



ASP





Introduction à ASP.NET





Présentation de ASP.NET

ASP.NET est un framework de développement web développé par Microsoft pour créer des applications web.

Il est construit sur le framework .NET et offre une large gamme de fonctionnalités pour créer des applications web riches et complexes.

Dans ce cours, nous allons apprendre à utiliser ASP.NET pour créer des applications web.





Présentation des avantages d'ASP.NET

ASP.NET Propose divers fonctionnalités :

- ◆ Création de pages web dynamiques
- ◆ Validation des données
- ◆ Sécurité
- ◆ Gestion des bases de données
- ◆ Gestion des sessions et des cookies





Présentation du modèle d'architecture

MVC signifie Modèle-Vue-Contrôleur. C'est un modèle d'architecture de développement logiciel populaire pour les applications web.

Le but de ce modèle est de séparer les différents aspects d'une application en trois couches distinctes :

- ◆ Modèle
- ◆ Vue
- ◆ Contrôleur





Le Modèle

Il représente les données et la logique métier de l'application.

Il gère la logique de l'application, telle que la récupération et la modification de données à partir de la base de données, les calculs, etc.

Exemple :

Dans une application de gestion de stock, le Modèle gère les données des produits, des clients, des commandes, etc.





La Vue

Elle représente l'interface utilisateur de l'application.

Elle affiche les données à l'utilisateur et gère les entrées utilisateur.

C'est la partie visible de l'application, telle que les pages web, les formulaires, etc.





Le Contrôleur

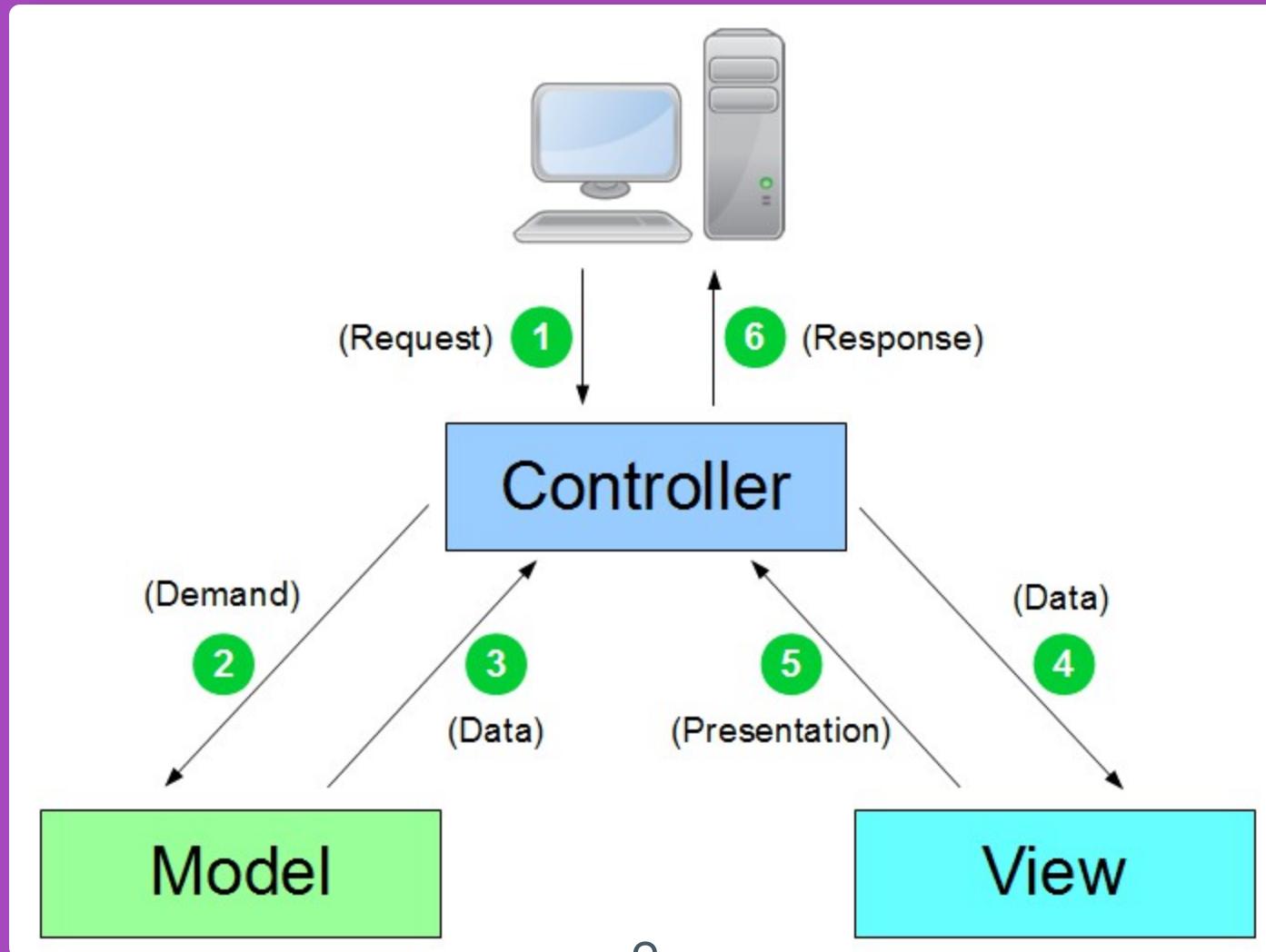
Il est la pièce de liaison entre le Modèle et la Vue.

Il reçoit les entrées de l'utilisateur, modifie les données du Modèle en conséquence, puis met à jour la Vue pour refléter les modifications apportées aux données.

Il contient aussi le routage de l'application, c'est-à-dire la logique qui détermine quelle page doit être affichée en fonction de l'URL.



Diagramme de l'architecture MVC





Les bases de ASP.NET





Comprendre le modèle de requête/réponse

Le modèle de requête/réponse est le modèle de base d'ASP.NET.

Toutes les requêtes HTTP sont traitées par le serveur web, qui les transmet ensuite à ASP.NET.

ASP.NET traite la requête et génère une réponse, qui est ensuite envoyée au navigateur web.





Différences fonctionnalités des requêtes

Nous allons commencer par prendre en mains ces deux requêtes :

- ◆ GET
- ◆ POST

Les requêtes GET sont utilisées pour récupérer des données.
Les requêtes POST sont utilisées pour envoyer des données.





Les requêtes GET

Dans quel cas utiliser une requête GET ?

- ◆ Lorsque vous souhaitez récupérer des données
- ◆ Lorsque vous souhaitez afficher une page web

Lorsqu'une requête GET est envoyée, les données sont envoyées dans l'URL.





Les requêtes POST

Dans quel cas utiliser une requête POST ?

- ◆ Lorsque vous souhaitez envoyer des données
- ◆ Lorsque vous souhaitez modifier des données

Lorsqu'une requête POST est envoyée, les données sont envoyées dans le corps de la requête.





Création d'une application ASP.NET avec visual studio code

Pour commencer nous pouvons créer une application ASP.NET avec visual studio code.

Nous pouvons créer un nouveau projet ASP.NET avec la commande suivante :

```
dotnet new mvc -o NomDuProjet
```





Création de pages web simples avec ASP.NET

Si nous voulons créer une page web simple, nous pouvons créer un fichier .cshtml dans le dossier Pages.

Ce fichier contiendra le code HTML de la page web.

Il peut également contenir du code C#.

Dans ce fichier, nous pouvons utiliser des balises Razor pour afficher des variables C#.





Razor ?

Razor est un langage de balisage qui permet d'ajouter du code C# dans des fichiers HTML.

Il est utilisé par ASP.NET pour créer des pages web dynamiques.

Les balises Razor servent à faire le lien entre le code C# et le code HTML.





Les balises Razor

Les balises Razor ont la forme suivante :

```
@{ code C# }
```

Si nous voulons afficher une variable C# dans le code HTML, nous pouvons utiliser la balise suivante :

```
@variable
```





Exemple

Nous pouvons créer un fichier Index.cshtml dans le dossier Pages.

Nous pouvons écrire le code suivant dans ce fichier :

```
<h2>Bienvenue dans ma Vue Razor</h2>

@{
    var message = "Ceci est un message créé directement dans la Vue.";
}

<p>@message</p>
```





Les Conditions Razor

Nous pouvons utiliser des conditions Razor pour afficher du code HTML en fonction de la valeur d'une variable.

Les conditions Razor ont la forme suivante :

```
@if (condition) {  
    code HTML  
}
```





Les boucles Razor

Nous pouvons utiliser des boucles Razor pour afficher du code HTML plusieurs fois.

Les boucles Razor ont la forme suivante :

```
@foreach (var element in liste) {  
    code HTML  
}
```





Exemple

```
<h2>Bienvenue dans ma Vue Razor</h2>

@{
    var liste = new List<string> { "Element 1", "Element 2", "Element 3" };
}

@if (liste.Count > 0) {
    <ul>
        @foreach (var element in liste) {
            <li>@element</li>
        }
    </ul>
}
```





Le Model Binding

Le Model Binding permet de lier les données d'un formulaire HTML avec un objet C#.

Pour utiliser le Model Binding, nous devons créer un objet C# qui contient les mêmes propriétés que le formulaire HTML.





Exemple

Nous pouvons créer un fichier Personne.cs dans le dossier Models.

Nous pouvons écrire le code suivant dans ce fichier :

```
public class Personne {  
    public string Nom { get; set; }  
    public string Prenom { get; set; }  
    public int Age { get; set; }  
}
```





Exemple

```
<form method="post">
    <label>Nom :</label>
    <input type="text" name="Nom" />
    <br />
    <label>Prenom :</label>
    <input type="text" name="Prenom" />
    <br />
    <label>Age :</label>
    <input type="text" name="Age" />
    <br />
    <input type="submit" value="Envoyer" />
</form>
```





Transmettre des données a une page web

Nous pouvons transmettre des données a une page web depuis un controller avec la méthode suivante :

```
return View("NomDuFichier.cshtml", objet);  
return View(objet);
```





Transmettre des données à une page web

Nous recevons les données dans la page web avec la balise suivante :

```
@model NomDuModel
```

Cette balise permet de lier le model de la page web avec le model transmis par le controller.





Exemple

En utilisant le code suivant, nous pouvons transmettre un objet Personne à la page web Index.cshtml :

```
public IActionResult Index() {
    var personne = new Personne {
        Nom = "Doe",
        Prenom = "John",
        Age = 25
    };
    return View(personne);
}
```





Exemple

Dans le fichier Index.cshtml, nous pouvons écrire le code suivant :

```
@model Personne  
  
<h2>Bienvenue dans ma Vue Razor</h2>  
  
<p>Nom : @Model.Nom</p>  
<p>Prenom : @Model.Prenom</p>  
<p>Age : @Model.Age</p>
```





Modèles de page et modèles master





Présentation des modèles de page et des modèles master

Un modèle master est une page web qui contient le code HTML commun à plusieurs pages web.

Il peut également contenir des zones qui seront remplacées par du code HTML dans les pages web.

Le mécanisme de modèles master permet de créer des pages web plus simples.





Exemple

Le meilleur exemple est le site web de Microsoft.

Le site web de Microsoft utilise un modèle master pour afficher le code HTML commun à toutes les pages web.

Ainsi toutes les pages web du site web de Microsoft ont le même design.





Comment créer un modèle master ?

Nous pouvons créer un modèle master en créant un fichier .cshtml dans le dossier Shared.

Celui-ci contiendra le code HTML commun à toutes les pages web.

Lorsque nous interrogeons une page web, le code HTML de la page web est inséré dans le code HTML du modèle master.





Créer une page web qui utilise un modèle master ?

Pour créer une page web qui utilise un modèle master, nous devons créer un fichier .cshtml dans le dossier Pages.

Ce fichier contiendra le code HTML de la page web.

Il sera donc affiché à la place de la zone @RenderBody() dans le modèle master.





@RenderBody()

La balise @RenderBody() est une balise Razor qui permet d'afficher le code HTML de la page web.

Il est donc possible de créer plusieurs pages web qui utilisent le même modèle master.

Dites-vous que la balise @RenderBody() est une zone qui sera remplacée par le code HTML de la page web.





@RenderSection()

La balise @RenderSection() est une balise Razor qui permet d'afficher le code HTML d'une zone.

Il est donc possible de créer plusieurs zones dans un modèle master.

RenderSection() permet de définir des sections optionnelles dans le layout parent qui peuvent être remplies ou non par les vues enfants





Exemple

Nous pouvons créer un fichier _Layout.cshtml dans le dossier Shared.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderBody()
</body>
</html>
```





Exemple

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderBody()
    @RenderSection("Scripts", required: false)
</body>
</html>
```





ViewBag

ViewBag est un objet dynamique qui peut contenir n'importe quelle donnée.

Il est très simple à utiliser, car il n'a pas besoin d'être typé, mais peut rendre le code plus difficile à maintenir à mesure que la complexité de l'application augmente.

Il peut être utilisé pour stocker des données temporaires pour une vue spécifique.





Exemple

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Username = "John Doe";
        return View();
    }
}
```

```
@{
    ViewBag.Title = "Home Page";
}
```





ViewData

ViewData est un dictionnaire de clés et de valeurs qui peut contenir des données de n'importe quel type.

Il est plus sûr et plus facile à maintenir que ViewBag, car les données doivent être typées lors de la déclaration, ce qui élimine les erreurs de type.

Cependant, il est moins facile à utiliser en raison de sa syntaxe de dictionnaire.





Exemple

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewData[ "Username" ] = "John Doe";
        return View();
    }
}
```

```
@{
    ViewBag.Username = "John Doe";
}
```





_ViewStart.cshtml

Le fichier `_ViewStart.cshtml` est un fichier qui est automatiquement inclus dans toutes les vues.

Il est utilisé pour définir des éléments communs à toutes les vues, comme des directives de compilation, des espaces de noms, des helpers, des filtres, etc.





_ViewImports.cshtml

Dans le fichier _ViewImports.cshtml, nous pouvons définir des espaces de noms, des helpers, des filtres, etc.

Exemple :

```
@using System.Web.Mvc  
@using System.Web.Mvc.Ajax  
@using System.Web.Mvc.Html  
@using System.Web.Routing
```





RedirectToAction

Redirige vers une action spécifique dans le même contrôleur.

```
public ActionResult Index()
{
    return RedirectToAction("About");
}
```





Redirection

Redirige vers une action spécifique dans un contrôleur spécifique.

```
public ActionResult Index()
{
    return RedirectToAction("About", "Home");
}
```





Requete POST

La méthode POST est utilisée pour envoyer des données à un serveur pour créer ou mettre à jour une ressource.

```
[HttpPost]  
public ActionResult Create(Product product)  
{  
    // ...  
}
```





Formulaires

Les formulaires HTML sont utilisés pour envoyer des données à un serveur.

Elles envoient les données en utilisant la méthode GET ou POST.

En utilisant la méthode GET, les données sont visibles dans l'URL.

Nous pouvons créer un formulaire en utilisant la balise
`@Html.BeginForm()`.





Exemple

```
@using (Html.BeginForm("Create", "Home", FormMethod.Post))  
{  
    @Html.TextBox("Name")  
    @Html.TextBox("Price")  
    <input type="submit" value="Create" />  
}
```





Validation

La validation des données est une étape importante dans le développement d'une application web.

Elle permet de s'assurer que les données saisies par l'utilisateur sont valides.

Si les données ne sont pas valides, nous pouvons afficher un message d'erreur.





Validation dans les vues

La validation des données peut être effectuée dans les vues.

Pour cela, nous pouvons utiliser la balise `@Html.ValidationMessage()`.

```
@Html.ValidationMessage("Name")
```





Le ModelState

L'objet ModelState contient l'état de validation des données soumises par les utilisateurs dans une application ASP.NET MVC.

Il stocke les résultats de la validation des données modèle à partir des règles définies sur les modèles ainsi que les erreurs générées par la validation.

Lorsque les données sont soumises à l'application, le ModelState est automatiquement mis à jour pour refléter l'état de validation des données.





Exemple

```
[HttpPost]
public ActionResult Create(Task task)
{
    if (ModelState.IsValid)
    {
        // Les données sont valides, elles peuvent être enregistrées
        db.Tasks.Add(task);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    // Les données ne sont pas valides, une erreur est affichée
    return View(task);
}
```





L'actionName

L'attribut ActionName permet de spécifier le nom de l'action.

```
[ActionName("Create")]
public ActionResult CreateTask()
{
    // ...
}
```





L'actionName

De cette manière, nous pouvons créer une action qui a le même nom qu'une autre action.

Nous pourrons donc appeler l'action `CreateTask()` en utilisant l'URL `/Home/Create`.

La différence entre les deux actions est que l'une est une action POST et l'autre une action GET.





Connection a notre BDD

Pour utiliser Entity Framework, nous devons ajouter une référence à System.Data.Entity.

Ensuite, nous devons créer une classe qui hérite de DbContext.

```
public class TaskContext : DbContext
{
    public DbSet<Task> Tasks { get; set; }
}
```





Récupérer les données de la BDD

Pour récupérer les données de la BDD, nous devons créer une instance de notre classe TaskContext.

```
public ActionResult Index()
{
    using (var db = new TaskContext())
    {
        var tasks = db.Tasks.ToList();
        return View(tasks);
    }
}
```





Exemple

```
public ActionResult Index()
{
    using (var db = new TaskContext())
    {
        var tasks = db.Tasks.ToList();
        return View(tasks);
    }
}
```





Exemple

```
public ActionResult Details(int id)
{
    using (var db = new TaskContext())
    {
        var task = db.Tasks.Find(id);
        return View(task);
    }
}
```





Exemple

```
[ActionName("Create")]
public ActionResult Create()
{
    return View();
}
```





```
[HttpPost]
[ActionName("Create")]
public ActionResult CreateTask(Task task)
{
    if (ModelState.IsValid)
    {
        using (var db = new TaskContext())
        {
            db.Tasks.Add(task);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    return View(task);
}
```





Helpers

Les helpers sont des méthodes qui permettent de générer du code HTML.

asp-action : permet de spécifier l'action à appeler.

asp-controller : permet de spécifier le contrôleur à appeler.

asp-route-id : permet de spécifier un paramètre à passer à l'action.

asp-route : permet de spécifier un paramètre à passer à l'action.





Model Binding

Le model binding est un processus qui permet de lier les données d'une requête HTTP à des paramètres d'une action.

Il permet de récupérer les données d'une requête HTTP et de les convertir en objets C#.





Model Binding

Pour utiliser le model binding, nous allons lier notre modèle à notre vue.

Pour cela, nous devons utiliser la balise @model.

```
@model Task
```





Edit

Nous pouvons utiliser le model binding pour modifier les données d'une tâche.

Dans l'action Edit(), nous récupérons l'objet Task à partir de la BDD.

```
public ActionResult Edit(int id)
{
    var task = db.Tasks.Find(id);
    return View(task);
}
```





Edit

Maintenant, nous pouvons modifier les données de la tâche et les enregistrer dans la BDD.

```
[HttpPost]
public ActionResult Edit(Task task)
{
    db.tasks.Update(task);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```





Delete

Pour supprimer une tâche, nous devons récupérer l'objet Task à partir de la BDD.

```
public ActionResult Delete(int id)
{
    var task = db.Tasks.Find(id);
    return View(task);
}
```





Validation

Pour valider les données, nous devons ajouter l'attribut [Required] sur les propriétés de notre modèle.

```
public class Task
{
    [Required]
    public string Name { get; set; }
    [Required]
    public decimal Price { get; set; }
}
```



