Build an ASP.NET Core Service, and App with Core 2.2 Two-Day Hand-On Lab

Lab 15

This lab is the fifth in a series that creates the ASP.NET Core web application. This optional lab walks you through creating custom validation attributes and the related client-side scripts. Prior to starting this lab, you must have completed Lab 14.

Part 1: Create the Server-Side validation attributes

- Add the following global using statements to the GlobalUsings.cs file:

```
global using System.ComponentModel.DataAnnotations;
global using System.Reflection;
global using System.ComponentModel;
global using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;
global using AutoLot.Mvc.Validation;
```

Step 1: Create the MustBeGreaterThanZeroAttribute attribute

- Create a new folder in the `AutoLot. Mvc` project named `Validation`. Add a new class named `MustBeGreaterThanZeroAttribute.cs`. Make the class public, inherit from `ValidationAttribute`, and implement `IClientModelValidator`:

```
namespace AutoLot.Mvc.Validation;
public class MustBeGreaterThanZeroAttribute : ValidationAttribute, IClientModelValidator
{
  public void AddValidation(ClientModelValidationContext context)
  {
  }
}
```

- Add two constructors. One that takes a custom error message and another that uses a default error message:

```
public MustBeGreaterThanZeroAttribute() : this("{0} must be greater than 0") { }
public MustBeGreaterThanZeroAttribute(string errorMessage) : base(errorMessage) { }
```

- Override the `FormatErrorMessage` method to properly format the `ErrorMessageString` (which is a property on the base `ValidationAttribute` class)

```
public override string FormatErrorMessage(string name)
{
  return string.Format(ErrorMessageString, name);
}
```

- Override the `IsValid` method to test if the value is greater than zero. This is used for server-side processing:

```
protected override ValidationResult IsValid(object value, ValidationContext validationContext)
{
  if (!int.TryParse(value.ToString(), out int result))
  {
    return new ValidationResult(FormatErrorMessage(validationContext.DisplayName));
  }
  if (result > 0)
  {
    return ValidationResult.Success;
  };
  return new ValidationResult(FormatErrorMessage(validationContext.DisplayName));
}
```

- Implement the `AddValidation` method. This method is used when generating the client-side implementation of the property.

```
public void AddValidation(ClientModelValidationContext context)
{
  string propertyDisplayName =
    context.ModelMetadata.DisplayName ?? context.ModelMetadata.PropertyName;
  string errorMessage = FormatErrorMessage(propertyDisplayName);
  context.Attributes.Add("data-val-greaterthanzero", errorMessage);
}
```

Step 2: Create the MustNotBeGreaterThanAttribute attribute

- Add a new class named `MustNotBeGreaterThanAttribute.cs`.
- Make the class public, inherit from `ValidationAttribute`, and implement `IClientModelValidator`. Also, add the `AttributeUsage` attribute to the class so it targets properties and can be used more than once in a class:

```
namespace AutoLot.Mvc.Validation;
[AttributeUsage(AttributeTargets.Property, AllowMultiple = true)]
public class MustNotBeGreaterThanAttribute : ValidationAttribute, IClientModelValidator
{
  public void AddValidation(ClientModelValidationContext context)
  {
  }
}
```

- Add two constructors. Since this attribute compares the value of this property to another property of the class instance, the constructors need to take in the other property name and an optional prefix. Just like the previous example, one takes a custom error message and the other uses a default error message:

```
readonly string _otherPropertyName;
string _otherPropertyDisplayName;
readonly string _prefix;
public MustNotBeGreaterThanAttribute(string otherPropertyName, string prefix = "")
  : this(otherPropertyName, "{0} must not be greater than {1}", prefix) { }
public MustNotBeGreaterThanAttribute(string otherPropertyName, string errorMessage, string prefix)
  : base(errorMessage)
{
  _otherPropertyName = otherPropertyName;
  _otherPropertyDisplayName = otherPropertyName;
  _prefix = prefix;
}
```

- Override the `FormatErrorMessage` method to properly format the `ErrorMessageString`:

```
public override string FormatErrorMessage(string name)
{
  return string.Format(ErrorMessageString, name, _otherPropertyDisplayName);
}
```

- Add the `SetOtherPropertyName` method, which will get the Display property of the other property.

```
internal void SetOtherPropertyName(PropertyInfo otherPropertyInfo)
{
  var displayAttribute =
    otherPropertyInfo.GetCustomAttributes<DisplayAttribute>().FirstOrDefault();
  if (displayAttribute != null)
  {
    _otherPropertyDisplayName = displayAttribute.Name;
    return;
  }
  var displayNameAttribute =
    otherPropertyInfo.GetCustomAttributes<DisplayNameAttribute>().FirstOrDefault();
  if (displayNameAttribute != null)
  {
    _otherPropertyDisplayName = displayNameAttribute.DisplayName;
    return;
  }
  _otherPropertyDisplayName = _otherPropertyName;
}
```

- Override the `IsValid` method to test if the value is less than or equal to the other property. Once again, this is used for server-side processing:

```
protected override ValidationResult IsValid(object value, ValidationContext validationContext)
{
  var otherPropertyInfo = validationContext.ObjectType.GetProperty(_otherPropertyName);
  SetOtherPropertyName(otherPropertyInfo);
  if (!int.TryParse(value.ToString(), out int toValidate))
  {
    return new ValidationResult($"{validationContext.DisplayName} must be numeric.");
  }
  var otherValue = (int)otherPropertyInfo.GetValue(validationContext.ObjectInstance, null);
  return toValidate > otherValue
    ? new ValidationResult(FormatErrorMessage(validationContext.DisplayName))
    : ValidationResult.Success;
}
```

- Implement the `AddValidation` method. This method uses a helper method to get the `Display` attribute (if it exists) or the straight property name of the other property. This method is used when generating the client-side implementation of the property.

```
public void AddValidation(ClientModelValidationContext context)
{
  string propertyDisplayName = context.ModelMetadata.GetDisplayName();
  var propertyInfo = context.ModelMetadata.ContainerType.GetProperty(_otherPropertyName);
  SetOtherPropertyName(propertyInfo);
  string errorMessage = FormatErrorMessage(propertyDisplayName);
  context.Attributes.Add("data-val-notgreaterthan", errorMessage);
  context.Attributes.Add("data-val-notgreaterthan-otherpropertyname", _otherPropertyName);
  context.Attributes.Add("data-val-notgreaterthan-prefix", _prefix);
}
```

Part 2: Create the Client-Side validation scripts

Step 1: Create the Validators

- Add a new folder named validations under the `wwwroot/js` folder. Add a new JavaScript file named `validators.js` in the new folder.
- Add the validator method for the `GreaterThanZero` validation. This name must match the name from the `AddValidation` method in the C# class:

```
$.validator.addMethod("greaterthanzero", function (value, element, params) {
    return value > 0;
});
```

- Add the unobtrusive adapter for the `GreaterThanZero` validation next. The rules property is simply set to true to enable validation, and the message is message from the `AddValidation` method:

```
$.validator.unobtrusive.adapters.add("greaterthanzero", function (options) {
    options.rules["greaterthanzero"] = true;
    options.messages["greaterthanzero"] = options.message;
});
```

- Add the validator method for the `NotGreaterThan` validation. As with the previous example, the name must match the name from the `AddValidation` method:

```
$.validator.addMethod("notgreaterthan", function (value, element, params) {
    return +value <= +$(params).val();
});
```

- Add the adapter for the `NotGreaterThan` validation:

```
$.validator.unobtrusive.adapters.add("notgreaterthan", ["otherpropertyname","prefix"], function
(options) {
    options.rules["notgreaterthan"] = "#" + options.params.prefix +
options.params.otherpropertyname;
    options.messages["notgreaterthan"] = options.message;
});
```

Step 2: Create the formatter code

This code prettifies errors in the UI.

- Create a new JavaScript file named `errorFormatting.js` in the validations folder.
- Update the code to match the following:

```
$.validator.setDefaults({
    highlight: function (element, errorClass, validClass) {
        if (element.type === "radio") {
            this.findByName(element.name).addClass(errorClass).removeClass(validClass);
        } else {
            $(element).addClass(errorClass).removeClass(validClass);
            $(element).closest('.form-group').addClass('has-error'); //.removeClass('has-
success');
        }
    },
    unhighlight: function (element, errorClass, validClass) {
        if (element.type === "radio") {
            this.findByName(element.name).removeClass(errorClass).addClass(validClass);
        } else {
            $(element).removeClass(errorClass).addClass(validClass);
            $(element).closest('.form-group').removeClass('has-error');  //.addClass('has-
success');
        }
    }
});
```

Part 3: Minify the JavaScript Files

To minimize specific files or to create bundles, add configuration options into the `AddWebOptimizer` method.

- Use `AddJavaScriptBundle` to bundle files. First argument is the bundle name, next are the files to be bundled:

```
services.AddWebOptimizer(options =>
{
  options.MinifyCssFiles(); //Minifies all CSS files
  //options.MinifyJsFiles(); //Minifies all JS files
  options.MinifyJsFiles("js/site.js");
  options.MinifyJsFiles("js/**/*.js");
});
```

Part 4: Update the _ValidationScriptsPartial.cshtml

- Open `Views\Shared\_ValidationScriptsPartial.cshtml`.
- Add the following at the end, outside of the environment tag helpers:

```
<script src="~/js/validations/validators.js" asp-append-version="true"></script>
<script src="~/js/validations/errorFormatting.js" asp-append-version="true"></script>
```

Part 5: Use the attribute

Step 1: Create the AddToCartViewModel

- Create a new class named AddToCartViewModel in the Models directory, and update it to the following:

```
namespace AutoLot.Mvc.Models;

public class AddToCartViewModel
{
  public int Id { get; set; }
  [Display(Name="Stock Quantity")]
  public int StockQuantity { get; set; }
  public int ItemId { get; set; }
  [Required, MustBeGreaterThanZero, MustNotBeGreaterThan(nameof(StockQuantity))]
  public int Quantity { get; set; }
}
```

Step 2: Add the View

- Create a new view named Validation.cshtml in the Views\Home folder and update it to match the following:

```
@model AutoLot.Mvc.Models.AddToCartViewModel

@{
    ViewData["Title"] = "Validation";
}

<h1>Validation</h1>

<h4>AddToCartViewModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Validation">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div>
                <label asp-for="Id" class="col-form-label"></label>
                <input asp-for="Id" class="form-control" />
                <span asp-validation-for="Id" class="text-danger"></span>
            </div>
            <div>
                <label asp-for="StockQuantity" class="col-form-label"></label>
                <input asp-for="StockQuantity" class="form-control" />
                <span asp-validation-for="StockQuantity" class="text-danger"></span>
            </div>
            <div>
                <label asp-for="ItemId" class="col-form-label"></label>
                <input asp-for="ItemId" class="form-control" />
                <span asp-validation-for="ItemId" class="text-danger"></span>
            </div>
            <div>
                <label asp-for="Quantity" class="col-form-label"></label>
                <input asp-for="Quantity" class="form-control" />
                <span asp-validation-for="Quantity" class="text-danger"></span>
            </div>
            <div style="margin-top:5px">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Step 3: Add the Controller Method

- Add the method to the HomeController:

```
[HttpGet]
public IActionResult Validation()
{
  var vm = new AddToCartViewModel
  {
    Id = 1,
    ItemId = 1,
    StockQuantity = 2,
    Quantity = 0
  };
  return View(vm);
};
```

Step 4: Update the _Menu Partial

- Add the following menu item to the _Menu.cshtml partial:

```
<li class="nav-item">
  <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Validation"
title="Validation Example">Validation<i class="fas fa-check"></i></a>
</li>
```

Summary

The lab created the custom validation attribute, client-side validation scripts and formatting, minified the scripts, and updated the validation partial view. Then used the attributes on a view model. This completes this hands on lab.