UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

ASP.NET MVC

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

1. Introduction à ASP.NET MVC

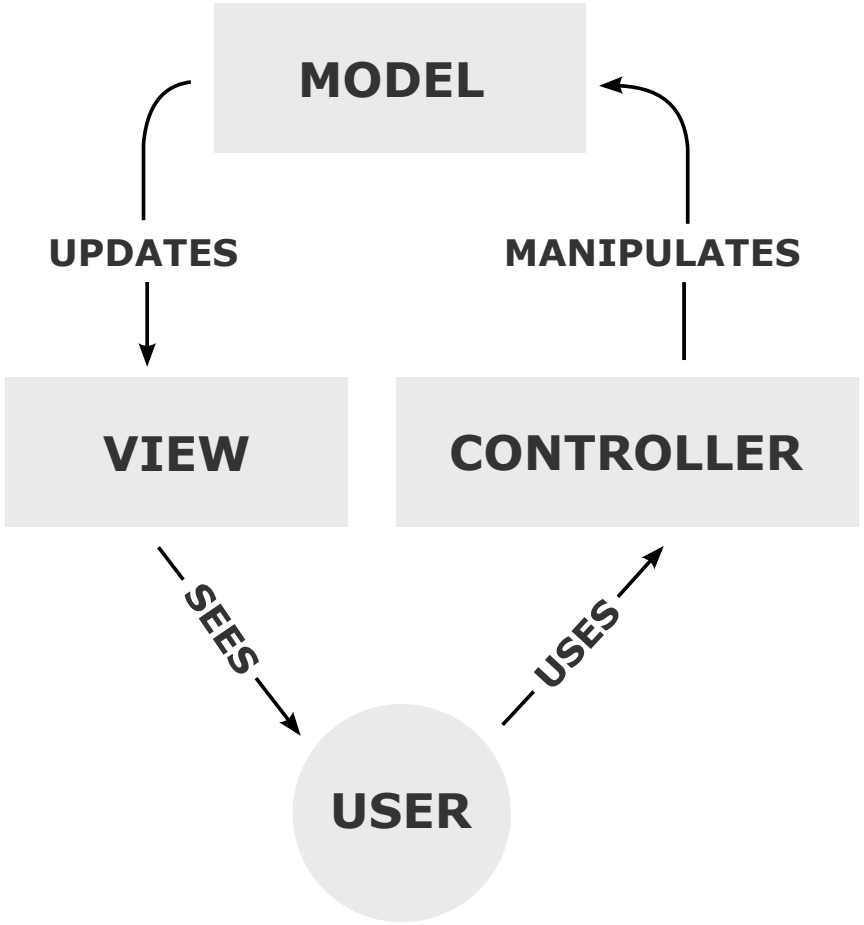
Qu'est-ce qu'ASP.NET MVC ?

- **ASP.NET MVC** est un framework pour le développement d'applications web basé sur l'architecture **Model-View-Controller (MVC)**.
- Il permet de séparer la logique de l'application en trois couches principales :
  - **Modèle (Model)** : Gestion des données et des règles métiers.
  - **Vue (View)** : Interface utilisateur.
  - **Contrôleur (Controller)** : Gestion des interactions utilisateur.

Comparaison avec Web Forms

Aspect	ASP.NET MVC	ASP.NET Web Forms
Architecture	Basé sur MVC (séparation claire).	Basé sur événements (Postbacks).
Flexibilité	Contrôle total sur le HTML.	Génération automatique du HTML.
Performances	Plus léger et rapide.	Consomme plus de ressources.
Testabilité	Facilite les tests unitaires.	Tests plus complexes.

Architecture Model-View-Controller



Avantages :

- Séparation des préoccupations (SoC).
- Amélioration de la maintenabilité et de l'évolutivité.
- Plus grande testabilité du code.

Installation de l'environnement de développement

Prérequis :

1. **Visual Studio** (ou VS Code avec les extensions nécessaires).
2. **.NET SDK** (téléchargé depuis dotnet.microsoft.com).
3. **SQL Server** (facultatif pour la base de données).

Installation de Visual Studio

1. Téléchargez Visual Studio : https://visualstudio.microsoft.com/.
2. Pendant l'installation :
   - Sélectionnez le **workload** : **ASP.NET and web development**.
   - Installez également l'outil de gestion des bases de données si nécessaire.

Commandes CLI pour configurer l'environnement

Vérification de l'installation de .NET :

```
dotnet --version
```

Création d'un projet ASP.NET MVC :

```
dotnet new mvc -n MonPremierProjetMVC
```

Lancement du projet :

```
cd MonPremierProjetMVC
dotnet run
```

Démarrage de l'application

1. Lancez le projet depuis Visual Studio avec `Ctrl + F5` ou utilisez la commande `dotnet run`.
2. Par défaut, le projet affiche une page d'accueil générée automatiquement :
   - URL : `https://localhost:5001` ou `http://localhost:5000`.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

2. Premier projet ASP.NET MVC

Création d'un projet MVC

Commande CLI pour créer un projet :

```
dotnet new mvc -n MonPremierProjetMVC
```

Structure du projet :

Une fois créé, le projet contiendra les éléments principaux suivants :

- `Controllers/` : Contient les fichiers des contrôleurs.
- `Views/` : Contient les fichiers des vues.
- `Models/` : Contient les fichiers des modèles.
- `Program.cs` : Point d'entrée de l'application.

Structure d'un projet ASP.NET MVC

Voici une vue d'ensemble des dossiers et fichiers clés dans un projet :

```
MonPremierProjetMVC/
├── Controllers/
│   └── HomeController.cs
├── Views/
│   ├── Shared/
│   └── Home/
│       └── Index.cshtml
├── Models/
├── wwwroot/
│   ├── css/
│   ├── js/
│   └── lib/
└── Program.cs
```

Le fichier Program.cs

Dans .NET 8, il remplace `Startup.cs`. Voici un exemple de configuration minimaliste :

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
```

Création de la page d'accueil

Etape 1 : Modifier le contrôleur

Dans `Controllers/HomeController.cs`, remplacez l'action `Index` par :

```
public IActionResult Index()
{
    ViewData["Message"] = "Bienvenue dans votre première application ASP.NET MVC !";
    return View();
}
```

Création de la page d'accueil

Étape 2 : Modifier la vue

Dans `Views/Home/Index.cshtml`, ajoutez :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Accueil</title>
  </head>
  <body>
    <h1>@ViewData["Message"]</h1>
  </body>
</html>
```

Exécution du projet

1. Lancez l'application avec la commande :

```
dotnet run
```

2. Par défaut, l'application sera accessible sur :
   - `http://localhost:5000`
   - `https://localhost:5001`
3. Vous devriez voir votre message personnalisé :
   "Bienvenue dans votre première application ASP.NET MVC !"

Points importants de configuration

- **Fichiers statiques** :
  Les fichiers dans le dossier `wwwroot` (CSS, JS, images) sont servis automatiquement.

- **Routing par défaut** :
  Le routage est configuré dans `Program.cs`. Le contrôleur par défaut est `Home` et l'action par défaut est `Index`.

Points importants de configuration

- **Personnalisation** :
  Vous pouvez changer le contrôleur ou l'action par défaut dans la ligne suivante :

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

3. Les contrôleurs (Controllers)

Qu'est-ce qu'un contrôleur ?

Un contrôleur est une classe qui gère les requêtes entrantes, traite les données et retourne une réponse au client (une vue ou un résultat JSON, par exemple).

- Les contrôleurs sont placés dans le dossier `Controllers`.
- Par convention, le nom des classes de contrôleurs se termine par `Controller`.

Exemple de contrôleur :

```
using Microsoft.AspNetCore.Mvc;

public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    public IActionResult About()
    {
        return Content("Ceci est une page 'À propos'.");
    }
}
```

Créer un contrôleur personnalisé

Étapes pour créer un contrôleur :

1. Ajoutez une classe dans le dossier `Controllers`.
2. Héritez de `Controller`.
3. Ajoutez des actions (méthodes publiques retournant `IActionResult`).

Exemple de Controller personnalisé

```
using Microsoft.AspNetCore.Mvc;

public class ProductController : Controller
{
    public IActionResult List()
    {
        return Content("Liste des produits.");
    }

    public IActionResult Details(int id)
    {
        return Content($"Détails du produit avec ID : {id}");
    }
}
```

Retourner une vue avec des données

Pour passer des données à une vue, utilisez :

- `ViewData` : dictionnaire temporaire pour transmettre des données.
- `ViewBag` : wrapper dynamique pour `ViewData`.

Exemple :

```
public IActionResult Index()
{
    ViewData["Message"] = "Bienvenue dans le contrôleur Home.";
    ViewBag.Date = DateTime.Now;
    return View();
}
```

Vue correspondante

La vue `Views/Home/Index.cshtml` :

```
<h1>@ViewData["Message"]</h1>
<p>Date : @ViewBag.Date</p>
```

Gestion des paramètres dans les actions

Les actions de contrôleurs peuvent recevoir des paramètres via :

- **L'URL** (RouteData).
- **La chaîne de requête** (QueryString).

Exemple :

```
public IActionResult Details(int id)
{
    return Content($"Détails pour l'ID : {id}");
}
```

- `http://localhost:5000/Product/Details/5` retourne "Détails pour l'ID : 5".

Redirection entre actions

Vous pouvez rediriger vers une autre action ou un autre contrôleur :

- Redirection vers une autre action dans le même contrôleur :

```
return RedirectToAction("About");
```

- Redirection vers une action dans un autre contrôleur :

```
return RedirectToAction("List", "Product");
```

- Redirection vers une URL spécifique :

```
return Redirect("https://example.com");
```

4. Les vues (Views)

Qu'est-ce qu'une vue ?

Une vue est un composant de l'interface utilisateur dans ASP.NET MVC. Elle est responsable de la présentation des données envoyées par le contrôleur à l'utilisateur. Les vues sont écrites en **Razor**, une syntaxe permettant de combiner du HTML avec du code C#.

Création d'une vue

Les vues sont situées dans le dossier `Views`, organisées par sous-dossiers correspondant aux contrôleurs.

Exemple : Pour un contrôleur `HomeController`, les vues associées seront dans `Views/Home`.

Création de la vue

Pour créer une vue associée à l'action `Index` :

1. Ajoutez un fichier `Index.cshtml` dans le dossier `Views/Home`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Accueil</title>
  </head>
  <body>
    <h1>Bienvenue sur la page d'accueil</h1>
  </body>
</html>
```

Passer des données à une vue

Pour transmettre des données du contrôleur à la vue, vous pouvez utiliser :

- **ViewData** : un dictionnaire clé-valeur.
- **ViewBag** : une propriété dynamique.
- **Un modèle (Model)** : une instance d'objet passée directement.

Exemple avec ViewData

```
public IActionResult Index()
{
    ViewData["Message"] = "Bienvenue sur ma page.";
    return View();
}
```

Vue correspondante (`Index.cshtml`) :

```
<h1>@ViewData["Message"]</h1>
```

Présentation de Razor

Razor est une syntaxe légère permettant de mélanger du HTML avec du code C#.

Les principales caractéristiques de Razor sont :

- Le code C# est précédé par **@**.
- Les blocs de code sont entourés par `{}`.
- Les expressions Razor s'insèrent directement dans le HTML.

Exemple simple :

```
<p>La date et l'heure actuelles sont : @DateTime.Now</p>
```

Variables et expressions

Les variables peuvent être définies dans Razor et utilisées directement dans le HTML :

```
@{ var message = "Bienvenue sur mon site !"; }
<p>@message</p>
```

Les expressions Razor insèrent directement le résultat de l'exécution du code :

```
<p>5 + 3 = @(5 + 3)</p>
```

Affichage conditionnel

Razor permet d'écrire des conditions pour afficher ou non certains éléments HTML.

```
@{ bool isAdmin = true; } @if (isAdmin) {
<p>Bienvenue, administrateur !</p>
} else {
<p>Bienvenue, utilisateur !</p>
}
```

Boucle foreach

Les boucles `for`, `foreach` et `while` sont utilisables avec Razor :

```
@{ var items = new[] { "Article 1", "Article 2", "Article 3" }; }
<ul>
  @foreach (var item in items) {
  <li>@item</li>
  }
</ul>
```

Boucle for

`for` avec un index :

```
<ul>
  @for (int i = 1; i <= 5; i++) {
  <li>Élément numéro @i</li>
  }
</ul>
```

Inclusion de blocs de code

Vous pouvez inclure des blocs de code Razor dans le HTML pour effectuer des opérations complexes.

```
@{ var somme = 0; for (int i = 1; i <= 5; i++) { somme += i; } }
<p>La somme des nombres de 1 à 5 est : @somme</p>
```

Appel de méthodes C#

Vous pouvez appeler directement des méthodes C# dans Razor :

```
<p>Longueur de la chaîne : @("Hello".Length)</p>
<p>Résultat d'une méthode : @Math.Pow(2, 3)</p>
```

Si vous travaillez avec des méthodes personnalisées :

```
@functions {
    string ObtenirMessage()
    {
        return "Bonjour depuis Razor !";
    }
}
<p>@ObtenirMessage()</p>
```

Encodage HTML automatique

Razor encode automatiquement les données pour éviter les attaques XSS.

Par exemple :

```
@{ var texte = "
<script>
  alert("XSS");
</script>
"; }
<p>@texte</p>
```

Affichera :

```
<p>&lt;script&gt;alert('XSS');&lt;/script&gt;</p>
```

Désactiver l'encodage

Pour désactiver l'encodage, utilisez `Html.Raw` :

```
<p>@Html.Raw(texte)</p>
```

Affichage de blocs de texte

Pour afficher de longs blocs de texte ou de HTML, utilisez des blocs `text`.

```
@{ var contenu = @"
<p>Ceci est un paragraphe.</p>
<ul>
  <li>Élément 1</li>
  <li>Élément 2</li>
</ul>
"; } @Html.Raw(contenu)
```

Layouts : Vues réutilisables

Un layout est un modèle de vue réutilisable pour définir une structure commune (comme un en-tête et un pied de page).

1. Créez un fichier `_Layout.cshtml` dans `Views/Shared`.

Layouts : Vues réutilisables

2. Ajoutez-y une structure de base :

```
<!DOCTYPE html>
<html>
  <head>
    <title>@ViewData["Title"]</title>
  </head>
  <body>
    <header>
      <h1>Mon site</h1>
    </header>
    <main>@RenderBody()</main>
  </body>
</html>
```

Layouts : Vues réutilisables

3. Associez ce layout dans les autres vues :

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Vues partielles

Les vues partielles permettent de réutiliser des fragments d'interface utilisateur.
Exemple :

1. Créez une vue partielle `_Menu.cshtml` dans `Views/Shared`.
2. Ajoutez du contenu :

```
<ul>
  <li><a href="/Home/Index">Accueil</a></li>
  <li><a href="/Home/About">À propos</a></li>
</ul>
```

Vues partielles

3. Intégrez la vue partielle dans une autre vue :

```
@Html.Partial("_Menu")
```

5. Les modèles (Models)

Qu'est-ce qu'un modèle ?

Un modèle est une classe représentant la structure des données utilisées dans l'application.
Les modèles servent à :

- Transférer des données entre la couche de contrôle et la vue.
- Interagir avec une base de données (souvent via Entity Framework).

Exemple de modèle

Dans ASP.NET MVC, les modèles sont généralement situés dans le dossier `Models`.

Exemple simple de modèle :

```
public class Produit
{
    public int Id { get; set; }
    public string Nom { get; set; }
    public decimal Prix { get; set; }
}
```

Utilisation d'un modèle dans une vue

1. Créez un modèle :

```
public class Produit
{
    public int Id { get; set; }
    public string Nom { get; set; }
    public decimal Prix { get; set; }
}
```

Utilisation d'un modèle dans une vue

2. Passez une instance de ce modèle depuis le contrôleur :

```
public IActionResult Details()
{
    var produit = new Produit
    {
        Id = 1,
        Nom = "Ordinateur Portable",
        Prix = 999.99m
    };
    return View(produit);
}
```

Utilisation d'un modèle dans une vue

3. Ajoutez la directive `@model` dans la vue pour spécifier le type de données :

```
@model Produit

<h1>Détails du produit</h1>
<p>Nom : @Model.Nom</p>
<p>Prix : @Model.Prix €</p>
```

Modèles complexes

Un modèle peut contenir des propriétés complexes ou des collections.

```
public class Categorie
{
    public int Id { get; set; }
    public string Nom { get; set; }
    public List<Produit> Produits { get; set; }
}
```

Exemple d'utilisation

```
public IActionResult Index()
{
    var categorie = new Categorie
    {
        Id = 1,
        Nom = "Électronique",
        Produits = new List<Produit>
        {
            new Produit { Id = 1, Nom = "Téléphone", Prix = 699.99m },
            new Produit { Id = 2, Nom = "Tablette", Prix = 499.99m }
        }
    };
    return View(categorie);
}
```

Vue associée (Index.cshtml) :

```
@model Categorie

<h1>Catégorie : @Model.Nom</h1>
<ul>
  @foreach (var produit in Model.Produits) {
  <li>@produit.Nom - @produit.Prix €</li>
  }
</ul>
```

Validation des données

ASP.NET MVC prend en charge la validation des modèles à l'aide d'attributs de validation.

```
using System.ComponentModel.DataAnnotations;

public class Utilisateur
{
    public int Id { get; set; }
    [Required]
    public string Nom { get; set; }
    [EmailAddress]
    public string Email { get; set; }
    [Range(18, 99)]
    public int Age { get; set; }
}
```

Validation ASP.NET

depuis ASP.NET Core 3.0 (et toujours actif dans .NET 8.0), la validation des modèles est automatiquement gérée par un middleware.

Cela signifie que si les données envoyées au contrôleur ne passent pas les règles de validation (définies via des annotations comme `[Required]`, `[EmailAddress]`, etc.), une réponse avec un code **400 Bad Request** est automatiquement retournée.

Pouir personnaliser la gestion des erreurs ou enrichir les messages d'erreur, on peut utiliser le `ModelState`.

Validation dans le contrôleur

Pour valider un modèle dans le contrôleur, utilisez `ModelState`:

```
[HttpPost]
public IActionResult Creer(Utilisateur utilisateur)
{
    if (!ModelState.IsValid)
    {
        return View(utilisateur);
    }

    // Logique si le modèle est valide
    return RedirectToAction("Index");
}
```

Affichage des erreurs de validation dans la vue

Dans la vue, utilisez `Html.ValidationMessageFor` pour afficher les erreurs associées à un champ :

```
@model Utilisateur

<form method="post">
  <label>Nom :</label>
  <input asp-for="Nom" />
  <span asp-validation-for="Nom"></span>

  <label>Email :</label>
  <input asp-for="Email" />
  <span asp-validation-for="Email"></span>

  <button type="submit">Envoyer</button>
</form>
```

Gérer les erreurs globales

Pour afficher des erreurs globales, utilisez `Html.ValidationSummary`:

```
@Html.ValidationSummary()
```

6. Routage

Qu'est-ce que le routage ?

Le routage (routing) est le mécanisme par lequel ASP.NET MVC associe une URL à un contrôleur et à une action spécifiques.

Le **middleware de routage** permet de déterminer quelle action du contrôleur doit être exécutée en fonction de l'URL demandée.

Routage par défaut

Dans une application ASP.NET MVC, le routage par défaut est défini dans `Program.cs`. Il utilise un modèle de routage basé sur les conventions :

```
builder.Services.AddControllersWithViews();

builder.Services.AddRouting(options =>
{
    options.LowercaseUrls = true;
});

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Routage par défaut

Cela signifie que :

- L'URL `/Home/Index/1` sera envoyée à l'action `Index` du `HomeController` avec un paramètre `id` égal à `1`.
- Si aucun contrôleur n'est spécifié, le contrôleur `Home` sera utilisé par défaut.
- Si aucune action n'est spécifiée, l'action `Index` sera appelée.

Structure d'URL

Le modèle de routage par défaut est :

```
{controller}/{action}/{id?}
```

- `{controller}` : le nom du contrôleur (par exemple, `Home`).
- `{action}` : l'action à exécuter dans le contrôleur (par exemple, `Index`).
- `{id?}` : un paramètre optionnel, utilisé pour passer des données à l'action (par exemple, `id=5`).

Exemple d'URL :

```
/Product/Details/5
```

Personnalisation des routes

Vous pouvez personnaliser les routes en ajoutant des modèles de routage dans `Program.cs`.

Voici un exemple où vous définissez une route personnalisée pour les actions d'un contrôleur `Product` :

```
app.MapControllerRoute(
    name: "productDetails",
    pattern: "products/{id}/{name}",
    defaults: new { controller = "Product", action = "Details" });
```

Cela permettrait d'avoir une URL comme :

```
/products/5/phone
```

Routes attributs (Attribute Routing)

Au lieu de définir des routes globalement, vous pouvez utiliser des attributs sur les actions ou les contrôleurs pour spécifier une route. Cela permet plus de flexibilité.

Exemple avec des attributs sur un contrôleur

```
[Route("products")]
public class ProductController : Controller
{
    [Route("{id}")]
    public IActionResult Details(int id)
    {
        return View();
    }

    [Route("create")]
    public IActionResult Create()
    {
        return View();
    }
}
```

Exemple avec des attributs sur un contrôleur

Cela génère des URL comme :

- `/products/5` pour `Details`.
- `/products/create` pour `Create`.

Paramètres de route et contraintes

Vous pouvez définir des contraintes sur les paramètres de la route pour limiter les valeurs qu'ils peuvent prendre. Par exemple :

```
[Route("products/{id:int}")]
public IActionResult Details(int id)
{
    return View();
}
```

Ici, le paramètre `id` est contraint à être un entier (`int`). Si l'URL contient un autre type (comme une chaîne), une erreur 404 sera retournée.

Routes avec des paramètres optionnels

Les paramètres peuvent être rendus optionnels en ajoutant un point d'interrogation (`?`) à la fin du paramètre dans le modèle de routage. Par exemple :

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Ici, le paramètre `id` est optionnel. Si l'URL ne contient pas `id`, l'action sera exécutée avec la valeur par défaut.

Redirection et génération d'URL

Vous pouvez utiliser des méthodes comme `RedirectToAction`, `RedirectToRoute`, ou `Url.Action` pour générer des URL ou rediriger vers d'autres actions.

Redirection vers une action :

```
return RedirectToAction("Details", "Product", new { id = 5 });
```

Générer une URL :

```
string url = Url.Action("Details", "Product", new { id = 5 });
```

Cela génère une URL telle que `/Product/Details/5`.

Routage et sécurité

Il est important de noter que les routes peuvent être sécurisées avec des attributs comme `[Authorize]`, limitant l'accès à certaines actions selon les rôles ou les permissions des utilisateurs.

```
[Authorize(Roles = "Admin")]
public IActionResult AdminPage()
{
    return View();
}
```

7. Interactions avec la base de données

Introduction à Entity Framework Core

Entity Framework (EF) Core est un ORM (Object-Relational Mapper) qui permet d'interagir avec la base de données à travers des objets C#.

EF Core supporte plusieurs bases de données, telles que SQL Server, SQLite, PostgreSQL, MySQL, et bien d'autres.

Installation

1. **Installation des packages nécessaires**
   Pour utiliser EF Core, il faut installer les packages appropriés. Dans le cas de SQL Server :

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

Configuration

2. **Configurer la chaîne de connexion**

   La chaîne de connexion est généralement stockée dans le fichier `appsettings.json` :

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyAppDB;Trusted_Connection=True;"
  }
}
```

Connexion

3. **Configurer le DbContext dans `Program.cs`**

   Le `DbContext` est la classe qui gère les opérations de base de données. Voici comment le configurer dans `Program.cs` :

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Création du modèle de données

1. **Définition des modèles**
   Crée des classes qui représentent les entités de ta base de données. Par exemple, une entité `Produit` :

```
public class Produit
{
    public int Id { get; set; }
    public string Nom { get; set; }
    public decimal Prix { get; set; }
}
```

DbContext

2. **Création du DbContext**

   Le `DbContext` représente une session avec la base de données et contient des propriétés de type `DbSet<T>`.

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options) { }

    public DbSet<Produit> Produits { get; set; }
}
```

Migration et mise à jour de la base de données

1. **Ajout d'une migration**
   Les migrations permettent de mettre à jour la base de données en fonction des changements apportés au modèle.
   Pour ajouter une migration, utilise la commande suivante :

```
dotnet ef migrations add InitialCreate
```

2. **Mise à jour de la base de données**

```
dotnet ef database update
```

Interactions avec la base de données

1. **Ajouter des données**

   Pour ajouter une entité à la base de données, crée une instance de l'entité et utilise `Add` ou `AddAsync` :

```
public class ProductController : Controller
{
    private readonly ApplicationDbContext _context;

    public ProductController(ApplicationDbContext context)
    {
        _context = context;
    }
}
```

Méthode de création

```
[HttpPost]
public async Task<IActionResult> Create(Produit produit)
{
    if (ModelState.IsValid)
    {
        _context.Add(produit);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(produit);
}
```

Lecture des données

2. Lire des données

Utilise `DbSet<T>` pour récupérer les entités de la base de données :

```
public async Task<IActionResult> Index()
{
    var produits = await _context.Produits.ToListAsync();
    return View(produits);
}
```

Mise à jour des données

3. **Mettre à jour des données**

   Pour mettre à jour un enregistrement existant, récupère d'abord l'entité, modifie-la, puis appelle `SaveChangesAsync` :

```
public async Task<IActionResult> Edit(int id)
{
    var produit = await _context.Produits.FindAsync(id);
    if (produit == null)
    {
        return NotFound();
    }
    return View(produit);
}
```

Méthode d'édition

```
[HttpPost]
public async Task<IActionResult> Edit(int id, Produit produit)
{
    if (id != produit.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        _context.Update(produit);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(produit);
}
```

Méthode de suppression

4. **Supprimer des données**

Pour supprimer une entité, utilise `Remove` et `SaveChangesAsync` :

```
public async Task<IActionResult> Delete(int id)
{
    var produit = await _context.Produits.FindAsync(id);
    if (produit == null)
    {
        return NotFound();
    }

    _context.Produits.Remove(produit);
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Requêtes LINQ avec Entity Framework

EF Core permet de réaliser des requêtes complexes en utilisant LINQ. Par exemple, pour obtenir tous les produits dont le prix est supérieur à 100€ :

```
var produits = await _context.Produits
    .Where(p => p.Prix > 100)
    .ToListAsync();
```

Tu peux également utiliser d'autres opérations LINQ comme `Select`, `OrderBy`, `FirstOrDefault`, etc.

Transactions

Si tu as besoin d'effectuer plusieurs opérations de base de données dans une même transaction, utilise les transactions avec EF Core :

```
using var transaction = await _context.Database.BeginTransactionAsync();

try
{
    // Effectuer des opérations
    await _context.SaveChangesAsync();

    // Valider la transaction
    await transaction.CommitAsync();
}
catch
{
    // Annuler la transaction en cas d'erreur
    await transaction.RollbackAsync();
    throw;
}
```

Lazy Loading et Eager Loading

1. **Lazy Loading** : EF Core charge les données de manière paresseuse, c'est-à-dire qu'il charge les entités liées uniquement lorsque cela est nécessaire.
   - Pour activer le lazy loading, il faut installer le package `Microsoft.EntityFrameworkCore.Proxies` et configurer les propriétés de navigation comme `virtual`.
2. **Eager Loading** : Cette approche charge les données liées dans une seule requête en utilisant `Include` :

Lazy Loading et Eager Loading

```
var produit = await _context.Produits
    .Include(p => p.Categorie)
    .FirstOrDefaultAsync(p => p.Id == id);
```

8. Gestion des formulaires

Introduction à la gestion des formulaires

Dans ASP.NET MVC, les formulaires permettent à l'utilisateur d'interagir avec l'application en envoyant des données au serveur. Ces données sont généralement envoyées via une requête HTTP POST et sont utilisées pour effectuer des opérations telles que la création, la mise à jour ou la suppression d'entités.

ASP.NET MVC simplifie la gestion des formulaires en utilisant des **modèles** (Models) et en exploitant les **tag helpers** Razor pour lier les champs de formulaire aux propriétés du modèle.

Création d'un formulaire de création

Pour créer un formulaire qui envoie des données au serveur, nous utilisons la syntaxe Razor dans une vue `.cshtml`. Voici un exemple d'un formulaire pour créer un produit :

```
@model Produit

<form asp-action="Create" method="post">
  <div>
    <label for="Nom">Nom</label>
    <input type="text" id="Nom" name="Nom" asp-for="Nom" />
  </div>
  <div>
    <label for="Prix">Prix</label>
    <input type="number" id="Prix" name="Prix" asp-for="Prix" />
  </div>
  <button type="submit">Créer</button>
</form>
```

Récupération du modèle dans le controlleur

Le contrôleur recevra les données envoyées par le formulaire via la méthode POST. Voici un exemple avec la méthode `Create` :

```
[HttpPost]
public IActionResult Create(Produit produit)
{
    if (ModelState.IsValid)
    {
        _context.Produits.Add(produit);
        _context.SaveChanges();
        return RedirectToAction(nameof(Index));
    }
    return View(produit);
}
```

Validation des données de formulaire

ASP.NET MVC permet d'ajouter des règles de validation sur les propriétés du modèle en utilisant des **annotations de données**. Ces annotations seront automatiquement appliquées lors de la soumission du formulaire.

Utilisation du model validation

Utiliser des attributs comme `[Required]`, `[StringLength]`, `[Range]`, etc. pour valider les données entrées :

```
public class Produit
{
    public int Id { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "Le nom ne peut dépasser 100 caractères.")]
    public string Nom { get; set; }

    [Required]
    [Range(0.01, double.MaxValue, ErrorMessage = "Le prix doit être supérieur à zéro.")]
    public decimal Prix { get; set; }
}
```

Validation dans la vue

Lors de l'affichage du formulaire, les erreurs de validation seront automatiquement affichées si le modèle n'est pas valide. Utiliser `asp-validation-for` pour afficher les messages d'erreur :

```
<form asp-action="Create" method="post">
  <div>
    <label for="Nom">Nom</label>
    <input type="text" id="Nom" name="Nom" asp-for="Nom" />
    <span asp-validation-for="Nom"></span>
  </div>
  <div>
    <label for="Prix">Prix</label>
    <input type="number" id="Prix" name="Prix" asp-for="Prix" />
    <span asp-validation-for="Prix"></span>
  </div>
  <button type="submit">Créer</button>
</form>
```

Affichage des erreurs

Utiliser `@Html.ValidationSummary()` pour afficher un résumé des erreurs de validation en haut du formulaire :

```
@Html.ValidationSummary(true)
```

Formulaires avec des données complexes

Dans le cas de modèles plus complexes, comme des relations entre plusieurs entités, on peut utiliser des formulaires imbriqués. Par exemple, si un `Produit` a une `Catégorie` associée, on peut avoir un formulaire pour gérer cette relation.

Validation d'un modèle avec relation

Voici un exemple de modèle avec une relation entre `Produit` et `Categorie` :

```
public class Produit
{
    public int Id { get; set; }

    [Required]
    public string Nom { get; set; }

    [Required]
    public decimal Prix { get; set; }

    public int CategorieId { get; set; }
    public Categorie Categorie { get; set; }
}
```

```
public class Categorie
{
    public int Id { get; set; }
    public string Nom { get; set; }
}
```

Vue avec formulaire imbriqué

Voici un formulaire pour un `Produit` avec une liste déroulante pour sélectionner une `Catégorie` :

```
@model Produit

<form asp-action="Create" method="post">
    <div>
        <label for="Nom">Nom</label>
        <input type="text" id="Nom" name="Nom" asp-for="Nom" />
    </div>
    <div>
        <label for="Prix">Prix</label>
        <input type="number" id="Prix" name="Prix" asp-for="Prix" />
    </div>
    <div>
        <label for="CategorieId">Catégorie</label>
        <select id="CategorieId" asp-for="CategorieId" asp-items="@(new SelectList(ViewBag.Categories, "Id", "Nom"))"></select>
    </div>
    <button type="submit">Créer</button>
</form>
```

Contrôleur pour gérer les relations

Dans le contrôleur, ont peut préparer la liste des catégories à afficher dans le formulaire :

```
public IActionResult Create()
{
    ViewBag.Categories = _context.Categories.ToList();
    return View();
}
```

Utilisation de Tag Helpers

ASP.NET MVC fournit des **Tag Helpers** pour simplifier l'écriture des formulaires. Ceux-ci permettent de lier facilement des propriétés de modèle à des éléments HTML sans avoir à écrire beaucoup de code.

Exemple de formulaire avec des Tag Helpers

```
<form asp-action="Create" method="post">
  <div class="form-group">
    <label asp-for="Nom" class="control-label"></label>
    <input asp-for="Nom" class="form-control" />
    <span asp-validation-for="Nom" 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>
  <button type="submit" class="btn btn-primary">Créer</button>
</form>
```

Le **Tag Helper** `asp-for` lie un champ du modèle à un élément de formulaire, et `asp-validation-for` affiche les messages de validation.

Formulaires asynchrones avec AJAX

Pour éviter le rechargement de la page lors de l'envoi du formulaire, l'utilisation d'AJAX avec ASP.NET MVC permet d'envoyer les données du formulaire de manière asynchrone.

AJAX avec JQuery

Exemple d'utilisation d'AJAX avec jQuery pour envoyer le formulaire :

```
$(document).ready(function () {
  $("#createForm").submit(function (event) {
    event.preventDefault();
    $.ajax({
      url: '@Url.Action("Create", "Product")',
      type: "POST",
      data: $(this).serialize(),
      success: function (response) {
        alert("Produit créé avec succès");
      },
      error: function () {
        alert("Erreur lors de la création du produit");
      },
    });
  });
});
```

9. Authentification et autorisation

Introduction à l'authentification et l'autorisation

L'**authentification** permet d'identifier un utilisateur, tandis que l'**autorisation** contrôle l'accès à différentes parties de l'application en fonction des rôles ou des droits d'un utilisateur.

ASP.NET Core fournit un système intégré pour gérer l'authentification et l'autorisation via des services comme **Identity**.

Configuration d'Identity

Le package `Microsoft.AspNetCore.Identity.EntityFrameworkCore` doit être ajouté au projet pour utiliser ASP.NET Core Identity. Cela peut être fait en utilisant la commande NuGet suivante dans le terminal :

```
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
```

Dans le fichier `Program.cs`, ajouter le service `AddIdentity` pour configurer la gestion des utilisateurs et des rôles :

```
builder.Services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

Modification du Program.cs

La configuration des options, telles que la gestion des cookies d'authentification, peut également être effectuée dans `Program.cs` :

```
builder.Services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = "/Account/Login";
    options.AccessDeniedPath = "/Account/AccessDenied";
});
```

Création d'un modèle d'utilisateur personnalisé

Un modèle utilisateur personnalisé peut être créé en étendant la classe `IdentityUser` :

```
public class ApplicationUser : IdentityUser
{
    public string FullName { get; set; }
}
```

Mise à jour de la base de données

Après avoir modifié ou ajouté des propriétés à `ApplicationUser`, une migration doit être ajoutée et la base de données mise à jour :

```
dotnet ef migrations add AddFullNameToApplicationUser
dotnet ef database update
```

Vue - Register

La vue d'enregistrement permet à l'utilisateur de s'inscrire. Elle contient des champs pour le nom, l'email, le mot de passe, etc. :

```
<form asp-action="Register" method="post">
  <div>
    <label for="Email">Email</label>
    <input type="email" asp-for="Email" />
  </div>
  <div>
    <label for="Password">Mot de passe</label>
    <input type="password" asp-for="Password" />
  </div>
  <button type="submit">S'inscrire</button>
</form>
```

Controller - Register

Le contrôleur gère l'enregistrement de l'utilisateur en créant un utilisateur dans la base de données :

```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
    var result = await _userManager.CreateAsync(user, model.Password);
    if (result.Succeeded)
    {
        await _signInManager.SignInAsync(user, isPersistent: false);
        return RedirectToAction("Index", "Home");
    }
    return View(model);
}
```

Vue - Login

La vue de connexion permet à l'utilisateur de saisir ses informations d'identification :

```
<form asp-action="Login" method="post">
  <div>
    <label for="Email">Email</label>
    <input type="email" asp-for="Email" />
  </div>
  <div>
    <label for="Password">Mot de passe</label>
    <input type="password" asp-for="Password" />
  </div>
  <button type="submit">Se connecter</button>
</form>
```

Controller - Login

Pour connecter un utilisateur, il faut utiliser les services d'authentification d'Identity. La méthode `SignInAsync` est utilisée pour authentifier un utilisateur :

```
[HttpPost]
public async Task<IActionResult> Login(LoginViewModel model)
{
    var user = await _userManager.FindByEmailAsync(model.Email);
    if (user != null && await _userManager.CheckPasswordAsync(user, model.Password))
    {
        await _signInManager.SignInAsync(user, isPersistent: model.RememberMe);
        return RedirectToAction("Index", "Home");
    }
    return View(model);
}
```

Création de rôle

Les rôles peuvent être créés et attribués aux utilisateurs pour contrôler l'accès à différentes ressources.

```
public class SeedData
{
    public static async Task Initialize(IServiceProvider serviceProvider, UserManager<ApplicationUser> userManager, RoleManager<IdentityRole> roleManager)
    {
        var roleNames = new[] { "Admin", "User" };
        foreach (var roleName in roleNames)
        {
            var roleExist = await roleManager.RoleExistsAsync(roleName);
            if (!roleExist)
            {
                await roleManager.CreateAsync(new IdentityRole(roleName));
            }
        }
    }
}
```

La classe `SeedData` est généralement placée dans un répertoire `Data` ou `Infrastructure` du projet.

Attribution des rôles aux utilisateurs

Une fois les rôles créés, les utilisateurs peuvent se voir attribuer un rôle spécifique :

```
var user = await userManager.FindByEmailAsync("user@example.com");
if (user != null)
{
    await userManager.AddToRoleAsync(user, "Admin");
}
```

Autorisation avec les rôles

ASP.NET Core permet de restreindre l'accès aux actions en utilisant des attributs comme `[Authorize]` et en spécifiant les rôles nécessaires :

```
[Authorize(Roles = "Admin")]
public IActionResult AdminOnly()
{
    return View();
}
```

Autorisation basée sur des politiques

Des politiques d'autorisation peuvent être définies dans `Program.cs` pour contrôler l'accès de manière plus fine. Une politique peut être définie avec des critères comme des rôles ou des revendications :

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));
});
```

Autorisation basée sur des politiques

Ensuite, cette politique peut être appliquée dans les actions :

```
[Authorize(Policy = "AdminOnly")]
public IActionResult AdminPanel()
{
    return View();
}
```

Déconnexion d'un utilisateur

Pour déconnecter un utilisateur, la méthode `SignOutAsync` du `SignInManager` est utilisée :

```
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    return RedirectToAction("Index", "Home");
}
```

10. Gestion des erreurs

Gestion des erreurs dans ASP.NET MVC

La gestion des erreurs dans une application web est cruciale pour garantir une expérience utilisateur fluide et sécurisée. ASP.NET Core MVC offre plusieurs mécanismes pour gérer les erreurs côté serveur et côté client.

Utilisation de la gestion des erreurs globale

Dans ASP.NET Core, le middleware de gestion des erreurs peut être utilisé pour capturer les exceptions non gérées et afficher une page d'erreur spécifique. Cela se configure dans le fichier `Program.cs` :

```
var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();  // Affiche des informations détaillées sur l'erreur en mode développement
}
else
{
    app.UseExceptionHandler("/Home/Error");  // Redirige vers la page d'erreur en mode production
    app.UseHsts();  // Utilisé pour sécuriser les connexions
}
```

Page d'erreur personnalisée

Il est possible de définir une action spécifique dans un contrôleur pour gérer les erreurs et afficher une page d'erreur personnalisée :

```
public class HomeController : Controller
{
    [Route("Home/Error")]
    public IActionResult Error()
    {
        return View();  // Vue d'erreur personnalisée
    }
}
```

Page d'erreur

La vue associée, `Error.cshtml`, peut être définie comme suit :

```
<h2>Une erreur s'est produite</h2>
<p>Nous sommes désolés pour ce désagrément. Veuillez réessayer plus tard.</p>
```

Gestion des exceptions dans les actions

Des blocs `try-catch` peuvent être utilisés dans les actions pour capturer les exceptions spécifiques et fournir des messages d'erreur appropriés à l'utilisateur :

```
public IActionResult Index()
{
    try
    {
        // Code susceptible de lancer une exception
        var result = SomeMethodThatMightThrowException();
        return View(result);
    }
    catch (Exception ex)
    {
        // Log de l'erreur et redirection vers une page d'erreur
        _logger.LogError(ex, "Une erreur est survenue dans l'action Index.");
        return RedirectToAction("Error");
    }
}
```

Validation côté client

ASP.NET Core MVC propose une validation automatique côté serveur et côté client. Les erreurs de validation peuvent être retournées dans le `ModelState` et affichées dans la vue. Le framework génère également des messages d'erreur côté client en utilisant **jQuery Validation**.

Exemple d'affichage des erreurs dans une vue :

```
<form asp-action="Create" method="post">
  <div>
    <label for="Name">Nom</label>
    <input type="text" asp-for="Name" />
    <span asp-validation-for="Name"></span>
    <!-- Affichage de l'erreur -->
  </div>
  <button type="submit">Envoyer</button>
</form>
```

Log des erreurs

ASP.NET Core propose un système de journalisation intégré qui permet de capturer et de stocker les erreurs. Cela peut être configuré dans `Program.cs` pour loguer les erreurs dans des fichiers, bases de données ou autres systèmes de stockage :

```
builder.Logging.AddFile("logs/app.log");  // Exemple de journalisation dans un fichier
```

Interface ILogger

Il est possible d'utiliser l'interface `ILogger` pour loguer les erreurs dans un contrôleur :

```
private readonly ILogger<HomeController> _logger;

public HomeController(ILogger<HomeController> logger)
{
    _logger = logger;
}

public IActionResult Error()
{
    _logger.LogError("Une erreur est survenue sur la page d'erreur.");
    return View();
}
```

Gestion des erreurs HTTP

Il est possible de personnaliser la page d'erreur affichée pour des erreurs HTTP spécifiques, comme les erreurs 404 ou 500. Cela peut être configuré dans `Program.cs` :

```
app.UseStatusCodePagesWithReExecute("/Home/Error", "?statusCode={0}");
```

Action d'erreur HTTP dans le contrôleur

Une action dédiée peut être créée pour gérer les erreurs HTTP spécifiques :

```
public IActionResult Error(int? statusCode)
{
    if (statusCode == 404)
    {
        return View("NotFound");  // Vue d'erreur personnalisée pour 404
    }
    return View();
}
```

11. Optimisation et bonnes pratiques

Importance de l'optimisation dans ASP.NET MVC

L'optimisation vise à garantir que l'application web fonctionne efficacement, même sous une forte charge, tout en fournissant une expérience utilisateur optimale. En suivant les bonnes pratiques, la maintenance du code devient également plus facile et l'application plus évolutive.

Mise en cache

La mise en cache permet de réduire les charges inutiles sur le serveur et d'améliorer les temps de réponse.

Exemple d'utilisation de la mise en cache en mémoire :

```
[ResponseCache(Duration = 60)]
public IActionResult Index()
{
    return View();
}
```

Compresser les réponses HTTP

Activez la compression des réponses pour réduire la taille des fichiers transférés vers le client :

```
builder.Services.AddResponseCompression();
app.UseResponseCompression();
```

Utilisation des jetons CSRF

- **Validation des entrées** : Toujours valider et filtrer les entrées utilisateur pour éviter les injections SQL et les scripts malveillants.
- **Utilisation d'Anti-Forgery Tokens** pour prévenir les attaques CSRF :

```
<form asp-action="Submit" method="post">
  @Html.AntiForgeryToken()
  <!-- Champs du formulaire -->
  <button type="submit">Envoyer</button>
</form>
```

12. Publication et déploiement

Étapes générales de publication et déploiement

La publication et le déploiement d'une application ASP.NET Core MVC permettent de rendre l'application accessible sur un serveur ou une plateforme cloud. Voici les étapes principales :

1. **Préparation de l'application**
   - Vérifier que toutes les dépendances sont installées et à jour.
   - S'assurer que les configurations pour les environnements (développement, production) sont correctement définies dans le fichier `appsettings.json`.

Étapes générales de publication et déploiement

2. **Génération de l'application publiée**
   Utiliser la commande `dotnet publish` pour produire une version optimisée de l'application.
   Exemple :

```
dotnet publish -c Release -o ./publish
```

   Cela génère une version compilée dans un dossier, généralement nommé `publish`.

Hébergement sur IIS

- Installer le **Hosting Bundle** de .NET Core sur le serveur Windows.
- Activer le rôle IIS avec le module ASP.NET Core.
- Copier les fichiers du dossier `publish` vers le serveur.
- Configurer un nouveau site dans le **Gestionnaire IIS** :
  - Définir le chemin physique vers le dossier `publish`.
  - Associer l'application à un pool d'applications utilisant le runtime .NET Core.

Fichier web.config

ASP.NET Core utilise un fichier `web.config` pour configurer le déploiement sous IIS. Il est automatiquement généré lors de la publication, mais peut être personnalisé si nécessaire.

Bonnes pratiques pour la publication

- Séparer les configurations en utilisant : `appsettings.{Environment}.json`.
- Activer HTTPS sur le serveur.
- Configurer des certificats SSL.