

INTRODUCTION À ASP.NET

QU'EST-CE QUE ASP.NET ?

ASP.NET est un framework de développement web créé par Microsoft. Il permet de construire des applications web dynamiques, des services web et des sites internet interactifs. ASP.NET fait partie de la plate-forme .NET, et les développeurs peuvent écrire leur code en C#, VB.NET, ou d'autres langages pris en charge par .NET.

HISTOIRE ET ÉVOLUTION D'ASP.NET

- 2002: Lancement d'ASP.NET avec .NET Framework 1.0
- 2005: ASP.NET 2.0 apporte des contrôles serveur, des master pages
- 2008: ASP.NET 3.5 introduit LINQ et le support AJAX intégré
- 2010: ASP.NET 4.0 supporte le développement MVC
- 2016: ASP.NET Core est lancé, une refonte modulaire et multiplateforme

LES AVANTAGES D'UTILISER ASP.NET

- **Performance:** Optimisé pour la vitesse et la gestion des ressources.
- **Sécurité:** Fonctionnalités de sécurité robustes intégrées.
- **Ecosystème:** Large éventail de bibliothèques et de soutien de la communauté.
- **Outils de développement:** Intégration avec Visual Studio pour un développement efficace.
- **Compatibilité:** Prend en charge le développement multiplateforme avec ASP.NET Core.

STRUCTURE DE BASE D'UN PROJET ASP.NET

CRÉATION D'UN PROJET ASP.NET CORE

Pour créer un projet ASP.NET Core, suivez ces étapes :

1. Ouvrez Visual Studio.
2. Cliquez sur "Créer un nouveau projet".
3. Sélectionnez "Application web ASP.NET Core".
4. Choisissez le modèle de projet et la version .NET.
5. Nommez le projet et sélectionnez l'emplacement.
6. Cliquez sur "Créer".

ARBORESCENCE DES DOSSIERS ET FICHIERS

Un projet ASP.NET Core typique contient :

- `wwwroot` : fichiers statiques (CSS, JS, images).
- `Controllers` : logique de traitement des requêtes.
- `Views` : fichiers HTML pour l'interface utilisateur.
- `Models` : classes représentant les données.
- `Startup.cs` : configuration de l'application.
- `Program.cs` : point d'entrée de l'application.

FICHIER DE DÉMARRAGE (PROGRAM.CS)

Program.cs est le point d'entrée de l'application ASP.NET Core :

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

FICHIER DE CONFIGURATION (APPSETTINGS.JSON)

appsettings.json stocke la configuration de l'application :

```
{  
    "Logging": {  
        "LogLevel": {  
            "Default": "Information",  
            "Microsoft": "Warning"  
        }  
    },  
    "AllowedHosts": "*"  
}
```

- Paramètres de journalisation.
- Connexions à la base de données.
- Autres paramètres personnalisés.

FICHIER DE ROUTAGE (PROGRAM.CS)

Le routage est configuré dans **Program.cs** ou **Startup.cs** :

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

- Définit les chemins d'accès aux contrôleurs et actions.

DÉPENDANCES ET GESTION DES PACKAGES (FICHIER .CSPROJ)

Le fichier **.csproj** contient les informations sur le projet et les packages NuGet :

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.2.6" />
</ItemGroup>
```

- Liste des packages utilisés.
- Versions spécifiques pour chaque package.
- Gestion via Visual Studio ou CLI.

PRÉSENTATION DU MODÈLE MVC

DÉFINITION DU MODÈLE MVC

MVC est l'acronyme de Modèle-Vue-Contrôleur.

C'est un motif de conception architectural utilisé dans le développement web.

Le modèle MVC divise une application en trois composants principaux:

- Modèles pour la logique métier et les données
- Vues pour l'interface utilisateur
- Contrôleurs pour la logique de traitement des requêtes

MODÈLES (MODELS)

- Représentent la structure des données
- Contiennent la logique métier et la validation
- Interagissent avec la base de données
- Exemple : `ProductModel` contenant les détails d'un produit

VUES (VIEWS)

- Constituent l'interface utilisateur
- Affichent les données aux utilisateurs
- Utilisent le langage de balisage Razor pour la génération dynamique
- Exemple : `ProductView.cshtml` pour montrer les détails du produit

CONTRÔLEURS (CONTROLLERS)

- Gèrent les interactions utilisateur
- Reçoivent les entrées et les convertissent en actions
- Appellent les modèles et sélectionnent les vues à afficher
- Exemple : `ProductController` pour gérer les requêtes liées aux produits

AVANTAGES DE L'UTILISATION DE MVC

- Séparation des préoccupations : claire distinction entre logique métier, UI, et contrôle
- Facilité de maintenance : chaque composant peut être développé et testé indépendamment
- Développement parallèle : les développeurs peuvent travailler sur différents composants simultanément
- Réutilisabilité : les modèles et les contrôleurs peuvent être réutilisés à travers l'application

CONVENTIONS DE NOMMAGE DANS MVC POUR .NET 7

- Modèles : NomModel (ex. ProductModel)
- Vues : NomView.cshtml (ex. ProductView.cshtml)
- Contrôleurs : NomController (ex. ProductController)
- Il est important de suivre ces conventions pour la cohérence et la facilité de compréhension

STRUCTURE DE RÉPERTOIRES MVC DANS .NET

7

- **Models/** : contient les classes de modèles
- **Views/** : contient les fichiers Razor pour les vues
- **Controllers/** : contient les classes de contrôleurs
- **wwwroot/** : contient les fichiers statiques comme CSS, JS, et images
- Respecter cette structure facilite la navigation et le développement dans le projet

CRÉATION D'UN CONTROLEUR DANS MVC

STRUCTURE D'UN CONTRÔLEUR DANS ASP.NET CORE MVC

Un contrôleur est une classe C# qui gère les interactions utilisateur. Chaque contrôleur hérite de `Controller` ou `ControllerBase`. Il contient des méthodes d'action qui répondent aux requêtes HTTP. Les contrôleurs sont situés dans le dossier `Controllers`.

CRÉATION D'UN CONTRÔLEUR MANUELLEMENT

Pour créer un contrôleur :

1. Créez une classe dans le dossier **Controllers**.
2. Nommez la classe avec le suffixe "**Controller**".
3. Héritez de la classe **Controller** ou **ControllerBase**.

```
public class HomeController : Controller
{
    // Méthodes d'action ici
}
```

UTILISATION DE L'ATTRIBUT [CONTROLLER] ET CONVENTIONS DE NOMMAGE

L'attribut `[Controller]` n'est pas nécessaire si la classe suit la convention de nommage. Le nom du contrôleur doit se terminer par "Controller". Le framework identifie les contrôleurs par cette convention.

```
public class HomeController : Controller
{
    // Identifié comme un contrôleur
}
```

AJOUT DE MÉTHODES D'ACTION AU CONTRÔLEUR

Les méthodes d'action :

- Doivent être publiques.
- Peuvent retourner `IActionResult`, `ActionResult`, types primitifs, ou void.
- Correspondent aux opérations CRUD.

```
public IActionResult Index()
{
    return View();
}
```

GESTION DES REQUÊTES HTTP (GET, POST, ETC.)

Utilisez des attributs pour spécifier le type de requête HTTP :

- [HttpGet]
- [HttpPost]
- [HttpPut]
- [HttpDelete]

```
[HttpGet]  
public IActionResult Index()  
{  
    return View();  
}
```

RETOUR DE DONNÉES AU FORMAT IACTIONRESULT

Les méthodes d'action retournent des résultats implémentant `IAActionResult` :

- `ViewResult` pour les vues Razor.
- `JsonResult` pour les données JSON.
- `ContentResult` pour le contenu brut.

```
public IActionResult Index()
{
    return View();
}
```

UTILISATION DES SERVICES ET DE L'INJECTION DE DÉPENDANCES DANS LES CONTRÔLEURS

Injectez des services dans le contrôleur via le constructeur :

- Ajoutez les services nécessaires dans `Startup.cs`.
- Utilisez l'injection de dépendances pour les récupérer.

```
public class HomeController : Controller
{
    private readonly MonService _service;

    public HomeController(MonService service)
    {
        _service = service;
    }
}
```

ORGANISATION DES CONTRÔLEURS DANS L'ESPACE DE NOMS APPROPRIÉ

Placez les contrôleurs dans un espace de noms cohérent :

- Utilisez généralement `NomDuProjet.Controllers`.
- Aide à maintenir une structure claire du projet.

```
namespace MonApplication.Web.Controllers
{
    public class HomeController : Controller
    {
        // ...
    }
}
```

DÉFINITION D'UN MODÈLE DANS MVC

RÔLE DU MODÈLE DANS L'ARCHITECTURE MVC

- Le modèle représente la structure des données.
- Il contient les propriétés et la logique métier.
- Les modèles sont utilisés par les contrôleurs pour interagir avec la base de données.
- Ils sont responsables de la validation des données.

CRÉATION D'UNE CLASSE MODÈLE

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

- Une classe modèle est une simple classe C#.
- Elle définit les propriétés correspondant aux données.

ANNOTATIONS DES PROPRIÉTÉS DU MODÈLE

```
using System.ComponentModel.DataAnnotations;

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

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

    [Range(0.01, 10000)]
    public decimal Prix { get; set; }
}
```

- Les annotations ajoutent des métadonnées aux propriétés.
- Elles peuvent influencer la base de données et la validation.

UTILISATION DE DATA ANNOTATIONS POUR LA VALIDATION

Annotation	Description
[Required]	Champ obligatoire
[StringLength(50)]	Longueur maximale de la chaîne
[Range(1, 100)]	Valeur numérique dans un intervalle
[DataType(DataType.EmailAddress)]	Type de donnée spécifique

- Les Data Annotations sont utilisées pour la validation côté serveur.
- Elles garantissent que les données sont correctes avant d'être traitées.

CONVENTIONS DE NOMMAGE DES FICHIERS DE MODÈLE

- Le nom du fichier modèle correspond au nom de la classe.
- Les fichiers de modèle sont généralement placés dans le dossier `Models`.
- Exemple : `Utilisateur.cs` pour un modèle `Utilisateur`.

ORGANISATION DES MODÈLES DANS L'ARCHITECTURE DE FICHIERS DOTNET 7

- Les modèles sont organisés dans le dossier `Models`.
- Ils peuvent être regroupés dans des sous-dossiers par fonctionnalité.
- L'architecture de fichiers suit généralement le domaine d'application.

```
/Models
  /Account
    Utilisateur.cs
  /Product
    Produit.cs
```

DÉFINITION D'UNE VUE DANS MVC

RÔLE DES VUES DANS LE MODÈLE MVC

- Les vues sont responsables de la présentation des données.
- Elles génèrent le HTML à envoyer au navigateur.
- Utilisent le Razor pour mélanger HTML et C#.
- Affichent les données fournies par les contrôleurs.
- Peuvent utiliser des modèles de vue pour structurer les données.

STRUCTURE D'UN FICHIER DE VUE RAZOR (.CSHTML)

- Extension de fichier : `.cshtml`
- Contient du HTML avec des balises Razor.
- Les balises Razor sont préfixées par `@`.
- Peut inclure du code C# pour la logique de présentation.
- Les blocs de code sont encadrés par `@{ ... }`.

SYNTAXE RAZOR DE BASE POUR LES VUES

- @expression : affiche le résultat d'une expression C#.
- @{ ... } : bloc de code C# pour définir des variables, etc.
- @for, @foreach : boucles pour itérer sur des collections.
- @if, @else : conditions pour contrôler l'affichage.
- @Html.ActionLink("texte", "Action") : crée un lien.

UTILISATION DE MODÈLES DE VUE (VIEWMODELS)

- ViewModel : classe contenant les données pour une vue.
- Organise et structure les données nécessaires pour l'affichage.
- Permet de séparer les données de la logique métier.
- Passé du contrôleur à la vue via la méthode `View(model)`.

PASSAGE DE DONNÉES DU CONTRÔLEUR À LA VUE

- **ViewBag** : objet dynamique pour passer des données simples.
- **ViewData** : dictionnaire pour passer des données clé-valeur.
- Modèle (Model) : objet fortement typé passé à la vue.
- Utilisation dans la vue : `@Model.Propriete` pour accéder aux données.

GESTION DES VUES PARTIELLES ET DES COMPOSANTS DE VUE

- Vues partielles : morceaux de vues réutilisables.
- Composants de vue : comme les vues partielles mais avec logique C#.
- Utilisation : `@Html.Partial("NomPartial")` ou `@await Component.InvokeAsync("NomComponent")`.
- Permettent de découper les vues en éléments maintenables.

CONVENTIONS DE NOMMAGE POUR LES VUES DANS ASP.NET CORE

- Nom de vue par défaut : `NomAction.cshtml`.
- Recherche dans le dossier `views/NomContrôleur` puis `Views/Shared`.
- Nom de vue spécifique : `return View("NomVue")` dans le contrôleur.
- Respecter ces conventions facilite la maintenance et la compréhension.

NOUVEAUTÉS DES VUES DANS .NET 7

- Améliorations de performance pour le rendu des vues.
- Support étendu pour les composants de vue réutilisables.
- Intégration plus poussée avec les outils de développement front-end.
- Améliorations de la syntaxe Razor pour une écriture plus concise.
- Fonctionnalités de sécurité renforcées pour la manipulation des données.

ROUTAGE DANS ASP.NET MVC

FONCTIONNEMENT DU ROUTAGE DANS ASP.NET MVC

- Le routage dans ASP.NET MVC permet de mapper les URL aux actions des contrôleurs.
- Il utilise le modèle RouteTable pour définir des routes.
- Les routes sont définies dans le fichier Global.asax ou dans un fichier de démarrage.
- Chaque route est associée à un modèle d'URL et à un ensemble de valeurs par défaut pour les paramètres.

CONFIGURATION DE ROUTE PAR DÉFAUT

- La configuration de route par défaut est souvent définie dans Global.asax.
- Syntaxe de la route par défaut:

```
routes.MapRoute(  
    name: "Default",  
    url: "{controller}/{action}/{id}",  
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }  
);
```

ROUTE ATTRIBUTE

- Les attributs de route permettent de définir des routes directement sur les actions des contrôleurs.
- Syntaxe d'utilisation:

```
[Route("mon-action")]
public ActionResult MaMethode() { ... }
```

ROUTES CONVENTIONNELLES VS ROUTES ATTRIBUÉES

- **Routes conventionnelles:**

- Basées sur un modèle défini globalement.
- Moins flexibles pour des scénarios complexes.

- **Routes attribuées:**

- Déclarées directement sur les méthodes des contrôleurs.
- Plus de contrôle et de précision.

CONTRAINTES DE ROUTE

- Les contraintes de route permettent de restreindre les paramètres des routes.
- Exemple de contrainte sur un paramètre `id` pour qu'il soit numérique:

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index" },
    new { id = @"\d+" } // Contrainte regex pour un nombre
);
```

GESTION DES PARAMÈTRES DANS LES ROUTES

- Les paramètres dans les routes sont définis par des accolades {}.
- Ils sont passés aux actions des contrôleurs comme arguments.
- Exemple d'utilisation:

```
public ActionResult Details(int id) { ... }
```

ROUTAGE VERS LES ACTIONS DU CONTRÔLEUR

- Le système de routage mappe les URL aux actions des contrôleurs.
- Si une route correspond à une URL, le contrôleur et l'action correspondants sont invoqués.
- Les valeurs des paramètres sont extraites de l'URL et passées à l'action.

INTERACTION AVEC UNE BASE DE DONNÉES

CONTEXTE DE BASE DE DONNÉES (DBCONTEXT)

- `DbContext` est la classe principale d'Entity Framework.
- Fait le pont entre les objets C# et la base de données.
- Gère les connexions à la base et le cycle de vie des données.
- Permet de configurer des modèles et des relations.
- Utilisé pour effectuer des opérations CRUD sur la base.

CRÉATION D'UNE CONNEXION À LA BASE DE DONNÉES

- La connexion est définie dans le fichier `appsettings.json`.
- Utilise la chaîne de connexion pour accéder à la base de données.
- La classe `DbContext` est configurée avec `optionsBuilder.UseSqlServer`.
- Exemple de chaîne de connexion:

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

UTILISATION DE ENTITY FRAMEWORK CORE DB FIRST

- DB First: Génère des modèles à partir d'une base de données existante.
- Utilise la commande **Scaffold-DbContext** dans la console du gestionnaire de package.
- Exemple de commande:

```
Scaffold-DbContext "ConnectionString" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

- Les classes de modèles et de contexte sont créées automatiquement.

REQUÊTES LINQ POUR INTERROGER LA BASE DE DONNÉES

- LINQ (Language Integrated Query) permet de faire des requêtes en C#.
- Syntaxe intuitive et fortement typée.
- Exemple de requête pour obtenir tous les utilisateurs:

```
using (var context = new ApplicationDbContext())
{
    var utilisateurs = context.Utilisateurs.ToList();
}
```

- Supporte les opérations de filtrage, tri et agrégation.

CRUD (CREATE, READ, UPDATE, DELETE) OPÉRATIONS

- Create: Ajouter de nouvelles entités à la base.

```
var utilisateur = new Utilisateur { Nom = "Doe", Prenom = "John" };
context.Utilisateurs.Add(utilisateur);
context.SaveChanges();
```

- Read: Lire des données existantes.

```
var utilisateur = context.Utilisateurs.FirstOrDefault(u => u.Id == 1);
```

- Update: Mettre à jour des entités existantes.

```
utilisateur.Nom = "Smith";
context.SaveChanges();
```

- Delete: Supprimer des entités.

```
context.Utilisateurs.Remove(utilisateur);
context.SaveChanges();
```

SYNCHRONISATION DES DONNÉES AVEC LA BASE DE DONNÉES

- `SaveChanges()` applique les modifications dans le contexte à la base de données.
- Assure l'intégrité des données en utilisant des transactions.
- Peut être appelé plusieurs fois pour des mises à jour partielles.
- Gère automatiquement les conflits d'écriture.

GESTION DES TRANSACTIONS

- Les transactions garantissent l'atomicité des opérations.
- Utilisation explicite avec `BeginTransaction`:

```
using (var transaction = context.Database.BeginTransaction())
{
    try
    {
        // Opérations CRUD
        context.SaveChanges();
        transaction.Commit();
    }
    catch
    {
        transaction.Rollback();
    }
}
```

- Les transactions sont gérées implicitement par `SaveChanges()`.

GESTION DES ERREURS ET EXCEPTIONS LIÉES À LA BASE DE DONNÉES

- `DbUpdateException` pour les erreurs lors des opérations CRUD.
- `DbUpdateConcurrencyException` pour les conflits d'accès concurrents.
- Utiliser des blocs `try-catch` pour gérer les exceptions.

```
try
{
    context.SaveChanges();
}
catch (DbUpdateException ex)
{
    // Gestion des erreurs de mise à jour
}
```

- Importantes pour la robustesse et la stabilité de l'application.

LES FORMULAIRES DANS ASP.NET MVC

CRÉATION D'UN FORMULAIRE DANS UNE VUE RAZOR

Pour créer un formulaire dans une vue Razor :

- Utiliser la balise `<form>` avec l'attribut `action`.
- L'attribut `method` définit comment les données seront envoyées (GET ou POST).
- Les champs de formulaire sont créés avec `<input>`, `<select>`, etc.

```
<form action="/MonController/MaMethode" method="post">
    <input type="text" name="nom" />
    <input type="submit" value="Envoyer" />
</form>
```

UTILISATION DE TAG HELPERS POUR LES CHAMPS DE FORMULAIRE

Tag Helpers permettent de simplifier la syntaxe HTML :

- Utiliser **asp-for** pour lier un champ de formulaire à une propriété de modèle.
- **asp-action** et **asp-controller** pour spécifier le contrôleur et l'action.

```
<form asp-action="MaMethode" asp-controller="MonController" method="post">
    <input asp-for="Nom" type="text" />
    <input type="submit" value="Envoyer" />
</form>
```

ENVOI DE DONNÉES DE FORMULAIRE AU CONTRÔLEUR

Les données du formulaire sont envoyées au contrôleur via :

- L'attribut **action** de la balise **<form>**.
- La méthode HTTP spécifiée par **method** (POST pour l'envoi de données).

```
<form action="/MonController/MaMethode" method="post">
    <!-- Champs de formulaire -->
</form>
```

RÉCEPTION DES DONNÉES DE FORMULAIRE DANS UNE ACTION DE CONTRÔLEUR

Pour recevoir des données de formulaire dans une action de contrôleur :

- Créer une méthode avec le même nom que l'attribut `action` du formulaire.
- Utiliser `[HttpPost]` pour traiter les données envoyées par POST.

```
[HttpPost]
public IActionResult MaMethode(string nom)
{
    // Traiter les données reçues
    return View();
}
```

UTILISATION DE L'OBJET IFORMCOLLECTION

IFormCollection permet de collecter les valeurs du formulaire :

- Passer **IFormCollection** en paramètre de l'action de contrôleur.
- Accéder aux données avec **formCollection["nomDuChamp"]**.

```
[HttpPost]
public IActionResult MaMethode(IFormCollection formCollection)
{
    string nom = formCollection[ "nom" ];
    // Autres traitements
    return View();
}
```

BINDING DE MODÈLE AUTOMATIQUE

Le binding de modèle automatique lie les champs de formulaire aux propriétés :

- Définir un modèle avec des propriétés correspondant aux champs du formulaire.
- Passer le modèle en paramètre de l'action de contrôleur.

```
public class MonModele
{
    public string Nom { get; set; }
}

[HttpPost]
public IActionResult MaMethode(MonModele modele)
{
    // Utiliser les propriétés de modele
    return View();
}
```

UTILISATION DE VALIDATEANTIFORGERTOKEN POUR LA SÉCURITÉ DES FORMULAIRES

ValidateAntiForgeryToken empêche les attaques de type CSRF :

- Ajouter @Html.AntiForgeryToken() dans la vue Razor.
- Utiliser l'attribut [ValidateAntiForgeryToken] dans l'action de contrôleur.

```
<form method="post">
    @Html.AntiForgeryToken()
    <!-- Champs de formulaire -->
</form>
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult MaMethode(MonModele modele)
{
    // Traiter les données
    return View();
}
```

GESTION DES MÉTHODES HTTP GET ET POST DANS LES ACTIONS DE CONTRÔLEUR

Pour gérer GET et POST dans les actions de contrôleur :

- Utiliser [HttpGet] pour afficher le formulaire.
- Utiliser [HttpPost] pour traiter les données soumises.

```
[HttpGet]
public IActionResult Formulaire()
{
    // Afficher le formulaire
    return View();
}

[HttpPost]
public IActionResult Formulaire(MonModele modele)
{
    // Traiter les données soumises
    return View();
}
```

VALIDATION CÔTÉ SERVEUR

ATTRIBUTS DE VALIDATION INTÉGRÉS

Les attributs de validation intégrés permettent de spécifier des règles de validation pour les propriétés des modèles.

Exemples courants :

- **Required** : champ obligatoire
- **StringLength** : longueur maximale et minimale d'une chaîne
- **Range** : valeurs numériques dans un intervalle spécifique
- **RegularExpression** : correspondance à une expression régulière
- **Compare** : comparaison de deux propriétés pour l'égalité

DATA ANNOTATIONS

Les Data Annotations sont des attributs placés sur les propriétés des classes de modèle pour spécifier les règles de validation et de formatage.

Exemple d'utilisation :

```
public class User
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [StringLength(100, MinimumLength = 6)]
    public string Password { get; set; }
}
```

MODELSTATE

ModelState est un objet qui contient l'état de la validation des données soumises au serveur.

- Utilisé pour vérifier si les données soumises sont valides.
- Contient tous les erreurs de validation.
- Utilisé dans les actions du contrôleur pour prendre des décisions.

Exemple de vérification :

```
public ActionResult SaveUser(User user)
{
    if (ModelState.IsValid)
    {
        // Enregistrer l'utilisateur
    }
    else
    {
        // Retourner la vue avec erreurs
    }
}
```