MVC

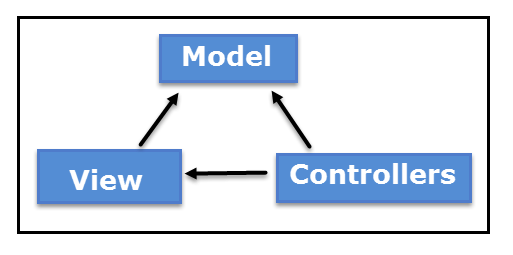
1 What is MVC?

**(MVC)** is an architectural pattern that separates an application into three main logical components: the **model**, the **view**, and the **controller**. Each of these components are built to handle specific development aspects of an application.

Component

**Model:** The Model component corresponds to all the data related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic related data.

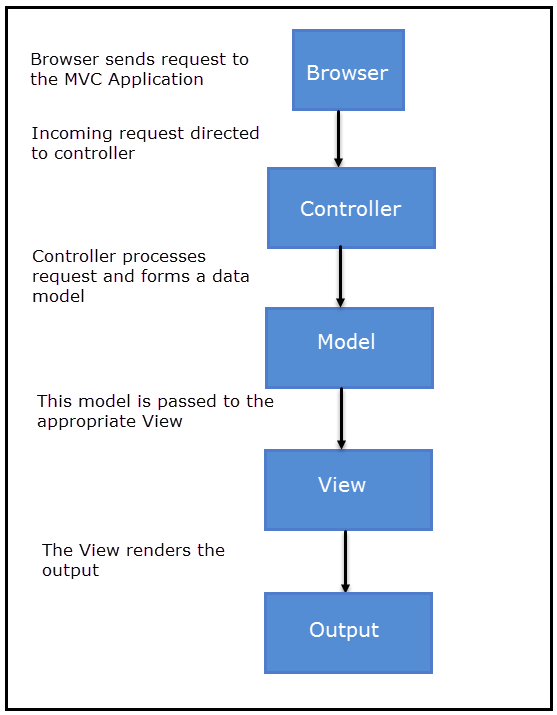
**View:** The View component is used for all the UI logic of the application. For example, the Customer view would include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

**Controller:** Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller would handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller would be used to view the Customer data. 

Feature:

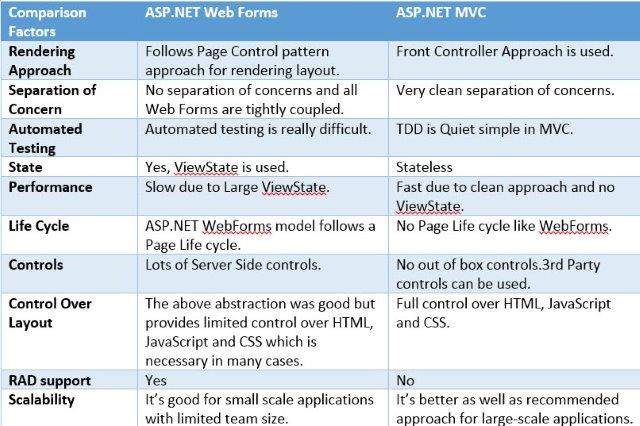
* Ideal for **developing complex but light weight applications**
* It **provides an extensible and pluggable framework which can be easily replaced and customized**. For example, if you do not wish to use the in-built Razor or ASPX View Engine, then you can use any other third-party view engines or even customize the existing ones.
* 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.
* 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.
* Supports all the existing vast ASP.NET functionalities such as Authorization and Authentication, Master Pages, Data Binding, User Controls, Memberships, ASP.NET Routing, etc.
* 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.

2 MVC Architecture

Let us have a look at how the execution of an MVC application takes place when certain request comes from the client. The diagram below shows the flow

* The client browser sends request to the MVC Application.
* Global.ascx receives this request and performs routing based on the URL of incoming request using the RouteTable, RouteData, UrlRoutingModule and MvcRouteHandler objects.
* This routing operation calls the appropriate controller and executes it using the IControllerFactory object and MvcHandler object's Execute method.
* The Controller processes the data using Model and invokes the appropriate method using ControllerActionInvoker object
* The processed Model is then passed to the View which in turn renders the final output.

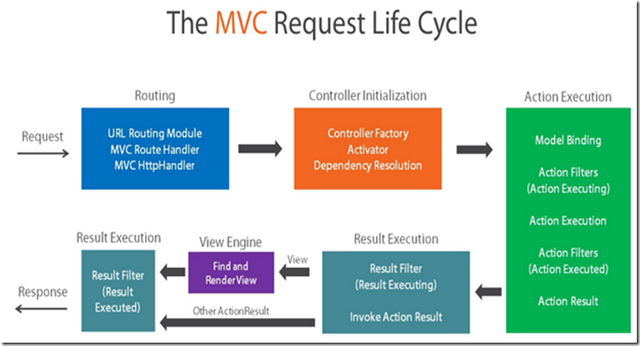
3 Compare Asp.net Web Form And Asp.net MVC



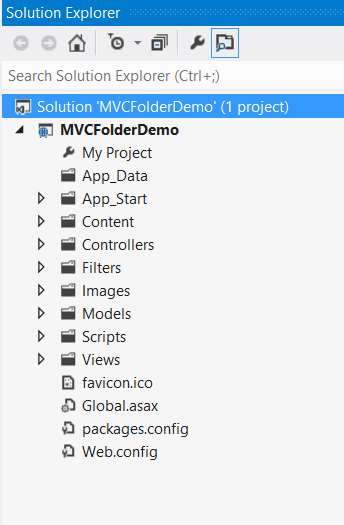
4 Asp.net MVC Page life cycle

The entry point of MVC Request life cycle is URL Routing module, the incoming request from IIS pipeline is handed over to URL Routing module which analyses the request and looks up Routing table to figure out which controller the incoming request maps to. Routing Table is a static container of routes defined in MVC application with corresponding controller action mapping. If the route is found in the routing table MVCRouteHandlerexecutes and brings the instance of MVCHttpHandler. Together they act as a gateway into the MVC Framework.

After the controller instance is created the next major step is to find and execute the corresponding action. A component called ActionInvoker finds and executes the action defined in routing table. Before the action method is called model bindings takes place which maps data from http request to action method parameters.



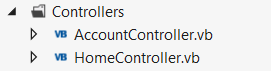
5 MVC Application folder structure



Controllers Folder

This folder will contain all the Controller classes. MVC requires name of all the controller files to end with Controller.

* In our example, the Controllers folder contains two class files: AccountController and HomeController.



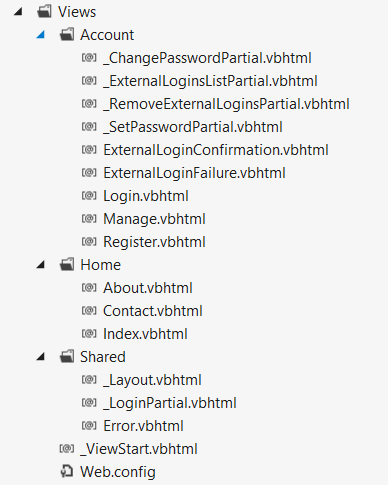
Models Folder

* This folder will contain all the Model classes which are used to work on application data.
* In our example, the Models folder contains AccountModels. You can open and look at the code in this file to see how the data model is created for managing accounts in our example.

mvc_models

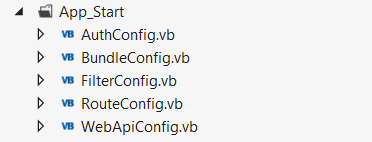
Views Folder

* This folder stores the HTML files related to application display and user interface.
* It contains one folder for each controller.
* In our example, you will see three sub-folders under Views namely Account, Home and Shared which contains html files specific to that view area.



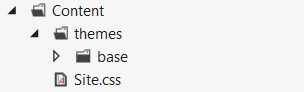
App\_Start Folder

* This folder contains all the files which are needed during the application load.
* For e.g., the RouteConfig file is used to route the incoming URL to the correct Controller and Action



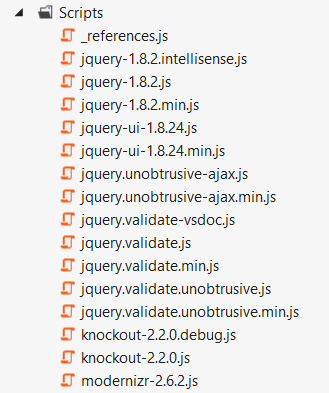
Content Folder

* This folder contains all the static files such as css, images, icons, etc.
* The Site.css file inside this folder is the default styling that the application applies.



Scripts Folder

* This folder stores all the JS files in the project. By default Visual Studio adds MVC, jQuery and other standard JS libraries.



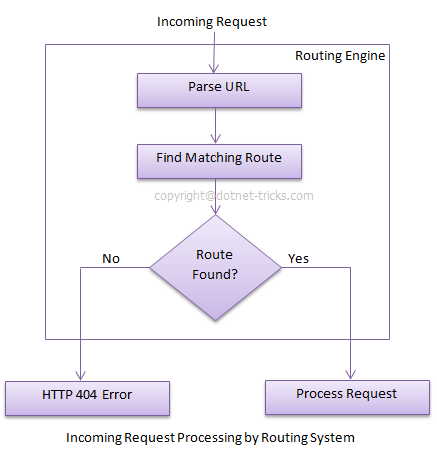
6 What is Routing in MVC…?

Basically, Routing is a pattern matching system that monitor the incoming request and figure out what to do with that request. At runtime, Routing engine use the Route table for matching the incoming request's URL pattern against the URL patterns defined in the Route table. You can register one or more URL patterns to the Route table at Application\_Start event.

Define Routing

1. **public static void RegisterRoutes(RouteCollection routes)**
2. **{**
3. **routes.MapRoute(**
4. **"Default", *// Route name***
5. **"{controller}/{action}/{id}", *// Route Pattern***
6. **new { controller = "Home", action = "Index", id = UrlParameter.Optional } *// Default values for above defined parameters***
7. **);**
8. **}**
9. **protected void Application\_Start()**
10. **{**
11. **RegisterRoutes(RouteTable.Routes);**
12. ***//To:DO***
13. **}**

When the routing engine finds a match in the route table for the incoming request's URL, it forwards the request to the appropriate controller and action. If there is no match in the route table for the incoming request's URL, it returns a 404 HTTP status code.



## Difference between Routing and URL Rewriting

Many developers compare routing to URL rewriting that is wrong. Since both the approaches are very much different. Moreover, both the approaches can be used to make SEO friendly URLs. Below is the main difference between these two approaches.

1. URL rewriting is focused on mapping one URL (new url) to another URL (old url) while routing is focused on mapping a URL to a resource.
2. Actually, URL rewriting rewrites your old url to new one while routing never rewrite your old url to new one but it map to the original route.

# 7 What is Route Table and how it works?

The RouteTable is a class that stores the URL routes for your application.

A RouteCollection provides a collection of route information to be used when mapping a URI to a controller action.

The RouteTable contains a property called Routes that will return a RouteCollection. The RouteTable uses a RouteCollection in order to store all the URL routing information it needs to accurately direct URI's to the correct controller action.

/// <summary>

/// Executed when the application starts.

/// </summary>

protected void Application\_Start()

{

RegisterRoutes(RouteTable.Routes);

}

# 8 what is attribute routing in mvc?

MVC5 and WEB API 2 supports a new type of routing, called attribute routing. In this routing, attributes are used to define routes. Attribute routing provides you more control over the URIs by defining routes directly on actions and controllers in your ASP.NET MVC application and WEB API.

#### **Note**

1. Attribute routing should configure before the convention-based routing.
2. When you combine attribute routing with convention-based routing, actions which do not have Route attribute for defining attribute-based routing will work according to convention-based routing. In above example Contact action will work according to convention-based routing.
3. When you have only attribute routing, actions which do not have Route attribute for defining attribute-based routing will not be the part of attribute routing. In this way they can’t be access from outside as a URI.

## Action level attribute routing

You can define routes at action level which apply to a specific action with in the controller.

1. **public class HomeController : Controller**
2. **{**
3. **[Route("users/{id:int:min(100)}")] *//route: /users/100***
4. **public ActionResult Index(int id)**
5. **{**
6. ***//TO DO:***
7. **return View();**
8. **}**
10. **[Route("users/about")] *//route" /users/about***
11. **public ActionResult About()**
12. **{**
13. **ViewBag.Message = "Your application description page.";**
14. **return View();**
15. **}**
17. ***//route: /Home/Contact***
18. **public ActionResult Contact()**
19. **{**
20. **ViewBag.Message = "Your contact page.";**
21. **return View();**
22. **}**
23. **}**

## When to use Attribute Routing

The convention-based routing is complex to support certain URI patterns that are common in RESTful APIs. But by using attribute routing you can define these URI patterns very easily.

**For example**, resources often contain child resources like Clients have orders, movies have actors, books have authors and so on. It’s natural to create URIs that reflects these relations like as: /clients/1/orders

This type of URI is difficult to create using convention-based routing. Although it can be done, the results don’t scale well if you have many controllers or resource types.

With attribute routing, it’s pretty much easy to define a route for this URI. You simply add an attribute to the controller action as:

1. **[Route("clients/{clientId}/orders")]**
2. **public IEnumerable GetOrdersByClient(int clientId)**
3. **{**
4. ***//TO DO***
5. **}**

## Enabling Attribute Routing in ASP.NET MVC

Enabling attribute routing in your ASP.NET MVC5 application is simple, just add a call to routes.MapMvcAttributeRoutes() method with in RegisterRoutes() method of RouteConfig.cs file.

1. **public class RouteConfig**
2. **{**
3. **public static void RegisterRoutes(RouteCollection routes)**
4. **{**
5. **routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");**
7. ***//enabling attribute routing***
8. **routes.MapMvcAttributeRoutes();**
9. **}**
10. **}**

## Defining Attribute Routing for Area in ASP.NET MVC

You can also define attribute routing for a controller that belongs to an area by using the RouteArea attribute. When you define attribute routing for all controllers with in an area, you can safely remove the AreaRegistration class for that area.

1. **[RouteArea("Admin")]**
2. **[RoutePrefix("menu")]**
3. **[Route("{action}")]**
4. **public class MenuController : Controller**
5. **{**
6. ***// route: /admin/menu/login***
7. **public ActionResult Login()**
8. **{**
9. **return View();**
10. **}**
12. ***// route: /admin/menu/products***
13. **[Route("products")]**
14. **public ActionResult GetProducts()**
15. **{**
16. **return View();**
17. **}**
19. ***// route: /categories***
20. **[Route("~/categories")]**
21. **public ActionResult Categories()**
22. **{**
23. **return View();**
24. **}**
26. **[Route("customers/{customerId}/orders")]**
27. **public IEnumerable GetOrdersByCustomer(int customerId)**
28. **{**
29. ***//TO DO***
30. **}**
31. **}**

# 9 What is difference between attribute and conventional routing?

Routing is a mechanism which is used to handle the incoming requests coming from browsers and it represent the particular action rather than any static or physical files. In ASP.NET, the Url hits any resources or files which physically exists but ASP.NET MVC Routing represents action. It is an approach to perform some action based on their definition defined in **RouteConfig**.**cs**.

**Convention Based Routing**

1. **public** **static** **void** RegisterRoutes(RouteCollection routes)
2. {
3. routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");
4. routes.IgnoreRoute("favicon.ico");
5. routes.IgnoreRoute("Content/img/dotnet-tutorial.ico");
6. routes.MapRoute("LogOff", "logOff", **new**
7. {
8. controller = "Account", action = "logoff"
9. });
10. routes.MapRoute("Register", "register", **new**
11. {
12. controller = "Account", action = "Register"
13. });
14. routes.MapRoute("AboutUs", "aboutus", **new**
15. {
16. controller = "Home", action = "AboutUs"
17. });
18. routes.MapRoute("IndividualArticlesPost", "articles/{categoryslug}/{url}", **new**
19. {
20. controller = "Articles", action = "View"
21. }, **new**
22. {
23. categoryslug = @ "\S+", url = @ "\S+"
24. });
25. routes.MapRoute("Default", // Route name
26. "{controller}/{action}/{id}", // URL with parameters
27. **new**
28. {
29. controller = "Home",
30. action = "Index",
31. id = UrlParameter.Optional
32. });
33. }

In the above code, you can see three things: “routes”, “**IgnoreRoute**” and “**MapRoute**”.

**Routes:** It is nothing but only a table which is a collection of routes defined in route table. When someone hits some url in the browser, application first check the existing routes table and match with routes.  
  
**IgnoreRoute:**It is also a collection of url that should be ignored by application.  
  
**MapRoute:**It is used to add new route into the route table.

# 10 Advantage of routing?

MVCgives you great control over how URLs are mapped to your controllers. It gives you the ability to define your URLs in a human readable SEO (Search Engine Optimization) friendly fashion, to remap old URLs to new functionality and side-by-side utilizes classic ASP.NETsites inside of MVC. It also results in hiding what kind of page a user is calling and what environment we are working in. Most of the new websites are following this and**it is important to understand that routing is not URL rewriting as routing** will have customizations and many attachments towards request/response.

One of the many factors frequently considered by search engines to determine the relevance of a particular page to a particular search term is whether or not the URL link itself includes a particular term. In a classic ASP.NET site for a magazine, you might have a URL that looks like www.abc.com/ViewArticle.aspx?id=123. This URL passes the ID number of the article to view, but the URL itself doesn't describe the content in any human readable way. If the URL instead was www.abc.com/MVC\_Routing/123 a human-or a web crawler-could read that and know that the article is about MVC Routing.

# 11 Advantages of mvc?

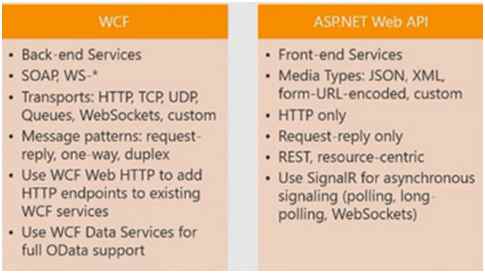
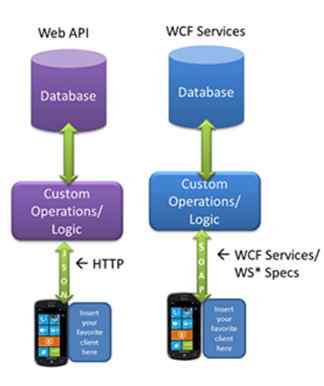
#### Advantages

* MVC provides straight separation concerns.
* MVC patronize code re-use, code regimentation, scalability and extensibility.
* It's very easier to test the code using unit testing with this pattern.
* It's very easy to change the visualization of application using this pattern.
* MVC provides facility to use multiple view.
* Main benefit in MVC is reduced the size of page controls and viewstates. The is not viewstate in MVC architecture. So this will optimize the page load speed.
* Using MVC easy to growth, we can use older version of views and controller in a common interface. We can use same model in multiple modules.
* MVC is very easy to integrate with JavaScript frameworks. For ex. AngularJs.
* MVC provides restful urls so it will very useful for SEO.

#### Disadvantages

* Developer should familiar with html and css.
* We can't drag and drop controls like web forms.
* Developer have knowledge of client side code and html code.
* MVC increasing complexity compare to web forms.

# 12 Difference between web API and wcf?

As you know their are lots of technologies available that allow you to create services which can be consumed by different client, the client can be web application, window application, mobile application, etc. Services can be http enabled or support multiple protocols.   
  
As you know today we will discuss only about the WCF and Web API.   
  
**WCF** stands for Windows Communication Foundation, created by Microsoft with .NET Framework 3.0. It provides us a platform where we can build secure, reliable service that can integrate across the platforms and interoperate smoothly.  
  
**Web API** is also a framework that is used to make HTTP services. As you know, now a days we are using mobiles, tablets, apps and different types of services, so Web API is a simple and reliable platform to create HTTP enabled services that can reach wide range of clients. It is used to create complete REST services.  
  
  
  
So, let's start the race between Web API and WCF and see at the last that who is the winner.  
  
**SOAP**  
  
**Simple Object Access Protocol [SOAP]** is a protocol invented by Microsoft that is used to create a structured format data or message that can transfer over the internet. SOAP protocol is the main foundation of WCF and uses XML format. Web Service creates WSDL [Web Services Description Language] file which provides the information about the service data, so WSDL file is used by the client to understand the methods available on the service.  
  
**REST**  
  
REST stands for **Representational State Transfer**. It is not a protocol like SOAP whether it is architecture design pattern that is used for building and creating Web Services.  
  
Web API used this pattern to build web services. Basically Web API is a resource driven architecture that exposes endpoint to the client. Web API is fully REST supported. It uses GET, PUT, POST, DELETE verbs to communicate with the clients.   
  
**Interoperability**  
  
RESTful services like Web API are simple and light weight and easy to access by any client which supports HTTP calls. It is not like to SOAP where client need to understand WSDL to implement the service.   
  
WSDL uses some extra configuration and require some additional logic that might not be available on many devices such as phones, smart TVs, Tablets, etc.  
  
As WCF supports many protocol so it can consumed by lots of client that are using different protocol such as TCP, MSMQ, etc. not only HTTP protocol to access service.   
  
  
  
**Configuration Overhead**  
  
Client needs to make more configurations as compared to Web API to access WCF Service. WCF uses more configurations such as endpoints, behaviors, etc to expose the service. But Web API has more standard way to customize the service which helps it to reach more clients.   
  
**Speed**  
  
As RESTful services use HTTP protocol to access client and HTTP request/response are smaller than a SOAP request, so Web API is the best choice if you are going to create a service for low bandwidth devices or mobile devices to access client. HTTP request/response is also more readable as compared to SOAP because it contains header, body, etc. which makes it complex.  
  
**Security**  
  
WCF service provides us high level security framework which provide enterprise level security. It uses WS-I standard to provide secure service.  
  
But Web API uses web standard security such as basic authentication, token authentication and for more complex such as OAuth; Web API provides more flexibility. Then use SOAP when you are going to authenticate with external services such as google, facebook, etc.  
  
  
  
**Winner?**  
  
So, finally to see the above point, we cannot decide which is better than the other. But I want to focus on the two points. Firstly, if you are going to create a service which would be used on different platforms, then go with WCF. Secondly, if you are creating internet service which is going to use external resource, then go with Web API.

# 13 What is filters?

Earlier, we used traditional try..catch block or globally catch exception in ASP.NET. Also ASP.NET has several traditional ways to handle authorization like session storage of user information and others. But in ASP.NET MVC has a much more smarter way to handle authorization and exception handling capability using filtering while requesting to Action through controller. I am going to describe a practical implementation for ASP.NET MVC application to implement authorization and exception. Eventually, I also discussed with the fundamental of filtering of ASP.NET MVC.

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

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

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

## Types of Filters

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

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

### **Authentication Filters**

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

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

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

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

### **Authorization Filters**

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

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

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

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

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

### **Action Filters**

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

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

### **Result Filters**

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

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

### **Exception Filters**

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

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

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

## Order of Filter Execution

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

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

## Configuring Filters

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

### **Global level**

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

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

### **Controller level**

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

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

### **Action level**

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

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

# 14 What is view model?

In ASP.NET MVC, ViewModel is a class that contains the fields which are represented in the strongly-typed view. It is used to pass data from controller to strongly-typed view.

## Key Points about ViewModel

1. ViewModel contain fields that are represented in the view (for LabelFor,EditorFor,DisplayFor helpers)
2. ViewModel can have specific validation rules using data annotations or IDataErrorInfo.
3. ViewModel can have multiple entities or objects from different data models or data source.

## Some Tips for using ViewModel

1. In ViewModel put only those fields/data that you want to display on the view/page.
2. Since view reperesents the properties of the ViewModel, hence it is easy for rendering and maintenance.
3. Use a mapper when ViewModel become more complex.

# 15 How many types of selectors in jquer?

# 18 Design patterns?

Design patterns have picked up a lot of importance off late and rightfully so. To define design patterns in simple words they are "popular solutions for common design problems". They are very helpful in designing architecture and they also increase ease of communication among the developers.

To start off GOF(Gang of Four) design patterns which are considered as the foundation of design patterns have been categorized into three sections namely creational which involves creation of objects, structural and behavioral patterns.

1 **Singleton Pattern:**

This is a very popular creational pattern which restricts a class to have only one instance.   
  
**Example:** "Prime Minister of India" is a singleton pattern as he/she has unique responsibilities and attributes.  
  
**Implementation:** A singleton class can't have a public constructor and it has to be sealed. The entry point to get the singleton instance would be a static method or a static property.

public sealed class PrimeMinister  
{   
private static PrimeMinister instance = new PrimeMinister();   
private PrimeMinister(){ }   
public static PrimeMinister Instance   
{   
get { return instance; }   
}  
}

2 **Abstract Factory Pattern:**

Abstract Factory is commonly known as factory pattern. In this pattern all the classes involved implement/realize the same interface and the compiler only knows that the created object implements the specific interface but not the object's type. This is very flexible when you need to make a creation decision at runtime depending on some aspect of object's behavior rather than it's type.   
  
***Example:*** Imagine that there is a rental store of cars, bikes, trucks, etc but you want to deliver a vehicle depending on the customer's choice.  
  
***Implementation:*** Programmatically you can make a run time decision using a switch statement to return a new instance of a class which implements the specific interface depending on the customer's choice. As you can see in the constructor of "Journey" class we are getting vehicletype of the current customer and passing it to the VehicleSupplier class's static method "GetVehicle". VehicleSupplier relies on a switch statement to return the vehicle needed, so if you observe here compiler just knows that newly created object implements the "IVehicle" interface but not if it's a car, truck or a bike upfront.  
  
public class Journey   
{   
public IVehicle rentedVehicle;   
public Journey(string customerID)   
{   
VehicleType vType = Customer.GetVehicleType(customerID);   
/\*Here is the late binding. Compiler doesn't know the type of the object except that it implements the IVehicle interface\*/  
rentedVehicle = VehicleSupplier.GetVehicle(vType);   
}   
//Method for beginning the journey   
public void BeginJourney()   
{   
if(rentedVehicle != null)   
{   
rentedVehicle.Drive();   
}   
}   
//Method for parking the vehicle   
public void ParkTheVehicle()   
{   
if(rentedVehicle != null)   
{   
rentedVehicle.Park();  
}   
}   
}  
//The class which returns the new vehicle instance depending on the //vehicle type  
public class VehicleSupplier  
{  
public static IVehicle GetVehicle(VehicleType vType)  
{  
switch(vType)  
{  
case VehicleType.CAR:  
return new Car();  
case VehicleType.TRUCK:  
return new Truck();  
case VehicleType.BIKE:  
return new Bike();  
}   
return null;  
}   
}   
//The interface which will be implemented by Car, Truck and Bike //classes  
public interface IVehicle  
{  
void Drive();  
void Park();  
}  
//enum of the vehicle types  
public enum VehicleType{CAR=1,TRUCK,BIKE};

**Structural Section:**  
Structural patterns depend on the structure of the classes involved. There are many structural patterns like Adapter, Builder, Decorator, etc. I will explain adapter pattern with an example here.  
  
**Adapter Pattern:**   
  
By converting the interface of one class into another interface which is expected by the clients we can make incompatible classes work together and this is the adapter pattern.  
  
***Example:*** Consider that you bought a gadget which expects 110v power supply but you are getting 240v supply, you need to get a power adapter in this case for converting 110v to 240v. This is a perfect example for the adapter pattern. In the software context you may need to have adapters for converting data from legacy systems like AS400 to XML or some other format for export purposes.  
  
***Implementation:*** As you can see the "StepUpStepDownPowerAdapter" object is used for converting power supply of 240V to 110V in the "TestUSGadget" class. The "USGadget" class defines the required voltage of the gadget and other properties. The converted power supply is given to the instance of "USGadget".   
  
/// Class for testing the gadget   
public class TestUSGadget  
{  
public static void Test()  
{  
USGadget gadget = new USGadget();  
//Indian or European power supply  
PowerSupply supply = new PowerSupply(VoltageTypes.V240);  
if(gadget.ExpectedVoltage != supply.Voltage)  
{  
StepUpStepDownPowerAdapter adapter = new StepUpStepDownPowerAdapter();  
//Getting the converted power  
supply = adapter.Convert(supply,gadget.ExpectedVoltage);  
gadget.Power = supply;  
gadget.Start();  
}  
}  
}  
public class StepUpStepDownPowerAdapter  
{  
public StepUpStepDownPowerAdapter()  
{  
}  
//Method for coverting voltages  
public PowerSupply Convert(PowerSupply supply,VoltageTypes convertToVoltage)  
{  
if(supply == null) return supply;  
//Convert iff the voltage is not in expected way  
if(supply.Voltage != convertToVoltage)  
supply.Voltage = convertToVoltage;  
return supply;  
}  
}   
/// The power supply class   
public class PowerSupply  
{  
VoltageTypes voltageType;  
//There will be other properties as well  
public PowerSupply(VoltageTypes vType)  
{  
voltageType = vType;  
}  
public VoltageTypes Voltage  
{  
get  
{  
return voltageType;  
}  
set  
{  
voltageType = value;  
}  
}  
}  
//Voltage Types enum  
public enum VoltageTypes{V110,V240};  
//Gadget which expects 110V  
public class USGadget  
{   
VoltageTypes reqVoltage;  
PowerSupply supply;  
public USGadget()  
{  
reqVoltage = VoltageTypes.V110;  
}  
public VoltageTypes ExpectedVoltage  
{  
get  
{  
return reqVoltage;  
}  
}  
public PowerSupply Power  
{  
get   
{  
return supply;  
}  
set  
{  
supply = value;  
}  
}  
public void Start()  
{  
}  
}

**Behavioral Section:**  
Behavioral Patterns define the behavior of the classes involved. The popular behavioral patterns include Chain of Responsibilities,Interpreter, Mediator, Iterator, Observer, etc. I will explain Observer pattern here.  
  
Observer pattern involves a one-many dependency between objects where a change in an object(subject) needs to be notified to all it's dependents(observers).  
  
***Example:*** Consider a scenario where a job posting at some company got multiple applications. Whenever the job status changes (filled, removed or suspended) all the applicants of the job should be notified. In this case job object is subject and all the applicants are observers.   
  
***Implementation:*** As you can see below "Job" is the subject class and all applicants of that particular job are observers. Job class has "Add" and "Remove" methods for adding and removing applicants to it's list. Whenever job status changes all the applicant objects would be notified through Notify method which in turn calls the "Update" method of the applicant object.   
  
/// This the subject in the observer pattern.   
public class Job  
{  
private ArrayList applicants;  
private JobStatus statusOfJob;  
public Job()  
{  
applicants = new ArrayList();  
}  
public void Add(Applicant candidate)  
{  
applicants.Add(candidate);  
}  
public void Remove(Applicant candidate)  
{  
applicants.Remove(candidate);  
}  
public void Notify()  
{  
foreach (Applicant candidate in applicants)  
{  
candidate.Update(this);  
}  
}  
public JobStatus Status  
{  
get  
{  
return statusOfJob;  
}  
set  
{  
statusOfJob = value;  
Notify();  
}  
}  
}  
//Jobstatus enumerator  
public enum JobStatus{FILLED,SUSPENDED,REMOVED};  
/// This is Observer.   
public class Applicant  
{  
//declare variables  
string fname;  
string lname;  
string emailID;  
string phoneNo;  
public Applicant()  
{  
//  
// TODO: Add constructor logic here  
//  
}  
#region Properties for exposing the member variables  
#endregion  
public void Update(Job appliedJob)  
{  
switch(appliedJob.Status)  
{  
case JobStatus.FILLED:  
//Do something like sending email, //updating database, etc  
break;  
case JobStatus.REMOVED:  
//Do something like sending email, //updating database, etc  
break;  
case JobStatus.SUSPENDED:  
//Do something like sending email, //updating database, etc  
break;  
}  
//Your own functionality  
//End Of Functionality  
}  
}

# 19 Explain process of applying validation and validating models?

In MVC Model are the class which contains domain classes and the validations. Validation plays a core part in a Model.

namespace SampleApplication.Models

{

public class UserModel

{

public string Name { get; set; }

public string Email { get; set; }

public string MobileNo { get; set; }

}

}

Basically, this article contains the following approaches:

1. By using Data Annotations
2. By using ModelState object (Explicit Model Validation)
3. By using jQuery Validation plug-in
4. By using Fluent Validation package

### **By using Data Annotations**

In ASP.NET MVC one of the best ways to validate a model is by using Data Annotations. It has a set of attributes and classes defined in the System.ComponentModel.DataAnnotations assembly. Data Annotations allow us to decorate model classes with metadata. This metadata describes a set of rules that are used to validate a property.

For using DataAnnotations, we need to add a reference to the System.ComponentModel.DataAnnotations assembly to model class.

Hide   Copy Code

using System.ComponentModel.DataAnnotations;

namespace SampleApplication.Models

{

public class UserModel

{

[Required(ErrorMessage = "Please Enter Name")]

[Display(Name = "Name")]

public string Name { get; set; }

[Required(ErrorMessage = "Please Enter Email Address")]

[Display(Name = "Email")]

[RegularExpression(@"^([a-zA-Z0-9\_\.\-])+\@(([a-zA-Z0-9\-])+\.)+([a-zA-Z0-9]{2,4})+$",

ErrorMessage = "Please Enter Correct Email Address")]

public string Email { get; set; }

[Required(ErrorMessage = "Please Enter Mobile No")]

[Display(Name = "Mobile")]

[StringLength(10, ErrorMessage = "The Mobile must contains 10 characters", MinimumLength = 10)]

public string MobileNo { get; set; }

}

}

Now, we need to enable validationsummary and to add validation message to view page.

The @Html.ValidationSummary() method shows all the validation errors of a model in view page.

@Html.ValidationSummary(true)

@model SampleApplication.Models.UserModel

@{

ViewBag.Title = "User Details";

}

<br /><br />

@using (Html.BeginForm())

{

@Html.ValidationSummary(true)

<fieldset>

<legend>Form</legend>

#### UserController.cs

namespace SampleApplication.Controllers

{

public class UserController : Controller

{

public ActionResult User()

{

return View();

}

[HttpPost]

public ActionResult ServerMeta(RegistrationMetaModel mRegister)

{

if (ModelState.IsValid)

{

return View("Completed");

}

else

{

return View();

}

}

}

}

### **By using ModelState object (Explicit Model Validation)**

If you want to validate a model explicitly, then it is the best way to use ModelState Object within ActionMethod in controller. Just add an error message on ModelState with (key, value) pair manner and that message will be shown on the view whenever the data will not be validated in the model.

Model class will be shown as below:

Hide   Copy Code

namespace SampleApplication.Models

{

public class UserModel

{

public string Name { get; set; }

public string Email { get; set; }

public string MobileNo { get; set; }

}

}

Now we are going to add errormessage to ModelState object.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System.Text.RegularExpressions;

using System.Web.Mvc;

using SampleApplication.Models;

namespace SampleApplication.Controllers

{

public class UserController : Controller

{

public ActionResult User()

{

return View();

}

[HttpPost]

public ActionResult User(UserModel model)

{

if (string.IsNullOrEmpty(model.Name))

{

ModelState.AddModelError("Name", "Please Enter Name");

}

if (!string.IsNullOrEmpty(model.Email))

{

string emailRegex = @"^([a-zA-Z0-9\_\.\-])+\@(([a-zA-Z0-9\-])+\.)+([a-zA-Z0-9]{2,4})+$";

Regex re = new Regex(emailRegex);

if (!re.IsMatch(model.Email))

{

ModelState.AddModelError("Email", "Please Enter Correct Email Address");

}

}

else

{

ModelState.AddModelError("Email", "Please Enter Email Address");

}

if (string.IsNullOrEmpty(model.MobileNo))

{

ModelState.AddModelError("MobileNo", "Please enter your mobile no");

}

if (ModelState.IsValid)

{

return View("Completed");

}

else

{

return View();

}

}

}

}

The view User.cshtml will display the validation message is as follows:

Hide   Copy Code

@model SampleApplication.Models.UserModel

@{

ViewBag.Title = "User Details";

}

<br /><br />

@using (Html.BeginForm())

{

@Html.ValidationSummary(true)

<fieldset>

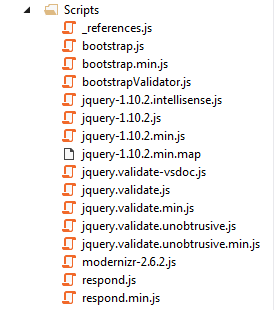
<legend>Form</legend>

### **By Using jQuery Validation Plug-in**

The client side validation helps the application to reduce server loads and also it saves time of end user. So for client side validation, we are using following jQuery validation plug-in.

1. jquery.validate.min.js
2. jquery.validate.unobtrusive.min.js

Just add these files to your application if there are not present inside Scripts folder.



First of all, we need to add the following config setting in web.config file.

Hide   Copy Code

<appSettings>

<add key="ClientValidationEnabled" value="true" />

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

</appSettings>

In MVC4, the above appsetting is by default enabled in config file. But we need to ensure that the ClientValidationEnabled and UnobtrusiveJavaScriptEnabled value are true before doing client side validation.

Now, we can write clientside JavaScript code to validate data or by using ~~DataAnnotations~~ attributes in Model.

Let's take advantage of DataAnnotations API in client side validation. So add DataAnnotations to model.

#### UserModel.cs

Hide   Copy Code

using System.ComponentModel.DataAnnotations;

namespace SampleApplication.Models

{

public class UserModel

{

[Required(ErrorMessage = "Please Enter Name")]

[Display(Name = "Name")]

public string Name { get; set; }

[Required(ErrorMessage = "Please Enter Email Address")]

[Display(Name = "Email")]

[RegularExpression(@"^([a-zA-Z0-9\_\.\-])+\@(([a-zA-Z0-9\-])+\.)+([a-zA-Z0-9]{2,4})+$",

ErrorMessage = "Please Enter Correct Email Address")]

public string Email { get; set; }

[Required(ErrorMessage = "Please Enter Mobile No")]

[Display(Name = "Mobile")]

[StringLength(10, ErrorMessage = "The Mobile must contains 10 characters", MinimumLength = 10)]

public string MobileNo { get; set; }

}

}

#### UserController.cs

Hide   Copy Code

namespace SampleApplication.Controllers

{

public class UserController : Controller

{

public ActionResult User()

{

return View();

}

[HttpPost]

public ActionResult ServerMeta(RegistrationMetaModel mRegister)

{

if (ModelState.IsValid)

{

return View("Completed");

}

else

{

return View();

}

}

}

}

During MVC project creation, the application adds a BundleConfig.cs file under App\_Start folder. This class is used to include scripts and styles in bundle.

Include validation plug-in to BundleCollection object in BundleConfig.cs file.

The bundle code is:

Hide   Copy Code

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

"~/Scripts/jquery.validate\*"));

The page template (\_Layout.cshtml) created for a new MVC 4 project contains this code at the end of the page which does not pull in the jQuery validation library.

Hide   Copy Code

@RenderSection("scripts", required: false)

In order to "turn on" jQuery validation for a single view, the following code needs to reside at the end of view.

Hide   Copy Code

@section Scripts {

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

}

#### User.cshtml

Hide   Copy Code

@model SampleApplication.Models.UserModel

@{

ViewBag.Title = "User Details";

}

<br /><br />

@using (Html.BeginForm())

{

@Html.ValidationSummary(true)

<fieldset>

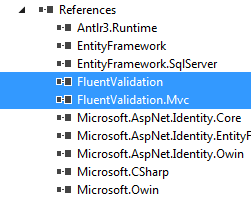
<legend>Form</legend>

### **By Using Fluent Validation Package**

FluentValidation library provides some easy way to unit test validation rules, one can split the validation rules completely from the underlying model. Specially, it will be helpful when you need to inject dependencies into your validation rules.

Now we are going to use FluentValidation class library by replacing DataAnnotations from Model.

Fluent Validation is available as a Nuget package, so search FluentValidation.MVC4 and install from Nuget Package Manager. After installing, you can find two new assemblies added to your application named as Fluent Validation and Fluent Validation.Mvc.



Add a new class called UserModelValidator and place all the validation rules.

Hide   Copy Code

namespace SampleApplication.Models

{

public class UserModelValidator : AbstractValidator<UserModel>

{

public UserModelValidator()

{

RuleFor(x => x.Name)

.NotNull();

RuleFor(x => x.Email)

.NotNull()

.EmailAddress();

RuleFor(x => x.MobileNo)

.NotNull()

.Length(6, 10);

}

}

?}

#### Existing Model with DataAnnotations

Hide   Copy Code

using System.ComponentModel.DataAnnotations;

namespace SampleApplication.Models

{

public class UserModel

{

[Required(ErrorMessage = "Please Enter Name")]

[Display(Name = "Name")]

public string Name { get; set; }

[Required(ErrorMessage = "Please Enter Email Address")]

[Display(Name = "Email")]

[RegularExpression(@"^([a-zA-Z0-9\_\.\-])+\@(([a-zA-Z0-9\-])+\.)+([a-zA-Z0-9]{2,4})+$", ErrorMessage = "Please Enter Correct Email Address")]

public string Email { get; set; }

[Required(ErrorMessage = "Please Enter Mobile No")]

[Display(Name = "Mobile")]

[StringLength(10, ErrorMessage = "The Mobile must contains 10 characters", MinimumLength = 10)]

public string MobileNo { get; set; }

}

}

Now link UserModelValidator validation class to the UserModel class by specifying it in the Validatorattribute as follows:

Hide   Copy Code

[FluentValidation.Attributes.Validator(typeof(UserModelValidator))]

public class UserModel

{

[Display(Name = "Name")]

public string Name { get; set; }

[Display(Name = "Email")]

public string Email { get; set; }

[Display(Name = "Mobile")]

public string Mobileno { get; set; }

}

The last step is to call the FluentValidationModelValidatorProvider.Configure() method inside your global.asax file.

Hide   Copy Code

namespace SampleApplication

{

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

AuthConfig.RegisterAuth();

FluentValidationModelValidatorProvider.Configure();

}

}

}

## Adding Validation Rules to the Movie Model

You'll begin by adding some validation logic to the Movie class.

Open the Movie.cs file. Notice the [System.ComponentModel.DataAnnotations](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx) namespace does not contain System.Web.  DataAnnotations provides a built-in set of  validation attributes that you can apply declaratively to any class or property.  (It also contains formatting attributes like [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatype.aspx) that help with formatting and don't provide any validation.)

Now update the Movie class to take advantage of the built-in [Required](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.requiredattribute.aspx), [StringLength](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.stringlengthattribute.aspx),  [RegularExpression](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.regularexpressionattribute.aspx), and [Range](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.rangeattribute.aspx) validation attributes. Replace the Movie  class with the following:

public class Movie

{

    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]

    public string Title { get; set; }

    [Display(Name = "Release Date")]

    [DataType(DataType.Date)]

    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]

    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]\*$")]

    [Required]

    [StringLength(30)]

    public string Genre { get; set; }

    [Range(1, 100)]

    [DataType(DataType.Currency)]

    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]\*$")]

    [StringLength(5)]

    public string Rating { get; set; }

}

# 20 What Solid principles?

**Intro to SOLID principles**  
  
SOLID principles are the design principles that enable us to manage with most of the software design problems. Robert C. Martin compiled these principles in the 1990s. These principles provide us ways to move from tightly coupled code and little encapsulation to the desired results of loosely coupled and encapsulated real needs of a business properly. SOLID is an acronym of the following.

* **S: Single Responsibility Principle (SRP)**
* **O: Open closed Principle (OSP)**
* **L: Liskov substitution Principle (LSP)**
* **I: Interface Segregation Principle (ISP)**
* **D: Dependency Inversion Principle (DIP)**

**S: Single Responsibility Principle (SRP)**  
  
SRP says "Every software module should have only one reason to change".  
  
  
  
This means that every class, or similar structure, in your code should have only one job to do. Everything in that class should be related to a single purpose. Our class should not be like a Swiss knife wherein if one of them needs to be changed then the entire tool needs to be altered. It does not mean that your classes should only contain one method or property. There may be many members as long as they relate to the single responsibility.  
  
The Single Responsibility Principle gives us a good way of identifying classes at the design phase of an application and it makes you think of all the ways a class can change. A good separation of responsibilities is done only when the full picture of how the application should work. Let us check this with an example.

1. **public** **class** UserService
2. {
3. **public** **void** Register(**string** email, **string** password)
4. {
5. **if** (!ValidateEmail(email))
6. **throw** **new** ValidationException("Email is not an email");
7. var user = **new** User(email, password);
9. SendEmail(**new** MailMessage("mysite@nowhere.com", email) { Subject="HEllo foo" });
10. }
11. **public** **virtual** **bool** ValidateEmail(**string** email)
12. {
13. **return** email.Contains("@");
14. }
15. **public** **bool** SendEmail(MailMessage message)
16. {
17. \_smtpClient.Send(message);
18. }
19. }

It looks fine, but it is not following SRP. The SendEmail and ValidateEmail methods have nothing to do within the UserService class. Let's refract it.

1. **public** **class** UserService
2. {
3. EmailService \_emailService;
4. DbContext \_dbContext;
5. **public** UserService(EmailService aEmailService, DbContext aDbContext)
6. {
7. \_emailService = aEmailService;
8. \_dbContext = aDbContext;
9. }
10. **public** **void** Register(**string** email, **string** password)
11. {
12. **if** (!\_emailService.ValidateEmail(email))
13. **throw** **new** ValidationException("Email is not an email");
14. var user = **new** User(email, password);
15. \_dbContext.Save(user);
16. emailService.SendEmail(**new** MailMessage("myname@mydomain.com", email) {Subject="Hi. How are you!"});
18. }
19. }
20. **public** **class** EmailService
21. {
22. SmtpClient \_smtpClient;
23. **public** EmailService(SmtpClient aSmtpClient)
24. {
25. \_smtpClient = aSmtpClient;
26. }
27. **public** **bool** **virtual** ValidateEmail(**string** email)
28. {
29. **return** email.Contains("@");
30. }
31. **public** **bool** SendEmail(MailMessage message)
32. {
33. \_smtpClient.Send(message);
34. }
35. }

**O: Open/Closed Principle**  
  
The Open/closed Principle says "A software module/class is open for extension and closed for modification".  
  
  
  
Here "Open for extension" means, we need to design our module/class in such a way that the new functionality can be added only when new requirements are generated. "Closed for modification" means we have already developed a class and it has gone through unit testing. We should then not alter it until we find bugs. As it says, a class should be open for extensions, we can use inheritance to do this. Okay, let's dive into an example.  
  
Suppose we have a Rectangle class with the properties Height and Width.

1. **public** **class** Rectangle{
2. **public** **double** Height {**get**;**set**;}
3. **public** **double** Wight {**get**;**set**; }
4. }

Our app needs the ability to calculate the total area of a collection of Rectangles. Since we already learned the Single Responsibility Principle (SRP), we don't need to put the total area calculation code inside the rectangle. So here I created another class for area calculation.

1. **public** **class** AreaCalculator {
2. **public** **double** TotalArea(Rectangle[] arrRectangles)
3. {
4. **double** area;
5. **foreach**(var objRectangle **in** arrRectangles)
6. {
7. area += objRectangle.Height \* objRectangle.Width;
8. }
9. **return** area;
10. }
11. }

Hey, we did it. We made our app without violating SRP. No issues for now. But can we extend our app so that it could calculate the area of not only Rectangles but also the area of Circles as well? Now we have an issue with the area calculation issue, because the way to do circle area calculation is different. Hmm. Not a big deal. We can change the TotalArea method a bit, so that it can accept an array of objects as an argument. We check the object type in the loop and do area calculation based on the object type.

1. **public** **class** Rectangle{
2. **public** **double** Height {**get**;**set**;}
3. **public** **double** Wight {**get**;**set**; }
4. }
5. **public** **class** Circle{
6. **public** **double** Radius {**get**;**set**;}
7. }
8. **public** **class** AreaCalculator
9. {
10. **public** **double** TotalArea(**object**[] arrObjects)
11. {
12. **double** area = 0;
13. Rectangle objRectangle;
14. Circle objCircle;
15. **foreach**(var obj **in** arrObjects)
16. {
17. **if**(obj **is** Rectangle)
18. {
19. objRectangle = (Rectangle)obj;
20. area += obj.Height \* obj.Width;
21. }
22. **else**
23. {
24. objCircle = (Circle)obj;
25. area += objCircle.Radius \* objCircle.Radius \* Math.PI;
26. }
27. }
28. **return** area;
29. }
30. }

Wow. We are done with the change. Here we successfully introduced Circle into our app. We can add a Triangle and calculate it's area by adding one more "if" block in the TotalArea method of AreaCalculator. But every time we introduce a new shape we need to alter the TotalArea method. So the AreaCalculator class is not closed for modification. How can we make our design to avoid this situation? Generally we can do this by referring to abstractions for dependencies, such as interfaces or abstract classes, rather than using concrete classes. Such interfaces can be fixed once developed so the classes that depend upon them can rely upon unchanging abstractions. Functionality can be added by creating new classes that implement the interfaces. So let's refract our code using an interface.

1. **public** **abstract** **class** Shape
2. {
3. **public** **abstract** **double** Area();
4. }

Inheriting from Shape, the Rectangle and Circle classes now look like this:

1. **public** **class** Rectangle: Shape
2. {
3. **public** **double** Height {**get**;**set**;}
4. **public** **double** Width {**get**;**set**;}
5. **public** **override** **double** Area()
6. {
7. **return** Height \* Width;
8. }
9. }
10. **public** **class** Circle: Shape
11. {
12. **public** **double** Radius {**get**;**set**;}
13. **public** **override** **double** Area()
14. {
15. **return** Radius \* Radus \* Math.PI;
16. }
17. }

Every shape contains its area show with it's own way of calculation functionality and our AreaCalculator class will become simpler than before.

1. **public** **class** AreaCalculator
2. {
3. **public** **double** TotalArea(Shape[] arrShapes)
4. {
5. **double** area=0;
6. **foreach**(var objShape **in** arrShapes)
7. {
8. area += objShape.Area();
9. }
10. **return** area;
11. }
12. }

Now our code is following SRP and OCP both. Whenever you introduce a new shape by deriving from the "Shape" abstract class, you need not change the "AreaCalculator" class. Awesome. Isn't it?  
  
**L: Liskov substitution Principle**  
  
  
  
The Liskov Substitution Principle (LSP) states that "**you should be able to use any derived class instead of a parent class and have it behave in the same manner without modification**". It ensures that a derived class does not affect the behavior of the parent class, in other words that a derived class must be substitutable for its base class.  
  
This principle is just an extension of the Open Close Principle and it means that we must ensure that new derived classes extend the base classes without changing their behavior. I will explain this with a real world example that violates LSP.  
  
A father is a doctor whereas his son wants to become a cricketer. So here the son can't replace his father even though they both belong to the same family hierarchy.  
  
Now jump into an example to learn how a design can violate LSP. Suppose we need to build an app to manage data using a group of SQL files text. Here we need to write functionality to load and save the text of a group of SQL files in the application directory. So we need a class that manages the load and save of the text of group of SQL files along with the SqlFile Class.

1. **public** **class** SqlFile
2. {
3. **public** **string** FilePath {**get**;**set**;}
4. **public** **string** FileText {**get**;**set**;}
5. **public** **string** LoadText()
6. {
7. /\* Code to read text from sql file \*/
8. }
9. **public** **string** SaveText()
10. {
11. /\* Code to save text into sql file \*/
12. }
13. }
14. **public** **class** SqlFileManager
15. {
16. **public** List<SqlFile> lstSqlFiles {**get**;**set**}
18. **public** **string** GetTextFromFiles()
19. {
20. StringBuilder objStrBuilder = **new** StringBuilder();
21. **foreach**(var objFile **in** lstSqlFiles)
22. {
23. objStrBuilder.Append(objFile.LoadText());
24. }
25. **return** objStrBuilder.ToString();
26. }
27. **public** **void** SaveTextIntoFiles()
28. {
29. **foreach**(var objFile **in** lstSqlFiles)
30. {
31. objFile.SaveText();
32. }
33. }
34. }

OK. We are done with our part. the functionality looks good for now. After some time our lead might tell us that we may have a few read-only files in the application folder, so we need to restrict the flow whenever it tries to do a save on them.  
  
OK. We can do that by creating a "ReadOnlySqlFile" class that inherits the "SqlFile" class and we need to alter the SaveTextIntoFiles() method by introducing a condition to prevent calling the SaveText() method on ReadOnlySqlFile instances.

1. **public** **class** SqlFile
2. {
3. **public** **string** LoadText()
4. {
5. /\* Code to read text from sql file \*/
6. }
7. **public** **void** SaveText()
8. {
9. /\* Code to save text into sql file \*/
10. }
11. }
12. **public** **class** ReadOnlySqlFile: SqlFile
13. {
14. **public** **string** FilePath {**get**;**set**;}
15. **public** **string** FileText {**get**;**set**;}
16. **public** **string** LoadText()
17. {
18. /\* Code to read text from sql file \*/
19. }
20. **public** **void** SaveText()
21. {
22. /\* Throw an exception when app flow tries to do save. \*/
23. **throw** **new** IOException("Can't Save");
24. }
25. }

To avoid an exception we need to modify "SqlFileManager" by adding one condition to the loop.

1. **public** **class** SqlFileManager
2. {
3. **public** List<SqlFile? lstSqlFiles {**get**;**set**}
4. **public** **string** GetTextFromFiles()
5. {
6. StringBuilder objStrBuilder = **new** StringBuilder();
7. **foreach**(var objFile **in** lstSqlFiles)
8. {
9. objStrBuilder.Append(objFile.LoadText());
10. }
11. **return** objStrBuilder.ToString();
12. }
13. **public** **void** SaveTextIntoFiles()
14. {
15. **foreach**(var objFile **in** lstSqlFiles)
16. {
17. //Check whether the current file object is read only or not.If yes, skip calling it's
18. // SaveText() method to skip the exception.
20. **if**(! objFile **is** ReadOnlySqlFile)
21. objFile.SaveText();
22. }
23. }
24. }

Here we altered the SaveTextIntoFiles() method in the SqlFileManager class to determine whether or not the instance is of ReadOnlySqlFile to avoid the exception. We can't use this ReadOnlySqlFile class as a substitute of it's parent without altering SqlFileManager code. So we can say that this design is not following LSP. Let's make this design follow the LSP. Here we will introduce interfaces to make the SqlFileManager class independent from the rest of the blocks.

1. **public** **interface** IReadableSqlFile
2. {
3. **string** LoadText();
4. }
5. **public** **interface** IWritableSqlFile
6. {
7. **void** SaveText();
8. }

Now we implement IReadableSqlFile through the ReadOnlySqlFile class that reads only the text from read-only files.

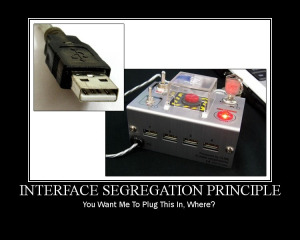
1. **public** **class** ReadOnlySqlFile: IReadableSqlFile
2. {
3. **public** **string** FilePath {**get**;**set**;}
4. **public** **string** FileText {**get**;**set**;}
5. **public** **string** LoadText()
6. {
7. /\* Code to read text from sql file \*/
8. }
9. }

Here we implement both IWritableSqlFile and IReadableSqlFile in a SqlFile class by which we can read and write files.

1. **public** **class** SqlFile: IWritableSqlFile,IReadableSqlFile
2. {
3. **public** **string** FilePath {**get**;**set**;}
4. **public** **string** FileText {**get**;**set**;}
5. **public** **string** LoadText()
6. {
7. /\* Code to read text from sql file \*/
8. }
9. **public** **void** SaveText()
10. {
11. /\* Code to save text into sql file \*/
12. }
13. }

Now the design of the SqlFileManager class becomes like this:

1. **public** **class** SqlFileManager
2. {
3. **public** **string** GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)
4. {
5. StringBuilder objStrBuilder = **new** StringBuilder();
6. **foreach**(var objFile **in** aLstReadableFiles)
7. {
8. objStrBuilder.Append(objFile.LoadText());
9. }
10. **return** objStrBuilder.ToString();
11. }
12. **public** **void** SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles)
13. {
14. **foreach**(var objFile **in** aLstWritableFiles)
15. {
16. objFile.SaveText();
17. }
18. }
19. }

Here the GetTextFromFiles() method gets only the list of instances of classes that implement the IReadOnlySqlFile interface. That means the SqlFile and ReadOnlySqlFile class instances. And the SaveTextIntoFiles() method gets only the list instances of the class that implements the IWritableSqlFiles interface, in other words SqlFile instances in this case. Now we can say our design is following the LSP. And we fixed the problem using the Interface segregation principle by (ISP) identifying the abstraction and the responsibility separation method.  
  
**I: Interface Segregation Principle (ISP)**  
  
The Interface Segregation Principle states "that clients should not be forced to implement interfaces they don't use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one sub module.".  
  
  
  
We can define it in another way. An interface should be more closly related to the code that uses it than code that implements it. So the methods on the interface are defined by which methods the client code needs than which methods the class implements. So clients should not be forced to depend upon interfaces that they don't use.  
  
Like classes, each interface should have a specific purpose/responsibility (refer to SRP). You shouldn't be forced to implement an interface when your object doesn't share that purpose. The larger the interface, the more likely it includes methods that not all implementers can do. That's the essence of the Interface Segregation Principle. Let's start with an example that breaks ISP. Suppose we need to build a system for an IT firm that contains roles like TeamLead and Programmer where TeamLead divides a huge task into smaller tasks and assigns them to his/her programmers or can directly work on them.  
  
Based on specifications, we need to create an interface and a TeamLead class to implement it.

1. **public** Interface ILead
2. {
3. **void** CreateSubTask();
4. **void** AssginTask();
5. **void** WorkOnTask();
6. }
7. **public** **class** TeamLead : ILead
8. {
9. **public** **void** AssignTask()
10. {
11. //Code to assign a task.
12. }
13. **public** **void** CreateSubTask()
14. {
15. //Code to create a sub task
16. }
17. **public** **void** WorkOnTask()
18. {
19. //Code to implement perform assigned task.
20. }
21. }

OK. The design looks fine for now. Later another role like Manager, who assigns tasks to TeamLead and will not work on the tasks, is introduced into the system. Can we directly implement an ILead interface in the Manager class, like the following?

1. **public** **class** Manager: ILead
2. {
3. **public** **void** AssignTask()
4. {
5. //Code to assign a task.
6. }
7. **public** **void** CreateSubTask()
8. {
9. //Code to create a sub task.
10. }
11. **public** **void** WorkOnTask()
12. {
13. **throw** **new** Exception("Manager can't work on Task");
14. }
15. }

Since the Manager can't work on a task and at the same time no one can assign tasks to the Manager, this WorkOnTask() should not be in the Manager class. But we are implementing this class from the ILead interface, we need to provide a concrete Method. Here we are forcing the Manager class to implement a WorkOnTask() method without a purpose. This is wrong. The design violates ISP. Let's correct the design.  
  
Since we have three roles, 1. Manager, that can only divide and assign the tasks , 2. TeamLead that can divide and assign the tasks and can work on them as well, 3. Programmer that can only work on tasks, we need to divide the responsibilities by segregating the ILead interface. An interface that provides a contract for WorkOnTask().

1. **public** **interface** IProgrammer
2. {
3. **void** WorkOnTask();
4. }

An interface that provide contracts to manage the tasks:

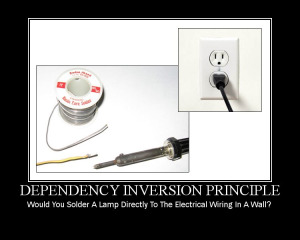
1. **public** **interface** ILead
2. {
3. **void** AssignTask();
4. **void** CreateSubTask();
5. }

Then the implementation becomes:

1. **public** **class** Programmer: IProgrammer
2. {
3. **public** **void** WorkOnTask()
4. {
5. //code to implement to work on the Task.
6. }
7. }
8. **public** **class** Manager: ILead
9. {
10. **public** **void** AssignTask()
11. {
12. //Code to assign a Task
13. }
14. **public** **void** CreateSubTask()
15. {
16. //Code to create a sub taks from a task.
17. }
18. }

TeamLead can manage tasks and can work on them if needed. Then the TeamLead class should implement both of the IProgrammer and ILead interfaces.

1. **public** **class** TeamLead: IProgrammer, ILead
2. {
3. **public** **void** AssignTask()
4. {
5. //Code to assign a Task
6. }
7. **public** **void** CreateSubTask()
8. {
9. //Code to create a sub task from a task.
10. }
11. **public** **void** WorkOnTask()
12. {
13. //code to implement to work on the Task.
14. }
15. }

Wow. Here we separated responsibilities/purposes and distributed them on multiple interfaces and provided a good level of abstraction too.  
  
**D: Dependency Inversion Principle**  
The Dependency Inversion Principle (DIP) states that high-level modules/classes should not depend upon low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.  
  
  
  
High-level modules/classes implement business rules or logic in a system (application). Low-level modules/classes deal with more detailed operations, in other words they may deal with writing information to databases or passing messages to the operating system or services.  
  
A high-level module/class that has dependency on low-level modules/classes or some other class and knows a lot about the other classes it interacts with is said to be tightly coupled. When a class knows explicitly about the design and implementation of another class, it raises the risk that changes to one class will break the other class. So we must keep these high-level and low-level modules/class loosely coupled as much as we can. To do that, we need to make both of them dependent on abstractions instead of knowing each other. Let's start an with an example.  
  
Suppose we need to work on an error logging module that logs exception stack traces into a file. Simple, isn't it? The following are the classes that provide functionality to log a stack trace into a file.

1. **public** **class** FileLogger
2. {
3. **public** **void** LogMessage(**string** aStackTrace)
4. {
5. //code to log stack trace into a file.
6. }
7. }
8. **public** **static** **class** ExceptionLogger
9. {
10. **public** **static** **void** LogIntoFile(Exception aException)
11. {
12. FileLogger objFileLogger = **new** FileLogger();
13. objFileLogger.LogMessage(GetUserReadableMessage(aException));
14. }
15. **private** **string** GetUserReadableMessage(Exception ex)
16. {
17. **string** strMessage = **string**. Empty;
18. //code to convert Exception's stack trace and message to user readable format.
19. ....
20. ....
21. **return** strMessage;
22. }
23. }

A client class exports data from many files to a database.

1. **public** **class** DataExporter
2. {
3. **public** **void** ExportDataFromFile()
4. {
5. **try** {
6. //code to export data from files to database.
7. }
8. **catch**(Exception ex)
9. {
10. **new** ExceptionLogger().LogIntoFile(ex);
11. }
12. }
13. }

Looks good. We sent our application to the client. But our client wants to store this stack trace in a database if an IO exception occurs. Hmm... okay, no problem. We can implement that too. Here we need to add one more class that provides the functionality to log the stack trace into the database and an extra method in ExceptionLogger to interact with our new class to log the stack trace.

1. **public** **class** DbLogger
2. {
3. **public** **void** LogMessage(**string** aMessage)
4. {
5. //Code to write message in database.
6. }
7. }
8. **public** **class** FileLogger
9. {
10. **public** **void** LogMessage(**string** aStackTrace)
11. {
12. //code to log stack trace into a file.
13. }
14. }
15. **public** **class** ExceptionLogger
16. {
17. **public** **void** LogIntoFile(Exception aException)
18. {
19. FileLogger objFileLogger = **new** FileLogger();
20. objFileLogger.LogMessage(GetUserReadableMessage(aException));
21. }
22. **public** **void** LogIntoDataBase(Exception aException)
23. {
24. DbLogger objDbLogger = **new** DbLogger();
25. objDbLogger.LogMessage(GetUserReadableMessage(aException));
26. }
27. **private** **string** GetUserReadableMessage(Exception ex)
28. {
29. **string** strMessage = **string**.Empty;
30. //code to convert Exception's stack trace and message to user readable format.
31. ....
32. ....
33. **return** strMessage;
34. }
35. }
36. **public** **class** DataExporter
37. {
38. **public** **void** ExportDataFromFile()
39. {
40. **try** {
41. //code to export data from files to database.
42. }
43. **catch**(IOException ex)
44. {
45. **new** ExceptionLogger().LogIntoDataBase(ex);
46. }
47. **catch**(Exception ex)
48. {
49. **new** ExceptionLogger().LogIntoFile(ex);
50. }
51. }
52. }

Looks fine for now. But whenever the client wants to introduce a new logger, we need to alter ExceptionLogger by adding a new method. If we continue doing this after some time then we will see a fat ExceptionLogger class with a large set of methods that provide the functionality to log a message into various targets. Why does this issue occur? Because ExceptionLogger directly contacts the low-level classes FileLogger and and DbLogger to log the exception. We need to alter the design so that this ExceptionLogger class can be loosely coupled with those class. To do that we need to introduce an abstraction between them, so that ExcetpionLogger can contact the abstraction to log the exception instead of depending on the low-level classes directly.

1. **public** **interface** ILogger
2. {
3. **public** **void** LogMessage(**string** aString);
4. }

Now our low-level classes need to implement this interface.

1. **public** **class** DbLogger: ILogger
2. {
3. **public** **void** LogMessage(**string** aMessage)
4. {
5. //Code to write message in database.
6. }
7. }
8. **public** **class** FileLogger: ILogger
9. {
10. **public** **void** LogMessage(**string** aStackTrace)
11. {
12. //code to log stack trace into a file.
13. }
14. }

Now, we move to the low-level class's intitiation from the ExcetpionLogger class to the DataExporter class to make ExceptionLogger loosely coupled with the low-level classes FileLogger and EventLogger. And by doing that we are giving provision to DataExporter class to decide what kind of Logger should be called based on the exception that occurs.

1. **public** **class** ExceptionLogger
2. {
3. **private** ILogger \_logger;
4. **public** ExceptionLogger(ILogger aLogger)
5. {
6. **this**.\_logger = aLogger;
7. }
8. **public** **void** LogException(Exception aException)
9. {
10. **string** strMessage = GetUserReadableMessage(aException);
11. **this**.\_logger.LogMessage(strMessage);
12. }
13. **private** **string** GetUserReadableMessage(Exception aException)
14. {
15. **string** strMessage = **string**.Empty;
16. //code to convert Exception's stack trace and message to user readable format.
17. ....
18. ....
19. **return** strMessage;
20. }
21. }
22. **public** **class** DataExporter
23. {
24. **public** **void** ExportDataFromFile()
25. {
26. ExceptionLogger \_exceptionLogger;
27. **try** {
28. //code to export data from files to database.
29. }
30. **catch**(IOException ex)
31. {
32. \_exceptionLogger = **new** ExceptionLogger(**new** DbLogger());
33. \_exceptionLogger.LogException(ex);
34. }
35. **catch**(Exception ex)
36. {
37. \_exceptionLogger = **new** ExceptionLogger(**new** FileLogger());
38. \_exceptionLogger.LogException(ex);
39. }
40. }
41. }

We successfully removed the dependency on low-level classes. This ExceptionLogger doesn't depend on the FileLogger and EventLogger classes to log the stack trace. We don't need to change the ExceptionLogger's code any more for any new logging functionality. We need to create a new logging class that implements the ILogger interface and must add another catch block to the DataExporter class's ExportDataFromFile method.

1. **public** **class** EventLogger: ILogger
2. {
3. **public** **void** LogMessage(**string** aMessage)
4. {
5. //Code to write message in system's event viewer.
6. }
7. }

And we need to add a condition in the DataExporter class as in the following:

1. **public** **class** DataExporter
2. {
3. **public** **void** ExportDataFromFile()
4. {
5. ExceptionLogger \_exceptionLogger;
6. **try** {
7. //code to export data from files to database.
8. }
9. **catch**(IOException ex)
10. {
11. \_exceptionLogger = **new** ExceptionLogger(**new** DbLogger());
12. \_exceptionLogger.LogException(ex);
13. }
14. **catch**(SqlException ex)
15. {
16. \_exceptionLogger = **new** ExceptionLogger(**new** EventLogger());
17. \_exceptionLogger.LogException(ex);
18. }
19. **catch**(Exception ex)
20. {
21. \_exceptionLogger = **new** ExceptionLogger(**new** FileLogger());
22. \_exceptionLogger.LogException(ex);
23. }
24. }
25. }

Looks good. But we introduced the dependency here in the DataExporter class's catch blocks. Yeah, someone must take the responsibility to provide the necessary objects to the ExceptionLogger to make the work done.  
  
Let me explain it with a real world example. Suppose we want to have a wooden chair with specific measurements and the kind of wood to be used to make that chair from. Then we can't leave the decision making on measurements and the wood to the carpenter. Here his job is to make a chair based on our requirements with his tools and we provide the specifications to him to make a good chair.  
  
So what is the benefit we get by the design? Yes, we definitely have a benefit with it. We need to modify both the DataExporter class and ExceptionLogger class whenever we need to introduce a new logging functionality. But in the updated design we need to add only another catch block for the new exception logging feature. Coupling is not inherently evil. If you don't have some amount of coupling, your software will not do anything for you. The only thing we need to do is understand the system, requirements and environment properly and find areas where DIP should be followed.  
  
Great, we have gone through the all five SOLID principles successfully. And we can conclude that using these principles we can build an application with tidy, readable and easily maintainable code.

Here you may have some doubt. Yes, about the quantity of code. Because of these principles, the code might become larger in our applications. But my dear friends, you need to compare it with the quality that we get by following these principles. Hmm, but anyway 27 lines are much less than 200 lines. I am not saying that these principles should be followed 100%, you need to draw a Yellow so that you can hold the control over the things like quality and delivery to maintain their balance.  
  
This is my little effort to share the uses of SOLID principles. I hope you enjoyed this article.

# 21 Explain Attribute Routing.

#### Why Attribute Routing?

For example, a socially enhanced e-commerce website could have the following routes:

* {productId:int}/{productTitle}   
  Mapped to ProductsController.Show(int id)
* {username}   
  Mapped to ProfilesController.Show(string username)
* {username}/catalogs/{catalogId:int}/{catalogTitle}   
  Mapped to CatalogsController.Show(string username, int catalogId)

(Don’t mind the specific syntax right now, we will touch on this later.)

In previous version of ASP.NET MVC, the rules would be set in the RouteConfig.cs file, and point to the actual controller actions, as such:

* routes.MapRoute(
* name: “ProductPage”,
* url: “{productId}/{productTitle}”,
* defaults: new { controller = “Products”, action = “Show” },
* constraints: new { productId = “\\d+” }
* );

When the route definitions are co-located with the actions, within the same source file rather than being declared on an external configuration class, it can make it easier to reason about the mapping between URIs and actions. The previous route definition would be set using the following, simple attribute:

* [Route(“{productId:int}/{productTitle}”)]
* public ActionResult Show(int productId) { … }

#### Enabling Attribute Routing

To enable attribute routing, call MapMvcAttributeRoutes during configuration.

* public class RouteConfig
* {
* public static void RegisterRoutes(RouteCollection routes)
* {
* routes.IgnoreRoute(“{resource}.axd/{\*pathInfo}”);
* routes.MapMvcAttributeRoutes();
* }
* }

You can also combine attribute routing with convention-based routing.

* public static void RegisterRoutes(RouteCollection routes)
* {
* routes.IgnoreRoute(“{resource}.axd/{\*pathInfo}”);
* routes.MapMvcAttributeRoutes();
* routes.MapRoute(
* name: “Default”,
* url: “{controller}/{action}/{id}”,
* defaults: new { controller = “Home”, action = “Index”, id = UrlParameter.Optional }
* );
* }

#### Optional URI Parameters and Default Values

You can make a URI parameter optional by adding a question mark to the route parameter. You can also specify a default value by using the form parameter=value.

* public class BooksController : Controller
* {
* // eg: /books
* // eg: /books/1430210079
* [Route(“books/{isbn?}”)]
* public ActionResult View(string isbn)
* {
* if (!String.IsNullOrEmpty(isbn))
* {
* return View(“OneBook”, GetBook(isbn));
* }
* return View(“AllBooks”, GetBooks());
* }
* // eg: /books/lang
* // eg: /books/lang/en
* // eg: /books/lang/he
* [Route(“books/lang/{lang=en}”)]
* public ActionResult ViewByLanguage(string lang)
* {
* return View(“OneBook”, GetBooksByLanguage(lang));
* }
* }

In this example, both /books and /books/1430210079 will route to the “View” action, the former will result with listing all books, and the latter will list the specific book. Both /books/lang and /books/lang/en will be treated the same.

#### Route Prefixes

Often, the routes in a controller all start with the same prefix. For example:

* public class ReviewsController : Controller
* {
* // eg: /reviews
* [Route(“reviews”)]
* public ActionResult Index() { … }
* // eg: /reviews/5
* [Route(“reviews/{reviewId}”)]
* public ActionResult Show(int reviewId) { … }
* // eg: /reviews/5/edit
* [Route(“reviews/{reviewId}/edit”)]
* public ActionResult Edit(int reviewId) { … }
* }

You can set a common prefix for an entire controller by using the [RoutePrefix] attribute:

* [RoutePrefix(“reviews”)]
* public class ReviewsController : Controller
* {
* // eg.: /reviews
* [Route]
* public ActionResult Index() { … }
* // eg.: /reviews/5
* [Route(“{reviewId}”)]
* public ActionResult Show(int reviewId) { … }
* // eg.: /reviews/5/edit
* [Route(“{reviewId}/edit”)]
* public ActionResult Edit(int reviewId) { … }
* }

Use a tilde (~) on the method attribute to override the route prefix if needed:

* [RoutePrefix(“reviews”)]
* public class ReviewsController : Controller
* {
* // eg.: /spotlight-review
* [Route(“~/spotlight-review”)]
* public ActionResult ShowSpotlight() { … }
* …
* }

#### Default Route

You can also apply the [Route] attribute on the controller level, capturing the action as a parameter. That route would then be applied on all actions in the controller, unless a specific [Route] has been defined on a specific action, overriding the default set on the controller.

* [RoutePrefix(“promotions”)]
* [Route(“{action=index}”)]
* public class ReviewsController : Controller
* {
* // eg.: /promotions
* public ActionResult Index() { … }
* // eg.: /promotions/archive
* public ActionResult Archive() { … }
* // eg.: /promotions/new
* public ActionResult New() { … }
* // eg.: /promotions/edit/5
* [Route(“edit/{promoId:int}”)]
* public ActionResult Edit(int promoId) { … }
* }

#### Route Constraints

Route constraints let you restrict how the parameters in the route template are matched. The general syntax is {parameter:constraint}. For example:

* // eg: /users/5
* [Route(“users/{id:int}”]
* public ActionResult GetUserById(int id) { … }
* // eg: users/ken
* [Route(“users/{name}”]
* public ActionResult GetUserByName(string name) { … }

Here, the first route will only be selected if the “id” segment of the URI is an integer. Otherwise, the second route will be chosen.

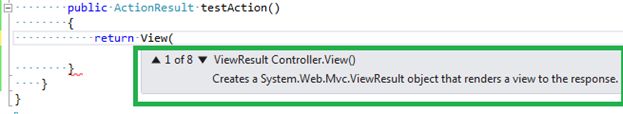
# 22 Explain result types or various result type of MVC?

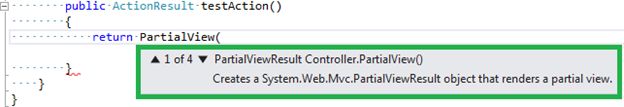
Developers with limited hands-on experience with MVC should be able to provide an answer to the question, because the scenario is very common and every now and then it needs to return something from the controller to the presentation environment.  
  
We are very familiar with the “ActionResult” class that is the base class of many classes and we can return an object of those classes. The class hierarchy is as in the following:

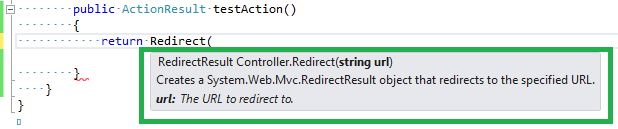
System.Object   
System.Web.Mvc.ActionResult

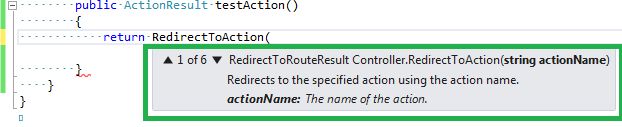
* System.Web.Mvc.ContentResult
* System.Web.Mvc.EmptyResult
* System.Web.Mvc.FileResult
* System.Web.Mvc.HttpStatusCodeResult
* System.Web.Mvc.JavaScriptResult
* System.Web.Mvc.JsonResult
* System.Web.Mvc.RedirectResult
* System.Web.Mvc.RedirectToRouteResult
* System.Web.Mvc.ViewResultBase

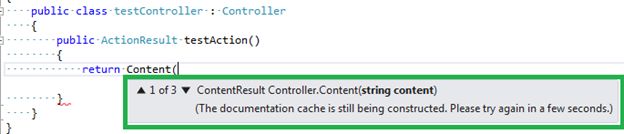
In this example we will see all of the derived classes that is inherited from the “ActionResult” base class. So, let's start one by one.  
  
**Return View**  
This is a most common and very frequently used type. We see that we can pass eight parameters when we return the view. We can specify the view name explicitly or may not.

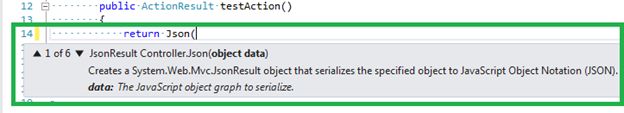
  
  
**Return partial View**  
The concept of a partial view is very similar to the master page concept in Web Form applications. The partial view is nothing but  pagelet, that we can return from the controller and that merges with the main view and generates one concrete HTML page.

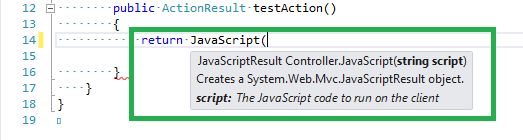
  
  
It may take 4 parameters to render in the partial view.  
  
**Redirect**  
This is equivalent to Response.redirect() or Server.Transfer() functions. It takes the URL path to be redirect , though we can use Response.Redirect() or Server.Transfer() in MVC too.

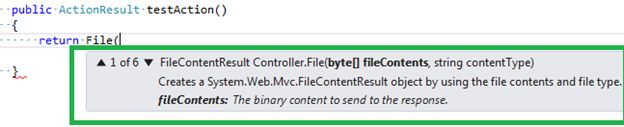
  
  
**Redirect To Action**  
Sometimes it is necessary to call another action after completion of one action, this is very similar to a function call in traditional function oriented programming or Object Oriented Programming. It may take 6 parameters. The first parameter is very simple, only action name.

  
  
**Return content**  
This is useful when we want to return a small amount of strings from a controller/action. It takes three parameters. The first one is a simple string and the remaining two are strings with little information.

  
  
**Return JSON**  
This is very useful when we don't want an entire HTML page but only want a value. Generally in AJAX-based single-page applications we do not load an entire page again and again but load fresh data from the DB using AJAX. In this scenario we can return only a JSON object and in the success function of jQuery ajax (let's assume we are using the jQuery library to implement AJAX) we can just manipulate data.

  
  
**Return JavaScript**  
When we wanted to return a JavaScript string , we may use this function. It takes only one parameter, the string only.

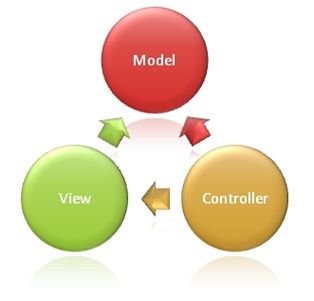
  
  
**Return File**  
We are allowed to return a binary file if needed from a controller. It takes 6 parameters maximum.



# Top 50 MVC questions

**Question 1: What is MVC (Model view controller)?**  
**Answer:**   
  
Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representation of information from the way that information is presented to or accepted from the user.  
  
MVC is a framework for building web applications using a MVC (Model View Controller) design:

* The Model represents the application core (for instance a list of database records).
* The View displays the data (the database records).
* The Controller handles the input (to the database records).

The MVC model also provides full control over HTML, CSS, and JavaScript.  
  
  
  
**The MVC model defines web applications with 3 logic layers:**

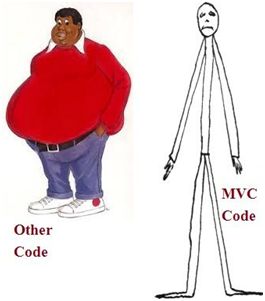
* The business layer (Model logic)
* The display layer (View logic)
* The input control (Controller logic)

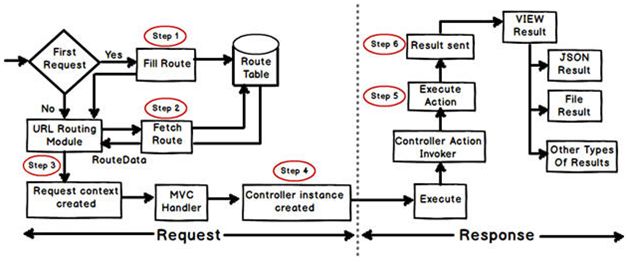
**The Model** is the part of the application that handles the logic for the application data.  
  
Often model objects retrieve data (and store data) from a database.  
  
**The View** is the part of the application that handles the display of the data.  
  
Most often the views are created from the model data.  
  
**The Controller** is the part of the application that handles user interaction.  
  
Typically controllers read data from a view, control user input, and send input data to the model.  
  
The MVC separation helps you manage complex applications, because you can focus on one aspect a time. For example, you can focus on the view without depending on the business logic. It also makes it easier to test an application.  
  
The MVC separation also simplifies group development. Different developers can work on the view, the controller logic, and the business logic in parallel.

* [Overview Of ASP.NET MVC](http://www.c-sharpcorner.com/UploadFile/8a67c0/an-overview-on-Asp-Net-mvc/)

**Question 2: What are the advantages of MVC?**  
**Answer: Benefits of MVC:**

* **Multiple view support**Due to the separation of the model from the view, the user interface can display multiple views of the same data at the same time.
* **Change Accommodation**User interfaces tend to change more frequently than business rules (different colors, fonts, screen layouts, and levels of support for new devices such as cell phones or PDAs) because the model does not depend on the views, adding new types of views to the system generally does not affect the model. As a result, the scope of change is confined to the view.

**SoC – Separation of Concerns**Separation of Concerns is one of the core advantages of ASP.NET MVC . The MVC framework provides a clean separation of the UI, Business Logic, Model or Data.  
  
**More Control**The ASP.NET MVC framework provides more control over HTML, JavaScript and CSS than the traditional Web Forms.  
  
**Testability**  
ASP.NET MVC framework provides better testability of the Web Application and good support for the test driven development too.  
  
**Lightweight**  
ASP.NET MVC framework doesn’t use View State and thus reduces the bandwidth of the requests to an extent.  
  
**Full features of ASP.NET**One of the key advantages of using ASP.NET MVC is that it is built on top of ASP.NET framework and hence most of the features of the ASP.NET like membership providers, roles, etc can still be used.  
  


**Question 3: Explain MVC application life cycle?  
  
Answer:**Any web application has two main execution steps, first understanding the request and depending on the type of the request sending out appropriate response. MVC application life cycle is not different it has two main phases, first creating the request object and second sending our response to the browser.  
  
**Creating the request object:**The request object creation has four major steps. The following is the detailed explanation of the same.  
  
**Step 1: Fill route**MVC requests are mapped to route tables which in turn specify which controller and action to be invoked. So if the request is the first request the first thing is to fill the route table with routes collection. This filling of route table happens in the global.asax file.  
  
**Step 2: Fetch route**Depending on the URL sent “UrlRoutingModule” searches the route table to create “RouteData” object which has the details of which controller and action to invoke.  
  
**Step 3: Request context created**The “RouteData” object is used to create the “RequestContext” object.  
  
**Step 4: Controller instance created**This request object is sent to “MvcHandler” instance to create the controller class instance. Once the controller class object is created it calls the “Execute” method of the controller class.  
  
**Creating Response object**This phase has two steps executing the action and finally sending the response as a result to the view.  
  


* [ASP.Net MVC Life Cycle](http://www.c-sharpcorner.com/UploadFile/00a8b7/Asp-Net-mvc-life-cycle/)

**Question 4: List out different return types of a controller action method?**  
**Answer:**There are total nine return types we can use to return results from controller to view.   
  
The base type of all these result types is ActionResult.

1. **ViewResult (View)**: This return type is used to return a webpage from an action method.
2. **PartialviewResult (Partialview)**: This return type is used to send a part of a view which will be rendered in another view.
3. **RedirectResult (Redirect)**: This return type is used to redirect to any other controller and action method depending on the URL.
4. **RedirectToRouteResult (RedirectToAction, RedirectToRoute):** This return type is used when we want to redirect to any other action method.
5. **ContentResult (Content)**: This return type is used to return HTTP content type like text/plain as the result of the action.
6. **jsonResult (json):** This return type is used when we want to return a JSON message.
7. **javascriptResult (javascript):** This return type is used to return JavaScript code that will run in browser.
8. **FileResult (File):** This return type is used to send binary output in response.
9. **EmptyResult:** This return type is used to return nothing (void) in the result.

* [Various Return Types From MVC Controller](http://www.c-sharpcorner.com/UploadFile/dacca2/various-return-types-from-mvc-controller/)

**Question 5: What are Filters in MVC?**  
**Answer:**In MVC, controllers define action methods and these action methods generally have a one-to-one relationship with UI controls such as clicking a button or a link, etc. For example, in one of our previous examples, the UserController class contained methods UserAdd, UserDelete, etc.  
  
But many times we would like to perform some action before or after a particular operation. For achieving this functionality, ASP.NET MVC provides feature to add pre and post action behaviors on controller's action methods.  
  
**Types of Filters:**  
ASP.NET MVC framework supports the following action filters:

* **Action Filters:** Action filters are used to implement logic that gets executed before and after a controller action executes. We will look at Action Filters in detail in this chapter.
* **Authorization Filters:**Authorization filters are used to implement authentication and authorization for controller actions.
* **Result Filters:** Result filters contain logic that is executed before and after a view result is executed. For example, you might want to modify a view result right before the view is rendered to the browser.
* **Exception Filters:**Exception filters are the last type of filter to run. You can use an exception filter to handle errors raised by either your controller actions or controller action results. You can also use exception filters to log errors.

Action filters are one of most commonly used filters to perform additional data processing, or manipulating the return values or cancelling the execution of action or modifying the view structure at run time.

* [Understanding Filters in MVC](http://www.c-sharpcorner.com/UploadFile/0ef46a/filters-in-mvc/)

**Question 6: What are Action Filters in MVC?**  
**Answer: Action Filters:**Action Filters are additional attributes that can be applied to either a controller section or the entire controller to modify the way in which action is executed. These attributes are special .NET classes derived from System.Attribute which can be attached to classes, methods, properties and fields.  
 **ASP.NET MVC provides the following action filters:**

* **Output Cache:**This action filter caches the output of a controller action for a specified amount of time.
* **Handle Error:** This action filter handles errors raised when a controller action executes.
* **Authorize:**This action filter enables you to restrict access to a particular user or role.

Now we will see the code example to apply these filters on an example controller ActionFilterDemoController. (ActionFilterDemoController is just used as an example. You can use these filters on any of your controllers.)  
  
**Output Cache  
  
E.g.:** Specifies the return value to be cached for 10 seconds.

1. publicclassActionFilterDemoController: Controller
2. {
3. [HttpGet]
4. OutputCache(Duration = 10)]
5. publicstringIndex()
6. {
7. returnDateTime.Now.ToString("T");
9. }
10. }

* [ASP.NET MVC with Action Filters](http://www.c-sharpcorner.com/UploadFile/15812c/Asp-Net-mvc-application-with-action-filters/)

**Question 7: Explain what is routing in MVC? What are the three segments for routing important?  
  
Answer:**Routing is a mechanism to process the incoming url that is more descriptive and give desired response. In this case, URL is not mapped to specific files or folder as was the case of earlier days web sites.  
  
There are two types of routing (after the introduction of ASP.NET MVC 5).

1. **Convention based routing:** to define this type of routing, we call MapRoute method and set its unique name, url pattern and specify some default values.
2. **Attribute based routing:** to define this type of routing, we specify the Route attribute in the action method of the controller.

Routing is the URL pattern that is mappped together to a handler,rounting is responsible for incoming browser request for particular MVC controller. In other ways let us say routing help you to define a URL structure and map the URL with controller. There are three segments for routing that are important,

1. ControllerName
2. ActionMethodName
3. Parammeter

**i.e:** ControllerName/ActionMethodName/{ParamerName} and also route map coding written in a Global.asax file.

* [Routing in MVC](http://www.c-sharpcorner.com/UploadFile/0ef46a/routing-in-mvc/)

**Question 8: What is Route in MVC? What is Default Route in MVC?**  
**Answer**: A route is a URL pattern that is mapped to a handler. The handler can be a physical file, such as a .aspx file in a Web Forms application. A handler can also be a class that processes the request, such as a controller in an MVC application. To define a route, you create an instance of the [Route](https://msdn.microsoft.com/en-us/library/system.web.routing.route.aspx) class by specifying the URL pattern, the handler, and optionally a name for the route.  
  
You add the route to the application by adding the Route object to the static Routes property of the RouteTable class. The Routesproperty is a RouteCollection object that stores all the routes for the application.  
  
You typically do not have to write code to add routes in an MVC application. Visual Studio project templates for MVC include preconfigured URL routes. These are defined in the Mvc Application class, which is defined in the Global.asax file.

|  |  |
| --- | --- |
| **Route definition** | **Example of matching URL** |
| {controller}/{action}/{id} | /Products/show/beverages |
| {table}/Details.aspx | /Products/Details.aspx |
| blog/{action}/{entry} | /blog/show/123 |
| {reporttype}/{year}/{month}/{day} | /sales/2008/1/5 |
| {locale}/{action} | /US/show |
| {language}-{country}/{action} | /en-US/show |

**Default Route**  
  
The default ASP.NET MVC project templates add a generic route that uses the following URL convention to break the URL for a given request into three named segments.  
  
**URL:**"{controller}/{action}/{id}"  
  
This route pattern is registered via call to the **MapRoute()** extension method of **RouteCollection**.  
  


* [Routing in MVC](http://www.c-sharpcorner.com/UploadFile/3d39b4/routing-in-mvc/)

**Question 9: Mention what is the difference between Temp data, View, and View Bag?  
  
Answer:**In ASP.NET MVC there are three ways to pass/store data between the controllers and views.  
  
**ViewData**

1. ViewData is used to pass data from controller to view.
2. It is derived from ViewDataDictionary class.
3. It is available for the current request only.
4. Requires typecasting for complex data type and checks for null values to avoid error.
5. If redirection occurs, then its value becomes null.

**ViewBag**

1. ViewBag is also used to pass data from the controller to the respective view.
2. ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0
3. It is also available for the current request only.
4. If redirection occurs, then its value becomes null.
5. Doesn’t require typecasting for complex data type.

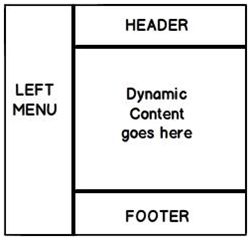
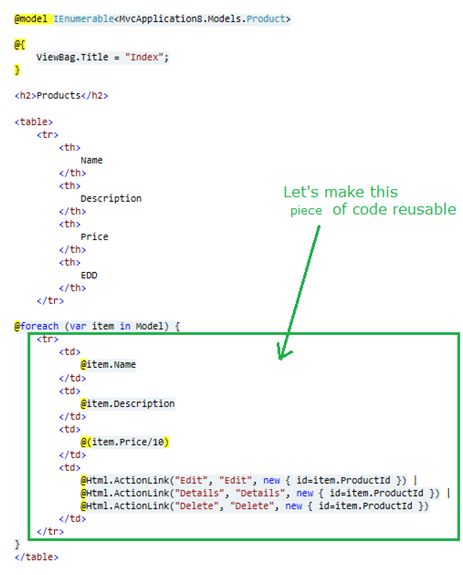
**TempData**

1. TempData is derived from TempDataDictionary class
2. TempData is used to pass data from the current request to the next request
3. It keeps the information for the time of an HTTP Request. This means only from one page to another. It helps to maintain the data when we move from one controller to another controller or from one action to another action
4. It requires typecasting for complex data type and checks for null values to avoid error. Generally, it is used to store only one time messages like the error messages and validation messages

**Question 10:What is Partial View in MVC?**

If you want to reuse a view in your web application, you can go for the partial view concept. Partial view is like a regular view with a file extension .cshtml. We can use partial views in a situation where we need a header, footer reused for an MVC web application. We can say that it’s like a user control concept in ASP.NET.

@Html.Partial("~/Views/Shared/\_Product.cshtml", product);

**Answer:** A partial view is a chunk of HTML that can be safely inserted into an existing DOM. Most commonly, partial views are used to componentize Razor views and make them easier to build and update. Partial views can also be returned directly from controller methods. In this case, the browser still receives text/html content but not necessarily HTML content that makes up an entire page. As a result, if a URL that returns a partial view is directly invoked from the address bar of a browser, an incomplete page may be displayed. This may be something like a page that misses title, script and style sheets. However, when the same URL is invoked via script, and the response is used to insert HTML within the existing DOM, then the net effect for the end user may be much better and nicer.  
  
Partial view is a reusable view (like a user control) which can be embedded inside other view. For example, let’s say all the pages of your site have a standard structure with left menu, header, and footer as in the following image,  
  
  
  


**Question 11:Explain what is the difference between View and Partial View?  
  
Answer:   
  
View:**

* It contains the layout page.
* Before any view is rendered, viewstart page is rendered.
* View might have markup tags like body, html, head, title, meta etc.
* View is not lightweight as compare to Partial View.

**Partial View:**

* It does not contain the layout page.
* Partial view does not verify for a viewstart.cshtml.We cannot put common code for a partial view within the viewStart.cshtml.page.
* Partial view is designed specially to render within the view and just because of that it does not consist any mark up.
* We can pass a regular view to the RenderPartial method.

**Question 12: What are HTML helpers in MVC?  
  
Answer:**

An HTML Helper is just a method that returns a HTML string. The string can represent any type of content that you want. For example, you can use HTML Helpers to render standard HTML tags like HTML <input>, <button> and <img> tags etc.

You can also create your own HTML Helpers to render more complex content such as a menu strip or an HTML table for displaying database data.

These helpers are used to render the most common types of HTML elements like as HTML text boxes, checkboxes etc. A list of most common standard html helpers is given below:

TextBox

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

TextArea

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

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

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

**Question 13: Explain attribute based routing in MVC?  
  
Answer:**   
  
In ASP.NET MVC 5.0 we have a new attribute route,cBy using the "Route" attribute we can define the URL structure. For example in the below code we have decorated the "GotoAbout" action with the route attribute. The route attribute says that the "GotoAbout" can be invoked using the URL structure "Users/about".  
  
**Hide Copy Code**

1. **public** **class** HomeController: Controller
2. {
3. [Route("Users/about")]
4. publicActionResultGotoAbout()
5. {
6. **return** View();
7. }
8. }

* [Attribute Based Routing in ASP.Net MVC 5](http://www.c-sharpcorner.com/UploadFile/bhushangawale/attribute-based-routing-in-Asp-Net-mvc-5/)

**Question 14: What is TempData in MVC?**  
**Answer:**TempData is a dictionary object to store data temporarily. It is a TempDataDictionary class type and instance property of the Controller base class.   
  
TempData is able to keep data for the duration of a HTP request, in other words it can keep live data between two consecutive HTTP requests. It will help us to pass the state between action methods. TempData only works with the current and subsequent request. TempData uses a session variable to store the data. TempData Requires type casting when used to retrieve data.  
  
TempDataDictionary is inherited from the IDictionary<string, object>, ICollection<KeyValuePair<string, object>>, IEnumerable<KeyValuePair<string, object>> and IEnumerable interfaces.  
  
**Example**

1. **public** ActionResult FirstRequest()
2. {
3. List < **string** > TempDataTest = **new** List < **string** > ();
4. TempDataTest.Add("Tejas");
5. TempDataTest.Add("Jignesh");
6. TempDataTest.Add("Rakesh");
7. TempData["EmpName"] = TempDataTest;
8. **return** View();
9. }
10. **public** ActionResult ConsecutiveRequest()
11. {
12. List < **string** > modelData = TempData["EmpName"] **as** List < **string** > ;
13. TempData.Keep();
14. **return** View(modelData);
15. }

* [All About the TempData in MVC](http://www.c-sharpcorner.com/UploadFile/ff2f08/all-about-the-tempdata-in-mvc/)

**Question 15: What is Razor in MVC?**  
**Answer:**ASP.NET MVC has always supported the concept of "view engines" - which are the pluggable modules that implement different template syntax options. The "default" view engine for ASP.NET MVC uses the same .aspx/.ascx/. master file templates as ASP.NET Web Forms. Other popular ASP.NET MVC view engines are Spart&Nhaml.  
  
MVC 3 has introduced a new view engine called Razor.  
  
**Why is Razor?**

1. Compact & Expressive.
2. Razor minimizes the number of characters and keystrokes required in a file, and enables a fast coding workflow. Unlike most template syntaxes, you do not need to interrupt your coding to explicitly denote server blocks within your HTML. The parser is smart enough to infer this from your code. This enables a really compact and expressive syntax which is clean, fast and fun to type.
3. Easy to Learn: Razor is easy to learn and enables you to quickly be productive with a minimum of effort. We can use all your existing language and HTML skills.
4. Works with any Text Editor: Razor doesn't require a specific tool and enables you to be productive in any plain old text editor (notepad works great).
5. Has great Intellisense:
6. Unit Testable: The new view engine implementation will support the ability to unit test views (without requiring a controller or web-server, and can be hosted in any unit test project - no special app-domain required).

* [Brief Introduction to MVC3](http://www.c-sharpcorner.com/UploadFile/aravindbenator/brief-introduction-of-mvc3/)

**Question 16: Differences between Razor and ASPX View Engine in MVC?  
  
Answer: Razor View Engine VS ASPX View Engine**

|  |  |
| --- | --- |
| **Razor View Engine** | **ASPX View Engine (Web form view engine)** |
| 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 uses cshtml with C# and vbhtml with vb for views, partial view, templates and layout pages. | 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 Controls or templates and master extensions for layout/master pages. |
| The Razor View Engine is an advanced view engine that was introduced with MVC 3.0. This is not a new language but it is markup. | A web form view engine is the default view engine and available from the beginning of MVC |
| Razor has a syntax that is very compact and helps us to reduce typing. | The web form view engine has syntax that is the same as an ASP.Net forms application. |
| The Razor View Engine uses @ to render server-side content. | The ASPX/web form view engine uses "<%= %>" or "<%: %>" to render server-side content. |
| By default all text from an @ expression is HTML encoded. | There is a different syntax ("<%: %>") to make text HTML encoded. |
| Razor does not require the code block to be closed, the Razor View Engine parses itself and it is able to decide at runtime which is a content element and which is a code element. | A web form view engine requires the code block to be closed properly otherwise it throws a runtime exception. |
| The Razor View Engine prevents Cross Site Scripting (XSS) attacks by encoding the script or HTML tags before rendering to the view. | A web form View engine does not prevent Cross Site Scripting (XSS) attack. |
| 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. |
| Razor uses "@\* â€¦ \*@" for multiline comments. | The ASPX View Engine uses "<!--...-->" for markup and "/\* â€¦ \*/" for C# code. |
| There is only three transition characters with the Razor View Engine. | There are only three transition characters with the Razor View Engine. |

The Razor View Engine is a bit slower than the ASPX View Engine.  
  
**Conclusion**  
Razor provides a new view engine with streamlined code for focused templating. Razor's syntax is very compact and improves readability of the markup and code. By default MVC supports ASPX (web forms) and Razor View Engine. MVC also supports third-party view engines like Spark, Nhaml, NDjango, SharpDOM and so on. ASP.NET MVC is open source.

* [ASPX View Engine VS Razor View Engine](http://www.c-sharpcorner.com/UploadFile/ff2f08/aspx-view-engine-vs-razor-view-engine/)

**Question 17: What are the Main Razor Syntax Rules?  
  
Answer:**

* Razor code blocks are enclosed in @{ ... }
* Inline expressions (variables and functions) start with @
* Code statements end with semicolon
* Variables are declared with the var keyword
* Strings are enclosed with quotation marks
* C# code is case sensitive
* C# files have the extension .cshtml

**C# Example:**

1. <!-- Single statement block -->
2. @ {
3. varmyMessage = "Hello World";
4. }
5. <!-- Inline expression or variable -->
6. < p > The value of myMessage **is**: @myMessage < /p>
7. <!-- Multi-statement block -->
8. @ {
9. var greeting = "Welcome to our site!";
10. varweekDay = DateTime.Now.DayOfWeek;
11. vargreetingMessage = greeting + " Here in Huston it is: " + weekDay;
12. } < p > The greeting **is**: @greetingMessage < /p>

* [Introduction to Microsoft ASP.NET MVC 3 Razor View Engine](http://www.c-sharpcorner.com/UploadFile/c25b6d/introduction-to-microsoft-Asp-Net-mvc-3-razor-view-engine/)

**Question18: How do you implement Forms authentication in MVC?  
  
Answer:**  
Authentication is giving access to the user for a specific service by verifying his/her identity using his/her credentials like username and password or email and password. It assures that the correct user is authenticated or logged in for a specific service and the right service has been provided to the specific user based on their role that is nothing but authorization.  
  
ASP.NET forms authentication occurs after IIS authentication is completed. You can configure forms authentication by using forms element with in web.config file of your application. The default attribute values for forms authentication are shown below:

1. <system.web>
2. <authenticationmode="Forms">
3. <formsloginUrl="Login.aspx" protection="All" timeout="30" name=".ASPXAUTH" path="/" requireSSL="false" slidingExpiration="true" defaultUrl="default.aspx" cookieless="UseDeviceProfile" enableCrossAppRedirects="false" />
4. </authentication>
5. </system.web>

The FormsAuthentication class creates the authentication cookie automatically when SetAuthCookie() or RedirectFromLoginPage() methods are called. The value of authentication cookie contains a string representation of the encrypted and signed FormsAuthenticationTicket object.

* [Form Authentication in MVC 5: Part 1](http://www.c-sharpcorner.com/UploadFile/219d4d/form-authentication-in-mvc-5-part-1/)

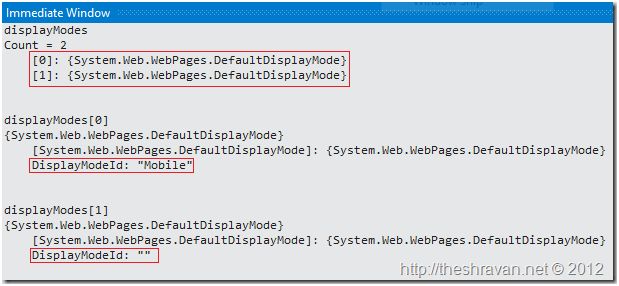
**Question 19: Explain Areas in MVC?**  
By convention in ASP.NET MVC project template, to separate application code based on responsibilities, we have top level folders as Controllers, Views, and Models. In Controllersfolder we can have several controllers, each of them is focused on specific group of functionalities. Inside top level Viewsfolder, there are folders, corresponding to controllers (having the same name), and containing views files for the controller. Corresponding to Actionnames in controller, we have view files (.cshtml file) in it. Finally in top level Modelsfolder we have models which manages the behaviour and data for the application domain.

This approach works well for small applications having less number of controllers and views. As the number of controllers and views grow, we feel the need of grouping the controllers and views. For large applications, we often face such problem to organize the code effectively. In such cases, Areas help us to organize the code.

#### Areas are logical grouping of Controller, Models and Views and other related folders for a module in MVC applications. By convention, a top Areas folder can contain multiple areas. Using areas, we can write more maintainable code for an application cleanly separated according to the modules.

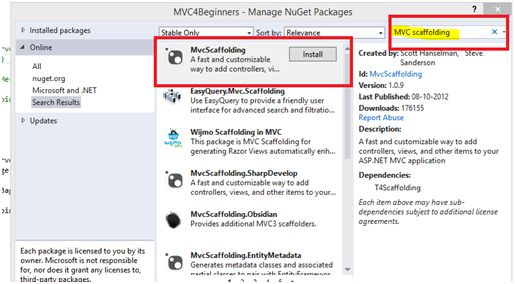
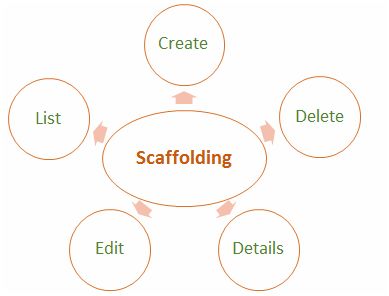
**Question 20: Explain the need of display mode in MVC?**  
**Answer:**DisplayModes give you another level of flexibility on top of the default capabilities we saw in the last section. DisplayModes can also be used along with the previous feature so we will simply build off of the site we just created.  
  
**Using display modes involves in 2 steps:**

1. We should register Display Mode with a suffix for particular browser using “DefaultDisplayMode”e class inApplication\_Start() method in the Global.asax file.
2. View name for particular browser should be appended with suffix mentioned in first step.

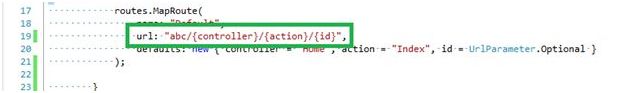
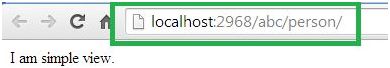


1. Desktop browsers (without any suffix. e.g.: Index.cshtml, \_Layout.cshtml).
2. Mobile browsers (with a suffix “Mobile”. e.g.: Index.Mobile.cshtml,Layout.Mobile.cshtml)  
   If you want design different pages for different mobile device browsers (any different browsers) and render them depending on the browser requesting. To handle these requests you can register custom display modes. We can do that using DisplayModeProvider.Instance.Modes.Insert(int index, IDisplayMode item) method.

* [Display Mode Provider in MVC 5 Application](http://www.c-sharpcorner.com/UploadFile/b696c4/display-mode-provider-in-mvc-5-application/)

**Question 21: Explain the concept of MVC Scaffolding?**  
**Answer:**ASP.NET Scaffolding is a code generation framework for ASP.NET Web applications. Visual Studio 2013 includes pre-installed code generators for MVC and Web API projects. You add scaffolding to your project when you want to quickly add code that interacts with data models. Using scaffolding can reduce the amount of time to develop standard data operations in your project.  
Scaffolding consists of page templates, entity page templates, field page templates, and filter templates. These templates are called Scaffold templates and allow you to quickly build a functional data-driven Website.  
  
  
  
**Scaffolding Templates:**  
  
  
  
**Create**: It creates a View that helps in creating a new record for the Model. It automatically generates a label and input field for each property in the Model.  
  
**Delete**: It creates a list of records from the model collection along with the delete link with delete record.  
  
**Details**: It generates a view that displays the label and an input field of the each property of the Model in the MVC framework.  
  
**Edit**: It creates a View with a form that helps in editing the current Model. It also generates a form with label and field for each property of the model.  
  
**List**: It generally creates a View with the help of a HTML table that lists the Models from the Model Collection. It also generates a HTML table column for each property of the Model.

* [Terminologies in MVC: Part 3 (Scaffolding)](http://www.c-sharpcorner.com/UploadFile/2072a9/terminologies-in-mvc-part-3scaffolding/)

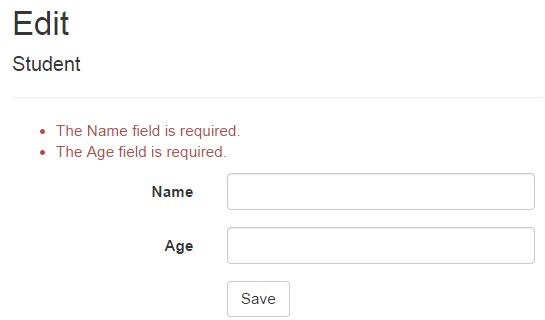
**Question 22:What is Route Constraints in MVC?**  
**Answer:**Routing is a great feature of MVC, it provides a REST based URL that is very easy to remember and improves page ranking in search engines.  
  
This article is not an introduction to Routing in MVC, but we will learn a few features of routing and by implementing them we can develop a very flexible and user-friendly application. So, let's start without wasting valuable time.  
  
**Add constraint to URL**  
  
This is very necessary for when we want to add a specific constraint to our URL. Say, for example we want a [URL](http://www.myhost/myconstraint/controller/action).  
  
So, we want to set some constraint string after our host name. Fine, let's see how to implement it.  
  
It's very simple to implement, just open the RouteConfig.cs file and you will find the routing definition in that. And modify the routing entry as in the following. We will see that we have added “abc” before.  
  
  
  
Controller name, now when we browse we need to specify the string in the URL, as in the following:  
  


* [Route Constraints in MVC](http://www.c-sharpcorner.com/UploadFile/dacca2/route-constraints-in-mvc/)

**Question 23: What is Razor View Engine in MVC?**  
**Answer:**ASP.NET MVC has always supported the concept of "view engines" that are the pluggable modules that implement various template syntax options. The "default" view engine for ASP.NET MVC uses the same .aspx/.ascx/.master file templates as ASP.NET Web Forms. In this article I go through the Razor View Engine to create a view of an application. "Razor" was in development beginning in June 2010 and was released for Microsoft Visual Studio in January 2011.  
  
Razor is not a new programming language itself, but uses C# syntax for embedding code in a page without the ASP.NET delimiters: <%= %>. It is a simple-syntax view engine and was released as part of ASP.NET MVC 3. The Razor file extension is "cshtml" for the C# language. It supports TDD (Test Driven Development) because it does not depend on the System.Web.UI.Page class.

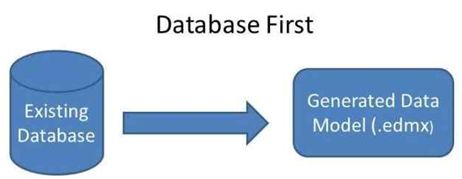
* [Getting Started with Razor View Engine in MVC 3](http://www.c-sharpcorner.com/UploadFile/3d39b4/getting-started-with-razor-view-engine-in-mvc-3/)

**Question 25: What is Bundling and Minification in MVC?  
  
Answer:**Bundling and minification are two new techniques introduced to improve request load time. It improves load time by reducing the number of requests to the server and reducing the size of requested assets (such as CSS and JavaScript).  
  
**Bundling**: It lets us combine multiple JavaScript (.js) files or multiple cascading style sheet (.css) files so that they can be downloaded as a unit, rather than making individual HTTP requests.  
  
**Minification**: It squeezes out whitespace and performs other types of compression to make the downloaded files as small as possible. At runtime, the process identifies the user agent, for example IE, Mozilla, etc. and then removes whatever is specific to Mozilla when the request comes from IE.

**Question 26:What is Validation Summary in MVC?**  
**Answer:**The ValidationSummary helper method generates an unordered list (ul element) of validation messages that are in the ModelStateDictionary object.  
  
The ValidationSummary can be used to display all the error messages for all the fields. It can also be used to display custom error messages. The following figure shows how ValidationSummary displays the error messages.  
  
  
  
**ValidationSummary() Signature:**  
MvcHtmlStringValidateMessage(bool excludePropertyErrors, string message, object htmlAttributes)  
  
**Display field level error messages using ValidationSummary:**By default, ValidationSummary filters out field level error messages. If you want to display field level error messages as a summary then specify excludePropertyErrors = false.  
 **Example: ValidationSummary to display field errors:**  
@Html.ValidationSummary(false, "", new { @class = "text-danger" })  
  
So now, the following Edit view will display error messages as a summary at the top. Please make sure that you don't have a ValidationMessageFor method for each of the fields.  
  


**Question 27:What is Database First Approach in MVC using Entity Framework?  
  
Answer:**   
  
Database First Approach is an alternative to the Code First and Model First approaches to the Entity Data Model which creates model codes (classes,properties, DbContextetc) from the database in the project and that classes behaves as the link between database and controller.  
  
There are the following approach which is used to connect with database to application.

* Database First
* Model First
* Code First

  
  
Database first is nothing but only a approach to create web application where database is available first and can interact with the database. In this database, database is created first and after that we manage the code. The Entity Framework is able to generate a business model based on the tables and columns in a relational database.

**Question 30:What are the methods of handling an Error in MVC?**  
**Answer:**Exception handling may be required in any application, whether it is a web application or a Windows Forms application.   
  
ASP.Net MVC has an attribute called "HandleError" that provides built-in exception filters. The HandleError attribute in ASP.NET MVC can be applied over the action method as well as Controller or at the global level. The HandleError attribute is the default implementation of IExceptionFilter. When we create a MVC application, the HandleError attribute is added within the Global.asax.cs file and registered in the Application\_Start event.

1. **public** **static** **void** RegisterGlobalFilters(GlobalFilterCollection filters)
2. {
3. filters.Add(**new** HandleErrorAttribute());
4. }
5. **protected** **void** Application\_Start()
6. {
7. AreaRegistration.RegisterAllAreas();
8. RegisterGlobalFilters(GlobalFilters.Filters);
9. RegisterRoutes(RouteTable.Routes);
10. }

**Important properties of HandleError attribute**  
  
The HandleError Error attribute has a couple for properties that are very useful in handling the exception.  
  
**ExceptionType**: Type of exception to be catch. If this property is not specified then the HandleError filter handles all exceptions.  
  
**View**: Name of the view page for displaying the exception information.  
  
**Master**: Master View for displaying the exception.  
  
**Order**:   
  
Order in which the action filters are executed. The Order property has an integer value and it specifies the priority from 1 to any positive integer value. 1 means highest priority and the greater the value of the integer is, the lower is the priority of the filter.  
  
**AllowMultiple**: It indicates whether more than one instance of the error filter attribute can be specified.  
  
**Example**

1. [HandleError(View = "Error")]
2. **public** **class** HomeController: Controller
3. {
4. **public** ActionResult Index()
5. {
6. ViewBag.Message = "Welcome to ASP.NET MVC!";
7. **int** u = Convert.ToInt32(""); // Error line
8. **return** View();
9. }
10. }

HandleError Attribute at Action Method Level,

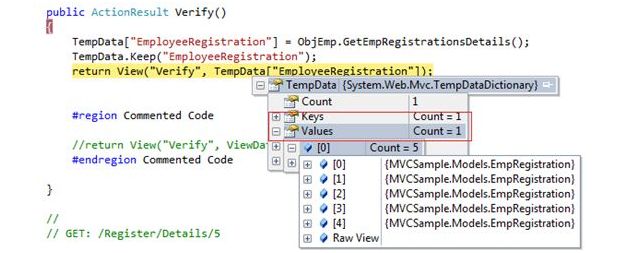
1. [HandleError(View = "Error")]
2. **public** ActionResult Index()
3. {
4. ViewBag.Message = "Welcome to ASP.NET MVC!";
5. **int** u = Convert.ToInt32(""); // Error line
6. **return** View();
7. }

* [Exception or Error Handling in ASP.Net MVC Using HandleError Attribute](http://www.c-sharpcorner.com/UploadFile/ff2f08/exception-or-error-handling-in-Asp-Net-mvc-using-handleerror/)

**Question 33:What is ViewStart?**  
  
**Answer:**   
  
Razor View Engine introduced a new layout named \_ViewStart which is applied on all view automatically. Razor View Engine firstly executes the \_ViewStart and then start rendering the other view and merges them.   
  
**Example of Viewstart:**

1. @ {
2. Layout = "~/Views/Shared/\_v1.cshtml";
3. } < !DOCTYPE html >
4. < html >
5. < head >
6. < meta name = "viewport"
7. content = "width=device-width" / >
8. < title > ViewStart < /title> < /head> < body >
9. < /body> < /html>

**Question 35: What is TempData?**TempData is a dictionary object derived from the TempDataDictionary class.

* TempData is used to pass data from the current request to a subsequent request, in other words in the case of redirection.
* The life of a TempData is very short and it retains its value for a short period of time.
* It requires typecasting for complex data type as I’ve used in my example:
* @foreach (var item in (List<MVCSample.Models.EmpRegistration>)TempData["EmployeeRegistration"])
* You can retain its value using the Keep method for subsequent requests.  
    
  

**Question 36: How to use ViewBag?**  
**Answer:**ViewBag is dynamic property that takes advantage of new dynamic features in C# 4.0. It's also used to pass data from a controller to a view. In short, The ViewBag property is simply a wrapper around the ViewData that exposes the ViewData dictionary as a dynamic object. Now create an action method "StudentSummary" in the "DisplayDataController" controller that stores a Student class object in ViewBag.

1. **public** ActionResult StudentSummary()
2. {
3. var student = **new** Student()
4. {
5. Name = "Sandeep Singh Shekhawat",
6. Age = 24,
7. City = "Jaipur"
8. };
9. ViewBag.Student = student;
10. **return** View();
11. }

Thereafter create a view StudentSummary ("StudentSummary.cshtml") that shows student object data. ViewBag does not require typecasting for complex data type so you can directly access the data from ViewBag.

1. @ {
2. ViewBag.Title = "Student Summary";
3. var student = ViewBag.Student;
4. }
5. < table >
6. < tr >
7. < th > Name < /th> < th > Age < /th> < th > City < /th> < /tr> < tr >
8. < td > @student.Name < /td> < td > @student.Age < /td> < td > @student.City < /td> < /tr>
9. < /table>

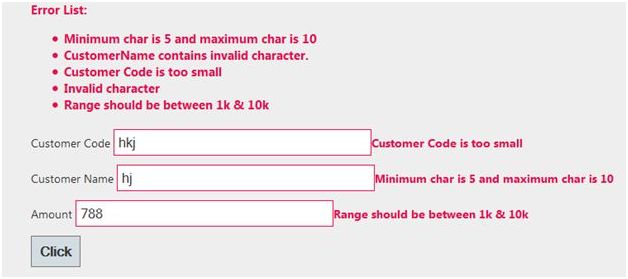
Here we used one more thing, "ViewBag.Title", that shows the title of the page.

* [Displaying Data on View From Controller](http://www.c-sharpcorner.com/UploadFile/3d39b4/displaying-data-on-view-from-controller/)

**Question 37: What are the Difference between ViewBag&ViewData?**  
**Answer: Difference between ViewBag&ViewData?**

* ViewData is a dictionary of objects that is derived from ViewDataDictionary class and accessible using strings as keys.
* ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0.
* ViewData requires typecasting for complex data type and check for null values to avoid error.
* ViewBag doesn't require typecasting for complex data type.
* Calling of ViewBag is:  
    
  ViewBag.Name = "Yogesh";Calling of ViewDatais :ViewData["Name"] = "yogesh";

**Question 38: What is Data Annotation Validator Attributes in MVC?  
  
Answer: Using the Data Annotation Validator Attributes:**  
DataAnnotation plays a vital role in added validation to properties while designing the model itself. This validation can be added for both the client side and the server side.  
  
You understand that decorating the properties in a model with an Attribute can make that property eligible for Validation.  
  
Some of the DataAnnotation used for validation are given below:



1. **Required**  
   Specify a property as required.  
   1. [Required(ErrorMessage="CustomerName is mandatory")]
2. **RegularExpression**  
   Specifies the regular expression to validate the value of the property.  
   1. [RegularExpression("[a-z]", ErrorMessage = "Invalid character")]
3. **Range**  
   Specifies the Range of values between which the property values are checked.  
   1. [Range(1000,10000,ErrorMessage="Range should be between 1k & 10k")]
4. **StringLength**  
   Specifies the Min & Max length for a string property.  
   1. [StringLength(50, MinimumLength = 5, ErrorMessage = "Minimum char is 5 and maximum char is 10")]
5. **MaxLength**  
   Specifies the Max length for the property value.  
   1. [MaxLength(10,ErrorMessage="Customer Code is exceeding")]
6. **MinLength**  
   It is used to check for minimum length.  
   1. [MinLength(5, ErrorMessage = "Customer Code is too small")]

**Question 39:How can we done Custom Error Page in MVC?  
  
Answer:**The HandleErrorAttribute allows you to use a custom page for this error. First you need to update your web.config file to allow your application to handle custom errors.

1. <system.web>
2. <customErrors mode="On">
3. </system.web>

Then, your action method needs to be marked with the atttribute.

1. [HandleError]
2. **public** **class** HomeController: Controller
3. {
4. [HandleError]
5. publicActionResultThrowException()
6. {
7. **throw** **new** ApplicationException();
8. }
9. }

By calling the ThrowException action, this would then redirect the user to the default error page. In our case though, we want to use a custom error page and redirect the user there instead.So, let's create our new custom view page.  
  
  
  
Next, we simply need to update the HandleErrorAttribute on the action method.

1. [HandleError]
2. **public** **class** HomeController: Controller
3. {
4. [HandleError(View = "CustomErrorView")]
5. publicActionResultThrowException()
6. {
7. **throw** **new** ApplicationException();
8. }
9. }

* [Custom Error Page in ASP.NET MVC](http://www.c-sharpcorner.com/UploadFile/618722/custom-error-page-in-Asp-Net-mvc/)

**Question 40: Server Side Validation in MVC?**  
**Answer:**The ASP.NET MVC Framework validates any data passed to the controller action that is executing, It populates a ModelState object with any validation failures that it finds and passes that object to the controller. Then the controller actions can query the ModelState to discover whether the request is valid and react accordingly.  
  
I will use two approaches in this article to validate a model data. One is to manually add an error to the ModelState object and another uses the Data Annotation API to validate the model data.  
  
**Approach 1: Manually Add Error to ModelState object**  
  
I create a User class under the Models folder. The User class has two properties "Name" and "Email". The "Name" field has required field validations while the "Email" field has Email validation. So let's see the procedure to implement the validation. Create the User Model as in the following:

1. **namespace** ServerValidation.Models
2. {
3. **public** **class** User
4. {
5. **public** **string** Name
6. {
7. **get**;
8. **set**;
9. }
10. **public** **string** Email
11. {
12. **get**;
13. **set**;
14. }
15. }
16. }

After that I create a controller action in User Controller (UserController.cs under Controllers folder). That action method has logic for the required validation for Name and Email validation on the Email field. I add an error message on ModelState with a key and that message will be shown on the view whenever the data is not to be validated in the model.

1. **using** System.Text.RegularExpressions;
2. **using** System.Web.Mvc;
3. **namespace** ServerValidation.Controllers
4. {
5. **public** **class** UserController: Controller
6. {
7. **public** ActionResult Index()
8. {
9. **return** View();
10. }
11. [HttpPost]
12. **public** ActionResult Index(ServerValidation.Models.User model)
13. {
15. **if** (**string**.IsNullOrEmpty(model.Name))
16. {
17. ModelState.AddModelError("Name", "Name is required");
18. }
19. **if** (!**string**.IsNullOrEmpty(model.Email))
20. {
21. **string** emailRegex = @ "^([a-zA-Z0-9\_\-\.]+)@((\[[0-9]{1,3}" +
22. @ "\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\" +
23. @ ".)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";
24. Regex re = **new** Regex(emailRegex);
25. **if** (!re.IsMatch(model.Email))
26. {
27. ModelState.AddModelError("Email", "Email is not valid");
28. }
29. } **else** {
30. ModelState.AddModelError("Email", "Email is required");
31. }
32. **if** (ModelState.IsValid)
33. {
34. ViewBag.Name = model.Name;
35. ViewBag.Email = model.Email;
36. }
37. **return** View(model);
38. }
39. }
40. }

Thereafter I create a view (Index.cshtml) for the user input under the User folder.

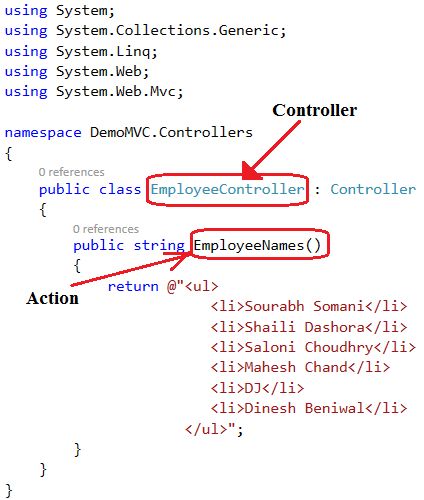
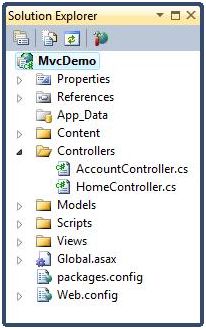
1. @model ServerValidation.Models.User
2. @ {
3. ViewBag.Title = "Index";
4. }
5. @**using**(Html.BeginForm())
6. {
7. **if** (@ViewData.ModelState.IsValid)
8. {
9. **if** (@ViewBag.Name != **null**)
10. { < b >
11. Name: @ViewBag.Name < br / >
12. Email: @ViewBag.Email < /b>
13. }
14. } < fieldset >
15. < legend > User < /legend>  < div **class** = "editor-label" >
16. @Html.LabelFor(model => model.Name) < /div> < div **class** = "editor-field" >
17. @Html.EditorFor(model => model.Name)
18. @**if**(!ViewData.ModelState.IsValid)
19. {
20. < span **class** = "field-validation-error" > @ViewData.ModelState["Name"].Errors[0].ErrorMessage < /span>
22. }
23. < /div>  < div **class** = "editor-label" >
25. @Html.LabelFor(model => model.Email) < /div> < div **class** = "editor-field" >
26. @Html.EditorFor(model => model.Email)
27. @**if**(!ViewData.ModelState.IsValid)
28. {
29. < span **class** = "field-validation-error" > @ViewData.ModelState["Email"].Errors[0].ErrorMessage < /span>
30. }
31. < /div> < p >
32. < input type = "submit"
33. value = "Create" / >
34. < /p> < /fieldset>
35. }

**Question 41: What is the use of remote validation in MVC?  
  
Answer:**Remote validation is the process where we validate specific data posting data to a server without posting the entire form data to the server. Let's see an actual scenario, in one of my projects I had a requirement to validate an email address, whetehr it already exists in the database. Remote validation was useful for that; without posting all the data we can validate only the email address supplied by the user.  
  
**Practical Explanation**  
Let's create a MVC project and name it accordingly, for me its “TestingRemoteValidation”. Once the project is created let's create a model named UserModel that will look like:

1. **public** **class** UserModel
2. {
3. [Required]
4. **public** **string** UserName
5. {
6. **get**;
7. **set**;
8. }
9. [Remote("CheckExistingEmail", "Home", ErrorMessage = "Email already exists!")]
10. **public** **string** UserEmailAddress
11. {
12. **get**;
13. **set**;
14. }
15. }

Let's get some understanding of the remote attribute used, so the very first parameter “CheckExistingEmail” is the the name of the action. The second parameter “Home” is referred to as controller so to validate the input for the UserEmailAddress the “CheckExistingEmail” action of the “Home” controller is called and the third parameter is the error message. Let's implement the “CheckExistingEmail” action result in our home controller.

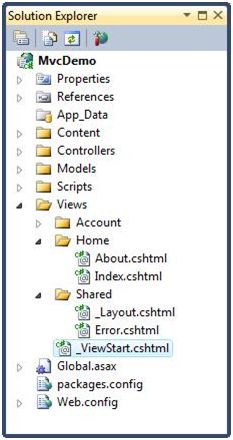
1. **public** ActionResult CheckExistingEmail(**string** UserEmailAddress)
2. {
3. **bool** ifEmailExist = **false**;
4. **try**
5. {
6. ifEmailExist = UserEmailAddress.Equals("mukeshknayak@gmail.com") ? **true** : **false**;
7. **return** Json(!ifEmailExist, JsonRequestBehavior.AllowGet);
8. } **catch** (Exception ex)
9. {
10. **return** Json(**false**, JsonRequestBehavior.AllowGet);
11. }
12. }

**Question 44: Define Controller in MVC?**  
**Answer:**The controller provides model data to the view, and interprets user actions such as button clicks. The controller depends on the view and the model. In some cases, the controller and the view are the same object.  
  
  
  
**The Controllers Folder:**  
  
The Controllers Folder contains the controller classes responsible for handling user input and responses. MVC requires the name of all controllers to end with "Controller".  
  
In our example, Visual Web Developer has created the following files: HomeController.cs (for the Home and About pages) and AccountController.cs (For the Log On pages):  
  


* [ASP.Net MVC Controller](http://www.c-sharpcorner.com/UploadFile/75a48f/asp-net-mvc-controller/)

**Question 45:Explain Model in MVC?**  
**Answer:**The model represents the data, and does nothing else. The model does NOT depend on the controller or the view. The MVC Model contains all application logic (business logic, validation logic, and data access logic), except pure view and controller logic. With MVC, models both hold and manipulate application data.  
  
**The Models Folder:**The Models Folder contains the classes that represent the application model.  
  
Visual Web Developer automatically creates an AccountModels.cs file that contains the models for application security.

* [Model in ASP.Net MVC : Part 1](http://www.c-sharpcorner.com/UploadFile/e95fe7/model-in-Asp-Net-mvc-part-1/)

**Question 46: Explain View in MVC?  
  
Answer:**A view is responsible for displaying all of, or a portion of, data for users. In simple terms, whatever we see on the output screen is a view.  
  
**The Views Folder:**The Views folder stores the files (HTML files) related to the display of the application (the user interfaces). These files may have the extensions html, asp, aspx, cshtml, and vbhtml, depending on the language content.  
  
The Views folder contains one folder for each controller. Visual Web Developer has created an Account folder, a Home folder, and a Shared folder (inside the Views folder). The Account folder contains pages for registering and logging in to user accounts. The Home folder is used for storing application pages like the home page and the about page. The Shared folder is used to store views shared between controllers (master pages and layout pages).  
  


* [ASP.Net MVC View](http://www.c-sharpcorner.com/UploadFile/75a48f/Asp-Net-mvc-view/)

**Question 47:What is Attribute Routing in MVC?  
  
Answer**:   
  
A route attribute is defined on top of an action method. The following is the example of a Route Attribute in which routing is defined where the action method is defined.  
  
In the following example, I am defining the route attribute on top of the action method

1. **public** **class** HomeController: Controller
2. {
3. //URL: /Mvctest
4. [Route(“Mvctest”)]
5. **public** ActionResult Index()
6. ViewBag.Message = "Welcome to ASP.NET MVC!";
7. **return** View();
8. }

**Question 48: Explain RenderSection in MVC?  
  
Answer:**   
  
RenderSection() is a method of the WebPageBase class. Scott wrote at one point, The first parameter to the "RenderSection()" helper method specifies the name of the section we want to render at that location in the layout template. The second parameter is optional, and allows us to define whether the section we are rendering is required or not. If a section is "required", then Razor will throw an error at runtime if that section is not implemented within a view template that is based on the layout file (that can make it easier to track down content errors). It returns the HTML content to render.

1. <div id="body">
2. @RenderSection("featured", required: **false**)
3. <section **class**="content-wrapper main-content clear-fix">
4. @RenderBody()
5. </section>
6. </div>

**Question 49: What is GET and POST Actions Types?  
  
Answer:   
  
GET**  
GET is used to request data from a specified resource. With all the GET request we pass the URL which is compulsory, however it can take the following overloads.  
.get(url [, data ] [, success(data, textStatus, jqXHR) ] [, dataType ] ).done/.fail  
  
**POST**  
POST is used to submit data to be processed to a specified resource. With all the POST requests we pass the URL which is compulsory and the data, however it can take the following overloads.   
  
.post(url [, data ] [, success(data, textStatus, jqXHR) ] [, dataType ] )

# 3 Loosely coupled and tightly coupled in MVC

**Short Introduction Loose and Tight Coupling**

Loose Coupling means reducing dependencies of a class that use a different class directly. In tight coupling, classes and objects are dependent on one another. In general, tight coupling is usually bad because it reduces flexibility and re-usability of code and it makes changes much more difficult and impedes testability etc.

**Tight Coupling**

A Tightly Coupled Object is an object that needs to know quite a bit about other objects and are usually highly dependent on each other's interfaces. Changing one object in a tightly coupled application often requires changes to a number of other objects. In a small application we can easily identify the changes and there is less chance to miss anything. But in large applications these inter-dependencies are not always known by every programmer or there is a chance of overlooking changes. But each set of loosely coupled objects are not dependent on each other. (Stackoverfow-Jom George)

**Code**namespace  
TightCoupling  
{  
    public class Remote  
    {  
       private  Television Tv  { get; set;}  
       protected Remote()  
       {  
           Tv = new Television();  
       }

       static Remote()  
       {  
           \_remoteController = new Remote();  
       }  
       static Remote \_remoteController;  
       public static Remote Control  
       {  
           get  
           {  
               return \_remoteController;  
           }  
       }

         public void RunTv()  
         {  
             Tv.Start();  
         }  
    }  
}

**Difficulties**

Tight Coupling creates some difficulties. Here, the task of the control object, the object needs to be able to television, the television remote control is dependent on the other phrase. So, what's the harm of the following dependencies:

* TV without a remote control does not work.
* TV changes the control directly affected by this change.
* The Control can only control the TV, cannot control other devices.

**Loose Coupling**

Loose coupling is a design goal that seeks to reduce the inter-dependencies between components of a system with the goal of reducing the risk that changes in one component will require changes in any other component. Loose coupling is a much more generic concept intended to increase the flexibility of a system, make it more maintainable, and makes the entire framework more "stable".

**Code**  
public interface IRemote  
    {  
        void Run();  
    }  
public class Television : IRemote  
    {  
        protected Television()  
        {

        }

        static Television()  
        {  
            \_television = new Television();  
        }  
        private static Television \_television;  
        public static Television Instance  
        {  
            get  
            {  
                return \_television;  
            }  
        }

        public void Run()  
        {  
            Console.WriteLine("Television is started!");  
        }  
    }

We need a managing class that will produce an instance. The instance is generated from the implemented class. The Management Class constructor needs an interface which implements to any Class.

**Code**

public class Remote

    {

         IRemote \_remote;

        public Remote(IRemote remote)  
        {  
            \_remote = remote;  
        }  
  
        public void Run()  
        {  
            \_remote.Run();  
        }  
    }

**Usage  
  
class Program**{  
        static void Main(string[] args)  
        {

            Remote remote = new Remote(Television.Instance);  
            remote.Run();  
            Console.Read();  
        }  
    }  
  
**Advantages**

It will save you a lot of time for any project that isn't trivially small, where I define trivially small as less than a couple thousand lines of code (depending on the language). The reason is that once you get past super small projects, each change or update gets harder the more tightly coupled it is. Being loosely coupled enables you to keep moving forward, adding features, fixing bugs, etc.

At a certain point I think any program becomes a nightmare to maintain, update and add on to. The more loosely coupled the design is, the further that point is delayed. If it's tightly coupled, maybe after about 10,000 lines of code it becomes unmaintainable; adding features becomes impossible without essentially rewriting from scratch.

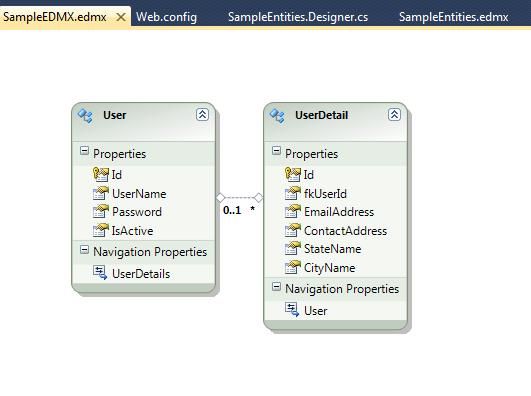
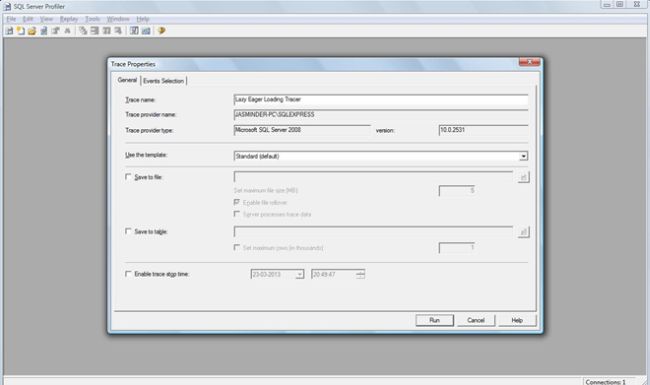
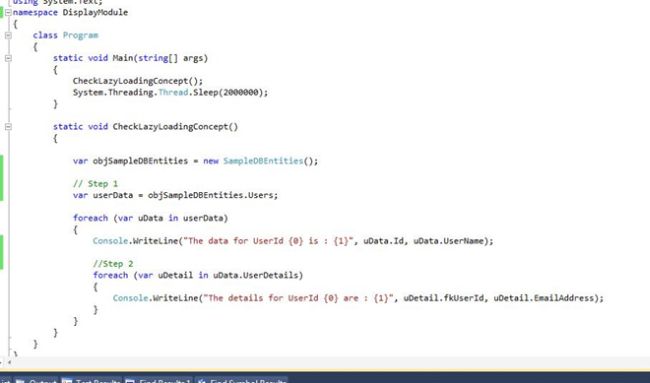
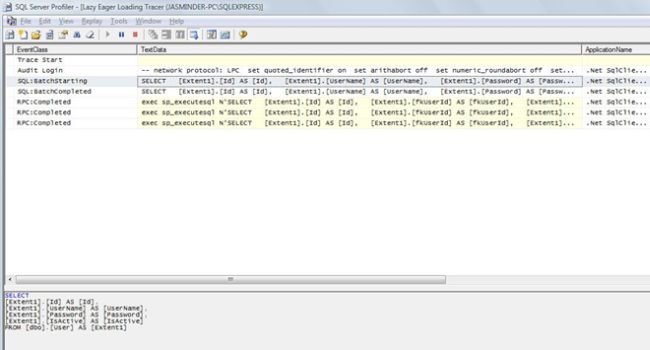
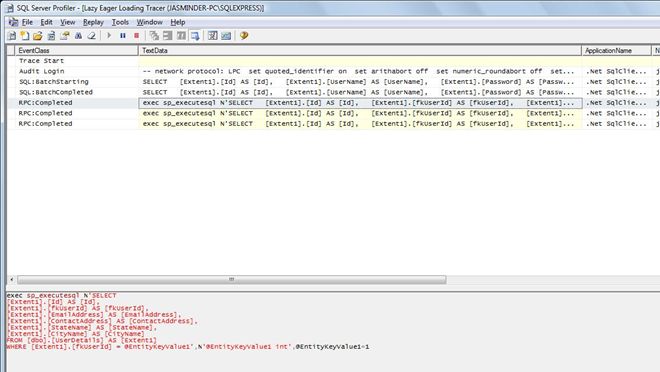
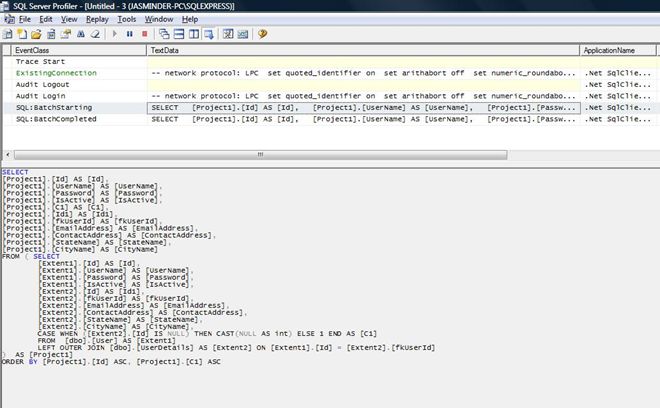
Being loosely coupled allows it to grow to 1,000,000 - 10,000,000 lines of code while still being able to make changes and add new features within a reasonable amount of time. These numbers aren't meant to be taken literally as they're just made up, but to provide a sense of where it becomes helpful. If you never need to update the program and it's fairly simple then sure, it's fine to be tightly coupled. It's even okay to start that way but understand that when it's time to separate stuff out, but you still need experience writing loosely coupled code to know at what point it becomes beneficial.;  (From Stackoverflow-Davy8)

* It improves testability.
* It helps you follow the GOF principle of Program to Interfaces, not implementations.
* The benefit is that it's much easier to swap other pieces of code/modules/objects/components when the pieces aren't dependent on one another.
* It's highly changeable. One module does not break other modules in unpredictable ways

**Summary**

As with all OO design, there are trade-offs you have to make; is it more important for you to have highly modular code that is easy to swap in and out? Or is it more important to have easily understandable code that is simpler? You'll have to decide that.

# 37 Lazy Loading vs Eager Loading

This article describes lazy and Eager Loading of data in the Entity Framework. This concept is one of the most important concepts related to the database hits in an application that can affect the performance of the application due to the basics behind these two. So let's start with it.  
  
In Entity Framework, it is very normal to have entities that are related to each other. For example, we may have a User table that contains basic user information like UserId, Username, Password, emailaddress etc. and another table UserDetails that contains contact details of the same user with attributes like Id, UserId (Foreign key from Users table), Contact Address, State, City etc. So here, the user table has a one-to-many relation with the UserDetails table.  
  
Entity Framework provides the ability to load the data of a parent entity as well as its related child entity at the same time, in other words when we load the data of the user table, we also get the related data of each User from the UserDetails table. This is known as Eager Loading.  
  
On the other hand, if we do not want to load the related entity data at the same time as when the main entity is fetched, we use Lazy Loading.  
  
To begin, we will create a sample project and add an .edmx model into it with two entities, User (parent entity) & UserDetails (child entity). We will also be using the SQL Server Profiler to check how the queries are executed in the database at the back end.  
  
1. So, hope you have set up the project and added the two entities into the database and the edmx model into the sample project as in the following:  
  
  
  
2. Next, start the SQL Profiler and select "File" -> "New Trace" to start a new trace. Connect using the settings of the SQL Server.  
  
3. Set the basic details like Trace Name as per your requirements and keep the default settings in the Events Selection Tab. Click on "Run".  
  
  
  
The preceding 3 steps will be common to the Lazy/Eager Loading. After that they will differ.  
  
**Here comes the Lazy Loading**  
  
4. Add the following code in your application. I have created a console based application for convenience.  
  
  
  
This is the case where the Lazy Loading is enabled by default. You can check the same in the Edmx.designer.cs file also.  
  
5. Now run the application. In the code above, note the Step 1. At this step, the data of only the main entity, the Users, is loaded but not that of the UserDetails. The data of the related entity is loaded when Step 2 is being executed; in other words when the nested foreach loop is executed.  
  
Let's see what the SQLProfiler has in store for us, for the queries executed for this process. See the screenshots below:  
  
  
  
In the above image, the highlighted line displays the query that is executed for the outer foreach that fetches the details of the Users entity. The query is being displayed at the bottom.  
  
  
  
In the second image, you can see that the selected query fetches the related data of the first record of the main entity, in other words  this query fetches the records from UserDetails with the fkUserId as 1. You will see the query at the bottom. Similarly, the following two queries will fetch the related records of the User entity records having the ids 2 and 3. You can check that by changing the selection.  
  
This means that to fetch the data of the related entity, database calls are being made again and again, after the data for main entity has been fetched, that could negatively hamper the efficiency of the application. So we need to take care of the scenarios when we should enable this feature .  
  
In the above case, if you do no need the UserDetails, then you can remove the foreach loop . You can remove the inner foreach loop and you will see that the SQL will not fetch the related entity data.  
  
**And now the Eager Loading**  
6. Now we will modify the code that we had in Step 1 to:  
  
var userData = objSampleDBEntities.Users.Include("UserDetails");  
  
Here we are explicitily fetching the data of the related entity, when the data of the main entity is being fetched.  
  
7. Start a new trace by following the steps that we performed earlier & run the application.  
  
8. Now observe the SQL profiler details. See the screenshot below:  
  
  
  
Now what happens in this case is that the SQL query is generated using the JOIN and it fetches the data of the related entity, in other words the UserDetails along with the main User entity data. You will see the query at the bottom of the selection that shows the join being applied at the back end by the SQL Server. This is known as Eager Loading which means loading the related entity data along with the data of the main entity.  
  
Now the question arises when to use what option:

1. Use Eager Loading when the data in related entities is too costly to be loaded at the cost of the queries being made to the database. In other words, fetch all of them at once along with the main entity using Eager Loading.
2. Use Lazy Loading when you only need the main entity data to be fetched and you know that the related data will not be required.

# **ASP.NET MVC’s Authorize Attribute**

**Authentication** and **Authorization** are perhaps the most prominent things in web-based application development nowadays. As developers, we always need to ensure at each instance whether we are showing up the authorized content to the user.

Traditionally in ASP.NET, we achieve these concepts by isolating critical modules from the rest of the application, i.e., by segregating ASPX pages in a folder under the control of a custom *web.config*file that redirects not-authenticated and unauthorized users to a custom login page.

Now **ASP.NET MVC** alleviates the pain in attaining the role based security just by a simple yet powerful attribute known as **Authorize**. Let’s dive deep into it but if you are new to ASP.NET MVC, it’s recommended to review a comprehensive list of [ASP.NET MVC Interview Questions for experienced](http://www.webdevelopmenthelp.net/2013/09/top-asp-net-mvc-interview-questions.html) and beginners available here.

# Authorize Attribute in ASP.NET MVC

In default, all the Controllers and Action methods are accessible by both Anonymous and Authenticated users.  
All the public methods inside the Controllers can be easily accessed if one knows the method name and the  
route pattern. Oops, that’s not a security breach, Wait!

**Note**: If you want to understand or learn all about Controllers and Action Methods in **ASP.NET MVC**, please follow [here](http://www.webdevelopmenthelp.net/2014/11/controllers-and-action-methods-asp-net-mvc.html).

So how to sway and protect the Controllers and Methods of **ASP.NET MVC**? There comes our attribute called Authorize into the play. Just by preceding this piece of word before any controllers or its action methods,  
protect them from unauthorized access:

Hide   Copy Code

[Authorize]

public ActionResult About()

{

ViewBag.Message = "Web Development Help!";

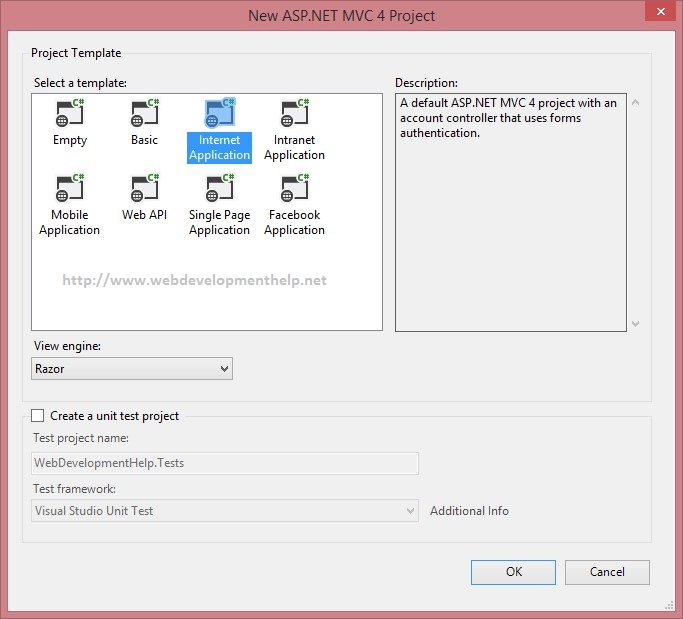
return View();

}

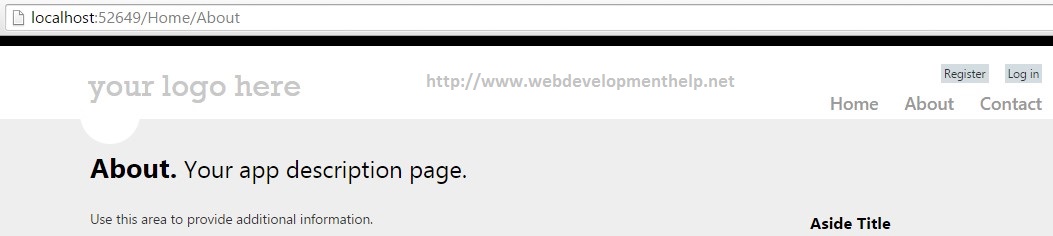
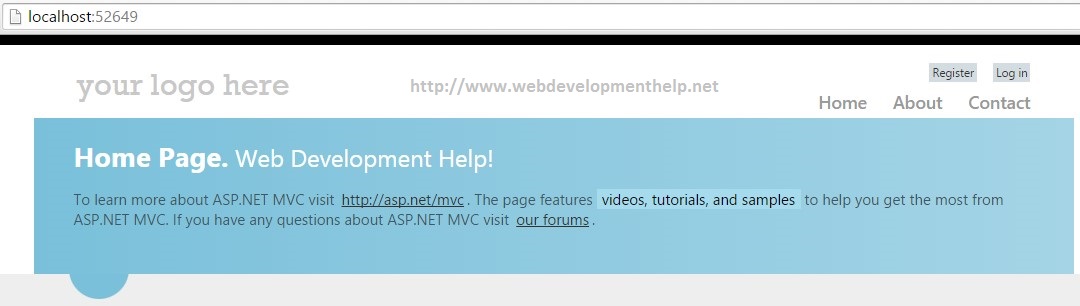
In the above snippet, we have decorated the action method About with [Authorize] attribute. So if any anonymous user, try routing to the above method, then he will be navigated to Login page.

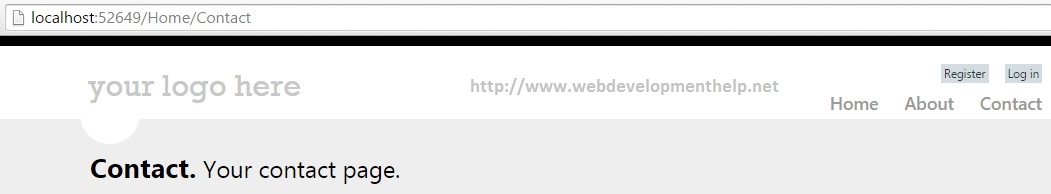
### **Let Us Dirty Our Hands**

* Open up Visual Studio 2012 and create an **ASP.NET MVC application** by choosing the Project template as Internet.



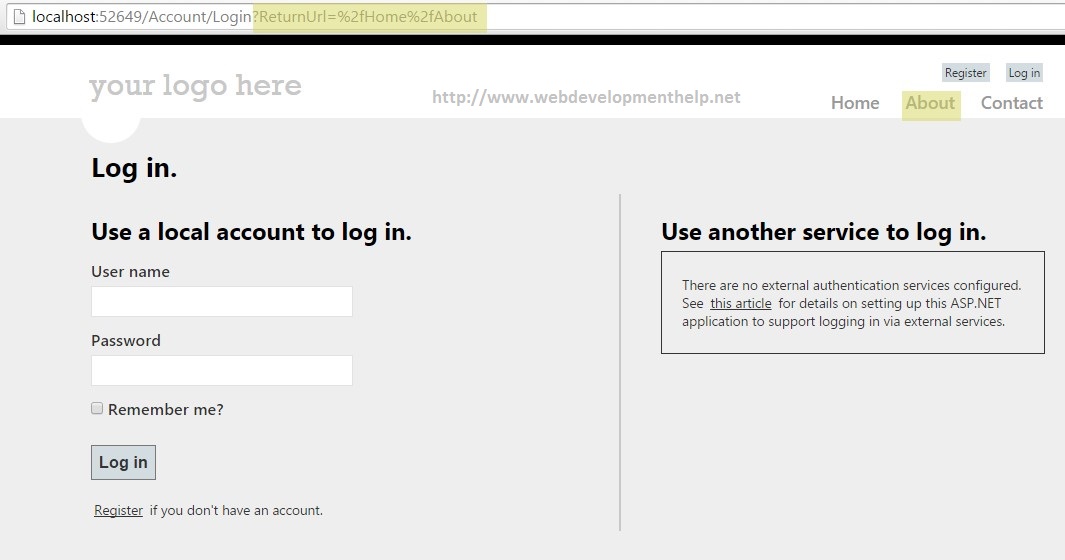
* We have **HomeController** and three **Action Methods** in it. If we launch of application now, one can access all the three methods without any restriction.





* For instance, let’s consider that **Anonymous** users should be prevented from accessing About method. If he tries to access, then MVC should navigate him to Login page. As discussed before, we can just decorate the About method with [Authorize] attribute. **Note**: While navigating user to Login page for authentication, note that the Re-Direct Url is been handled automatically for us.



* Wala! That works with much ease. So what if we need to protect all the three methods from anonymous user. Yes, preceding it before necessary Controller does that with charm.

This means that Authorize attribute is inheritable, we can add it to a base controller class and thereby it ensure that any methods of any derived controllers are subject to authentication.

We can make certain Action methods alone to anonymously access by preceding it with the attribute [AllowAnonymous].

Hide   Copy Code

[AllowAnonymous] public ActionResult Contact()

{

ViewBag.Message = "Your contact page.";

return View();

}

So here in the above snippet, the Contact method is decorated with AllowAnonymous attribute so that it can be accessed without any Login authentication.

# Handling Authorization

ASP.NET MVC newbies often get confused with the Authorize attribute’s name because it triggers the Authentication process but the name proclaim as Authorize. It’s actually not a mislead, let us dive into Authorize parameters to clarify this.

Additionally, there are two parameters that supports Authorize attribute in restricting the execution of the action method only to certain user names and/or users with a given role.

Hide   Copy Code

[Authorize(Roles="admin",

Users="Imghani, Ren")] public ActionResult About()

{

ViewBag.Message = "Web Development Help!";

return View();

}

So while logging, it checks whether the user is Imghani or Ren and holds the Admin role or not. If not, then it redirects the user to the Login Url.

# HTTP 401 or 403 (Custom Attribute)

Authorize attribute doesn’t provide a clear cut HTTP status in return if the process gets failed. Here the reason could be either of two ways, mainly, i.e.:

* Authentication Failure
* Authorization Failure

For both the instances, Authorize just returns the HTTP code as 401 which is generic but it is a tedious task for a developer to debug the exact reason behind that. To overcome this, MVC provides us the facility to override the Authorize attribute.

Hide   Copy Code

public class Error401or403 : AuthorizeAttribute

{

public Error401or403()

{ }

Public override void OnAuthorization(AuthorizationContext filterContext)

{

base.OnAuthorization(filterContext);

CheckIfUserIsAuthenticated(filterContext);

}

}

Hence, by overriding the method OnAuthorization and if we handle some extra logic like getting back the response from Authorize, we gain complete control.

If the response is 401, instead of navigating the user again to the Login page (it works as default), we can create a custom beautiful UnAuthorized error page and can navigate the user accordingly.

Hide   Copy Code

[Error401or403(Roles=”admin”, Users=”Imghani”)] public ActionResult Index()

{

….

}

The custom attribute doesn’t change the basic functionality. Rather it helps us to gain some control over the process. It returns a ViewResult object, so that we can easily navigate it to our custom error page.

**Got something to say!**

Keep learning always, hope you have enjoyed reading it. We do have lot of awesome articles on ASP.NET MVC as well as free Online Tests and MCSD Online Practice Exams, please check out this website [here](http://www.webdevelopmenthelp.net/free-online-test).  
Thanks, happy coding!

## What is a ViewEngine in ASP.NET MVC?

“View Engine in ASP.NET MVC is used to translate our views to HTML and then render to browser.” There are few View Engines available for ASP.NET MVC but commonly used View Engines are Razor, Web Forms/ASPX, NHaml and Spark etc. Most of the developers are familiar with Web Forms View Engine (ASPX) and Razor View Engine.

* Web Form View Engine was with ASP.NET MVC since beginning.
* Razor View Engine was introduced later in MVC3.
* NHaml is an open source view engine since 2007.
* Spark is also an open source since 2008.

## Is it possible to remove the unused ViewEngine in ASP.NET MVC project?

Yes, it’s possible. As we know that by default two view engines are installed.

* WebForm ViewEngine
* Razor ViewEngine

If we are 100% sure that we are only going to use say ‘Razor ViewEngine’, and WebForm ViewEngine will remain idle/unused all the time, then we can remove the unused View engine by following the below simple steps:

1. In our Gloabal.asax file, clear all the ViewEngines on Application\_Start method.
2. Add the specific one i.e Razor ViewEngine to our ViewEngine collection.

ViewEngines.Engines.Clear();

ViewEngines.Engines.Add(new RazorViewEngine());