Contents

[1. Basics of MVC Architecture 3](#_Toc451641172)

[2. Advantages over Web Form Application 3](#_Toc451641173)

[3. HTML Controls, button type, Form Tag and methods 3](#_Toc451641174)

[4. Model, View Model, dealing with collections in Model 4](#_Toc451641175)

[5. Entity Framework for DAL, Code First Approach, LINQ Query 6](#_Toc451641176)

[6. Data Flow, ViewBag, ViewData, TempData, Session Object, Hidden Field, Cookies 8](#_Toc451641177)

[7. Client Side Validation 9](#_Toc451641178)

[8. Action Result Method Overloading 9](#_Toc451641179)

[9. Concept of Model Binder and Model state in MVC 9](#_Toc451641180)

[10. Server Side Validation (Model Binder + Data Annotation) 15](#_Toc451641181)

[11. Custom Server Side Validation 20](#_Toc451641182)

[12. Maintaining State on HTML form using Model Binder 20](#_Toc451641183)

[13. Client Side Validation using Java Script 21](#_Toc451641184)

[14. Authentication in MVC 21](#_Toc451641185)

[4. Filters 25](#_Toc451641186)

[5. Client Side Validation using JQuery Unobtrusive 25](#_Toc451641187)

[6. Partial View 26](#_Toc451641188)

[7. Role Based Authorization 28](#_Toc451641189)

[8. MVC Action Filters 30](#_Toc451641190)

[Types of Filters 31](#_Toc451641191)

[Authentication Filters 31](#_Toc451641192)

[Authorization Filters 31](#_Toc451641193)

[Action Filters 32](#_Toc451641194)

[Result Filters 32](#_Toc451641195)

[Exception Filters 33](#_Toc451641196)

[Order of Filter Execution 33](#_Toc451641197)

[Configuring Filters 33](#_Toc451641198)

[**1.** Global level 33](#_Toc451641199)

[**2.** Controller level 33](#_Toc451641200)

[**3.** Action level 33](#_Toc451641201)

[9. How to stop direct execution of Partial View Action 34](#_Toc451641202)

[10. Sections in View Page [Concept of Layout Page] 34](#_Toc451641203)

[11. Avoiding Repetitive Code using Filters 37](#_Toc451641204)

[12. Async Controller 39](#_Toc451641205)

[13. Error or exception Handling log exception, Exception Filters 41](#_Toc451641206)

[14. Routing Concept, User friendly URL 44](#_Toc451641207)

[15. Difference between Route Parameter and Query String 45](#_Toc451641208)

[16. Alternate ways of Mapping URL pattern 46](#_Toc451641209)

[17. Single Page Application 47](#_Toc451641210)

[18. Areas in MVC 47](#_Toc451641211)

[19. Temp Data in MVC 50](#_Toc451641212)

[20. Keep and Peek Methods 53](#_Toc451641213)

[21. Stop CSRF Attack 54](#_Toc451641214)

[22. MVC Bundling and Minification 56](#_Toc451641215)

[23. JSON 58](#_Toc451641216)

[24. Angular JS 59](#_Toc451641217)

# Basics of MVC Architecture

# Advantages over Web Form Application

# HTML Controls, button type, Form Tag and methods

First Name: <input type="text" id="TxtFName" name="FirstName" value="" /><br />

<input type="submit" name="BtnSave" value="Save Employee" />

<ahref="/Employee/AddNew">Add New</a>

<form action="/Employee/SaveEmployee" method="post">

<input type="file" name="fileUpload" value="" />

<input type="submit" name="name" value="Upload" />

Web world won’t follow Event driven programming model. It follows request‐response model. End user make the request and server sends response.” Form tag is one of the way to make request in HTML. As soon as the submit button inside form tag gets clicked, a request will be sent to the URL specified in action attribute.

What is method attribute in Form tag?

It decides the type of request. Request may be one of the following four types ‐ get, post, put and delete.

As per the web standards we should use–

Get ‐ > When we want to get something

Post ‐> When we want to create something

Put ‐> When we want to update something

Delete ‐> when we want to delete something.

When request is made with the help of Form tag, values of all the input controls are sent with the request for processing.

When request is of type Get, Put or Delete, values will be sent as Query string parameters.

When it’s a post request values will be sent as posted data.

What is the purpose of name attribute in input controls?

As discussed before values of all input controls will be sent along with request when submit button is clicked. It makes server receive more than one value at a time. To distinguish each value separately while sending every value is attached with one key and that key will be simply “name” attribute.

Does name and id attribute serves same purpose?

No, as per last question “name” attribute will be used internally by HTML when the request is being sent whereas “id” attribute will be used by developers inside JavaScript for some dynamic stuffs.

**What is the difference between “input type=submit” and “input type=button”?**

Submit button will be specially used when we want to make request to the server whereas simple button will be used to perform some custom client side actions. Simple button won’t do anything by its own.

## **Different types of HTML Helpers**

### **Built-In Html Helpers**

### **Standard Html Helpers**

HTML Element

Example

TextBox

@Html.TextBox("Textbox1", "val")   
Output: <input id="Textbox1" name="Textbox1" type="text" value="val" />

TextArea

@Html.TextArea("Textarea1", "val", 5, 15, null)   
Output: <textarea cols="15" id="Textarea1" name="Textarea1" rows="5">val</textarea>

Password

@Html.Password("Password1", "val")   
Output: <input id="Password1" name="Password1" type="password" value="val" />

Hidden Field

@Html.Hidden("Hidden1", "val")   
Output: <input id="Hidden1" name="Hidden1" type="hidden" value="val" />

CheckBox

@Html.CheckBox("Checkbox1", false)   
Output: <input id="Checkbox1" name="Checkbox1" type="checkbox" value="true" /> <input name="myCheckbox" type="hidden" value="false" />

RadioButton

@Html.RadioButton("Radiobutton1", "val", true)   
Output: <input checked="checked" id="Radiobutton1" name="Radiobutton1" type="radio" value="val" />

Drop-down list

@Html.DropDownList (“DropDownList1”, new SelectList(new [] {"Male", "Female"}))   
Output: <select id="DropDownList1" name="DropDownList1"> <option>M</option> <option>F</option> </select>

Multiple-select

Html.ListBox(“ListBox1”, new MultiSelectList(new [] {"Cricket", "Chess"}))   
Output: <select id="ListBox1" multiple="multiple" name="ListBox1"> <option>Cricket</option> <option>Chess</option> </select>

### **Strongly Typed HTML Helpers**

These helpers are used to render the most common types of HTML elements in strongly typed view like as HTML text boxes, checkboxes etc. The HTML elements are created based on model properties.

The strongly typed HTML helpers work on lambda expression. The model object is passed as a value to lambda expression, and you can select the field or property from model object to be used to set the id, name and value attributes of the HTML helper. A list of most common strongly-typed html helpers is given below:

HTML Element

Example

TextBox

@Html.TextBoxFor(m=>m.Name)   
Output: <input id="Name" name="Name" type="text" value="Name-val" />

TextArea

@Html.TextArea(m=>m.Address , 5, 15, new{}))   
Output: <textarea cols="15" id="Address" name=" Address " rows="5">Addressvalue</textarea>

Password

@Html.PasswordFor(m=>m.Password)   
Output: <input id="Password" name="Password" type="password"/>

Hidden Field

@Html.HiddenFor(m=>m.UserId)   
Output: <input id=" UserId" name=" UserId" type="hidden" value="UserId-val" />

CheckBox

@Html.CheckBoxFor(m=>m.IsApproved)   
Output: <input id="Checkbox1" name="Checkbox1" type="checkbox" value="true" /> <input name="myCheckbox" type="hidden" value="false" />

RadioButton

@Html.RadioButtonFor(m=>m.IsApproved, "val")   
Output: <input checked="checked" id="Radiobutton1" name="Radiobutton1" type="radio" value="val" />

Drop-down list

@Html.DropDownListFor(m => m.Gender, new SelectList(new [] {"Male", "Female"}))   
Output: <select id="Gender" name="Gender"> <option>Male</option> <option>Female</option> </select>

Multiple-select

Html.ListBoxFor(m => m.Hobbies, new MultiSelectList(new [] {"Cricket", "Chess"}))   
Output: <select id="Hobbies" multiple="multiple" name="Hobbies"> <option>Cricket</option> <option>Chess</option> </select>

### **Templated HTML Helpers**

These helpers figure out what HTML elements are required to render based on properties of your model class. This is a very flexible approach for displaying data to the user, although it requires some initial care and attention to set up. To setup proper HTML element with Templated HTML Helper, make use of DataType attribute of DataAnnitation class.

For example, when you use DataType as Password, A templated helper automatically render Password type HTML input element.

Templated Helper

Example

Display

Renders a read-only view of the specified model property and selects an appropriate HTML element based on property’s data type and metadata.   
Html.Display("Name")

DisplayFor

Strongly typed version of the previous helper   
Html.DisplayFor(m => m. Name)

Editor

Renders an editor for the specified model property and selects an appropriate HTML element based on property’s data type and metadata.   
Html.Editor("Name")

EditorFor

Strongly typed version of the previous helper   
Html.EditorFor(m => m. Name)

Get posted data in Server side/Controllers

Inside Employee Controller create an action method called SaveEmployee as follows.

public string SaveEmployee(Employee e)

{

return e.FirstName + "|"+ e.LastName+"|"+e.Salary;

}

# Model, View Model, dealing with collections in Model

Reason for all three problems of ViewData and ViewBag is the data type. Data type of values inside ViewData, which is “Object”.

Somehow if we were able to set the type of data which need to be passed between Controller and View problem will get solved and that’s where strongly typed Views comes to picture.

Make View a strongly typed view:

@model WebApplication1.Models.Employee

Employee Name : @Model.FirstName @Model.LastName

Pass Model data from Controller Action method

Employee emp = new Employee();

emp.FirstName = "Sukesh";

emp.LastName="Marla";

emp.Salary = 20000;

return View("MyView",emp);

View With collection

EmployeeViewModel class

public class EmployeeViewModel

{

public string EmployeeName { get; set; }

public string Salary { get; set; }

public string SalaryColor { get; set; }

}

Create Collection View Model

public List<EmployeeViewModel> Employees { get; set; }

public string UserName { get; set; }

@using WebApplication1.ViewModels

@model EmployeeListViewModel

|  |
| --- |
| <table>  <tr>  <th>Employee Name</th>  <th>Salary</th>  </tr>  @foreach (EmployeeViewModel item in Model.Employees)  {  <tr>  <td>@item.EmployeeName</td>  <td style="background‐color:@item.SalaryColor">@item.Salary</td>  </tr>  }  </table> |

Business Layer Code:

public class EmployeeBusinessLayer

{

public List<Employee> GetEmployees()

{

List<Employee> employees = new List<Employee>();

Employee emp = new Employee();

emp.FirstName = "johnson";

emp.LastName = " fernandes";

emp.Salary = 14000;

employees.Add(emp);

}

}

Pass Data from Controller:

|  |
| --- |
| EmployeeListViewModel employeeListViewModel = new EmployeeListViewModel();  EmployeeBusinessLayer empBal = new EmployeeBusinessLayer();  List<employee> employees = empBal.GetEmployees();  List<EmployeeViewModel><employeeviewmodel> empViewModels = new List<EmployeeViewModel> <employeeviewmodel>();  foreach (Employee emp in employees)  {  EmployeeViewModel empViewModel = new EmployeeViewModel(); empViewModel.EmployeeName = emp.FirstName + " " + emp.LastName; empViewModel.Salary = emp.Salary.ToString("C");  if (emp.Salary > 15000)  {  empViewModel.SalaryColor = "yellow";  }  else  {  empViewModel.SalaryColor = "green";  }  empViewModels.Add(empViewModel);  }  employeeListViewModel.Employees = empViewModels; employeeListViewModel.UserName = "Admin"; return View("MyView", employeeListViewModel); |

# Entity Framework for DAL, Code First Approach, LINQ Query

What is Entity Framework in simple words?

It’s an ORM tool. ORM stands for Object Relational Mapping.

In RDBMS world, we speak in terms of Tables and Columns whereas in .net world ﴾which is an object oriented world﴿, we speak in terms of Classes, objects and properties.

When we think about any data driven application we end up with following two things.

* Write code for communicating with database ﴾called Data Access Layer or Database logic﴿
* Write code for mapping Database data to object oriented data or vice versa.

ORM is a tool which will automate these two things. Entity framework is Microsoft ORM tool.

In Entity framework we can follow one of these three approaches

Database First approach – Create database with tables, columns, relations etc. and Entity framework will generates corresponding Model classes ﴾Business entities﴿ and Data Access Layer code.

Model First approach – In this approach Model classes and relationship between them will be

defined manually using Model designer in Visual studio and Entity Framework will generate Data

Access Layer and Database with tables, columns, relations automatically.

Code First approach – In this approach manually POCO classes will be created. Relationship between those classes will be defined by means of code. When application executes for the first time Entity framework will generate Data Access Layer and Database with tables, column and relations automatically in the database server.

What is mean by POCO classes?

POCO stands for “Plain Old CLR objects”. POCO classes means simple .Net classes we create. In our previous example Employee class was simply a POCO class.

Open Web.config file and inside Configuration section add following section

<connectionStrings>

<add connectionString="Data Source=(local);Initial Catalog=SalesERPDB;Integrated Security=True" name="SalesERPDAL"

providerName="System.Data.SqlClient"/> </connectionStrings>

Create a new folder called “DataAccessLayer” in the root folder and inside it create a new class called “SalesERPDAL”

public class SalesERPDAL: DbContext

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<employee>().ToTable("TblEmployee"); base.OnModelCreating(modelBuilder);

}

public DbSet<employee> Employees{get;set;}

Model Class:

public class Employee

{

[Key]

public int EmployeeId { get; set; } public string FirstName { get; set; } public string LastName { get; set; } public int Salary { get; set; }

}

Business Layer Code:

|  |
| --- |
| public List<employee> GetEmployees()  {  SalesERPDAL salesDal = new SalesERPDAL(); return salesDal.Employees.ToList();  } |

Insert Data into SQL.

**What is DbSet?**

DbSet simply represent the collection of all the entities that can be queried from the database. When we write a Linq query again DbSet object it internally converted to query and fired against database.

In our case “Employees” is a DbSet which will hold all the “Employee” entities which can be queried from database. Every time we try to access “Employees” it gets all records in the “TblEmployee” table and convert it to “Employee” object and return the collection.

**How connection string and data access layer is connected?**

Mapping will be done based on name. In our example ConnectionString Name and Data Access Layer class name is same that is “SalesERPDAL”, hence automatically mapped.

**Can we change the ConnectionString name?**

Yes, in that case we have to define a constructor in Data Access Layer class as follows.

public SalesERPDAL():base("NewName")

{

}

# Data Flow, ViewBag, ViewData, TempData, Session Object, Hidden Field, Cookies

**ViewData** is a dictionary, which will contains data to be passed between controller and views. (C🡪 V)

Syntax:

ViewData["Employee"] = emp;

@{

WebApplication1.Models.Employee emp=(WebApplication1.Models.Employee)

ViewData["Employee"];

}

ViewData holds objects internally. Every time a new value is added into it, it get boxed to object type.

So unboxing is required every time we try to extract value out of it.

**View Bag**

ViewBag is just a syntactic sugar for ViewData. ViewBag uses the dynamic feature of C# 4.0 and makes ViewData dynamic.

ViewBag internally uses ViewData.

ViewBag.Employee = emp;

@{

WebApplication1.Models.Employee emp = (WebApplication1.Models.Employee)

ViewBag.Employee;

Can we pass ViewData and get it as ViewBag?

Yes, We can. Vice versa is also possible. As I said before, ViewBag is just a syntactic sugar for ViewData,

**Problems with ViewData and ViewBag**

1. Values inside the ViewData are of type Object. We have to cast the value to correct before using it. It adds additional overhead on performance.
2. If we try to cast values to wrong type or if we use wrong keys while retrieving the values, we will get runtime error. As a good programming practice, error should be tackled in compiled time.
3. In MVC, controller and View are loosely connected to each other. Controller is completely unaware about what’s happening in View and View is unaware about what’s happening in Controller. From Controller we can pass one or more ViewData/ViewBag values. Now when Developer writes a View, he/she have to remember what is coming from the controller. If Controller developer is different from View developer then it becomes even more difficult. Complete unawareness. It leads to many run time issues and inefficiency in development.

# Client Side Validation

# Action Result Method Overloading

ViewResult ‐ Renders a specified view to the response stream

PartialViewResult ‐ Renders a specified partial view to the response stream

EmptyResult ‐ An empty response is returned

RedirectResult ‐ Performs an HTTP redirection to a specified URL

RedirectToRouteResult ‐ Performs an HTTP redirection to a URL that is determined by the routing engine,

based on given route data

JsonResult ‐ Serializes a given ViewData object to JSON format

JavaScriptResult ‐ Returns a piece of JavaScript code that can be executed on the client

ContentResult ‐ Writes content to the response stream without requiring a view

FileContentResult ‐ Returns a file to the client

FileStreamResult ‐ Returns a file to the client, which is provided by a Stream

FilePathResult ‐ Returns a file to the client

# Concept of Model Binder and Model state in MVC

**How Textbox values are updated in Employee object inside action method?**

In Asp.Net MVC there is a concept called as Model Binder.

Model Binder will executes automatically whenever a request is made to an action method containing parameter.

Model binder will iterate though all primitive parameters of a method and then it will compare name of the parameter with each key in the incoming data ﴾Incoming data means either posted data or query string﴿. When match is found, corresponding incoming data will be assigned to the parameter.

After that Model binder will iterate through each and every property of each and every class parameter and compare each property name with each key in incoming data. When match is found, corresponding incoming value will be assigned to the parameter.

What will happen when two parameters are specified, one as “Employee e” and second as “string FirstName”?

FirstName will be updated in both primitive FirstName variable and e.FirstName property.

**Will Model Binder workwith composition relationship?**

Yes it will, but in that case name of the control should be given accordingly.

Example

Let say we have Customer class and Address class as follows

public class Customer

{

public string FName{get;set;} public Address address{get;set;}

}

public class Address

{

public string CityName{get;set;} public string StateName{get;set;}

}

In this case Html should look like this

<input type="text" name="FName">

<input type="text" name="address.CityName">

<input type="text" name="address.StateName">

Cancel button:

<input type="submit" name="BtnSubmit; value="Save Employee" />

<input type="submit" name="BtnSubmit" value="Cancel" />

public ActionResult SaveEmployee(Employee e, string BtnSubmit)

{

switch (BtnSubmit)

{

case "Save Employee":

return Content(e.FirstName + "|" + e.LastName + "|" + e.Salary); case "Cancel":

return RedirectToAction("Index");

}

return new EmptyResult();

}

Why same name is given to both Save and Cancel button?

We know that, as soon as submit button is clicked, a request is sent to the server. Along with the request values of all the input controls will be sent.

Submit button is also an input button. Hence value of the submit button ﴾which is responsible for the request﴿ will be sent too.

When Save button will be clicked, value of Save button that is “Save Employee” will be sent with request and when Cancel button is clicked, value of Cancel button that is “Cancel” will sent with request.

In Action method, Model Binder will do remaining work. It will update the parameter values with values in input data ﴾coming with request﴿

**What are the other ways to implement multiple submit buttons?**

There are many ways.

1. Hidden Form element

Step 1 – Create a hidden form element in View as follows.



<form action="/Employee/CancelSave" id="CancelForm" method="get" style="display:none">

</form>

<form action="/Employee/CancelSave" id="CancelForm" method="get" style="display:none"> </form>

Step 2 – Change Submit button to normal button and post above form with the help of JavaScript.

<input type="button" name="BtnSubmit" value="Cancel" onclick="document.getElementById('CancelForm').submit()" />

2. Change action URL dynamically using JavaScript

<form action="" method="post" id="EmployeeForm" >

...

...

<input type="submit" name="BtnSubmit" value="Save Employee" onclick="document.getElementById('EmployeeForm').action = '/Employee/SaveEmployee'" />

...

<input type="submit" name="BtnSubmit" value="Cancel" onclick="document.getElementById('EmployeeForm').action = '/Employee/CancelSave'" /> </form>

3. Ajax

Instead of submit button use simple input button and onclick of it make pure Ajax request using jQuery or any other library.

Why we have not used input type=reset for implementing Reset functionality?

Input type=reset control won’t clear the values, it just set the value to default value of a control. Example:

<input type="text" name="FName" value="Sukesh">

In above example default value of control is “Sukesh”.

If we use input type=reset for implementing Reset functionality then by default “Sukesh” will be set in the textbox every time “reset” button is clicked.

What if names are not matching with property names of the classes?

This is a very common question during interviews.

Let say we have HTML as follows:

First Name: <input type="text" id="TxtFName" name="FName" value="" /><br />

Last Name: <input type="text" id="TxtLName" name="LName" value="" /><br />

Salary: <input type="text" id="TxtSalary" name="Salary" value="" /><br />

Now our Model class contain property names as FirstName, LastName and Salary. Hence default model binder won’t work here.

In this situation we have following three solutions

Inside action method, retrieve posted values using Request.Form syntax and manually construct the Model object as follows.

public ActionResult SaveEmployee()

{

Employee e = new Employee(); e.FirstName = Request.Form["FName"]; e.LastName = Request.Form["LName"];

e.Salary = int.Parse(Request.Form["Salary"])

...

...

}

Use parameter names and Creates Model object manually as follows.

public ActionResult SaveEmployee(string FName, string LName, int Salary)

{

Employee e = new Employee(); e.FirstName = FName; e.LastName = LName; e.Salary = Salary;

...

...

}

Custom Model Binder and replace default model binder as follows:

|  |
| --- |
| public class MyEmployeeModelBinder : DefaultModelBinder  {  protected override object CreateModel(ControllerContext controllerContext, ModelBindingContext bindingContext, Type modelType)  {  Employee e = new Employee();  e.FirstName = controllerContext.RequestContext.HttpContext.Request.Form["FName"]; e.LastName = controllerContext.RequestContext.HttpContext.Request.Form["LName"]; e.Salary =  int.Parse(controllerContext.RequestContext.HttpContext.Request.Form["Salary"]); return e;  }  } |

Step 2‐ Replace default model binder with this new model binder

public ActionResult SaveEmployee([ModelBinder(typeof(MyEmployeeModelBinder))]Employee e, string BtnSubmit)

{

......

**What does RedirecttToFunction do?**

It generates RedirectToRouteResult Just like ViewResult and ContentResult ﴾discussed in Day 1﴿, RedirectToRouteResult is a child of ActionResult. It represents the redirect response. When browser receives RedirectToRouteResult, it makes new request to new action method.

Note: Here browser is responsible for new request hence URL will get updated to new URL.

**What is EmptyResult?**

One more child of ActionResult. When browser receives EmptyResult as a response it simply displays

blank white screens. It simply represents “No Result”.

In our example this situation won’t happen. Just to make sure that all code paths returns a value EmptyResult statement was written.

Note: When ActionMethod return type is Void, it is equivalent to EmptyResult

Save records in database and update Grid

public Employee SaveEmployee(Employee e)

{

SalesERPDAL salesDal = new SalesERPDAL(); salesDal.Employees.Add(e); salesDal.SaveChanges();

return e;

}

Step 2 – Change SaveEmployee Action method

In EmployeeController change the SaveEmployee action method code as follows.

public ActionResult SaveEmployee(Employee e, string BtnSubmit)

{

switch (BtnSubmit)

{

case "Save Employee":

EmployeeBusinessLayer empBal = new EmployeeBusinessLayer(); empBal.SaveEmployee(e);

return RedirectToAction("Index"); case "Cancel":

return RedirectToAction("Index");

}

return new EmptyResult();

}

# Server Side Validation (Model Binder + Data Annotation)

Model binder updates the Employee object with the posted data. But this is not the only functionality performed by Model Binder. Model Binder also updates ModelState. ModelState encapsulates the state of the Model.

It have a property called IsValid which determines whether the Model ﴾that is Employee object﴿ gets successfully updated or not. Model won’t update if any of the server side validation fails.

It holds validation errors.

Example: ModelState["FirstName "]. Errors will contain all errors related to First Name It holds the incoming value﴾Posted data or queryString data﴿

It holds the incoming value﴾Posted data or queryString data﴿

In Asp.net MVC we use DataAnnotations to perform server side validations.

Before we get into Data Annotation lets understand few more things about Model Binder

How Model Binder work with primitive datatypes

When Action method contain primitive type parameter, Model Binder will compare name of the parameter with each key in the incoming data ﴾Incoming data means either posted data or query string﴿.

When match is found, corresponding incoming data will be assigned to the parameter. When match is not found, parameter will be assigned with default value. ﴾Default value – For integer it is 0 ﴾zero﴿, for string it is null etc.﴿

In case assignment is not possible because of datatype mismatch exception will be thrown.

**How Model Binder work with classes**

When parameter is a Class parameter, Model Binder will iterate through all properties of all the class and compare each property name with each key in incoming data.

When match is found,

If corresponding incoming value is empty, then Null value will be assigned to property. If null assignment is not possible, default value will be set and ModelState.IsValid will be set to false.

If null assignment is possible but will be considered as invalid value because of the validation attached to the property then null be assigned as value and ModelState.IsValid will be set to false.

If corresponding incoming value is non empty,

In case assignment is not possible because of datatype mismatch or

Server side validation failure null value will be assigned and

ModelState.IsValid will be set to false.

If null assignment is not possible, default value will be set

When match is not found, parameter will be assigned with default value. ﴾Default value – For integer it is 0 ﴾zero﴿, for string it is null etc.﴿In this case ModelState.IsValid will remain unaffected.

Step 1 – Decorate Properties with DataAnnotations

Open Employee class from Model folder and decorate FirstName and LastName property with DataAnnotation attribute as follows.

public class Employee

{

[Required(ErrorMessage="Enter First Name")] public string FirstName { get; set; }

StringLength(5,ErrorMessage="Last Name length should not be greater than 5")]

public string LastName { get; set; }

}

Step 2 – Change SaveEmployee Action method

Open EmplyeeController and Change SaveEmployee Action method as follows.

public ActionResult SaveEmployee(Employee e, string BtnSubmit)

{

switch (BtnSubmit)

{

case "Save Employee":

if (ModelState.IsValid)

{

EmployeeBusinessLayer empBal = new EmployeeBusinessLayer(); empBal.SaveEmployee(e);

return RedirectToAction("Index");

}

else

{

return View("CreateEmployee ");

}

case "Cancel":

return RedirectToAction("Index");

}

return new EmptyResult();

}

Note: As you can see, When ModelState.IsValid is false response of SaveEmployee button click is ViewResult pointing to “CreateEmployee” view.

Step 3 – Display Error in the View

Change HTML in the “Views/Index/CreateEmployee.cshtml” to following.

This time we will format our UI a little with the help of “table” tag;

<table> <tr>

<td>

First Name: </td>

<td>

<input type="text" id="TxtFName" name="FirstName" value="" /> </td>

</tr> <tr>

<td colspan="2" align="right">

@Html.ValidationMessage("FirstName") </td>

</tr> <tr>

<td>

Last Name: </td>

<td>

<input type="text" id="TxtLName" name="LastName" value="" /> </td>

</tr> <tr>

<td colspan="2" align="right"> @Html.ValidationMessage("LastName")

</td> </tr>

<tr> <td>

Salary: </td> <td>

<input type="text" id="TxtSalary" name="Salary" value="" /> </td>

</tr> <tr>

<td colspan="2" align="right"> @Html.ValidationMessage("Salary")

</td> </tr>

<tr>

<td colspan="2">

<input type="submit" name="BtnSubmit" value="Save Employee" /> <input type="submit" name="BtnSubmit" value="Cancel" />

<input type="button" name="BtnReset" value="Reset" onclick="ResetForm();" /> </td>

</tr> </table>

**What does @Html.ValidationMessage do?**

@ means it’s a razor code

Html is an instance of HtmlHelper class available inside view.

ValidationMessage is a function of HtmlHelper class which displays the error message

**How ValidationMessage function works?**

ValidationMessage is a function. It executes at runtime. As we discussed earlier, ModelBinder updates the ModelState. ValidationMessage displays the error message from ModelState based on Key.

Example: ValidationMessage﴾“FirstName”﴿ displays the error message related to First Name.

**Do we have more attributes like required and StringLength?**

Yes, here are some

DataType – Make sure that data is of particular type like email, credit card number, URL etc. EnumDataTypeAttribute–Make sure that value exist in an enumeration.

Range Attribute – Make sure that value is in a particular range.

Regular expression‐ Validates the value against a particular regular expression.

Required – Make sure that value is provided.

StringthLength – Validates the string for maximum and minimum number of characters.

**Can we explicitly ask Model Binder to execute?**

Yes simply remove parameters from action method. It stops default model binder from executing by default.

In this case we can use UpdateModel function as follows.

Employee e = new Employee();

UpdateModel<employee>(e);

**What is the difference between UpdateModel and TryUpdateModel method?**

TryUpdateModel will be same as UpdateModel except one added advantage.

UpdateModel will throw an exception if Model adaption fails because of any reason. In case of UpdateModel function ModelState.IsValid will not be of any use.

TryUpdateModel is exactly same as keeping Employee object as function parameter. If updating fails ModelState.IsValid will be false;

**What about client side validation?**

It should be done manually unless and until we are using HTML Helper classes.

We are going to talk about both manual client side validation and automatic client side validation with the help of HTML helper classes in day 4.

**Can we attach more than one DataAnnotation attribute to same property?**

Yes we can. In that case both validations will fire.

# Custom Server Side Validation

Step 1 – Create Custom Validation

Create a new class Called FirstNameValidation as follow

|  |
| --- |
| public class FirstNameValidation:ValidationAttribute  {  protected override ValidationResult IsValid(object value, ValidationContext validationContext)  {  if (value == null) *// Checking for Empty Value*  {  return new ValidationResult("Please Provide First Name");  }  else  {  if (value.ToString().Contains("@"))  {  return new ValidationResult("First Name should contain @");  }  }  return ValidationResult.Success;  }  } |

Step 2‐ Attach it to First Name

Open Employee class and remove the default “Required” attribute from FirstName property and attach FirstNameValidation as follows.

[FirstNameValidation]

public string FirstName { get; set; }

# Maintaining State on HTML form using Model Binder

if (ModelState.IsValid)

{

EmployeeBusinessLayer empBal = new EmployeeBusinessLayer();

empBal.SaveEmployee(e);

return RedirectToAction("Index");

}

else

{

CreateEmployeeViewModel vm = new CreateEmployeeViewModel();

vm.FirstName = e.FirstName;

vm.LastName = e.LastName;

if (e.Salary.HasValue)

{

vm.Salary = e.Salary.ToString();

}

Else

{

vm.Salary = ModelState["Salary"].Value.AttemptedValue;

}

return View("CreateEmployee", vm); *// Day 4 Change ‐ Passing e here*

}

case "Cancel":

return RedirectToAction("Index");

}

return new EmptyResult();

}

Repopulate values in View

<input type="text" id="TxtFName" name="FirstName" value="@Model.FirstName" />

**Do we have any automated way to achieve same functionality?**

Yes, we can use HTML helper classes for that. We will talk about this in one of the upcoming lab.

# Client Side Validation using Java Script

Step 1 – Create JavaScript validation file

Create a JavaScript File called “Validations.js” and keep it inside Scripts folder

Create Validation functions

e.g.

function IsFirstNameEmpty() {

if (document.getElementById('TxtFName').value == "") {

return 'First Name should not be empty';

}

else { return ""; }

}

Include Validation file in View

<script src="~/Scripts/Validations.js"></script>

<input type="submit" name="BtnSubmit" value="Save Employee" onclick="return IsValid();" />

**Does server side validation is required anymore?**

Yes, In case someone disables JavaScript, Server side validation keep everything in place.

# Authentication in MVC

Before we start with lab first let’s understand how Forms Authentication works in ASP.NET

1. End user make a request to Forms authentication enabled application with the help of browser.

2. Browser will send all the associated cookies stored in the client machine with request.

1. When request is received as server end, server examines the request and check for the special cookie called “Authentication Cookie”.
2. If valid authentication cookie is found, server confirms the identity of the user or in simple words, consider user as a valid user and make him go further.
3. If valid authentication cookie is not found server considers user as anonymous ﴾or unauthenticated﴿ user. In this case if the requested resource is marked as protected/secured user will be redirected to login page.

Create AuthenticationController and Login action method.

public class AuthenticationController : Controller

{

// GET: Authentication

public ActionResult Login()

{

return View();

}

}

Create Model

namespace WebApplication1.Models

{

public class UserDetails

{

public string UserName { get; set; }

public string Password { get; set; }

}

}

Create Login View

@using (Html.BeginForm("DoLogin", "Authentication", FormMethod.Post))

{

@Html.LabelFor(c=>c.UserName)

@Html.TextBoxFor(x=>x.UserName)

<br />

@Html.LabelFor(c => c.Password)

@Html.PasswordFor(x => x.Password)

<br />

<input type="submit" name="BtnSubmit" value="Login" />

As you can see, this time for generating View instead of Pure HTML we are using HtmlHelper class.

In view we will get a readymade object of HtmlHelper class called “Html”

HtmlHelper class functions simply returns html string.

@Html.TextBoxFor(x=>x.UserName)

Above code will generate following HTML.

<input id="UserName" name="UserName" type="text" value="" />

Web.Config file:

<authentication mode="Forms">

<forms loginurl="~/Authentication/Login"></forms>

</authentication>

Make action method secured.

[Authorize]

public ActionResult Index()

{

EmployeeListViewModel employeeListViewModel = new EmployeeListViewModel();

Open AuthenticationController class and create a new action method called DoLogin.

This DoLogin action method will be invoked when Login button in Login view is clicked ﴾Check Step 3﴿.

Now let’s list down the points need to be done in DoLogin

1. Check for validity of user by invoking business layer function.

2. If user is a valid user create an authentication cookie. It makes futures requests authenticated request.

1. If user is invalid, add a new error to current ModelState. This error will be displayed in View.

[HttpPost]

public ActionResult DoLogin(UserDetails u)

{

EmployeeBusinessLayer bal = new EmployeeBusinessLayer();

if (bal.IsValidUser(u))

{

FormsAuthentication.SetAuthCookie(u.UserName, false);

return RedirectToAction("Index", "Employee");

}

else

{

ModelState.AddModelError("CredentialError", "Invalid Username or Password");

return View("Login");

}

}

ModelState contains all the errors related to current model. In above

code snippet we are adding a new error when user is an invalid user ﴾new error with key

“CredentialError” and message “Invalid Username or Password”﴿.

FormsAuthentication.SetAuthCookie will create a new cookie in client’s machine.

Display message in view

@Html.ValidationMessage("CredentialError", new {style="color:red;" })

@using (Html.BeginForm("DoLogin", "Authentication", FormMethod.Post))

{

**Why DoLogin is attached with HttpPost attribute?**

This attribute makes DoLogin action method open for only Post request. If someone try to make a get request to DoLogin it won’twork out.

**Is it must to write FormsAuthentication.SetAuthCookie?**

Yes.

Let’s understand a small process.

* Client make a first request to server with the help of browser.
* When request is made via browser, all the associated cookies will be sent with the request.
* Server receives the request and prepares the response.

Now as we know request‐response happens via HTTP protocol and HTTP is a stateless protocol. For server every request will be a new request hence when the same client makes second request server won’t recognize it. To solve this issue Server will add a cookie in the prepared response and send back.

* When client’s browsers receives the response with cookies, it creates cookies in client’s machine.
* Now if client make a request once again server will recognize him/her because request contains cookies.
* FormsAuthentication.SetAuthCookie will add a special cookie called “Authentication” cookie to the response.

**Does it means FormsAuthentication won’t work without cookies?.**

No. We have an alternative for it. We can use URI instead of cookies.

Open Web.Config and change “Authentication/Forms” section as follows

<forms cookieless="UseUri" loginurl="~/Authentication/Login"></forms>

Now authentication cookie is passed in URL itself. By default cookieless property is set to “AutoDetect”. It means authentication works via cookies and in case cookies are not supported URL will do the required work.

**What does second parameter of FormsAuthentication.SetAuthCookie do?**

It will decide whether we want to create a persistent cookie or not. Non persistent cookies will get deleted automatically when browser is closed. Persistent cookies wont deleted automatically. We have to remove it manually either via coding or by using browser settings.

**What does Authorize attribute do?**

In Asp.net MVC there is a concept called Filters. Which will be used to filter out requests and response. There are four kind of filters. We will discuss each one of them in our 7 days journey. Authorize attribute falls under Authorization filter. It will make sure that only authenticated requests are allowed for an action method.

**Attach Authorize attribute:**

Controller Level

Global level:

Step 1 ‐ Open FilterConfig.cs file from App\_start folder.

Step 2 ‐ Add one more line RegisterGlobalFilters as follows.

public static void RegisterGlobalFilters(GlobalFilterCollection filters)

{

filters.Add(new HandleErrorAttribute());//Old line

filters.Add(new AuthorizeAttribute());//New Line

}

Step 3 ‐ Attach AllowAnonymous attribute to Authentication controller.

[AllowAnonymous]

public class AuthenticationController : Controller

{

Step 4 – Execute and Test the application in the same way we did before.

**Why AllowAnonymous attribute is required for AuthenticationController?**

We have attached Authorize filter at global level. That means now everything is protected including Login and DoLogin action methods. AllowAnonymous opens action method for non‐authenticated requests.

**How come this RegisterGlobalFilters method inside FilterConfig class invoked?**

It was invoked in Application\_Startevent written inside Global.asax file.

Create Logout Action method;

public ActionResult Logout()

{

FormsAuthentication.SignOut();

return RedirectToAction("Login");

}

# Client Side Validation using JQuery Unobtrusive

Step 1 – Download jQuery unobtrusive Validation files.

Step 2 – Include jQuery Validation files in View.

Above steps adds three JavaScript files in Scripts folder.

jQuery‐Someversion.js

jQuery.valiadte.js

jquery.validate.unobtrusive

Open Login.cshtml and in head section include all three js files ﴾in same sequence﴿.

<script src="~/Scripts/jquery‐1.8.0.js"></script>

<script src="~/Scripts/jquery.validate.js"></script>

<script src="~/Scripts/jquery.validate.unobtrusive.js"></script>

**How come client side validation is implemented?**

As you can see, without much effort client side validation is implemented. In Login view, HTML elements are generated using HTML helper classes. Helper functions while generating HTML markup attach some attributes based on the data annotation attributes used.

These custom HTML attributes will be used by “jQuery Unobtrusive validation files” and thus validation get implemented at client side automatically.

Automatic client side validation is the second advantage of Html helper classes.

**What is unobtrusive JavaScript means?**

“Write your JavaScript in such way that, JavaScript won’t be tightly connected to HTML. JavaScript may

access DOM elements, JavaScript may manipulate DOM elements but won’t directly connected to it.” In the above example, jQuery Unobtrusive JavaScript simply used some input element attributes and implemented client side validation.

**Can we use these JavaScript validation without HTML helper classes?**

Yes, for that we have to manually attach such attributes to elements.

# Partial View

**What are “Partial Views”?**

Logically, Partial View is a reusable view which will never get displayed directly. It will be included inside other views and get displayed as a part of that view. It resembles user controls in Asp.Net web forms, but without Code behind.

Create ViewModel for Partial View

Create Partial View

Display data in the Partial View

Include Footer data in Main ViewModel:

public class EmployeeListViewModel

{

public List<EmployeeViewModel> Employees { get; set; }

public string UserName { get; set; }

public FooterViewModel FooterData { get; set; }*//New Property*

}

We will pass necessary data to Footer from Index View.

Index View is a strongly typed view of type EmployeeListViewModel and hence all the data required for Footer view should be encapsulated in EmployeeListViewModel.

Set Footer Data

Display Footer:

@{

Html.RenderPartial("Footer", Model.FooterData);

}

Just like Html.RenderPartial, Html.Partial will be used to display Partial View in the View.

This is the syntax

@Html.Partial("Footer", Model.FooterData);

**What’s the difference between two**?

Html.RenderPartial will write result of the Partial View directly to the HTTP response stream whereas Html.Partial will return result as MvcHtmlString.

Look at the following code.

@{

string MyString = "My Simple String";

}@

MyString

It will generate following output:

<b>My Simple String</b>

We use MvcHtmlString when we don’t want razor to do the encoding. MvcHtmlString is an indication to razor that “string is already encoded, no more encoding is required”.

For example look at the following code.

@{

string MyString = "My Simple String";

}@

MvcHtmlString.Create(MyString)

It will generate following output :

**My Simple String**

**Why does Html.Partial return MvcHtmlString instead of string?**

We already understood a fact that “razor will always encode strings but it never encodes MvcHtmlString”.

It doesn’t make sense if Partial View contents are considered as pure string gets displayed as it is. We want it to be considered as a HTML content and for that we have to stop razor from encoding thus Partial method is designed to return MvcHtmlString.

**What is recommended Html.RenderPartial or Html.Partial?**

Html.RenderPartial is recommended because it is faster.

**When Html.Partial will be preferred?**

It is recommended when we want to change the result returned by Partial View before displaying.

Open Index.cshtml and open Footer code to below code and test.

@{

MvcHtmlString result = Html.Partial ("Footer", Model.FooterData);

string finalResult = result.ToHtmlString().Replace("2015", "20000");

}@

MvcHtmlString.Create(finalResult)

# Role Based Authorization

Step 1 – Create Enum for identifying UserStatus

public enum UserStatus

{

AuthenticatedAdmin,

AuthentucatedUser,

NonAuthenticatedUser

}

Step 2 – Set Value of Status

if (u.UserName == "Admin" && u.Password == "Admin")

{

return UserStatus.AuthenticatedAdmin;

}

else if (u.UserName == "Sukesh" && u.Password == "Sukesh")

{

return UserStatus.AuthentucatedUser;

}

else

{

return UserStatus.NonAuthenticatedUser;

}

Step 3: Change DoLogin action method

bool IsAdmin = false;

if (status==UserStatus.AuthenticatedAdmin)

{

IsAdmin = true;

}

else if (status == UserStatus.AuthentucatedUser)

{

IsAdmin = false;

}

else

{

ModelState.AddModelError("CredentialError", "Invalid Username or Password");

return View("Login");

}

FormsAuthentication.SetAuthCookie(u.UserName, false);

Session["IsAdmin"] = IsAdmin;

return RedirectToAction("Index", "Employee");

Step 4 – Remove Existing AddNew link

<a href="/Employee/AddNew">Add New</a>

Step 5 – Crete partial view

Step 6 – Put contents in Partial View

Simply put following contents inside newly created Partial View.

<a href="/Employee/AddNew">Add New</a>

Step 7 – Create Action Method

Open EmployeeController and create a new action method called “GetAddNewLink” as follows.

public ActionResult GetAddNewLink()

{

if (Convert.ToBoolean(Session["IsAdmin"]))

{

return Partial View("AddNewLink");

}

else

{

return new EmptyResult();

}

}

Step 8 – Display AddNew link

Open Index.html and simply put following code inside it.

<a href="/Authentication/Logout">Logout</a>

</div>

<hr />

@{

Html.RenderAction("GetAddNewLink");

}<

div>

<table border="1">

<tr>

Html.RenderAction executes the Action Method and writes result directly to response stream.

With above logic one thing is guaranteed. Now a Non‐Admin User won’t be able to navigate to AddNew action via hyperlink.

**Is it enough?**

No, It not enough. What if a Non‐Admin user directly try to navigate to AddNew action via URL.

A Non‐Admin user is able to access the AddNew action.

To solve this problem we will use MVC ActionFilters. Action Filters let us add some pre‐processing and post‐processing logic to our action methods. In this lab we will look after pre‐processing support of ActionFilters and in coming up lab we will look into post‐processing functionality as well.

# MVC Action Filters

Just like AuthorizationFilter ActionFilter is kind of Filter in Asp.Net MVC. It allows us to add pre‐processing and post‐processing logic to action method execution.

ASP.NET MVC provides a simple way to inject your piece of code or logic either before or after an action is executed. This is achieved by decorating the controllers or actions with ASP.NET MVC attributes or custom attributes. An attribute or custom attribute implements the ASP.NET MVC filters(filter interface) and can contain your piece of code or logic. You can make your own custom filters or attributes either by implementing ASP.NET MVC filter interface or by inheriting and overriding methods of ASP.NET MVC filter attribute class if available.

Typically, Filters are used to perform the following common functionalities in your ASP.NET MVC application.

1. Custom Authentication
2. Custom Authorization(User based or Role based)
3. Error handling or logging
4. User Activity Logging
5. Data Caching
6. Data Compression

## Types of Filters

The ASP.NET MVC framework provides five types of filters.

1. Authentication filters (New in ASP.NET MVC5)
2. Authorization filters
3. Action filters
4. Result filters
5. Exception filters

### Authentication Filters

This filter is introduced with ASP.NET MVC5. The IAuthenticationFilter interface is used to create CustomAuthentication filter. The definition of this interface is given below-

1. **public interface IAuthenticationFilter**
2. **{**
3. **void OnAuthentication(AuthenticationContext filterContext);**
5. **void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext);**
6. **}**

You can create your CustomAuthentication filter attribute by implementing IAuthenticationFilter as shown below-

1. **public class CustomAuthenticationAttribute : ActionFilterAttribute, IAuthenticationFilter**
2. **{**
3. **public void OnAuthentication(AuthenticationContext filterContext)**
4. **{**
5. ***//Logic for authenticating a user***
6. **}**
7. ***//Runs after the OnAuthentication method***
8. **public void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext)**
9. **{**
10. ***//TODO: Additional tasks on the request***
11. **}**
12. **}**

### Authorization Filters

The ASP.NET MVC Authorize filter attribute implements the IAuthorizationFilter interface. The definition of this interface is given below-

1. **public interface IAuthorizationFilter**
2. **{**
3. **void OnAuthorization(AuthorizationContext filterContext);**
4. **}**

The AuthorizeAttribute class provides the following methods to override in the CustomAuthorize attribute class.

1. **public class AuthorizeAttribute : FilterAttribute, IAuthorizationFilter**
2. **{**
3. **protected virtual bool AuthorizeCore(HttpContextBase httpContext);**
4. **protected virtual void HandleUnauthorizedRequest(AuthorizationContext filterContext);**
5. **public virtual void OnAuthorization(AuthorizationContext filterContext);**
6. **protected virtual HttpValidationStatus OnCacheAuthorization(HttpContextBase httpContext);**
7. **}**

In this way you can make your CustomAuthorize filter attribute either by implementing IAuthorizationFilter interface or by inheriting and overriding above methods of AuthorizeAttribute class.

### Action Filters

Action filters are executed before or after an action is executed. The IActionFilter interface is used to create an Action Filter which provides two methods OnActionExecuting and OnActionExecuted which will be executed before or after an action is executed respectively.

1. **public interface IActionFilter**
2. **{**
3. **void OnActionExecuting(ActionExecutingContext filterContext);**
4. **void OnActionExecuted(ActionExecutedContext filterContext);**
5. **}**

### Result Filters

Result filters are executed before or after generating the result for an action. The Action Result type can be ViewResult, PartialViewResult, RedirectToRouteResult, RedirectResult, ContentResult, JsonResult, FileResult and EmptyResult which derives from the ActionResult class. Result filters are called after the Action filters. The IResultFilter interface is used to create an Result Filter which provides two methods OnResultExecuting and OnResultExecuted which will be executed before or after generating the result for an action respectively.

1. **public interface IResultFilter**
2. **{**
3. **void OnResultExecuted(ResultExecutedContext filterContext);**
4. **void OnResultExecuting(ResultExecutingContext filterContext);**
5. **}**

### Exception Filters

Exception filters are executed when exception occurs during the actions execution or filters execution. The IExceptionFilter interface is used to create an Exception Filter which provides OnException method which will be executed when exception occurs during the actions execution or filters execution.

1. **public interface IExceptionFilter**
2. **{**
3. **void OnException(ExceptionContext filterContext);**
4. **}**

ASP.NET MVC HandleErrorAttribute filter is an Exception filter which implements IExceptionFilter. When HandleErrorAttribute filter receives the exception it returns an Error view located in the Views/Shared folder of your ASP.NET MVC application.

## Order of Filter Execution

All ASP.NET MVC filter are executed in an order. The correct order of execution is given below:

1. Authentication filters
2. Authorization filters
3. Action filters
4. Result filters

## Configuring Filters

You can configure your own custom filter into your application at following three levels:

### Global level

By registering your filter into Application\_Start event of Global.asax.cs file with the help of FilterConfig class.

* 1. **protected void Application\_Start()**
  2. **{**
  3. **FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);**
  4. **}**

### Controller level

By putting your filter on the top of the controller name as shown below-

* 1. **[Authorize(Roles="Admin")]**
  2. **public class AdminController : Controller**
  3. **{**
  4. ***//***
  5. **}**

### Action level

By putting your filter on the top of the action name as shown below-

* 1. **public class UserController : Controller**
  2. **{**
  3. **[Authorize(Users="User1,User2")]**
  4. **public ActionResult LinkLogin(string provider)**
  5. **{**
  6. ***// TODO:***
  7. **return View();**
  8. **}**
  9. **}**

# How to stop direct execution of Partial View Action

Using MVC Action Filters

# Sections in View Page [Concept of Layout Page]

Implement Consistent look across project

In ASP.NET world consistent layout means MasterPage.

ASP.NET MVC is not different from it. In Razor Master Pages are called as Layout Pages.

Before we go and start with our lab, first let’s discuss what all things we have to place in Layout Page

1. Header with Welcome Message

2. Footer with Footer Data

**Biggest problem?**

Data for Footer and Header is passed to view from Controller as a part of ViewModel

Now the big question is, how data will be passed from View to Layout Page after header and footer are moved to Layout Page.

**Solution – Inheritance**

We can simply follow the Object oriented Inheritance principle here. Let’s understand it with a lab.

Step 1 – Create Base class ViewModel

public class BaseViewModel

{

public string UserName { get; set; }

public FooterViewModel FooterData { get; set; }*//New Property*

}

Step 2 – Prepare EmployeeListViewModel

public class EmployeeListViewModel:BaseViewModel

{

public List<EmployeeViewModel> Employees { get; set; }

}

Step 3 – Create Layout Page

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device‐width" />

<title>@ViewBag.Title</title>

</head>

<body>

<div>

@RenderBody()

</div>

</body>

</html>

Step 4 – Convert Layout to strongly typed Layout

@using WebApplication1.ViewModels

@model BaseViewModel

Step 5 – Design Layout Page

In the layout page add header, footer and three sections for contents as below.

<html>

<head>

<meta name="viewport" content="width=device‐width" />

<title>@RenderSection("TitleSection")</title>

@RenderSection("HeaderSection",false)

</head>

<body>

<div style="text‐align:right">

Hello, @Model.UserName

<a href="/Authentication/Logout">Logout</a>

</div>

<hr />

<div>

@RenderSection("ContentBody")

</div>

@Html.Partial("Footer",Model.FooterData)

</body>

</html>

As you can see we have created three sections in the layout page. TitleSection, HeaderSection and ContentBody. Content pages will use these sections for defining appropriate contents.

Note: While defining HeaderSection second parameter is passed. This parameter decides whether it’s the optional section or compulsory section. False indicates it’s an optional section.

Step 6 – Attach Layout Page to Index View

@{

Layout = "~/Views/Shared/MyLayout.cshtml";

}

Step 7 – Design Index View

1. Take out Headers and Footers from Index View.

2. Copy remaining contents in the body tag and keep it somewhere

3. Now copy contents of title tag.

4. Remove all the HTML contents from the view. Make sure you just remove HTML, @model and layout statement shouldn’t be touched.

5. Define TitleSection and Contentbody with the contents copied earlier.

Complete View will look like below.

@using WebApplication1.ViewModels

@model EmployeeListViewModel

@{

Layout = "~/Views/Shared/MyLayout.cshtml";

}

@section TitleSection{MyView}

@section ContentBody{

<div>

@{

Html.RenderAction("GetAddNewLink");

}

<table border="1">

<tr>

<th>Employee Name</th>

<th>Salary</th>

</tr>

@foreach (EmployeeViewModel item in Model.Employees)

{

<tr>

<td>@item.EmployeeName</td>

<td style="background‐color:@item.SalaryColor">@item.Salary</td>

</tr>

}

</table>

</div>

}

CreateEmployee View is a strongly typed view of type CreateEmployeeViewModel and it’s not the child of BaseViewModel and hence such error occurred.

Solution: Inherit CreateEmployeeViewModel from BaseViewModel as follow

**What does RenderBody do?**

When we first created in the Layout page it had one razor statement something like below.

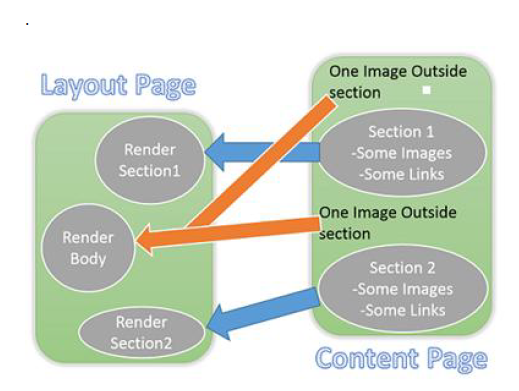
@Html.RenderBody()

In content pages we normally define sections which are declared in the Layout Page.

But a strange thing is, Razor allow us to define some contents outside the section too.

All the non‐section contents in content page will be rendered by RenderBody function

Below image explains it better.



**Can we have nested layouts?**

Yes we can. We can create a layout page which will use some other layout page. Syntax will be same.

**Is it required to specify layout page in each and every view?**

You will find a special view called \_\_ViewStart.cshtml inside Views folder. Setting defined inside this will get applied to all Views.

Example – simply put following code in \_\_ViewStart.cshtml and it will set layout page to all other views.

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

**Is it required to put Header and FooterData code in each and every Action Method?**

No, it’s not required. We can remove this repetition with the help of Action Filter.

**Is it compulsory to define all the sections in child view?**

Yes if section is declared as a required section. Default value will be true.

@RenderSection("HeaderSection",false) // Not required

@RenderSection("HeaderSection",true) // required

@RenderSection("HeaderSection") // required

# Avoiding Repetitive Code using Filters

Making Header and FooterData code more efficient with Action Filter

Step 1 – Remove redundant code from Action Methods

Step 2 – Create HeaderFooterFilter

Create a new class called HeaderFooterFilter in Filters folder and upgrade it to Action filter by inheriting it from ActionFilterAttribute class.

Step 3 ‐ Upgrade ViewModel

Override OnActionExecuted in HeaderFooterFilter class. In this method get the current view model and attach Header and Footer Data.

public class HeaderFooterFilter : ActionFilterAttribute

{

public override void OnActionExecuted(ActionExecutedContext filterContext)

{

ViewResult v = filterContext.Result as ViewResult;

if(v!=null) // v will null when v is not a ViewResult

{

BaseViewModel bvm = v.Model as BaseViewModel;

if(bvm!=null)//bvm will be null when we want a view without Header and footer

{

bvm.UserName = HttpContext.Current.User.Identity.Name;

bvm.FooterData = new FooterViewModel();

bvm.FooterData.CompanyName = "StepByStepSchools";//Can be set to dynamic

value

bvm.FooterData.Year = DateTime.Now.Year.ToString();

}

}

}

}

OnActionExecuted will be used to add post processing logic to action method execution.

Step 4 – Attach Filter

Attach HeaderFooterFilter to Index, AddNew and SaveEmployee action methods.

[HeaderFooterFilter]

public ActionResult Index()

{

EmployeeListViewModel employeeListViewModel = new EmployeeListViewModel();

...

}.

..

[AdminFilter]

[HeaderFooterFilter]

public ActionResult AddNew()

{

CreateEmployeeViewModel employeeListViewModel = new CreateEmployeeViewModel();

//employeeListViewModel.FooterData = new FooterViewModel();

//employeeListViewModel.FooterData.CompanyName = "StepByStepSchools";

...

}.

..

[AdminFilter]

[HeaderFooterFilter]

public ActionResult SaveEmployee(Employee e, string BtnSubmit)

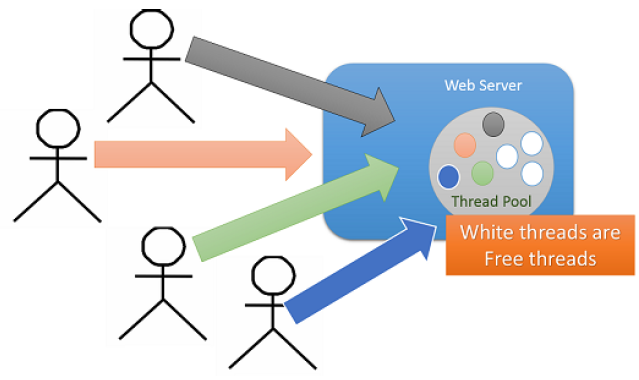
{

switch (BtnSubmit

# Async Controller

A simple programming rule – everything in a program is executed by a thread even a request. In case of Asp.net on the webserver .net framework maintains a pool of threads.

Each time a request is sent to the webserver a free thread from the pool is allocated to serve the request. This thread will be called as worker thread.



Worker thread will be blocked while the request is being processed and cannot serve another request. Now let’s say an application receives too many requests and each request will take long time to get completely processed. In this case we may end up at a point where new request will get into a state where there will be noworker thread available to serve that request. This is called as Thread Starvation. In our case sample file had 2 employee records but in real time it may contain thousands or may be lacks of records. It means request will take huge amount of time to complete the processing. It may leads to Thread Starvation.

**Solution**

Now the request which we had discussed so far is of type synchronous request.

Instead of synchronous if client makes an asynchronous request, problem of thread starvation get solved. In case of asynchronous request as usual worker thread from thread pool get allocated to serve the request.

Worker thread initiates the asynchronous operation and returned to thread pool to serve another request.

Asynchronous operation now will be continued by CLR thread.

Now the problem is, CLR thread can’t return response so once it completes the asynchronous operation it notifies ASP.NET.

Webserver again gets a worker thread from thread pool and processes the remaining request and renders the response.

In this entire scenario two times worker thread is retrieved from thread pool. Now both of them may be same thread or they may not be.

Now in our example file reading is an I/O bound operation which is not required to be processed by worker thread. So it’s a best place to convert synchronous requests to asynchronous requests.

**Does asynchronous request improves response time?**

No, response time will remain same. Here thread will get freed for serving other requests.

Solve thread starvation problem

In ASP.NET MVC we can convert synchronous requests to asynchronous requests by converting synchronous action methods to asynchronous action methods.

Step 1 ‐ Create asynchronous Controller

Change base class of UploadController to AsynController from Controller.

{

public class BulkUploadController : AsyncController

{

Step 2 – Convert synchronous action method to asynchronous action method

It can be done easily with the help of two keywords – async and await.

[AdminFilter]

public async Task<ActionResult> Upload(FileUploadViewModel model)

{

int t1 = Thread.CurrentThread.ManagedThreadId;

List<Employee> employees = await Task.Factory.StartNew<List<Employee>>

(() => GetEmployees(model));

int t2 = Thread.CurrentThread.ManagedThreadId;

EmployeeBusinessLayer bal = new EmployeeBusinessLayer();

bal.UploadEmployees(employees);

return RedirectToAction("Index", "Employee");

}

As you can see we are storing thread id in a variable in the beginning and in the end of action method.

Let’s understand the code.

* When upload button is clicked by the client, new request will be sent to the server.
* Webserver will take a worker thread from thread pool and allocate it to serve the request.
* Worker thread make action method to execute.
* Worker method starts an asynchronous operation with the help of Task.Factory.StartNew method.
* As you can see action method is marked as asynchronous with the help of async keyword. It will make sure that worker thread get released as soon as asynchronous operation starts. Now logically asynchronous operation will continue its execution in the background by a separate CLR thread.
* Now asynchronous operation call is marked with await keyword. It will make sure that next line wont executes unless asynchronous operation completes.
* Once Asynchronous operation completes next statement in the action method should execute and for that again a worker thread is required. Hence webserver will simply take up a new free worker thread from thread pool and allocate it to serve the remaining request and to render response.

While debug you can see we have different thread id in the beginning and different in end. Output is going to became as previous lab.

# Error or exception Handling log exception, Exception Filters

**What are Exception Filters?**

Exception Filters will be used in the same way other filters are used. We will use them as an attribute.

Steps for using exception filter

* Enable them.
* Apply them as an attribute to an action method or to a controller. We can also apply exception filter at global level.

**What they will do?**

Exception filter will take control of the execution and start executing code written inside it automatically as soon exception occurs inside action method.

**Is there any automation?**

ASP.NET MVC provides us one readymade Exception Filter called HandeError.

As we said before it will execute when exception occurs in an action method. This filter will find a view inside “~/Views/[current controller]” or “~/Views/Shared” folder with name “Error” , create the ViewResult of that view and return as a response.

Step 1 – Create a sample Upload file with Error

Step 2 – Execute and Test for exception

Step 3 – Enable Exception Filters

Exception filters get enabled when custom exception is enabled. To enable Custom Exception, open web.config file and navigate to System.Web Section. Add a new section for custom errors as below.

<system.web>

<customErrors mode="On"></customErrors>

Step 4 – Create Error View

In “~/Views/Shared” folder you will find a view called “Error.cshtml”. This file was created as a part of MVC template in the beginning only. In case if it is not created , create it manually.

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device‐width" />

<title>Error</title>

</head>

<body>

<hgroup>

<h1>Error.</h1>

<h2>An error occurred while processing your request.</h2>

</hgroup>

</body>

</html>

Step 5 – Attach Exception filter

Open FilterConfig.cs file from App\_Start folder. In RegisterGlobalFilters method you will see that HandleError filter is already attached at global level.

public static void RegisterGlobalFilters(GlobalFilterCollection filters)

{

filters.Add(new HandleErrorAttribute());//ExceptionFilter

filters.Add(new AuthorizeAttribute());

}

If required remove global filter and attach it at action or controller level as below.

[AdminFilter]

[HandleError]

public async Task<ActionResult> Upload(FileUploadViewModel model)

{<actionresult>

</actionresult>

Step 7 – Display Error message in the view

In order to achieve this convert Error view to strongly typed view of type HandleErrorInfo class and then display error messages in the view.

@model HandleErrorInfo

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device‐width" />

<title>Error</title>

</head>

<body>

<hgroup>

<h1>Error.</h1>

<h2>An error occurred while processing your request.</h2>

</hgroup>

Error Message :@Model.Exception.Message<br />

Controller: @Model.ControllerName<br />

Action: @Model.ActionName

</body>

</html>

But if we put in url anything weired then it will not show the custom error page.

Step 9 – Create ErrorController as follows

Create a new controller called ErrorController in Controller folder and create an action method called Index as follows.

public class ErrorController : Controller

{

// GET: Error

public ActionResult Index()

{

Exception e=new Exception("Invalid Controller or/and Action Name");

HandleErrorInfo eInfo = new HandleErrorInfo(e, "Unknown", "Unknown");

return View("Error", eInfo);

}

}

HandleErrorInfo constructor takes 3 arguments – Exception object, controller name and Action method Name.

Step 10 – Display Custom Error View on Invalid URL

In web.config define setting for “Resource not found error” as follows.

<system.web>

<customErrors mode="On">

<error statusCode="404" redirect="~/Error/Index"/>

</customErrors>

Step 11 ‐ Make ErrorController accessible to everyone

[AllowAnonymous]

public class ErrorController : Controller

{

**Is it possible to change the view name?**

Yes, it’s not required to keep view name as “Error” always.

In that case we have to specify view name while attaching HandlError filter.

[HandleError(View="MyError")]

Or

filters.Add(new HandleErrorAttribute()

{

View="MyError"

});

[HandleError(View="DivideError",ExceptionType=typeof(DivideByZeroException))]

[HandleError(View = "NotFiniteError", ExceptionType = typeof(NotFiniteNumberException))]

[HandleError]

Or

filters.Add(new HandleErrorAttribute()

{

ExceptionType = typeof(DivideByZeroException),

View = "DivideError"

});

filters.Add(new HandleErrorAttribute()

{

ExceptionType = typeof(NotFiniteNumberException),

View = "NotFiniteError"

});

filters.Add(new HandleErrorAttribute());

In above case we are adding Handle Error filter thrice. First two are specific to exception whereas last one is more general one and it will display Error View for all other exceptions.

we are not logging our exception anywhere

# Routing Concept, User friendly URL

routes.MapRoute(

name: "CabsInJhansi",

url: "CabsInJhansi",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

routes.MapRoute(

name: "BookCabsJhansi",

url: "BookCabsJhansi",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

routes.MapRoute(

name: "CabsAtJhansi",

url: "CabsAtJhansi",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

routes.MapRoute(

name: "CabsAtJhansiRailwayStation",

url: "CabsAtJhansiRailwayStation",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

# Difference between Route Parameter and Query String

**What is the difference between Route Parameter and Query String?**

* Query String have size limitation whereas we can define any number of Route Parameters.
* We cannot add constraints to Query String values but we can add constraints to Route Parameters.
* Default Value for Route Parameter is possible whereas default value for Query String is not possible.
* Query String makes URL cluttered whereas Route Parameter keep it clean.

**How to apply constraints to Route Parameter?**

It can be done with the help of regular expressions.

Example: look at the following route.

routes.MapRoute(

"MyRoute",

"Employee/{EmpId}",

new {controller=" Employee ", action="GetEmployeeById"},

new { EmpId = @"\d+" }

);

Action method will look like below.

public ActionResult GetEmployeeById(int EmpId)

{

...

}

Now when someone make a request with URL “http://..../Employee/1” or “http://..../Employee/111”, action

method will get executed but when someone make a request with URL <http://..../Employee/Sukesh> he/she will get “Resource Not Found” Error.

**Is it a required to keep parameter name in action method same as Route Parameter Name?**

Basically a single Route pattern may contain one or more RouteParameters involved in it. To identify each route parameter independently it is must to keep parameter name in action method same as Route Parameter Name.

Does sequence while defining custom routes matters?

Yes, it matters. If you remember UrlRoutingModule will take first matching route object.

# Alternate ways of Mapping URL pattern

**Is there an easier way to define URL pattern for Action method?**

We can use Attribute based routing for that.

Step 1 – Enable Attribute based routing.

In RegisterRoutes method keep following line after IgnoreRoute statement.

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapMvcAttributeRoutes();

routes.MapRoute(

...

Step 2 – Define route pattern for an action method

Simply attach Route attribute to Action method to Index action of EmployeeController as follows.

[Route("Employee/List")]

public ActionResult Index()

{

**Can we define Route Parameters with attribute based routing? Can we define Route Parameters with attribute based routing?**

Yes, look at the following syntax.

[Route("Employee/List/{id}")]

publicActionResult Index (string id) { ... }

**What about the constraints in this case?**

[Route("Employee/List/{id:int}")]

We can have following constraints

1. {x:alpha} – string validation

2. {x:bool} – Boolean validation

3. {x:datetime} – Date Time validation

4. {x:decimal} – Decimal validation

5. {x:double} – 64 bit float point value validation

6. {x:float} – 32 bit float point value validation

7. {x:guid} – GUID validation

8. {x:length﴾6﴿} –length validation

9. {x:length﴾1,20﴿} – Min and Max length validation

10. {x:long} – 64 int validation

11. {x:max﴾10﴿} – Max integer number validation

12. {x:maxlength﴾10﴿} – Max length validation

13. {x:min﴾10﴿} – Min Integer number validation

14. {x:minlength﴾10﴿} – Min length validation

15. {x:range﴾10,50﴿} – Integer range validation

16. {x:regex﴾SomeRegularExpression﴿} – Regular Expression validation

# Single Page Application

# Areas in MVC

What are Areas?

Areas are simply a way to implement modules in Asp.Net MVC project.

Every project is made up of more than one module. Example ‐ Account Module, Customer Relationship module, Payment gateway module etc.

In traditional application development style we usually use “Folders” to achieve this. We create more than one folder inside single project. Each folder represents one module. We will keep all the respective files of respective module in respective folders.

Now custom folders will be a big issue when it comes to Asp.Net MVC.

Let’s say in Asp.Net MVC we will use simple folders for implementing module.

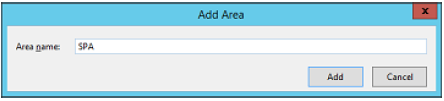
* Now DataAccessLayer, BusinessLayer, BusinessEntities and ViewModels won’t create any issue. They are simple classes so can be kept anywhere.
* Controllers ‐ We can’t keep it anywhere. It must be kept inside Controller folder. But it is not going to be big issue because from MVC 4 restriction over controller location was removed. Now we can keep it wherever we want.
* Views ‐unfortunately for views it’s not possible. All the views must be placed inside

“~/Views/ControllerName” or “~/Views/Shared” folder.

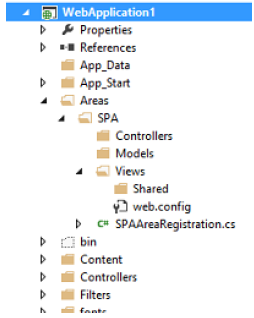
This is where Areas comes to picture.

Step 1 – Create a new Area

Right click the project and Select Add>>Area. A dialogue box will popup. Put it as SPA and select ok.



It will create a new folder structure inside our project as follows.



Obviously Model folder related to the area is not required. Delete it.

Step 2 – Create required ViewModels

Create a new folder called SPA in ViewModel class library project and create a ViewModel called MainViewModel as follows.

using WebApplication1.ViewModels;

namespace WebApplication1.ViewModels.SPA

{

public class MainViewModel

{

public string UserName { get; set; }

public FooterViewModel FooterData { get; set; }*//New Property*

}

}

Step 3–Create Index action method

Put following using statement in MainController.

using WebApplication1.ViewModels.SPA;

using OldViewModel=WebApplication1.ViewModels;

Now create a new action method called Index inside MainController as follows.

public ActionResult Index()

{

MainViewModel v = new MainViewModel();

v.UserName = User.Identity.Name;

v.FooterData = new OldViewModel.FooterViewModel();

v.FooterData.CompanyName = "StepByStepSchools";*//Can be set to dynamic value*

v.FooterData.Year = DateTime.Now.Year.ToString();

return View("Index", v);

}

As you can see an alias called OldViewModelis added for WebApplication1.ViewModels namespace. Now instead of writing WebApplication1.ViewModels.ClassName we can simply write

OldViewModel.ClassName.

Not specifying alias name will leads to ambiguity error. Both namesapces

WebApplication1.ViewModels.SPA and WebApplication1.ViewModels have some similar classes.

Step 4 – Create Index View

Create a view associated with above index method as follows.

@using WebApplication1.ViewModels.SPA

@model MainViewModel

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device‐width" />

<title>Employee Single Page Application</title>

**Why SPA keyword is required before controller name?**

When we add an area to an ASP.NET MVC application, Visual Studio creates a file named

[AreaName]AreaRegistration.cs containing a class that derives from AreaRegistration. This class defines the AreaName property and the RegisterArea method, which registers the route information for the new area.

In our case you can find that file with name SpaArealRegistration.cs and will be located inside

“~/Areas/Spa” folder. RegisterArea method of SpaArealRegistration class contain following code.

context.MapRoute(

"SPA\_default",

"SPA/{controller}/{action}/{id}",

new { action = "Index", id = UrlParameter.Optional }

);

This is why, when it comes to controllers in the area we put SPA keyword before controller name.

**How come RegisterArea in SPAAreaRegistration class get invoked?**

Open global.asax file. Very first line in Application\_Start will be something like below.

AreaRegistration.RegisterAllAreas();

The RegisterAllAreas method finds all types in the application domain that derive from AreaRegistration and calls each of their RegisterArea methods.

**Can we invoke MainController actions without using SPA?**

Let’s make this question easy.

Question will this URL works? “localhost:8870/Main/Index”

Answer is – Yes.

AreaRegistration class creates new route but it won’t remove other routes. Route defined in RouteConfig class still work correctly. As I said before there is no restriction for controller location. Hence it will work but output won’t be displayed properly because it won’t be able to find the view. I suggest you to execute your application once and try it.

# Temp Data in MVC

Understand TempData in Asp.Net MVC

TempData behaves differently in different situations. Incorporating all of the scenario in single project is not possible. That’s why we will just try to cover and understand all the scenarios step by step.

**What is TempData?**

TempData is a session with shorten life cycle. Just like session it’s a key value pair used to hold user specific values. Difference between Session and TempData is, TempData let us maintain data only within single request cycle.

To understand TempData properly you must understand the meaning “Single Request” properly

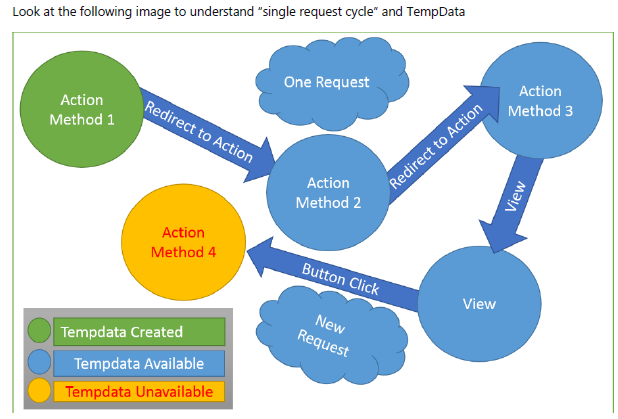
User make a request via browser’s address bar which is handled by an action method. Let’s call it Request 1.

Now unless end user get a final response it will be called as request 1. Once end user get a

response request ends there.

Now by clicking a hyperlink, button or again with the help of browser’s address bar end user can make a new request.

Look at the following image to understand “single request cycle” and TempData



Code Example:

public ActionResult M1()

{

TempData["a"] = "Value";

string s = TempData["a"].ToString(); *// TempData will be available*

return RedirectToAction("M2");

}

public ActionResult M2()

{

string s = TempData["a"].ToString(); *//TempData will be available*

return RedirectToAction ("M3");

}

public ActionResult M3()

{

string s = TempData["a"].ToString();*// TempData will be available*

return view ("Some View"); *// TempData will be available inside view also􀀀*

}

It doesn’t matter if it is a view or an action method. Unless request completes values in the TempData will be available.

**When the TempData values get removed?**

At the end of the request, values in the TempData get removed automaticallyifvalues are marked

for deletion.

Values will be marked for deletion as soon as we read it.

**Example 1 ‐**

public ActionResult M1()

{

TempData["a"] = "Value";

return view("MyView");

}.

..

...

...

View Code

...

...

@{

string s = TempData["a"].ToString(); // TempData is marked for deletion

}

**Example 2 ‐**

public ActionResult M1()

{

TempData["a"] = "Value";

string s = TempData["a"].ToString(); // TempData is marked for deletion

return view("MyView"); // MyView can access TempData because it will be deleted at the end

of the request

}

Note: In above two examples values are marked for deletion as soon it get read. Now for the next request

this values won’t be available because at the end of the current request current TempData get deleted.

What will happen if the TempData values are not read?

As per the final explanation we had, TempData values will be deleted at the end of the current request considering they are marked for deletion in current request and they will get marked for deletion only if we read it. What will happened if we won’t read it?

Simple answer, in that case TempData will be available to the next request as well. Now TempData will get removed at the end of the next request if it is marked for deletion in next request else this story continues.

**Code Example**

public ActionResult M1()

{

TempData["a"] = "Value";

return view("MyView"); *// TempData will be available inside view but let&rsquo;s assume MyView*

*won&rsquo;t read the value*

}

*//Immediate next request after M1 made by end user*

public ActionResult M2()

{

string s = TempData["a"].ToString(); *// TempData will be available and now its marked for*

*deletion*

return view("MyView2"); *// TempData will be available inside view also because request is not*

*ended yet.*

}

# Keep and Peek Methods

**Keep**

Keep method makes the values in the TempData persistent for next request.

Example 1

public ActionResult M1()

{

TempData["a"] = "Value";

TempData["b"] = "Value1";

TempData["c"] = "Value2";

TempData.Keep();

return view("MyView");

}

View:

@{

string s= TempData["a"].ToString();

string s1= TempData["b"].ToString();

string s2= TempData["c"].ToString();

}

**Peek**

Peek will let us retrieve the TempData value without marking it for deletion.

Example with Peek:

public ActionResult M1()

{

TempData["a"] = "Value";

return RedirectToAction("M2");

}p

ublic ActionResult M2()

{

string s = TempData.Peek("a").ToString(); // TempData values will be read but will not be

marked for deletion

return view ("Some View");

}...p

ublic ActionResult M3()

{

string s = TempData["a"].ToString(); // Will just work

&hellip;

}

In above example TempData value will be available in next request also.

# Stop CSRF Attack

CSRF stands for Cross site request forgery.

Here hacker will develop a two faced UI. Two faced UI means, a UI which will do something different according to end user but in reality it will do something different.

Explanation: ‐

You were expecting to get a download pdf link but it actually created a new employee.

Let’s understand what exactly happened. End user is able to create an employee only if he is an Authenticated Admin user. In our case Admin user will login using his admin credentials and after sometime he will go and open a new browser instance ﴾or browser tab﴿ and start doing some surfing. ﴾Remember his current session is active and he is still authenticated user for our application﴿

All of the sudden above forgery page ﴾created by some enemy﴿ comes in front of him and he/she click the “Send link” button. “Send link” button will actually send a post request to our

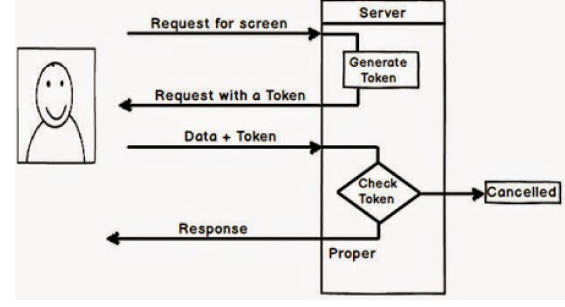
application. You know what happens next.

This is called CSRF attack.

Solution for this problem in tokens.

In actual data entry screen server will inject a secret token in the form of hidden field.

It means when post request is made from actual data entry screen, secret token will be sent as a part of posted data. At server side received token will be validated against generated token and if they are not same request won’t be served.



Step 1 – Inject token in original data entry screen

<td colspan="2">

@Html.AntiForgeryToken()

<input type="submit" name="BtnSubmit" value="Save Employee" onclick="return IsValid();" />

<input type="submit" name="BtnSubmit" value="Cancel" />

<input type="button" name="BtnReset" value="Reset" onclick="ResetForm();" />

</td>

Step 2 –Enable forgery validation on Action Method

Attach ValidateAntiForgeryToken attribute to SaveEmployee action method of EmployeeController as

follows.

[AdminFilter]

[HeaderFooterFilter]

[ValidateAntiForgeryToken]

public ActionResult SaveEmployee(Employee e, string BtnSubmit)

{

switch (BtnSubmit)

{

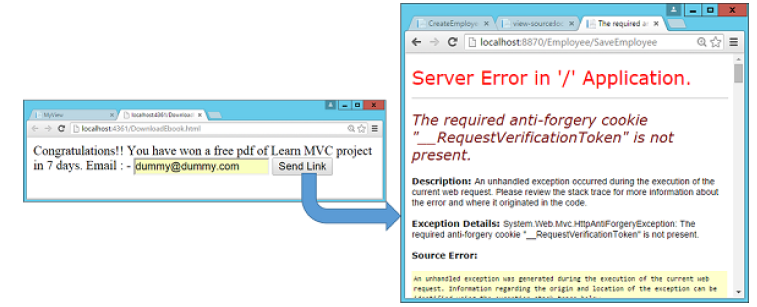
Step 3 – Check the generated token in Data Entry screen

Press F5 and execute the application. Complete the login process. Navigate to AddNew screen and check the view source.



Step 4 – Check forgery

Keep the project login open and execute the forgery project and perform the same testing we did before.



Now it’s secured.

# MVC Bundling and Minification

Understanding Bundling

We cannot imagine a project without CSS and JavaScript file.

Bundling is concept of combining multiple JavaScript or CSS files into a single fileat runtime. Now the million dollar question is why should we combine?

Just try to imagine a situation when we have lots of JavaScript file. It leads to multiple requests thus decreases performance.

Solution will be combine all the JS files into single bundle and request it in a single request as a single unit. This process is termed as bundling.

Understanding Minification

Minification reduces the size of script and CSS files by removing blank spaces, comments etc. For example below is a simple JavaScript code with comments.

// This is test

var x = 0;

x = x + 1;

x = x \* 2;

After implementing Minification the JavaScript code looks something as below. You can see how whitespaces and comments are removed to minimize file size and thus increases performance as the file size has get reduced.

var x=0;x=x+1;x=x\*2;

**Implementing bundling and minification in Asp.Net MVC**

Step 1 – Create Script bundles

Open BundleConfig.cs file from App\_Start folder. You will find a RegisterBundle method. Remove all existing code inside it and redefine it as follows.

public static void RegisterBundles(BundleCollection bundles)

{

bundles.Add(new ScriptBundle("~/bundles/jqueryValidation").Include(

"~/Scripts/jquery‐1.8.0.js").Include(

"~/Scripts/jquery.validate.js").Include(

"~/Scripts/jquery.validate.unobtrusive.js"));

}

Step 2 ‐ Include bundle in Login View

Open Login.cshtml view from “~/Views/Authentication” folder.

Remove the references of three script files and include it has one bundle.

<title>Login</title>

@Scripts.Render("~/bundles/jqueryValidation")

</head>

<body>

Step 3 – Execute and test

Execute the application and check the network tab again.

Unfortunately you will not find any difference.

Step 4 – **Enable bundling and minification**

Open Global.asax file and in put following line in the top of Application\_Start event.

**BundleTable.EnableOptimizations = true;**

As you can see a single file is requested instead of three. All three JavaScript files are bundled into one.

Check the size. It is just 294B. Earlier it was ﴾289+294+310﴿. It proves that minification also did his work.

﴾Don’t forget to check the validation.﴿

In the same we create bundles for CSS files called StyleBundle.

# JSON

To make the communication of Data from one technology to another technology we need some common data type, because one data type in one technology is different from the other technology and there are different solution approached as below:

1. Convert to string: because in any technology we can convert any data type to string like int, float can be stores in string. But the problem with this approach is that complex data type cannot be treated as in string.

Hence the second approach is:

1. XML Data type: So complex data type also can be store in the XML format and can be read also in different technologies.

But again we have few drawbacks:

* 1. XML format requires huge amount of data flow on the network.
  2. It requires another technology that they are able to parse and read these XML data.

But if we are sending data from .Net to Java Script it does not have that much rich library that it can serialize and desterilize the object from XML.

1. The another solution is JSON

In order to tackle the problems of XML format, industry came up with a new format called JSON. It’s an abbreviation for “JavaScript Object Notation”.

Employee object created in above C# code may be represented in JSON format as below.

{

EmpName: "Sukesh",

Address: "Mumbai"

}

Data represented in JSON format look like JavaScript object hence this format is named as JSON ﴾JavaScript Object Notation﴿

As you can see its lightweight compared to earlier.

Readymade functions are available in JavaScript for converting JavaScript object to JSON formatted string and for parsing JSON formatted string to JavaScript object.

Following code explains creation and parsing of JSON string.

var e={

EmpName= &ldquo;Sukesh&rdquo;,

Address= &ldquo;Mumbai&rdquo;

};

var EmployeeJsonString = JSON.stringify(e);//This EmployeeJsonString will be send to other

technologies.

var EmployeeJsonString=GetFromOtherTechnology();

var e=JSON.parse(EmployeeJsonString);

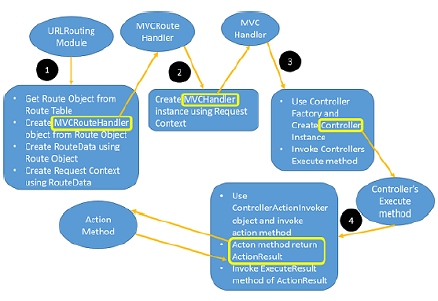
alert(e.EmpName);

alert(e.Address);

**What does JSON method do?**

It will return JSONResult.

JSONResult is the one of the child of ActionResult.



ExecuteResult is declared as abstract in ActionResult class. All the children’s of ActionResult class defines it in its own way. In Day 1 we spoke about ViewResult.In ViewResult class ExecuteResult method will do following things.

* It will creates the object of ViewPageActivator class.
* It will choose the correct ViewEngine and passes ViewPageActivator object as argument to ViewEngine’s constructor. ViewEngine create the object of View class
* It will invoke the RenderView method of View which will render final HTML output the response.

When it comes to JsonResult, ExecuteResult method will,

* Set the response content type to “Application/Json”
* Using JavaScript Serializerit will convert the data passed to JSON formatted string
* Writes the final JSON formatted String to response stream.

# Angular JS