

CHAPITRE 2 : ASP.NET Core MVC

Plan du Chapitre

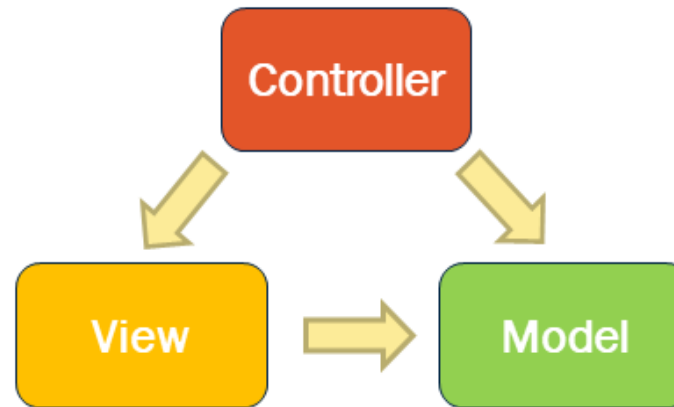
- Le modèle MVC
- Installer le package MVC
- Ajout de modèle
- Injection des dépendances
- Le contrôleur
- Les méthodes d'action
- ActionResult
- Sélecteurs d'action
- Les vues
- Création de vues
- Afficher une collection dans une vue
- Création d'une vue Edit

Plan du Chapitre

- Passer les données du contrôleur à la vue
- Notion de ViewModel
- Layout Views
- Les sections
- Le fichier _ViewStart.cshtml
- Le fichier _ViewImports.cshtml
- Les vues partielles (Partial View)
- Syntaxe Razor
- Tag Helpers

Le Modèle MVC

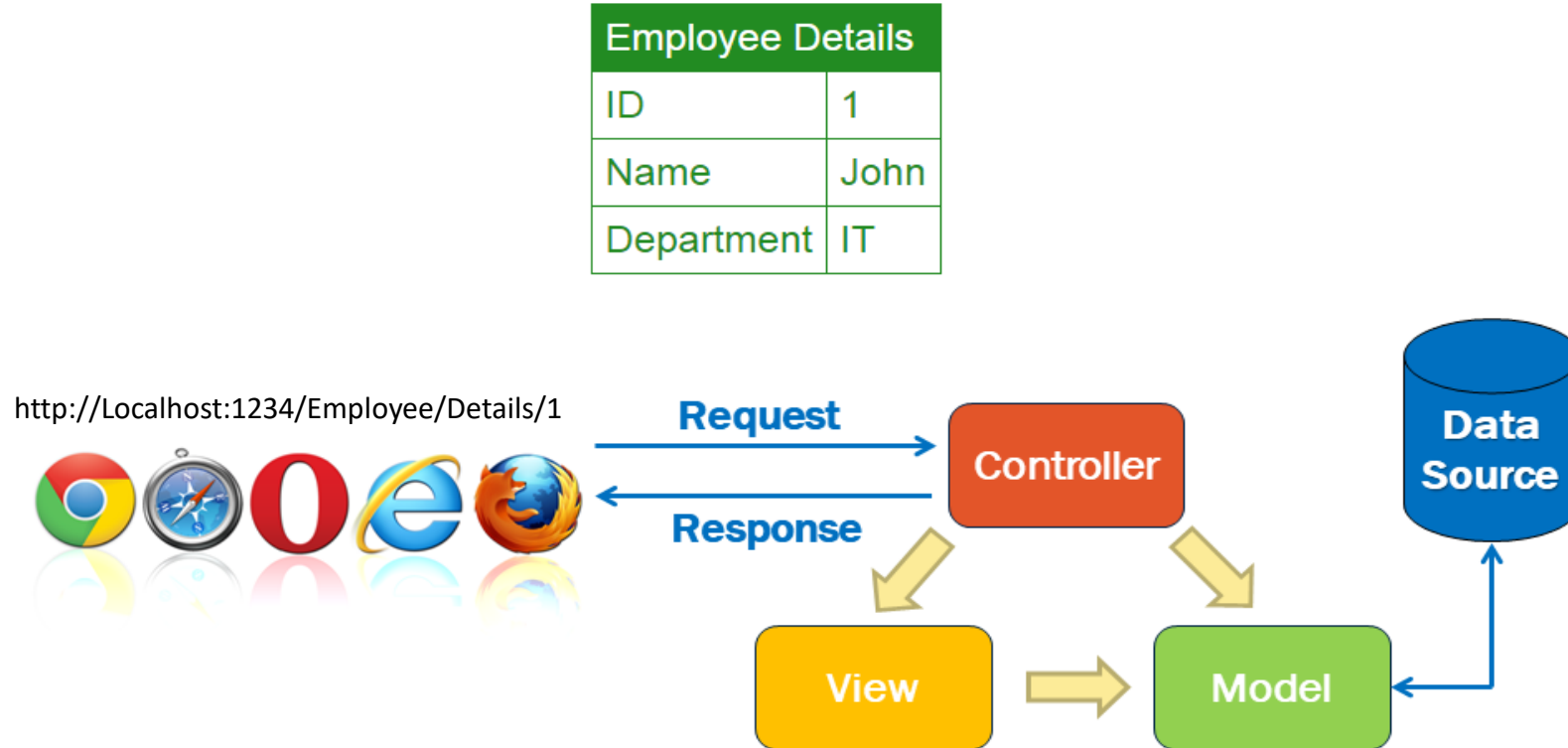
- MVC est un modèle architectural qui sépare les applications en trois composants : modèle, vue et contrôleur.



Le Modèle MVC

- **Modèle** : Le modèle représente la forme des données et la logique métier. Il maintient les données de l'application. Les objets de modèle récupèrent et stockent l'état du modèle dans une base de données.
- **View** : View est une interface utilisateur. Affiche les données en utilisant le modèle à l'utilisateur et permet également de modifier les données.
- **Contrôleur** : le contrôleur gère la demande de l'utilisateur. En règle générale, l'utilisateur interagit avec un View, ce qui déclenche à son tour la demande d'URL appropriée. Cette demande sera gérée par un contrôleur. Le contrôleur rend la vue appropriée avec les données du modèle en réponse.

Le Modèle MVC



- Lorsque l'utilisateur entre une adresse URL dans le navigateur, celui-ci se connecte au serveur et appelle le contrôleur approprié. Ensuite, le contrôleur utilise la vue et le modèle appropriés, crée la réponse et la renvoie à l'utilisateur.

Le Modèle MVC

MVC Model

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Department { get; set; }
}
```

Employee Details	
ID	1
Name	John
Department	IT

```
public interface IEmployeeRepository
{
    Employee GetEmployee(int id);
    void Save(Employee employee);
}
```

Model

Employee
+
EmployeeRepository

```
public class EmployeeRepository : IEmployeeRepository
{
    public Employee GetEmployee(int id)
    {
        // Logic to retrieve employee details
    }

    public void Save(Employee employee)
    {
        // Logic to save employee details
    }
}
```

Le Modèle MVC

MVC Controller

<http://localhost:1234/Employee/Details/1>

```
public class EmployeeController : Controller
{
    private IEmployeeRepository _employeeRepository;

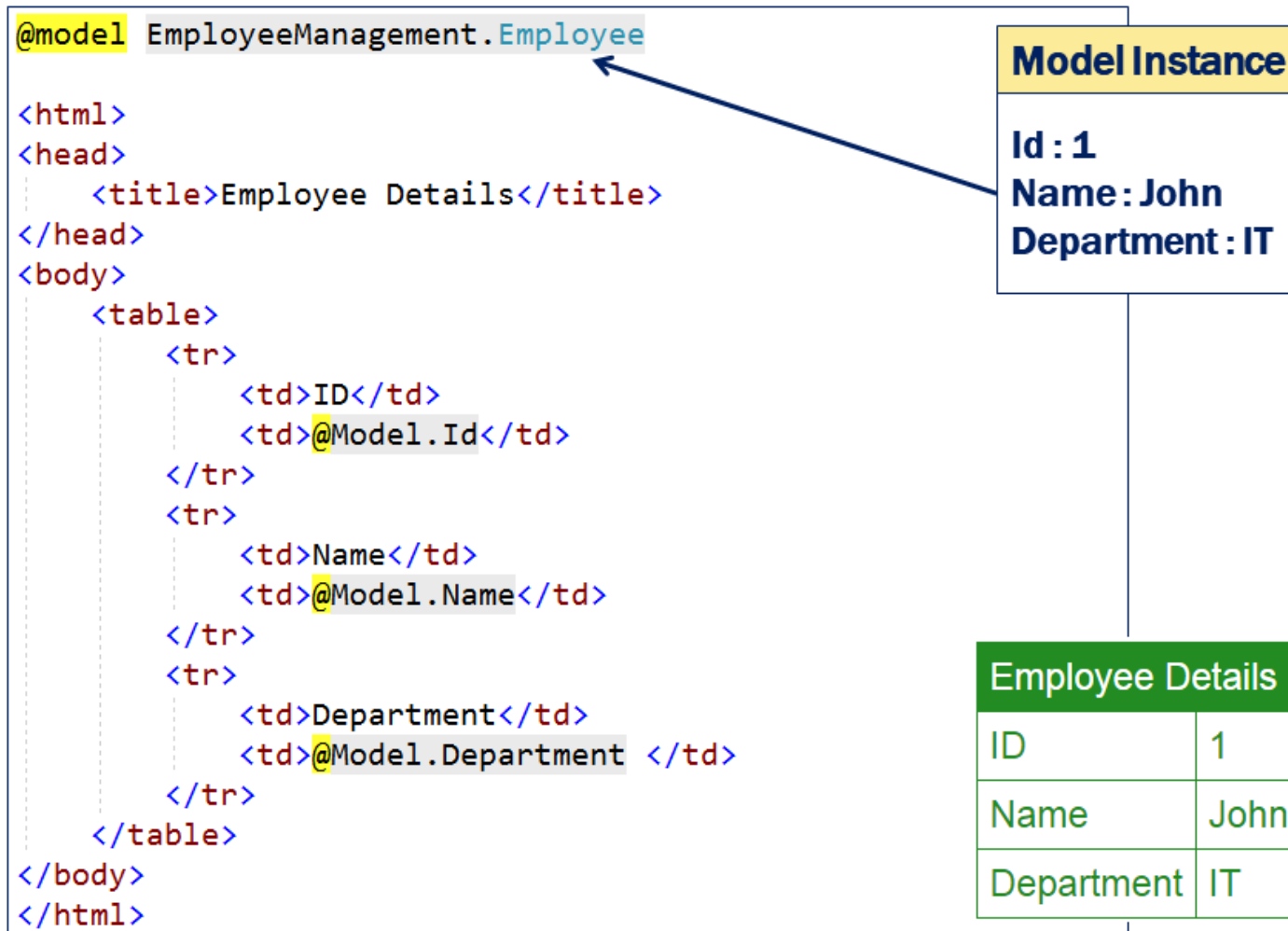
    public EmployeeController(IEmployeeRepository employeeRepository)
    {
        _employeeRepository = employeeRepository;
    }

    public IActionResult Details(int id)
    {
        Employee model = _employeeRepository.GetEmployee(id);
        return View(model);
    }
}
```

Routing Rules map URLs to Controller Action Methods

Le Modèle MVC

MVC View



Le Modèle MVC

Conclusion :

- **MVC** est un modèle de conception architecturale pour la mise en œuvre de la couche d'interface utilisateur d'une application.
- **Modèle:** ensemble de classes qui représentent des données + la logique pour gérer ces données. Exemple - Classe Employee qui représente les données de l'employé + la classe EmployeeRepository qui enregistre et récupère les données des employés à partir de la source de données sous-jacente telle qu'une base de données.
- **Vue:** contient la logique d'affichage pour présenter les données du modèle qui lui sont fournies par le contrôleur.
- **Contrôleur:** gère la requête http, travaille avec le modèle et sélectionne une vue pour rendre ce modèle.

Ajouter un modèle

- Un ensemble de classes qui représentent les données de l'application et les classes qui gèrent ces données. Par exemple pour gérer les données des employés d'une entreprise, une classe employé doit être créée.
- Ajouter un dossier Models sous la racine du projet.

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public string Department { get; set; }
}
```

- Pour gérer les données des employés, il faut créer une interface et une classe

```
public interface IEmployeeRepository
{
    Employee GetEmployee(int Id);
}
```

Ajouter un modèle

```
public class EmployeeRepository : IEmployeeRepository
{
    private List<Employee> employeeList;

    public EmployeeRepository()
    {
        employeeList = new List<Employee>() {
            new Employee() { Id = 1, Name = "Ali", Department = "HR", Email = "ali@societe.com" },
            new Employee() { Id = 2, Name = "Mourad", Department = "IT", Email = "mourad@societe.com" },
            new Employee() { Id = 3, Name = "Sofien", Department = "IT", Email = "sofien@societe.com" },
        };
    }

    public Employee GetEmployee(int Id)
    {
        return this.employeeList.SingleOrDefault(e => e.Id == Id);
    }
}
```

Injection de dépendances

- L'ajout de l'interface `IEmployeeRepository` et la classe d'implémentation `EmployeeRepository` est très important. Cette abstraction d'interface nous permet d'utiliser l'injection de dépendances, ce qui rend notre application flexible et facilement testable par unité.
- Etape suivante : Créer un contrôleur `HomeController` dans le dossier `Controllers`.
- La méthode `Index` affiche le nom de l'employé ayant `ID=1`.

```
public class HomeController : Controller
{
    public IEmployeeRepository _employeeRepository;

    public HomeController(IEmployeeRepository employeeRepository)
    {
        _employeeRepository = employeeRepository;
    }

    public string Index()
    {
        return _employeeRepository.GetEmployee(1).Name;
    }
}
```

Injection de dépendances

- HomeController dépend de IEmployeeRepository pour la récupération des données des employés.
- Au lieu de créer une nouvelle instance de IEmployeeRepository , nous injectons IEmployeeRepository dans HomeController à l' aide du constructeur.
- C'est ce qu'on appelle l'**injection de constructeur** , car nous utilisons le constructeur pour injecter la dépendance.
- Si on exécute le projet l'erreur suivante est affichée:

An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'model_exemple.Models.IEmployeeRepository' while attempting to activate 'model_exemple.Controllers.HomeController'.

Microsoft.Extensions.DependencyInjection.ActivatorUtilities.GetService(IServiceProvider sp, Type type, Type requiredBy, bool isDefaultParameterRequired)

Stack

Query

Cookies

Headers

Routing

InvalidOperationException: Unable to resolve service for type 'model_exemple.Models.IEmployeeRepository' while attempting to activate 'model_exemple.Controllers.HomeController'.

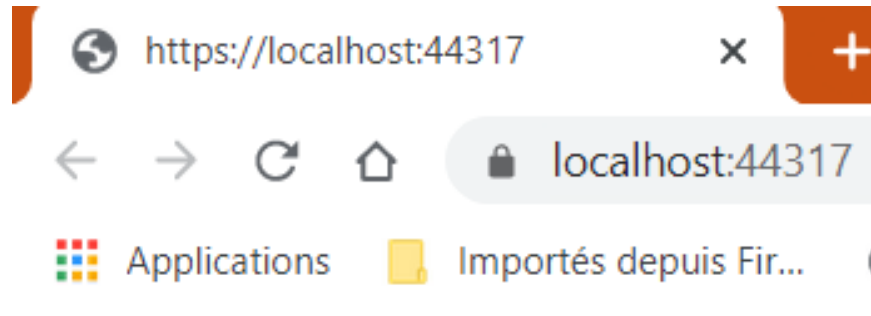
Injection de dépendances

Inscription des services avec le conteneur d'injection de dépendances (IOC Container)

- Dans le fichier Program.cs ajouter l'inscription de services suivantes :

```
builder.Services.AddSingleton<IEmployeeRepository, EmployeeRepository>();
```

puis refaire l'exécution et maintenant l'erreur va disparaître.



Ali

Injection de dépendances

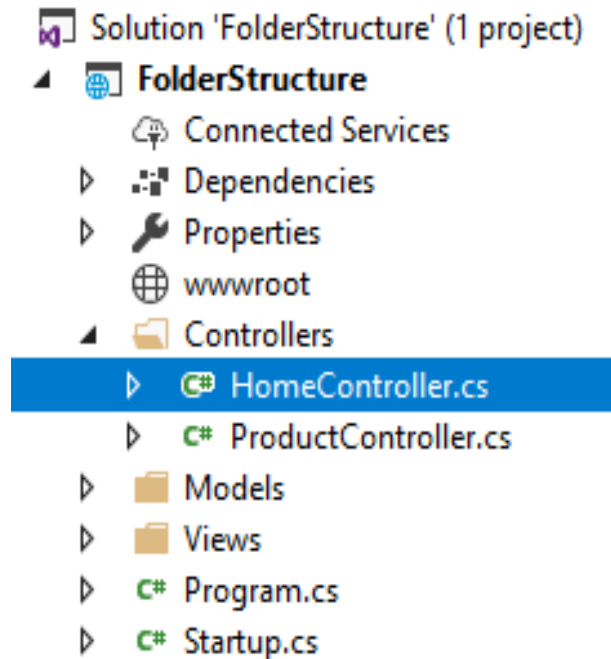
3 méthodes sont disponibles pour inscrire un service :

- **AddSingleton()** : Une instance de la classe est créée lors de sa première demande. Cette même instance est ensuite utilisée par toutes les requêtes suivantes. Ainsi, en général, un service avec AddSingleton n'est créé qu'une seule fois par application et cette seule instance est utilisée pendant toute la durée de vie de l'application.
- **AddTransient()** : Une nouvelle instance du service est créée chaque fois qu'elle est demandée.
- **AddScoped()** - Cette méthode crée un **service Scoped** . Une nouvelle instance d'un service Scoped est créée une fois par demande. Par exemple, dans une application Web, il crée 1 instance pour chaque requête http mais utilise la même instance dans les autres appels de cette même requête Web.

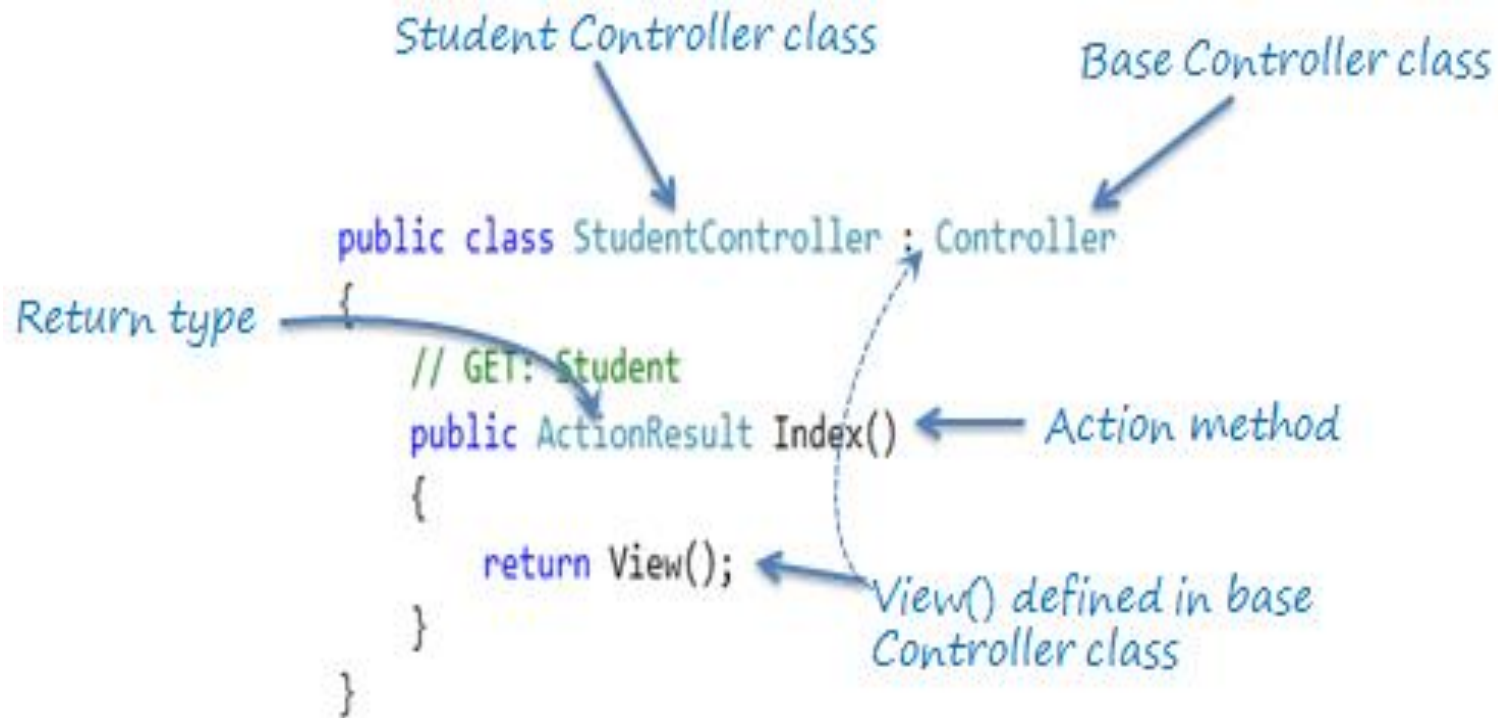
Le Contrôleur dans MVC



- Un contrôleur est une classe qui hérite de `Microsoft.AspNetCore.Mvc.Controller`.
- Le nom du contrôleur doit se terminer par le mot `Controller`.
- Gère les requêtes http entrante et répond à l'action de l'utilisateur.
- Généralement placé dans un dossier nommé `Controllers`.
- Contient un ensemble de méthodes appelées Méthodes d'action.
- Une méthode d'action peut retourner le résultat au format `JsonResult`, `ObjectResult`, `ViewResult` ou autres.



Méthodes d'action dans un contrôleur

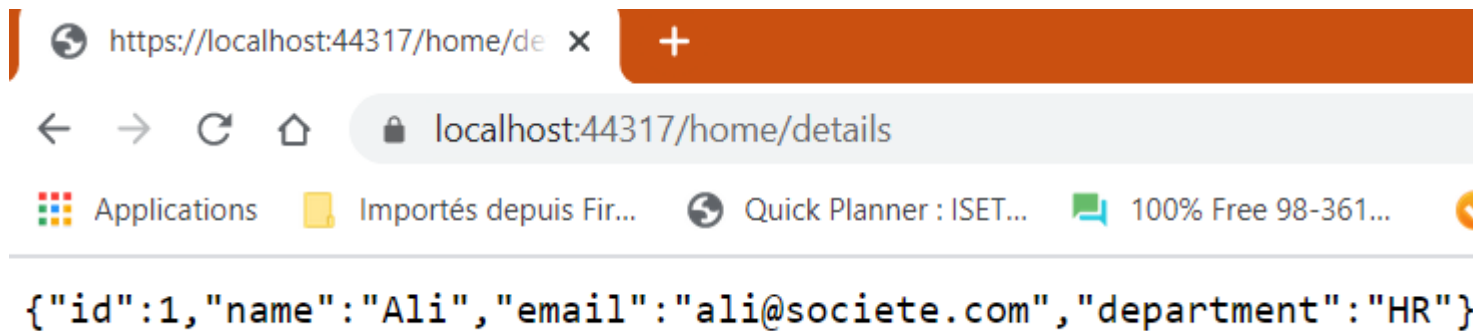


- La méthode d'action doit être publique. Elle ne peut pas être privée ou protégée
- La méthode d'action ne peut pas être surchargée
- La méthode d'action ne peut pas être une méthode statique.

Exemples de méthodes d'action

- On ajoute à HomeController une méthode Details qui retourne les informations de l'employé ayant l'ID=1.
- Avec le type JsonResult Le résultat retourné est au format Json.

```
public JsonResult Details()  
{  
    Employee model = _employeeRepository.GetEmployee(1);  
    return Json(model);  
}
```



Exemples de méthodes d'action

- Le type `ObjectResult` retourne par défaut un résultat json mais on peut aussi avoir le résultat au format xml.
- Le type `ViewResult` retourne les données résultats a un view qui doit être ajoutée.

```
public ObjectResult Details()
{
    Employee model = _employeeRepository.GetEmployee(1);
    return new ObjectResult(model);
}
```

```
public ViewResult Details()
{
    Employee model = _employeeRepository.GetEmployee(1);
    return View(model);
}
```

Action Results

- Lorsque l'action a terminé son travail, elle retournera généralement quelque chose au client et ce quelque chose implémentera généralement l'interface **IActionResult** (ou Task <IActionResult> si la méthode est asynchrone).
- ActionResult est l'implémentation par défaut de IActionResult
- Le tableau suivant liste quelques résultats d'action qui implémentent IActionResult.

Classe Résultat	Description
ViewResult	Résultat en HTML
EmptyResult	Aucun résultat
ContentResult	Représente une chaîne de caractères (string)
FileContentResult/ FilePathResult/ FileStreamResult	Représente un fichier
JavaScriptResult	Résultat un JavaScript
JsonResult	Résultat JSON utilisé dans AJAX
RedirectResult	Représente une redirection vers un nouvel URL
RedirectToRouteResult	Redirige vers une autre action pour le même ou un autre contrôleur
PartialViewResult	Retourne une vue partielle HTML

Les sélecteurs d'action

- Le sélecteur d'action est l'attribut qui peut être appliqué aux méthodes d'action. Il aide le moteur de routage à sélectionner la méthode d'action appropriée pour traiter une requête particulière.
 - **ActionName**
 - **NonAction**
 - **ActionVerbs**
- L'attribut `ActionName` nous permet de spécifier un nom d'action différent du nom de la méthode.
- Cette méthode d'action sera invoquée sur la requête `http://localhost/student/find/1` au lieu de la requête `http://localhost/student/getbyid/1`.

```
public class StudentController : Controller
{
    public StudentController()
    {
    }
    [ActionName("find")]
    public ActionResult GetById(int id)
    {
        // get student from the database
        return View();
    }
}
```

Les sélecteurs d'action

- **Non action** : indique qu'une méthode publique d'un contrôleur n'est pas une méthode d'action.
- **les ActionVerb** : Le sélecteur ActionVerb est utilisé lorsque vous souhaitez contrôler la sélection d'une méthode d'action basée sur une méthode de requête Http.

```
[NonAction]
public Student GetStudent(int id)
{
    return studentList.Where(s => s.StudentId == id).FirstOrDefault();
}
```

http://localhost/Student/Edit/1



```
public ActionResult Edit(int Id)
{
    var std = students.Where(s => s.StudentId == Id).FirstOrDefault();

    return View(std);
}
```

http://localhost/Student/Edit



```
[HttpPost]
public ActionResult Edit(Student std)
{
    //update database here..

    return RedirectToAction("Index");
}
```

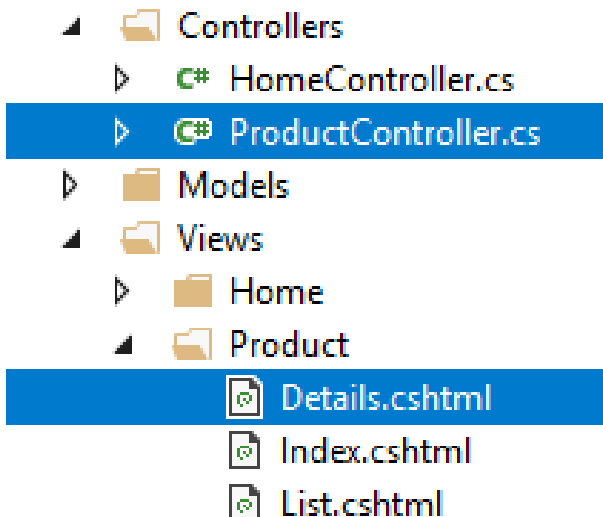
Les sélecteurs d'action

Le tableau suivant répertorie l'utilisation des méthodes http :

Méthodes Http	Usage
GET	Récupérer les informations du serveur. Les paramètres seront ajoutés à la chaîne de requête
POST	Pour créer une nouvelle ressource
PUT	Pour mettre à jour une ressource existante
HEAD	Identique à GET, mais le serveur ne renvoie pas le corps du message
OPTIONS	La méthode OPTIONS représente une demande d'informations sur les options de communication prises en charge par le serveur Web
DELETE	Pour supprimer une ressource existante
PATCH	Pour mettre à jour complètement ou partiellement la ressource

Les Vues

- Une vue est une interface utilisateur.
- La vue affiche les données du modèle à l'utilisateur et lui permet également de les modifier.
- Les vues ASP.NET MVC sont stockées dans le dossier Views du projet.
- Les différentes méthodes d'action d'une même classe de contrôleur peuvent rendre différentes vues. Par conséquent, le dossier Views contient un dossier distinct pour chaque contrôleur, portant le même nom que le contrôleur, afin de prendre en charge plusieurs vues.



Les Vues

- L'emplacement et le nom des vues doit respecter la convention suivantes :

/Views/[Controller Name]/[Action Name].cshtml

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

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

- Pour spécifier une vue particulière:

```
public IActionResult Test()
{
    return View("Details");
}
```

- Ou spécifier le chemin d'accès de la vue

```
public IActionResult Test()
{
    return View
        ("/ViewFolderName/SomeFolderName/ViewName.cshtml");
}
```

Overloaded Method

Description

View(object model)

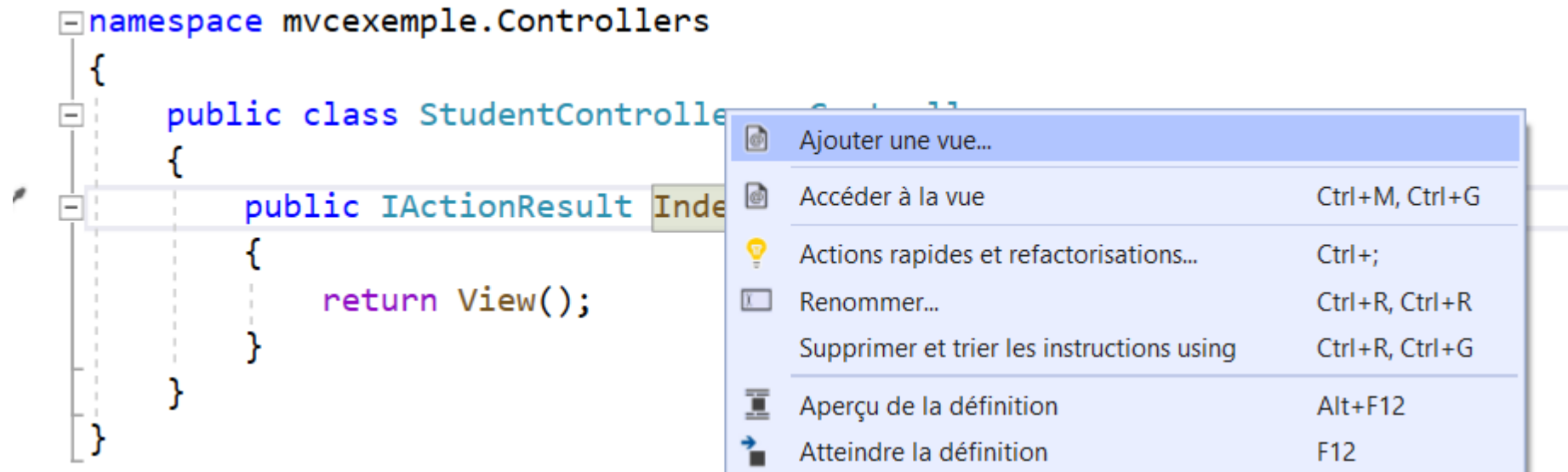
Passer le modèle de données du contrôleur à la vue

View(string viewName, object model)

Passer le nom de la vue et le modèle à la vue

Créer un View

- Créons maintenant une vue pour la méthode Index et comprenons comment utiliser le modèle dans la vue.
- Nous allons créer une vue, qui sera rendue à partir de la méthode Index de StudentController, Alors ouvrez la classe StudentController -> cliquez avec le bouton droit de la souris sur la méthode Index -> cliquez sur Ajouter une vue (Add View).



Créer un View

- Sélectionnez le modèle de scaffolding.
- La liste déroulante des modèles affiche les modèles par défaut disponibles pour les vues Créer, Supprimer, Détails, Modifier, Liste ou Vide.
- Sélectionnez le modèle "Liste" car nous voulons afficher la liste des étudiants dans la vue.

```
namespace mvcexemple.Controllers
{
    public class StudentController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

Ajouter Vue MVC

Nom de la vue : Index

Modèle : Empty (sans modèle)

Classe de modèle :

Options :

☐ Créer en tant que vue

☒ Bibliothèques de scripts

☒ Utiliser une page de disposition

(Laissez vide s'il est défini dans un fichier Razor _viewstart)

Ajouter Annuler

Créer un View

- Sélectionnez Student dans le dropdown.
- La liste déroulante des classes de modèle affiche automatiquement le nom de toutes les classes du dossier de modèle.
- Nous avons déjà créé la classe Student dans le dossier Model, elle serait donc incluse dans la liste déroulante.

Ajouter Vue MVC

Nomde_la_vue :

Index

Modèle :

List

Classe de modèle :

Student (mvcexemple.Models)

Options :

☐

Créer en tant que vue partielle

☒

Bibliothèques de scripts de référence

☒

Utiliser une page de disposition :

...

(Laissez vide s'il est défini dans un fichier Razor _viewstart)

Ajouter

Annuler

Créer un View

The screenshot displays the Visual Studio IDE with the following components:

- Menu Bar:** Fichier, Edition, Affichage, Projet, Générer, Déboguer, Test, Analyser, Outils, Extensions, Fenêtre, Aide.
- Toolbar:** Includes icons for file operations, debugging, and running the application.
- Search Bar:** Rechercher (Ctrl+Q).
- Explorer de solutions:** Shows the project structure with folders for Controllers, Models, Views, and Shared. The file `Index.cshtml` under the `Student` folder is selected.
- Code Editor:** Displays the content of `Index.cshtml`. The code includes a model declaration, a title assignment, a header, a link to create a new student, a table header with columns for Student ID, Name, and Age, and a table body with a single row for the first item in the model.
- Propriétés:** Shows the properties of the selected file, including its full path and name.
- Output Window:** Displays the status of the build and deployment process.

```
1 @model IEnumerable<Mvcexemple.Models.Student>
2
3 @{
4     ViewData["Title"] = "Index";
5 }
6
7 <h1>Index</h1>
8
9 <p>
10     <a asp-action="Create">Create New</a>
11 </p>
12 <table class="table">
13     <thead>
14         <tr>
15             <th>
16                 @Html.DisplayNameFor(model => model.StudentId)
17             </th>
18             <th>
19                 @Html.DisplayNameFor(model => model.StudentName)
20             </th>
21             <th>
22                 @Html.DisplayNameFor(model => model.Age)
23             </th>
24         </tr>
25     </thead>
26     <tbody>
27         @foreach (var item in Model) {
28             <tr>
29                 <td>
30                     @Html.DisplayFor(modelItem => item.StudentId)
31                 </td>
32             </tr>
33         }
```

Affichage du contenu d'une collection de type liste

```
public ActionResult Index()
{
    var studentList = new List<Student>
    {
        new Student() { StudentId = 1, StudentName = "John", Age = 18 } ,
        new Student() { StudentId = 2, StudentName = "Steve", Age = 21 } ,
        new Student() { StudentId = 3, StudentName = "Bill", Age = 25 } ,
        new Student() { StudentId = 4, StudentName = "Ram" , Age = 20 } ,
        new Student() { StudentId = 5, StudentName = "Ron" , Age = 31 } ,
        new Student() { StudentId = 6, StudentName = "Chris" , Age = 17 } ,
        new Student() { StudentId = 7, StudentName = "Rob" , Age = 19 }
    };
    // dans des applications réelles on récupère la liste à partir d'une base de données

    return View(studentList);
}
```

Affichage du contenu d'une collection de type liste

Si on exécute le projet par F5, et on tape l'URL : `http://localhost:xxxx/Student` on obtient le résultat suivant :

Index

[Create New](#)

StudentId	StudentName	Age	
1	John	18	Edit Details Delete
2	Steve	21	Edit Details Delete
3	Bill	25	Edit Details Delete
4	Ram	20	Edit Details Delete
5	Ron	31	Edit Details Delete
6	Chris	17	Edit Details Delete
7	Rob	19	Edit Details Delete

Création d'une vue Edit

- Ajouter une méthode d'action Edit dans StudentController.

```
public class StudentController : Controller
{
    List<Student> studentList = new List<Student>
    {
        new Student() { StudentId = 1, StudentName = "John", Age = 18 } ,
        new Student() { StudentId = 2, StudentName = "Steve", Age = 21 } ,
        new Student() { StudentId = 3, StudentName = "Bill", Age = 25 } ,
        new Student() { StudentId = 4, StudentName = "Ram" , Age = 20 } ,
        new Student() { StudentId = 5, StudentName = "Ron" , Age = 31 } ,
        new Student() { StudentId = 6, StudentName = "Chris" , Age = 17 } ,
        new Student() { StudentId = 7, StudentName = "Rob" , Age = 19 }
    };

    public IActionResult Index()
    {
        return View(studentList);
    }
    public ActionResult Edit(int id)
    {
        var std = studentList.Where(s => s.StudentId == id).FirstOrDefault();

        return View(std);
    }
}
```

Création d'une vue Edit

- Choisir le Template Edit et la classe Student comme modèle.

Ajouter Vue MVC

Nom de la vue :

Modèle :

Classe de modèle :

Options :

☐ Créer en tant que vue partielle

☒ Bibliothèques de scripts de référence

☒ Utiliser une page de disposition :

(Laissez vide s'il est défini dans un fichier Razor _viewstart)

Ajouter Annuler

```
Edit.cshtml | Index.cshtml | StudentController.cs | Startup.cs
1  @model mvcexemple.Models.Student
2
3  @{
4      ViewData["Title"] = "Edit";
5  }
6
7  <h1>Edit</h1>
8
9  <h4>Student</h4>
10 <hr />
11 <div class="row">
12     <div class="col-md-4">
13         <form asp-action="Edit">
14             <div asp-validation-summary="ModelOnly" class="text-danger"></div>
15             <div class="form-group">
16                 <label asp-for="StudentId" class="control-label"></label>
17                 <input asp-for="StudentId" class="form-control" />
18                 <span asp-validation-for="StudentId" class="text-danger"></span>
19             </div>
20             <div class="form-group">
21                 <label asp-for="StudentName" class="control-label"></label>
22                 <input asp-for="StudentName" class="form-control" />
23                 <span asp-validation-for="StudentName" class="text-danger"></span>
24             </div>
25             <div class="form-group">
26                 <label asp-for="Age" class="control-label"></label>
27                 <input asp-for="Age" class="form-control" />
28                 <span asp-validation-for="Age" class="text-danger"></span>
```

Création d'une vue Edit

- Dans Index.cshtml ajouter StudentId comme paramètre du lien Html.ActionLink

```
@foreach (var item in Model) {  
    <tr>  
        <td>  
            @Html.DisplayFor(modelItem => item.StudentId)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.StudentName)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.Age)  
        </td>  
        <td>  
            @Html.ActionLink("Edit", "Edit", new { id=item.StudentId }) |  
            @Html.ActionLink("Details", "Details", new { /* id=item.PrimaryKey */ }) |  
            @Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })  
        </td>  
    </tr>  
}
```

Création d'une vue Edit

- Exécuter la vue Index, choisir un étudiant et cliquer sur Edit

Index

[Create New](#)

StudentId	StudentName	Age	
1	John	18	Edit Details Delete
2	Steve	21	Edit Details Delete
3	Bill	25	Edit Details Delete
4	Ram	20	Edit Details Delete
5	Ron	31	Edit Details Delete
6	Chris	17	Edit Details Delete
7	Rob	19	Edit Details Delete

mvcexemple [Home](#) [Privacy](#)

Edit Student

StudentId

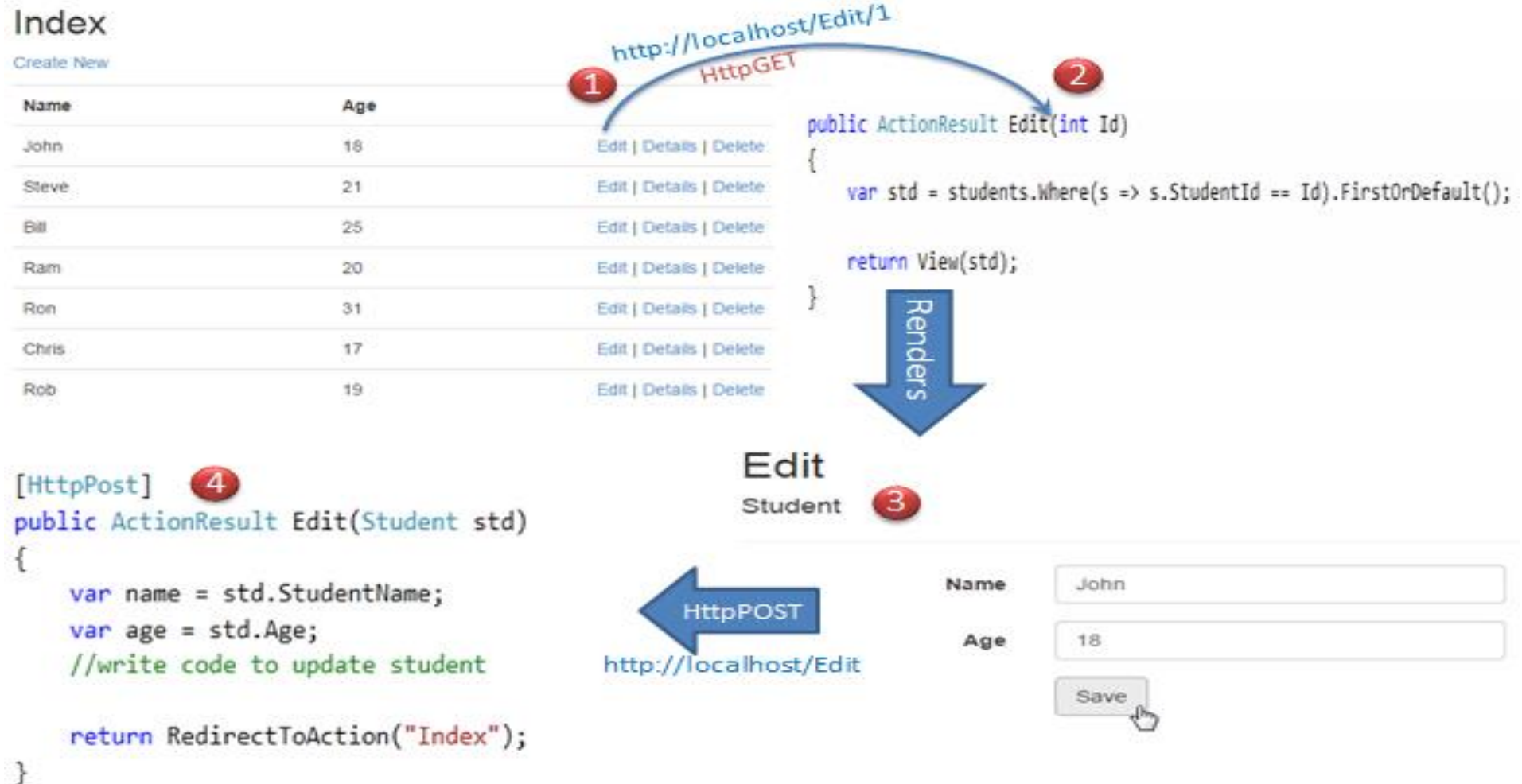
StudentName

Age

[Save](#)

Création d'une vue Edit

- La figure suivante décrit les étapes de la fonctionnalité Edit d'un étudiant :



Création d'une vue Edit

- La dernière étape consiste à ajouter la méthode en version Httppost de l'action Edit dans le contrôleur Student.

```
[HttpPost]
public ActionResult Edit(Student std)
{
    //écrire ici le code de modification d'un Student

    return RedirectToAction("Index");
}
```

Passer les données du contrôleur à la vue

- Dans ASP.NET Core MVC, il existe 3 façons pour transmettre des données d'un contrôleur à une vue:
 - Utilisation de ViewData
 - Utilisation de ViewBag
 - Utilisation d'un objet modèle fortement typé. Ceci est également appelé **vue fortement typée**.
- **Utilisation de ViewData :**
 - ViewData est un dictionnaire qui peut contenir des paires clé-valeur où chaque clé doit être une chaîne.
 - ViewData transfère uniquement les données du contrôleur à la vue, et non l'inverse, en général des données autres que celles du modèle.



Passer les données du contrôleur à la vue

- Définir des objets ViewData dans une méthode d'un contrôleur

```
public ActionResult Details()
{
    Employee emp =
        _employeeRepository.GetEmployee(1);

    ViewData["PageTitle"] = "Employee Details";
    ViewData["Employee"] = emp;

    return View();
}
```

- Accès à l'objet ViewData dans la vue:

```
<html>
<head>
    <title></title>
</head>
<body>
    <h3>@ViewData["PageTitle"]</h3>
```

```
@{
    var employee =
        ViewData["Employee"] as EmployeeManagement.
        Models.Employee;
}

<div>
    Name : @employee.Name
</div>
<div>
    Email : @employee.Email
</div>
<div>
    Department : @employee.Department
</div>
</body>
</html>
```


Passer les données du contrôleur à la vue

- Les données de chaîne sont accessibles à partir du dictionnaire ViewData sans qu'il soit nécessaire de convertir les données en type de chaîne .
- Si nous accédons à tout autre type de données, nous devons les convertir explicitement dans le type attendu.
- Dans notre exemple, nous convertissons l' objet vers le Type Employee afin d'accéder aux propriétés **Name** , **Email** et **Department** de l' objet Employee .
- ViewData est résolu dynamiquement au moment de l'exécution, il ne fournit donc pas de vérification de type à la compilation et, par conséquent, nous n'obtenons pas d'intellisense.

Passer les données du contrôleur à la vue

- **Utilisation de ViewBag :**

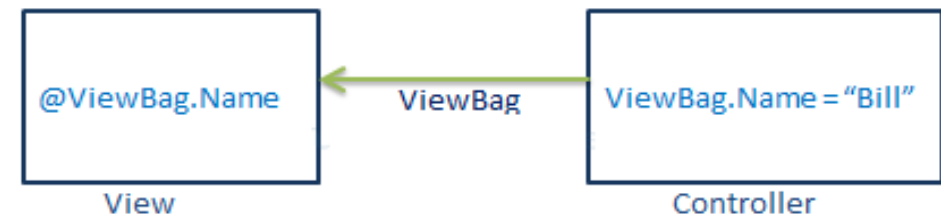
- ViewBag utilise des propriétés dynamiques pour transmettre les données du contrôleur à la vue.
- Les propriétés dynamiques **ViewBag** sont résolues dynamiquement lors de l'exécution.
- ViewBag ne nécessite pas de conversion de type lors de la transmission de ses valeurs.

Exemple :

```
public ActionResult Details()
{
    Employee model = _employeeRepository.GetEmployee(1);

    ViewBag.PageTitle = "Employee Details";
    ViewBag.Employee = model;

    return View();
}
```



Passer les données du contrôleur à la vue

- **Accès aux données d'un ViewBag dans la vue :**

```
<html>
<head>
  <title></title>
</head>
<body>
  <h3>@ViewBag.PageTitle</h3>

  <div>
    Name : @ViewBag.Employee.Name
  </div>
  <div>
    Email : @ViewBag.Employee.Email
  </div>
  <div>
    Department : @ViewBag.Employee.Department
  </div>
</body>
</html>
```

Passer les données du contrôleur à la vue

Vue fortement typée:

- L'approche préférée pour transmettre des données d'un contrôleur à une vue consiste à utiliser une **vue fortement typée**.
- Pour créer une vue fortement typée, dans la méthode d'action du contrôleur, transmettez l'objet modèle comme paramètre de la méthode View ().

```
public ActionResult Details()
{
    Employee emp = _employeeRepository.GetEmployee(1);
    ViewBag.PageTitle = "Employee Details";

    return View(emp);
}
```

- Pour créer une vue fortement typée, spécifiez le type de modèle dans la vue à l'aide de la directive @model.

Passer les données du contrôleur à la vue

@model EmployeeManagement.Models.Employee

```
< html >
< head >
  < title > </ title >
</ head >
< body >
  <h3> @ViewBag.PageTitle </h3>

  <div>
    Nom: @Model.Name
  </div>
  <div>
    Email: @Model.Email
  </ div >
  < div >
    Département : @Model.Department
  </div>
</body>
</html>
```

ViewModels

- Dans certains cas, un objet de modèle peut ne pas contenir toutes les données dont une vue a besoin. C'est à ce moment que nous créons un **ViewModel**.
- Les classes ViewModel sont généralement placés dans un dossier ViewModels.
- les classes **ViewModels** servent à **transférer** des données entre une vue et un contrôleur. Pour cette raison, les ViewModels sont également appelés **objets de transfert de données** ou **DTO**.
- L' exemple suivant montre un ViewModel permettant de passer les détails d'un employé et le titre de la page à la vue à partir d'un contrôleur.

```
public class HomeDetailsViewModel
{
    public Employee Employee { get; set; }

    public string PageTitle { get; set; }
}
```

ViewModels

- **Code du contrôleur :**

```
using EmployeeManagement.Models;

public ActionResult Details()
{
    HomeDetailsViewModel homeDetailsViewModel = new HomeDetailsViewModel()
    {
        Employee = _employeeRepository.GetEmployee(1),
        PageTitle = "Employee Details"
    };

    return View(homeDetailsViewModel);
}
```

ViewModels

- **Code de la vue:**

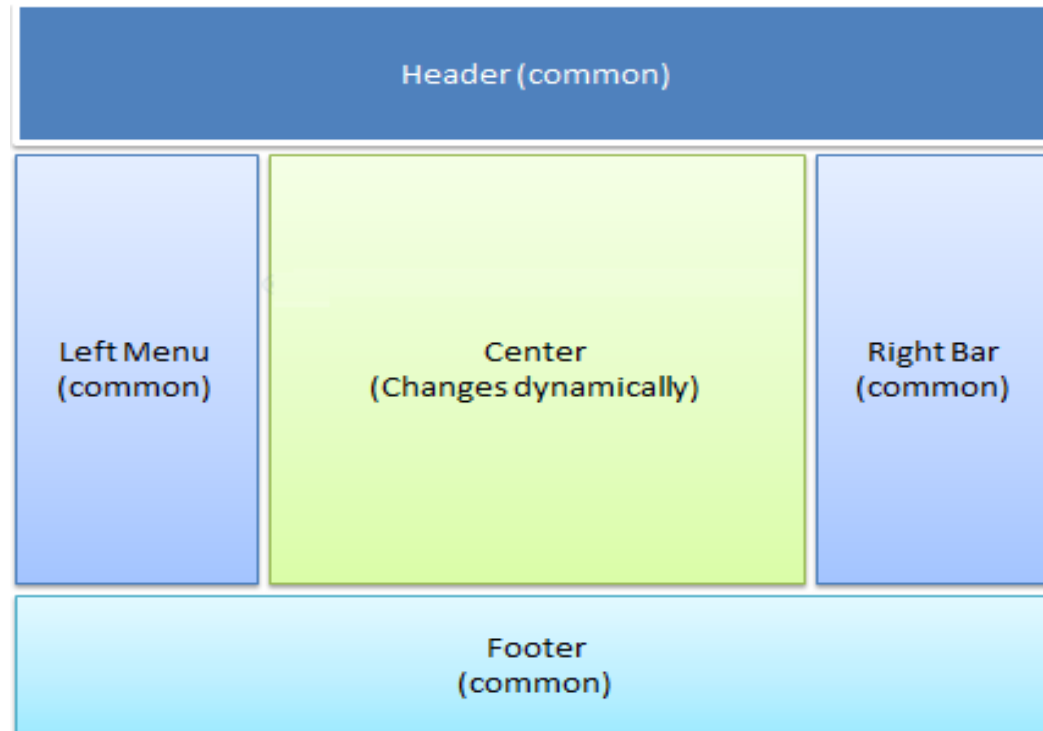
@model EmployeeManagement.ViewModels.HomeDetailsViewModel

```
<html>
<head>
  <title></title>
</head>
<body>
  <h3>@Model.PageTitle</h3>

  <div>
    Name : @Model.Employee.Name
  </div>
  <div>
    Email : @Model.Employee.Email
  </div>
  <div>
    Department : @Model.Employee.Department
  </div>
</body>
</html>
```


Layout Views

- Dans une application web des sections se répètent dans toutes les vues tel que l'en-tête, la barre de navigation gauche, la barre droite ou le pied de page.
- Seule la section centrale change de manière dynamique, comme indiqué ci-dessous.



Layout Views

- Un Layout View est un fichier avec une extension **.cshtml**
- Le Layout View n'est pas spécifique à un contrôleur, placé dans un sous-dossier appelé **"Shared"** dans le dossier **"Views"**
- Par défaut nommé **_Layout.cshtml**.
- Le trait de soulignement au début du nom de fichier indique que ces fichiers ne sont pas destinés à être servis directement par le navigateur.
- Il est également possible d'avoir plusieurs fichiers Layout dans une seule application.

Layout Views

- **Création d'un Layout View:**

1. Faites un clic droit sur le dossier «**Views**» et ajoutez le dossier «**Shared**» .
 2. Faites un clic droit sur le dossier " **Shared** " et sélectionnez "**Ajouter**" - "**Nouvel élément**"
 3. Dans la fenêtre "**Ajouter un nouvel élément**" , recherchez Disposition
 4. Sélectionnez « **Disposition Razor** » et cliquez sur le bouton « **Ajouter** »
 5. Un fichier avec le nom **_Layout.cshtml** sera ajouté au dossier "**Shared**"
- le code HTML généré par défaut dans un nouveau fichier **_Layout.cshtml** :

```
_Layout1.cshtml* X _ViewImports.cshtml _Layout.cshtml
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta name="viewport" content="width=device-width" />
5      <title>@ViewBag.Title</title>
6  </head>
7  <body>
8      <div>
9          @RenderBody()
10     </div>
11 </body>
12 </html>
```

Layout Views

- la vue layout contient un document html classique contenant un head et un body, la seule différence est l'appel aux méthodes `RenderBody ()` et `RenderSection ()`.
- `RenderBody` agit comme un espace réservé pour les autres vues c'est une méthode obligatoire dans un Layout View.
- Par exemple, `Index.cshtml` sera injectée et rendue dans la vue Layout, où la méthode `RenderBody ()` est appelée.
- `RenderSection(string name)` : Rend un contenu de section nommée et spécifie si la section est requise. `RenderSection ()` est facultatif dans layout view.
- L'exemple suivant montre comment attribuer un Layout View à une vue:

Layout Views

```
@model EmployeeManagement.ViewModels.HomeDetailsViewModel
```

```
@ {  
    Layout = "~ /Views/Shared/_Layout.cshtml" ;  
    ViewBag.Title = "Détails de l'employé" ;  
}
```

```
< h3 > @Model.PageTitle </ h3 >
```

```
< div >  
    Nom: @Model.Employee.Name
```

```
</ div >
```

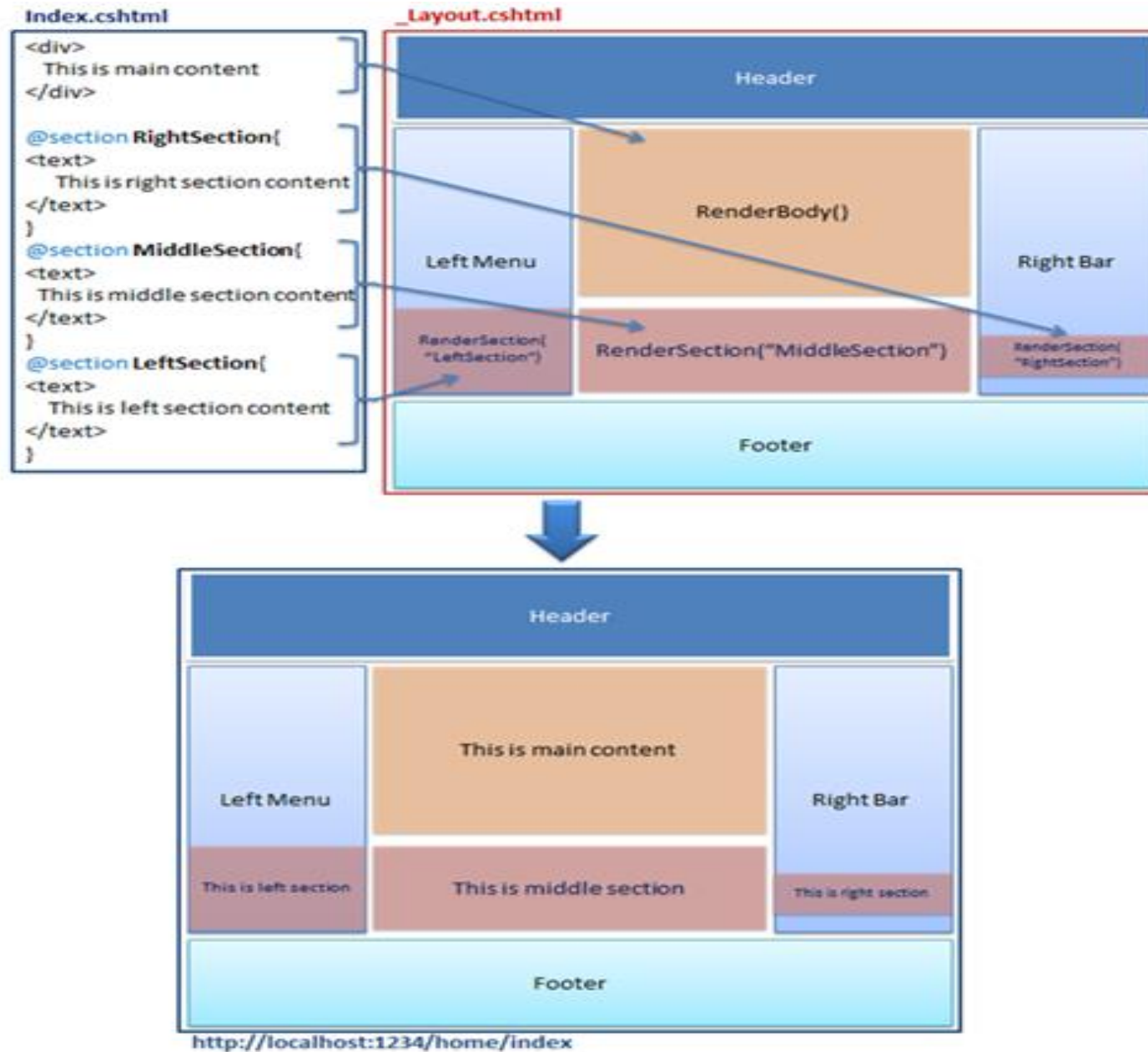
```
< div >  
    Courriel: @Model.Employee.Email
```

```
</ div >
```

```
< div >  
    Département: @Model.Employee.Department
```

```
</ div >
```

Les Sections dans un Layout View



Les Sections dans un Layout View

- Une page de Layout dans ASP.NET Core MVC peut également inclure une ou plusieurs sections.
- Une section peut être facultative ou obligatoire.
- Il fournit un moyen d'organiser où certains éléments de page doivent être placés.

Exemple:

- Vous disposez d'un fichier JavaScript personnalisé qui n'est nécessaire que pour quelques vues de votre application.
- Il est recommandé de placer les fichiers de script au bas de la page avant la balise de fermeture `</ body >`

Les Sections dans un Layout View

- Si la section n'est requis que pour certaines vues, il faut utiliser la méthode **RenderSection()** avec le paramètre `required` à `false`.

```
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
</head>
<body>
  <div>
    @RenderBody()
  </div>

  @RenderSection("Scripts", required: false)
</body>
</html>
```


Les Sections dans un Layout View

Fournir le contenu de la section :

- Chaque vue qui a l'intention de fournir du contenu pour la section doit inclure une section portant le même nom.
- Nous utilisons la directive **@section** pour inclure la section et fournir le contenu comme indiqué ci-dessous.
- Dans l'exemple suivant la vue Details.cshtml donne une définition de contenu à la section

Les Sections dans un Layout View

```
@model EmployeeManagement.ViewModels.HomeDetailsViewModel
```

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

```
<h3>@Model.PageTitle</h3>
```

```
<div>
```

```
    Name : @Model.Employee.Name
```

```
</div>
```

```
<div>
```

```
    Email : @Model.Employee.Email
```

```
</div>
```

```
<div>
```

```
    Department : @Model.Employee.Department
```

```
</div>
```

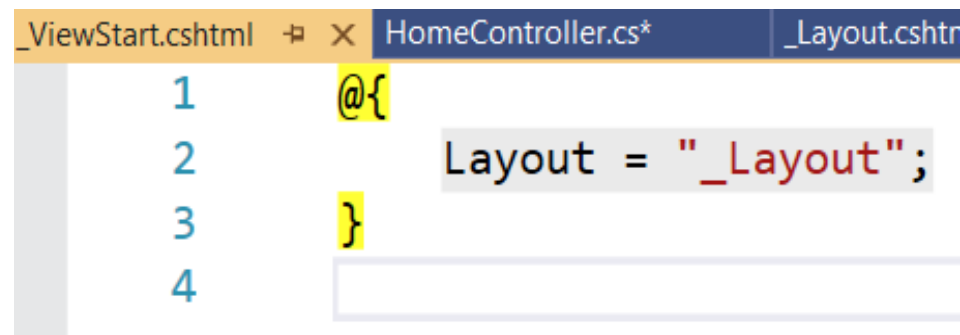
```
@section Scripts {
```

```
    <script src="~/js/CustomScript.js"></script>
```

```
}
```

La vue _ViewStart.cshtml

- Dans une vue, la propriété Layout sert à associer une vue à un Layout View.
- Sans le fichier _ViewStart.cshtml , on doit définir la propriété Layout dans chaque vue.
- D'où : Code redondant et maintenance difficile.
- _ViewStart.cshtml est un fichier spécial dans ASP.NET Core MVC. Le code de ce fichier est exécuté avant que le code d'une vue individuelle ne soit exécuté.
- Au lieu de définir la propriété Layout dans chaque vue individuelle, nous pouvons déplacer ce code dans le fichier _ViewStart.cshtml.
- Le contenu par défaut de ce fichier est le suivant:



```
1 @{
2     Layout = "_Layout";
3 }
4
```

La vue _ViewStart.cshtml

Comment Changer la vue Layout selon le besoin?

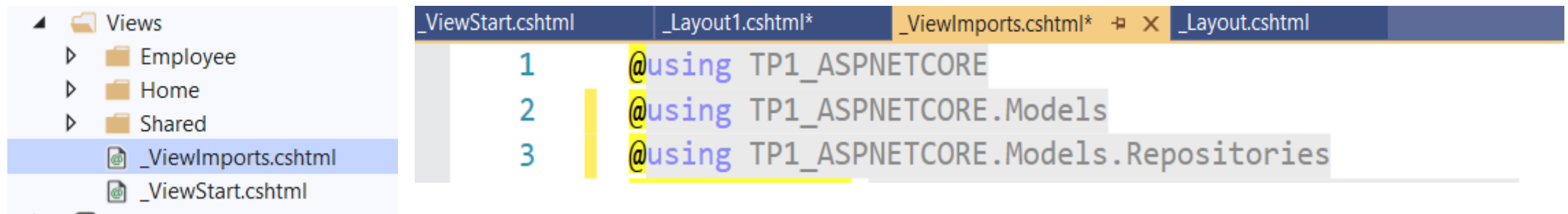
- Dans une application ASP.NET Core MVC, nous pouvons avoir plusieurs vues de Layout.
- Supposons qu'on a 2 vues de Layout dans notre application : _AdminLayout.cshtml et _NonAdminLayout.cshtml. On peut choisir une vue Layout selon le rôle de l'utilisateur dans _ViewStart.cshtml.

Exemple :

```
@{  
    if (User.IsInRole("Admin"))  
    {  
        Layout = "_AdminLayout";  
    }  
    else  
    {  
        Layout = "_NonAdminLayout";  
    }  
}
```

_ViewImports.cshtml

- Le fichier **_ViewImports.cshtml** est généralement placé dans le dossier **Views** .
- Il est utilisé pour inclure les espaces de noms communs afin que nous n'ayons pas à les inclure dans chaque vue qui a besoin de ces espaces de noms.



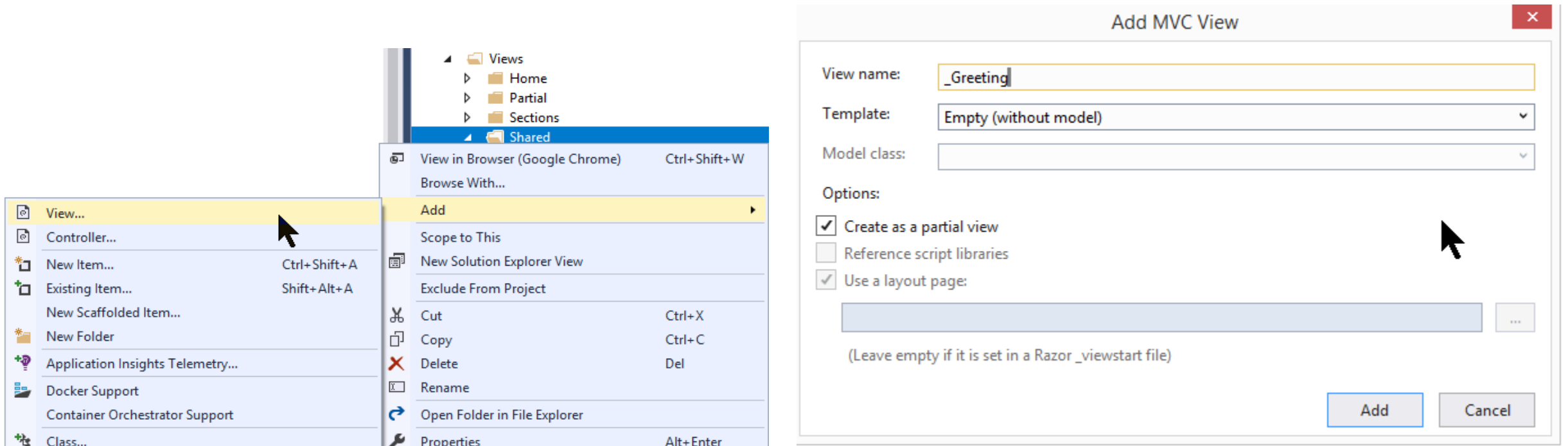
- la directive @using est utilisée pour inclure les espaces de noms communs.
- Le fichier _ViewImports prend également en charge les directives suivantes :
 - @addTagHelper
 - @removeTagHelper
 - @tagHelperPrefix
 - @model
 - @inherits
 - @inject

Partial Views

- La vue partielle est une vue réutilisable, qui peut être utilisée comme vue enfant dans plusieurs autres vues.
- Elimine la double écriture de code en réutilisant la même vue partielle à plusieurs endroits.
- Vous pouvez utiliser la vue partielle dans la vue Layout, ainsi que d'autres vues de contenu.
- Placée généralement dans le dossier Shared et son nom commence par _ pour le distinguer des vues classiques.

Partial Views

Création de vue partielle:



Ajoutons le code suivant à la vue partielle créée :

```
<div>Hello, world!</div>  
<div>Today is @DateTime.Now.ToString()</div>
```

Partial Views

- Vous pouvez appeler la vue partielle dans une vue parent à l'aide de la méthode HTML helper: `Html.Partial()`

Exemple : code de la vue Index.cshtml

```
@{  
    ViewData["Title"] = "Index";  
}
```

```
<h2>Index</h2>
```

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

```
<span>More stuff here....</span>
```

Appel Asynchrone : `@await Html.PartialAsync("_Greeting")`

Le Moteur de vues Razor

- Razor est un moteur de vues pris en charge dans ASP.NET MVC.
- Permet d'écrire un mélange de code HTML et de code côté serveur à l'aide de C# ou de Visual Basic.
- La syntaxe Razor présente les caractéristiques suivantes :
 - Compacte : la syntaxe Razor est compacte, ce qui vous permet de réduire le code à écrire.
 - Facile à apprendre.
 - Intellisense : la syntaxe Razor prend en charge la complétion des instructions dans Visual Studio.

Syntaxe Razor

- **Expression en ligne**

Commencez par le symbol @ pour écrire du code C# côté serveur intégré avec du code HTML.

```
<h1>Razor syntax demo</h1>
```

```
<h2>@DateTime.Now.ToShortDateString()</h2>
```

- **Bloc de code multi-instruction**

Vous pouvez écrire plusieurs lignes de code côté serveur entre accolades @{ ... }. Chaque ligne doit se terminer par un point-virgule identique à C #.

```
@{
```

```
    var date = DateTime.Now.ToShortDateString();
```

```
    var message = "Hello World";
```

```
}
```

```
<h2>Today's date is: @date </h2>
```

```
<h3>@message</h3>
```

Syntaxe Razor

- **Affichage de texte à partir de bloc de code RAZOR**

Utiliser le symbole @: ou <text>/<text> pour afficher du texte dans le bloc de code.

```
@{  
    var date = DateTime.Now.ToShortDateString();  
    string message = "Hello World!";  
    @:Today's date is: @date <br />  
    @message  
}
```

Avec la balise <text> dans le code RAZOR

```
@{ var date = DateTime.Now.ToShortDateString();  
    string message = "Hello World!";  
    <text>Today's date is:</text> @date <br />  
    @message  
}
```

Syntaxe Razor

- **Structure conditionnelle if-else**

Utiliser le symbole @ pour décrire une structure conditionnelle if mais le corps du bloc if et celui de else doit être mis entre accolades.

```
@if(DateTime.IsLeapYear(DateTime.Now.Year) )  
{ @DateTime.Now.Year @:is a leap year.// Année bissextile  
}  
else  
{ @DateTime.Now.Year @:is not a leap year.  
}
```

Syntaxe Razor

- **La boucle for**

```
@for(int i = 0; i < 5; i++)  
{  
    @i.ToString() <br />  
}
```

- **Utiliser un modèle du projet avec le mot clé : @Model**

```
@model Student  
<h2>Student Detail:</h2>  
<ul>  
    <li>Student Id: @Model.StudentId</li>  
    <li>Student Name: @Model.StudentName</li>  
    <li>Age: @Model.Age</li>  
</ul>
```

Syntaxe Razor

- Déclarer des variables

```
@{  
    string str = "";  
    if(1 > 0)  
    {  
        str = "Hello World!";  
    }  
}  
<p>@str</p>
```

Syntaxe Razor

- **Création d'une liste**

```
@{  
List<string> names = new List<string>()  
{  
    "John",  
    "Jane",  
    "Joe",  
    "Jenna",  
    "Doggy"  
};  
}
```

- **Parcours de la liste avec for**

```
<ul>  
@for (int i = 0; i < names.Count; i++)  
{  
    <li>@names[i]</li>  
}  
</ul>
```

- **Parcours de liste avec Foreach**

```
<ul>  
@foreach (string name in names)  
{  
    <li>@name</li>  
}  
</ul>
```

Syntaxe Razor

- **Parcours de la liste avec boucle While**

```
<ul>
@{
    int counter = 0;
}
@while(counter < names.Count)
{
    <li>@names[counter++]</li>
}
</ul>
```

- **Parcours avec la boucle Do While**

```
<ul>
@{
    counter = 0;
}
@do
{
    <li>@names[counter++]</li>
```

```
} while (counter < names.Count);
</ul>
```

- **Utilisation de break**

```
<ul>
@for (int i = 0; i < names.Count; i++)
{
    <li>@names[i]</li>
    @if(i >= 2)
    {
        <li>...Texte à afficher</li>
        break;
    }
}
</ul>
```


Syntaxe Razor

- **Utilisation de switch**

```
@switch(DateTime.Now.DayOfWeek)
```

```
{ case DayOfWeek.Monday:
```

```
<span>working....</span>
```

```
break;
```

```
case DayOfWeek.Friday:
```

```
<span>Weekend coming up!</span>
```

```
break;
```

```
case DayOfWeek.Saturday:
```

```
case DayOfWeek.Sunday:
```

```
<span>Finally weekend!</span>
```

```
break;
```

```
default: <span>Nothing special about this day...</span> break;
```

```
}
```

TAG Helpers

- Les Tag Helpers sont des composants côté serveur. Ils sont traités sur le serveur pour créer et rendre des éléments HTML dans des fichiers Razor.
- Les Tag Helpers sont similaires aux assistants HTML Helpers. Mais ne les remplacent pas toutes.
- Il existe de nombreux Tag Helpers intégrés pour les tâches courantes telles que la génération de liens, la création de formulaires, le chargement d'éléments, etc.
- Pour pouvoir utiliser les Tag Helpers, il faut importer le namespace nécessaire via la directive **@addTagHelper**.
- Ajouter la directive suivante dans le fichier `_ViewImports.cshtml` :
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
- Le caractère générique `*` indique que nous voulons importer tous les Tag Helpers. `Microsoft.AspNetCore.Mvc.TagHelpers` est l'assembly qui contient les tag helpers intégrés.

Anchor TAG Helper

- Anchor Tag Helper permet de créer des liens hypertextes et améliore l'ancrage HTML standard (`< a ... > < /a >`) en ajoutant de nouveaux attributs tels que :
 - `asp-controller` : Nom du contrôleur.
 - `asp-action` : Nom de la méthode d'action à appeler.
 - `asp-route-{value}` : paramètre de route à ajouter si nécessaire.
- Pour créer un lien hypertexte afin d'appeler une méthode d'action d'un contrôleur donné et un paramètre de route Id, on peut utiliser le Tag Helper suivant:

```
<a asp-controller="home" asp-action="details"
    asp-route-id="@employee.Id">View</a>
```
- Avec les Html Helpers le code devient :

```
@Html.ActionLink("Details", "details", "home", new { id = employee.Id })
```

Form TAG Helper

- Le Form Tag Helper permet la génération des balises FORM pour la création de formulaires et renvoyer les données au serveur.
- Vous pouvez facilement référencer le contrôleur et l'action à laquelle soumettre le FORMULAIRE, ou même référencer une route.
- Pour créer un formulaire, on a besoin généralement des Tag Helpers suivants:
 - Form Tag Helper
 - Label Tag Helper
 - Input Tag Helper
 - Select Tag Helper
 - Textarea Tag Helper

Form TAG Helper

- Pour créer un formulaire comme celui-ci on a besoin tout d'abord de créer un Form Tag Helper:

```
<form asp-controller="home"  
      Asp-action="create" method="post">  
</form>
```

- asp-controller : Nom du contrôleur.
- Asp-action : Nom de l'action à appeler.
- Ce code génère le Html suivant :

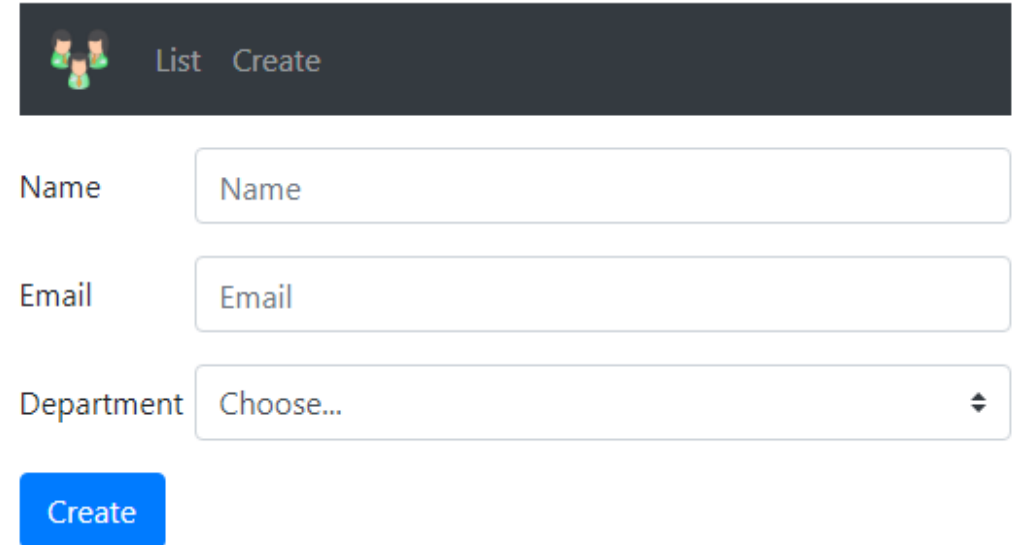
```
<form method="post" action="/home/create"></form>
```

- Spécifier une route personnalisée:

```
<form method="post" asp-route="MyRouteName"></form>
```

- Ajouter un paramètre de route:

```
<form method="post" asp-route=" MyRouteName " asp-route-id="@Model.Id"></form>
```



The screenshot shows a web form with a dark header bar containing a group of three people icon, 'List', and 'Create' links. Below the header, there are three input fields: 'Name', 'Email', and 'Department' (a dropdown menu with 'Choose...' selected). A blue 'Create' button is positioned below the 'Department' field.

Input TAG Helper

- Input Tag Helper permet de lier une balise Html <input> à un modèle de données.
- La vue doit incorporer la directive @Model pour définir un modèle pour la vue.
- Exemple : @Model Employee
- Pour faire un Bind de la zone de texte avec la propriété Name de la classe Employee on utilise l'attribut asp-for: `<input asp-for="Name" placeholder="Name">`
- La balise Html générée est la suivante :

```
<input placeholder="Name" type="text" id="Name" name="Name" value="">
```

Textarea Tag Helper

- Utiliser la balise Textarea à la place de Input si la saisie d'une zone de texte nécessite plusieurs lignes.
- Comme Input Tag Helper , utiliser l'attribut for pour lier la zone à une propriété de le classe de modèle.

```
<textarea asp-for="Description"></textarea>
```

- Ce qui génère le HTML suivant :

```
<textarea id="Description" name="Description"></textarea>
```

- On peut paramétrer la taille de la zone Textarea et donner un style

```
<textarea asp-for="Description" cols="25" rows="3"></textarea>
```

```
<textarea asp-for="Description" style="height: 40px; width: 200px;"></textarea>
```

- Avec les classes CSS vous pouvez donner plus d'options de style.

Label TAG Helper

- Le Label Tag Helper génère une étiquette avec l'attribut **for**.
- L'attribut **for** lie l'étiquette à son élément d'entrée associé.

```
< label asp-for = " Nom "> </ label >  
< input asp-for = " Nom ">
```

- Ce code génère le code Html suivant:

```
< label for = "Name"> Nom </ label >  
< input type = "text" id = "Name" name = "Name" value = "">
```


Select Tag Helper

- Génère l'élément de **sélection** et ses éléments d'option associés.
- Dans notre cas, nous voulons qu'un élément de sélection affiche la liste des départements.
- Cette liste peut être récupérée à partir d'une énumération ou une base de données. Faisons un exemple avec une énumération.

```
namespace EmployeeManagement.Models
{
    public enum Dept
    {
        None,
        HR,
        Finance,
        IT
    }
}
```

Select Tag Helper

- Modifier la classe Employee, la propriété Department de type énumération Dept.
- Dans la vue Create.cshtml ajouter le code suivant :
- L'attribut asp-items définit la liste de valeurs à afficher dans <select>

```
namespace EmployeeManagement.Models
{
    public class Employee
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Email { get; set; }
        public Dept Department { get; set; }
    }
}
```

Select Tag Helper

- Code de la vue:

```
<label asp-for="Department"></label>  
<select asp-for="Department"  
  asp-items="Html.GetEnumSelectList<Dept>()"></select>
```

- En cas d'utilisation d'une liste de valeurs dans une énumération, l'attribut asp-items définit cette enum via le HTML Helper : `Html.GetEnumSelectList<enum>()`
- En cas où la liste des valeurs est stockée dans un objet collection comme le type `List` utiliser la méthode `SelectList(Nom_Liste)`

```
<select asp-items="@ (new SelectList(Model.Countries))" asp-for="User.Country">  
</select>
```

Select Tag Helper

- Le code de tout le formulaire sans classes Bootstrap est le suivant :

```
@model Employee
```

```
@{  
    ViewBag.Title = "Create Employee";  
}
```

```
<form asp-controller="home" asp-action="create" method="post">
```

```
    <div>  
        <label asp-for="Name"></label>  
        <input asp-for="Name" />  
    </div>
```

```
    <div>  
        <label asp-for="Email"></label>  
        <input asp-for="Email">  
    </div>
```

```
    <div>  
        <label asp-for="Department"></label>  
        <select asp-for="Department"  
            asp-items="Html.GetEnumSelectList<Dept>()"></select>  
    </div>
```

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

Select Tag Helper

- Le code de tout le formulaire avec classes Bootstrap est le suivant :

@model Employee

```
@{
    ViewBag.Title = "Create Employee";
}
<form asp-controller="home" asp-action="create" method="post" class="mt-3">
    <div class="form-group row">
        <label asp-for="Name" class="col-sm-2 col-form-label"></label>
        <div class="col-sm-10">
            <input asp-for="Name" class="form-control" placeholder="Name">
        </div>
    </div>
    <div class="form-group row">
        <label asp-for="Email" class="col-sm-2 col-form-label"></label>
        <div class="col-sm-10">
            <input asp-for="Email" class="form-control" placeholder="Email">
        </div>
    </div>
    <div class="form-group row">
        <label asp-for="Department" class="col-sm-2 col-form-label"></label>
        <div class="col-sm-10">
            <select asp-for="Department" class="custom-select mr-sm-2"
                asp-items="Html.GetEnumSelectList<Dept>()"></select>
        </div>
    </div>
    <div class="form-group row">
        <div class="col-sm-10">
            <button type="submit" class="btn btn-primary">Create</button>
        </div>
    </div>
</form>
```