Tutorial : EF Code First VF

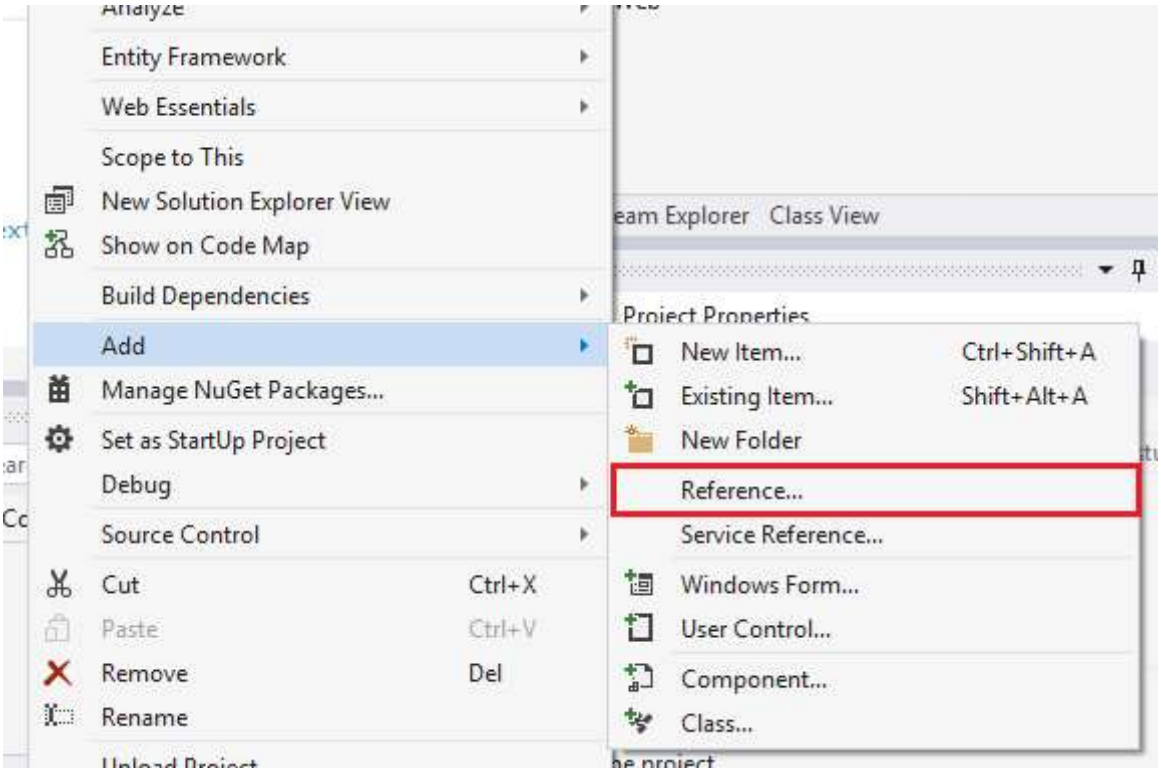
Part 1 - Setting up a solution:

Create these projects in one solution named “MyFinance”

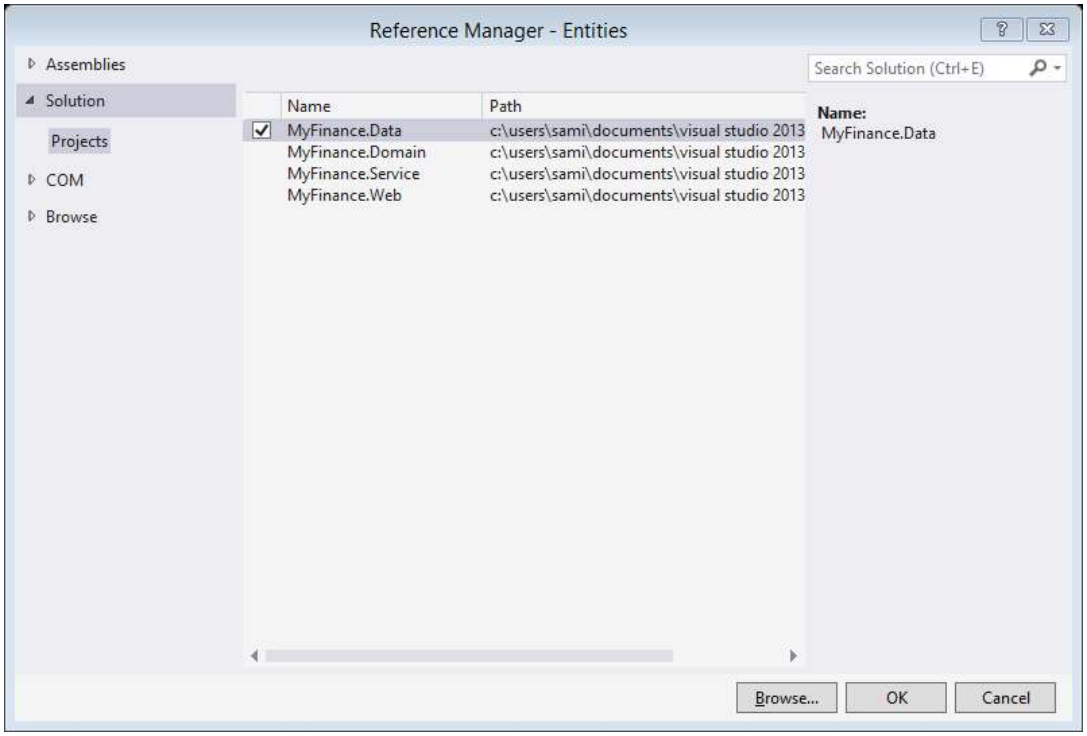
- “MyFinance.Domain”, “MyFinance.Service”,“MyFinance.Service.pattern”, “MyFinance.Data” (4 Class Library Project)
- MyFinance.console(console project)

To add a reference follow these steps:

1. Right click on project that needs the reference
2. Click Add
3. Click Reference



4. In the Reference Manager Message box, check referenced projects

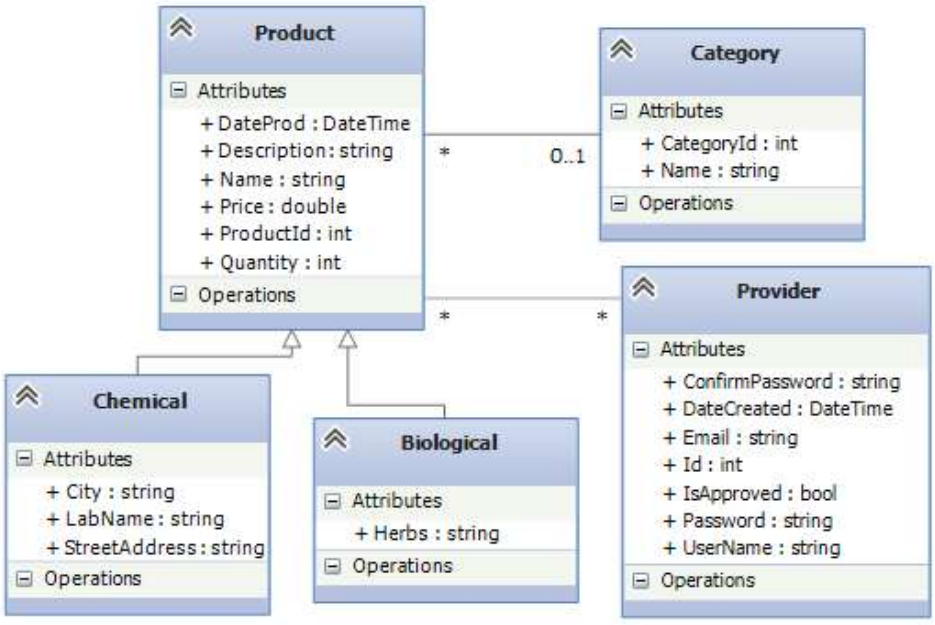


5. Click OK

Part 2– Entities and Context Implementation:

Step 1 :

In the "MyFinance.Domain", you should create a new folder named "***Entities***" that includes all the classes of the followingclasses' diagram, don't forget navigation properties.



The following code expose the different entities:

```
Public class Category
    {
```

```
publicintCategoryId { get; set; }
publicstring Name { get; set; }
//navigation properties
virtualpublicICollection<Product> Products { get; set; }
    }

public class Provider
    {
publicint Id { get; set; }
publicstringUserName { get; set; }
publicstring Password { get; set; }
publicstringConfirmPassword { get; set; }
publicstring Email { get; set; }
publicboolIsApproved { get; set; }
publicDateTime? DateCreated{ get; set; }  // ? ➔nullable
//navigation properties
virtualpublicICollection<Product> Products { get; set; }
    }

public class Product
    {
publicintProductId { get; set; }
        publicstring Name { get; set; }
       publicstring Description { get; set; }
publicdouble Price { get; set; }
publicint Quantity { get;  set; }
publicDateTimeDateProd { get; set; }
//foreign Key properties
publicint?CategoryId{ get; set; }
//navigation properties
publicvirtualCategoryCategory { get; set; }
publicvirtualICollection<Provider> Providers { get; set; }
    }

publicclassBiological : Product
    {
publicstring Herbs { get; set; }
    }

publicclassChemical : Product
    {
publicstringLabName { get; set; }
publicstring City { get; set; }
publicstringStreetAddress { get; set; }
    }
```

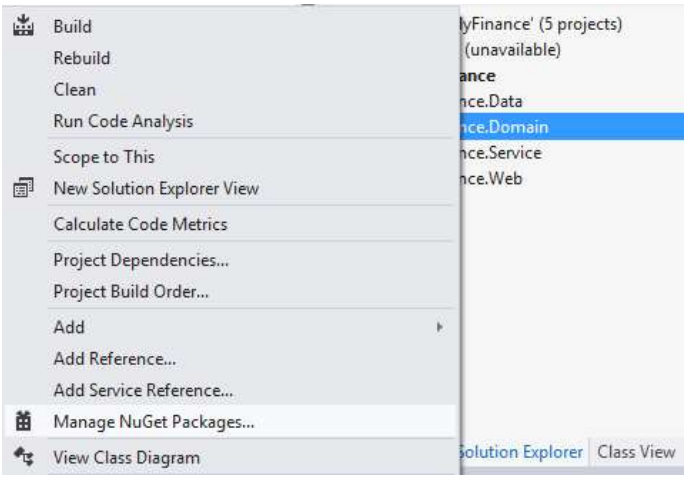
We've made the navigation properties virtual so that we get lazy loading.
DateTime? Type is the nullableDateTime
We used ICollection interface so we can use later any collection implementation.

Step 2 : Entity Framework Installation

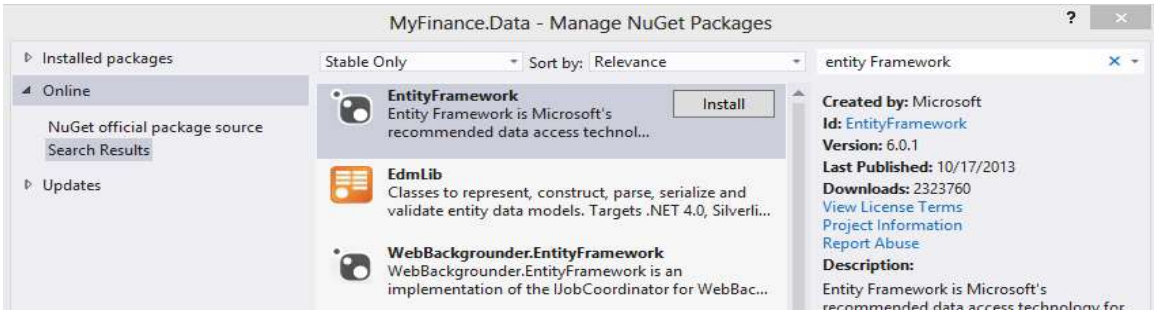
Now, let's move to "MyFinance.Data" project and add a reference to the Entity-Framework. So let's open Manage NuGet package.

We need this reference so we can implement the context.

Do the same for "MyFinance.console" project. Because entity framework won't be deployed unless it's referenced by the web project



We install the Entity Framework package



Step 3 : Context Implémentation

A Context is a class inherits DbContext, and manage entities and database.

Add a new Context by following these steps:

1. Create a class “MyFinanceContext” in “MyFinance.Data” project
2. Add an inheritance from DbContext
3. Add a default constructor that calls the parent one, and give it the named connection string “DefaultConnection” that exists in the web.Config file in “MyFinance.web” project

```
<connectionStrings>
  <add name="DefaultConnection"
       connectionString="Data Source=(Local
       providerName="System.Data.SqlClient"
</connectionStrings>
```

4. Add sets (Collection for entities management)
5. Add necessary using.

```
using MyFinance.Domain.Entities;
```

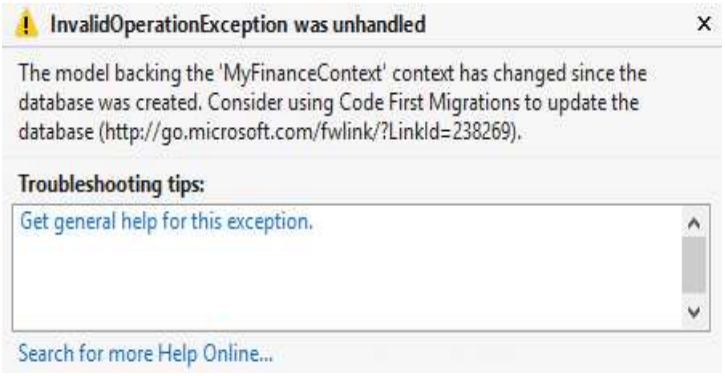
```
usingSystem.Data.Entity;

publicclassMyFinanceContext : DbContext
    {
publicMyFinanceContext()
            : base("Name=DefaultConnection")
        {
        }

publicDbSet<Category> Categories { get; set; }
publicDbSet<Provider> Providers { get; set; }
publicDbSet<Product> Products { get; set; }
```

}– Update the model:

Step 1(Optional):-Let's try to add a property public string Image { get; set; } to the entity Product.

1. Run the application. Try to list products. What happen?

The application throws this exception:



-That exception is thrown because the model changed and the the Entity Framework Code First doesn't know how it will behave. Therefore we have to tell the framework how it must behave.

2. Let's add another class ***MyFinanceContextInitialize***in the same file of the Context.

This class inherit from***DropCreateDatabaseIfModelChanges<MyFinanceContext>,*** which give the Entity Framework **the strategy of initializing the application**. In other word, this class informs the Entity Framework to drop and create the database if the model changes. Wecan replace this class by **DropCreateDatabaseAlways**or **CreateDatabaseIfNotExists**.

3. Override the **Seed** method inherited from the base class.

This method serves to seed the database in case on dropping and creating the database, so we can seed some data for test instead of adding data manually in the database each time the database is created.

```
publicclassMyFinanceContextInitializer :
DropCreateDatabaseIfModelChanges<MyFinanceContext>
    {
protectedoverridevoid Seed(MyFinanceContext context)
        {
varlistCategories = newList<Category>{
       newCategory{Name="Medicament" },
       newCategory{Name="Vetement" },
       newCategory{Name="Meuble" },
       };

context.Categories.AddRange(listCategories);
context.SaveChanges();
        }
    }
```

4. In order to **set the initializer** we have to add this code to the **constructor** of the class **MyFinanceContext**.

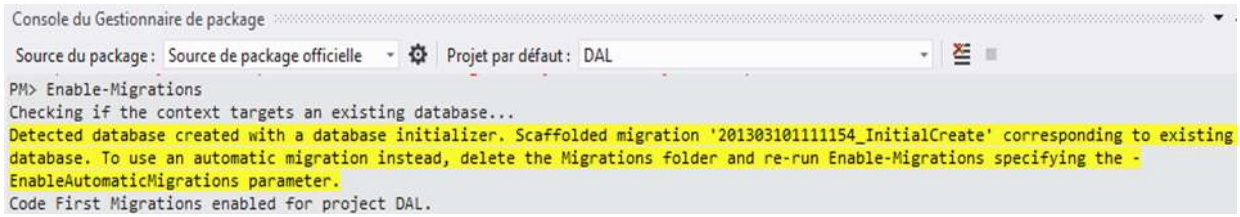
```
Database.SetInitializer<MyFinanceContext>(newMyFinanceContextInitializer());
```

5. In “MyFinance.Web” project, delete the controller “ProductsController” and the “Products” folder under the “Views” Folder and scaffold “ProductsController”. (same steps as Part2, Step4) to refresh the views with the new added property
6. Go to the server explorer panel, right click on your database and close connection.
   This Step is necessary to close all connection to the database, you delete the database in the SQL Server Object Explorer or Restart the visual studio
7. Run the application another time and check the database

Step 2 :-

Assuming that we are using a **production environment** and we have important data. Then we are not allowed to lose this data. In this case the initialisation strategy doesn't seem to be a good choice. So let’s try something else, **Migration** !!!!Let’s do it.

1. The first step: we have to **enable the Migration**: Open the **Package Manager Console**the choose the project “MyFinance.Data” which contains the class Context. Thenexecutethis command **Enable-Migrations**



Entity Framework Code First generate a new folder “Migration” which contains two classesxxx_InitialCreateand Configuration.
-The first class contains an Up () method that contains instructions to create the table with its original definition, and another method Down () to delete the table.
-The second class define the behavior of the migration for DbContext used. This is where you can specify the data to insert, enter the providers of other databases …

2. The second step: we will change the name of the property publicstring Image { get; set; } topublicstringImageName { get; set; }.
3. Execute the command **Add-Migration ModifyNameImage** in the Package Manager Console.
4. The **Add-migration** command checks for changes since your last migration and scaffold a new migration with any changes found. We can give a name for our migration, in this case, we call migration "**ModifyNameImage**".
5. Go to the “Configuration” class in “Migrations” folder and add the following code in the seed method of the migration :

```
context.Categories.AddOrUpdate(
p =>p.Name,  //Uniqueness property
newCategory { Name = "Medicament" },
newCategory { Name = "Vetement" },
newCategory { Name = "Meuble" }
            );
context.SaveChanges();
```

6. Execute the command **Update-Database -TargetMigration:"ModifyNameImage"** in the Package Manager Console. This command will apply any pending migration to the database and then check the database.

Part 4– Complex Type:

Step 1 :Let's add a**Complex Type**"Address"in "entities" folder in "MyFinance.Domain" project

```
public class Address {
public string StreetAddress{ get; set; }
public string City { get; set; }

}
```

To create a new Complex Type we have to follow those rules otherwise we have to configure a complex type using either the data annotations or the fluent API.

Conventional Complex Type Rules

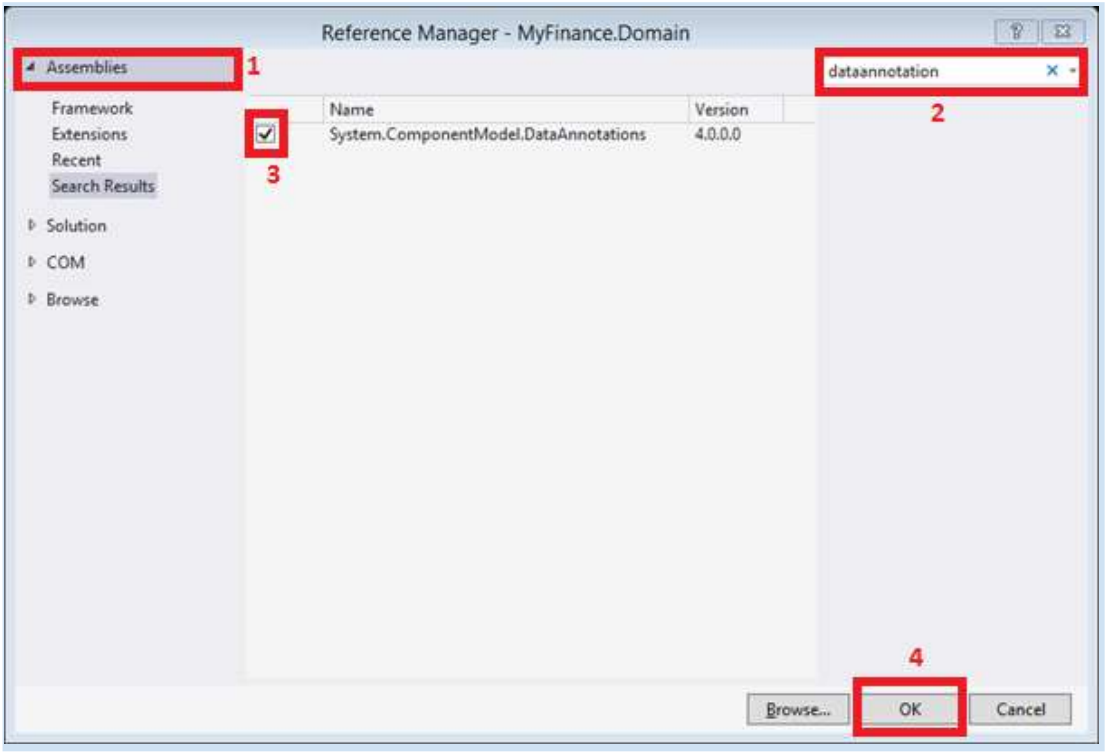
1. Complex types have no key property.
2. Complex types can only contain primitive properties.
3. When used as a property in another class, the property must represent a single instance. It cannot be a collection type.

Step 2 :Let's update "Chemical" entity with the complex type as the following :

```
publicclassChemical : Product
    {
publicstringLabName { get; set; }
publicAddressLabAddress  { get; set; }
    }
```

Part 5– Configuration using annotations:

Step 1 :Add a reference to "System.ComponentModel.DataAnnotions" assembly to the "MyFinance.Model" project



Step 2 : Now , we will add different data annotations to configure those entities.

- In class ***Product***: properties should be configured as the following :
  - the property Name should be
    - required
    - The user input string have the length 25 (max)
    - The property have length 50 (max)
    - An error message will be displayed if the rules are not respected.
  - The property Description should be
    - Multiline
  - The property Price should be
    - Currency
  - The property Quantity should be
    - Positive integer
  - The property DateProd should be
    - Displayed as “Production Date”
    - Valid Date
  - The property CategoryId should be
    - The foreign Key property to the Category entity.

```
publicclassProduct
    {
publicintProductId { get; set; }

        [DataType(DataType.MultilineText)]
publicstring Description { get; set; }

        [Required(ErrorMessage = "Name Required")]
        [StringLength(25, ErrorMessage = "Must be less than 25 characters")]
        [MaxLength(50)]
publicstring Name { get; set; }
```

```
        [Range(0,double.MaxValue)]
        [DataType(DataType.Currency)]
publicdouble Price { get; set; }

        [Range(0, int.MaxValue)]
publicint Quantity { get;  set; }

        [DataType(DataType.ImageUrl),Display(Name = "Image")]
publicstringImageName { get; set; }

        [Display(Name = "Production Date")]
        [DataType(DataType.DateTime)]
publicDateTimeDateProd { get; set; }

        //foreign Key properties
publicint?CategoryId { get; set; }

 //navigation properties
        [ForeignKey("CategoryId")] //useless in this case
        publicvirtualCategoryCategory { get; set; }
publicvirtualICollection<Provider> Providers { get; set; }

}
```

MaxLengthis used for the Entity Framework to decide how large to make a string value field when it creates the database.
StringLengthis a data annotation that will be used for validation of user input.
DataType : is used to specify a specific type : email, currency, card number …
Display is used to change the displayed name of the property
Range is used to limit valid inputs between min and max values.
Foreignkey : To configure the foreign key

- In class ***Provider***: properties should be configured as the following :
  - the property Idshould be
    - Key (Id is already a primary key By Convention)
  - The property Password should be
    - Password (hidden characters in the input)
    - Minimum length 8 characters
    - Required
  - The property ConfirmPassword should be
    - Required
    - Not mapped in the database
    - Password
    - Same value as “Password” property
  - The property Email should be
    - Email
    - Required

```
publicclassProvider
    {
        [Key]   // optional !
publicint Id { get; set; }

publicstringUserName { get; set; }
```

```
        [Required(ErrorMessage = "Password is required")]
        [DataType(DataType.Password)]
        [MinLength(8)]
publicstring Password { get; set; }

        [NotMapped]
        [Required(ErrorMessage = "Confirm Password is required")]
        [DataType(DataType.Password)]
        [Compare("Password")]
publicstringConfirmPassword { get; set; }

        [Required,EmailAddress]
publicstring Email { get; set; }

pub licboolIsApproved { get; set; }

publicDateTime? DateCreated { get; set; }

virtualpublicList<Product> Products { get; set; }
}
```

KeyDefine the Key column in the database..
NotMapped: no column would be generated in the database
Comapre : compare two properties
MinLength the opposite of MaxLength

Step 3 :Let's Test

1. Update the database using migration
2. Delete « ProductsController » and products folder under Views folder
3. scaffold a new « ProductsController »
4. scaffold « ProvidersController »
5. Run and test

Part 6– Configuration using fluent API:

Note that the data annotations and the fluent API configuration could coexist.

Step 1 : Create new Folder named "Configurations" in "MyFinance.Data" project

Step 2 :Add CategoryConfigurationclasses in the "Configurations" folder

```
publicclassCategoryConfiguration : EntityTypeConfiguration<Category>
        {
        publicCategoryConfiguration()
              {
                 ToTable("MyCategories");
                 HasKey(c =>c.CategoryId);
                 Property(c =>c.Name).HasMaxLength(50).IsRequired();
        }
        }
```

-This class inherit from the generic class EntityTypeConfiguration , defined in
« `System.Data.Entity.ModelConfiguration` », then we pass the type that we want to configure

-In fact, Following the convention Entity Framework names the table created in the database with the name of the entity in plural. In this case the table generated will be named Categories. Therefore, when we want to change the name of the table we use the method ToTable("MyCategories").
-We will apply some rules on the Name and the Description proprieties

Step 3 :Add ProductConfigurationclasses in the "Configurations" folder : This step contains 3 main parts. The first part refers to the **many-to-many** association between products and providers, the second part configure the **inheritance,** the third part configure the **one-to-many** association between the **Product** and the **Category**.

```
publicclassProductConfiguration : EntityTypeConfiguration<Product>
       {
       publicProductConfiguration()
        {
//properties configuration
            Property(e =>e.Description).HasMaxLength(200).IsOptional();

//Many to Many configuration
HasMany(p =>p.Providers)
            .WithMany(v =>v.Products)
            .Map(m =>
            {
m.ToTable("Providings");   //Table d'association
m.MapLeftKey("Product");
m.MapRightKey("Provider");
            });

//Inheritance
            Map<Biological>(c =>
            {
c.Requires("IsBiological").HasValue(1);  //isBiological is the descreminator
            });
            Map<Chemical>(c =>
            {
c.Requires("IsBiological").HasValue(0);
            });

//One To Many
HasOptional(p =>p.Category)   // 0,1..*   //if you need 1..* use HasRequired()
                .WithMany(c =>c.Products)
                .HasForeignKey(p =>p.CategoryId)
.WillCascadeOnDelete(false);
        }     }
```

The Many to many part configure the association between the Product class and The Provider class

- Those classes have a many to many association. Entity Framework uses the convention to create an association table following this pattern. The name of the table will be the name of the first entity concatenated with the name of the second one. This table will contain two columns, those names will follow the next pattern:

[The_name_of_the_navigation_propriety][The_name_of_the_primary_Key]. This example explains better.
In this case we want to alter the name of the association table as well as the names of the columns.

The inheritance part customizes the TPH inheritance:

- Indeed Entity Framework create a flag column named **Discriminator**, so we want to change the name of the column to **IsBiological**. This column will contain the value 1 in the case of adding a Biological object. Otherwise it will contain the value 0.

The final part customizes the one-to-many association between the **Product** class and the **Category** class.

- The Product **can** belong to a Category, which explains the optional relationship. In the **HasOptional**method, we pass the **navigation propriety Category** of the Product Class. The method **WithMany**specifies that the Category contains many Products. As well as the other method, we pass the navigation **navigation propertyProducts** of the Category class.
- We also want to disable the Cascade On Delete to conserve products after deleting an associated category. For this aim the foreign key "CategoryId" should be nullable

Step 4 :Finally we configure the **complex type Address** by creating this class:

```
publicclassAddressConfiguration : ComplexTypeConfiguration<Address>
    {
publicAddressConfiguration()
        {
            Property(p =>p.City).IsRequired();
            Property(p =>p.StreetAddress).HasMaxLength(50)
.IsOptional();
        }
    }
```

Step 5 :Update the Context to apply these configurations so, in the "MyFinanceContext" class override the "OnModelCreating" method like this :

```
protectedoverridevoidOnModelCreating(DbModelBuildermodelBuilder)
        {
//If you want to remove all Convetions and work only with configuration :
//  modelBuilder.Conventions.Remove<IncludeMetadataConvention>();
modelBuilder.Configurations.Add(newCategoryConfiguration());
modelBuilder.Configurations.Add(newProductConfiguration());
modelBuilder.Configurations.Add(newAddressConfiguration());
        }
```

Step 6 :Update the database by migration and explore it to see what the configurations have done.

Part 7– Create of custom convention:

Step 1 :Add a new folder in "MyFinance.Data" project called "CustomConventions"

Step 2 :We will create a class called DateTime2Convention in order to change to type of the columns

generated by DateTime proprieties.

```
publicclassDateTime2Convention : Convention
    {
publicDateTime2Convention()
        {
this.Properties<DateTime>().Configure(c =>c.HasColumnType("datetime2"));
        }
    }
```

Step 3 :Update the Context to apply these custom convention so, in the “MyFinanceContext” class override the “OnModelCreating” method like this :

```
protectedoverridevoidOnModelCreating(DbModelBuildermodelBuilder)
        {
//If you want to remove all Convetions and work only with configuration :
//  modelBuilder.Conventions.Remove<IncludeMetadataConvention>();
modelBuilder.Configurations.Add(newCategoryConfiguration());
modelBuilder.Configurations.Add(newProductConfiguration());
modelBuilder.Configurations.Add(newAddressConfiguration());
modelBuilder.Conventions.Add(newDateTime2Convention());
}
```

Part 8– Design Patterns:

Step 1 :Add a new folder in “MyFinance.Data” project called “Infrastructure”

Step 2 :-Let’s move now to the implementation of CRUD methods. No , we won’t use the DAO classes. Why?

-In fact, the pattern DAO is an anti-pattern rather than a design pattern. Supposing that we have 100 pocos, then we have to create 100 DAO classes where we will replicate the same code 100 times.

-Moreover , when we want to modify the behavior of an update method for example , then we have to do the same thing 100 times.

-Therefore , we will use the **Repository pattern**. The repository pattern is an abstraction layer which provides a well-organised approach to maintaining a separation between an application's data access and business logic layers. This gives us the important advantages of making code more maintainable and readable and improving the testability of our code. It also works great with dependency injection!