettre en œuvre un audit pour savoir qui fait quoi. Le piège dans ce type de scénario serait d'utiliser un seul compte pour tous les utilisateurs, et de plus, un compte ayant tous les privilèges administratifs sur la base.

1. Dupliquer le code de la méthode **GetStructTable** dans la méthode **ExecQuery** et adapter le code actuel pour demander une requête et non uniquement le nom d'une table

2. Dans le bloc **try**, adaptez le code pour affichez dans un premier temps le nom des colonnes de la requête (séparées par une tabulation, à savoir \t en C#)

3. Ajoutez ensuite le code de parcours des données pour afficher chaque valeur de la ligne séparée par une tabulation

Exercice 2
Approche déconnectée

Un certain nombre de développement dans votre entreprise sont basés sur des fichiers plats pour le stockage de données (solution historique). Il vous est demandé de réaliser une première migration en guise de test sur une petite application Console de gestion des employés. Si cette migration s'avère concluante, d'autres pourront être envisagées.

Il va donc vous falloir remplacer une logique fichier par une logique basée sur les **DataSet** (nous conserverons ici une approche déconnectée car il est assuré qu'aucun conflit ne peut survenir). Une table de test vous a été mise à disposition dans la base **TSQL2012** ainsi que le code source de l'application initiale.

Tâche 1

1. Ouvrez le projet **GestionRH** à partir du fichier **GestionRH.sln** dans le dossier E:\Ateliers\Mod02\Starter\GestionRH

2. Prenez le temps d'analyser comment fonctionne l'application :

i. Chargement des données

ii. Gestion de l'entreprise (listing d'employé, embauche, calcul de charge…)

iii. Gestion d'un employé (augmentation, affectation…)

Vous noterez que ce programme ne gère pas la sauvegarde des données. Un module supplémentaire avait été envisagé pour générer un classeur Excel mais le projet a été arrêté avant sa réalisation.

Tâche 2

Dans un premier temps, vous allez devoir créer un **DataSet** typé en fonction de la table de base de données **HR.TestEmp** afin de l'utiliser à la place de la classe Entreprise. Vous obtiendrez une structure fortement typée pour décrire les employés qui se substituera à la classe existante. De ce fait, vous allez pouvoir supprimer la référence à la bibliothèque Info. Pour simplifier le code, vous noterez que le salaire est de type entier, et sera accessible en lecture/écriture avec la solution mise en œuvre (alors qu'il était en lecture seul auparavant).

1. Ajouter au projet un nouvel élément de type **DataSet**, nommé **DsEmp**, et à l'aide de l'explorateur de serveur, créer un nouveau **DataTable** nommé **Employes**, avec un **TableAdapter** nommé **EmployesTableAdapter** (par un glisser-déplacer de la table **TestEmp** depuis l'explorateur de Serveur, puis en renommant le **DataTable** créé)

2. Supprimer la référence à la bibliothèque **Info**. Les erreurs de compilation vous permettront de savoir où corriger le code.

3. Dans le **static void Main**, localisez le bloc **using** qui instancie la variable ms de type **Entreprise** et l'appel à la méthode **chargeData**. Remplacez ce code par l'instanciation du **DataSet** et son chargement à l'aide du **TableAdapter**.

a. Instancier la classe **DataSet**

b. Instancier le **TableAdapter**

c. Invoquer la méthode **Fill** du **TableAdapter** en lui passant le **DataTable** du **DataSet** en paramètre.

4. Corriger le code pour qu'il fonctionne avec l'instance de **DataSet** en lieu et place de la classe **Entreprise**. Notez la présence des classes **EmployesDataTable** et **EmployesRow**, définies en tant que classe partielles dans le fichier **.Designer.cs** adjoint au fichier .xsd du **DataSet**. Il est donc possible de recréer certains des membres des classes initiales, soit en ajoutant une classe statique pour définir des méthodes d'extension à ces types, soit en ajoutant un nouveau fichier de classe partielle.

Dans la classe **Entreprise** initiale, une collection **Employes** permettait de gérer la liste d'employés. Nous allons la remplacer par le **DataTable** typé que nous avons créé dans le **DataSet**. Ensuite, il faut faire en sorte de remplacer toutes les références à la classe **Employe**, par la nouvelle classe **DataRow** qui aura été mise en œuvre dans le **DataSet**.

5. A la fin du **static void Main**, demandez à l'utilisateur s'il souhaite enregistrer ses modifications et s'il répond par l'affirmative, sauvegardez les données en base (pour rappel, nous n'envisagerons pas de problèmes de conflit).

Module 3 – Entity Framework

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer un modèle Code First Entity Framework ;
- appliquer des attributs ;
- définir un initialiseur de base de données
- tester un modèle Entity Framework dans un site ASP.NET MVC

Exercice 1
Création d'un modèle Code First Entity Framework

Compte-tenu des problèmes rencontrés sur le test d'utilisation d'ADO.NET, il a été décidé de refaire l'application de gestion à l'aide d'Entity Framework. Pour cela, il vous est demandé de créer le modèle objet POCO nécessaire à la gestion des employés et des services, en partant du principe qu'une nouvelle base va être créée.

1. Créez un nouveau projet de type Bibliothèque de classe que vous appellerez **DAL_Library**.

2. Ajoutez les deux classes permettant de décrire employés et services selon les caractéristiques suivantes :

Employe : EmployeId, Nom, Prenom, Age, Ville, Salaire, Fonction, ServiceId plus une propriété de type Service nommée Departement

Service : ServiceId, Denomination, Description

Les champs seront soit de type **String**, soit de type **int** pour plus de facilité.

3. Installez Entity Framework dans votre projet via le gestionnaire de package **NuGet** (soit graphiquement, soit par la console de gestion de package et la commande **Install-Package**)

4. Ajoutez la classe de contexte EmployeContext avec ses deux propriétés de type **DbSet<T>**

Exercice 2
Personnalisation du modèle

Afin d'assurer la validité des données en base, il vous est également demandé d'ajouter un certain nombre d'attribut de validation à votre modèle. De plus, afin de pouvoir tester votre modèle, vous allez ajouter une classe d'initialiseur qui permettra de créer la base de données si elle n'existe pas, à l'aide de données de test.

1. En utilisant l'espace de nom **System.ComponentModel.DataAnnotations**, rendez obligatoire la dénomination d'un service, nom, prénom, fonction et ServiceId d'un employé

2. Mettez en œuvre une nouvelle classe **SalaireValid**, héritante de **ValidationAttribute**

3. Ajoutez à cette classe un constructeur vide initialisant la propriété **ErrorMessage** avec une chaîne du type « Le salaire doit être compris entre 0 et 100000 »

4. Mettez en œuvre la méthode polymorphe **IsValid(object value)** afin de tester si la valeur passée en paramètre est un entier entre 0 et 100000

5. Utilisez votre attribut personnalisé sur le champ **Salaire** de la classe **Employe**

6. Dans le fichier de votre classe de contexte, ajoutez une nouvelle classe ayant pour déclaration :

Internal class EmployeInitializer : CreateDatabaseIfNotExists<EmployeContext>

7. Mettez en œuvre la méthode polymorphe **Seed** et copiez-y le contenu du fichier E:\Ateliers\Mod03\Seed.txt

8. Ajoutez un constructeur à votre contexte avec le code suivant :

Database.SetInitializer<EmployeContext>(new EmployeInitializer()) ;

Exercice 3
Test du modèle à l'aide d'ASP.NET MVC

De façon à tester votre modèle et l'initialiseur de base de données, vous allez mettre en œuvre un site Web ASP.NET MVC. Dans cette approche, le développement est décomposé en trois parties :

- Le Modèle : dans notre cas, notre bibliothèque d'accès aux données. Le modèle joue le rôle d'intermédiaire entre le site et la base de données
- Le Contrôleur : c'est l'orchestrateur du site, à savoir que toute navigation à travers le site redirige vers une action d'un contrôleur qui se charge alors d'appeler le modèle et de retourner les données à la vue
- La Vue : il s'agit tout simplement la partie afffichage du site.

Comme vous allez le constater, il est très facile de créer un site Web ASP.NET MVC, en particulier grâce aux différents assistants de Visual Studio.

1. Ajoutez un nouveau projet à votre solution existante, en choisissant un projet type Application Web ASP.NET que vous nommerez **SiteTest**. Dans la page d'option qui s'ouvrira, choisissez un modèle Vide (Empty) et assurez-vous de cocher l'ajout de dossier et références MVC et inversement de décocher l'hébergement dans le cloud, ainsi que l'ajout de tests unitaires

2. Ajoutez votre chaîne de connexion au fichier **Web.config** de ce site, juste après la section **appSettings**, en indiquant comme nom de base **GestionRH**.

3. Ajoutez au site Web une référence vers votre projet **DAL_Library**

4. Ajoutez un contrôleur et les vues associées pour chaque classe de votre modèle (**Employe** et **Service**) :

- Clic-droit sur le dossier Controllers

- Ajouter -> Contrôleur…
- Choisir Contrôleur MVC 5 avec vues, utilisant Entity Framework
- Spécifier votre classe de contexte puis la classe de modèle, et décocher l'utilisation de page de disposition

Note : Si vous recevez un message d'erreur « La référence d'objet n'est pas définie à une instance d'un objet », assurez-vous de recompiler votre solution avant de réessayer.

5. En effectuant un clic-droit sur le projet **SiteTest**, lancez une nouvelle instance du site en mode de débogage (menu Déboguer puis Démarrer une nouvelle instance)

Note : Vous devriez voir apparaître le navigateur par défaut de la machine, et surtout un message d'erreur « La ressource est introuvable ». C'est normal puisque notre site n'a pas de page d'accueil de définie.

6. Dans la barre d'adresse de votre navigateur, ajoutez **Employes** ou **Services** et assurez-vous que les données soient accessibles

Note : Visual Studio a normalement généré pour vous deux classes **EmployesController** et **ServicesController** et ce sont eux que l'on invoque par l'adresse http://localhost:xxx/Employes (sans le mot Controller)

7. En accédant au mode édition sur un employé, assurez-vous que le salaire est bien obligatoire, et doit être compris entre les valeurs spécifiées par votre validateur

Module 4 – LINQ

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- utiliser un contexte Entity Framework pour lire des données;
- manipuler la syntaxe LINQ;

Exercice 1
Utilisation d'Entity Framework

Une nouvelle application de gestion des ressources humaines a été créée. Il s'agit d'une application type WinForms car l'équipe en place ne maîtrise pas encore WPF et l'approche MVVM.
L'application se présente sous deux modes :

- Le mode Service, qui permet d'afficher les détails d'un service et la liste des employés correspondants
- Le mode Employé, qui permet d'afficher la liste complète des employés

Dans le premier cas, une navigation simple sera proposée pour aller de service en service.

Dans le deuxième cas, des fonctions de filtrage seront proposées (par fonction ou par nom).

Dans les deux cas, il sera alors possible d'obtenir des le montant de la charge salariale : totale, pour un service donnée, pour une fonction donnée ou pour la liste des employés sélectionnés.

Il vous a été demandé de mettre en place Entity Framework dans le cadre de ce développement en réutilisant la bibliothèque créée précédement.

Rappel : afin d'afficher une nouvelle fenêtre, déclarée dans un fichier frmExemple.cs, il faut tout d'abord l'instancier puis invoquer la méthode ShowDialog. Celle-ci retourne habituellement le choix fait par l'utilisateur (bouton OK ou Annuler) par le biais d'une variable de type DialogResult. Ce qui donne un code du type :

```
frmExemple frm = new frmExemple();
if (frm.ShowDialog() == DialogResult.OK)
{ …
```

Avant toute chose, ouvrez la solution du dossier **E:\Ateliers\Mod04\Starter\GestionRH**.

1. Déclarez une nouvelle instance nommée context de la classe **EmployeContext** de sorte à ce qu'elle soit accessible dans l'ensemble du formulaire **Form1**

2. Dans la région **Navigation**, repérez la méthode **Form1_Load** (qui est associé à l'événement de chargement du formulaire) et utilisez une requêt **LINQ** pour charger la liste des services dans la variable services de type **List<Service>**, en faisant en sorte de charger également dans le contexte l'ensemble des employés.

3. Repérez la méthode **DisplayData** qui permet d'assurer le rafraichissement de l'affichage et affectez à la propriété **DataSource** de l'objet **dgvEmployes** :

- La liste des employés du service sélectionné en mode Service
- La liste des employés du contexte en mode Employé

4. Assurez-vous du bon fonctionnement du programme (les calculs de charge salariale et filtrage d'employés seront mis en œuvre dans l'exercice suivant)

Exercice 2
Manipulation de requêtes LINQ

A l'aide de LINQ, soit sous forme d'opérateurs de requête ou de méthodes d'extension (plusieurs approches sont possibles), vous allez devoir calculer les différentes charges salariales demandées. Ensuite, vous devrez compléter le code nécessaire au filtrage des employés.

1. Calcul de la charge salariale totale : méthode **mnuTotal_Click** de la région **Charge Salariale**

2. Calcul de la charge salariale pour le service affiché : méthode **mnuService_Click** de la région **Charge Salariale**

3. Calcul de la charge salariale pour les employés affichés : méthode **mnuVisible_Click** de la région **Charge Salariale**

Note : la grille de données est un contrôle nommé **dgvEmployes** ; les employés affichés sont définis par la propriété **DataSource** de ce contrôle. Il nous suffit donc de récupérer cette propriété et de la convertir en tant que **List<Employe>** pour pouvoir faire la somme des salaires demandée.

4. Calcul de la charge salariale pour une fonction choisie : méthode **mnuFonction_Click** de la région **Charge Salariale**

Note : Il vous faut pour cela affecter la liste des fonctions à la propriété **Fonctions** de l'objet **frm** qui représente la fenêtre à travers laquelle l'utilisateur choisira la fonction (et qui s'affichera par la méthode **ShowDialog**). Une fois que l'utilisateur a cliqué OK dans cette fenêtre (test sur la valeur **DialogResult.OK**), vous retrouverez par la propriété **Choix** de l'objet **frm**, le nom de la fonction choisie.

5. Filtrage par nom de fonction : méthode **mnuEmpFonc_Click** de la région **Filtrage**

Note : Le code est relativement similaire à celui de la méthode précédente, à l'exception du fait que sera ajouté la valeur [Tous] à la liste des fonctions, de sorte à pouvoir afficher tous les employés quelque soit leur fonction. Cette fois, au lieu de calculer le montant de la charge salariale, vous allez devoir définir le **DataSource** de la grille de données en fonction du choix de l'utilisateur. Comme précédement, c'est la propriété **Choix** de l'objet **frm** qui vous indiquera la fonction choisie (ou [Tous]).

6. Filtrage par nom d'employé : méthode **mnuEmpNom_Click** de la région **Filtrage**

Note : Ici la fenêtre présentée à l'utilisateur est constituée d'une zone de texte (dont la valeur sera récupérée par la propriété **Recherche**) et deux boutons radios permettanr d'effectuer une recherche de type 'Commence par' ou 'Contient' sur le nom des employés. Vous aurez accès à une propriété **Choix** sur l'objet **frm** qui vous indiquera le souhait de l'utilisateur (soit **RechercheNom.CommencePar** soit **RechercheNom.Contient**). Il vous est également demandé d'afficher la liste de tous les employés si la valeur de la propriété **Recherche** est vide.

Note : une erreur s'est glissée dans l'interface et la fenêtre a pour intitulé *Fonction*. Il faut bien évidement comprendre *Nom de l'employé*

Module 5 – Mise à jour des données à travers Entity Framework

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- modifier les données d'un objet de contexte;
- enregistrer les données en base;

Compte-tenu de la nature des données manipulées par l'application, les mises à jours sur les entités existantes sont temporaires jusqu'à enregistrement par l'utilisateur. Par contre, toute insertion doit être effectuée directement en base (en particulier pour assurer de récupérer les identifiants uniques nécessaires). Un système de gestion de commande déconnecté (commercial en déplacement) ne pourrait pas envisager cette approche mais pourrait utiliser soit un stockage local en charge de la génération des valeurs de clé, soit un mécanisme de clé géré par l'application.

Lors de la mise en œuvre du modèle Entity Framework, vous avez utilisez des attributs pour la validation des données. Or seul le modèle ASP.NET MVC permet d'en tirer partie directement et simplement. C'est pour cette raison que notre application gérera directement les règles de validation dans son interface.

Exercice 1
Modification et ajout d'entité

Vous devez à présent modifier l'application de gestion pour intégrer la mise à jour des données. Pour commencer, vous allez gérer la mise à jour et l'ajout de services mais également gérer les problématiques de mise à jour à travers l'interface (centralisation des sauvegardes, gestion de la fermeture…).

Pour rappel, l'application gère un contexte Entity Framework contenant toutes les données, mais également une liste de service qui a facilité la mise en place de la navigation. Il faut bien comprendre que le contenu de cette liste est une copie des données du contexte, que l'on pourrait considérer comme des objets détachés. Il ne faut donc pas effectuer de modification sur ces objets, mais bel et bien dans le contexte.

Afin de gérer l'insertion d'un nouveau service, il vous faudra dans un premier temps vous assurez que toutes les données ont bien été enregistrées en base afin d'éliminer des problèmes potentiels. Concrétement, l'insertion se passera en plusieurs étape :

- demande d'insertion de l'utilisateur
- vérification de l'état du contexte et si besoin annulation de la demande d'insertion
- activation de l'interface pour permettre la saisie des nouvelles données de service
- demande d'enregistrement du service par l'utilisateur
- validation des données en base et mise à jour des éléments nécessaires à l'interface graphique.

Avant toute chose, ouvrez le projet du dossier **E:\Ateliers\Mod05\Starter\GestionRH**.

1. Dans la région **Modification Service**, localisez la méthode **btnSave_Click**.

2. Entre les deux régions **Validation** et **Affichage**, ajoutez le code nécessaire pour appliquer le contenu des deux zones de texte **txtDenomination** et **txtDescription** aux propriétés correspondantes de l'objet **Service** du contexte actuellement affiché.

3. Dans la région **Gestion Mise à jour/Interface**, localisez la méthode **mnuSave_Click**.

4. A l'aide d'un bloc **try/catch**, mettez en œuvre la sauvegarde des données du contexte. En cas d'exception, vous utiliserez la méthode **MessageBox.Show** pour afficher le message d'erreur et invoquerez la fonction **ResolveError** fournie.

Note : la méthode **ResolveError** permet de lister les potentielles erreurs et de laisser le choix à l'utilisateur de soit les corriger lui-même (clic Ok) soit de laisser le programme recharger les données depuis la base (clic Cancel). Il faut noter que dans ce dernier cas, le code présenté n'est pas le plus efficace (il aurait suffit de recréer le contexte) et de plus, il ne devrait pas y avoir ici d'erreurs. Le but de cette méthode est avant tout pédagogique.

5. Invoquez la méthode **DisplayData** en fin de méthode afin d'assurer le rafraichissement de l'interface utilisateur.

6. Localisez la méthode **Form1_Closing** dans la même région que précédemment. Il s'agit du code invoqué par le runtime en cas de demande de fermeture de la fenêtre (soit par la méthode **this.Close()** soit par la croix d'interface graphique).

7. Ajoutez le code nécessaire pour demander à l'utilisateur s'il souhaite enregistrer ses modifications pour le cas où il ne l'ait pas fait avant.

> a. Ajoutez un test pour vérifier la présence de modification non enregistrées dans le contexte. Utilisez pour cela la collection **context.ChangeTracker.Entries()** qui contient toutes les entités du contexte sous forme d'objet **Entry** afin de maintenir leur état (propriété **State**).
>
> Nous utiliserons pour cela la méthode d'extension **Any** afin de connaître l'état des entités :
>
> context.ChangeTracker.Entries().Any(en => en.State == System.Data.Entity.EntityState.Added || en.State == System.Data.Entity.EntityState.Modified || en.State == System.Data.Entity.EntityState.Deleted)
>
> b. En cas de modifications non enregistrées, demandez à l'utilisateur s'il souhaite les enregistrer (Oui), les abandonner (Non) ou bien annuler la fermeture de l'application (Annuler). Vous utiliserez pour cela la méthode **MessageBox.Show**, en utilisant plus particliuèrement le paramètre **MessageBoxButtons.YesNoCancel**. La valeur de retour de la méthode vous aidera alors à connaître le choix de l'utilisateur (**DialogResult.Yes**, **DialogResult.No** ou **DialogResult.Cancel**)
>
> Si l'utilisateur veut enregistrer ses données, utilisez un bloc **try/catch** pour tenter la sauvegarde. En cas d'erreurs, un message sera retourné à l'utilisateur pour lui indiquer que ses modifications seront perdues.

Si l'utilisateur ne veut pas enregistrer, ne rien faire.

Si l'utilisateur veut annuler la fermeture, affectez la valeur **true** à la propriété **Cancel** du paramètre **e** de la méthode.

8. Localisez dans la région **Insertion Service**, la méthode **btnAdd_Click**.

9. En vous inspirant du code mis en place dans la méthode **Form1_Closing**, vérifiez qu'il n'y a pas de mises à jour en attente de sauvegarde avant la région **Affichage**. Vous proposerez alors les mêmes choix à l'utilisateur.

Si l'utilisateur veut enregistrer ses données, utilisez un bloc **try/catch** pour tenter la sauvegarde. En cas d'erreurs, un message sera retourné à l'utilisateur et vous invoquerez la méthode **ResolveError**. Si la valeur de retour de cet appel est **false**, cela signifie que les erreurs doivent être corrigées par l'utilisateur, vous devez donc vous assurez de demander la sortie de la méthode.

Si l'utilisateur ne veut pas enregistrer, recharger les données depuis la base (vous pouvez par exemple réutiliser le code de la méthode **ResolveError** ou recharger le contexte).

Si l'utilisateur veut annuler la fermeture, demander la sortie de la méthode.

10. Toujours dans la même région, localisez la méthode **btnSaveAdd_Click**.

11. Entre les régions **Validation** et **Affichage**, ajoutez le code nécessaire pour instancier un nouvel objet **Service** à l'aide des champs textes de l'interface.

12. Ajoutez ce nouvel objet au contexte et demandez une sauvegarde en base. En cas d'erreur, signalez le problème à l'utilisateur et supprimez l'objet du contexte. Si la sauvegarde se déroule sans erreur, ajoutez le nouveau service à la liste de services de l'interface et mettez à jour les variables **current** (indice de position du service à afficher, en l'occurrence le nouveau) et **nombreService** (compteur du nombre de service à afficher).

13. Testez votre programme.

Exercice 2
Gestion d'entité enfant

Vous allez devoir à présent gérer les employés, à savoir la modification d'informations personnelles, l'ajout d'un nouvel employé mais également la modification de l'association parent-enfant entre un employé et un service.

Toutes les modifications d'employés seront initialisées par une touche clavier lorsque le contrôle grille sera sélectionné par l'utilisateur (touche Insert pour l'ajout, Entrée pour la modification et Suppr pour la suppression). De plus, un formulaire de saisie d'informations a été créé pour gérer les employés (**frmEmploye**). Celui-ci expose une propriété **Employe** qui pourra servir à fournir l'employé à éditer ou à récupérer un nouvel employé. Il faut noter que cette propriété correspondra systématiquement à un objet détaché. **frmEmploye** expose également une propriété en écriture seule **Services** qui devra contenir la liste des objets services disponibles.

1. Localisez la méthode **dgvEmployes_KeyDown** de la région **Gestion des employés**

2. Stockez dans une variable de type **Employe**, celui actuellement sélectionné dans la grille en utilisant la propriété **dgvEmployes.CurrentRow.DataBoundItem** qui correspond à l'objet lié au contrôle sur la ligne en cours.

3. Testez la touche clavier enfoncée par l'utilisateur (**e.KeyCode**) par rapport aux trois valeurs qui nous intéressent : **Keys.Enter**, **Keys.Insert** et **Keys.Delete**.

Cas Ajout :

a. Instanciez et affichez le formulaire d'édition en lui fournissant la liste des services existants.
b. Si l'utilisateur valide son formulaire (**DialogResult.OK**), récupérez le nouvel employé et ajoutez-le au contexte.

Cas Modification :

a. Testez si l'employé sélectionné est différent de **null** et n'est pas déjà en attente de suppression.
b. Instanciez et affichez le formulaire d'édition en lui fournissant la liste des services existants et l'employé à éditer.
c. Si l'utilisateur valide son formulaire (**DialogResult.OK**), récupérez l'employé modifié et fusionnez-le avec l'objet correspondant dans le contexte.

Cas Suppression :

a. Demandez l'utilisateur s'il confirme son choix de suppression en fournissant nom et prénom de l'employé
b. Si l'utilisateur confirme son choix, demandez la suppression de l'employé dans le contexte

Note : les modifications sont uniquement au niveau du contexte, et ne seront sauvegardés en base que si l'utilisateur sélectionne le menu correspondant dans l'interface.

4. Invoquez la méthode **DisplayData** pour rafraichir l'affichage.

Exercice 3
Gestion de conflits

Vous allez ici prendre en charge les erreurs de conflits optimistes qui peuvent être déclenchés lors de la sauvegarde en base de données. Pour cela, vous allez ajouter aux gestions d'erreur déjà mises en place, la gestion de l'exception DbUpdateConcurrency. Ceci vous obligera également à modifier les classes de notre modèle pour qu'Entity Framework puisse détecter ce type d'erreur, et propager cete modification en base.

1. Ajoutez une nouvelle propriété **Modified** de type **byte[]** dans la classe **Employe** et affectez-lui l'attribut **Timestamp**.

2. De façon à propager la modification du schéma sans avoir à supprimer la base, vous allez mettre en œuvre une migration :

a. Ouvrez la console du gestionnaire de package (Menu Outils/Gestionnaire de package NuGet)

b. Activez la prise en charge des migrations avec la commande suivante (en une seule ligne)

```
enable-migrations -ContextProjectName DAL_Library -ContextTypeName DAL_Library.EmployeContext
-ProjectName GestionRH
```

Note : Le contexte n'étant pas défini dans le projet principal, nous indiquons ici le projet contenant le contexte et le nom du contexte, ainsi que le projet qui devra contenir la migration

c. Ajoutez la nouvelle migration

```
add-migration ConflictUpdate
```

d. Envoyez la migration dans la base de données

```
Update-database
```

3. Localisez la méthode **mnuSave_Click** de la région **Gestion Mise à Jour/Interface**.

4. Dans le bloc **try/cacth** précédemment créé, ajoutez un nouveau bloc **catch** pour capturer les exceptions de type **DbUpdateConcurrencyException**

5. Affichez un message à l'utilisateur pour indiquer qu'un conflit a été détecté et que ses modifications vont être perdues

6. Rafraichissez le cache du contexte par rapport à l'entité en erreur (cf. diapositive du cours)

7. Localisez la méthode **DisplayData** de la région **Navigation**

8. Notez que la colonne **Departement** est masquée à l'affichage dans la grille (nous avons déjà l'indication du numéro de service et de plus, s'agissant d'un objet personnalisé, la grille afficherait par défaut **DAL_Library.Service** à moins de polymorpher la méthode **ToString**). Utilisez la même approche pour masquer la colonne **Modified**

9. Pour tester votre programme, lancez deux instances (clic-droit sur le projet dans l'explorateur de solutions, Déboguer puis Démarrer une nouvelle instance). Dans l'une des applications, modifiez un employé et appliquez les modifications dans la base. Avec la deuxième application, modifiez le même employé et constatez la résolution de conflit en cas de sauvegarde dans la base.

Module 6 – Architectures Orientées Service.

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer un service WCF ;
- utiliser un service WCF à travers un proxy ;

Exercice 1
Création d'une service WCF

Afin d'augmenter la visibilité des produits de l'entreprise, il a été décidé de lancer le développement d'une gamme d'applications à destination de tous. Pour cela, il vous est demandé de mettre en place un service fournissant l'accès au catalogue des produits, soit par catégorie, soit par prix.
Afin de vous permettre de tester votre service, il vous a été fournit le prototype d'une des applications qui sera proposée. Il s'agit d'une application WPF assez élémentaire où vous n'aurez qu'à intégrer l'appel à votre service.

1. Ouvrez la solution du dossier **E:\Ateliers\Mod06\Starter\ProduitsSrv**.
2. Ajoutez un nouveau projet de type WCF, Application de service WCF, nommé **ProduitsSrv**
3. Dans le nouveau projet, renommé l'interface **IService1** en **IProductData** et acceptez que la modification soit propagée à tout le code.
4. Renommez le fichier **Service1.svc** en **DataSrv.svc** et assurez-vous que la classe décrite dans le fichier code-behind a bien été renommée également.
5. Affichez le code de balisage du fichier **DataSrv.svc** (clic-droit dans l'exporateur de solutions) et assurez-vous que les noms référencés soient bons.
6. Dans l'interface **IProductData**, remplacez les déclarations existantes par trois déclarations :
GetCategories, sans paramètre qui retournera **List<String>**
GetProduits, acceptant une chaîne en entrée et qui retournera **List<Produit>**
GetProduits, acceptant deux entiers **prixMin** et **prixMax**, et qui retournera **List<Produit>**

Note : même si syntaxiquement, il est possible de créer deux surcharges de la méthode GetProduits, la mise en œuvre en tant que service WCF sera refusée. Il est donc nécessaire de renommer ces méthodes grâce à l'attribut **OperationContract** et la valeur **Name**. Nous nommerons donc nos méthodes **GetProdCat** et **GetProdPrix**.

7. Ajoutez un nouveau modèle de données Code First Entity Framework à partir des tables **Production.Products** et **Production.Categories** de la base **TSQL2012** (vous pouvez utilisez la génération automatique de Visual Studio).
8. Ajoutez une nouvelle classe nommé **Produit**, disposant de 3 propriétés : **Categorie** de type string, **Nom** de type string et **Prix** de type decimal. Cette classe sera celle exposée par notre service, plutôt que d'exposer notre modèle interne. Il arrive d'appeler cette approche DTO (Data Transfer Object).

La classe sera marquée par l'attribut **DataContract**, et les propriétés par **DataMember**, ce qui nous assurera de transmettre la définition de cette classe à nos clients en tant que contrat de données.

9. Ajouter un constructeur à la classe **Produit** acceptant un objet **Product** en entrée et affectant le nom de catégorie, le nom du produit et le prix aux propriétés correspondantes de notre nouvelle classe.
10. Implémentez l'interface de service dans la classe **DataSrv** en prenant en compte les points suivants :
GetCategories : retourne une liste des noms de catégories triés par ordre alphabétique
GetProduits(string) : retourne la liste des produits actifs (discontinued différent de false) et ayant pour nom de catégorie, celui passé en paramètre. La liste devra être de type **List<Produit>** et non **List<Product>**
GetProduits(int prixMini, int prixMax) : même principe que la méthode précédente, mais cette fois pour les produits dont le prix est entre les deux valeurs

Note : Attention à la méthode **ToList**. En effet, appliquée à une requête Linq sur un DbContext, elle attend uniquement des objets d'entité (méthode Linq To Entities et non Linq To Objects).
Ex : *var r = (from p in context.Datas select p).ToList()* fonctionne car s'applique à des objets d'entité.
Si la requête Linq génère des objets non entité, cela échoue à l'exécution (Seuls les constructeurs et les initialiseurs sans paramètre sont pris en charge dans LINQ to Entities.).
Ex : *var r = (from p in context.Datas select new DataB(p)).ToList()*
Une solution est de générer une liste d'objets d'entités avant d'utiliser Linq To Objects pour les transformer en autre chose.

Exercice 2
Mise en œuvre d'une classe proxy

1. Ajoutez une référence de service vers le service **DataSrv** dans le projet **Prototype**

Note : N'hésitez pas à demander à votre formateur de vous remontrer la démarche

2. Localisez la méthode **Window_Loaded** du fichier **MainWindow.xaml.cs** et ajoutez le code nécessaire pour invoquer le proxy généré afin de fournir à la propriété **ItemsSource** de la variable **cbCat**, la liste des noms de catégories (**cbCat** est une liste déroulante)
3. Localisez la méthode **btnCatSearch_Click**
4. Testez si la propriété **SelectedValue** de l'objet **cbCat** est non **null** (il s'agit de la valeur sélectionnée dans la liste déroulante) et dans ce cas, utilisez cette valeur pour invoquer la méthode recherche par nom de catégorie. Le résultat devra être affecté à la propriété **ItemsSource** de l'objet **dgProd** (grille de données en charge d'afficher les produits).

Note : Le contrat de données définit précédemment induit la création d'une nouvelle propriété qui n'est pas significative pour nous. Vous allez la masquer par la ligne :
dgProd.Columns[0].Visibility=Visibility.Collapsed;

Note : il est bon d'invoquer un proxy à travers un bloc **try/catch**.

5. Localisez la méthode **btnPrixSearch_Click**.
6. Recopier le code précédent et adaptez-le à une recherche de produits dont le prix sera entre les deux valeurs fournies par txtMini.Text et txtMax.Text (qui ont été configurées pour n'autoriser que des saisies numériques). Si toutefois, l'utilisateur donne une valeur minimum supérieure à la valeur maximum, il faudra lui signaler par un message sans invoquer le service.
7. Testez votre application

Module 7 – Services REST

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer des services REST ;
- utiliser des services REST ;

Exercice 1
Encapsulation de données

Votre supérieur hiérarchique vient de découvrir la notion de services REST et souhaiterait que vous lui en fassiez la démonstration en lieu et place du service WCF actuellement utilisé pour rechercher les produits. Vous allez donc reprendre votre projet prototype et effectuer les corrections qui s'imposent.

Avant toute chose, ouvrez le projet du dossier **E:\Ateliers\Mod07\Exercice 1\Starter\ProduitsSrv**

1. Dans le projet **ProduitsSrv**, ajoutez un nouvel élément de type WCF Data Service et nommé le **DataSrv2.svc**

2. Ajoutez le fournisseur pre-release EntityFramework pour OData

Par la console du Gestionnaire de package NuGet, bien sélectionner ProduitsSrv comme projet par défaut et exécuter la commande

```
Install-Package Microsoft.OData.EntityFrameworkProvider –Pre
```

Par le Gestionnaire de package NuGet, cochez la case Inclure la version préliminaire et recherchez Microsoft.OData.EntityFrameworkProvider

3. Modifiez le service **DataSrv2** (fichier **DataSrv2.svc.cs**) pour hériter de **EntityFramework<ProduitsModel>**

4. Dans la méthode **InitializeService**, décommentez les lignes d'exemples utilisant les méthodes **SetEntitySetAccessRule** et **SetServiceOperationAccessRule**, et modifiez-les de sorte à autoriser toutes les entités et opérations avec tous les droits de lecture

5. Copiez les trois méthodes de l'ancien service WCF dans le nouveau service et appliquez les modifications suivantes :

GetCategories : ajoutez l'attribut **WebGet**

GetProduits(string categorie) : ajoutez l'attribut **WebGet**, renommez la méthode en **GetProdCat**, changez le type de sortie en **IQueryable<Product>** et corrigez l'expression **LINQ** en conséquence, tout en forçant le chargement de la propriété enfant **Category** (méthode **LINQ Include**)

GetProduits(int prixMin,int prixMax) : ajoutez l'attribut **WebGet**, renommez la méthode en **GetProdPrix**, changez le type de sortie en **IQueryable<Product>** et corrigez l'expression **LINQ** en conséquence, tout en forçant le chargement de la propriété enfant **Category** (méthode **LINQ Include**)

Notes :

- l'attribut **WebGet** assure que l'opération soit visible à travers le service
- une opération de service ne peut retourner que des types primitifs ou d'entité. Il n'est donc plus possible d'utiliser notre classe Produit
- par défaut, nous sommes en chargement **LazyLoading** pour les propriétés de navigation, or nous allons avoir besoin des informations de catégorie côté client, ce qui nous oblige à réaliser un **EagerLoading** via la méthode **Include**.

6. Effectuez un clic-droit dans l'explorateur de solutions sur le fichier **DataSrv2.svc** et affichez le dans le navigateur.

7. Vérifiez que les adresses URL suivantes fonctionnent (XXXX représente le port de votre service) :

http://localhost:XXXX/DataSrv2.svc retourne la description des entités

http://loccalhost:XXXX/DataSrv2.svc/Products retourne tous les produits

http://localhost:XXXX/DataSrv2.svc/Products(1) renvoie le premier produit

http://localhost:XXXX/DataSrv2.svc/GetCategories retourne les noms de catégories

8. Sous Visual Studio 2015 : Ajoutez une référence de service au projet **Prototype** vers **DataSrv2** et nommez-la **ProdServRef2**

Sous Visual Studio 2017 : L'ajout de référence de service OData ne fonctionne plus sans l'installation de composants additionnels. La solution est de générer le proxy directement à l'aide de DataSvcUtil :

Lancer l'***invite de commande développeur pour VS2017*** (en recherchant dans le menu démarrer)

DataSvcUtil /DataServiceCollection /version:2.0 /language:CSharp /out:DataService.cs /uri:http://localhost:XXX/DataSrv2.cs

Le fichier généré doit ensuite être ajouté au projet **Prototype**. Notez que l'espace de nom sera alors **ProduitsSrv** au lieu de **ProdServRef2**.

9. Accédez au fichier **MainWindow.xaml.cs** du projet **Prototype**

9. Ajoutez un nouveau membre **serviceURI** de type **Uri** à la classe **MainWindow** et utilisez le constructeur du type pour l'initialiser à l'adresse de votre service (http://localhost:XXXX/DataSrv2.sv)

10. Dans la méthode **Window_Loaded**, remplacez la variable proxy par une nouvelle variable de type **ProdServRef2.ProduitsModel** et notez que le constructeur attend un objet **Uri** qui sera ici notre variable **serviceURI**

11. Affectez à la propriété **ItemsSource** de **cbCat** l'appel suivant :

```
context.Execute<String>(new Uri("/GetCategories", UriKind.Relative),"GET",false).ToList()
```

Il s'agit ici d'invoquer l'opération par son adresse URI, en indiquant un appel de type http GET. Le troisième paramètre nous permet d'indiquer qu'il ne s'agit pas d'un résultat unique (ici une collection de chaînes)

12. Dans la méthode **btnCatSearch_Click**, modifiez le code pour utilisez l'opération **GetProdCat** pour charger la variable **liste**

Notes :

- le nom de catégorie étant une chaîne, il doit être fourni entre apostrophe '…'
- par défaut, même si les données sont de type imbriqué (parent/enfant…), seul le premier niveau est sérialisé pour être transmis au client, ce qui oblige à demander explicitement le chargement des données enfant par l'opérateur **expand**
- l'opération nous renvoie un ensemble de produit, nous allons donc n'en conserver que les données utiles pour nous grâce à l'opérateur **LINQ Select**

```
context.Execute<Product>(new
Uri($"/GetProdCat?categorie='{cbCat.SelectedValue}'&$expand=Category", UriKind.Relative)).
ToList().Select(p=>new{Categorie=p.Category.categoryname,Nom=p.productname,
Prix=p.unitprice }).ToList()
```

13. Après avoir vérifié le bon fonctionnement de la recherche par catégorie, inspirez-vous du code existant pour modifier la méthode **btnPrixSearch_Click** de sorte à utiliser l'opération **GetProdPrix**

Exercice 2
Service Web API

Vous allez à présent remplacer les services précédents par un service Web API et l'intégrer via HttpClient à l'application Prototype.

Avant toute chose, ouvrez la solution du dossier **E:\Ateliers\Mod07\Exercice 2\ProduitsSrv**.

1. Ajoutez à la solution, un nouveau projet de type Application Web ASP.NET, et choisissez plus précisément un projet vide avec dossiers et références Web API. Nommez le **ProduitsSrv2**.

2. Par un glisser-déplacer, copier les classes **ProduitsModel**, **Category**, **Product** et **Produit** depuis le projet **ProduitsSrv** vers le dossier **Models** du nouveau projet.

3. Pour chacune des classes précédentes, modifiez l'espace de nom de sorte à ce qu'il soit **ProduitsSrv2**.

4. Installez Entity Framework dans ce nouveau projet via le gestionnaire de package NuGet.

5. Copiez la chaîne de connexion du fichier **Web.config** du projet **ProduitsSrv** dans le fichier **Web.config** du projet **ProduitsSrv2**.

6. Compilez le projet et vérifiez qu'il n'y ait pas d'erreurs.

7. Par un clic-droit dans l'explorateur de solutions sur le dossier Controllers, ajoutez deux nouvelles classes de contrôleurs pour **Product** et **Category**, en choisissant un contrôleur Web API avec actions utilisant Entity Framework

8. Dans la nouvelle classe CategoriesController, supprimez tout le code à l'exception de la déclaration du membre privé **db** de type **ProduitsModel**

9. Ajoutez une nouvelle méthode **GetCategoryName**, retournant un objet **IQueryable<string>**, qui sera la liste des noms de catégories triés par ordre alphabétique

Note : le code est très fortement similaire à celui de la méthode **GetCategories** de la classe **DataSrv** du projet **ProduitsSrv**, à l'exception du fait que nous avons déjà un contexte et que nous n'utiliserons pas la méthode **ToList**.

10. Dans la classe **ProductsController**, supprimez tout le code à l'exception du membre **db** et les méthodes **GetProducts()** et **GetProducts(int id)**

11. Modifiez la méthode **GetProducts** pour qu'elle retourne un **IQueryable<Produit>** et modifiez la requête de sorte à transformer les objets **Product** en objet **Produit**

Note :

- Comme nous l'avions constaté avec le service WCF, il n'est pas possible de directement faire *db.Products.Select(p=>new Produit(p))* . Il faut dans un premier temps généré une liste de **Product** pour ensuite les transformer en **Produit** : *db.Products.ToList().Select(p=>new Produit(p))*
- La méthode d'extension **Select** retournant un **IEnumerable<T>**, il nous faut invoquer la méthode **AsQueryable<T>** pour effectuer la conversion nécessaire à nos méthodes

12. Modifiez la méthode **GetProducts(int id)** selon les indications suivantes :

- Enlevez l'attribut **ResponseType**
- Modifiez le type de retour en **IQueryable<Produit>**
- Modifiez le paramètre en **string**
- Supprimez tout le corps de la méthode

13. Le paramètre id sera transmis à la méthode par l'adresse URL (http://serveur/api/Products/id) et sera soit le nom d'une catégorie, soit deux entiers séparés par un ;

En récupérant les requêtes **LINQ** des méthodes **GetProduits** de la classe **DataSrv** du projet **ProduitsSrv**, mettez en œuvre le corps de la méthode de sorte à tester dans un premier temps si le

paramètre **id** ne contient pas un ;, et si oui retourner l'ensemble des produits de la catégorie fournie, sinon ceux dont le prix est entre les deux valeurs séparées par le ;

Note : Pensez à utiliser la méthode **Split** de la classe **System.String**

14. Démarrer en débogage votre site ProduitsSrv2 et testez les adresses URL suivantes :

http://localhost:XXXX/ retourne un message d'erreur 403

http://localhost:XXXX/api/categories retourne la liste des noms de catégories

http://localhost:XXXX/api/products retourne la liste des Produits

http://localhost:XXXX/api/products/beverages retourne la liste des Produits, catégorie Beverages

http://localhost:XXXX/api/products/10;40 retourne la liste des Produits, prix entre 10 et 40

15. Localisez le fichier **WebApiConfig.cs** du dossier **App_Start** et ajoutez le code suivant après l'appel à la méthode **MapHttpAttributeRoutes** :

```
config.Formatters.Clear();
config.Formatters.Add(new JsonMediaTypeFormatter());
```

Le but est ici de forcer la génération de données uniquement au format JSON. En effet, en fonction de l'entête HTTP envoyé par le client, Web API permet de gérer par défaut 4 formats de données, dont le JSON et le XML.

A l'inverse en utilisant la classe **XmlMediaTypeFormatter**, les données sont alors générées en format XML (faites le test). Il est également possible de créer sa propre classe pour définir son propre format.

16. Installez le package **Microsoft.AspNet.WebApi.Client** dans le projet **Prototype**

17. Dans la classe **MainWindow** du projet **Prototype** (fichier **MainWindows.xaml.cs**), ajoutez la méthode suivante (en prenant soin de remplacer l'URL par la vôtre):

```
private T InvokeHttpClient<T>(string URL) where T:class
        {
            using (var client = new HttpClient())
            {
                client.BaseAddress = new Uri("http://localhost:24089/");
                client.DefaultRequestHeaders.Accept.Clear();
                client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));

                HttpResponseMessage response = client.GetAsync(URL).Result;
                if (response.IsSuccessStatusCode)
                {
                    var liste = response.Content.ReadAsAsync<T>().Result;
                   return liste;
                }
                else
                {
                    MessageBox.Show("Une erreur est sruvenue");
                    return null;
                }
```

```
        }
    }
```

Note : Cette méthode instancie un objet **HttpClient** pour communiquer avec notre service Web API. Le type **T** permet d'indiquer le type des informations retournées (pour nous ce sera soit **List<String>** soit **List<Produit>**), et le paramètre **URL** sera le chemin d'invocation (api/controller/id).

18. Ajoutez au projet Prototype une nouvelle classe **Produit** avec trois propriétés **Categorie**, **Nom** et **Prix** (similaire à celle utilisée jusqu'à présent mais sans constructeur ni attributs)

19. Modifiez les trois méthodes **Window_Loaded**, **btnCatSearch_Click** et **btnPrixSearch_Click** de sorte à utiliser la fonction **InvokeHttpClient<T>** à la place du code existant

20. Vérifiez le bon fonctionnement du programme

Annexe – Révision (facultatif)

Objectifs

Le développement de l'application graphique de gestion des employés n'a pas convaincu. De plus, votre entreprise se voit maintenant en charge de la gestion des employés d'autres entités qui viennent d'être racheté par votre groupe.

Il vous est demandé de reprendre l'application initiale GestionRH et de l'adapter pour utiliser Entity Framework tout en préservant les développements de classes initialement mis en œuvre, à savoir Entreprise et Employe.

Vous avez quasiment carte blanche pour cette adaptation, la seule consigne qui vous a été fourni et d'intégrer un premier menu permettant de choisir quelle sera l'entreprise gérée parmi la liste qui sera fourni à l'utilisateur, ou bien de pouvoir créer une nouvelle entreprise (qui sera alors directement gérée pour la suite de l'exécution).

Synopsis d'utilisation :

- Démarrage de l'application
- Affichage de la liste des entreprises (avec affichage de l'identifiant)
- Choix proposés : gérér une entreprise (il sera alors demandé l'identifiant de l'entreprise) ou bien créer une entreprise (il sera alors demandé toutes les informations nécessaires pour cette entreprise)
- Affichage du menu de gestion de l'entreprise (et ainsi reprise de l'existant).

Afin de démarrer votre projet, vous allez pouvoir réutiliser le fichier plat initialement utilisé pour mettre en œuvre le Database Initializer de votre projet et ainsi faire en sorte d'avoir déjà une première entreprise de disponible pour vos tests.

Vous créerez une nouvelle base de données dédiée qui hébergera ensuite vos données.

Pour commencer, localiser dans l'environnement virtuel, le fichier MS861_Ateliers.zip sur le disque E: et allez copier dans l'archive, le dossier Ateliers\Mod02\Starter pour le copier à la racine de E:, ce qui vous permettra de partir de l'application initiale plutôt que la version que vous avez modifié pendant le deuxième module.

MS861
Correction des Ateliers

Module 1

Exercice 1

- **Scénario d'une application de gestion de clients**

Dans ce scénario, la base étant potentiellement modifiable, cela nécessite d'envisager une solution découplée pour éviter que ces modifications n'obligent à corriger le code. Pour cette raison, Entity Framework semble la meilleure solution. De plus, afin d'envisager les futurs développements, la couche d'accès aux données doit être indépendante de sorte à être réutiliser plusieurs fois. Il faudra donc envisager une solution n-Tiers, voire SOA (celle-ci ne serait un avantage que dans l'option d'application d'entreprise de type extranet).

- **Scénario d'une application de gestion de commandes**

Dans ce scénario, l'accès distant est primordial, impliquant l'utilisation d'une solution orientée service. Il est d'ailleurs important de noter qu'il sera à la fois possible aux vendeurs d'accéder à leurs données en temps réel via le service, mais également de gérer l'envoi de leurs données hors connexions par un processus de synchronisation via le service.

- **Scénario d'une application de gestion de livraisons**

Dans ce scénario, le point clé est l'interopérabilité puisque les données seront consommées par tout type d'applications dont votre entreprise n'aura pas la responsabilité. Il est donc nécessaire d'envisager une solution orientée service de type Wep API permettant d'exposer les données de l'entreprise dans le format le plus adapté.

Annexe (Proposition de correction)

Classe Personne

```
public class Personne
    {
                #region Attributs
                private string nom;
                private string prenom;
                private string adresse;
                private int age;

                public string Nom
                {
                        get
                        {
                                return nom;
                        }

                        set
                        {
                                nom = value;
                        }
                }

                public string Prenom
                {
                        get
                        {
                                return prenom;
                        }

                        set
                        {
                                prenom = value;
                        }
                }

                public string Adresse
                {
                        get
                        {
                                return adresse;
                        }

                        set
                        {
                                adresse = value;
                        }
                }

                public int Age
                {
                        get
                        {
                                return age;
                        }

                        set
                        {
                                age = value;
                        }
                }
                #endregion
```

…

Classe Employe

```
public class Employe: Personne, IDisposable
    {
        #region Attributs
        private int numero;
        private string fonction;
        private double salaire;
        #endregion

        #region Constructeurs
        public Employe(string nom, string prenom, string adresse, int age, string
fonction, double salaire)
            : base(nom, prenom, adresse, age)
        {
            this.fonction = fonction;
            this.salaire = salaire;
        }
        public Employe(Personne p, string fonction, double salaire)
            : this(p.Nom, p.Prenom, p.Adresse, p.Age, fonction, salaire) { }

        public Employe() { }
        #endregion

        #region Méthodes
        … // Pas de modification
        #endregion

        #region Dispose
        … // Pas de modification
        #endregion

        #region Propriétés
              public int EmployeId { get; set; }

              public virtual Entreprise Entreprise { get; set; }


              public double Salaire
              {
                     get
                     {
                            return this.salaire;
                     }
                     set
                     {
                            this.salaire = value;
                     }
              }
              public string Fonction
              {
                     get
                     {
                            return fonction;
                     }

                     set
                     {
                            fonction = value;
                     }
              }
```

```
            public int Numero
            {
                get { return this.numero; }
                set { this.numero = value; }
            }
            #endregion
        }
```

Classe Entreprise

```
public class Entreprise: IDisposable, IEnumerable<Employe>
    {
        #region Attributs
            private string nom;
            private List<Employe> employes;
        // public event Action<Object, EntEventArgs> InfoEffectif=null;
         #endregion

         #region Constructeurs
         … // Pas de modification
         #endregion

         #region Méthodes
         … // Pas de modification
         #endregion

         #region Dispose
         … // Pas de modification
         #endregion

         #region Propriétés
            public int EntrepriseId { get; set; }

            public virtual ICollection<Employe> Employes
            {
                get { return employes; }
                set { employes = value.ToList(); }
            }

         public int effectif
         {
             get
             {
                 return employes.Count;
             }
         }

         public double ChargeSalariale
         {
             get
             {
                 double total = 0;
                 for (int i = 0; i < this.effectif; i++)
                 {
                     total += this.employes[i].Salaire;
                 }
                 return total;
             }
         }
```

```
        public string Nom
               {
                      get
                      {
                             return nom;
                      }

                      set
                      {
                             nom = value;
                      }
               }

        public Employe this[int index]
         {
              get
              {
                  if (index < 0 || index >= this.effectif)
                      return null;
                  else
                      return   this.employes[index];
              }
              set
              {

                  if (index < 0 || index >= this.effectif)
                      throw new IndexOutOfRangeException();
                  else
                      this.employes[index] = value;
              }
         }
         #endregion

         #region Enumerable
         … // Pas de modification
         #endregion
    }
}
```

Classe de contexte

```
public class GestionContext:DbContext
        {
               public GestionContext()
               {
                      Database.SetInitializer<GestionContext>(new DBInitializer());
               }
               public DbSet<Entreprise> Entreprises { get; set; }
               public DbSet<Employe> Employes { get; set; }
        }
```

Classe d'intialisation (penser à copier le fichier data.txt dans le projet Info)

```
class DBInitializer:CreateDatabaseIfNotExists<GestionContext>
	{
		protected override void Seed(GestionContext context)
		{
			Entreprise ent = new Info.Entreprise("Microsoft");
			FileStream f = File.OpenRead("data.txt");
			StreamReader sr = new StreamReader(f);
			string ligne = "";
			while ((ligne = sr.ReadLine()) != null)
			{
				string[] data = ligne.Split(';');
				ent.embauche(new Employe(data[0], data[1], data[2],
int.Parse(data[3]), data[4], double.Parse(data[5])));
			}
			sr.Close();
			f.Close();
			context.Entreprises.Add(ent);
			context.SaveChanges();
		}
	}
```

Modification du fichier app.config de l'application cliente

```
<connectionStrings>
	<add name="GestionContext" connectionString="Data Source=.;Initial
	Catalog=GestionEntreprise;Integrated Security=SSPI"
	providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Fichier Program.cs

```
static void Main(string[] args)
		{
			int codeMenu1, codeMenu2;
			GestionContext context = new GestionContext();

			Console.WriteLine("Liste des entreprises:");
			foreach (var item in context.Entreprises)
			{
				Console.WriteLine($"{item.EntrepriseId}: {item.Nom}");
			}
			int refEntId = getEntier("Choisissez le numéro d'entreprise à
gérer ou 0 pour en créer une nouvelle");
			Entreprise ms = null;
			if (refEntId == 0)
			{
				string nomEnt = getString("Veuillez donner le nom de la
 nouvelle entreprise");
				ms = new Entreprise(nomEnt);
				context.Entreprises.Add(ms);
				context.SaveChanges();
			}
			else
			{
				ms = (from e in context.Entreprises
					where e.EntrepriseId == refEntId
					select e).FirstOrDefault();
```

```
                        if (ms == null)
                        {
                                Console.WriteLine("Erreur de récupération des
données. L'application va se fermer.");
                                return;
                        }

                }
                // gestion du menu principal

… // Pas de modification

                } while (codeMenu1 != 5);

                context.SaveChanges();
        }
```