ASP.NET MVC

MVC 5

MVC Core

Eng. Basma Hussien

ModelState

ModelState

- ModelState is a property of controller that is used for validating form in server side, using ***ModelState.IsValid***.

- ModelState is a collection of "name and value" pairs of inputs that was submitted to the server during post. It also contains a collection of error messages for each value submitted.

- Anytime you have binding happening, ModelState will contain information about what happened during that model binding.

- The default model binder will add some errors for basic type conversion issues(for example, passing a non - number for something which is an "int")

```
41          [HttpPost]
            0 references
42          public ActionResult Create(Employee emp)
43          {
44
45              if (string.IsNullOrEmpty(emp.Name))
46              {
47                  ModelState.AddModelError("Name", "You must enter a name!");
48              }
49              if (emp.Age < 18)
50              {
51                  ModelState.AddModelError("Age", "Age must be +18");
52              }
53
54              if (ModelState.IsValid)
55              {
56                  employees.Add(emp);
57              //    return RedirectToAction("Index");
58              }
59
60              return View();
61          }
```

Custom DataAnnotation

Custom DataAnnotation Usage

```
3 references
public class User
{
    [Key]
    0 references
    public int ID { get; set; }

    [MinAge(21)]
    int Age;

    [Required]
    1 reference
    public string UserName { get; set; }
```

```
9     public class MinAge : ValidationAttribute
10    {
11        int Value;
          1 reference
12        public MinAge(int num)
13        {
14            Value = num;
15        }
          0 references
16        public override bool IsValid(object obj)
17        {
18            if (obj == null)
19            {
20                return false;
21            }
22            else
23            {
24                if (obj is int)
25                {
26                    int suppliedValue = (int)obj;
27                    if (suppliedValue > Value)
28                    {
29                        return true;
30                    }
31                    else
32                    {
33                        ErrorMessage = "Minimum value for age should be + " + Value; //user validation error
34                        return false;
35                    }
36                }
37                else
38                {
39                    return false;
40                }
41            }
42        }
43    }
44}
```

MVC Filters

MVC Filters

- In ASP.NET MVC, a user request is routed to the appropriate controller and action method. However, there may be circumstances where you want to execute some logic before or after an action method executes. ASP.NET MVC provides filters for this purpose.

- ASP.NET MVC Filter is a custom class where you can write custom logic to execute before or after an action method executes. Filters can be applied to an action method or controller

MVC Filters (Cont.)

- Filters can be applied to an action method or controller
- You could apply filters in a declarative or programmatic way:
  - Declarative means by applying a filter attribute to an action method or controller class
  - Programmatic means by implementing a corresponding interface or abstract class

MVC Filters Types

- The ASP.NET MVC framework supports four different types of filters:
  - **Authorization Filters** – Implements the IAuthorizationFilter attribute.
  - **Action Filters** – Implements the IActionFilter attribute.
  - **Result Filters** – Implements the IResultFilter attribute.
  - **Exception Filters** – Implements the IExceptionFilter attribute.
- Filters are executed in the order listed above. For example, authorization filters are always executed before action filters and exception filters are always executed after every other type of filter.
- Authorization filters are used to implement authentication and authorization for controller actions.

MVC Filters Types

- MVC provides different types of filters. The following table list filter types, built-in filters, and interface that must be implemented to create custom filters.

Filter Type	Description	Built-in Filter	Interface
Authorization filters	Performs authentication and authorizes before executing an action method.	[Authorize]	IAuthorizationFilter
Action filters	Performs some operation before and after an action method executes.	[ActionName]	IActionFilter
Result filters	Performs some operation before or after the execution of the view.	[OutputCache]	IResultFilter
Exception filters	Performs some operation if there is an unhandled exception thrown during the execution of the ASP.NET MVC pipeline.	[HandleError]	IExceptionFilter

Action Filters

- An action filter is an attribute that you can apply to a controller action or an entire controller that modifies the way in which the action is executed. The ASP.NET MVC framework includes several action filters:

  - **OutputCache** – Caches the output of a controller action for a specified amount of time.
  - **HandleError** – Handles errors raised when a controller action is executed.
  - **Authorize** – Enables you to restrict access to a particular user or role.

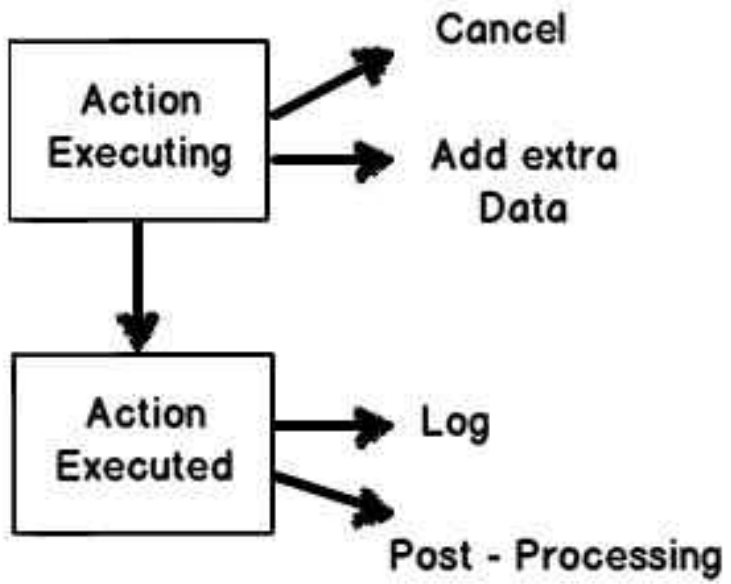
Custom Filters

- To create your own custom filter, ASP.NET MVC framework provides a base class which is known as **ActionFilterAttribute**. This class implements both ***IActionFilter*** and ***IResultFilter*** interfaces and both are derived from the Filter class.

Custom Action Filters

Now, you are going to create a Custom Action Filter which implements the pre-processing and post-processing logic. It will inherit from **ActionFilterAttribute** class and also implement **IActionFilter** interface.

- **ActionFilterAttribute** contains four important methods as in the following.
- **OnActionExecuting:** It is called just before the action method is going to call.
- **OnActionExecuted:** It is called just after the action method is called.
- **OnResultExecuting:** It is called just before the result is executed; it means before rendering the view.
- **OnResultExecuted:** It is called just after the result is executed, it means after rendering the view.



```
public class myLogFilter:ActionFilterAttribute
{
    0 references
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        Debug.WriteLine("OnActionExecuted Log: " +
            "Action: " + filterContext.RouteData.Values["action"] +
            ", Controller: " + filterContext.RouteData.Values["controller"]);
    }

    0 references
    public override void OnResultExecuted(ResultExecutedContext filterContext)
    {
        Debug.WriteLine("OnResultExecuted Log: " +
      "HttpMethod: " + filterContext.HttpContext.Request.HttpMethod +
      " Action: " + filterContext.RouteData.Values["action"] +
      ", Controller: " + filterContext.Controller);
    }
}
```

Action Selectors

- Action selector is the attribute that can be applied to the action methods. It helps the routing engine to select the correct action method to handle a particular request. MVC 5 includes the following action selector attributes:
  - ActionName
  - NonAction
  - ActionVerbs

Action Selectors (Cont.)

```
[ActionName("Find")]
    public ActionResult GetById(int id)
    {
        return View();
    }
```

```
[NonAction]
    public Student GetStudent(int id)
    {
        return studentList.Where(s => s.StudentId == id).FirstOrDefault();
    }
```

Attribute Routing

Attribute Routing

- To enable attribute routing:

  routes.MapMvcAttributeRoutes();

- Apply attribute routing on any action:

  Route[("customers/{customerId}/orders")]

RoutePrefix

- RoutePrefix on a controller:

  [RoutePrefix(“Company/Finance")]

- Then on each action in controller you should specify a specific route:

  [Route("customers/{id}")]

  [Route("")]

Override RoutePrefix

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

```
[RoutePrefix(“mvc/books")]
public class BooksController : Controller
{
    // GET /mvc/authors/1/books
    [Route("~/mvc/authors/{authorId:int}/books")]
    public IEnumerable<Book> GetByAuthor(int authorId) { ... }
}
```

Route Constraints

- To apply Route Constraints:

```
[Route("users/{id:int:min(10)}")]
public User GetUserById(int id) { ... }
```

Constraint	Description	Example
Alpha	Matches uppercase or lowercase Latin alphabet characters (a-z, A-Z)	{x:alpha}
Bool	Matches a Boolean value.	{x:bool}
Datetime	Matches a **DateTime** value.	{x:datetime
Decimal	Matches a decimal value.	{x:decimal}
Double	Matches a 64-bit floating-point value.	{x:double}
Float	Matches a 32-bit floating-point value.	{x:float}
Guid	Matches a GUID value.	{x:guid}
Int	Matches a 32-bit integer value.	{x:int}

Length	Matches a string with the specified length or within a specified range of lengths.	{x:length(6)} {x:length(1,20)}
Long	Matches a 64-bit integer value.	{x:long}
Max	Matches an integer with a maximum value.	{x:max(10)}
maxlength	Matches a string with a maximum length.	{x:maxlength(10)}
Min	Matches an integer with a minimum value.	{x:min(10)}
minlength	Matches a string with a minimum length.	{x:minlength(10)}
Range	Matches an integer within a range of values.	{x:range(10,50)}
Regex	Matches a regular expression.	{x:regex(^\d{3}-\d{3}-\d{4}$)}

Hosting MVC on IIS

Deploying ASP.net MVC Application

Deploying ASP.net MVC Application:

- Publish to IIS web server
- Publish to Azure App Service
- Publish to a Filesystem Folder

Deployment prerequisites:

- SQL server installed
- IIS installed

Membership & Identity Authentication

Demo