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

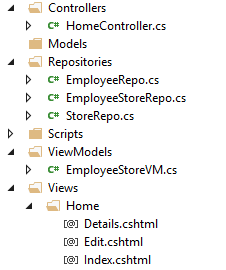
## View Models

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;  } |

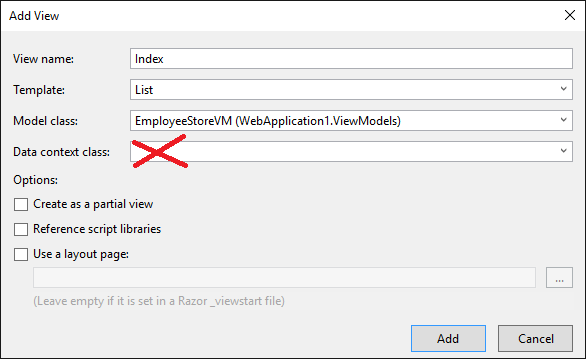
#### Adding the Action Method

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

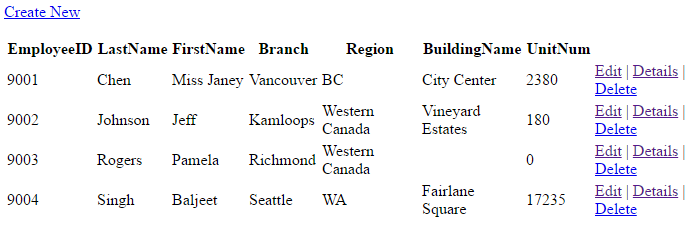
|  |
| --- |
| public ActionResult Index()  {  EmployeeStoreRepo esRepo = new EmployeeStoreRepo();  IEnumerable<EmployeeStoreVM> es = esRepo.GetAll();  return View(es);  } |

#### 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;  } |

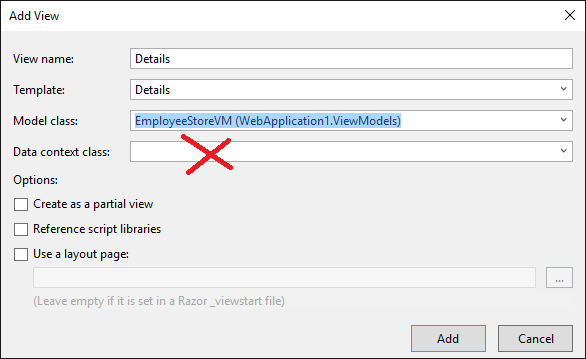
#### Adding the Action Method

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.

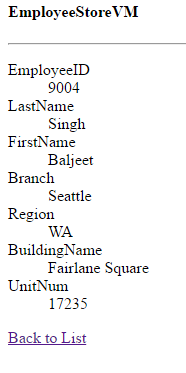


#### Setting the Hyperlink

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;  } |

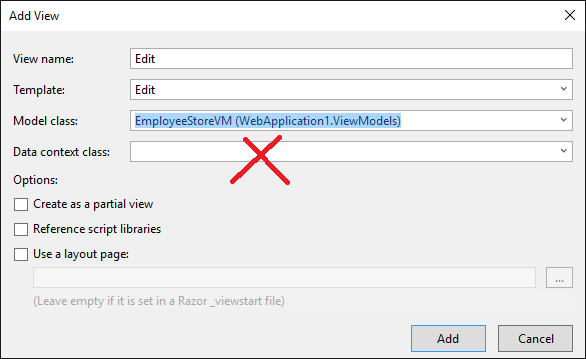
#### Adding the Action Method

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.

