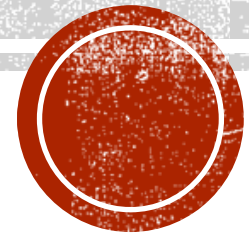


# **TP N°2**

**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 interface. On the left, the 'Parcourir' (Browse) tab is active, showing a search for '.NET' packages. Three packages are listed and highlighted with red boxes:

- Microsoft.EntityFrameworkCore.SqlServer** (6.0.3): Microsoft SQL Server database provider for Entity Framework Core.
- Microsoft.EntityFrameworkCore** (6.0.3): 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.
- Microsoft.EntityFrameworkCore.Tools** (6.0.3): 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 is set to 'Dernière version stable 6.0.3', and the 'Installer' button is visible. The description states: 'Microsoft SQL Server database provider for Entity Framework Core.' The version is 6.0.3, the author is Microsoft, the license is MIT, and the publication date is mardi 8 mars 2022 (08/03/2022).



# 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=ProductDB;Trusted_Connection=true"  
}
```



# CRÉATION DES CLASSES DU DOMAINE

- Commençons par créer la classe suivante dans le dossier **Models** :

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

- 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; }  
    }  
}
```



# CRÉATION DES CLASSES DU REPOSITORY

- Créer un dossier **Repositories** sous le dossier **Models**.
- Créer l'interface **IRepository** suivante :

```
public interface IRepository<T>
{
    T Get(int Id);
    IEnumerable<T> GetAll();
    T Add(T t);
    T Update(T t);
    T Delete(int Id);
}
```

- Créer maintenant la classe **SqlProductRepository** dans laquelle on va ajouter toutes les méthodes d'accès à la base de donnée. Cette classe implémente l'interface **IRepository** :



# CRÉATION DES CLASSES DU REPOSITORY

```
public class SqlProductRepository : IRepository<Product>
{
    private readonly AppDbContext context;
    public SqlProductRepository(AppDbContext context)
    {
        this.context = context;
    }
    public Product Add(Product P)
    {
        context.Products.Add(P);
        context.SaveChanges();
        return P;
    }
    public Product Delete(int Id)
    {
        Product P = context.Products.Find(Id);
        if (P != null)
        {
            context.Products.Remove(P);
            context.SaveChanges();
        }
        return P;
    }
    public IEnumerable<Product> GetAll()
    {
        return context.Products;
    }
}
```

```
public Product Get(int Id)
{
    return context.Products.Find(Id);
}
public Product Update(Product P)
{
    var Product =
        context.Products.Attach(P);
    Product.State = EntityState.Modified;
    context.SaveChanges();
    return P;
}
}
```

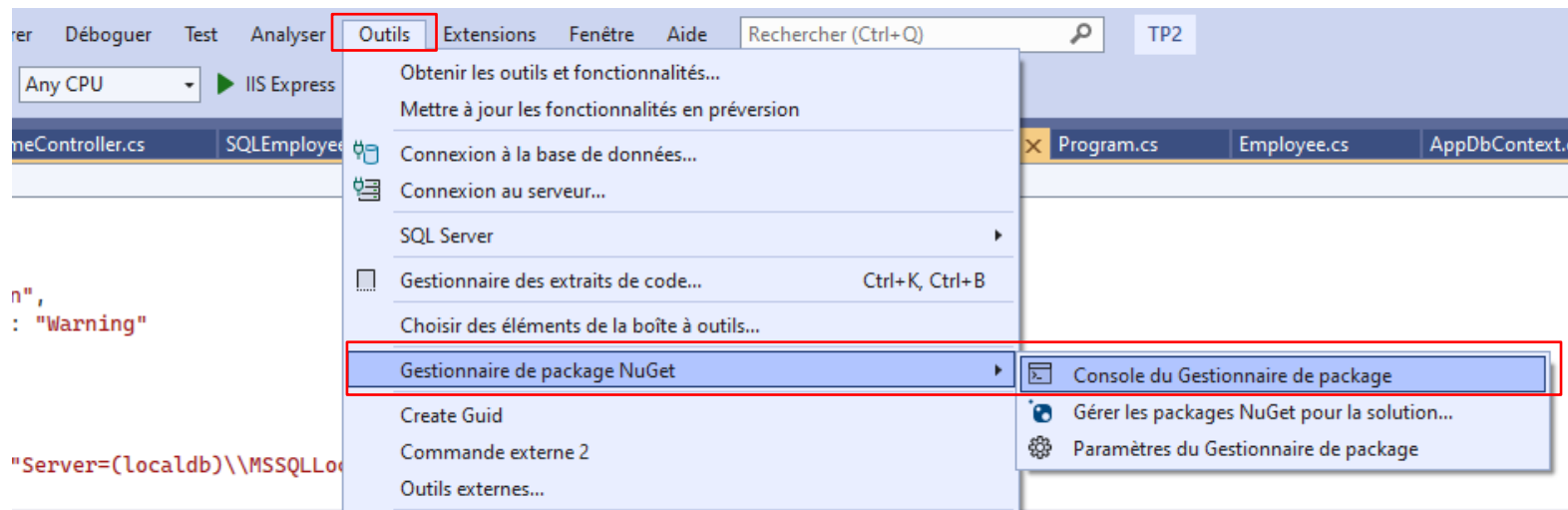


# INJECTION DES DÉPENDANCES ET CRÉATION DE LA BD

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

```
builder.Services.AddScoped<IRepository<Product>, SqlProductRepository>();
```

- 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)

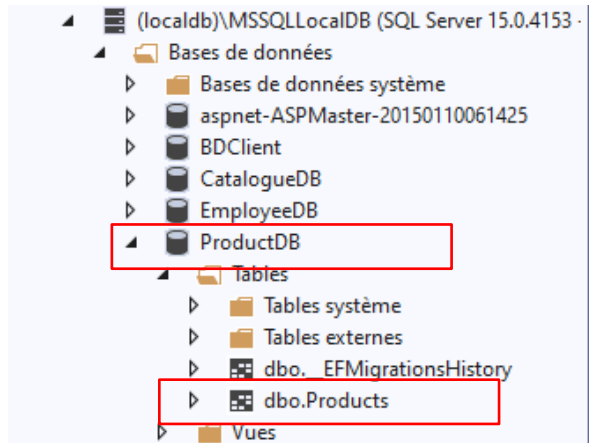
- La commande suivante crée la migration initiale. InitialCreate est le nom de la migration.

***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 la table Product.





# CRÉER UN CONTRÔLEUR

- Dans le dossier Controllers, Créer un nouveau contrôleur nommé **ProductController** (avec read/write actions) permettant de gérer les opérations sur les différents produits. Compléter le code des méthodes d'action en se basant sur le TP N°1.

```
public class ProductController : Controller
{
    //injection de dépendance
    readonly IRepository<Product> ProductRepository;

    0 références
    public ProductController(IRepository<Product> ProdRepository)
    {
        ProductRepository = ProdRepository;
    }

    // GET: ProductController
    3 références
    public ActionResult Index()
    {
        var Products = ProductRepository.GetAll();
        return View(Products);
    }

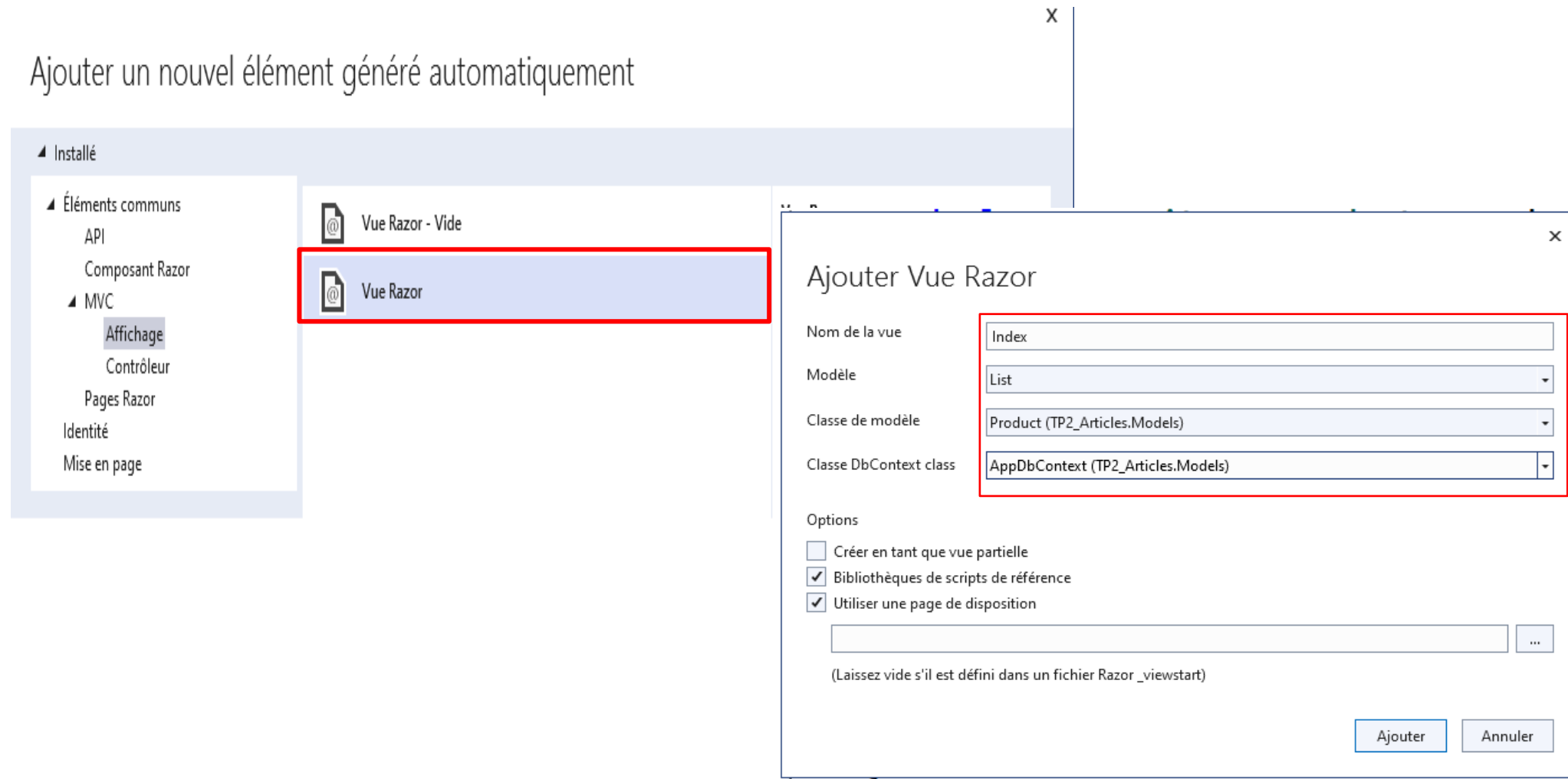
    //Compléter le code ...
}
```



# 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 :

Ajouter un nouvel élément généré automatiquement



# 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 Id { get; set; }

    [Required]
    [StringLength(50, MinimumLength = 5)]
    public string Désignation { get; set; }

    [Required]
    [Display(Name = "Prix en dinar :")]
    public float Prix { get; set; }

    [Required]
    [Display(Name = "Quantité en unité :")]
    public int Quantite { 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 il faut 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**.
- Ensuite, créer la classe **CreateViewModel** suivante :

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



# UPLOAD FILE

- Nous allons passer à modifier la méthode d'action Create du contrôleur.

```
public class ProductController : Controller
{
    //injection de dépendance
    readonly IRepository<Product> ProductRepository;
    private readonly IWebHostEnvironment hostingEnvironment;
    public ProductController(IRepository<Product> ProdRepository, IWebHostEnvironment hostingEnvironment)
    {
        ProductRepository = ProdRepository;
        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");

                // 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));
            }
        }
    }
}
```

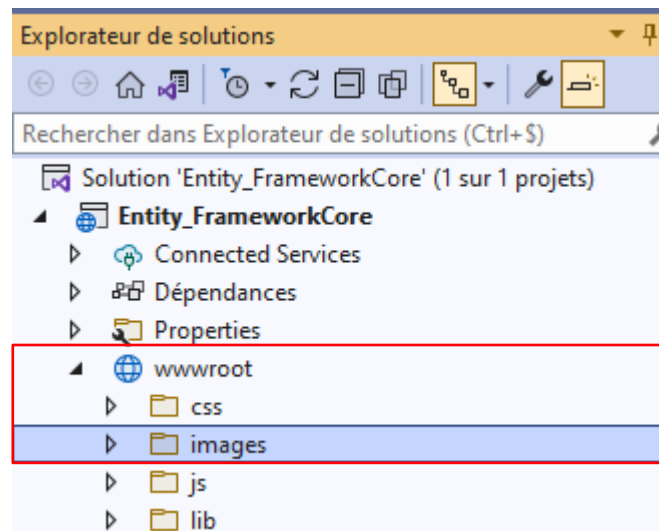


# UPLOAD FILE

```
Product newProduct = new Product
{
    Désignation = model.Désignation,
    Prix = model.Prix,
    Quantite = model.Quantite,
    // 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.Id });
}
return View();
}
```

- Ajouter un nouveau sous dossier nommé **images** dans le dossier **wwwroot** .



# UPLOAD FILE

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

```
@model TP2_Articles.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="Désignation" class="control-label"></label>
                <input asp-for="Désignation" class="form-control" />
                <span asp-validation-for="Désignation" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Prix" class="control-label"></label>
                <input asp-for="Prix" class="form-control" />
                <span asp-validation-for="Prix" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Quantite" class="control-label"></label>
                <input asp-for="Quantite" class="form-control" />
                <span asp-validation-for="Quantite" class="text-danger"></span>
            </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 () {
            $('.custom-file-input').on("change", function () {
                var fileName = $(this).val().split("\\").pop();
                $(this).next('.custom-file-label').html(fileName);
            });
        });
    </script>
}
```



# UPLOAD FILE

@model TP2\_Articles.Models.Product

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

```
@{
    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.Désignation)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Désignation)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Prix)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Prix)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Quantite)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Quantite)
        </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?.Id">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 Id { 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)
{
    Product product = ProductRepository.Get(id);
    EditViewModel productEditViewModel = new EditViewModel
    {
        Id = product.Id,
        Désignation = product.Désignation,
        Prix = product.Prix,
        Quantite = product.Quantite,
        ExistingImagePath = product.Image
    };
    return View(productEditViewModel);
}
```



# MODIFICATION D'UN PRODUIT EXISTANT

```
// POST: ProductController/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(EditViewModel model)
{
    // 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.Get(model.Id);
        // Update the product object with the data in the model object
        product.Désignation = model.Désignation;
        product.Prix = model.Prix;
        product.Quantite = model.Quantite;

        // 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

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

```
@model TP2_Articles.ViewModels.EditViewModel
@{
    ViewBag.Title = "Edit Product";
    // Get the full path of the existing product photo for display
    var photoPath = "~/images/" + (Model.ExistingImagePath ?? "noimage.jpg");
}

<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 employee id and ExistingPhotoPath
    which we need when we submit the form and update data in the database*@
    <input hidden asp-for="Id" />
    <input hidden asp-for="ExistingImagePath" />

    @*Bind to the properties of the EmployeeEditViewModel. The asp-for tag helper
    takes care of displaying the existing data in the respective input elements*@
    <div class="form-group row">
        <label asp-for="Désignation" class="col-sm-2 col-form-label"></label>
        <div class="col-sm-10">
            <input asp-for="Désignation" class="form-control" placeholder="Désignation">
            <span asp-validation-for="Désignation" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group row">
        <label asp-for="Prix" class="col-sm-2 col-form-label"></label>
        <div class="col-sm-10">
            <input asp-for="Prix" class="form-control" placeholder="Prix">
            <span asp-validation-for="Prix" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group row">
        <label asp-for="Quantite" class="col-sm-2 col-form-label"></label>
        <div class="col-sm-10">
            <input asp-for="Quantite" class="form-control" placeholder="Quantité">
            <span asp-validation-for="Quantite" class="text-danger"></span>
        </div>
    </div>
</div>
```



# 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">
      <label class="custom-file-label">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);
        $('#imageEmp').attr("src", "/" + fileName);
      });
    });
  </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 est le suivant :

```
@model IEnumerable<Product>
@{
    ViewBag.Title = "Liste des articles";
}
<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>Désignation : </b> @product.Désignation</h5>
                <h5><b>Prix : </b>@product.Prix</h5>
                <h5><b>Quantité : </b> @product.Quantite</h5>
            </div>

            <div class="card-footer text-center">
                <a asp-controller="Product" asp-action="Details" asp-route-id="@product.Id"
                    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.Id">Edit</a>
                <a asp-action="Delete" asp-controller="Product" class="btn btn-danger m-1" asp-route-id="@product.Id">Delete</a>
            </div>
        </div>
    }
</div>
```





# NOUVEL AFFICHAGE DE LA LISTE


- En exécutant, la liste des employés aura la forme suivante. Ajouter le code nécessaire dans le fichier « **\_layout.cshtml** » pour ajouter un élément dans le Navbar « **Nouvel Employé** » permettant d'exécuter l'action **Create** du contrôleur :

[Gestion d'articles](#) [Home](#) [Liste Articles ▾](#) [About](#)

Désignation  
Prix : 150  
Quantité : 10

Liste Articles

Nouvel Article




View

Edit

Delete

Désignation : Chaise  
Prix : 58  
Quantité : 3




View

Edit

Delete

Désignation : Canapé  
Prix : 450  
Quantité : 1



View

Edit

Delete

