FREE eBook

LEARNING asp.net-mvc

#asp.net-mvc

Table of Contents

About

Chapter 1: Getting started with asp.net-mvc

Remarks

The Model-View-Controller (MVC) architectural pattern separates an application into three main components: the model, the view, and the controller. The ASP.NET MVC framework provides an alternative to the ASP.NET Web Forms pattern for creating Web applications. The ASP.NET MVC framework is a lightweight, highly testable presentation framework that (as with Web Forms-based applications) is integrated with existing ASP.NET features, such as master pages and membership-based authentication. The MVC framework is defined in the System.Web.Mvc assembly.

The MVC framework includes the following components:

- **Models**. Model objects are the parts of the application that implement the logic for the application's data domain. Often, model objects retrieve and store model state in a database. For example, a Product object might retrieve information from a database, operate on it, and then write updated information back to a Products table in a SQL Server database. In small applications, the model is often a conceptual separation instead of a physical one. For example, if the application only reads a dataset and sends it to the view, the application does not have a physical model layer and associated classes. In that case, the dataset takes on the role of a model object.
- **Views**. Views are the components that display the application's user interface (UI). Typically, this UI is created from the model data. An example would be an edit view of a Products table that displays text boxes, drop-down lists, and check boxes based on the current state of a Product object.
- **Controllers**. Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render that displays UI. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles query-string values, and passes these values to the model, which in turn might use these values to query the database.

Versions

Version	.NET Version	Release Date
MVC 1.0	.NET 3.5	2009-03-13
MVC 2.0	.NET 3.5/4.0	2010-03-10
MVC 3.0	.NET 4.0	2011-01-13
MVC 4.0	.NET 4.0/4.5	2012-08-15
MVC 5.0	.NET 4.5	2013-10-17

Version	.NET Version	Release Date
MVC 5.1	.NET 4.5	2014-01-17
MVC 5.2	.NET 4.5	2014-08-28
MVC 6.0	.NET 4.5	2015-11-18
Core MVC 1.0	.NET 4.5	2016-07-12
Core MVC 1.1	.NET 4.5	2016-11-18

Examples

Hello MVC!

ASP.NET MVC is open source web application framework. MVC itself is a design pattern which is built around three main components: *model-view-controller*.

Model - Models reflect your business objects, and are a means to pass data between Controllers and Views.

View - Views are the pages that render and display the model data to the user. ASP.NET MVC Views are typically written using Razor syntax.

Controller - Controllers handle incoming HTTP requests from a client, and usually return one or more Models to an appropriate View.

The ASP.NET MVC features:

1. Ideal for developing complex but light weight applications
2. It provides an extensible and pluggable framework which can be easily replaced and customized. For example, if you do not wish to use the in-built Razor or ASPX View Engine, then you can use any other third-party view engines or even customize the existing ones.
3. Utilizes the component-based design of the application by logically dividing it into Model, View and Controller components. This enables the developers to manage the complexity of large-scale projects and work on individual components.
4. The MVC structure enhances the test-driven development and testability of the application since all the components can be designed interface-based and tested using mock objects. Hence the ASP.NET MVC Framework is ideal for projects with large team of web developers.
5. Supports all the existing vast ASP.NET functionalities such as Authorization and Authentication, Master Pages, Data Binding, User Controls, Memberships, ASP.NET Routing, etc.
6. It does not use the concept of View State (which is present in ASP.NET). This helps in building applications which are light-weight and gives full control to the developers.

Simple MVC application

We are going to create simple MVC application which displays person details. Create new MVC project using Visual Studio. Add new model named *Person* to Models folder as following:

```
public class Person
{
    public string Surname { get; set; }
    public string FirstName { get; set; }
    public string Patronymic { get; set; }
    public DateTime BirthDate { get; set; }
}
```

Add new controller to Controllers folder:

```
public class HomeController : Controller
{
    //Action Method
    public ActionResult Index()
    {
        // Initialize model
        Person person = new Person
        {
            Surname = "Person_SURNAME",
            FirstName = "Person_FIRSTNAME",
            Patronymic = "Person_PATRONYMIC",
            BirthDate = new DateTime(1990, 1, 1)
        };

        // Send model to View for displaying to user
        return View(person);
    }
}
```

Finally add View to */Views/Home/* folder named *Index.cshtml*:

```
@* Model for this view is Person *@
@model Hello_MVC.Models.Person

<h2>Hello @Model.FirstName !</h2>

<div>
    <h5>Details:</h5>
    <div>
        @Html.LabelFor(m => m.Surname)
        @Html.DisplayFor(m => m.Surname)
    </div>
    <div>
        @Html.LabelFor(m => m.FirstName)
        @Html.DisplayFor(m => m.FirstName)
    </div>
    <div>
        @Html.LabelFor(m => m.Patronymic)
        @Html.DisplayFor(m => m.Patronymic)
    </div>
    <div>
        @Html.LabelFor(m => m.BirthDate)
        @Html.DisplayFor(m => m.BirthDate)
    </div>
</div>
```

Read Getting started with asp.net-mvc online: https://riptutorial.com/asp-net-mvc/topic/769/getting-started-with-asp-net-mvc

Chapter 2: Action filters

Examples

A logging action filter

```
public class LogActionFilter : ActionFilterAttribute
{
     public override void OnActionExecuting(ActionExecutingContext filterContext)
     {
          Log("OnActionExecuting", filterContext.RouteData);
     }

     public override void OnActionExecuted(ActionExecutedContext filterContext)
     {
          Log("OnActionExecuted", filterContext.RouteData);
     }

     public override void OnResultExecuting(ResultExecutingContext filterContext)
     {
          Log("OnResultExecuting", filterContext.RouteData);
     }

     public override void OnResultExecuted(ResultExecutedContext filterContext)
     {
          Log("OnResultExecuted", filterContext.RouteData);
     }


     private void Log(string methodName, RouteData routeData)
     {
          var controllerName = routeData.Values["controller"];
          var actionName = routeData.Values["action"];
          var message = String.Format("{0} controller:{1} action:{2}", methodName,
controllerName, actionName);
          Debug.WriteLine(message, "Action Filter Log");
     }
}
```

Session Control action filter - page&ajax request

Usually authentication&authorization processes are performed by built-in cookie and token supports in .net MVC. But if you decide to do it yourself with `Session` you can use below logic for both page requests and ajax requests.

```
public class SessionControl : ActionFilterAttribute
{
    public override void OnActionExecuting ( ActionExecutingContext filterContext )
    {
        var session = filterContext.HttpContext.Session;

        /// user is logged in (the "loggedIn" should be set in Login action upon a successful
login request)
        if ( session["loggedIn"] != null && (bool)session["loggedIn"] )
```

```
            return;

        /// if the request is ajax then we return a json object
        if ( filterContext.HttpContext.Request.IsAjaxRequest() )
        {
            filterContext.Result = new JsonResult
            {
                Data = "UnauthorizedAccess",
                JsonRequestBehavior = JsonRequestBehavior.AllowGet
            };
        }
        /// otherwise we redirect the user to the login page
        else
        {
            var redirectTarget = new RouteValueDictionary { { "Controller", "Login" }, {
"Action", "Index" } };
            filterContext.Result = new RedirectToRouteResult(redirectTarget);
        }
    }

    public override void OnResultExecuting ( ResultExecutingContext filterContext )
    {
        base.OnResultExecuting(filterContext);

        /// we set a field 'IsAjaxRequest' in ViewBag according to the actual request type
        filterContext.Controller.ViewBag.IsAjaxRequest =
filterContext.HttpContext.Request.IsAjaxRequest();
    }
}
```

Action filter usage locations (global, controller, action)

You can place action filters at three possible levels:

1. Global
2. Controller
3. Action

Placing a filter **globally** means it will execute on requests to any route. Placing one on a **controller** makes it execute on requests to any action in that controller. Placing one on an **action** means it runs with the action.

If we have this simple action filter:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = true)]
public class CustomActionFilterAttribute : FilterAttribute, IActionFilter
{
    private readonly string _location;

    public CustomActionFilterAttribute(string location)
    {
        _location = location;
    }

    public void OnActionExecuting(ActionExecutingContext filterContext)
    {
        Trace.TraceInformation("OnActionExecuting: " + _location);
    }

    public void OnActionExecuted(ActionExecutedContext filterContext)
    {
        Trace.TraceInformation("OnActionExecuted: " + _location);
    }
}
```

We can add it on global level by adding it to the global filter collection. With the typical ASP.NET MVC project setup, this is done in App_Start/FilterConfig.cs.

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new CustomActionFilterAttribute("Global"));
    }
}
```

We can also add it on controller and action level like so in a controller:

```
[CustomActionFilter("HomeController")]
public class HomeController : Controller
{
    [CustomActionFilter("Index")]
    public ActionResult Index()
    {
        return View();
    }
}
```

If we run the application and look at the Output window, we will see the following messages:

```
iisexpress.exe Information: 0 : OnActionExecuting: Global
iisexpress.exe Information: 0 : OnActionExecuting: HomeController
iisexpress.exe Information: 0 : OnActionExecuting: Index
iisexpress.exe Information: 0 : OnActionExecuted: Index
iisexpress.exe Information: 0 : OnActionExecuted: HomeController
iisexpress.exe Information: 0 : OnActionExecuted: Global
```

As you can see, when the request comes in, the filters are executed:

1. Global
2. Controller
3. Action

Excellent examples of filters placed on global level include:

1. Authentication filters
2. Authorization filters
3. Logging filters

Exception Handler Attribute

This attribute handles all unhandled exceptions in the code, (this is mostly for Ajax Requests - that deal with JSON - but can be extended)

```
public class ExceptionHandlerAttribute : HandleErrorAttribute
{
    /// <summary>
    ///   Overriden method to handle exception
    /// </summary>
    /// <param name="filterContext"> </param>
    public override void OnException(ExceptionContext filterContext)
    {
        // If exeption is handled - return ( don't do anything)
        if (filterContext.ExceptionHandled)
            return;

        // Set the ExceptionHandled to true ( as you are handling it here)
        filterContext.ExceptionHandled = true;

        //TODO: You can Log exception to database or Log File

        //Set your result structure
        filterContext.Result = new JsonResult
        {
            Data = new { Success = false, Message = filterContext .Exception.Message, data =
new {} },
            JsonRequestBehavior = JsonRequestBehavior.AllowGet
        };

    }
}
```

So let's say you always have to send a JSON response similar to this:

```
{

    Success: true,  // False when Error

    data: {},

    Message:"Success" // Error Message when Error

}
```

So instead of handling exceptions in controller actions, like this:

```
public ActionResult PerformMyAction()
{
    try
    {
        var myData = new { myValue = 1};

        throw new Exception("Handled", new Exception("This is an Handled Exception"));

        return Json(new {Success = true, data = myData, Message = ""});

    }
    catch(Exception ex)
    {
        return Json(new {Success = false, data = null, Message = ex.Message});
    }
}
```

You can do this:

```
[ExceptionHandler]
public ActionResult PerformMyAction()
{
        var myData = new { myValue = 1};

        throw new Exception("Unhandled", new Exception("This is an unhandled Exception"));

        return Json(new {Success = true, data = myData, Message = ""});
}
```

OR you can add at Controller level

```
[ExceptionHandler]
public class MyTestController : Controller
{

    public ActionResult PerformMyAction()
    {
            var myData = new { myValue = 1};

            throw new Exception("Unhandled", new Exception("This is an unhandled Exception"));

            return Json(new {Success = true, data = myData, Message = ""});
    }
}
```

Read Action filters online: https://riptutorial.com/asp-net-mvc/topic/1450/action-filters

Chapter 3: ActionResult

Remarks

An `ActionResult` is best though of as an web endpoint in MVC. Ever ActionResult method can be reached by typing in the appropriate web address as configured by your Routing engine.

Examples

Return a View Page

This ActionResult returns a Razor view page. Under the standard routing template this ActionResult method would be reached at http://localhost/about/me

The View will be looked for automatically in your site at `~/Views/About/Me.cshtml`

```
public class AboutController : Controller
{
    public ActionResult Me()
    {
        return View();
    }
}
```

Return a File

An `ActionResult` can return `FileContentResult` by specifying file path and file type based from extension definition, known as MIME type.

The MIME type can be set automatically depending on file type using `GetMimeMapping` method, or defined manually in proper format, e.g. "text/plain".

Since `FileContentResult` requires a byte array to be returned as a file stream, `System.IO.File.ReadAllBytes` can be used to read file contents as byte array before sending requested file.

```
public class FileController : Controller
{
     public ActionResult DownloadFile(String fileName)
     {
          String file = Server.MapPath("~/ParentDir/ChildDir" + fileName);
          String mimeType = MimeMapping.GetMimeMapping(path);

          byte[] stream = System.IO.File.ReadAllBytes(file);
          return File(stream, mimeType);
     }
}
```

Return a Json

Action result can return Json.

1.Returning Json to transmit json in ActionResult

```
public class HomeController : Controller
{
    public ActionResult HelloJson()
    {
        return Json(new {message1="Hello", message2 ="World"});
    }
}
```

2.Returning Content to transmit json in ActionResult

```
public class HomeController : Controller
{
    public ActionResult HelloJson()
    {
        return Content("Hello World", "application/json");
    }
}
```

Read ActionResult online: https://riptutorial.com/asp-net-mvc/topic/6246/actionresult

Chapter 4: ActionResult

Examples

ViewResult

```
public ActionResult Index()
{
    // Renders a view as a Web page.
    return View();
}
```

Action methods typically return a result that is known as an action result. The ActionResult class is the base class for all action results. The ActionInvoker decide which type of action result to return based on the task that the action method is performing.

It is possible be explicit about what type to return, but generally it not necessary.

```
public ViewResult Index()
{
    // Renders a view as a Web page.
    return View();
}
```

PartialViewResult

```
public ActionResult PopulateFoods()
{
     IEnumerable<Food> foodList = GetAll();

    // Renders a partial view, which defines a section of a view that can be rendered inside 
another view.
    return PartialView("_foodTable", foodVms);;
}
```

Action methods typically return a result that is known as an action result. The ActionResult class is the base class for all action results. The ActionInvoker decide which type of action result to return based on the task that the action method is performing.

It is possible be explicit about what type to return, but generally it not necessary.

```
public PartialViewResult PopulateFoods()
{
    IEnumerable<Food> foodList = GetAll();

    // Renders a partial view, which defines a section of a view that can be rendered inside 
another view.
     return PartialView("_foodTable", foodVms);
}
```

RedirectResult

```
public ActionResult Index()
{
    //Redirects to another action method by using its URL.
    return new RedirectResult("http://www.google.com");
}
```

Action methods typically return a result that is known as an action result. The ActionResult class is the base class for all action results. The ActionInvoker decide which type of action result to return based on the task that the action method is performing.

It is possible be explicit about what type to return, but generally it not necessary.

```
public RedirectResult Index()
{
    //Redirects to another action method by using its URL.
    return new RedirectResult("http://www.google.com");
}
```

RedirectToRouteResult

```
public ActionResult PopulateFoods()
{
    // Redirects to another action method. In this case the index method
    return RedirectToAction("Index");
}
```

Action methods typically return a result that is known as an action result. The ActionResult class is the base class for all action results. The ActionInvoker decideы which type of action result to return based on the task that the action method is performing.

It is possible be explicit about what type to return, but generally it not necessary.

```
public RedirectToRouteResult PopulateFoods()
{
   // Redirects to another action method. In this case the index method
   return RedirectToAction("Index");
}
```

In case you want to redirect to another action with parameter - you can use RedirectToAction overload:

```
public ActionResult SomeActionWithParameterFromThisController(string parameterName)
{
   // Some logic
}
.....................
.....................
.....................
return RedirectToAction("SomeActionWithParameterFromThisController", new { parameterName =
parameter });
```

ContentResult

```
public ActionResult Hello()
{
    // Returns a user-defined content type, in this case a string.
    return Content("hello world!");
}
```

Action methods typically return a result that is known as an action result. The ActionResult class is the base class for all action results. The ActionInvoker decide which type of action result to return based on the task that the action method is performing.

It is possible be explicit about what type to return, but generally it not necessary.

```
public ContentResult Hello()
{
    // Returns a user-defined content type, in this case a string.
    return Content("hello world!");
}
```

You can know more about it here: Asp.Net Mvc: ContentResult vs. string

JsonResult

```
public ActionResult LoadPage()
{
    Student result = getFirst();

    //Returns a serialized JSON object.
    return Json(result, JsonRequestBehavior.AllowGet);
}
```

Action methods typically return a result that is known as an action result. The ActionResult class is the base class for all action results. The ActionInvoker decide which type of action result to return based on the task that the action method is performing.

It is possible be explicit about what type to return, but generally it not necessary.

```
public JsonResult LoadPage()
{
    Student result = getFirst();

    //Returns a serialized JSON object.
    return Json(result, JsonRequestBehavior.AllowGet);
}
```

Read ActionResult online: https://riptutorial.com/asp-net-mvc/topic/6487/actionresult

Chapter 5: ActionResult

Syntax

- // ActionResult method returns an instance that derives from ActionResult. You are able to create action method that can return any instance that is wrapped in appropriate ActionResult type.
- //Built-in ActionResult return types are:
- View(); // ViewResult renders a view as a WebPage
- PartialView(); // PartialViewResult renders a partial view, which can be used as a part of another view.
- Redirect(); // RedirectResult redirects to another action method by using its URL.
- RediectToAction(); RedirectToRoute(); // RedirectToRouteResult redirects to another action method.
- Content(); // ContentResult returns a user-defined content-type.
- Json(); // JsonResult returns a serialized JSON object.
- JavaScript(); // JavaScriptResult returns a script that can be executed on client side.
- File(); // FileResult returns a binary output to write to the reponse.
- // EmptResult represents a return value that is used if action method must return a null result.

Examples

Action Methods

When user enters an URL, for example: http://example-website.com/Example/HelloWorld, MVC application will use the routing rules to parse this url and extract the subpath, that will determine the controller, action and possible parameters. For the above url, the result will be /Example/HelloWorld, which by default routing rules results provides the name of the controller: Exmaple and the name of the action: HelloWorld.

```
public class ExampleController: Controller
{
    public ActionResult HelloWorld()
    {
        ViewData["ExampleData"] = "Hello world!";
        return View();
    }
}
```

The above ActionResult method "HelloWorld" will render the view called HelloWorld, where we can then use the data from ViewData.

Mapping Action-Method Parameters

If there would be another value in the URL like: /Example/ProcessInput/2, the routing rules will threat the last number as a parameter passed into the action ProcessInput of controller Example.

```
public ActionResult ProcessInput(int number)
{
    ViewData["OutputMessage"] = string.format("The number you entered is: {0}", number);
    return View();
}
```

Calling An ActionResult In Another ActionResult

We can call an action result in another action result.

```
public ActionResult Action1()
{
    ViewData["OutputMessage"] = "Hello World";
    return RedirectToAction("Action2","ControllerName");
    //this will go to second action;
}


public ActionResult Action2()
{
    return View();
    //this will go to Action2.cshtml as default;
}
```

Read ActionResult online: https://riptutorial.com/asp-net-mvc/topic/6635/actionresult

Chapter 6: Areas

Introduction

What is area?

An area is a smaller unit in MVC application which used as a way to separate large amount of application modules into functional groups. An application can contain multiple areas which stored in Areas folder.

Each area can contain different models, controllers and views depending on requirements. To use an area, it is necessary to register the area name in `RouteConfig` and define route prefix for it.

Remarks

if you want to go to this area through your default controller

```
      return RedirectToAction("Index","Home",new{area="areaname"});
```

Examples

Create a new area

Right click on your project folder/name and create new area and name it.

In mvc internet/empty/basic application a folder with the name of the area will be created,which will contain three different folders named controller , model and views and a class file called

"*areaname*AreaRegistration.cs"

Configure RouteConfig.cs

In your App_start folder open routeconfig.cs and do this

```
  routes.MapRoute(
             name: "Default",
             url: "{controller}/{action}/{id}",
             defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional
 },
             namespaces:new []{"nameofyourproject.Controllers"}// add this line ;
         );
```

Create a new controller and configure areanameAreaRegistration.cs maproute

Create a new controller foreg

ControllerName: "Home", ActionresultName :"Index"

open AreaRegistraion.cs and add the controller name and action name to be rerouted to

```
    context.MapRoute(
            "nameofarea_default",
            "nameofarea/{controller}/{action}/{id}",  // url shown will be like this in
browser
            new {controller="Home", action = "Index", id = UrlParameter.Optional }
        );
```

Read Areas online: https://riptutorial.com/asp-net-mvc/topic/6310/areas

Chapter 7: Asp.net mvc send mail

Examples

Contact Form In Asp MVC

1. Model :

```
public class ContactModel
{
    [Required, Display(Name="Sender Name")]
    public string SenderName { get; set; }
    [Required, Display(Name = "Sender Email"), EmailAddress]
    public string SenderEmail { get; set; }
    [Required]
    public string Message { get; set; }
}
```

2. Controller :

```
public class HomeController
{
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> Contact(ContectModel model)
    {
        if (ModelState.IsValid)
        {
            var mail = new MailMessage();
            mail.To.Add(new MailAddress(model.SenderEmail));
            mail.Subject = "Your Email Subject";
            mail.Body = string.Format("<p>Email From: {0} ({1})</p><p>Message:</p><p>{2}</p>",
model.SenderName, mail.SenderEmail, model.Message);
            mail.IsBodyHtml = true;
            using (var smtp = new SmtpClient())
            {
                await smtp.SendMailAsync(mail);
                return RedirectToAction("SuccessMessage");
            }
        }
        return View(model);
    }

    public ActionResult SuccessMessage()
    {
        return View();
    }

}
```

3. Web.Config :

```
<system.net>
  <mailSettings>
```

```
    <smtp from="you@outlook.com">
      <network host="smtp-mail.outlook.com"
               port="587"
               userName="you@outlook.com"
               password="password"
               enableSsl="true" />
    </smtp>
  </mailSettings>
</system.net>
```

4. View :

Contact.cshtml

```
 @model ContectModel
    @using (Html.BeginForm())
    {
        @Html.AntiForgeryToken()
        <h4>Send your comments.</h4>
        <hr />
        <div class="form-group">
            @Html.LabelFor(m => m.SenderName, new { @class = "col-md-2 control-label" })
            <div class="col-md-10">
                @Html.TextBoxFor(m => m.SenderName, new { @class = "form-control" })
                @Html.ValidationMessageFor(m => m.SenderName)
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(m => m.SenderEmail, new { @class = "col-md-2 control-label" })
            <div class="col-md-10">
                @Html.TextBoxFor(m => m.SenderEmail, new { @class = "form-control" })
                @Html.ValidationMessageFor(m => m.SenderEmail)
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(m => m.Message, new { @class = "col-md-2 control-label" })
            <div class="col-md-10">
                @Html.TextAreaFor(m => m.Message, new { @class = "form-control" })
                @Html.ValidationMessageFor(m => m.Message)
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" class="btn btn-default" value="Send" />
            </div>
        </div>
    }
```

SuccessMessage.cshtml

```
<h2>Your message has been sent</h2>
```

Sending Email From Class

This way can be so helpfull, but, some people (like me) are afreak of repeat code, and like you are showin us, it means that I need to create a contact controller with the same code on each proyect that we have, so, I thing that this can be helpfull too

This is my class, that can be on a DLL or whatever

```
 public class Emails
    {
        public static void SendHtmlEmail(string receiverEmail, string subject, string body,
bool Ssl = false)
        {
            //Those are read it from webconfig or appconfig
            var client = new SmtpClient(ConfigurationManager.AppSettings["MailServer"],
Convert.ToInt16

                (ConfigurationManager.AppSettings["MailPort"]))
            {
                Credentials = new
NetworkCredential(ConfigurationManager.AppSettings["MailSender"],
ConfigurationManager.AppSettings["MailSenderPassword"]),
                EnableSsl = Ssl
            };

            MailMessage message = new MailMessage();
            message.From = new MailAddress(ConfigurationManager.AppSettings["MailSender"]);
            message.To.Add(receiverEmail);
            // message.To.Add("sgermosen@praysoft.net");
            message.Subject = subject;
            message.IsBodyHtml = true;
            message.Body = body;
            client.Send(message);
        }

    }
```

like you see it will read from the webconfig, so, we need to configured it, this configuration is for Gmail, but, every host have their own configuration

```
 <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
    <add key="AdminUser" value="sgrysoft@gmail.com" />
    <add key="AdminPassWord" value="123456789" />
    <add key="SMTPName" value="smtp.gmail.com" />
    <add key="SMTPPort" value="587" />

  </appSettings>
```

Read Asp.net mvc send mail online: https://riptutorial.com/asp-net-mvc/topic/9736/asp-net--mvc-send-mail

Chapter 8: Automatic client-side validation from attributes

Remarks

By default, Safari does not enforce HTML5 element validation. You need to override this manually using other means.

Examples

Model

```
public class UserModel
{

        [Required]
        [StringLength(6, MinimumLength = 3)]
        [RegularExpression(@"(\S)+", ErrorMessage = "White space is not allowed")]
        public string UserName { get; set; }

        [Required]
        [StringLength(8, MinimumLength = 3)]
        public string FirstName { get; set; }

        [Required]
        [StringLength(9, MinimumLength = 2)]
        public string LastName { get; set; }

        [Required]
        public string City { get; set; }

}
```

web.config settings

```
<appSettings>
  <add key="ClientValidationEnabled" value="true"/>
  <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
 </appSettings>
```

Required Nuget Packages

```
<package id="jQuery" version="1.10.2" targetFramework="net452" />
<package id="jQuery.Validation" version="1.11.1" targetFramework="net452" />
<package id="Microsoft.jQuery.Unobtrusive.Validation" version="3.2.3" targetFramework="net452"
/>
```

Form View

```
@model WebApplication4.Models.UserModel
@{
    ViewBag.Title = "Register";
}

<h2>@ViewBag.Title.</h2>

@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-
horizontal", role = "form" }))
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.FirstName, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.FirstName, new { @class = "form-control" })
            @Html.ValidationMessageFor(m=>m.FirstName)
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.LastName, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.LastName, new { @class = "form-control" })
            @Html.ValidationMessageFor(m => m.LastName)
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.UserName, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.UserName, new { @class = "form-control" })
            @Html.ValidationMessageFor(m => m.UserName)
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Bundle configuration

```
  public class BundleConfig
    {

        public static void RegisterBundles(BundleCollection bundles)
        {
            bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                        "~/Scripts/jquery-{version}.js"));

            bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(
                        "~/Scripts/jquery.validate*"));
        }
    }
```

Global.asax.cs

```
public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            // Need to include your bundles
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
```

Read Automatic client-side validation from attributes online: https://riptutorial.com/asp-net-mvc/topic/6044/automatic-client-side-validation-from-attributes

Chapter 9: Bundling and Minification

Examples

Minification

The minification is used to reduce the size of CSS and Javascript files to speed up download times. This process is done by removing all of the unnecessary white-space, comments, and any other non-essential content from the files.

This process is done automatically when using a `ScriptBundle` or a `StyleBundle` object. If you need to disable it, you have to use a basic `Bundle` object.

Example using Minification

The following code uses preprocessor directives to apply bundling only during releases in order to allow for easier debugging during non-releases (as non-bundled files are typically easier to navigate through) :

```
public static void RegisterBundles(BundleCollection bundles)
{
    #if DEBUG
        bundles.Add(new Bundle("~/bundles/jquery").Include("~/Scripts/jquery-{version}.js"));
        bundles.Add(new Bundle("~/Content/css").Include("~/Content/site.css"));
    #else
        bundles.Add(new ScriptBundle("~/bundles/jquery").Include("~/Scripts/jquery-
{version}.js"));
        bundles.Add(new StyleBundle("~/Content/css").Include("~/Content/site.css"));
    #endif
}
```

Script and Style Bundles

The following is the default code snippet for the BundleConfig.cs file.

```
using System.Web.Optimization;

public class BundleConfig
{
    // For more information on Bundling, visit http://go.microsoft.com/fwlink/?LinkId=254725
    public static void RegisterBundles(BundleCollection bundles)
    {
        bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                    "~/Scripts/jquery-{version}.js"));

    // Use the development version of Modernizr to develop with and learn from. Then, when
you're
    // ready for production, use the build tool at http://modernizr.com to pick only the tests
you need.
    bundles.Add(new ScriptBundle("~/bundles/modernizr").Include(
                "~/Scripts/modernizr-*"));

    bundles.Add(new StyleBundle("~/Content/css").Include("~/Content/site.css"));

    bundles.Add(new StyleBundle("~/Content/themes/base/css").Include(
                "~/Content/themes/base/jquery.ui.core.css",
                "~/Content/themes/base/jquery.ui.resizable.css",
    }
 }
```

Bundles are registered in the Global.asax file inside the Application_Start() method:

```
using System.Web.Optimization;

protected void Application_Start()
{
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}
```

Bundles should be rendered in your Views as so:

```
@using System.Web.Optimization

@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/modernizr")
@Styles.Render("~/Content/css")
@Styles.Render("~/Content/themes/base/css")
```

Note that bundling does not occur when you are in development mode (where the compilation Element in the Web.config file is set to debug="true"). Instead, the Render statements in your Views will include each individual file in a non-bundled, non-minified format, for ease of debugging.

Once the application is in production mode (where the compilation Element in the Web.config file is set to debug="false"), bundling will take place.

This can lead to complications for scripts that reference relative paths of other files, such as references to Twitter Bootstrap's icon files. This can be addressed by using System.Web.Optimization's CssRewriteUrlTransform class:

```
 bundles.Add(new StyleBundle("~/bundles/css").Include(
                "~/Content/css/*.css", new CssRewriteUrlTransform()));
```

The CssRewriteUrlTransform class will rewrite relative Urls within the bundled files to absolute paths, so that the references will remain intact after the calling reference is moved to the location of the bundle (e.g. using the above code, moving from "~/Content/css/bootstrap.css" to "~/bundles/css/bootstrap.css").

Read Bundling and Minification online: https://riptutorial.com/asp-net-mvc/topic/1959/bundling-and-minification

Chapter 10: CRUD operation

Introduction

CRUD Operation refers to classic (create, read, update, delete) operations as it pertains to data.

In ASP MVC context there are several ways to CRUD your data using Models and subsequently views, Controllers.

One simple way is to make use of the scaffolding feature provided by the Visual studio templates and customize to your needs.

Please keep in mind that CRUD is very broadly defined and it has many variations to suit your requirements. For e.g. Database first, Entity first etc.

Remarks

For simplicity sake, this CRUD operation uses a entity framework context in the controller. It is not a good practice, but it is beyond this topic's scope. Click in entity framework if you want to learn more about it.

Examples

Create - Controller Part

To implement the create functionality we need two actions: *GET* and *POST*.

1. The *GET* action used to return view which will show a form allowing user to input data using HTML elements. If there are some default values to be inserted before user adding any data, it should be assigned to the view model properties on this action.

2. When the user fills the form and clicks the "Save" button we will be dealing with data from the form. Because of that now we need the *POST* action. This method will be responsible for managing data and saving it to database. In case of any errors, the same view returned with stored form data & error message explains what problem occurs after submit action.

We'll implement these two steps within two Create() methods within our controller class.

```
    // GET: Student/Create
    // When the user access this the link ~/Student/Create a get request is made to controller
Student and action Create, as the page just need to build a blank form, any information is
needed to be passed to view builder
    public ActionResult Create()
    {
        // Creates a ViewResult object that renders a view to the response.
        // no parameters means: view = default in this case Create and model = null
        return View();
    }

    // POST: Student/Create
    [HttpPost]
    // Used to protect from overposting attacks, see
http://stackoverflow.com/documentation/asp.net-mvc/1997/html-antiforgerytoke for details
    [ValidateAntiForgeryToken]
    // This is the post request with forms data that will be bind the action, if in the data
post request have enough information to build a Student instance that will be bind
    public ActionResult Create(Student student)
    {
        try
        {
        //Gets a value that indicates whether this instance received from the view is valid.
            if (ModelState.IsValid)
            {
                // Adds to the context
                db.Students.Add(student);
                // Persist the data
                db.SaveChanges();
                // Returns an HTTP 302 response to the browser, which causes the browser to
make a GET request to the specified action, in this case the index action.
                return RedirectToAction("Index");
            }
        }
        catch
        {
            // Log the error (uncomment dex variable name and add a line here to write a log).
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the
problem persists see your system administrator.");
        }
        // view = default in this case Create and model = student
        return View(student);
    }
```

Create - View Part

```
@model ContosoUniversity.Models.Student

//The Html.BeginForm helper Writes an opening <form> tag to the response. When the user
submits the form, the request will be processed by an action method.
@using (Html.BeginForm())
{
    //Generates a hidden form field (anti-forgery token) that is validated when the form is
submitted.
    @Html.AntiForgeryToken()

<div class="form-horizontal">
    <h4>Student</h4>
    <hr />

    //Returns an unordered list (ul element) of validation messages that are in the
ModelStateDictionary object.
    @Html.ValidationSummary(true, "", new { @class = "text-danger" })

    <div class="form-group">
        //Returns an HTML label element and the property name of the property that is
represented by the specified expression.
        @Html.LabelFor(model => model.LastName, htmlAttributes: new { @class = "control-label
col-md-2" })

        <div class="col-md-10">
            //Returns an HTML input element for each property in the object that is
represented by the Expression expression.
            @Html.EditorFor(model => model.LastName, new { htmlAttributes = new { @class =
"form-control" } })

            //Returns the HTML markup for a validation-error message for each data field that
is represented by the specified expression.
            @Html.ValidationMessageFor(model => model.LastName, "", new { @class = "text-
danger" })
        </div>
    </div>

    <div class="form-group">
        @Html.LabelFor(model => model.FirstMidName, htmlAttributes: new { @class = "control-
label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.FirstMidName, new { htmlAttributes = new { @class =
"form-control" } })
            @Html.ValidationMessageFor(model => model.FirstMidName, "", new { @class = "text-
danger" })
        </div>
    </div>

    <div class="form-group">
        @Html.LabelFor(model => model.EnrollmentDate, htmlAttributes: new { @class = "control-
label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.EnrollmentDate, new { htmlAttributes = new { @class
= "form-control" } })
            @Html.ValidationMessageFor(model => model.EnrollmentDate, "", new { @class =
"text-danger" })
        </div>
    </div>

    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Create" class="btn btn-default" />
        </div>
    </div>
</div>
}

<div>
    //Returns an anchor element (a element) the text is Back to List and action is Index
    @Html.ActionLink("Back to List", "Index")
</div>
```

Details - Controller part

By the url `~/Student/Details/5` being: (~: site root, Student: Controller, Details: Action, 5: student id), it is possible to retrieve the student by its id.

```
// GET: Student/Details/5
    public ActionResult Details(int? id)
    {
        // it good practice to consider that things could go wrong so,it is wise to have a
validation in the controller
        if (id == null)
        {
            // return a bad request
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        Student student = db.Students.Find(id);
        if (student == null)
        {
            // if doesn't found return 404
            return HttpNotFound();
        }
        return View(student);
    }
```

Details - View part

```
// Model is the class that contains the student data send by the controller and will be
rendered in the view
@model ContosoUniversity.Models.Student

<h2>Details</h2>

<div>
   <h4>Student</h4>
<hr />
<dl class="dl-horizontal">
    <dt>
        //Gets the display name for the model.
        @Html.DisplayNameFor(model => model.LastName)
    </dt>

    <dd>
        //Returns HTML markup for each property in the object that is represented by the
Expression expression.
        @Html.DisplayFor(model => model.LastName)
    </dd>

    <dt>
        @Html.DisplayNameFor(model => model.FirstMidName)
    </dt>

    <dd>
        @Html.DisplayFor(model => model.FirstMidName)
    </dd>

    <dt>
        @Html.DisplayNameFor(model => model.EnrollmentDate)
    </dt>

    <dd>
        @Html.DisplayFor(model => model.EnrollmentDate)
    </dd>
    <dt>
        @Html.DisplayNameFor(model => model.Enrollments)
    </dt>
    <dd>
        <table class="table">
            <tr>
```

```
                <th>Course Title</th>
                <th>Grade</th>
            </tr>
            @foreach (var item in Model.Enrollments)
            {
                <tr>
                    <td>
                        @Html.DisplayFor(modelItem => item.Course.Title)
                    </td>
                    <td>
                        @Html.DisplayFor(modelItem => item.Grade)
                    </td>
                </tr>
            }
        </table>
    </dd>
</dl>
</div>
<p>
    //Returns an anchor element (a element) the text is Edit, action is Edit and the route
value is the model ID property.
    @Html.ActionLink("Edit", "Edit", new { id = Model.ID }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

Edit - Controller part

```
 // GET: Student/Edit/5
 // It is receives a get http request for the controller Student and Action Edit with the id
of 5
    public ActionResult Edit(int? id)
    {
         // it good practice to consider that things could go wrong so,it is wise to have a
validation in the controller
        if (id == null)
        {
            // returns a bad request
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }

        // It finds the Student to be edited.
        Student student = db.Students.Find(id);
        if (student == null)
        {
            // if doesn't found returns 404
            return HttpNotFound();
        }
        // Returns the Student data to fill out the edit form values.
        return View(student);
    }
```

This method is very similar to the details action method, which is a good candidate to a refactoring, but it out of scope of this topic.

```
    // POST: Student/Edit/5
    [HttpPost]

    //used to To protect from overposting attacks more details see
```

```
http://stackoverflow.com/documentation/asp.net-mvc/1997/html-antiforgerytoke
    [ValidateAntiForgeryToken]

    //Represents an attribute that is used for the name of an action.
    [ActionName("Edit")]
    public ActionResult Edit(Student student)
    {
        try
        {
            //Gets a value that indicates whether this instance received from the view is
valid.
            if (ModelState.IsValid)
            {
                // Two thing happens here:
                // 1) db.Entry(student) -> Gets a DbEntityEntry object for the student entity
providing access to information about it and the ability to perform actions on the entity.
                // 2) Set the student state to modified, that means that the student entity is
being tracked by the context and exists in the database, and some or all of its property
values have been modified.
                db.Entry(student).State = EntityState.Modified;

                // Now just save the changes that all the changes made in the form will be
persisted.
                db.SaveChanges();

                // Returns an HTTP 302 response to the browser, which causes the browser to
make a GET request to the specified action, in this case the index action.
                return RedirectToAction("Index");
            }
        }
        catch
        {
            //Log the error add a line here to write a log.
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the
problem persists, see your system administrator.");
        }

        // return the invalid student instance to be corrected.
        return View(student);
    }
```

Delete - Controller part

Is good practice to resist the temptation of doing the delete action in the get request. It would be a huge security error, it has to be done always in the post method.

```
    // GET: Student/Delete/5
    public ActionResult Delete(int? id)
    {
        // it good practice to consider that things could go wrong so,it is wise to have a
validation in the controller
        if (id == null)
        {
            // returns a bad request
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }

        // It finds the Student to be deleted.
        Student student = db.Students.Find(id);
        if (student == null)
        {
            // if doesn't found returns 404
            return HttpNotFound();
        }
        // Returns the Student data to show the details of what will be deleted.
        return View(student);
    }

    // POST: Student/Delete/5
    [HttpPost]

    //Represents an attribute that is used for the name of an action.
    [ActionName("Delete")]

    //used to To protect from overposting attacks more details see
http://stackoverflow.com/documentation/asp.net-mvc/1997/html-antiforgerytoke
    [ValidateAntiForgeryToken]
    public ActionResult Delete(int id)
    {
        try
        {
            // Finds the student
            Student student = db.Students.Find(id);

            // Try to remove it
            db.Students.Remove(student);

            // Save the changes
            db.SaveChanges();
        }
        catch
        {
            //Log the error add a line here to write a log.
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the
problem persists, see your system administrator.");
        }

        // Returns an HTTP 302 response to the browser, which causes the browser to make a GET
request to the specified action, in this case the index action.
        return RedirectToAction("Index");
    }
```

Read CRUD operation online: https://riptutorial.com/asp-net-mvc/topic/6380/crud-operation

Chapter 11: Data annotations

Introduction

We can add validations to our application by adding Data Annotations to our model classes. Data Annotations allow us to describe the rules we want applied to our model properties, and ASP.NET MVC will take care of enforcing them and displaying appropriate messages to users.

Examples

Basic validation attributes used in ViewModel

Model

```
using System.ComponentModel.DataAnnotations;

public class ViewModel
{
    [Required(ErrorMessage="Name is required")]
    public string Name { get; set; }

    [StringLength(14, MinimumLength = 14, ErrorMessage = "Invalid Phone Number")]
    [Required(ErrorMessage="Phone Number is required")]
    public string PhoneNo { get; set; }

    [Range(typeof(decimal), "0", "150")]
    public decimal? Age { get; set; }

    [RegularExpression(@"^\d{5}(-\d{4})?$", ErrorMessage = "Invalid Zip Code.")]
    public string ZipCode {get;set;}

    [EmailAddress(ErrorMessage = "Invalid Email Address")]
    public string Email { get; set; }

    [Editable(false)]
    public string Address{ get; set; }
}
```

View

```
// Include Jquery and Unobstructive Js here for client side validation

@using (Html.BeginForm("Index","Home") {

    @Html.TextBoxFor(model => model.Name)
    @Html.ValidationMessageFor(model => model.Name)

    @Html.TextBoxFor(model => model.PhoneNo)
    @Html.ValidationMessageFor(model => model.PhoneNo)
    @Html.TextBoxFor(model => model.Age)
    @Html.ValidationMessageFor(model => model.Age)

    @Html.TextBoxFor(model => model.ZipCode)
    @Html.ValidationMessageFor(model => model.ZipCode)

    @Html.TextBoxFor(model => model.Email)
    @Html.ValidationMessageFor(model => model.Email)

    @Html.TextBoxFor(model => model.Address)
    @Html.ValidationMessageFor(model => model.Address)

    <input type="submit" value="submit" />
}
```

Controller

```
public ActionResult Index(ViewModel _Model)
{
    // Checking whether the Form posted is valid one.
    if(ModelState.IsValid)
    {
        // your model is valid here.
        // perform any actions you need to, like database actions,
        // and/or redirecting to other controllers and actions.
    }
    else
    {
        // redirect to same action
        return View(_Model);
    }
}
```

Remote validation

Remote Validation used to check whether the content enter in the input control is valid or not by sending an ajax request to server side to check it.

Working

The `RemoteAttribute` works by making an AJAX call from the client to a controller action with the value of the field being validated. The controller action then returns a `JsonResult` response indicating validation success or failure. Returning `true` from your action indicates that validation passed. Any other value indicates failure. If you return `false`, the error message specified in the attribute is used. If you return anything else such as a string or even an integer, it will be displayed as the error message. Unless you need your error message to be dynamic, it makes sense to return true or false and let the validator use the error message specified on the attribute.

ViewModel

```
public class ViewModel
{
    [Remote("IsEmailAvailable", "Group", HttpMethod = "POST", ErrorMessage = "Email already
exists. Please enter a different email address.")]
    public string Email{ get; set; }
}
```

Controller

```
[HttpPost]
public JsonResult IsEmailAvailable(string Email)
{
    // Logic to check whether email is already registered or Not.
    var emailExists = IsEmailRegistered();
    return Json(!emailExists);
}
```

Live Demo Fiddle

You can pass additional properties of the model to the controller method using the `AdditionalFields` property of `RemoteAttribute`. A typical scenario would be to pass the ID property of the model in an 'Edit' form, so that the controller logic can ignore values for the existing record.

Model

```
  public int? ID { get; set; }
  [Display(Name = "Email address")]
  [DataType(DataType.EmailAddress)]
  [Required(ErrorMessage = "Please enter you email address")]
  [Remote("IsEmailAvailable", HttpMethod="Post", AdditionalFields="ID", ErrorMessage = "Email
already exists. Please enter a different email address.")]
  public string Email { get; set; }
```

Controller

```
[HttpPost]
public ActionResult Validate(string email, int? id)
{
    if (id.HasValue)
    {
        return Json(!db.Users.Any(x => x.Email == email && x.ID != id);
    }
    else
    {
        return Json(!db.Users.Any(x => x.Email == email);
    }
}
```

Working Demo - Additional Fields

Additional Note

The default error message is understandably vague, so always remember to override the default error message when using the `RemoteAttribute`.

RequiredAttribute

The `Required` attribute specifies that a property is required. An error message can be specified on using the `ErrorMessage` property on the attribute.

First add the namespace:

```
using System.ComponentModel.DataAnnotations;
```

And apply the attribute on a property.

```
public class Product
{
   [Required(ErrorMessage = "The product name is required.")]
   public string Name { get; set; }

   [Required(ErrorMessage = "The product description is required.")]
   public string Description { get; set; }
}
```

It is also possible to use resources in the error message for globalized applications. In this case, the `ErrorMessageResourceName` must be specified with the resource key of the resource class (`resx` file) that must be setted on the `ErrorMessageResourceType`:

```
public class Product
{
   [Required(ErrorMessageResourceName = "ProductNameRequired",
             ErrorMessageResourceType = typeof(ResourceClass))]
   public string Name { get; set; }

   [Required(ErrorMessageResourceName = "ProductDescriptionRequired",
             ErrorMessageResourceType = typeof(ResourceClass))]
   public string Description { get; set; }
}
```

StringLengthAttribute

The `StringLength` attribute specifies the minimum and maximum length of characters that are allowed in a data field. This attribute can be applied on properties, public fields and parameters. The error message must be specified on the `ErrorMessage` property on the attribute. The properties `MinimumLength` and `MaximumLength` specifies the minimum and maximum respectively.

First add the namespace:

```
using System.ComponentModel.DataAnnotations;
```

And apply the attribute on a property.

```
public class User
{
   // set the maximum
```

```
    [StringLength(20, ErrorMessage = "The username cannot exceed 20 characters. ")]
    public string Username { get; set; }

    [StringLength(MinimumLength = 3, MaximumLength = 16, ErrorMessage = "The password must have
between 3 and 16 characters.")]
    public string Password { get; set; }
}
```

It is also possible to use resources in the error message for globalized applications. In this case, the `ErrorMessageResourceName` must be specified with the resource key of the resource class (`resx` file) that must be setted on the `ErrorMessageResourceType`:

```
public class User
{
    [StringLength(20, ErrorMessageResourceName = "StringLength",
                              ErrorMessageResourceType = typeof(ResoucesKeys))]
    public string Username { get; set; }

    [StringLength(MinimumLength = 3,
                        MaximumLength = 16,
                        ErrorMessageResourceName = "StringLength",
                        ErrorMessageResourceType = typeof(ResoucesKeys))]
    public string Password { get; set; }
}
```

Range Attribute

The `Range` attribute can decorate any properties or public fields and specifies a range that a numerical field must fall between to be considered valid.

```
[Range(minimumValue, maximumValue)]
public int Property { get; set; }
```

Additionally, it accepts an optional `ErrorMessage` property that can be used to set the message received by the user when invalid data is entered :

```
[Range(minimumValue, maximumValue, ErrorMessage = "{your-error-message}")]
public int Property { get; set; }
```

Example

```
[Range(1,100, ErrorMessage = "Ranking must be between 1 and 100.")]
public int Ranking { get; set; }
```

RegularExpression Attribute

The `[RegularExpression]` attribute can decorate any properties or public fields and specifies a regular expression that must be matched for the property be considered valid.

```
[RegularExpression(validationExpression)]
public string Property { get; set; }
```

Additionally, it accepts an optional `ErrorMessage` property that can be used to set the message received by the user when invalid data is entered :

```
[RegularExpression(validationExpression, ErrorMessage = "{your-error-message}")]
public string Property { get; set; }
```

Example(s)

```
[RegularExpression(@"^[a-z]{8,16}?$", ErrorMessage = "A User Name must consist of 8-16
lowercase letters")]
public string UserName{ get; set; }
[RegularExpression(@"^\d{5}(-\d{4})?$", ErrorMessage = "Please enter a valid ZIP Code (e.g.
12345, 12345-1234)")]
public string ZipCode { get; set; }
```

Compare Attribute

The `Compare` attribute compares two properties of a model.

The error message can be specified using property `ErrorMessage`, or using resource files.

To use `Compare` attribute include `using` for the following namespace:

```
using System.ComponentModel.DataAnnotations;
```

Then you can use the attribute in your model:

```
public class RegisterModel
{
    public string Email { get; set; }

    [Compare("Email",  ErrorMessage = "The Email and Confirm Email fields do not match.")]
    public string ConfirmEmail { get; set; }
}
```

When this model is validates, if `Email` and `ConfirmEmail` have different values, validation will fail.

Localized error messages

Just like with all validation attributes, it is possible to use error messages from resource files. In this sample the error message will be loaded from resource file `Resources`, resource name is `CompareValidationMessage`:

```
public class RegisterModel
{
    public string Email { get; set; }

    ["Email", ErrorMessageResourceType = typeof(Resources),  ErrorMessageResourceName =
"CompareValidationMessage")]
    public string ConfirmEmail { get; set; }
}
```

Avoid strings in property names

To avoid using string for property value, in C# 6+ you can use `nameof` keyword:

```
public class RegisterModel
{
    public string Email { get; set; }

    [Compare(nameof(Email),  ErrorMessage = "The Email and Confirm Email fields do not
match.")]
    public string ConfirmEmail { get; set; }
}
```

Placeholders in error messages

You can use placeholders in your error messages. Placeholder `{0}` is replaced with the display name of current property and `{1}` is replaced with display name of related property:

```
public class RegisterModel
{
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Display(Name = "Confirm Email")]
    [Compare("Email",  ErrorMessage = "The '{1}' and '{0}' fields do not match.")]
    public string ConfirmEmail { get; set; }
}
```

If validation of the model fails, the error message will be

> The 'Email' and 'Confirm Email' fields do not match.

Custom Validation Attribute

When it comes to validate some rules which are not generic data validation e.g ensuring a field is required or some range of values but they are specific to your business logic then you can create your own ***Custom Validator***. To create a custom validation attribute, you just need to `inherit` **`ValidationAttribute`** class and `override` its **`IsValid`** method. The **`IsValid`** method takes two parameters, the first is an `object` named as `value` and the second is a `ValidationContext object` named as `validationContext`. **`Value`** refers to the actual value from the field that your custom validator is going to validate.

Suppose you want to validate `Email` through `Custom Validator`

```
public class MyCustomValidator : ValidationAttribute
{
    private static string myEmail= "admin@dotnetfiddle.net";

    protected override ValidationResult IsValid(object value, ValidationContext
validationContext)
    {
        string Email = value.ToString();
        if(myEmail.Equals(Email))
```

```
            return new ValidationResult("Email Already Exist");
        return ValidationResult.Success;
    }
}

public class SampleViewModel
{
    [MyCustomValidator]
    [Required]
    public string Email { get; set; }

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

Here is its DotNetFiddle Demo

EDMx model - Data Annotation

Edmx model internel

```
public partial class ItemRequest
{
    public int RequestId { get; set; }
    //...
}
```

Adding data annotation to this - if we modify this model directly, when a update to the model is made, the changes are lost . so

To add a attribute in this case 'Required'

Create a new class - any name Then

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

//make sure the namespace is equal to the other partial class ItemRequest
namespace MvcApplication1.Models
{
    [MetadataType(typeof(ItemRequestMetaData))]
    public partial class ItemRequest
    {
    }

    public class ItemRequestMetaData
    {
        [Required]
        public int RequestId {get;set;}

        //...
    }
}
```

or

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace YourApplication.Models
{
    public interface IEntityMetadata
    {
        [Required]
        Int32 Id { get; set; }
    }

    [MetadataType(typeof(IEntityMetadata))]
    public partial class Entity : IEntityMetadata
    {
        /* Id property has already existed in the mapped class */
    }
}
```

Data annotations for Database first implementation (model code auto-generated)

```
[MetadataType(typeof(RoleMetaData))]
public partial class ROLE
{
}

public class RoleMetaData
{
    [Display(Name = "Role")]
    public string ROLE_DESCRIPTION { get; set; }

    [Display(Name = "Username")]
    public string ROLE_USERNAME { get; set; }
}
```

If you used database-first and your model code was auto-generated, this message will appear above your model code:

> This code was generated from a template. Manual changes to this file may cause unexpected behavior in your application. Manual changes to this file will be overwritten if the code is regenerated

If you want to use data-annotations and you don't want them to be over-written if you refresh the edmx just add another a partial class to your model folder that looks like the example above.

Read Data annotations online: https://riptutorial.com/asp-net-mvc/topic/1961/data-annotations

Chapter 12: Dependency Injection

Remarks

The whole point of Dependency Injection (DI) is to reduce code coupling. Imagine any kind if interaction which involves newing up something like in the "Hard coded dependency example".

A big part of writing code is the ability to test it. Every time we new up a new dependency, we make our code difficult to test because we have no control over that dependency.

How would you test code which depends on DataTime.Now for example? It always changes so you have no reference. This is when you inject a stable parameter as your starting point. You can control it, you can write tests based on various values and make sure you always get the right result.

A good option therefore is to pass an interface or an abstract class as a parameter in the constructor DI.

An interface represents a well defined contract, you can always rely on the methods to be there and you can always rely on the method signatures.

Once you start using DI other aspects will open up. For example, even if you pass an interface at some point you will need a real implementation to actually do any work. This is where other concepts appear. We can use IOC (Inversion of Control) to resolve our dependencies. This means that we instruct our code to always use a specific implementation for any contract. Of course there are other ways of doing this. We could always instantiate each contract with a specific implementation and from that point onwards our code can use that part :

```
public ILogging Logging { get; set }
```

at some point we initialise it.

```
Logging = new FileLogging();
```

this will always work as long as our class fulfils the expected contract :

```
public class FileLogging : ILogging
```

from the initialise moment onwards we always use the Logging object. This makes lif easier because if we ever decide to change and use a DatabaseLogging for example, we only have to change the code in one place and this is exactly where we initialise the Logging class.

Is DI only good for testing? No, DI is important when writing maintainable code as well. It allows the separation of concerns to be clear.

When you write any code, think ... is it testable, can I write a test, that's when injecting a DateTime value instead of using DateTime.Now makes sense.

Examples

Ninject Configurations

After the install of an IoC (Inversion of Control) container, some tweaks are needed to make it work. In this case, I'll use Ninject. In the NinjectWebCommon file, that is located in the App_Start folder, substitute the CreateKernel method with:

```
private static IKernel CreateKernel()
    {
        // Create the kernel with the interface to concrete bindings
        var kernel = RegisterServices();
        try
        {
            kernel.Bind<Func<IKernel>>().ToMethod(ctx => () => new Bootstrapper().Kernel);
            kernel.Bind<IHttpModule>().To<HttpApplicationInitializationHttpModule>();

            return kernel;
        }
        catch
        {
            kernel.Dispose();
            throw;
        }
    }
```

And the RegisterServices method with:

```
 private static StandardKernel RegisterServices()
        {
            Container container = new Container();
            // encapsulate the interface to concrete bindings in another class or even in
another layer
            StandardKernel kernel = container.GetServices();
            return kernel;
        }
```

Create a new class to to the binding that in this case is called Container:

```
public class Container
{
    public StandardKernel GetServices()
    {
        // It is good practice to create a derived class of NinjectModule to organize the
binding by concerns. In this case one for the repository, one for service and one for app
service bindings
        return new StandardKernel(new NinjectRepositoryModule(),
            new NinjectServiceModule(),
            new NinjectAppServiceModule());
    }
}
```

Finally in each derived NinjectModule class modify the bindings overloading the Load method like:

```
public class NinjectRepositoryModule: NinjectModule
{
    public override void Load()
    {
        // When we need a generic IRepositoryBase<> to bind to a generic RepositoryBase<>
        // The typeof keyword is used because the target method is generic
        Bind(typeof (IRepositoryBase<>)).To(typeof (RepositoryBase<>));

        // When we need a IUnitOfWorkbind to UnitOfWork concrete class that is a singleton
        Bind<IUnitOfWork>().To<UnitOfWork>().InSingletonScope();
    }
}
```

Another example of derived NinjectModule:

```
public class NinjectServiceModule :NinjectModule
{
    public override void Load()
    {
        // When we need a IBenefitService to BenefitService concrete class
        Bind<IBenefitService>().To<BenefitService>();

        // When we need a ICategoryService to CategoryService concrete class
        Bind<ICategoryService>().To<CategoryService>();

        // When we need a IConditionService to ConditionService concrete class
        Bind<IConditionService>().To<ConditionService>();
    }
}
```

Utilization of the interfaces

In the concrete class that need the service, use the interface to access the service instead of its implementation like:

```
 public class BenefitAppService
{
    private readonly IBenefitService _service;
    public BenefitAppService(IBenefitService service)
    {
        _service = service;
    }

    public void Update(Benefit benefit)
    {
        if (benefit == null) return
        _service.Update(benefit);
        _service.Complete();
    }
}
```

Now if you need something in the concrete class, won't interfere in the the code above. You may change the service implementation for another completely difference, and as long its satisfies the interface you are good to go. Also it makes it very easy to test it.

Constructor dependency injection

The Constructor Dependency Injection requires parameters in the constructor to inject dependencies. So you have to pass the values when you create a new object.

```
public class Example
{
    private readonly ILogging _logging;

    public Example(ILogging logging)
    {
        this._logging = logging;
    }
}
```

Hard coded dependency

```
public class Example
{
    private FileLogging _logging;

    public Example()
    {
        this._logging = new FileLogging();
    }
}
```

parameter DI

```
public DateTime SomeCalculation()
{
     return DateTime.Now.AddDays(3);
}
```

vs

```
public DateTime SomeCalculation(DateTime inputDate)
{
    return inputDate.AddDays(3);
}
```

Ninject Dependency Injection

Dependency resolver is used to avoid tightly-coupled classes, improve flexibility and make testing easy. You can create your own dependency injector (not recomended) or use one of well-written and tested dependency injectors. In this example I am going to use Ninject.

Step one: Create dependency resolver.

First of all, download *Ninject* from NuGet. Create folder named *Infrastructure* and add class named *NinjectDependencyResolver*:

```
using Ninject;
using System;
using System.Collections.Generic;
using System.Web.Mvc;

public class NinjectDependencyResolver
    : IDependencyResolver
{
    private IKernel kernel;

    public NinjectDependencyResolver()
    {
        // Initialize kernel and add bindings
        kernel = new StandardKernel();
        AddBindings();
    }

    public object GetService(Type serviceType)
    {
        return kernel.TryGet(serviceType);
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return kernel.GetAll(serviceType);
    }

    private void AddBindings()
    {
        // Bindings added here
    }
}
```

The MVC Framework will call the *GetService* and *GetServices* methods when it needs an insance of a class to service an incoming request.

Step two: Register dependency resolver.

Now we have our custom dependency resolver and we need to register it in order to tell MVC framework to use our dependency resolver. Register dependency resolver in *Global.asax.cs* file:

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    DependencyResolver.SetResolver(new NinjectDependencyResolver());

    // .....
}
```

Step three: Add bindings.

Imagine that we have following interface and implentation:

```
public interface ICustomCache
{
    string Info { get; }
}
```

```
public class CustomCache : ICustomCache
{
    public string Info
    {
        get
        {
            return "Hello from CustomCache.";
        }
    }
}
```

If we want to use CustomCache in our controller without tightly-coupling our controller with CustomCache, then we need to bind *ICustomCache to CustomCache* and inject it using Ninject. First things first, bind *ICustomCache to CustomCache* by adding following code to *AddBindings()* method of *NinjectDependencyResolver*:

```
private void AddBindings()
{
    // Bindings added here
    kernel.Bind<ICustomCache>().To<CustomCache>();
}
```

Then prepare your controller for injection as below:

```
public class HomeController : Controller
{
    private ICustomCache CustomCache { get; set; }

    public HomeController(ICustomCache customCacheParam)
    {
        if (customCacheParam == null)
            throw new ArgumentNullException(nameof(customCacheParam));

        CustomCache = customCacheParam;
    }

    public ActionResult Index()
    {
        // cacheInfo: "Hello from CustomCache."
        string cacheInfo = CustomCache.Info;

        return View();
    }
}
```

This is example of *costructor injection* and it is *one form of dependency injection*. As you see, our Home controller does not depend on CustomCache class itslef. If we want to use another inplementation of ICustomCache in our application, the only thing we need to change is to binding *ICustomCache* to another implentation and that is the only step we need to take. What happened here is, MVC Framework asked our *registered dependency resolver* to create instance of *HomeController* class via *GetService* method. GetService method ask Ninject kernel to create requested object and Ninject kernel examines the type in its term and finds out that constructor of HomeController requeires an *ICustomCache* and binding has already been added for

ICustomCache. Ninject creates instance of *binded class*, uses it to create *HomeController* and returns it MVC Framework.

Dependency chains.

When Ninject tries to create type, it examines other depenencies between type and other types and if there is any Ninject tries to create them also. For example, if our CustomCache class requires ICacheKeyProvider and if bining added for ICacheKeyProvider Ninject can provide it for our class.

ICacheKeyProvider interface and *SimpleCacheKeyProvider* implentation:

```
public interface ICacheKeyProvider
{
    string GenerateKey(Type type);
}

public class SimpleCacheKeyProvider
    : ICacheKeyProvider
{
    public string GenerateKey(Type type)
    {
        if (type == null)
            throw new ArgumentNullException(nameof(type));

        return string.Format("{0}CacheKey", type.Name);
    }
}
```

Modified *CustomCache* class

```
public class CustomCache : ICustomCache
{
    private ICacheKeyProvider CacheKeyProvider { get; set; }

    public CustomCache(ICacheKeyProvider keyProviderParam)
    {
        if (keyProviderParam == null)
            throw new ArgumentNullException(nameof(keyProviderParam));

        CacheKeyProvider = keyProviderParam;
    }

    ...........
}
```

Add binding for *ICacheKeyProvider*:

```
private void AddBindings()
{
    // Bindings added here
    kernel.Bind<ICustomCache>().To<CustomCache>();
    kernel.Bind<ICacheKeyProvider>().To<SimpleCacheKeyProvider>();
}
```

Now when we navigate to *HomeController* Ninject creates instance of *SimpleCacheKeyProvider* uses it to create *CustomCache* and uses CustomCache instance to create *HomeController*.

Ninject has number of great features like chained dependency injection and you should examine them if you want to use Ninject.

Read Dependency Injection online: https://riptutorial.com/asp-net-mvc/topic/6392/dependency-injection

Chapter 13: Display and Editor templates

Introduction

When dealing with objects in an MVC app, if any object should be shown in multiple places with the same format, we'd need some kind of standardized layout. ASP.NET MVC has made this kind of standardization easy to do with the inclusion of display and editor templates. In short, display and editor templates are used to standardize the layout shown to the user when editing or displaying certain types or classes.

Examples

Display Template

Model:

```
public class User
{
   public int ID { get; set; }
   public string FirstName { get; set; }
   public DateTime DateOfBirth { get; set; }
}
```

If we want to display the users in different Views, it would be better to create a standardized layout for these users wherever they need to be displayed. We can accomplish this using display templates.

A display template is simply a partial view that is model-bound to the object it wants to display, and exists in the `Views/Shared/DisplayTemplates` folder (though you can also put it in `Views/ControllerName/DisplayTemplates`). Further, **the name of the view (by default) should be the name of the object you want to use it as the template for**.

    Views/Shared/DisplayTemplates/User.cshtml

```
 @model TemplatesDemo.Models.User

<div style="padding-bottom: 10px">
    <p><strong>ID:</strong> @Html.DisplayFor(m => m.ID)</p>
    <p><strong>Name:</strong> @Html.DisplayFor(m => m.FirstName)</p>
    <p><strong>Date of Birth:</strong> @Html.DisplayFor(m => m.DateOfBirth)</p>
</div>
<hr/>
```

Now, if we want to display all the users from database and show them in different Views we can simply send the list of users to the View and and use the Display Template to show them. We can use one of two methods to do that:

```
Html.DisplayFor()
Html.DisplayForModel()
```

`DisplayFor` call the display template for the type of the property selected (e.g. `Html.DisplayFor(x => x.PropertyName)`. `DisplayForModel` calls the display template for the `@model` of the view

View

```
@model IEnumerable<TemplatesDemo.Models.User>
@{
    ViewBag.Title = "Users";
}

<h2>Users</h2>

@Html.DisplayForModel()
```

Editor Template

Display Templates can be used to standardize the layout of an object, so let's now see how we can do the same thing for these objects when editing them. Just like display templates, there's two ways to call editor templates for a given type:

```
Html.EditorFor()
Html.EditorForModel()
```

Editor templates, similarly to display templates, need to exist in either **Views/Shared/EditorTemplates** or **Views/ControllerName/EditorTemplates**. For this demo, we'll be creating them in the Shared folder. Again, **the name of the view (by default) should be the name of the object you want to use it as the template for.**

Model

```
public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public DateTime DateOfBirth { get; set; }
        public Roles Roles { get; set; }
        public int RoleId { get; set; }
    }

 public class Roles
    {
        public int Id { get; set; }
        public string Role { get; set; }
    }
```

Say we want to be able edit any user from the database in multiple views. We will use a **ViewModel** for this purpose.

ViewModel

```
public class UserEditorViewModel
    {
        public User User { get; set; }
        public IEnumerable<Roles> Roles { get; set; }
    }
```

Using this **ViewModel**, we will create an **Editor Template**

Views/Shared/EditorTemplates/UserEditorViewModel.cshtml

```
@model TemplatesDemo.Models.UserEditorViewModel

<div class="form-group">
    @Html.DisplayNameFor(m => m.User.Id)
    @Html.EditorFor(m => m.User.Id)
</div>
<div class="form-group">
    @Html.DisplayNameFor(m => m.User.Name)
    @Html.EditorFor(m => m.User.Name)
</div>
<div class="form-group">
    @Html.DisplayNameFor(m => m.User.DateOfBirth)
    @Html.EditorFor(m => m.User.DateOfBirth)
</div>
<div class="form-group">
    @Html.DisplayNameFor(m => m.User.Roles.Role)
    @Html.DropDownListFor(m => m.User.RoleId, new SelectList(Model.Roles,"Id","Role"))
</div>
```

We will get the desired user and the list of available roles and bind them in the viewModel **UserEditorViewModel** in the Controller Action and send the viewModel to the view. For simplicity, I am initiating the viewModel from the Action

Action

```
 public ActionResult Editor()
        {
            var viewModel = new UserEditorViewModel
            {
                User = new User
                {
                    Id = 1,
                    Name = "Robert",
                    DateOfBirth = DateTime.Now,
                    RoleId = 1
                },
                Roles = new List<Roles>()
                {
                    new Roles
                    {
                        Id = 1,
                        Role = "Admin"
                    },
                    new Roles
                    {
                        Id = 2,
                        Role = "Manager"
                    },
                    new Roles
                    {
                        Id = 3,
                        Role = "User"
                    }
                }
            };

            return View(viewModel);
        }
```

We can use the created **Editor Template** in any view we wish

View

```
@model TemplatesDemo.Models.UserEditorViewModel

@using (Html.BeginForm("--Your Action--", "--Your Controller--"))
{
    @Html.EditorForModel()
    <input type="submit" value="Save" />
}
```

Read Display and Editor templates online: https://riptutorial.com/asp-net-mvc/topic/9784/display-and-editor-templates

Chapter 14: Dockerization of ASP.NET Application

Examples

Dockerfile and Nuget

Dockerization of ASP.NET Application requires a Dockerfile for configuration and running it as a docker container.

```
FROM microsoft/dotnet:latest

RUN apt-get update && apt-get install sqlite3 libsqlite3-dev

COPY . /app

WORKDIR /app

RUN ["dotnet", "restore"]

RUN ["dotnet", "build"]

RUN npm install && npm run postscript

RUN bower install

RUN ["dotnet", "ef", "database", "update"]

EXPOSE 5000/tcp

ENTRYPOINT ["dotnet", "run", "--server.urls", "http://0.0.0.0:5000"]
```

A nuget feed configuration file helps in retrieving from the correct source. The usage of this file depends on the current configuration of the project and can change to suite project's requirement.

```
 <?xml version="1.0" encoding="utf-8"?>
   <configuration>
    <packageSources>
     <add key="nuget.org" value="https://api.nuget.org/v3/index.json"  protocolVersion="3" />
    <packageSources>
    <packageRestore>
     <add key="enabled" value="True" />
     <add key="automatic" value="True" />
    <packageRestore>
   <bindingRedirects>
    <add key="skip" value="False" />
   </bindingRedirects>
 </configuration>
```

POSTGRESQL Support.

```
"Data": {
    "DefaultConnection": {
        "ConnectionString":
"Host=localhost;Username=postgres;Password=******;Database=postgres;Port=5432;Pooling=true;"
    }
  },
```

Dockerization

It is nessecary to have .NET or a mono-aspnet package.

It is important to understand the importance of dockerization. Install dotnet on ubuntu or the OS you are working on.

Installing DOTNET

```
$ sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/dotnet-release/
trusty main" > /etc/apt/sources.list.d/dotnetdev.list'
$ sudo apt-key adv --keyserver apt-mo.trafficmanager.net --recv-keys 417A0893
$ sudo apt-get update

Ubuntu 16.04



$ sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/dotnet-release/
xenial main" > /etc/apt/sources.list.d/dotnetdev.list'
$ sudo apt-key adv --keyserver apt-mo.trafficmanager.net --recv-keys 417A0893
$ sudo apt-get update
```

Install .NET Core SDK

```
$ sudo apt-get install dotnet-dev-1.0.0-preview2-003121
```

COURTESY : https://www.microsoft.com/net/core#ubuntu

For installation of Docker follow, https://docs.docker.com/engine/installation/linux/ubuntulinux/

FOR PORT :

```
Kestrel server port : 5000
Docker Deamon will listen to port :

 EXPOSE 5000/tcp
```

For building docker :

```
 $ sudo docker build -t myapp .
```

For running the docker container :

```
 $ sudo docker run -t -d -p 8195:5000 myapp
```

For visiting site :

```
$ ifconfig

eth0 : ***.***.**
 server-ip-address
```

Site will be available on (given this configuration.) :

```
 http://server-ip-address:8195
```

Docker Processes. It will list running processes.

```
   $ sudo docker ps
```

To delete the process or the container.

```
 $ sudo docker rm -rf <process_id>
```

Read Dockerization of ASP.NET Application online: https://riptutorial.com/asp-net-mvc/topic/6740/dockerization-of-asp-net-application

Chapter 15: Error Logging

Examples

Simple Attribute

```
using System;
using System.Web;
using System.Web.Mvc;

namespace Example.SDK.Filters
{
    [AttributeUsage(AttributeTargets.Class, Inherited = false, AllowMultiple = false)]
    public sealed class CustomErrorHandlerFilter : HandleErrorAttribute
    {
        public override void OnException(ExceptionContext filterContext)
        {
            // RouteDate is useful for retrieving info like controller, action or other route
values
            string controllerName = filterContext.RouteData.Values["controller"].ToString();
            string actionName = filterContext.RouteData.Values["action"].ToString();

            string exception = filterContext.Exception.ToString(); // Full exception stack
            string message = filterContext.Exception.Message; // Message given by the
exception

            // Log the exception within database
            LogExtensions.Insert(exception.ToString(), message, controllerName + "." +
actionName);

            base.OnException(filterContext);
        }
    }
}
```

Then set it in `FilterConfig.cs`

```
filters.Add(new CustomErrorHandlerFilter());
```

returning custom error page

```
public ActionResult Details( string product)
{
   ....
    if (productNotFound) {
        // http://www.eidias.com/blog/2014/7/2/mvc-custom-error-pages
        Response.Clear();
        Response.TrySkipIisCustomErrors = true;
        Response.Write(product + " product not exists");
        Response.StatusCode = (int)HttpStatusCode.NotFound;
        Response.End();
        return null;
    }
```

}

Create Custom ErrorLogger In ASP.Net MVC

Step 1: Creating Custom Error Logging Filter which will write Errors in Text Files According to DateWise.

```
public class ErrorLogger : HandleErrorAttribute
{
    public override void OnException(ExceptionContext filterContext)
    {

        string strLogText = "";
        Exception ex = filterContext.Exception;
        filterContext.ExceptionHandled = true;
        var objClass = filterContext;
        strLogText += "Message ---\n{0}" + ex.Message;

        if (ex.Source == ".Net SqlClient Data Provider")
        {
            strLogText += Environment.NewLine + "SqlClient Error ---\n{0}" + "Check Sql
Error";
        }
        else if (ex.Source == "System.Web.Mvc")
        {
            strLogText += Environment.NewLine + ".Net Error ---\n{0}" + "Check MVC Code For
Error";
        }
        else if (filterContext.HttpContext.Request.IsAjaxRequest() == true)
        {
            strLogText += Environment.NewLine + ".Net Error ---\n{0}" + "Check MVC Ajax Code
For Error";
        }
        strLogText += Environment.NewLine + "Source ---\n{0}" + ex.Source;
        strLogText += Environment.NewLine + "StackTrace ---\n{0}" + ex.StackTrace;
        strLogText += Environment.NewLine + "TargetSite ---\n{0}" + ex.TargetSite;
        if (ex.InnerException != null)
        {
            strLogText += Environment.NewLine + "Inner Exception is {0}" +
ex.InnerException;//error prone
        }
        if (ex.HelpLink != null)
        {
            strLogText += Environment.NewLine + "HelpLink ---\n{0}" + ex.HelpLink;//error
prone
        }

        StreamWriter log;

        string timestamp = DateTime.Now.ToString("d-MMMM-yyyy", new CultureInfo("en-GB"));

        string error_folder = ConfigurationManager.AppSettings["ErrorLogPath"].ToString();

        if (!System.IO.Directory.Exists(error_folder))
        {
            System.IO.Directory.CreateDirectory(error_folder);
        }

        if (!File.Exists(String.Format(@"{0}\Log_{1}.txt", error_folder, timestamp)))
        {
            log = new StreamWriter(String.Format(@"{0}\Log_{1}.txt", error_folder,
timestamp));
        }
        else
        {
            log = File.AppendText(String.Format(@"{0}\Log_{1}.txt", error_folder, timestamp));
        }

        var controllerName = (string)filterContext.RouteData.Values["controller"];
        var actionName = (string)filterContext.RouteData.Values["action"];

        // Write to the file:
        log.WriteLine(Environment.NewLine + DateTime.Now);
        log.WriteLine("------------------------------------------------------------------------------------------------");
        log.WriteLine("Controller Name :- " + controllerName);
        log.WriteLine("Action Method Name :- " + actionName);
        log.WriteLine("------------------------------------------------------------------------------------------------");
        log.WriteLine(objClass);
        log.WriteLine(strLogText);
        log.WriteLine();

        // Close the stream:
        log.Close();
        filterContext.HttpContext.Session.Abandon();
        filterContext.Result = new RedirectToRouteResult
         (new RouteValueDictionary
         {
                {"controller", "Errorview"}, {"action", "Error"}
         });

    }

}
```

Step 2: Adding Physical Path on Server or Local drive where text file will be stored

```
<add key="ErrorLogPath" value="C:\ErrorLog\DemoMVC\" />
```

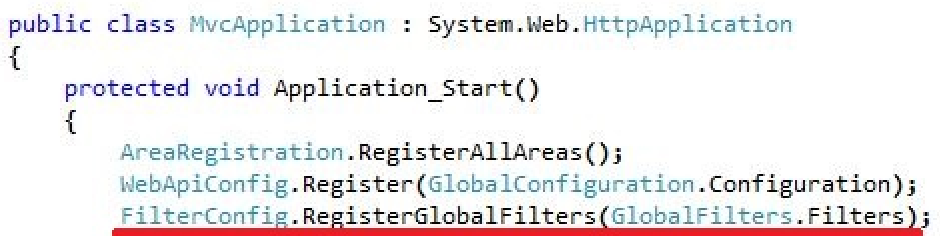
Step 3: Adding **Errorview** Controller with **Error** ActionMethod

Step 4: Adding **Error.cshtml View and Display Custom Error Message on View**

Step 5: Register ErrorLogger Filter in FilterConfig class

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new ErrorLogger());
    }
}
```

Step 6: Register FilterConfig in Global.asax



```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        WebApiConfig.Register(GlobalConfiguration.Configuration);
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
```

Read Error Logging online: https://riptutorial.com/asp-net-mvc/topic/2268/error-logging

Chapter 16: Html Helpers

Introduction

HTML helpers are methods used to render HTML elements in a view. They are part of the `System.Web.Mvc.HtmlHelper` namespace.

There are different types of HTML helpers:

Standard HTML Helpers: They are used to render normal HTML elements, e.g. `Html.TextBox()`.

Strongly Typed HTML Helpers: These helpers render HTML elements based on model properties, e.g. `Html.TextBoxFor()`.

Custom HTML Helpers: The user can create custom helper method which returns `MvcHtmlString`.

Examples

Custom HTML Helper - Display Name

```
/// <summary>
/// Gets displayName from DataAnnotations attribute
/// </summary>
/// <typeparam name="TModel"></typeparam>
/// <typeparam name="TProperty"></typeparam>
/// <param name="htmlHelper"></param>
/// <param name="expression"></param>
/// <returns></returns>
public static MvcHtmlString GetDisplayName<TModel, TProperty>(this HtmlHelper<TModel>
htmlHelper, Expression<Func<TModel, TProperty>> expression)
{
    var metaData = ModelMetadata.FromLambdaExpression(expression, htmlHelper.ViewData);
    var value = metaData.DisplayName ?? (metaData.PropertyName ??
ExpressionHelper.GetExpressionText(expression));
    return MvcHtmlString.Create(value);
}
```

Custom Helper - Render submit button

```
/// <summary>
/// Creates simple button
/// </summary>
/// <param name="poHelper"></param>
/// <param name="psValue"></param>
/// <returns></returns>
public static MvcHtmlString SubmitButton(this HtmlHelper poHelper, string psValue)
{
    return new MvcHtmlString(string.Format("<input type=\"submit\" value=\"{0}\">", psValue));
}
```

Exhaustive list of HtmlHelper samples including HTML output

`HtmlHelper.Action()`

- `@Html.Action(actionName: "Index")`
  output: **The HTML rendered by an action method called `Index()`**
- `@Html.Action(actionName: "Index", routeValues: new {id = 1})`
  output: **The HTML rendered by an action method called `Index(int id)`**
- `@(Html.Action("Index", routeValues: new RouteValueDictionary(new Dictionary<string, object>{ {"id", 1} })))`
  output: **The HTML rendered by an action method called `Index(int id)`**
- `@Html.Action(actionName: "Index", controllerName: "Home")`
  output: **The HTML rendered by an action method called `Index()` in the `HomeController`**
- `@Html.Action(actionName: "Index", controllerName: "Home", routeValues: new {id = 1})`
  output: **The HTML rendered by an action method called `Index(int id)` in the `HomeController`**
- `@Html.Action(actionName: "Index", controllerName: "Home", routeValues: new RouteValueDictionary(new Dictionary<string, object>{ {"id", 1} }))`
  output: **The HTML rendered by an action method called `Index(int id)` in the `HomeController`**

`HtmlHelper.ActionLink()`

- `@Html.ActionLink(linkText: "Click me", actionName: "Index")`
  output: **`<a href="Home/Index">Click me</a>`**
- `@Html.ActionLink(linkText: "Click me", actionName: "Index", routeValues: new {id = 1})`
  output: **`<a href="Home/Index/1">Click me</a>`**
- `@Html.ActionLink(linkText: "Click me", actionName: "Index", routeValues: new {id = 1}, htmlAttributes: new {@class = "btn btn-default", data_foo = "bar")`
  output: **`<a href="Home/Index/1" class="btn btn-default" data-foo="bar">Click me</a>`**
- `@Html.ActionLink()`
  output: **`<a href=""></a>`**

`@HtmlHelper.BeginForm()`

- `@using (Html.BeginForm("MyAction", "MyController", FormMethod.Post, new {id="form1",@class = "form-horizontal"}))`
  output: **`<form action="/MyController/MyAction" class="form-horizontal" id="form1" method="post">`**

Standard HTML Helpers with their HTML Outputs

Html.TextBox()

- `@Html.TextBox("Name", null, new { @class = "form-control" })`
  *output:***`<input class="form-control" id="Name"name="Name"type="text"value=""/>`**
- `@Html.TextBox("Name", "Stack Overflow", new { @class = "form-control" })`

output:`<input class="form-control" id="Name"name="Name"type="text" value="Stack Overflow"/>`

Html.TextArea()

- `@Html.TextArea("Notes", null, new { @class = "form-control" })`
  output:`<textarea class="form-control" id="Notes" name="Notes" rows="2" cols="20"></textarea>`
- `@Html.TextArea("Notes", "Please enter Notes", new { @class = "form-control" })`
  output:`<textarea class="form-control" id="Notes" name="Notes" rows="2" cols="20" >Please enter Notes</textarea>`

Html.Label()

- `@Html.Label("Name","FirstName")`
  output:`<label for="Name"> FirstName </label>`
- `@Html.Label("Name", "FirstName", new { @class = "NameClass" })`
  output:`<label for="Name" class="NameClass">FirstName</label>`

Html.Hidden()

- `@Html.Hidden("Name", "Value")`
  output:`<input id="Name" name="Name" type="hidden" value="Value" />`

Html.CheckBox()

- `@Html.CheckBox("isStudent", true)`
  output:`<input checked="checked" id="isStudent" name="isStudent" type="checkbox" value="true" />`

Html.Password()

- `@Html.Password("StudentPassword")`
  output:`<input id="StudentPassword" name="StudentPassword" type="password" value="" />`

Custom Helper - Render Radio Button with Label

```
       public static MvcHtmlString RadioButtonLabelFor<TModel, TProperty>(this
HtmlHelper<TModel> self, Expression<Func<TModel, TProperty>> expression, bool value, string
labelText)
    {
        // Retrieve the qualified model identifier
        string name = ExpressionHelper.GetExpressionText(expression);
        string fullName = self.ViewContext.ViewData.TemplateInfo.GetFullHtmlFieldName(name);

        // Generate the base ID
        TagBuilder tagBuilder = new TagBuilder("input");
        tagBuilder.GenerateId(fullName);
        string idAttr = tagBuilder.Attributes["id"];

        // Create an ID specific to the boolean direction
        idAttr = string.Format("{0}_{1}", idAttr, value);

        // Create the individual HTML elements, using the generated ID
        MvcHtmlString radioButton = self.RadioButtonFor(expression, value, new { id = idAttr
});
        MvcHtmlString label = self.Label(idAttr, labelText);
        return new MvcHtmlString(radioButton.ToHtmlString() + label.ToHtmlString());
    }
```

Example : `@Html.RadioButtonLabelFor(m => m.IsActive, true, "Yes")`

Custom Helper - Date Time Picker

```
public static MvcHtmlString DatePickerFor<TModel, TProperty>(this HtmlHelper<TModel>
htmlHelper, Expression<Func<TModel, TProperty>> expression, object htmlAttributes)
{
    var sb = new StringBuilder();
    var metaData = ModelMetadata.FromLambdaExpression(expression, htmlHelper.ViewData);
    var dtpId = "dtp" + metaData.PropertyName;
    var dtp = htmlHelper.TextBoxFor(expression, htmlAttributes).ToHtmlString();
    sb.AppendFormat("<div class='input-group date' id='{0}'> {1} <span class='input-group-
addon'><span class='glyphicon glyphicon-calendar'></span></span></div>", dtpId, dtp);
    return MvcHtmlString.Create(sb.ToString());
}
```

Example:

```
@Html.DatePickerFor(model => model.PublishedDate,  new { @class = "form-control" })
```

If you use **Bootstrap.v3.Datetimepicker** The your JavaScript is like below --

```
$('#dtpPublishedDate').datetimepicker({ format: 'MMM DD, YYYY' });
```

Read Html Helpers online: https://riptutorial.com/asp-net-mvc/topic/2290/html-helpers

Chapter 17: Html.AntiForgeryToken

Introduction

The anti-forgery token can be used to help protect your application against cross-site request forgery. To use this feature, call the AntiForgeryToken method from a form and add the ValidateAntiForgeryTokenAttribute attribute to the action method that you want to protect.

Generates a hidden form field (anti-forgery token) that is validated when the form is submitted.

Syntax

- @Html.AntiForgeryToken()

Remarks

When submitting an ajax request with CSRF token (`__RequestVerificationToken`) make sure that content type is not set to `application/json`. If you are using jQuery it automatically sets the content type to `application/x-www-form-urlencoded` which is then recognised by ASP.NET MVC.

Caution

Use caution when setting this value. Using it improperly can open security vulnerabilities in the application.

Examples

Basic usage

The `@Html.AntiForgeryToken()` helper method protects against cross-site request forgery (or CSRF) attacks.

It can be used by simply using the `Html.AntiForgeryToken()` helper within one of your existing forms and decorating its corresponding Controller Action with the `[ValidateAntiForgeryToken]` attribute.

Razor (YourView.cshtml)

```
@using (Html.BeginForm("Manage", "Account")) {
    @Html.AntiForgeryToken()
    <!-- ... -->
}
```

OR

```
<form>
    @Html.AntiForgeryToken()
    <!-- ... -->
</form>
```

Controller (YourController.cs)

The target action method:

```
[ValidateAntiForgeryToken]
[HttpPost]
public ActionResult ActionMethod(ModelObject model)
{
    // ...
}
```

Disable Identity Heuristic Check

Often times you will see an exception

```
Anti forgery token is meant for user "" but the current user is "username"
```

This is because the Anti-Forgery token is also linked to the current logged-in user. This error appears when a user logs in but their token is still linked to being an anonymous user for the site.

There are a few ways to fix this behavior, but if you would rather not have CSRF tokens linked to the logged-in state of a user you may disable this feature.

Put this line in your `Global.asax` file or similar application startup logic.

```
AntiForgeryConfig.SuppressIdentityHeuristicChecks = true;
```

Validating All Posts

Due to the vulnerability caused by CSRF, it is generally considered a good practice to check for an AntiForgeryToken on all HttpPosts unless there is a good reason to not do it (some technical issue with the post, there is another authentication mechanism and/or the post does not mutate state like saving to a db or file). To ensure that you don't forget, you can add a special GlobalActionFilter that automatically checks all HttpPosts unless the action is decorated with a special "ignore" attribute.

```
[AttributeUsage(AttributeTargets.Class)]
public class ValidateAntiForgeryTokenOnAllPosts : AuthorizeAttribute
{
    public override void OnAuthorization(AuthorizationContext filterContext)
    {
        var request = filterContext.HttpContext.Request;

        //  Only validate POSTs
        if (request.HttpMethod == WebRequestMethods.Http.Post)
        {
            bool skipCheck =
filterContext.ActionDescriptor.IsDefined(typeof(DontCheckForAntiForgeryTokenAttribute), true)
                ||
filterContext.ActionDescriptor.ControllerDescriptor.IsDefined(typeof(DontCheckForAntiForgeryTokenAttrib
true);

            if (skipCheck)
                return;


            //  Ajax POSTs and normal form posts have to be treated differently when it comes
            //  to validating the AntiForgeryToken
            if (request.IsAjaxRequest())
            {
                var antiForgeryCookie = request.Cookies[AntiForgeryConfig.CookieName];

                var cookieValue = antiForgeryCookie != null
                    ? antiForgeryCookie.Value
                    : null;

                AntiForgery.Validate(cookieValue,
request.Headers["__RequestVerificationToken"]);
            }
            else
            {
                new ValidateAntiForgeryTokenAttribute()
                    .OnAuthorization(filterContext);
            }
        }
    }
}

/// <summary>
/// this should ONLY be used on POSTS that DO NOT MUTATE STATE
/// </summary>
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false,
Inherited = true)]
public sealed class DontCheckForAntiForgeryTokenAttribute : Attribute { }
```

To make sure it gets checked on all requests, just add it to your Global Action Filters

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        //...
        filters.Add(new ValidateAntiForgeryTokenOnAllPosts());
        //...
    }
}
```

Advance usage: Apply default Antiforgery filter for every POST

We may forget to apply the `Antiforgery attribute` for each `POST` request so we should make it by default. This sample will make sure `Antiforgery filter` will always be applied to every `POST` request.

Firstly create new `AntiForgeryTokenFilter` filter:

```
//This will add ValidateAntiForgeryToken Attribute to all HttpPost action methods
public class AntiForgeryTokenFilter : IFilterProvider
{
    public IEnumerable<Filter> GetFilters(ControllerContext controllerContext,
ActionDescriptor actionDescriptor)
    {
        List<Filter> result = new List<Filter>();

        string incomingVerb = controllerContext.HttpContext.Request.HttpMethod;

        if (String.Equals(incomingVerb, "POST", StringComparison.OrdinalIgnoreCase))
        {
            result.Add(new Filter(new ValidateAntiForgeryTokenAttribute(), FilterScope.Global,
null));
        }

        return result;
    }
}
```

Then register this custom filter to MVC, Application_Start:

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        //Cactch generic error
        filters.Add(new HandleErrorAttribute());

        //Anti forgery token hack for every post request
        FilterProviders.Providers.Add(new AntiForgeryTokenFilter());
    }
}



public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

So now all your `POST` requests are protected by default using Antiforgery attributes so we are no longer need to have `[ValidateAntiForgeryToken]` attribute on each POST method.

Using AntiForgeryToken with Jquery Ajax Request

First off you create the form

```
@using (Html.BeginForm())
{
  @Html.AntiForgeryToken()
}
```

Action Method

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Test(FormViewModel formData)
{
    // ...
}
```

Script

```
<script src="https://code.jquery.com/jquery-1.12.4.min.js"></script>
<script>
var formData = new FormData($('form')[0]);
$.ajax({
    method: "POST",
    url: "/demo/test",
    data: formData ,
    success: function (data) {
  console.log(data);
    },
    error: function (jqXHR, textStatus, errorThrown) {
        console.log(errorThrown);
    }
})
</script>
```

Make sure contentType isn't set to anything apart from `application/x-www-form-urlencoded` and if its not specified Jquery defaults to `application/x-www-form-urlencoded`

Read Html.AntiForgeryToken online: https://riptutorial.com/asp-net-mvc/topic/1997/html-antiforgerytoken

Chapter 18: Html.RouteLink

Parameters

Parameter	Details
linkText	The text that will be displayed for the link.
routeName	The name of the route to return a virtual path for.

Examples

Basic Example Using Link Text and Route Name

As an alternative to using `Html.ActionLink` to generate links in a view, you can use

`Html.RouteLink`

To make use of this feature, you need to configure a route, for example:

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapRoute(
    "SearchResults",
    "{controller}/{action}",
    new { controller = "Search", action = "Results" });
}
```

Then in a view you can create a link to that route like so:

```
@Html.RouteLink("Search Results", "SearchResults");
```

Using `RouteLink()` is convenient if you end up changing controller names, or action method names, since using `Html.ActionLink()` means having to change the controller and action method name parameters in the call, so that they match the new names which have been changed.

With `RouteLink()` you can change the route details in the `MapRoute()` call, in other words in one location, and any code that is referencing that route via `RouteLink()` will not be required to change.

Read Html.RouteLink online: https://riptutorial.com/asp-net-mvc/topic/6209/html-routelink

Chapter 19: Http Error Handling

Introduction

Every website needs to handle errors. You could let your users see the stock 404 or 500 error pages that IIS dishes out or, using the Web.Config and a simple Controller you can capture these errors and deliver your own custom error pages.

Examples

Basic Setup

This example will cover creating a custom error page for 404 Page Not Found and 500 Server Error. You can extend this code to capture any error code you need to.

Web.Config

If you are using IIS7 and above, ignore the `<CustomError..` node and use `<httpErrors...` instead.

Add in the following in the `system.webServer` node:

```
<httpErrors errorMode="Custom" existingResponse="Replace">
    <remove statusCode="404" />
    <remove statusCode="500" />
    <error statusCode="404" path="/error/notfound" responseMode="ExecuteURL" />
    <error statusCode="500" path="/error/servererror" responseMode="ExecuteURL" />
 </httpErrors>
```

This tells the site to direct any 404 errors to `~/error/notfound` and any 500 error to `~/error/servererror`. It will also preserve your requested URL (think *transfer* rather than *redirect*) so the user will never see the `~/error/...` page URL.

Next, you need a new `Error` controller so...

```
public class ErrorController : Controller
{
    public ActionResult servererror()
    {
        Response.TrySkipIisCustomErrors = true;
        Response.StatusCode = (int)HttpStatusCode.InternalServerError;
        return View();
    }

    public ActionResult notfound()
    {
        Response.TrySkipIisCustomErrors = true;
        Response.StatusCode = (int)HttpStatusCode.NotFound;
        return View();
    }
}
```

The key thing to note here is the `Response.TrySkipIisCustomErrors = true;`. This will bypass IIS and force your error page through.

Lastly, create the corresponding `NotFound` and `ServerError` Views and style them up so it's all nice and seamless with your site's design.

Hey presto - custom error pages.

Read Http Error Handling online: https://riptutorial.com/asp-net-mvc/topic/9137/http-error-handling

Chapter 20: IIS Rewrite Rules

Examples

Force HTTPS using Rewrite rule

This example shows how you can use IIS Rewrite rules to force HTTPS by making all HTTP requests return a 301 (Permanent) Redirect to the HTTPS page.

This is usually better than using the `[RequireHttps]` attribute because the attribute uses a 302 redirect, and being in the MVC pipeline it is much slower than doing it at the IIS level.

```
    <rewrite xdt:Transform="Insert">
       <rules>
         <rule name="Enforce HTTPS WWW" stopProcessing="true">
           <match url=".*" />
           <conditions logicalGrouping="MatchAll" trackAllCaptures="true">
             <add input="{HTTP_HOST}" pattern="^(?!www)(.*)"/>
             <add input="{URL}" pattern="^(.*)"/>
             <!-- {URL} Gives the base portion of the URL, without any querystring or extra
path information, for example, "/vdir/default.asp". -->
           </conditions>
           <action type="Redirect" url="https://www.{C:1}{C:2}" appendQueryString="true"
redirectType="Permanent" />
         </rule>
       </rules>
     </rewrite>
```

Read IIS Rewrite Rules online: https://riptutorial.com/asp-net-mvc/topic/6358/iis-rewrite-rules

Chapter 21: jQuery Ajax Call With Asp MVC

Examples

Posting JavaScript objects with jQuery Ajax Call

Ajax calls, request and retrieve data for giving the user a sense of a better interactive user interface experience. This article will show you how to use jQuery and send data through Ajax calls. For this example, we're going to POST the following JavaScript object to our server.

```
var post = {
    title: " Posting JavaScript objects with jQuery Ajax Call",
    content: " Posting JavaScript objects with jQuery Ajax Call",
    tags: ["asp mvc", "jquery"]
};
```

The server side

The server side model corresponding the javascript object.

```
public class Post
{
    public string Title { get; set; }
    public string Content { get; set; }
    public string[] Tags { get; set; }
}
```

All we need to do is create a standard ASP.NET MVC controller method which takes a single parameter of the Person type, like so.

```
public class PostController : BaseController
{
    public bool Create(Post model)
    {
        //Do somthing
    }
}
```

The client side

To send JavaScript Objects we need to use the JSON.stringify() method for send the object to the data option.

```
$.ajax({
    url: '@Url.Action("create", "Post")',
    type: "POST",
    contentType: "application/json",
    data: JSON.stringify({ model: post })
}).done(function(result){
    //do something
});
```

Read jQuery Ajax Call With Asp MVC online: https://riptutorial.com/asp-net-mvc/topic/9734/jquery-ajax-call-with-asp-mvc

Chapter 22: Model binding

Introduction

Model binding is the process of taking HTTP parameters, typically in the Query String of a GET request, or within POST body, and applying it into an object that can then be validated and consumed in an object-oriented manner without the need for Controller actions having intimate knowledge of how to retrieve HTTP parameters.

In other words, model binding is what allows actions, in MVC, to have either parameter(s), whether it being a value type or an object.

Remarks

To try to create instance in the action, the bind model process will search data in various places:

- Form Data
- Route Data
- Query String
- Files Custom (cookies for example)

Examples

Route value binding

Given some default routing such as `{controller=Home}/{action=Index}/{id?}` if you had the url `https://stackoverflow.com/questions/1558902`

This would go to the QuestionsController and the value 1558902 would be mapped to an id parameter of an index action, i.e.

```
public ActionResult Index(int? id){
     //id would be bound to id of the route
}
```

Query string binding

To extend on the route binding say you had a url like `https://stackoverflow.com/questions/1558902?sort=desc`

and routing like `{controller=Home}/{action=Index}/{id?}`

```
public ActionResult Index(int? id, string sort){
     //sort would bind to the value in the query string, i.e. "desc"
}
```

Binding to objects

Often you'd be working with viewmodel classes in asp.net-mvc and would want to bind to properties on these. This works similar to mapping to individual parameters.

Say you had a simple view model call PostViewModel like this

```
public class PostViewModel{
  public int Id {get;set;}
  public int SnappyTitle {get;set;}
}
```

Then you had posted values of Id and SnappyTitle from a form in the http request then they would map right onto that model if the model itself was the action parameter, e.g.

```
public ActionResult UpdatePost(PostViewModel viewModel){
  //viewModel.Id would have our posted value
}
```

It's worth noting the binding is case insensitive for the parameter and property names. It will also cast values where possible. I'm leaving more edge cases for specific examples

Ajax binding

These are form values that go in the HTTP request using the POST method. (including jQuery POST requests).

Say you did an ajax post like

```
$.ajax({
    type: 'POST',
    url: window.updatePost,
    data:  { id: 21, title: 'snappy title' },
    //kept short for clarity
});
```

Here the two values in json, id and title, would be bound to the matching action, e.g.

```
public JsonResult UpdatePost(int id, string title) {
    ...
}
```

Generic, Session based model binding

Sometimes we need preserve *whole model* and transfer it across actions or even controllers. Storing model at session good solution for this type of requirements. If we combine this with powerful *model binding* features of MVC we get elegant way of doing so. We can create generic session based model binding in three easy steps:

Step one: Create model binder

Create a model binder itself. Personally I created *SessionDataModelBinder* class in */Infrastructure/ModelBinders* folder.

```
using System;
using System.Web.Mvc;

public class SessionDataModelBinder<TModel>
    : IModelBinder
    where TModel : class
{
    private string SessionKey { get; set; }

    public SessionDataModelBinder(string sessionKey)
    {
        if (string.IsNullOrEmpty(sessionKey))
            throw new ArgumentNullException(nameof(sessionKey));
        SessionKey = sessionKey;
    }

    public object BindModel(
        ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        // Get model from session
        TModel model = controllerContext
            .HttpContext
            .Session[SessionKey] as TModel;
        // Create model if it wasn't found from session and store it
        if (model == null)
        {
            model = Activator.CreateInstance<TModel>();
            controllerContext.HttpContext.Session[SessionKey] = model;
        }
        // Return the model
        return model;
    }
}
```

Step two: register binder

If we have model like below:

```
public class ReportInfo
{
    public int ReportId { get; set; }
    public ReportTypes TypeId { get; set; }
}

public enum ReportTypes
{
    NotSpecified,
    Monthly, Yearly
}
```

We can register session based model binder for this model in **Global.asax** in *Application_Start* method:

```
protected void Application_Start()
{
    .........

    // Model binders.
    // Remember to specy unique SessionKey
    ModelBinders.Binders.Add(typeof(ReportInfo),
        new SessionDataModelBinder<ReportInfo>("ReportInfo"));
}
```

Step three: use it!

Now we can benefit from this model binder simply by *adding parameter to our actions*:

```
public class HomeController : Controller
{
    public ActionResult Index(ReportInfo reportInfo)
    {
        // Simply set properties
        reportInfo.TypeId = ReportTypes.Monthly;

        return View();
    }

    public ActionResult About(ReportInfo reportInfo)
    {
        // reportInfo.TypeId is Monthly now because we set
        // it previously in Index action.
        ReportTypes currentReportType = reportInfo.TypeId;

        return View();
    }
}
```

Prevent binding on PostModel

Considering a (post)model:

```
public class User
{
    public string FirstName { get; set; }
    public bool IsAdmin { get; set; }
}
```

With a view like so:

```
@using (Html.BeginForm()) {
    @Html.EditorFor(model => model.FirstName)
    <input type="submit" value="Save" />
}
```

In order to prevent a malicious user from assigning IsAdmin you can use the `Bind` attribute in the action:

```
[HttpPost]
public ViewResult Edit([Bind(Exclude = "IsAdmin")] User user)
{
    // ...
}
```

File Upload

Model:

```
public class SampleViewModel
{
    public HttpPostedFileBase file {get;set;}
}
```

View:

```
@model HelloWorldMvcApp.SampleViewModel

@using (Html.BeginForm("Index","Home",FormMethod.Post, new { enctype = "multipart/form-data"
}))
{
    <div class="form-group">
        @Html.TextBoxFor(model => model.file, new {@class="form-control", type="file"})
        @Html.ValidationMessageFor(model => model.file)
    </div>

    <button type="submit" class="btn btn-success submit">Upload</button>
}
```

Action:

```
[HttpPost]
public ActionResult Index(SampleViewModel model)
{

    if (model.file.ContentLength > 0)
    {
        string fileName = Path.GetFileName(model.file.FileName);
        string fileLocation = "~/App_Data/uploads/"+ fileName;
        model.file.SaveAs(Server.MapPath(fileLocation));
    }
    return View(model);
}
```

Validating date fields manually with dynamic formats using model binder

If different users need different datetime format then you may need to parse your incoming date string to actual date according to the format. In this case this snippet may help you.

```
public class DateTimeBinder : DefaultModelBinder
{
    public override object BindModel(ControllerContext controllerContext, ModelBindingContext
bindingContext)
```

```
    {

                var value = bindingContext.ValueProvider.GetValue(bindingContext.ModelName);
                DateTime date;
                var displayFormat = Session["DateTimeFormat"];
                if (value.AttemptedValue != "")
                {
                    if (DateTime.TryParseExact(value.AttemptedValue, displayFormat,
CultureInfo.InvariantCulture, DateTimeStyles.None, out date))
                    {
                        return date;
                    }
                    else
                    {
                        bindingContext.ModelState.AddModelError(bindingContext.ModelName,
"Invalid date format");
                    }
                }
            }

        return base.BindModel(controllerContext, bindingContext);
    }
```

Read Model binding online: https://riptutorial.com/asp-net-mvc/topic/1258/model-binding

Chapter 23: Model validation

Examples

Validate Model in ActionResult

```
[HttpPost]
public ActionResult ContactUs(ContactUsModel contactObject)
{
    // This line checks to see if the Model is Valid by verifying each Property in the Model
meets the data validation rules
    if(ModelState.IsValid)
    {
    }
    return View(contactObject);
}
```

The model class

```
public class ContactUsModel
{
    [Required]
    public string Name { get; set; }
    [Required]
    [EmailAddress] // The value must be a valid email address
    public string Email { get; set; }
    [Required]
    [StringLength(500)] // Maximum length of message is 500 characters
    public string Message { get; set; }
}
```

Remove an object from validation

Say you have the following model:

```
public class foo
{
    [Required]
    public string Email { get; set; }

    [Required]
    public string Password { get; set; }

    [Required]
    public string FullName { get; set; }
}
```

But you want to exclude FullName from the modelvalidation because you are using the model also in a place where FullName is not filled in, you can do so in the following way:

```
ModelState.Remove("FullName");
```

Custom Error Messages

If you want to provide Custom Error Messages you would do it like this:

```
public class LoginViewModel
{
    [Required(ErrorMessage = "Please specify an Email Address")]
    [EmailAddress(ErrorMessage = "Please specify a valid Email Address")]
    public string Email { get; set; }

    [Required(ErrorMessage = "Type in your password")]
    public string Password { get; set; }
}
```

When your Error Messages are in a ResourceFile (.resx) you have to specify the ResourceType and the ResourceName:

```
public class LoginViewModel
{
    [Required(ErrorMessageResourceType = typeof(ErrorResources), ErrorMessageResourceName =
"LoginViewModel_RequiredEmail")]
    [EmailAddress(ErrorMessageResourceType = typeof(ErrorResources), ErrorMessageResourceName
= "LoginViewModel_ValidEmail")]
    public string Email { get; set; }

    [Required(ErrorMessageResourceType = typeof(ErrorResources), ErrorMessageResourceName =
"LoginViewModel_RequiredPassword")]
    public string Password { get; set; }
}
```

Creating Custom Error Messages in Model and in Controller

Let's say that you have the following class:

```
public class PersonInfo
{
    public int ID { get; set; }

    [Display(Name = "First Name")]
    [Required(ErrorMessage = "Please enter your first name!")]
    public string FirstName{ get; set; }

    [Display(Name = "Last Name")]
    [Required(ErrorMessage = "Please enter your last name!")]
    public string LastName{ get; set; }

    [Display(Name = "Age")]
    [Required(ErrorMessage = "Please enter your Email Address!")]
    [EmailAddress(ErrorMessage = "Invalid Email Address")]
    public string EmailAddress { get; set; }
}
```

These custom error messages will appear if your `ModelState.IsValid` returns false.

But, you as well as I know that there can only be 1 email address per person, or else you will be sending emails to potentially wrong people and/or multiple people. This is where checking in the controller comes into play. So let's assume people are creating accounts for you to save via the Create Action.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "ID, FirstName, LastName, EmailAddress")]
PersonInfo newPerson)
{
    if(ModelState.IsValid) // this is where the custom error messages on your model will
display if return false
    {
        if(database.People.Any(x => x.EmailAddress == newPerson.EmailAddress))  // checking if
the email address that the new person is entering already exists.. if so show this error
message
        {
            ModelState.AddModelError("EmailAddress", "This email address already exists!
Please enter a new email address!");
            return View(newPerson);
        }

        db.Person.Add(newPerson);
        db.SaveChanges():
        return RedirectToAction("Index");
    }

    return View(newPerson);
}
```

I hope this is able to help somebody!

Model Validation in JQuery.

In cases where you need to ensure model validation using Jquery, .valid() function can be used.

The model class fields

```
[Required]
[Display(Name = "Number of Hospitals")]
public int Hospitals{ get; set; }
[Required]
[Display(Name = "Number of Beds")]
public int Beds { get; set; }
```

The View code

```
@using (Html.BeginForm(new {id = "form1", @class = "form-horizontal" }))
{

<div class="divPanel">
  <div class="row">
    <div class="col-md-3">
            @Html.LabelFor(m => m.Hospitals)
            @Html.TextBoxFor(m => m.Hospitals, new { @class = "form-control", @type =
"number"})
            @Html.ValidationMessageFor(m => m.Hospitals)
    </div>
    <div class="col-md-3">

            @Html.LabelFor(m => m.Beds)
            @Html.TextBoxFor(m => m.Beds, new { @class = "form-control", @type = "number"})
            @Html.ValidationMessageFor(m => m.Beds)
    </div>
<div class="col-md-3">
        <button type=button  class="btn btn-primary" id="btnCalculateBeds"> Calculate
Score</button>
    </div>
 </div>

  </div>
 }
```

The script for Validation check.

```
$('#btnCalculateBeds').on('click', function (evt) {
evt.preventDefault();

if ($('#form1').valid()) {
//Do Something.
}
}
```

Ensure that the `jquery.validate` and `jquery.validate.unobtrusive` files are present in the solution.

Read Model validation online: https://riptutorial.com/asp-net-mvc/topic/2683/model-validation

Chapter 24: MVC Ajax Extensions

Introduction

This documents the use of the `System.Web.Mvc.Ajax` library.

Citing MSDN docs "Each extension method renders an HTML element. The ActionLink method renders an anchor (a) element that links to an action method. The RouteLink method renders an anchor (a) element that links to a URL, which can resolve to an action method, a file, a folder, or some other resource. This class also contains BeginForm and BeginRouteForm methods that help you create HTML forms that are supported by AJAX functions.

Parameters

AJAX Options	Description
Confirm	Gets or sets the message to display in a confirmation window before a request is submitted.
HttpMethod	Gets or sets the HTTP request method ("Get" or "Post").
InsertionMode	Gets or sets the mode that specifies how to insert the response into the target DOM element.
LoadingElementDuration	Gets or sets a value, in milliseconds, that controls the duration of the animation when showing or hiding the loading element.
LoadingElementId	Gets or sets the id attribute of an HTML element that is displayed while the Ajax function is loading.
OnBegin	Gets or sets the name of the JavaScript function to call immediately before the page is updated.
OnComplete	Gets or sets the JavaScript function to call when response data has been instantiated but before the page is updated.
OnFailure	Gets or sets the JavaScript function to call if the page update fails.
OnSuccess	Gets or sets the JavaScript function to call after the page is successfully updated.
UpdateTargetId	Gets or sets the ID of the DOM element to update by using the response from the server.
Url	Gets or sets the URL to make the request to.

Remarks

The package `Jquery.Unobtrusive-Ajax` is required in the project. The corresponding javascript files must be included in a bundle (`jquery.unobtrusive-ajax.js` or `jquery.unobtrusive-ajax.min.js`). Finally, it must be activated as well in the `web.config` file:

```
<appSettings>
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

The Actions invoked (`SomeAction` in the examples) must either return a `Json` or a `PartialView`.

Examples

Ajax Action Link

```
@* Renders an anchor (a) element that links to an action method.
 * The innerHTML of "target-element" is replaced by the result of SomeAction.
 *@
@Ajax.ActionLink("Update", "SomeAction", new AjaxOptions{UpdateTargetId="target-element" })
```

Ajax Forms

```
@* Adds AJAX functions support to a form.
 * The innerHTML of "target-element" is replaced by the result of SomeAction.
 *@
@using ( Ajax.BeginForm("SomeAction", "SomeController",
                        new AjaxOptions {
                            UpdateTargetId="target-element",
                            OnSuccess = "some_js_fun(context)"
                        })
)
{
    <!-- my form contents -->
}
```

Read MVC Ajax Extensions online: https://riptutorial.com/asp-net-mvc/topic/9007/mvc-ajax-extensions

Chapter 25: MVC vs Web Forms

Introduction

Before you jump into ASP. NET MVC to develop your web application you should consider the advantages and disavantages of the framework and you should know that there is another web framework made and maintained by Microsoft that is ASP .NET Web Forms.

Which one should you choose is a matter of knowledge of both techonologies.

Syntax

- The ASPX View Engine uses "<%= %>" or "<%: %>" to render server-side content.
- The Razor View Engine uses @ to render server-side content.

Remarks

https://www.asp.net/web-forms

https://www.asp.net/mvc

Examples

Advantages of ASP .NET Web Forms

- Pre build controls to handle Grids, Inputs, Graphs, Trees, and so on.
- It supports an event model that preserves state over HTTP, which benefits line-of-business Web application development. The Web Forms-based application provides dozens of events that are supported in hundreds of server controls.
- It uses a Page Controller pattern that adds functionality to individual pages. For more information, see Page Controller on the MSDN Web site.
- It uses view state or server-based forms, which can make managing state information easier.
- It works well for small teams of Web developers and designers who want to take advantage of the large number of components available for rapid application development.
- In general, it is less complex for application development, because the components (the Page class, controls, and so on) are tightly integrated and usually require less code than the MVC model.
- Easy development model for those developers coming from WindowsForm development.

What is Web Forms

Advantages of an MVC-Based Web Application

- It makes it easier to manage complexity by dividing an application into the model, the view, and the controller (Separation of concerns).
- It does not use view state or server-based forms. This makes the MVC framework ideal for developers who want full control over the behavior of an application.
- It uses a Front Controller pattern that processes Web application requests through a single controller. This enables you to design an application that supports a rich routing infrastructure. For more information, see Front Controller on the MSDN Web site.
- It provides better support for test-driven development (TDD).
- It works well for Web applications that are supported by large teams of developers and Web designers who need a high degree of control over the application behavior.

What is Web Forms

Disadvantages

Web Forms:

- Complex Page Life Cycle, whenever a Request is made to the server there are at least 5 methods to execute previous to the event handler.
- Dificult to work with Client-Side frameworks like JQuery or Angular.
- Hard to work with Asyncronous Javascript and XML (AJAX)
- Viewstate handling
- The page's client-side and the code behind are tightly coupled.

MVC:

- It takes more time to develop in comparision with Web Forms.
- Data is sent in clear text format to the server whereas in web forms view state data are encrypted by default.

Razor View Engine VS ASPX View Engine

Razor (MVC)	ASPX (Web Forms)
The namespace used by the Razor View Engine is System.Web.Razor	The namespace used by the ASPX View Engine is System.Web.Mvc.WebFormViewEngine
The file extensions used by the Razor View Engine are different from a web form view engine. It	The file extensions used by the Web Form View Engines are like ASP.Net web forms. It uses the ASPX extension to view the aspc extension for partial views or User

Razor (MVC)	ASPX (Web Forms)
uses cshtml with C# and vbhtml with vb for views, partial view, templates and layout pages.	Controls or templates and master extensions for layout/master pages.
The Razor Engine supports Test Driven Development (TDD).	Web Form view engine does not support Test Driven Development (TDD) because it depends on the System.Web.UI.Page class to make the testing complex.

ASPX View Engine VS Razor View Engine

Read MVC vs Web Forms online: https://riptutorial.com/asp-net-mvc/topic/8584/mvc-vs-web-forms

Chapter 26: Partial Views

Introduction

A partial view is a view that is rendered within another view. Partial views can be reused and thus prevent duplication of code. They can be rendered by Html.Partial or Html.RenderPartial

Syntax

- @Html.Partial("ViewName")

  @Html.Partial("ViewName",ViewModel)

  @{Html.RenderPartial("ViewName");}

  If your partial view is located in a different folder other than shared folder, then you will have to mention full path of the view as below:

  -@Html.RenderPartial("~/Areas/Admin/Views/Shared/partial/_subcat.cshtml")

Examples

Partial View with model

A model can also be added to the partial view :

```
@model Solution.Project.Namespace.MyModelClass
<p>@Model.Property</p>
```

In the View you can now just use:

```
<div>
    @Html.Partial("PartialViewExample", new MyModelClass(){Property="my property value"})
</div>
<div>
    @{ Html.RenderPartial("PartialViewExample", new MyModelClass(){Property="my property
value"}); }
</div>
```

Partial View to a String - for email content etc

Calling the function

```
string InvoiceHtml = myFunction.RenderPartialViewToString("PartialInvoiceCustomer",
ToInvoice); // ToInvoice is a model, you can pass parameters if needed
```

Function to generate HTML

```
public static string RenderPartialViewToString(string viewName, object model)
{
    using (var sw = new StringWriter())
    {
        BuyOnlineController controller = new BuyOnlineController(); // instance of the
required controller (you can pass this as a argument if needed)

        // Create an MVC Controller Context
        var wrapper = new HttpContextWrapper(System.Web.HttpContext.Current);

        RouteData routeData = new RouteData();

        routeData.Values.Add("controller",
controller.GetType().Name.ToLower().Replace("controller", ""));

        controller.ControllerContext = new ControllerContext(wrapper, routeData, controller);

        controller.ViewData.Model = model;

        var viewResult = ViewEngines.Engines.FindPartialView(controller.ControllerContext,
viewName);

        var viewContext = new ViewContext(controller.ControllerContext, viewResult.View,
controller.ViewData, controller.TempData, sw);
        viewResult.View.Render(viewContext, sw);

        return sw.ToString();
    }
}
```

Partial View - PartialInvoiceCustomer

```
@model eDurar.Models.BuyOnlineCartMaster
 <h2>hello customer - @Model.CartID </h2>
```

Html.Partial Vs Html.RenderPartial

Html.Partial returns a string on the other hand **Html.RenderPartial** returns void.

Html.RenderPartial

This method returns void and the result is directly written to the HTTP response stream. That means it uses the same TextWriter object used in the current webpage/template. For this reason, this method is faster than Partial method.This method is useful when the displaying data in the partial view is already in the corresponding view model.

Example : In a blog to show comments of an article, we would like to use RenderPartial method since an article information with comments are already populated in the view model.

```
@{Html.RenderPartial("_Comments");}
```

Html.Partial

This method returns an HTML-encoded string. This can be stored in a variable. Like RenderPartial method, Partial method is also useful when the displaying data in the partial view is already in the corresponding view model.

Example: In a blog to show comments of an article, you can use Partial method since an article information with comments are already populated in the view model.

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

Read Partial Views online: https://riptutorial.com/asp-net-mvc/topic/2171/partial-views

Chapter 27: Razor

Introduction

What is Razor?

Razor is a markup syntax that lets you embed server-based code (Visual Basic and C#) into web pages.

Server-based code can create dynamic web content on the fly, while a web page is written to the browser. When a web page is called, the server executes the server-based code inside the page before it returns the page to the browser. By running on the server, the code can perform complex tasks, like accessing databases.

Syntax

- @{ ... }
- @variableName
- @(variableName)
- @for(...){ }
- @(Explicit Expression)
- @* comments *@

Remarks

ASP.NET Razor includes view engines for both C# and VB.

The C# view engine processes files with a `.cshtml` extension, while the VB view engine works with `.vbhtml` files.

Examples

Add Comments

Razor has its own comment syntax which begins with `@*` and ends with `*@`.

Inline Comment:

```
<h1>Comments can be @*hi!*@ inline</h1>
```

Multi-line Comment:

```
@* Comments can spread
   over multiple
   lines *@
```

HTML Comment

You can also use the normal HTML comment syntax starting with `<!--` and ending with `-->` in Razor views. But unlike other comments, the Razor code inside a HTML comment is still executed normally.

```
@{
    var hello = "Hello World!";
}
<!-- @hello -->
```

The above example produces the following HTML output:

```
<!-- Hello World! -->
```

Comments within a code block:

```
@{
    // This is a comment
    var Input = "test";
}
```

Display HTML within Razor code block

While inside a Razor code block, the browser will only recognize HTML code if the code is escaped.

Use `@:` for a Single line:

```
@foreach(int number in Model.Numbers)
{
    @:<h1>Hello, I am a header!</h1>
}
```

Use `<text> ... </text>` for Multi-line:

```
@{
    var number = 1;

    <text>
        Hello, I am text
        <br / >
        Hello, I am more text!
    </text>
}
```

Note that Razor, when inside a code block, will understand HTML tags. Therefore, adding the `text` tag around HTML tags is unnecessary (although still correct), such as:

```
@{
    var number = 1;
    <text>
        <div>
            Hello, I am text
            <br / >
            Hello, I am more text!
        </div>
    </text>
}
```

Basic Syntax

Razor code can be inserted anywhere within HTML code. Razor code blocks are enclosed in `@{ ... }`. Inline variable and functions start with `@`. Code inside the Razor brackets follow the normal C# or VB rules.

Single line statement:

```
@{ var firstNumber = 1; }
```

Multi-line code block:

```
@{
    var secondNumber = 2;
    var total = firstNumber + secondNumber;
}
```

Using a variable inline:

```
<h1>The total count is @total</h1>
```

Using a variable inline explicitly:

```
<h2>Item@(item.Id)</h2>
```

For this particular example we will not be able to use the implicit syntax because `Item@item.Id` looks like an email and will be rendered as such by Razor.

Enclose code inside control flow statements:

```
<h1>Start with some HTML code</h1>

@for (int i = 0; i < total; i++){
    Console.Write(i);
}

<p>Mix in some HTML code for fun!</p>
<p>Add a second paragraph.</p>

@if (total > 3)
{
    Console.Write("The total is greater than 3");
}
```

```
else
{
    Console.Write("The total is less than 3");
}
```

This same syntax would be used for all statements such as `for`, `foreach`, `while`, `if`, `switch`, etc.

Adding code inside of code:

```
@if (total > 3)
{
    if(total == 10)
    {
        Console.Write("The total is 10")
    }
}
```

Note that you don't need to type the `@` at the second `if`. After code you can just type other code behind the existing code.

If you want to add code after a HTML element you **do** need to type a `@`.

Escaping @ character

In many cases, the Razor parser is smart enough to figure out when the `@` sign is meant to be used as part of code, as opposed to being part of something like an email address. In the example below, escaping the `@` sign is not necessary:

```
<p>Reach out to us at contact@mail.com</p>
```

However, in some cases, usage of the `@` sign is more ambiguous, and it must be explicitly escaped with `@@`, as in the example below:

```
<p>Join us @@ Stack Overflow!</p>
```

Alternatively, we can use a HTML encoded `@` character

```
<p>Join us &#64; Stack Overflow!</p>
```

Create inline classes and methods using @functions

Using Razor `@functions` keyword gives the capability of introducing classes and methods for inline use within a Razor file:

```
@functions
{
    string GetCssClass(Status status)
    {
        switch (status)
        {
            case Status.Success:
                return "alert-success";
            case Status.Info:
                return "alert-info";
            case Status.Warning:
                return "alert-warning";
            case Status.Danger:
            default:
                return "alert-danger";
        }
    }
}


<label class="alert @GetCssClass(status)"></label>
```

The same can be done for classes:

```
@functions
{
    class Helpers
    {
        //implementation
    }
}
```

Adding a custom attribute with - (hyphen) in name

If you need to add an attribute through razor that has a - (hyphen) in the name you cannot simply do

```
@Html.DropDownListFor(m => m.Id, Model.Values, new { @data-placeholder = "whatever" })
```

it will not compile. data-* attributes are valid and common in html5 for adding extra values to elements.

However the following will work

```
@Html.DropDownListFor(m => m.Id, Model.Values, new { @data_placeholder = "whatever" })
```

since "_" is replaced with "-" when rendered.

This works fine as underscores are not acceptable in attribute names in html.

Editor Templates

Editor templates are a good way to reuse Razor code. You can define editor templates as Razor partial views and then use them in other views.

Editor templates usually exist in the `Views/Shared/EditorTemplates/` folder, although they can also be saved to the `Views/ControllerName/EditorTemplates/` folder. The name of the view is typically the name of the object you want to use the template for, like `<type>.cshtml`.

Here is a simple editor template for DateTime:

```
@model DateTime
<div>
   <span>
      @Html.TextBox("", Model.ToShortDateString(), new { data_date_picker="true" })
   </span>
</div>
```

Save the file as **Views/Shared/EditorTemplate/DateTime.cshtml**.

Then, use `EditorFor` to call this template code in a another view:

```
@Html.EditorFor(m => m.CreatedDate)
```

There is also a UIHint attribute to specify the file name:

```
public class UiHintExampleClass
{
    [UIHint("PhoneNumber")]
    public string Phone { get; set; }
}
```

Define this phone number template in **Views/Shared/EditorTemplates/PhoneNumber.cshtml**.

Editor templates can be defined for Custom Types as well.

Here is a custom type called `SubModel`:

```
public class SubModel
{
    public Guid Id { get; set;}
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Model
{
    public Guid Id { get; set; }
    public DateTime Created {get; set; }

    public SubModel SubModel{get; set; }
}
```

This is the EditorTemplate for SubModel:

```
@model SubModel
<div class="form-group">
    @Html.LabelFor(m => m.FirstName)
    @Html.TextBoxFor(m => m.FirstName)
</div>
<div class="form-group">
```

```
    @Html.LabelFor(m => m.LastName)
    @Html.TextBoxFor(m => m.LastName)
</div>
```

Now, the View for Model simply becomes:

```
@model Model
@Html.EditorFor(m => m.CreatedDate)
@Html.EditorFor(m => m.SubModel, new { @Prefix = "New"})
@* the second argument is how you can pass viewdata to your editor template*@
```

Pass Razor content to a @helper

Send a Razor part to a @helper, for example a HTML div.

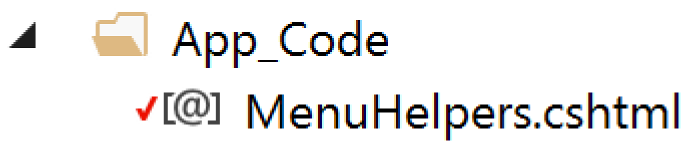
```
@helper WrapInBox(Func<Object, HelperResult> content)
{
    <div class="box">@content(null) </div>
}

//call
@WrapInBox(@<div>
                I'm a inner div
            </div>)
```

Share @helpers across views

@Helpers could be shared between views.

They should be created in the folder App_Code



```
@helper CreatePrimaryBootstrapButton(string label)
{
    <button type="button" class="btn btn-primary">@label</button>
}

//call

@MenuHelpers.CreatePrimaryBootstrapButton("my button")
```

The globals `@Url` and `@Html` aren't available by default in the @Helper defined in App_code. You could add them as follows (for every .cshtml in your App_code folder)

```
@*  Make @Html and @Url available *@
@functions
{
    private new static HtmlHelper<object> Html
    {
        get { return ((WebViewPage)CurrentPage).Html; }
    }

    private static UrlHelper Url
    {
        get { return ((WebViewPage)CurrentPage).Url; }
    }
}
```

Read Razor online: https://riptutorial.com/asp-net-mvc/topic/5266/razor

Chapter 28: Routing

Introduction

Routing is how ASP.NET MVC matches a URI to an action. Routing module is responsible for mapping incoming browser requests to particular MVC controller actions.

MVC 5 supports a new type of routing, called attribute routing. As the name implies, attribute routing uses attributes to define routes. Attribute routing gives you more control over the URIs in your web application.

Examples

Custom Routing

Custom routing provides specialized need of routing to handle specific incoming requests.

In order to defining custom routes, keep in mind that the order of routes that you add to the route table is important.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    // this is an advanced custom route
    // you can define custom URL with custom parameter(s) point to certain action method
    routes.MapRoute(
    "CustomEntry", // Route name
    "Custom/{entryId}", // Route pattern
    new { controller = "Custom", action = "Entry" } // Default values for defined parameters
above
    );

    // this is a basic custom route
    // any custom routes take place on top before default route
    routes.MapRoute(
    "CustomRoute", // Route name
    "Custom/{controller}/{action}/{id}", // Route pattern
    new { controller = "Custom", action = "Index", id = UrlParameter.Optional } // Default
values for defined parameters above
    );

    routes.MapRoute(
    "Default", // Route name
    "{controller}/{action}/{id}", // Route pattern
    new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Default
values for defined parameters above
    );
}
```

`controller` and `action` names are reserved. By default MVC maps `{controller}` part of the URL to the class `<controller>Controller`, and then looks for a method with the name `<action>` without adding any suffixes.

Though it may be tempting to create a family of routes using `{controller}/{action}/{parameter}` template consider that by doing this you disclose the structure of your application and make URLs somewhat brittle because changing the name of the controller changes the route and breaks the links saved by the user.

Prefer explicit route setting:

```
routes.MapRoute(
    "CustomRoute", // Route name
    "Custom/Index/{id}", // Route pattern
    new { controller = "Custom", action = nameof(CustomController.Index), id =
UrlParameter.Optional }
);
```

(you cannot use `nameof` operator for controller name as it will have additional suffix `Controller`) which must be omitted when setting controller name in the route.

Adding custom route in Mvc

User can add custom route, mapping an URL to a specific action in a controller. This is used for search engine optimization purpose and make URLs readable.

```
routes.MapRoute(
  name: "AboutUsAspx", // Route name
  url: "AboutUs.aspx",  // URL with parameters
  defaults: new { controller = "Home", action = "AboutUs", id = UrlParameter.Optional }  //
Parameter defaults
);
```

Attribute routing in MVC

Along with classic way of route definition MVC WEB API 2 and then MVC 5 frameworks introduced `Attribute routing`:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        // This enables attribute routing and must go  before other routes are added to the
routing table.
        // This makes attribute routes have higher priority
        routes.MapMvcAttributeRoutes();
    }
}
```

For routes with same prefix inside a controller, you can set a common prefix for entire action methods inside the controller using `RoutePrefix` attribute.

```
[RoutePrefix("Custom")]
public class CustomController : Controller
{
    [Route("Index")]
    public ActionResult Index()
    {
        ...
    }
}
```

`RoutePrefix` is optional and defines the part of the URL which is prefixed to all the actions of the controller.

If you have multiple routes, you may set a default route by capturing action as parameter then apply it for entire controller unless specific `Route` attribute defined on certain action method(s) which overriding the default route.

```
[RoutePrefix("Custom")]
[Route("{action=index}")]
public class CustomController : Controller
{
    public ActionResult Index()
    {
        ...
    }

    public ActionResult Detail()
    {
        ...
    }
}
```

Routing basics

When you request the url `yourSite/Home/Index` through a browser, the routing module will direct the request to the `Index` action method of `HomeController` class. How does it know to send the request to this specific class's specific method ? there comes the RouteTable.

Every application has a route table where it stores the route pattern and information about where to direct the request to. So when you create your mvc application, there is a default route already registered to the routing table. You can see that in the `RouteConfig.cs` class.

```
public static void RegisterRoutes(RouteCollection routes)
{
   routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
   routes.MapRoute("Default", "{controller}/{action}/{id}",
             new { controller = "Home", action = "Index", id = UrlParameter.Optional });
}
```

You can see that the entry has a name and a template. The template is the route pattern to be checked when a request comes in. The default template has `Home` as the value of the controller url segment and `Index` as the value for the action segment. That means, if you are not explicitly passing a controller name and action in your request, it will use these default values. This is the reason you get the same result when you access `yourSite/Home/Index` and `yourSite`

You might have noticed that we have a parameter called id as the last segment of our route pattern. But in the defaults, we specify that it is optional. That is the reason we did not have to specify the id value int he url we tried.

Now, go back to the Index action method in HomeController and add a parameter to that

```
public ActionResult Index(int id)
{
     return View();
}
```

Now put a visual studio breakpoint in this method. Run your project and access `yourSite/Home/Index/999` in your browser. The breakpoint will be hit and you should be able to see that the value 999 is now available in the `id` parameter.

Creating a second Route pattern

Let's say we would like a set it up so that the same action method will be called for a different route pattern. We can do that by adding a new route definition to the route table.

```
public static void RegisterRoutes(RouteCollection routes)
{
   routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

   // New custom route definition added
   routes.MapRoute("MySpecificRoute",
    "Important/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional });

   //Default catch all normal route definition
   routes.MapRoute("Default", "{controller}/{action}/{id}",
             new { controller = "Home", action = "Index", id = UrlParameter.Optional });
}
```

The new definition i added has a pattern `Important/{id}` where id is again optional. That means when you request `yourSiteName\Important` or `yourSiteName\Important\888` , It will be send to the Index action of HomeController.

Order of route definition registration

The order of route registration is important. You should always register the specific route patterns before generic default route.

Catch-all route

Suppose we want to have a route that allows an unbound number of segments like so:

- http://example.com/Products/ (view all products)
- http://example.com/Products/IT
- http://example.com/Products/IT/Laptops

- http://example.com/Products/IT/Laptops/Ultrabook
- http://example.com/Products/IT/Laptops/Ultrabook/Asus
- etc.

We would need to add a route, normally at the end of the route table because this would probably catch all requests, like so:

```
routes.MapRoute("Final", "Route/{*segments}",
      new { controller = "Product", action = "View" });
```

In the controller, an action that could handle this, could be:

```
public void ActionResult View(string[] segments /* <- the name of the parameter must match the
name of the route parameter */)
{
    // use the segments to obtain information about the product or category and produce data
to the user
    // ...
}
```

Catch-all route for enabling client-side routing

It's a good practice to encode state of Single Page Application (SPA) in url:

```
my-app.com/admin-spa/users/edit/id123
```

This allows to save and share application state.
When user puts url into browser's address bar and hits enter server must ignore client-side part of the requested url. If you serve your SPA as a rendered Razor view (result of calling controller's action) rather than a static html file, you can use a catch-all route:

```
public class AdminSpaController
{
    [Route("~/admin-spa/{clienSidePart*}")]
    ActionResult AdminSpa()
    {
        ...
    }
}
```

In this case server returns just SPA, and it then initializes itself according to the route. This approach is more flexible as it does not depend on url-rewrite module.

Attribute Routing in Areas

For using Attribute Routing in areas, registering areas and `[RouteArea(...)]` definitions are required.

In `RouteConfig.cs` :

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapMvcAttributeRoutes();
        AreaRegistration.RegisterAllAreas();
    }
}
```

In a sample area controller attribute routing definition :

```
[RouteArea("AreaName", AreaPrefix = "AreaName")]
[RoutePrefix("SampleAreaController")]
public class SampleAreaController : Controller
{
    [Route("Index")]
    public ActionResult Index()
    {
        return View();
    }
}
```

For using `Url.Action` links in Areas :

```
@Url.Action("Index", "SampleAreaController", new { area = "AreaName" })
```

Read Routing online: https://riptutorial.com/asp-net-mvc/topic/1534/routing

Chapter 29: T4MVC

Introduction

T4MVC is a T4 template that generates strongly-typed helpers for use in MVC Routing mechanisms, as opposed to magic strings. T4MVC will detect the various controllers, actions, and views, and create references to those views, making compile-time errors in the event that an attempt to Route or access a View is invalid.

Examples

Calling an Action

In MVC, there are some scenerios where you want to specify an action for routing purposes, either for a link, form action, or a redirect to action. You can specify an action via the MVC namespace.

When given a Controller, such as `HomeController`:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ...
    }
    public ActionResult MyAction()
    {
        ...
    }
    public ActionResult MyActionWithParameter(int parameter)
    {
        ...
    }
}
```

T4MVC will generate an inherited controller that overrides the action. This override will set the route data properly so that MVC's `UrlHelper` will generate the proper Url. You can call this method and pass it into the various methods for `UrlHelper`. The examples below assume the default MVC route is being used:

Link

To generate an `a` tag with the specified text:

```
@Html.ActionLink("Link Text", MVC.Home.Index() )
//result: <a href="/">Link Text</a>
@Html.ActionLink("Link Text", MVC.Home.MyAction() )
//result: <a href="/Home/MyAction">Link Text</a>
//T4MVC also allows you to specify the parameter without creating an anonymous object:
@Html.ActionLink("Link Text", MVC.Home.MyActionWithParameter(1) )
//result: <a href="/Home/MyActionWithParameter/1">Link Text</a>
```

To generate a url:

```
@Url.Action( MVC.Home.Index() )
//result: /
@Url.Action("Link Text", MVC.Home.MyAction() )
//result: /Home/MyAction
@Url.Action("Link Text", MVC.Home.MyActionWithParameter(1) )
//result: /Home/MyActionWithParameter/1
```

Notice that T4MVC follows the same rules as MVC Routing - it won't specify default route variables, so that the `Index` action on the `HomeController` doesn't generate `/Home/Index` but instead the perfectly valid and abbreviated form of `/`.

Form Method

To generate a `form` tag with the correct `action` specified:

```
@Html.BeginForm( MVC.Home.Index(), FormMethod.Get /* or FormMethod.Post */ )
{
    //my form
}
//result:
<form action="/" method="GET">
    //my form
</form>
@Html.BeginForm( MVC.Home.MyActionWithParameter(1), FormMethod.Get /* or FormMethod.Post */ )
{
    //my form
}
//result:
<form action="/Home/MyActionWithParameter/1" method="GET">
    //my form
</form>
```

Redirect To Action

When in a controller, you may want to redirect to an action from the current action. This can be done, likeso:

```
public class RedirectingController : Controller
{
    public ActionResult MyRedirectAction()
    {
        ...
        return RedirectToAction( MVC.Redirecting.ActionToRedirectTo() );
        //redirects the user to the action below.
    }
    public ActionResult ActionToRedirectTo()
    {
        ...
    }
}
```

Read T4MVC online: https://riptutorial.com/asp-net-mvc/topic/9147/t4mvc

Chapter 30: Using Multiple Models In One View

Introduction

The main focus of this topic using multiple model class in view layer of MVC

Examples

Using multiple model in a view with dynamic ExpandoObject

ExpandoObject (the `System.Dynamic` namespace) is a class that was added to the `.Net Framework 4.0`. This class allows us to dynamically add and remove properties onto an object at runtime. By using Expando object we can add our model classes into dynamically created Expando object. Following example explains how we can use this dynamic object.

Teacher and Student Model:

```
public class Teacher
{
    public int TeacherId { get; set; }
    public string Name { get; set; }
}

public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
}
```

Teacher and Student List Methods:

```
public List<Teacher> GetTeachers()
{
    List<Teacher> teachers = new List<Teacher>();
    teachers.Add(new Teacher { TeacherId = 1, Name = "Teacher1" });
    teachers.Add(new Teacher { TeacherId = 2, Name = "Teacher2" });
    teachers.Add(new Teacher { TeacherId = 3, Name = "Teacher3" });
    return teachers;
}

public List<Student> GetStudents()
{
    List<Student> students = new List<Student>();
    students.Add(new Student { StudentId = 1, Name = "Student1"});
    students.Add(new Student { StudentId = 2, Name = "Student2"});
    students.Add(new Student { StudentId = 3, Name = "Student3"});
    return students;
}
```

Controller (Using Dynamic Model):

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Message = "Hello World";
        dynamic mymodel = new ExpandoObject();
        mymodel.Teachers = GetTeachers();
        mymodel.Students = GetStudents();
        return View(mymodel);
    }
}
```

View:

```
@using ProjectName ; // Project Name
@model dynamic
@{
    ViewBag.Title = "Home Page";
}
<h2>@ViewBag.Message</h2>

<h2>Teacher List</h2>

<table>
    <tr>
        <th>Id</th>
        <th>Name</th>
    </tr>
    @foreach (Teacher teacher in Model.Teachers)
    {
        <tr>
            <td>@teacher.TeacherId</td>
            <td>@teacher.Name</td>
        </tr>
    }
</table>

<h2>Student List</h2>

<table>
    <tr>
        <th>Id</th>
        <th>Name</th>
    </tr>
    @foreach (Student student in Model.Students)
    {
        <tr>
            <td>@student.StudentId</td>
            <td>@student.Name</td>
        </tr>
    }
</table>
```

Read Using Multiple Models In One View online: https://riptutorial.com/asp-net-mvc/topic/10144/using-multiple-models-in-one-view

Chapter 31: ViewData, ViewBag, TempData

Introduction

`ViewData` and `ViewBag` are used to transfer data from controller to view.

ViewData is nothing but a dictionary of objects and it is accessible by string as key.

ViewBag is very similar to ViewData. ViewBag is a dynamic property. ViewBag is just a wrapper around the ViewData.

TempData keeps data for the time of HTTP Request, which means that it holds data between two consecutive requests. TempData helps us to transfer data between controllers or between actions. Internally uses session.

Syntax

1. ViewData[key] = value;
2. ViewBag.Key = value;
3. TempData[key] = value;

Examples

What are ViewData, ViewBag, and TempData?

ViewData is the mechanism for a controller to provide data to the view it presents, without using a ViewModel. Specifically, ViewData is a dictionary which is available both in MVC action methods and views. You may use ViewData to transfer some data from your action method to the view returned by the action method.

Since it is a dictionary, you can use the dictionary like syntax to set and get data from it.

```
ViewData[key] = value; // In the action method in the controller
```

For example, If you want to pass a string message from your Index action method to your Index view `Index.cshtml`, you can do this.

```
public ActionResult Index()
{
   ViewData["Message"] = "Welcome to ASP.NET MVC";
   return View(); // notice the absence of a view model
}
```

To access this in your `Index.cshtml` view, you can simply access the ViewData dictionary with the key used in the action method.

```
<h2>@ViewData["Message"]</h2>
```

ViewBag is the dynamic equivalent of the untyped ViewData dictionary. It takes advantage of the C# `dynamic` type for syntactical sugar experience.

The syntax for setting some data to ViewBag is

```
ViewBag.Key = Value;
```

So if we want to pass the message string in our previous example using ViewBag, it will be

```
public ActionResult Index()
{
   ViewBag.Message = "Welcome to ASP.NET MVC";
   return View(); // not the absence of a view model
}
```

and in your Index view,

```
<h2>@ViewBag.Message</h2>
```

Data is not shared between the ViewBag and the ViewData. ViewData requires typecasting for getting data from complex data types and check for null values to avoid error where as View Bag does not require typecasting.

TempData can be used when you want to persist data between one http request and the next HTTP request only. The life of data stored in the TempDataDictionary ends after the second request. So TempData is useful in scenarios where you are following the PRG pattern.

```
[HttpPost]
public ActionResult Create(string name)
{
   // Create a user
   // Let's redirect (P-R-G Pattern)
   TempData["Message"] = "User created successfully";
   return RedirectToAction("Index");
}
public ActionResult Index()
{
  var messageFromPreviousCall = TempData["Message"] as String;
  // do something with this message
  // to do : Return something
}
```

When we do `return RedirectToAction("SomeActionMethod")`, the server will send a 302 response to the client(browser) with location header value set to the URL to "SomeActionMethod" and browser will make a totally new request to that. ViewBag / ViewData won't work in that case to share some data between these 2 calls. You need to use TempData in such scenarios.

TempData life cycle

Data saved to TempData is stored in the session and will be automatically removed at the end of the first request where the data is accessed. If never read, it will be kept until finally read or the session times out.

The typicaly usage looks like the following sequence (where each line is invoked from a different request):

```
//first request, save value to TempData
TempData["value"] = "someValueForNextRequest";

//second request, read value, which is marked for deletion
object value = TempData["value"];

//third request, value is not there as it was deleted at the end of the second request
TempData["value"] == null
```

This behavior can be further controller with the `Peek` and `Keep` methods.

- With `Peek` you can retrieve data stored in TempData without marking it for deletion, so data will still be available on a future request

  ```
  //first request, save value to TempData
  TempData["value"] = "someValueForNextRequest";

  //second request, PEEK value so it is not deleted at the end of the request
  object value = TempData.Peek("value");

  //third request, read value and mark it for deletion
  object value = TempData["value"];
  ```

- With `Keep` you can specify that a key that was marked for deletion should actually be retained. In this case retrieving the data and saving it from deletion requires 2 method calls:

  ```
  //first request, save value to TempData
  TempData["value"] = "someValueForNextRequest";

  //second request, get value marking it from deletion
  object value = TempData["value"];
  //later on decide to keep it
  TempData.Keep("value");

  //third request, read value and mark it for deletion
  object value = TempData["value"];
  ```

With this in mind, use `Peek` when you always want to retain the value for another request and use `Keep` when retaining the value depends on additional logic.

Read ViewData, ViewBag, TempData online: https://riptutorial.com/asp-net-mvc/topic/1286/viewdata--viewbag--tempdata

Chapter 32: Web.config Encryption

Examples

How to protect your Web.config file

It is a good practice to encrypt your Web.config file if you have sensitive information there, for example a connection string with password.

With the ASP.NET IIS Registration tool (Aspnet_regiis.exe) you can easily encrypt specific sections of Web.config file. A command with elevated privileges is required.

Example using DataProtectionConfigurationProvider. This provider uses DPAPI to encrypt and decrypt data:

```
aspnet_regiis.exe -pef "connectionStrings" c:\inetpub\YourWebApp -prov
"DataProtectionConfigurationProvider"
```

Example using RSAProtectedConfigurationProvider:

```
aspnet_regiis.exe -pef "connectionStrings" c:\inetpub\YourWebApp -prov
"RSAProtectedConfigurationProvider"
```

If you do not specify the -prov parameter it uses **RSAProtectedConfigurationProvider** as default. This provider is recommended for Web Farm scenarios.

To get **connectionStrings** section back to clear text:

```
aspnet_regiis.exe -pdf "connectionStrings" c:\inetpub\YourWebApp
```

More information about the aspnet_regiis.exe is avaiable on MSDN.

Read Web.config Encryption online: https://riptutorial.com/asp-net-mvc/topic/6373/web-config-encryption

Credits

S. No	Chapters	Contributors
1	Getting started with asp.net-mvc	Aaron Hudon, Adil Mammadov, Aditya Korti, Ameya Deshpande, Ashley Medway, Community, Hywel Rees, Rifaj, Shog9, Shyju, Supraj v, Syed Farjad Zia Zaidi, SztupY
2	Action filters	Andrei Rînea, Dawood Awan, juunas, Laurel, Lokesh_Ram, Tolga Evcimen
3	ActionResult	hasan, SlaterCodes, Tetsuya Yamamoto
4	Areas	Himaan Singh, Tetsuya Yamamoto
5	Asp.net mvc send mail	Ashiquzzaman, hasan, sGermosen
6	Automatic client-side validation from attributes	Slick86
7	Bundling and Minification	Ashley Medway, Beofett, hasan, Laurel, Lokesh_Ram, Paul DS, rageit, Rion Williams, Robban, Tetsuya Yamamoto, tmg
8	CRUD operation	EvenPrime, Krzyserious, PedroSouki, Tetsuya Yamamoto
9	Data annotations	abiNerd, dotnetom, dove, Edathadan Chief aka Arun, Ehsan Sajjad, Felipe Oriani, gunr2171, Karthikeyan, LaCartouche, mmushtaq, Ollie P, Rion Williams, SailajaPalakodeti, Stephen Muecke, Tetsuya Yamamoto, The_Outsider, tmg, Tsahi Asher
10	Dependency Injection	Adil Mammadov, Andrei Dragotoniu, Cà phê đen, Dave, PedroSouki
11	Display and Editor templates	Adnan Niloy, SailajaPalakodeti
12	Dockerization of ASP.NET Application	SUMIT LAHIRI
13	Error Logging	Andrus, Leandro Soares, Saineshwar
14	Html Helpers	Ashiquzzaman, Laurel, Lokesh_Ram, Pavel Pája Halbich, Peter Mortensen, QuantumHive, Tassadaque, Testing123, Tetsuya Yamamoto, The_Outsider, TheFallenOne
15	Html.AntiForgeryToken	Aaron Hudon, Andrei Rînea, Art, felickz, Hanno,

		Jakotheshadows, Joshua Leigh, Martin Costello, Minh Nguyen , Rion Williams, SailajaPalakodeti, SlaterCodes, viggity
16	Html.RouteLink	Jason Evans
17	Http Error Handling	scgough
18	IIS Rewrite Rules	SlaterCodes
19	jQuery Ajax Call With Asp MVC	Ashiquzzaman, hasan
20	Model binding	Adil Mammadov, Andrei Rînea, dove, Ehsan Sajjad, James Haug, Md Dinar, PedroSouki, rdans, Tolga Evcimen
21	Model validation	Aaron Hudon, Ankit Kumar Singh, GTown-Coder, hasan, Marimba, Nikunj Patel, Pavel Voronin, SlaterCodes, Stephen Muecke, The_Outsider, TheFallenOne, Vincentw
22	MVC Ajax Extensions	rll
23	MVC vs Web Forms	DiegoS, Houssam Hamdan, The_Outsider
24	Partial Views	Adnan Niloy, Ashiquzzaman, Edathadan Chief aka Arun, glacasa, Jason Evans, Laurel, Lokesh_Ram, Marimba, SailajaPalakodeti, The_Outsider
25	Razor	Aditya Korti, aeliusd, Anik Saha, Arendax, Ashley Medway, Big Fan, Brandon Wood, Braydie, Denis Elkhov, dove, James Haug, Julian, Kuldeep Prajapati, Lee Chengkai, lloyd, RamenChef, SadiRubaiyet, Sain Pradeep, Sandro, Thennarasan, Tim Coker, TKharaishvili, Tot Zam, usr
26	Routing	Alex Art., Andrei Rînea, chsword, Ciaran Bruen, Jarrod Dixon, Jason Evans, Karthikeyan, kkakkurt, Laurel, Lokesh_Ram, mstaessen, Pavel Voronin, SailajaPalakodeti, Sandro, Shyju, SlaterCodes, Stephen Muecke, Tetsuya Yamamoto, tmg, Tot Zam, user270576
27	T4MVC	James Haug
28	Using Multiple Models In One View	hasan, Travis Tubbs
29	ViewData, ViewBag, TempData	bzlm, Daniel J.G., IanB, Rion Williams, SailajaPalakodeti, Shyju, TheFallenOne, tmg
30	Web.config Encryption	glaubergft, Jack Spektor