Due November 25, 9AM

Contents

[View Models: Hint for Assignment 1 1](#_Toc467136282)

[Creating a List View 2](#_Toc467136283)

[Creating a Details View 4](#_Toc467136284)

[Creating an Update View 7](#_Toc467136285)

[Hyperlinks 12](#_Toc467136286)

[Redirecting to a Different ActionResult Method 13](#_Toc467136287)

[Validation 13](#_Toc467136288)

[Enabling Server Side Validation 14](#_Toc467136289)

[Enabling Client Side Validation 14](#_Toc467136290)

[Disabling Client Side Validation 16](#_Toc467136291)

[Customizing Labels 17](#_Toc467136292)

[Partial Views 17](#_Toc467136293)

[Creating a Partial View for a Specific Controller 17](#_Toc467136294)

[Creating a Partial View for all Controllers 17](#_Toc467136295)

[Implicitly (Weakly) Typed Partial Views 17](#_Toc467136296)

[Strongly Typed Partial Views 17](#_Toc467136297)

[All Cases 17](#_Toc467136298)

## View Models: Hint for Assignment 1

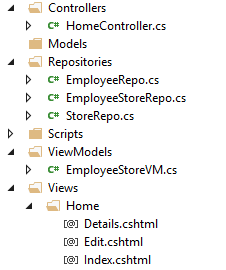
Walking through the steps in this section and taking time to understand them will help with assignment 1 in ASP.NET.

Last day we discussed how ‘models’ allow you to create a custom type. We also saw how we can use the Entity Framework to generate models for us. We also discussed how models only define a type and they do not contain logic.

Sometimes you will need to manage properties from more than one entity within a view. A popular way to accomplish this task is with a ‘view model’. On that note we will discuss how to implement the MVVM pattern which stands for model – view – view model.

Example : Managing Complex Types

This example demonstrates how to create and use a view model. To start, create a web application with ‘empty’ and ‘mvc’ options selected. Create an entity data model that references the FoodStore database. You do not need to add the following files yet, just keep in mind that we are going to build an application with the following structure:



Now add the first file. For this step, in the ViewModels folder add an EmployeeStoreVM class to it. Here is the code for our class. This view model contains properties that will be populated using data from the Employee and the Store tables.

|  |
| --- |
| public class EmployeeStoreVM {  public int EmployeeID { get; set; }  public string LastName { get; set; }  public string FirstName { get; set; }  public string Branch { get; set; }  public string Region { get; set; }  public string BuildingName { get; set; }  public int UnitNum { get; set; }  } |

### Creating a List View

Remembering that view models do not contain logic, we use repositories to manage the logic.

#### Creating the Repository

To stick with this pattern, create a folder named ‘Repositories’ and add a class named ‘EmployeeStoreRepo’ to it. The code needed to generate an IEnumerable listing of EmployeeStoreVM objects is contained below. Notice how the query performs a join using navigation properties as highlighted in green. This join with navigation properties is possible since the join is going from many-to-one (Employees to Stores). Notice that the properties highlighted in purple for each object in our list are defined in the Select() method. Since we are using the EmployeeStoreVM type highlighted in yellow, its properties must match the properties of our EmployeeStoreVM class that was defined above.

|  |
| --- |
| public IEnumerable<EmployeeStoreVM> GetAll() {  FoodStoreEntities db = new FoodStoreEntities();  IEnumerable<EmployeeStoreVM> esList  = db.Employees.Where(es=>es.Store.branch == es.branch)  // Assign properties within the 'Select' statement.  // Notice how we 'must' use the 'EmployeeStoreVM' type.  .Select(es => new EmployeeStoreVM() {  EmployeeID = es.employee\_id,  LastName = es.last\_name,  FirstName = es.first\_name,  Branch = es.branch,  Region = es.Store.region,  // Handle null values. 1st option selected if T otherwise 2nd if F.  BuildingName =  es.Store.building\_name == null?"":  es.Store.building\_name,    // Must handle null because a null exists for  // unit\_num in the database.  // Get integer if it exists otherwise get 0.  UnitNum = es.Store.unit\_num == null ? 0 :  (int)es.Store.unit\_num  });  return esList;  } |

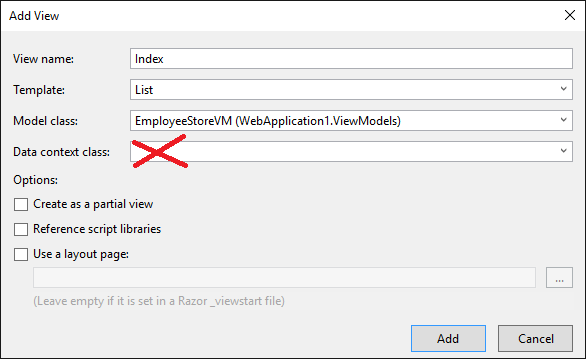
#### Adding the Action Method

Next, add an empty HomeController. Replace the Index() action method with the following code:

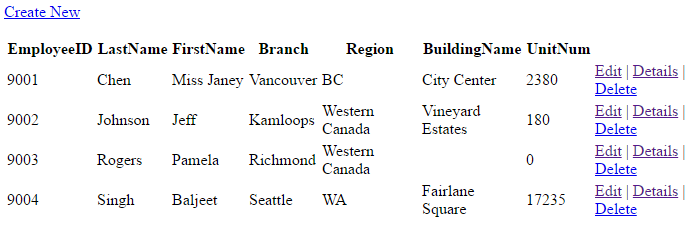
|  |
| --- |
| public ActionResult Index()  {  EmployeeStoreRepo esRepo = new EmployeeStoreRepo();  IEnumerable<EmployeeStoreVM> es = esRepo.GetAll();  return View(es);  } |

#### Adding a List View

To create a List view with our new type, select the List template and choose EmployeeStoreVM for the type. The EmployeeStoreVM is not a database object so do not select a database context.



After adding the view with the wizard, if you run the project you will see output similar to the following:



### Creating a Details View

To support our EmployeeStoreVM type further we must remember that it is pulling data from two separate entities which are Store and Employee. Chances are many of the queries, updates and deletes must be broken down specifically for the Store and Employee table.

#### Creating the Repositories

So, to set this up first create an **EmployeeRepo** class within the Repositories folder. Then, place the following code in it to return one Employee when the id is supplied:

|  |
| --- |
| public Employee Get(int id) {  FoodStoreEntities db = new FoodStoreEntities();  return db.Employees.Where(e=>e.employee\_id == id)  .FirstOrDefault();  } |

Next, create a **StoreRepo** class to manage the CRUD for the Store entity. Then, place this method in the class to return the Store object when its unique identifier is passed to it:

|  |
| --- |
| public Store Get(string branch) {  FoodStoreEntities db = new FoodStoreEntities();  return db.Stores.Where(s=>s.branch == branch)  .FirstOrDefault();  } |

Now back to our **EmployeeStoreVMRepo** class. Since we already have code in place to retrieve the Store and Employee objects by their unique identifies, we can ‘and should’ use it to populate our ‘EmployeeStoreVM’ object. The following Get() method returns the EmployeeStoreVM object when an employeeID and branch is passed to it.

|  |
| --- |
| public EmployeeStoreVM Get(int employeeID, string branch) {  // Get Employee from EmployeeRepo.  EmployeeRepo employeeRepo = new EmployeeRepo();  Employee employee = employeeRepo.Get(employeeID);  // Get Store from StoreRepo.  StoreRepo storeRepo = new StoreRepo();  Store store = storeRepo.Get(branch);  // Merge data into custom view model object.  EmployeeStoreVM esVM = new EmployeeStoreVM {  EmployeeID = employee.employee\_id,  LastName = employee.last\_name,  FirstName = employee.first\_name,  Branch = store.branch,  BuildingName = store.building\_name == null?"":  store.building\_name,  Region = store.region,  // Need condition to handle null  UnitNum = store.unit\_num == null ? 0 :  (int)store.unit\_num  };  return esVM;  } |

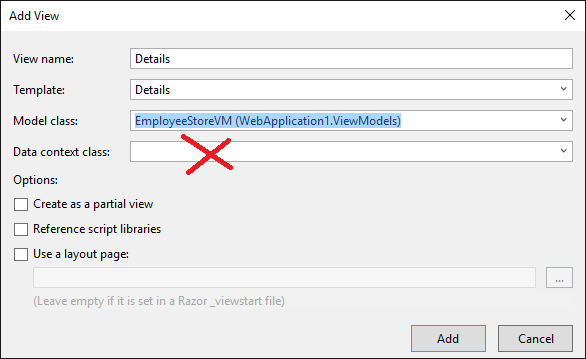
#### Adding the Action Method

Next, add this action method which calls the Get() method in our EmployeeStoreRepo class which in turn pulls data from the Store and Employee repository classes:

|  |
| --- |
| public ActionResult Details(int employeeID, string branch) {  EmployeeStoreRepo esRepo = new EmployeeStoreRepo();  EmployeeStoreVM esVM = esRepo.Get(employeeID, branch);  return View(esVM);  } |

#### Adding the Details View

Use the wizard to generate a details view which shows only one instance of the given type. We can use our EmployeeStoreVM model in this case. Remember not to select a data context because this type is not a database entity – it is a custom type that we defined in our code.

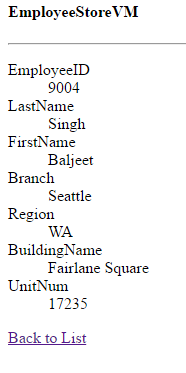


#### Setting the Hyperlink

To configure our hyperlink to the details view for the object, go to the list view. Replace the action link with the following. Notice how the parameter names in green below match the names in green within the header for the action method above.

|  |
| --- |
| @Html.ActionLink("Details", // Set text  "Details", // Set action method.  new { controller = "Home", // Set controller.  employeeID = item.EmployeeID, // Set parameters if they exist.  branch = item.Branch}) |

If you ran the project now you would see the detail view whenever you click on the Details action link. The details view out of the box can use some improvement. We will examine how to automate some of the improvements to the appearance in the near future:



### Creating an Update View

Now we can create an area for updating the columns that are in our view model. Remember, updating the view model properties usually means that you are updating column values in more than one table. So to avoid code duplication we must add some code to the appropriate repositories.

#### Creating the Repositories

This code can be placed in our EmployeeRepo class to allow a user to update the Employee’s first and last name:

|  |
| --- |
| public bool Update(int id, string first,  string last) {  FoodStoreEntities db = new FoodStoreEntities();  Employee employee = db.Employees  .Where(e=>e.employee\_id == id)  .FirstOrDefault();  // Remember you can't update the primary key without  // causing trouble. Just update the first and last names  // for now.  employee.first\_name = first;  employee.last\_name = last;  db.SaveChanges();  return true;  } |

This code can be placed in the StoreRepo class to allow updates to a Store’s region.

|  |
| --- |
| public bool Update(string branch, string region) {  FoodStoreEntities db = new FoodStoreEntities();  Store store = db.Stores.Where(s=>s.branch == branch)  .FirstOrDefault();  store.region = region;  db.SaveChanges(); // Commit changes to database.  // Error handling code goes here.  return true;  } |

Now we can add an Update() method to our EmployeeStoreRepo which calls on the EmployeeRepo and StoreRepo separately to perform updates to each table:

|  |
| --- |
| public bool Update(EmployeeStoreVM esVM) {  // Updating our ViewModel really requires updates to  // two separate tables.  // Update the 'Store'.  StoreRepo storeRepo = new StoreRepo();  storeRepo.Update(esVM.Branch, esVM.Region);  // Update the 'Employee'.  EmployeeRepo empRepo = new EmployeeRepo();  empRepo.Update(esVM.EmployeeID, esVM.FirstName, esVM.LastName);  // Error handling could go here and if problems are encountered  // 'false' could be returned.  // Otherwise if things go well return true.  return true;  } |

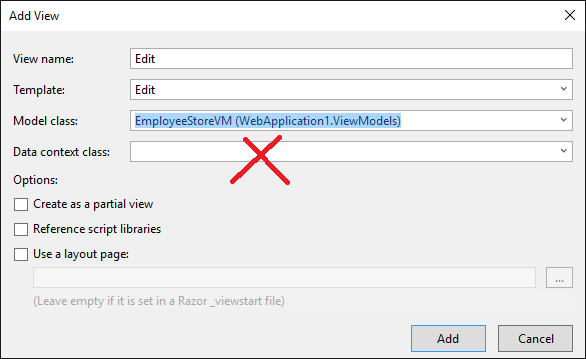
#### Adding the Action Method

We need two action methods. The first action method is used when the user selects a hyperlink to the edit page. It calls the EmployeeStoreRepo and returns the data for the object that is identified with the specified employeeID and branch. The second action method is used when the user posts changes in the object to the action method. Notice how the RedirectToAction method calls a different method in the application:

|  |
| --- |
| // This method is called when the user arrives at the edit page.  [HttpGet]  public ActionResult Edit(int employeeID, string branch) {  EmployeeStoreRepo esRepo = new EmployeeStoreRepo();  EmployeeStoreVM esVM = esRepo.Get(employeeID, branch);  return View(esVM);  }  // This method is called when the user clicks the submit  // button from the edit page.  [HttpPost]  public ActionResult Edit(EmployeeStoreVM esVM) {  EmployeeStoreRepo esRepo = new EmployeeStoreRepo();  esRepo.Update(esVM);  // go to index action method of the home controller.  return RedirectToAction("Index", "Home");  } |

#### Creating an Edit View

To create an edit view, right click the Edit action method and choose the Edit template for the EmployeeStoreVM. Do not choose the database context since we are not working with a database entity.



#### Disabling Edits to Specific Columns

Next in the edit view, we will only disable edits for everything except last name, first name and region. See the red highlights.

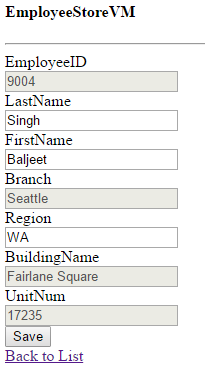
|  |
| --- |
| @model WebApplication1.ViewModels.EmployeeStoreVM  @{  Layout = null;  }  <!DOCTYPE html>  <html>  <head>  <meta name="viewport" content="width=device-width" />  <title>Edit</title>  </head>  <body>  @using (Html.BeginForm())  {  @Html.AntiForgeryToken()    <div class="form-horizontal">  <h4>EmployeeStoreVM</h4>  <hr />  @Html.ValidationSummary(true, "", new { @class = "text-danger" })  <div class="form-group">  @Html.LabelFor(model => model.EmployeeID, htmlAttributes: new { @class = "control-label col-md-2" })  <div class="col-md-10">  @Html.EditorFor(model => model.EmployeeID, new { htmlAttributes = new { @class = "form-control", @disabled = "disabled" } })  @Html.ValidationMessageFor(model => model.EmployeeID, "", new { @class = "text-danger" })  </div>  </div>    <div class="form-group">  @Html.LabelFor(model => model.LastName, htmlAttributes: new { @class = "control-label col-md-2" })  <div class="col-md-10">  @Html.EditorFor(model => model.LastName, new { htmlAttributes = new { @class = "form-control" } })  @Html.ValidationMessageFor(model => model.LastName, "", new { @class = "text-danger" })  </div>  </div>    <div class="form-group">  @Html.LabelFor(model => model.FirstName, htmlAttributes: new { @class = "control-label col-md-2" })  <div class="col-md-10">  @Html.EditorFor(model => model.FirstName, new { htmlAttributes = new { @class = "form-control" } })  @Html.ValidationMessageFor(model => model.FirstName, "", new { @class = "text-danger" })  </div>  </div>    <div class="form-group">  @Html.LabelFor(model => model.Branch, htmlAttributes: new { @class = "control-label col-md-2", @disabled = "disabled" })  <div class="col-md-10">  @Html.EditorFor(model => model.Branch, new { htmlAttributes = new { @class = "form-control", @disabled = "disabled" } })  @Html.ValidationMessageFor(model => model.Branch, "", new { @class = "text-danger" })  </div>  </div>    <div class="form-group">  @Html.LabelFor(model => model.Region, htmlAttributes: new { @class = "control-label col-md-2" })  <div class="col-md-10">  @Html.EditorFor(model => model.Region, new { htmlAttributes = new { @class = "form-control"} })  @Html.ValidationMessageFor(model => model.Region, "", new { @class = "text-danger" })  </div>  </div>    <div class="form-group">  @Html.LabelFor(model => model.BuildingName, htmlAttributes: new { @class = "control-label col-md-2" })  <div class="col-md-10">  @Html.EditorFor(model => model.BuildingName, new { htmlAttributes = new { @class = "form-control", @disabled = "disabled" } })  @Html.ValidationMessageFor(model => model.BuildingName, "", new { @class = "text-danger" })  </div>  </div>    <div class="form-group">  @Html.LabelFor(model => model.UnitNum, htmlAttributes: new { @class = "control-label col-md-2" })  <div class="col-md-10">  @Html.EditorFor(model => model.UnitNum, new { htmlAttributes = new { @class = "form-control", @disabled = "disabled" } })  @Html.ValidationMessageFor(model => model.UnitNum, "", new { @class = "text-danger" })  </div>  </div>    <div class="form-group">  <div class="col-md-offset-2 col-md-10">  <input type="submit" value="Save" class="btn btn-default" />  </div>  </div>  </div>  }    <div>  @Html.ActionLink("Back to List", "Index")  </div>  </body>  </html> |

#### Setting the Hyperlink

Finally, to enable the hyperlink to the edit page replace the existing link in the list view with the following:

|  |
| --- |
| @Html.ActionLink(  "Edit", // Set the text  "Edit", // Set action method  new {  controller = "Home", // Set controller  employeeID = item.EmployeeID, // Set parameters if they exist.  branch = item.Branch  }) |

After the changes have been implemented we are presented with a page where we can edit an employees last name, first name and region when the user navigates to the edit page.



# Hyperlinks

When creating hyperlinks to different pages in your ASP.Net application it is usually best to use the @Html.ActionLink() helper function in your views. This allows you to specify the text of the hyperlink the Action method, the controller and any parameters you might use during a GET request.

|  |
| --- |
| @Html.ActionLink(  "Edit Employee", // Set the text  "Edit", // Set action method  new {  controller = "Home", // Set controller  employeeID = item.EmployeeID, // Set parameters if they exist.  branch = item.Branch  }) |

In this case, the hyperlink would be able to pass values to an action method that uses the same parameters provided:

|  |
| --- |
| public ActionResult Edit(int employeeID, string branch) {  ...  return View();  } |

## Redirecting to a Different ActionResult Method

For different types of page requests you may want to redirect to a different view. This snippet shows how to redirect to the Index() action method of the Home controller.

|  |
| --- |
| [HttpPost]  public ActionResult Edit(EmployeeStoreVM esVM) {  EmployeeStoreRepo esRepo = new EmployeeStoreRepo();  esRepo.Update(esVM);  // go to index action method of the home controller.  return RedirectToAction("Index", "Home");  } |

This snippet shows how to redirect to an Index() action method which has parameters.

|  |
| --- |
| [HttpPost]  public ActionResult Edit(EmployeeStoreVM esVM) {  EmployeeStoreRepo esRepo = new EmployeeStoreRepo();  esRepo.Update(esVM);  // go to index action method of the home controller.  return RedirectToAction("Index", "Home", { id = 99 });  } |

Since the redirect is to the Index action method of the Home controller

|  |
| --- |
| public ActionResult Index(int id) {  ...  return View();  } |

# Validation

To help ensure our application is safe and user-friendly it is important to prevent invalid inputs. ASP.NET offers attributes to assist with automating the process of client and server side validation. ASP.NET MVC provides the namespace **System.ComponentModel.DataAnnotations** which can be included in a model to enable **required**, **range**, and **regular expression** validation attributes. (For more information on C# attributes please see the chapter on attributes in the booklet for C#). Validation is easy to set up so let’s look at an example.

Example : Client Side Validation

🞑 To begin, create an empty MVC application. Then, create this person view model. Note that the validation for **required**, **range**, and **regular expression** validation are included in attributes above the properties being validated. Note also that **ErrorMessage** parameters may also be included with the attribute parameters.

|  |
| --- |
| public class Person{  [Required(ErrorMessage="First name required.")]  [StringLength(50, ErrorMessage="Name must be maximum of 50 characters")]  public string FirstName { get; set; }  public string LastName { get; set; }    [Range(0, 120, ErrorMessage = "Age must be between 0 and 120")]  public int Age { get; set; }  [RegularExpression(@"^(([^<>()[\]\\.,;:\s@\""]+"  + @"(\.[^<>()[\]\\.,;:\s@\""]+)\*)|(\"".+\""))@"  + @"((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}"  + @"\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+"  + @"[a-zA-Z]{2,}))$", ErrorMessage = "Not a valid email address")]  public string Email { get; set; }  } |

**Home Controller**

Next, create an empty home controller.

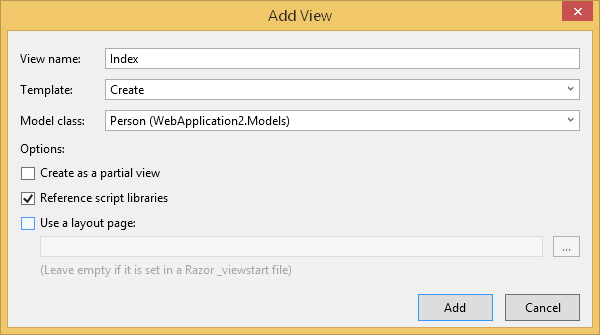
### Enabling Server Side Validation

The **ModelState.IsValid** property can be used to ensure the content submitted **is valid at the server.** You will need to reference the namespace where your *Person* model exists.

|  |
| --- |
| [HttpGet]  public ActionResult Index()  {  return View();  }  [HttpPost]  public ActionResult Index(Person person)  {  ViewBag.ErrorMessage = "";  if (ModelState.IsValid){  // model is valid...  // do something like save object and redirect  // to another page  ViewBag.ErrorMessage = "Success!";  }  else  ViewBag.ErrorMessage = "This entry is invalid.";  // return view with errors  return View(person);  } |

### Enabling Client Side Validation

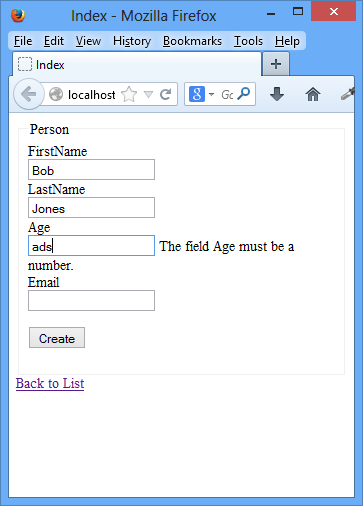
Once you have built your controller, you can then create a view for the Index() action method. When you create the view be sure to use the “Create” scaffold template. As well, select “Reference script libraries” which will import the javascript needed to display the validation messages and to also block users from submitting their form’s content to the server.



Next, add the following tag just before the form tag in the HTML to display whether the Person data submitted to the server is valid or invalid:

|  |
| --- |
| @Html.Raw(ViewBag.ErrorMessage)<br /> |

If you run the project, you will find that error messages associated with the validation attributes are displayed when the fields of the form are invalid. As well, you will notice that the ViewBag message from the server is never displayed as long as the form data is invalid. This is because javascript is included with the form which prevents an invalid form from being submitted.



### Disabling Client Side Validation

Normally you will always want client-side validation. To disable it though for observation purposes, you could remove the following tags from the view:

|  |
| --- |
| <script src="~/Scripts/jquery-1.8.2.min.js"></script>  <script src="~/Scripts/jquery.validate.min.js"></script>  <script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script> |

If you run your project again you will notice that the form can be submitted to the server even when it is invalid. In our example, we know the data was submitted to the server because the view bag message that was set in the controller is displayed.

Exercise

🖍 Revise your person class to implement required field validation for all attributes in the class. Also, add a string length validator to the last name and add a regular expression check to the first and last names to ensure that both are alphabetical characters only.

Show your revised class:

|  |
| --- |
|  |

# Customizing Labels

While we are on the topic of labels you can actually customize the text that appears in the property labels when you use the wizard to generate views based on them. For example, if you use the wizard to generate a view you want to see:

**Employee Last Name: Jones**

And not

**EmployeeLastName: Jones**

You can do this by placing a Display attribute above the property to assign the desired label text.

|  |
| --- |
| [Display(Name = "Employee Last Name")]  public string EmployeeLastName { get; set; } |

# Partial Views

Partial views are re-usable child templates which store blocks of HTML and Razor code. Partial views are excellent for eliminating redundant code.

### Creating a Partial View for a Specific Controller

To create the partial view for a specific controller, right click an action method inside the controller and choose **Add View**.

### Creating a Partial View for all Controllers

If you are creating a partial view that is shared by multiple controllers then right click the **View/Shared** directory and choosing **Add | View**.

## Implicitly (Weakly) Typed Partial Views

Most likely you will want to create implicitly typed partial views to store re-usuable HTML tags for things like logo and heading displays.

## Strongly Typed Partial Views

You may create strongly or implicitly typed views. **Please note though, strongly typed partial views can only work if it is set up to work with the same model that is given to it from the parent.**

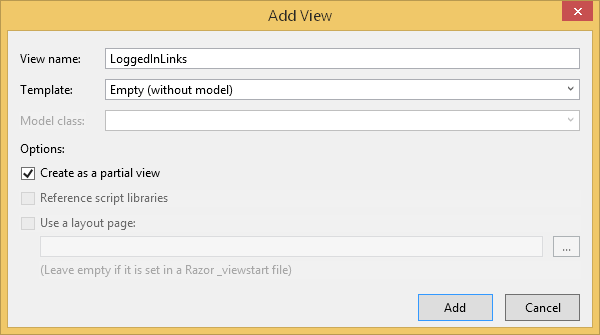
## All Cases

Name your partial view something different than the view it will reside in and check **‘Create as Partial View’**. Then, your Partial View can be referenced in your View with the line of code:

|  |
| --- |
| @Html.Partial("ScheduledPersonPartial") |

Example : Creating an Implicitly Typed Partial View

**🞑** This example shows how to create and use a partial view for re-using HTML tags. To start, create an empty MVC application. Then, create a **Views/Shared/Partials** directory. Right click the **Partials** directory, and choose **Add | View**. In the Add View dialog that appears, enter the name **LoggedInLinks** for the View. Ensure **Create as partial view** is selected and then click Add.



In the Loggedinlinks.cshtml page that appears, add the following tags:

|  |
| --- |
| <a href="#">My Profile</a> | <a href="#">My Past Orders</a> |

Next, create an empty Home controller. Then, add an empty view for the Home controller's Index action method. Replace the contents of the view with the following tags and notice the reference to the partial view:

|  |
| --- |
| @{ ViewBag.Title = "Index";  }  <h2>Welcome Back</h2>  @Html.Partial("~/Views/Shared/Partials/LoggedInLinks.cshtml") |

When you run your project, you will notice that the parent view contents are displayed together with the partial view contents.

Example : Creating a Strongly Typed Partial View

**🞑** In this example, we are going to create an MVC project that re-uses a strongly typed partial view. Remember, a partial view can only implement the same model type as the parent view so this creates some limitation on how the partial view is implemented. If you must share tags with multiple views while using a specific data model that is not from the parent view then you may consider using an HTML helper instead.

To start, create an empty MVC application. Add a ViewModels folder and add the following view model:

**ViewModels\PersonSchedule.cs**

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  namespace PartialViewDemo.ViewModels {  public class PersonSchedule {  public string First { get; set; }  public string Last { get; set; }  public bool Confirmed { get; set; }  public PersonSchedule(string first, string last, bool confirmed) {  First = first;  Last = last;  Confirmed = confirmed;  }  }  } |

Next, add a PersonRepository class in a Models folder:

**PersonRepository.cs**

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using PartialViewDemo.ViewModels;  namespace PartialViewDemo.Models {  public class PersonRepository {  List<PersonSchedule> personSchedules;  public PersonRepository() {  GeneratePeople();  }  void GeneratePeople() {  const bool SCHEDULE\_CONFIRMED = true;  personSchedules = new List<PersonSchedule>();  personSchedules.Add(new PersonSchedule("Bill", "Oakely",  SCHEDULE\_CONFIRMED));  personSchedules.Add(new PersonSchedule("Cheryl", "Ko",  !SCHEDULE\_CONFIRMED));  personSchedules.Add(new PersonSchedule("Anne", "Davies",  SCHEDULE\_CONFIRMED));  }  public IEnumerable<PersonSchedule> AllPeople() {  return personSchedules;  }  public IEnumerable<PersonSchedule> ConfirmedPersonsOnSchedule() {  return personSchedules.Where(x => x.Confirmed);  }  }  } |

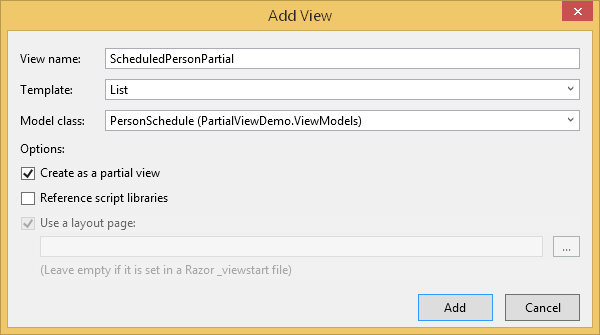
Next, create an empty Home controller and replace the Index action method with the following:

|  |
| --- |
| public ActionResult Index() {  PersonRepository personRepo = new PersonRepository();  return View(personRepo.AllPeople());  }  public ActionResult Confirmed() {  PersonRepository personRepo = new PersonRepository();  return View(personRepo.ConfirmedPersonsOnSchedule());  } |

You will need to reference the namespace of the repository model in the controller file:

|  |
| --- |
| using PartialViewDemo.Models; |

Next, we are going to add a partial view that can be used by any action method within the Home controller as long as the parent view receives an IEnumerable list of *PersonSchedule* objects. To do this, right click any of the action methods in the Home controller and choose Add | View with the settings shown in the screenshot:



Next, add weakly typed views for both the Index and Confirmed action methods. In the **Index** view add the following tags to the body:

|  |
| --- |
| @Html.ActionLink("Confirmed Schedule", "Confirmed")<h1>All Persons With Confirmed and Unconfirmed Schedules</h1>  @Html.Partial("ScheduledPersonPartial") |

In the **Confirmed** view, add the following tags to the body:

|  |
| --- |
| @Html.ActionLink("Scheduled and Unscheduled Persons", "Index")<h1>Persons with Confirmed Schedules</h1>  @Html.Partial("ScheduledPersonPartial") |

When you run the project, you will notice each view appears with the same partial view.

Exercise

🖍 You may find that you need to create a strongly typed partial view which has a different model than the parent view. To do this, you can store your data in a ViewBag. Revise the example above by passing the data to the view with the ViewBag property ScheduledPeople.

For hints on how to store a list in the ViewBag and iterate through it, please see example 6 in Part 1 Inclass 1. Show your revised PartialView here:

|  |
| --- |
|  |