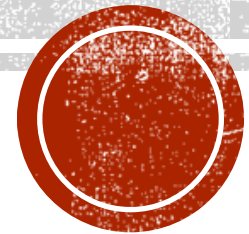


TP N°2

Accès à une base de données SQL SERVER
Avec Entity Framework Core (Approche Code First)
GESTION DES ARTICLES

Enseignants :

Malek Zribi & Lilia Ayadi



PROJET BASÉ SUR L'APPROCHE CODE FIRST

- Ouvrez Visual Studio 2022 et cliquez sur **Créer un nouveau projet** , de type **Application web ASP .Net Core (Modèle-Vue-Contrôleur)**.
- Sous Visual Studio, Click droit sur le nom du projet → Gérer les packages Nuget
- Installer les packages NuGet suivants pour utiliser EF Core dans votre application :

The screenshot displays the Visual Studio NuGet Package Manager window. On the left, a list of search results is shown under the 'Parcourir' (Browse) tab. Four packages are listed, with the first three highlighted by red boxes:

- Microsoft.EntityFrameworkCore.SqlServer** (version 6.0.3) by Microsoft. Description: Microsoft SQL Server database provider for Entity Framework Core.
- Microsoft.EntityFrameworkCore** (version 6.0.3) by Microsoft. Description: Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL, and other databases through a provider plugin API.
- Newtonsoft.Json** (version 13.0.1) by James Newton-King. Description: Json.NET is a popular high-performance JSON framework for .NET.
- Microsoft.EntityFrameworkCore.Tools** (version 6.0.3) by Microsoft. Description: Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.

On the right, the details for **Microsoft.EntityFrameworkCore.SqlServer** are shown. The version selected is 'Dernière version stable 6.0.3'. The 'Installer' button is visible. Below the 'Options' section, the 'Description' is provided: 'Microsoft SQL Server database provider for Entity Framework Core.' Other metadata includes: Version: 6.0.3, Auteur(s): Microsoft, Licence: MIT, and Date de publication: mardi 8 mars 2022 (08/03/2022).



CRÉATION DES CLASSES DU DOMAINE

- Commençons par créer les classes **Product** et **Category** suivantes dans le dossier **Models** :

```
public class Product    {
    public int ProductId { get; set; }
    [Required]
    [StringLength(50, MinimumLength = 5)]
    public string Name { get; set; }
    [Required]
    [Display(Name = "Prix en dinar :")]
    public float Price { get; set; }
    [Required]
    [Display(Name = "Quantité en unité :")]
    public int QteStock { get; set; }
    public int CategoryId { get; set; }
    public Category Category { get; set; }
}

public class Category
{
    public int CategoryId { get; set; }
    [Required]
    [Display(Name = "Nom")]
    public string CategoryName { get; set; }
    public virtual ICollection<Product> Products { get; set; }
}
```



CRÉATION DES CLASSES DU DOMAINE

- Ensuite, il faut créer la classe de contexte **AppDbContext** qui hérite de **DbContext** pour pouvoir communiquer avec une base de données.

```
using Microsoft.EntityFrameworkCore;
namespace TP2.Models {
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
        {
        }
        public DbSet<Product> Products { get; set; }
        public DbSet<Category> Categories { get; set; }
    }
}
```



CRÉATION DES CLASSES DU REPOSITORY

- Créer un dossier **Repositories** sous le dossier **Models**.
- Créer les interfaces **IProductRepository**, **ICategoryRepository** suivante :

```
public interface IProductRepository
{
    Product GetById(int Id);
    IList<Product> GetAll();
    void Add(Product t);
    Product Update(Product t);
    void Delete(int Id);
    public IList<Product> GetProductsByCategID(int? CategId);
    public IList<Product> FindByName(string name);
}

public interface ICategoryRepository
{
    Category GetById(int Id);
    IList<Category> GetAll();
    void Add(Category t);
    Category Update(Category t);
    void Delete(int Id);
}
```

- Créer maintenant les classes **ProductRepository** et **CategoryRepository** qui implémentent respectivement **IProductRepository** et **ICategoryRepository** .



CRÉATION DES CLASSES DU REPOSITORY

```
public class ProductRepository : IProductRepository
{
    readonly AppDbContext context;
    public ProductRepository(AppDbContext context) {
        this.context = context;
    }
    public IList<Product> GetAll() {
        return context.Products
            .OrderBy(x => x.Name)
            .Include(x => x.Category).ToList();
    }
    public Product GetById(int id) {
        return context.Products
            .Where(x => x.ProductId == id)
            .Include(x => x.Category)
            .SingleOrDefault();
    }
    public void Add(Product p) {
        context.Products.Add(p);
        context.SaveChanges();
    }
    public IList<Product> FindByName(string name) {
        return context.Products
            .Where(p => p.Name.Contains(name) ||
                p.Category.CategoryName.Contains(name))
            .Include(c => c.Category).ToList();
    }
}
```

```
    public Product Update(Product p) {
        Product p1 = context.Products.Find(p.ProductId);
        if (p1 != null) {
            p1.Name = p.Name;
            p1.Price = p.Price;
            p1.QteStock = p.QteStock;
            p1.CategoryId = p.CategoryId;
            context.SaveChanges();
        }
        return p1;
    }
    public void Delete(int ProductId) {
        Product p1 = context.Products.Find(ProductId);
        if (p1 != null)
        {
            context.Products.Remove(p1);
            context.SaveChanges();
        }
    }
    public IList<Product> GetProductsByCategID(int? CategId)
    {
        return context.Products
            .Where(p => p.CategoryId.Equals(CategId))
            .OrderBy(p => p.ProductId)
            .Include(p => p.Category).ToList();
    }
}
```



CRÉATION DES CLASSES DU REPOSITORY

```
public class CategoryRepository : ICategoryRepository
{
    readonly AppDbContext context;
    public CategoryRepository(AppDbContext context)
    {
        this.context = context;
    }
    public IList<Category> GetAll()
    {
        return context.Categories
            .OrderBy(c => c.CategoryName).ToList();
    }
    public Category GetById(int id)
    {
        return context.Categories.Find(id);
    }
    public void Add(Category c)
    {
        context.Categories.Add(c);
        context.SaveChanges();
    }
}
```

```
    public Category Update(Category c)
    {
        Category c1 = context.Categories.Find(c.CategoryId);
        if (c1 != null)
        {
            c1.CategoryName = c.CategoryName;
            context.SaveChanges();
        }
        return c1;
    }
    public void Delete(int CategoryId)
    {
        Category c1 = context.Categories.Find(CategoryId);
        if (c1 != null)
        {
            context.Categories.Remove(c1);
            context.SaveChanges();
        }
    }
}
```



CONFIGURATION DU FOURNISSEUR DE BD

- Nous pouvons utiliser la *méthode* `AddDbContext ()` ou `AddDbContextPool ()` pour inscrire notre classe `DbContext` spécifique à l'application avec le système d'injection de dépendances ASP.NET Core au niveau de la classe **Program.cs** :

```
builder.Services.AddDbContextPool<AppDbContext>(options =>  
options.UseSqlServer(builder.Configuration.GetConnectionString("ProductDBConnection"  
)));
```

- La chaîne de connexion nommée **"ProductDBConnection"** doit être définie dans le fichier de configuration du projet **appsettings.json** plutôt que dans le code. Pour cela, il faut ajouter cette clé dans le fichier **appsettings.json** :

```
"ConnectionStrings": {  
  "ProductDBConnection":  
    "server=(localdb)\\MSSQLLocalDB;database=MyBaseDB;Trusted_Connection=true"  
}
```

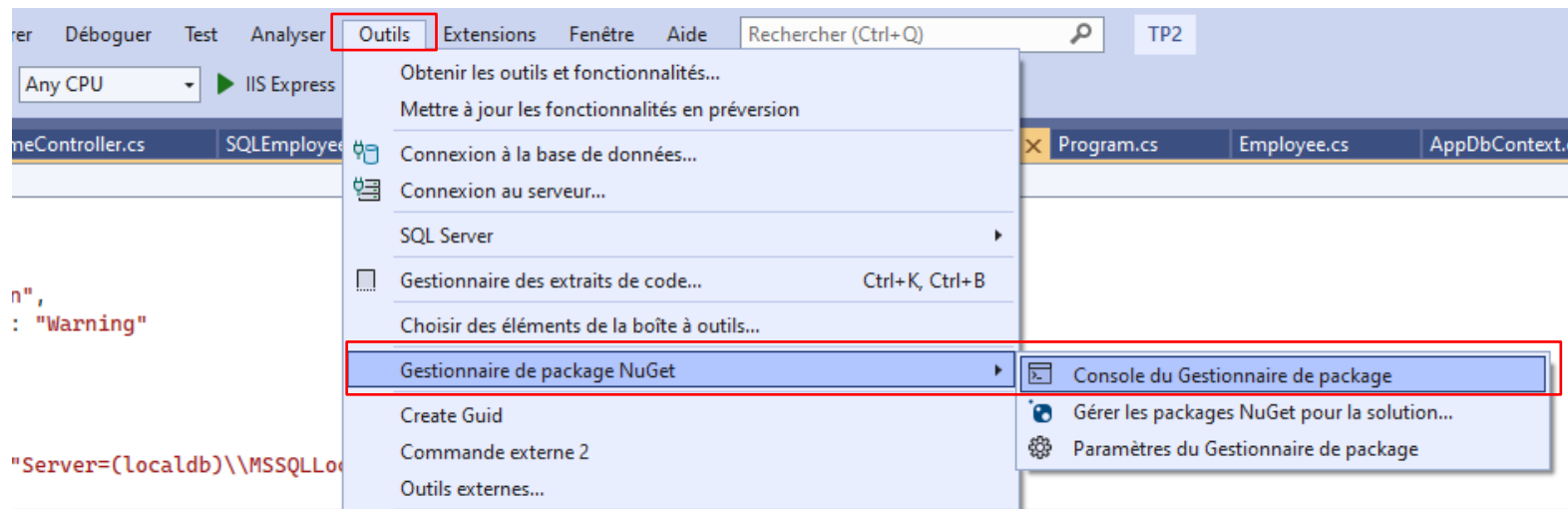


INJECTION DES DÉPENDANCES ET CRÉATION DE LA BD

- N'oubliez pas d'injecter les dépendances dans le code de la classe **Program.cs** comme suit :

```
builder.Services.AddScoped<IProductRepository, ProductRepository>();  
builder.Services.AddScoped<ICategorieRepository, CategoryRepository>();
```

- Pour créer la base de données, il faut appliquer la migration comme suit dans la **console du gestionnaire de package NuGet** à partir du **menu Options** :



CRÉATION DE LA BD (SUITE)

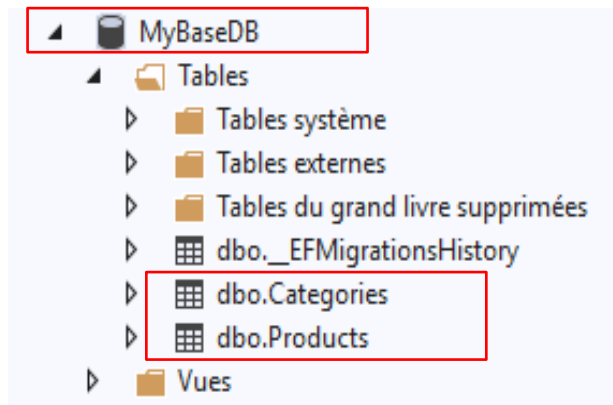
- La commande suivante crée la migration initiale. InitialCreate est le nom de la migration.

PM> Add-Migration InitialCreate

- Lorsque la commande ci-dessus se termine, vous verrez un fichier dans le dossier "Migrations" qui contient le nom InitialCreate.cs . Ce fichier contient le code requis pour créer les tables de la base de données.
- Pour mettre à jour la base de données, nous utilisons la commande *Update-Database*.

PM> Update-Database

- Vous pouvez maintenant aller à l'explorateur d'objets SQL SERVER, et vérifier si la base de données a été bien créée avec les tables Product et Category.



CRÉER UN CONTRÔLEUR

- Dans le dossier Controllers, Créer deux nouveaux contrôleurs nommés **ProductController** et **CategoryController** (avec read/write actions) permettant de gérer les opérations sur les différents produits et catégories. Compléter le code des méthodes d'action en se basant sur le TP N°1.

```
namespace TP_Product.Controllers
{
    1 référence
    public class ProductController : Controller
    {
        readonly IProductRepository ProductRepository;

        0 références
        public ProductController(IProductRepository ProdRepository) {
            ProductRepository = ProdRepository;
        }
        // GET: ProductController
        1 référence
        public ActionResult Index()
        {
            var Products = ProductRepository.GetAll();
            return View(Products);
        }
    }
}
```

```
namespace TP_Product.Controllers
{
    1 référence
    public class CategoryController : Controller
    {
        readonly ICategorieRepository CategRepository;

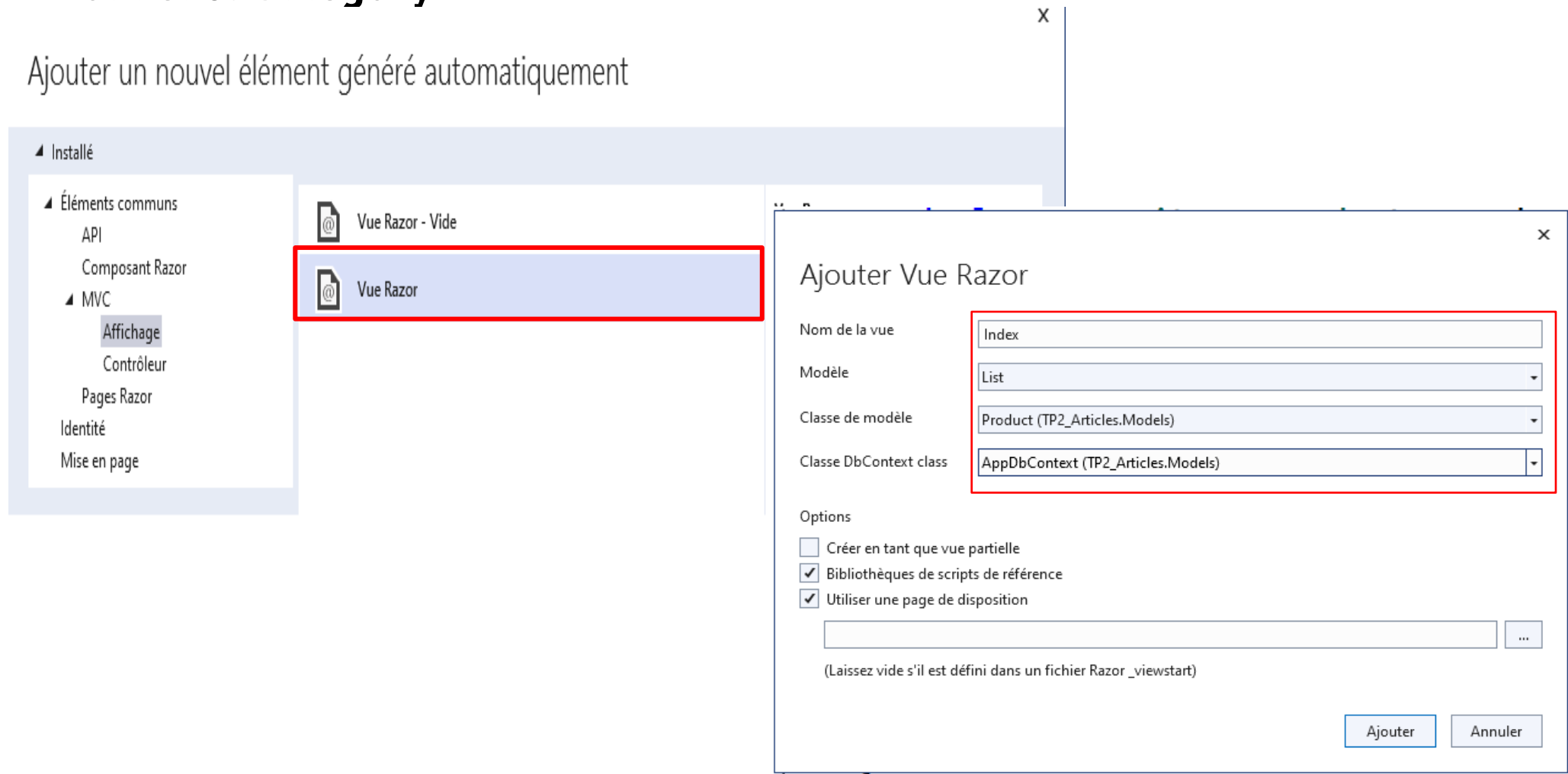
        0 références
        public CategoryController(ICategorieRepository categRepository)
        {
            CategRepository = categRepository;
        }
        // GET: CategoryController
        3 références
        public ActionResult Index()
        {
            var Categories = CategRepository.GetAll();
            return View(Categories);
        }
    }
    //Compléter le code ...
}
```



CRÉER LES VUES

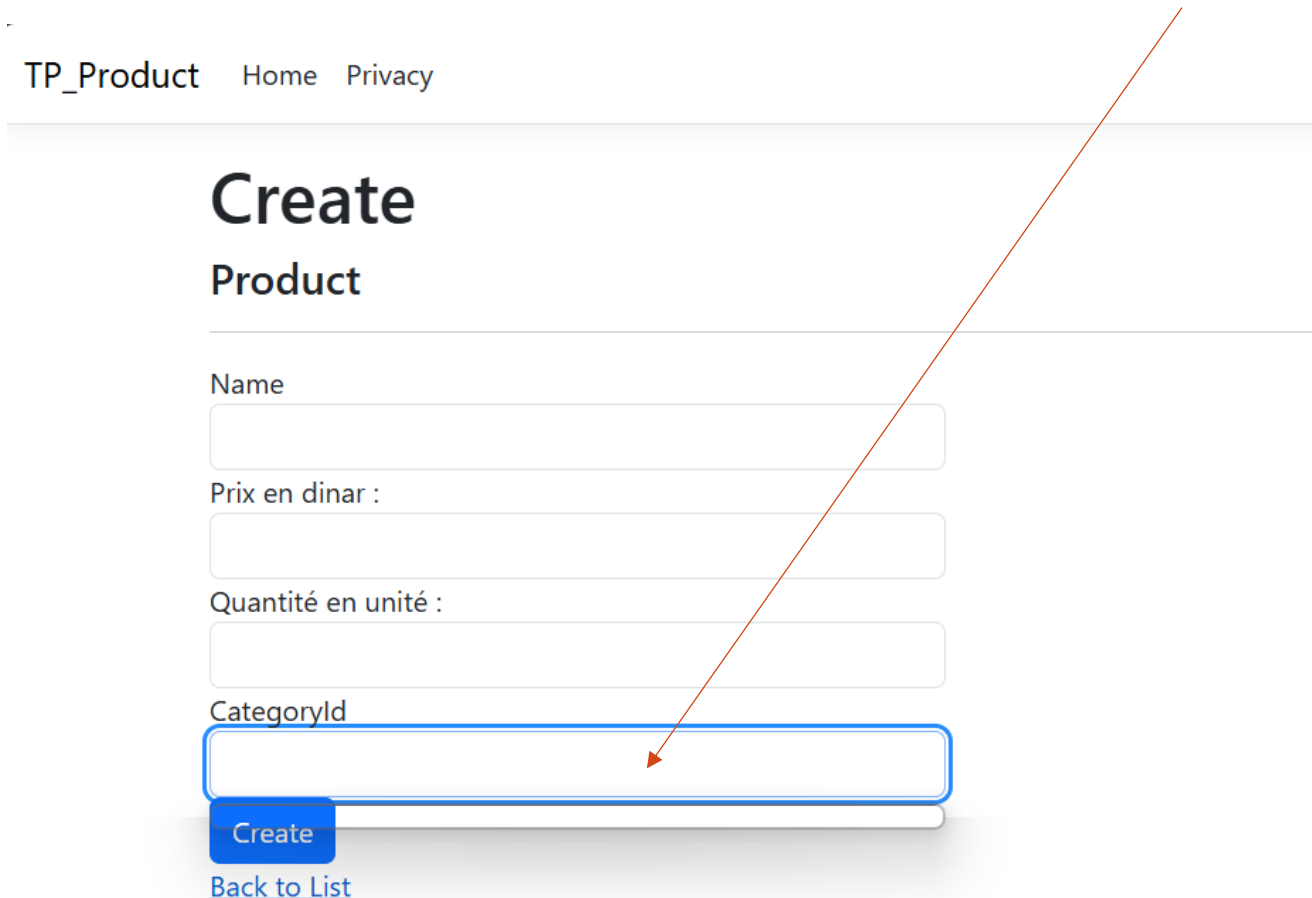
- Passons maintenant à la création des vues de notre application, et commençons par la vue de la méthode Index, puis compléter les vues Create, Edit, Delete et Details pour **Product** et **Category**:

Ajouter un nouvel élément généré automatiquement



EXÉCUTION

- Une fois les vues créées et on exécute notre projet, on va remarquer que lorsqu'on désire ajouter un produit, le champ **CategoryId** est représenté sous forme d'une liste déroulante vide créée sous forme d'un ViewBag nommé : "ViewBag.CategoryID" dans la vue Create.



TP_Product Home Privacy

Create

Product

Name

Prix en dinar :

Quantité en unité :

CategoryId

Create

[Back to List](#)



MODIFICATION DE PRODUCTCONTROLLER

- On va alors relier la ViewBag à la base de données et précisément au champ **CategoryName** de la table **Category**.
- Pour cela, apporter les modifications suivantes à **ProductController** :

```
public class ProductController : Controller
{
    readonly IProductRepository ProductRepository;
    readonly ICategorieRepository CategRepository;

    public ProductController(IProductRepository ProdRepository, ICategorieRepository
    categRepository)
    {
        ProductRepository = ProdRepository;
        CategRepository = categRepository;
    }
}
```

- Ajouter cette instruction dans les actions Get et Post de l'action **Create** et **Edit** du contrôleur **ProductController** :

```
ViewBag.CategoryId = new SelectList(CategRepository.GetAll(),
"CategoryId", "CategoryName");
```



AJOUTER UNE MIGRATION

- On veut modifier la classe "**Product**" en ajoutant une propriété **Image** qui va contenir le chemin d'accès à l'emplacement de l'image correspondante à un article :

```
public class Product
{
    public int ProductId { get; set; }
    [Required]
    [StringLength(50, MinimumLength = 5)]
    public string Name { get; set; }
    [Required]
    [Display(Name = "Prix en dinar :")]
    public float Price { get; set; }
    [Required]
    [Display(Name = "Quantité en unité :")]
    public int QteStock { get; set; }
    public int CategoryId { get; set; }
    public Category Category { get; set; }

    [Required]
    [Display(Name = "Image :")]
    public string Image { get; set; }
}
```



AJOUTER UNE MIGRATION

- Pour grader la synchronisation entre la base de données et les classes de modèle, il faut lancer une Migration via la commande Add-Migration.

```
PM> Add-Migration AddPhotoPathToProducts
```

- Pour appliquer la migration et mettre à jour la base de données, utilisez la commande suivante :

```
PM> Update-Database
```



UPLOAD FILE

- Pour pouvoir faire un **upload** d'une image, nous avons besoin de créer un attribut de type **IFormFile**. Comme il n'est pas pratique de déclarer cet attribut dans notre classe de modèle **Product**, on a besoin alors de créer une classe **ViewModel**.
- Commençons alors par créer un dossier **ViewModels** dans notre projet.
- Ensuite, créer la classe **CreateViewModel** suivante :

```
public class CreateViewModel {  
    public int ProductId { get; set; }  
    [Required]  
    [StringLength(50, MinimumLength = 5)]  
    public string Name { get; set; }  
    [Required]  
    [Display(Name = "Prix en dinar :")]  
    public float Price { get; set; }  
    [Required]  
    [Display(Name = "Quantité en unité :")]  
    public int QteStock { get; set; }  
    public int CategoryId { get; set; }  
    public Category Category { get; set; }  
    [Required]  
    [Display(Name = "Image :")]  
    public IFormFile ImagePath { get; set; }  
}
```

```
public IFormFile ImagePath { get; set; }  
}
```



UPLOAD FILE

- Nous allons passer à modifier la méthode d'action **Create** du contrôleur pour pouvoir télécharger une image.

```
public class ProductController : Controller
{
    readonly IProductRepository ProductRepository;
    readonly ICategorieRepository CategRepository;
    private readonly IWebHostEnvironment hostingEnvironment;
    public ProductController(IProductRepository ProdRepository, ICategorieRepository categRepository,
        IWebHostEnvironment hostingEnvironment)
    {
        ProductRepository = ProdRepository;
        CategRepository = categRepository;
        this.hostingEnvironment = hostingEnvironment;
    } //...
    // POST: ProductController/Create
    [HttpPost]
    [ValidateAntiForgeryToken]
    public ActionResult Create(CreateViewModel model) {
        if (ModelState.IsValid) {
            string uniqueFileName = null;
            // If the Photo property on the incoming model object is not null,
            // then the user has selected an image to upload.
            if (model.ImagePath != null)
            {
                // The image must be uploaded to the images folder in wwwroot
                // To get the path of the wwwroot folder we are using the inject
                // HostingEnvironment service provided by ASP.NET Core
                string uploadsFolder = Path.Combine(hostingEnvironment.WebRootPath, "images");
```



UPLOAD FILE

```
// To make sure the file name is unique we are appending a new
// GUID value and an underscore to the file name
uniqueFileName = Guid.NewGuid().ToString() + "_" + model.ImagePath.FileName;
string filePath = Path.Combine(uploadsFolder, uniqueFileName);

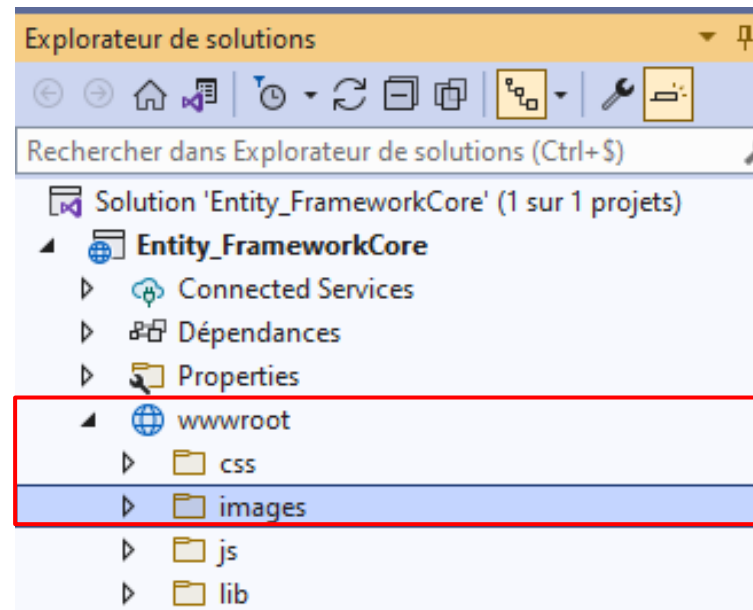
// Use CopyTo() method provided by IFormFile interface to
// copy the file to wwwroot/images folder
model.ImagePath.CopyTo(new FileStream(filePath, FileMode.Create));
}
Product newProduct = new Product
{
    Name = model.Name,
    Price = model.Price,
    QteStock = model.QteStock,
    CategoryId = model.CategoryId,
    // Store the file name in PhotoPath property of the employee object
    // which gets saved to the Employees database table
    Image = uniqueFileName
};

ProductRepository.Add(newProduct);
return RedirectToAction("details", new { id = newProduct.ProductId });
}
return View();
}
```



UPLOAD FILE

- Ajouter un nouveau sous dossier nommé **images** dans le dossier **wwwroot** dans lequel les images y seront automatiquement enregistrées.



UPLOAD FILE

- Maintenant, nous allons remplacer le code de la vue Create par le code suivant :

```
@model TP_Product.ViewModels.CreateViewModel
@{
    ViewData["Title"] = "Create";
}
<h1>Create</h1>
<h4>Product</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        @*To support file upload set the form element enctype="multipart/form-data" *@
        <form enctype="multipart/form-data" asp-action="Create">
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="QteStock" class="control-label"></label>
                <input asp-for="QteStock" class="form-control" />
                <span asp-validation-for="QteStock" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="CategoryId" class="control-label"></label>
                <select asp-for="CategoryId" class="form-control" asp-items="ViewBag.CategoryId"></select>
            </div>
        </form>
    </div>
</div>
```



UPLOAD FILE

```
@* asp-for tag helper is set to "ImagePath" property. "ImagePath" property type is IFormFile so at runtime asp.net core generates
file upload control (input type=file)*@
<div class="form-group row">
  <label asp-for="ImagePath" class="col-sm-2 col-form-label"></label>
  <div class="col-sm-10">
    <div class="custom-file">
      <input asp-for="ImagePath" class="form-control custom-file-input">
      <label class="custom-file-label">Choose File...</label>
    </div>
  </div>
</div>
<div class="form-group">
  <input type="submit" value="Create" class="btn btn-primary" />
</div>
</form>
</div>
</div>
<div>
  <a asp-action="Index">Back to List</a>
</div>
@*@section Scripts {
  @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}*@
@*This script is required to display the selected file in the file upload element*@
@section Scripts {
  <script>
    $(document).ready(function () {
      $('<div class="custom-file">').on("change", function () {
        var fileName = $(this).val().split("\\").pop();
        $(this).next('<div class="custom-file-label">').html(fileName);
      });
    });
  </script>
}
```



UPLOAD FILE

- Le code de la vue **Details** sera le suivant :

```
@model TP_Product.Models.Product
```

```
@{
```

```
    ViewData["Title"] = "Details";
```

```
    var photoPath = "~/images/" + (Model.Image ?? "noimage.jpg");
```

```
}
```

```
<h1>Details</h1>
```

```
<div>
```

```
    <h4>Product</h4>
```

```
    <hr />
```

```
    <dl class="row">
```

```
        <dt class = "col-sm-2">
```

```
            @Html.DisplayNameFor(model => model.Name)
```

```
        </dt>
```

```
        <dd class = "col-sm-10">
```

```
            @Html.DisplayFor(model => model.Name)
```

```
        </dd>
```

```
        <dt class = "col-sm-2">
```

```
            @Html.DisplayNameFor(model => model.Price)
```

```
        </dt>
```

```
        <dd class = "col-sm-10">
```

```
            @Html.DisplayFor(model => model.Price)
```

```
        </dd>
```

```
        <dt class = "col-sm-2">
```

```
            @Html.DisplayNameFor(model => model.QteStock)
```

```
        </dt>
```

```
        <dd class = "col-sm-10">
```

```
            @Html.DisplayFor(model => model.QteStock)
```

```
        </dd>
```

```
        <dt class = "col-sm-2">
```

```
            @Html.DisplayNameFor(model => model.Category)
```

```
        </dt>
```

```
    <dd class = "col-sm-10">
```

```
        @Html.DisplayFor(model => model.Category.CategoryName)
```

```
    </dd>
```

```
    <dt class="col-sm-2">
```

```
        <label><b>Image</b> </label>
```

```
    </dt>
```

```
    <dd class="col-sm-10">
```

```
        <div class="card-body text-left">
```

```
            
```

```
        </div>
```

```
    </dd>
```

```
    </dl>
```

```
</div>
```

```
<div>
```

```
    <a asp-action="Edit" asp-route-id="@Model?.ProductId">Edit</a> |
```

```
    <a asp-action="Index">Back to List</a>
```

```
</div>
```



MODIFICATION D'UN PRODUIT EXISTANT

- Pour pouvoir modifier l'image d'un produit, nous allons suivre les étapes suivantes :
- Créer une classe **EditViewModel** sous le répertoire **ViewModels** dont le code est le suivant :

```
public class EditViewModel : CreateViewModel
{
    public int ProductId { get; set; }
    public string ExistingImagePath { get; set; }
}
```

- Remplacer le code de la méthode **Edit** du contrôleur par le code suivant :

```
// GET: ProductController/Edit/5
public ActionResult Edit(int id)
{
    ViewBag.CategoryId = new SelectList(CategRepository.GetAll(),
    "CategoryId", "CategoryName");
    Product product = ProductRepository.GetById(id);
    EditViewModel productEditViewModel = new EditViewModel
    {
        ProductId = product.ProductId,
        Name = product.Name,
        Price = product.Price,
        QteStock = product.QteStock,
        CategoryId = product.CategoryId,
        ExistingImagePath = product.Image
    };
    return View(productEditViewModel);
}
```



MODIFICATION D'UN PRODUIT EXISTANT

```
// POST: ProductController/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(EditViewModel model){
    ViewBag.CategoryId = new SelectList(CategRepository.GetAll(), "CategoryId", "CategoryName");
    // Check if the provided data is valid, if not rerender the edit view
    // so the user can correct and resubmit the edit form
    if (ModelState.IsValid) {
        // Retrieve the product being edited from the database
        Product product = ProductRepository.GetById(model.ProductId);
        // Update the product object with the data in the model object
        product.Name = model.Name;
        product.Price = model.Price;
        product.QteStock = model.QteStock;
        product.CategoryId = model.CategoryId;
        // If the user wants to change the photo, a new photo will be
        // uploaded and the Photo property on the model object receives
        // the uploaded photo. If the Photo property is null, user did
        // not upload a new photo and keeps his existing photo
        if (model.ImagePath != null) {
            // If a new photo is uploaded, the existing photo must be
            // deleted. So check if there is an existing photo and delete
            if (model.ExistingImagePath != null)
            {
                string filePath = Path.Combine(hostingEnvironment.WebRootPath, "images", model.ExistingImagePath);
                System.IO.File.Delete(filePath);
            }
            // Save the new photo in wwwroot/images folder and update
            // PhotoPath property of the product object which will be
            // eventually saved in the database
            product.Image = ProcessUploadedFile(model);
        }
    }
}
```



MODIFICATION D'UN PRODUIT EXISTANT

```
// Call update method on the repository service passing it the
// product object to update the data in the database table
Product updatedProduct = ProductRepository.Update(product);

    if (updatedProduct != null)
        return RedirectToAction("Index");
    else
        return NotFound();

}

return View(model);
}
[NonAction]
private string ProcessUploadedFile(EditViewModel model)
{
    string uniqueFileName = null;

    if (model.ImagePath != null)
    {
        string uploadsFolder = Path.Combine(hostingEnvironment.WebRootPath, "images");
        uniqueFileName = Guid.NewGuid().ToString() + "_" + model.ImagePath.FileName;
        string filePath = Path.Combine(uploadsFolder, uniqueFileName);
        using (var fileStream = new FileStream(filePath, FileMode.Create))
        {
            model.ImagePath.CopyTo(fileStream);
        }
    }

    return uniqueFileName;
}
```



MODIFICATION D'UN PRODUIT EXISTANT

```
@model TP_Product.ViewModels.EditViewModel
@{
    ViewData["Title"] = "Edit";
    // Get the full path of the existing product photo for display
    var photoPath = "~/images/" + (Model.ExistingImagePath ?? "noimage.jpg");
}
<h1>Edit</h1>
<h4>Product</h4>
<hr />
<form asp-controller="Product" asp-action="edit" enctype="multipart/form-data" method="post" class="mt-3">
    <div asp-validation-summary="All" class="text-danger"> </div>
    @*Use hidden input elements to store productid and ExistingImagePath which we need when we submit the form and update data in the DB*@
    <input hidden asp-for="ProductId" />
    <input hidden asp-for="ExistingImagePath" />
    @*Bind to the properties of the ProductEditViewModel.*@
    <div class="form-group row">
        <label asp-for="Name" class="control-label"></label>
        <input asp-for="Name" class="form-control" />
        <span asp-validation-for="Name" class="text-danger"></span>
    </div>
    <div class="form-group row">
        <label asp-for="Price" class="control-label"></label>
        <input asp-for="Price" class="form-control" />
        <span asp-validation-for="Price" class="text-danger"></span>
    </div>
    <div class="form-group row">
        <label asp-for="QteStock" class="control-label"></label>
        <input asp-for="QteStock" class="form-control" />
        <span asp-validation-for="QteStock" class="text-danger"></span>
    </div>
    <div class="form-group row">
        <label asp-for="CategoryId" class="control-label"></label>
        <select asp-for="CategoryId" class="form-control" asp-items="ViewBag.CategoryId"></select>
        <span asp-validation-for="CategoryId" class="text-danger"></span>
    </div>
</form>
```

- Le code de la vue **Edit** sera le suivant :



MODIFICATION D'UN PRODUIT EXISTANT

```
<div class="form-group row">
  <label asp-for="ImagePath" class="col-sm-2 col-form-label"></label>
  <div class="col-sm-10">
    <div class="custom-file">
      <input asp-for="ImagePath" class="custom-file-input form-control" id="imageInput">
      <label class="custom-file-label" for="imageInput">Cliquer ici pour changer la photo</label>
    </div>
  </div>
</div>
@* Display the existing Product photo *@
<div class="form-group row col-sm-4 offset-4">
  
</div>
<div class="form-group row">
  <div class="col-sm-10">
    <button type="submit" class="btn btn-primary">Update</button>
    <a asp-action="index" asp-controller="Product" class="btn btn-primary">Cancel</a>
  </div>
</div>
@section Scripts {
  <script>
    $(document).ready(function () {
      $('.custom-file-input').on("change", function () {
        var fileName = $(this).val().split("\\").pop();
        $(this).next('.custom-file-label').html(fileName);
        // Mise à jour de l'image de prévisualisation
        var reader = new FileReader();
        reader.onload = function (e) {
          $('#imageEmp').attr('src', e.target.result); // Utilisation de la nouvelle image sélectionnée
        };
        reader.readAsDataURL(this.files[0]); // Lecture du fichier sélectionné
      });
    });
  </script>
}
</form>
```



NOUVEL AFFICHAGE DE LA LISTE

- Pour afficher la liste des articles avec leurs images, nous allons utiliser la classe **bootstrap card-group**. Le nouveau code de la vue Index sera le suivant :

```
@model IEnumerable<TP_Product.Models.Product>
@{
    ViewData["Title"] = "Liste des produits";
}
<div class="card-group">
    @foreach (var product in Model)
    {
        var photoPath = "~/images/" + (product.Image ?? "noimage.jpg");
        <div class="card m-3" style="min-width: 18rem; max-width:30.5%;">
            <div class="card-header">
                <h5><b>Nom : </b> @product.Name</h5>
                <h5><b>Prix : </b>@product.Price</h5>
                <h5><b>Quantité : </b> @product.QteStock</h5>
                <!--Afficher le nom de la catégorie-->
                <h5><b>Catégorie : @product.Category.CategoryName </b></h5>
            </div>
            <!-- Ajout de la classe pour contrôler la taille des images -->
            
            <div class="card-footer text-center">
                <a asp-controller="Product" asp-action="Details" asp-route-id="@product.ProductId"
                    class="btn btn-primary m-1">View</a>
                <a asp-action="Edit" asp-controller="Product" class="btn btn-primary m-1" asp-route-id="@product.ProductId">Edit</a>
                <a asp-action="Delete" asp-controller="Product" class="btn btn-danger m-1" asp-route-id="@product.ProductId">Delete</a>
            </div>
        </div>
    }
</div>
```



NOUVEL AFFICHAGE DE LA LISTE

- Pour ajuster la taille des images, ajouter la classe suivante dans le fichier **site.css** :



```
.card-img-top {  
    width: 100%;  
    height: 200px; /* Vous pouvez ajuster la hauteur selon votre préférence */  
    object-fit: contain; /* Coupe et ajuste l'image pour éviter la distorsion */  
    border-radius: 5px; /* Coins arrondis pour un meilleur rendu */  
}
```



PROGRAMMER LA RECHERCHE

- Apporter les modifications nécessaires pour le bouton « **search** » du **NavBar** pour effectuer une recherche soit par nom du produit ou le nom de la catégorie comme suit :
- Au niveau du contrôleur « ProductController » :

```
public ActionResult Search(string val)
{
    var result = ProductRepository.FindByName(val);

    return View("Index", result);
}
```

- Et au niveau du fichier « _Layout.cshtml »

```
<form class="d-flex" asp-area="" asp-controller="Product" asp-action="Search">
    <input class="form-control me-sm-2" type="search" placeholder="Search" name="val">
    <button class="btn btn-secondary my-2 my-sm-0" type="submit">Search</button>
</form>
```

