The Complete Guide To Validation In ASP.NET MVC 3 - Part 2

This is second part our comprehensive guide to ASP.NET MVC 3 validation. In this article, we will write several custom validators by subclassing ValidationAttribute. We will include functionality to output HTML5 data-* attributes for use with client side validation. We will then add the client-side validation and learn about jQuery.validate and the Microsoft adapters that are used to convert the HTML5 data-* attributes into a format that jQuery.validate understands.

If you have not read part one yet, then please do take a look as it offers a broad introduction into validation in ASP.NET MVC 3 and includes several topics that will be useful for this article.

The Complete Guide To Validation In ASP.NET MVC 3 - Part 1

Writing our own custom validator

There are four distinct parts to creating a fully functional custom validator that works on both the client and the server.

1. Subclass ValidationAttribute and add our server side validation logic.
2. Implement IClientValidatable on our attribute to allow HTML5 data-* attributes to be passed to the client.
3. Write a custom javascript function that performs validation on the client.
4. Create an adapter to transform the HTML5 attributes into a format that our custom function can understand.

While this sounds like a lot of work, once you get started you will find it relatively straightforward.

Subclassing ValidationAttribute

In this example, we are going to write a **NotEqual**To validator that simply checks that the value of one property does not equal the value of another. In effect, the opposite of the new CompareAttribute. We are going to use it on our password property to ensure that the

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = true)]
public sealed class NotEqualToAttribute : ValidationAttribute
{
  private const string DefaultErrorMessage = "{0} cannot be the same as {1}.";

  public string OtherProperty { get; private set; }

  public NotEqualToAttribute(string otherProperty)
    : base(DefaultErrorMessage)
  {
    if (string.IsNullOrEmpty(otherProperty))
    {
      throw new ArgumentNullException("otherProperty");
    }

    OtherProperty = otherProperty;
  }

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

  protected override ValidationResult IsValid(object value,
                      ValidationContext validationContext)
  {
    if (value != null)
    {
      var otherProperty = validationContext.ObjectInstance.GetType()
                        .GetProperty(OtherProperty);

      var otherPropertyValue = otherProperty
                  .GetValue(validationContext.ObjectInstance, null);

      if (value.Equals(otherPropertyValue))
      {
        return new ValidationResult(
```

```
        }

        return ValidationResult.Success;
    }
}
```

If you have read the previous article and examined the code for the ValidatePasswordLengthAttribute, you will find the code very familiar. Let's examine this code in detail, starting from the top.

Our validator subclasses the built-in abstract class ValidationAttribute. The class is decorated with an AttributeUsage attribute that tell .NET how the attribute can be used. Here, the **NotEqual**ToAttribute attribute is only permitted on properties and only a single instance of the attribute may appear on each property.

The next thing of note is the default error message constant. If we examine this value, we can see that it is in the form of a format string with placeholders for dynamic content (the name of the two properties being compared). The default error message is passed into the constructor of the base class, giving the user the option to override this default by setting either ErrorMessage or ErrorMessageResourceName and ErrorMessageResourceType which are properties on the base class. We also override FormatErrorMessage allowing us to add the two dynamic property names into the error message string.

Unlike the ValidatePasswordLengthAttribute example that we saw in part 1, our **NotEqual**ToAttribute needs some configuration. Therefore, the name of the property that we are to check must be passed to the class constructor. We have a null check in place as this property is always required.

The next thing we need to do is actually implement the validation logic. This is done by overriding the IsValid method. It is generally a good idea to only validate if the property has a value, so we add a null check and return *ValidationResult.Success* if the value is null. Otherwise we access the object with validationContext.ObjectInstance and use reflection to obtain the value of the other property. Finally, we compare the two property values and return a validation failure if they match.

In some situations, you might be tempted to use the second constructor overload of ValidationResult that takes in an *IEnumerable* of member names. For example, you may decide that you want to display the error message on both fields being compared, so you

```
return new ValidationResult(FormatErrorMessage(validationContext.DisplayName),
  new[] { validationContext.MemberName, OtherProperty });
```

If you run your code, you will find absolutely no difference. This is because although this overload is present and presumably used elsewhere in the .NET framework, the MVC framework completely ignores ValidationResult.MemberNames.

It is important to note that the ValidationAttribute class is not new, but the .NET 4.0 version has important differences from its' .NET 3.5 counterpart making it a much more powerful tool. The important difference that makes the .NET 4.0 version far more useful lies in the signature of the IsValid method. The .NET 3.5 method looked like this:

```
public abstract bool IsValid(object value);
```

Using this version, if you needed access to multiple properties within a validator, you had to declare the AttributeUsage as Class rather than Property and thus put the attribute on the model rather than an individual property of the model. When you did this, the value argument of the IsValid method contained the whole object instance so you could use this object to access the properties using reflection:

```
public override bool IsValid(object value)
{
  var properties = TypeDescriptor.GetProperties(value);
  var originalValue = properties.Find(OriginalProperty, true /* ignoreCase */).GetValue(value);
  var confirmValue = properties.Find(ConfirmProperty, true /* ignoreCase */).GetValue(value);
  return Object.Equals(originalValue, confirmValue);
}
```

Whilst this worked, it sets your validation error against the model rather than a property. This means that at the UI, none of the model properties would display the validation error and in fact, the error would only be displayed at all if you had added a validation summary in your view.

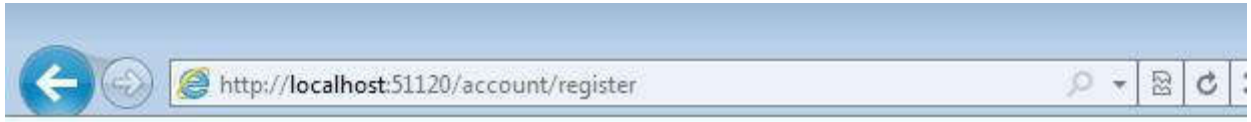
```
public virtual bool IsValid(object value);

protected virtual ValidationResult IsValid(object value, ValidationContext validationContext);
```

The .NET 4.0 version still has the single value method signature (for backward compatability) but it is no longer marked as abstract meaning that we can choose to override the second overload which includes a validationContext argument. This (among other things) gives us access to the whole model even when the validator is at the property level. The massive advantage of using a property level validator is that the error is set against the property itself instead of the class removing the requirement to use an Html.ValidationSummary. Since the validation error is set against the property correctly, your normal Html.ValidationFor helpers will pick up and display the error against the invalid form field.

That is our initial server side validation code complete. Let's add the new attribute to the password property of the RegisterModel and run the application.

```
[Required]
[DataType(DataType.Password)]
[Display(Name = "Password")]
[NotEqualTo("UserName")]
public string Password { get; set; }
```

If you enter the same value for username and password, then on post back you will find the server side **NotEqual**ToAttribute validation fires and returns a validation failure to the client. Now try entering different values and you will find that after a short delay, the username validates successfully and the error message disappear. We have our server side validation working, but for a better user experience, it makes sense for our custom validator to behave in the same way as the built-in validators and offer client side validation. This is exactly what we are going to do next.



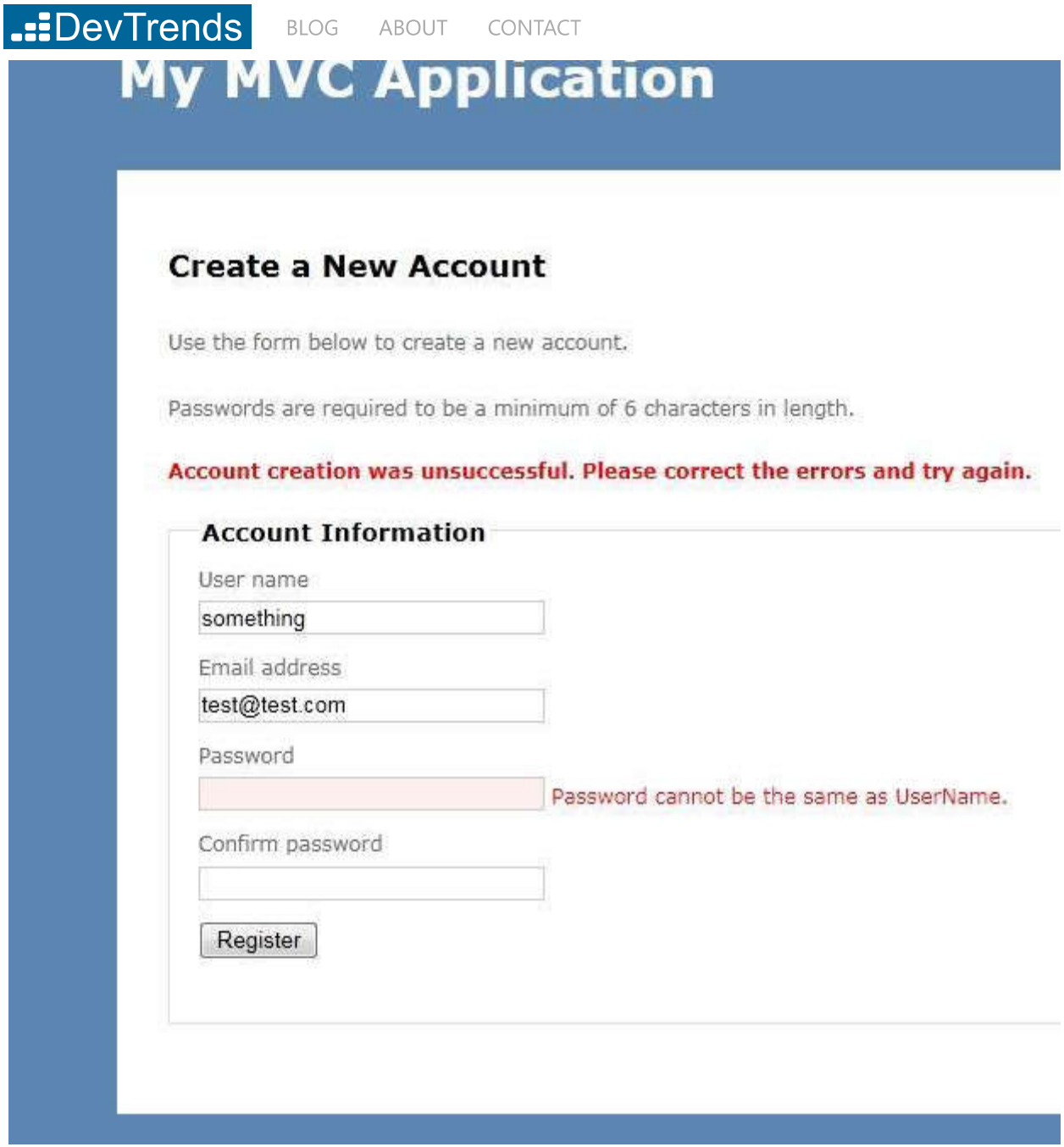


Figure 1: NotEqual Validator Server Side Validation

ASP.NET MVC 2 had a mechanism for adding client side validation but it was not very pretty. Thankfully in MVC 3, things have improved and the process is now fairly trivial and thankfully does not involve changing the Global.asax as in the previous version.

The first step is for your custom validation attribute to implement IClientValidatable. This is a simple, one method interface:

```
public IEnumerable<ModelClientValidationRule> GetClientValidationRules(ModelMetadata metadata, ControllerContext
context)
{
  var clientValidationRule = new ModelClientValidationRule()
  {
    ErrorMessage = FormatErrorMessage(metadata.GetDisplayName()),
    ValidationType = "notequalto"
  };

  clientValidationRule.ValidationParameters.Add("otherproperty", OtherProperty);

  return new[] { clientValidationRule };
}
```

The GetClientValidationRules method that you need to implement has one simple function. Using the metadata parameter, you are required to construct one or more ModelClientValidationRules which are returned from the method and used by the framework to output the client-side HTML5 data-* attributes that are necessary for client-side validation. Therefore, ModelClientValidationRule must contain all data necessary for validation to be performed on the client.

The ModelClientValidationRule class has three properties, two of which must always be set: ErrorMessage and ValidationType. For ErrorMessage, we simply call the same method as our server side validation making use of the ModelMetadata argument to obtain the DisplayName. ValidationType is a string unique to the type of validator and is used as a key to get the right client side validation code. There is also a ValidationParameters collection which will contain zero or more entries depending on your validator type. In this example, the client only requires one additional piece of data in order to carry out validation: the name of the other property.

returns the value of a DisplayName or Display(Name=...) attribute if present on the property. If neither of these attributes is present, nothing will be returned. In our example, if we remove the DisplayAttribute from the password property, our client error message would be displayed as ' cannot be the same as UserName.' which is obviously not what we want. Therefore, always use the GetDisplayName() method instead. This has the same functionality as the property unless the attributes are not present in which case, it will default to the name of the property.

If you run the application now and view source, you will see that the password input html now contains your **notequal**to data attributes:

```
<div class="editor-field">
  <input data-val="true" data-val-notequalto="Password cannot be the same as UserName."
    data-val-notequalto-otherproperty="UserName"
    data-val-regex="Weak password detected."
    data-val-regex-pattern="^(?!password$)(?!12345$).*"
    data-val-required="The Password field is required."
    id="Password" name="Password" type="password" />
  <span class="hint">Enter your password here</span>
  <span class="field-validation-valid" data-valmsg-for="Password"
    data-valmsg-replace="true"></span>
</div>
```

If you happen to point reflector at the MVC 3 dll, you might find that there are a number of classes in the form ModelClientValidation{Attribute Name}Rule such as ModelClientValidationRequiredRule and ModelClientValidationRegexRule. These are subclassed from the ModelClientValidationRule that we used above and just provide a constructor which sets the base values. I do not really see the benefit of this approach in our situation where reuse is not an option, but if you do want to follow this convention, then you would change the code to:

```
public IEnumerable<ModelClientValidationRule> GetClientValidationRules(ModelMetadata metadata, ControllerContext
context)
{
  return new[] { new ModelClientValidationNotEqualToRule(FormatErrorMessage(metadata.DisplayName), OtherProperty) };
}
```

```
  public ModelClientValidationNotEqualToRule(string errorMessage, string otherProperty)
  {
    ErrorMessage = errorMessage;
    ValidationType = "notequalto";
    ValidationParameters.Add("otherproperty", otherProperty);
  }
}
```

That's the c# code complete. The next stage is the javascript. The good news is that since the move to jQuery and jQuery.validate, the javascript code is easier than ever.

Creating a custom jQuery validate function

There are two parts to client side validation. The first part is the creation of the validation function itself and the second is writing an adapter that converts the HTML5 data-* attributes into a form that jQuery.validate can handle.

All of this code is best placed in a separate javascript file, so in our example, create a new *customValidation.js* file and put in the following:

```
(function ($) {
  $.validator.addMethod("notequalto", function (value, element, params) {
    if (!this.optional(element)) {
      var otherProp = $('#' + params)
      return (otherProp.val() != value);
    }
    return true;
  });
  $.validator.unobtrusive.adapters.addSingleVal("notequalto", "otherproperty");

} (jQuery));
```

late in the process. Here we are just wrapping our code in a javascript closure and passing in the jQuery object aliased as $.

Within the closure, the majority of the rest of the code (all but the last line) is the validation function. It is fairly easy to understand. First we check that the element has a value using jquery.validate's optional method. If so, we look up the other property and compare the values. The method returns true if the value is valid or false if it is invalid.

All validation functions can take in three arguments: value, element and params though you can omit params from your function if you do not require any further information other than the element being validated. If your validation function requires a single value (as in our example) then the params argument will be the value itself. On the other hand if your validation function requires multiple values, then you can access each one using the syntax param.{parametername}. For example, param.otherproperty or param.whatever. We will look at a more complicated scenario later in the article but in this case, we only require one parameter - the name of the other property.

Depending on your validation requirements, you may find that the jquery.validate library already has the code that you need for the validation itself. There are lots of validators in jquery.validate that have not been implemented or mapped to data annotations, so if these fulfil your need, then all you need to write in javascript is an adapter or even a call to a built-in adapter which can be as little as a single line. Take a look inside jquery.validate.js to find out what is available.

Using an existing jquery.validate.unobtrusive adapter

The job of the adapter is to read the HTML5 data-* attributes on your form element and convert this data into a form that can be understood by *jquery.validate* and your custom validation function. You are not required to do all the work yourself though and in many cases, you can call a built-in adapter. *jquery.validate.unobtrusive* declares three built-in adapters which can be used in the majority of situations. These are:

- *jQuery.validator.unobtrusive.adapters.addBool* - used when your validator does not need any additional data.
- *jQuery.validator.unobtrusive.adapters.addSingleVal* - used when your validator takes in one piece of additional data.
- *jQuery.validator.unobtrusive.adapters.addMinMax* - used when your validator deals with minimum and maximum values such as range or string length.

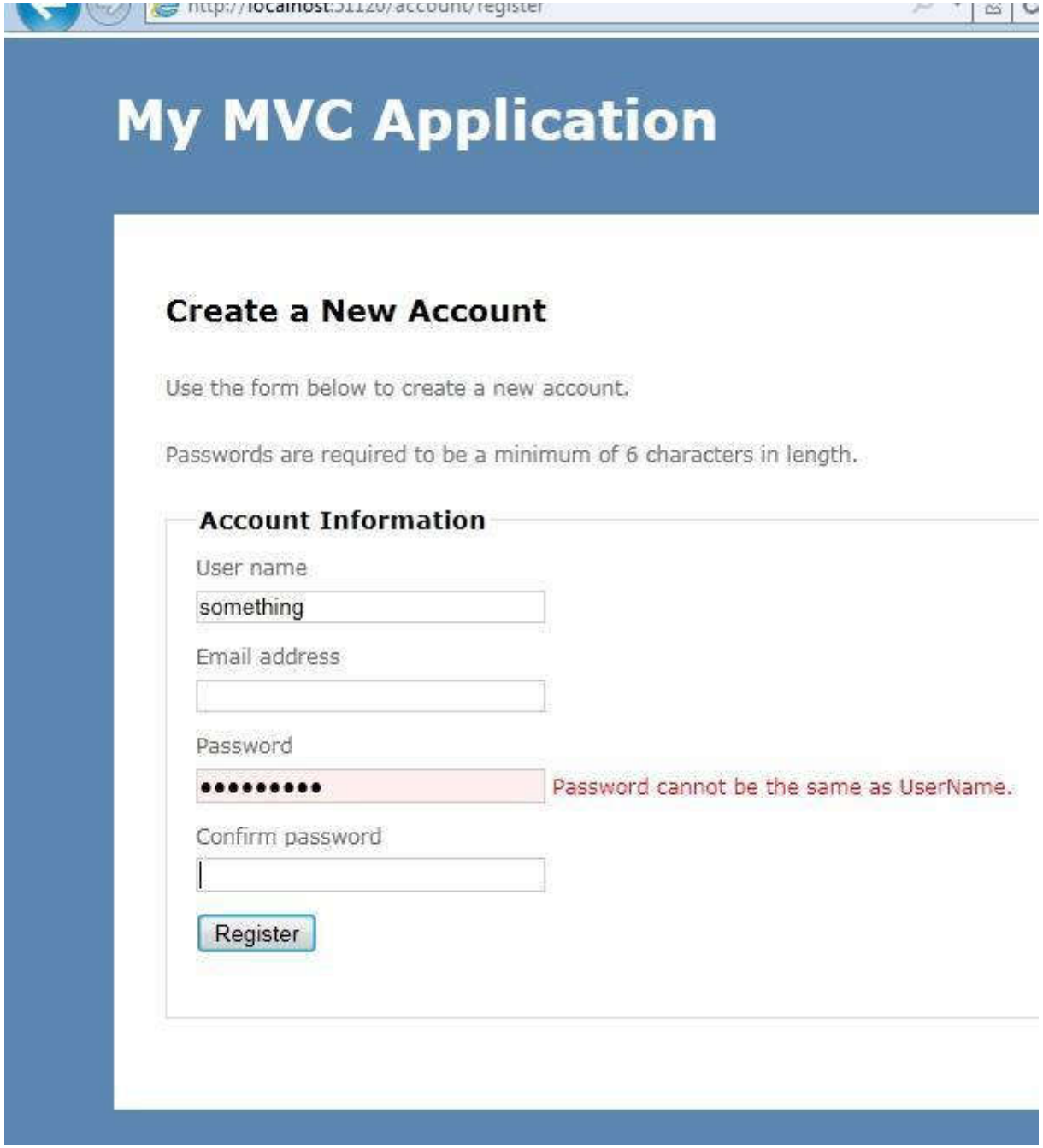
jQuery.validator.unobtrusive.adapters.add method. This is not as difficult as it sounds and we'll see an example later in the article.

Brad Wilson goes into much more detail about these adapters in his blog post Unobtrusive Client Validation in ASP.NET MVC 3

So, in our case, we make use of the addSingleVal method, passing in the name of the adapter and the name of the single value that we want to pass. Should the name of the validation function differ from the adapter, you can pass in a third parameter (ruleName):

```
jQuery.validator.unobtrusive.adapters.addSingleVal("notequalto", "otherproperty", "mynotequaltofunction");
```

At this point, our custom validator is complete. If we run the application once more, we should have our server side and client side validation working.


My MVC Application
Create a New Account
Use the form below to create a new account.
Passwords are required to be a minimum of 6 characters in length.
Account Information
User name
something
Email address
Password
Password cannot be the same as UserName.
Confirm password
Register

A more complex custom validator

Now that we have created our **NotEqual**ToAttribute, we can take a look at a slightly more complicated example. We are going to create a RequiredIfAttribute that allows a field to be mandatory only if another field contains a certain value. This is a common UI requirement where for example, if you tick a checkbox, then a text field must be filled in. A typical real world example is a dropdown list that contains multiple values including 'Other'. If the user selects this option, then some javascript displays a 'Please specify...' text input which is a required field.

Subclassing ValidationAttribute again

As before, we will start with the C# code before moving on to the javascript validator and finally joining them together with the adapter. Given that we have already been through this process already with the **NotEqual**ToAttribute, I will not explain every line of code and will instead concentrate on the important differences between this example and the previous one.

```
public enum Comparison
{
  IsEqualTo,
  IsNotEqualTo
}

[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = true)]
public sealed class RequiredIfAttribute : ValidationAttribute, IClientValidatable
{
  private const string DefaultErrorMessageFormatString = "The {0} field is required.";

  public string OtherProperty { get; private set; }
  public Comparison Comparison { get; private set; }
  public object Value { get; private set; }

  public RequiredIfAttribute(string otherProperty, Comparison comparison, object value)
  {
        if (string.IsNullOrEmpty(otherProperty))
        {
```

```
        OtherProperty = otherProperty;
        Comparison = comparison;
        Value = value;

        ErrorMessage = DefaultErrorMessageFormatString;
    }

    private bool Validate(object actualPropertyValue)
    {
        switch (Comparison)
        {
          case Comparison.IsNotEqualTo:
                return actualPropertyValue == null || !actualPropertyValue.Equals(Value);
          default:
                return actualPropertyValue != null &amp;amp;&amp;amp; actualPropertyValue.Equals(Value);
        }
    }

    protected override ValidationResult IsValid(object value,
                                                ValidationContext validationContext)
    {
        if (value == null)
        {
          var property = validationContext.ObjectInstance.GetType()
                                          .GetProperty(OtherProperty);

          var propertyValue = property.GetValue(validationContext.ObjectInstance, null);

          if (!Validate(propertyValue))
          {
                return new ValidationResult(
                  string.Format(ErrorMessageString, validationContext.DisplayName));
          }
        }
        return ValidationResult.Success;
    }

    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
```

```
  {
        return new[]
        {
          new ModelClientValidationRequiredIfRule(string.Format(ErrorMessageString,
                metadata.GetDisplayName()), OtherProperty, Comparison, Value)
        };
  }
}

public class ModelClientValidationRequiredIfRule : ModelClientValidationRule
{
  public ModelClientValidationRequiredIfRule(string errorMessage,
                                             string otherProperty,
                                             Comparison comparison,
                                             object value)
  {
        ErrorMessage = errorMessage;
        ValidationType = "requiredif";
        ValidationParameters.Add("other", otherProperty);
        ValidationParameters.Add("comp", comparison.ToString().ToLower());
        ValidationParameters.Add("value", value.ToString().ToLower());
  }
}
```

The C# code is very similar to the previous validator. The main difference is that the constructor takes in three parameters instead of one. These parameters are subsequently used in server side validation and also returned from GetClientValidationRules in a custom ModelClientValidationRule built as a container object for all the data that should be rendered as HTML5 data-* attributes. I am not going to say anything further about this code as it is fairly readable and follows exactly the same pattern as the **NotEqual**Attribute that we created earlier in the article.

Before we look at the javascript side of things, we need to change our model to use the new validator. In the existing RegisterModel class, there is really nowhere that the validator can be added, so let's make a couple of modifications. Firstly we are going to add two new fields to the model. HasPromotionalCode is a boolean and will map to a checkbox. If the user indicates that they do have a promotional code, then we want the PromotionalCode string property to be required.

```
public bool HasPromotionalCode { get; set; }

[Display(Name = "Promotional code")]
[RequiredIf("HasPromotionalCode", Comparison.IsEqualTo, true)]
public string PromotionalCode { get; set; }
```

Next we change our view to display the two new properties:

```
<div class="editor-label">
  @Html.LabelFor(m => m.PromotionalCode)
</div>
<div class="editor-field">
  @Html.TextBoxFor(m => m.PromotionalCode)
  @Html.ValidationMessageFor(m => m.PromotionalCode)
</div>
```

Whilst this works, if we want to create a really friendly UI then we might want to hide the PromotionalCode text input until the user checks the HasPromotionalCode field. This is very easy to achieve with jQuery. We'll just wrap the label and text input in a hidden div and add some javascript to show/hide the div when the checkbox value changes.

```
<div id="promotionalCodeDiv" style="display: none;">
  <div class="editor-label">
        @Html.LabelFor(m => m.PromotionalCode)
  </div>
  <div class="editor-field">
        @Html.TextBoxFor(m => m.PromotionalCode)
        @Html.ValidationMessageFor(m => m.PromotionalCode)
  </div>
</div>
<script type="text/javascript">
  $(document).ready(function () {
        $('#HasPromotionalCode').click(function () {
          $('#promotionalCodeDiv').toggle();
        });
```

If you run the application at this point, you will find all your fields present and correct. Let's move on to the javascript validation code now which needs a little more explanation.

Creating another custom jQuery validate function

```
jQuery.validator.addMethod("requiredif", function (value, element, params) {
  if ($(element).val() != '') return true

  var $other = $('#' + params.other);

  var otherVal = ($other.attr('type').toUpperCase() == "CHECKBOX") ?
                  ($other.attr("checked") ? "true" : "false") : $other.val();

  return params.comp == 'isequalto' ? (otherVal != params.value)
                                    : (otherVal == params.value);
});
```

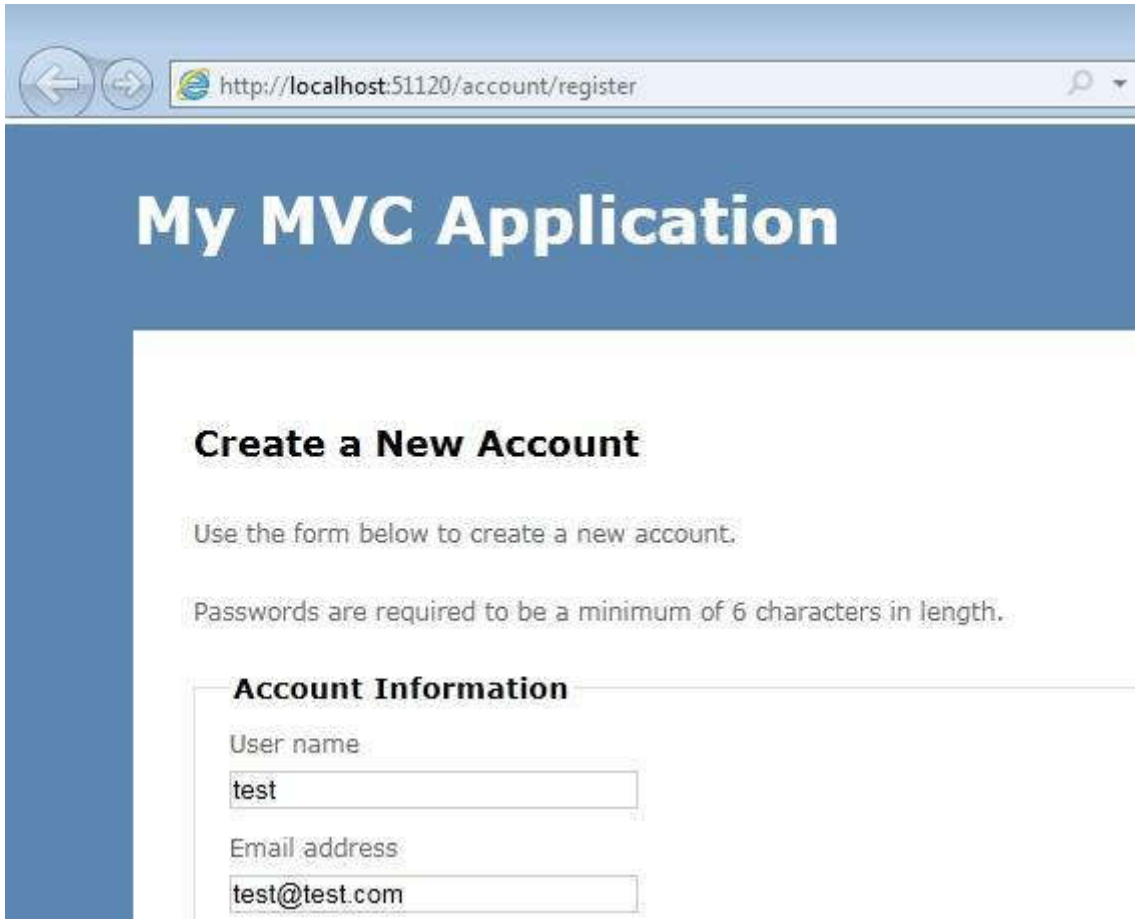
Creating a jquery.validate.unobtrusive adapter

As our client validation method requires three different values, we cannot use any of the built-in adapters this time and must write our own version. As previously mentioned, the unobtrusive validation library contains an add method that we can use to register our adapter.

```
jQuery.validator.unobtrusive.adapters.add("requiredif", ["other", "comp", "value"],
  function (options) {
        options.rules['requiredif'] = {
          other: options.params.other,
          comp: options.params.comp,
          value: options.params.value
        };
        options.messages['requiredif'] = options.message;
```

It is not obvious what is going on here until I explain to you that the job of the adapter is two-fold. Firstly it must map parameters from options.params.* to options.rules['rulename'] and secondly it must map the error message from options.message to options.messages['rulename']. There is often more to it than that, but as Brad Wilson has already done a comprehensive post on the intricacies of unobtrusive adapters, there is no point in me repeating it here so I suggest you head over there to fill in the gaps.

We are finished and now have a RequiredIf validation attribute that validates input at the client using javascript and at the server, just like the built-in validators. This provides the ultimate in user experience with immediate feedback as soon as you tab out of a field, plus the security of the server side checks which cannot be circumvented.



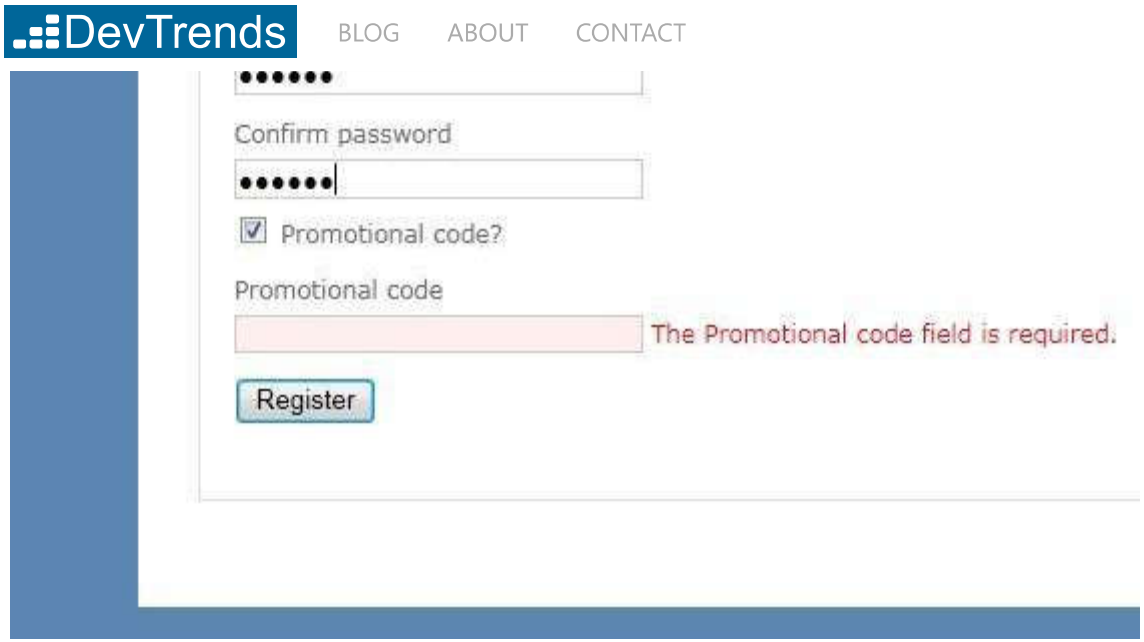


Figure 3: The Completed RequiredIf Validator

To show that adapters are not always so simple, imagine a scenario where our requirements are not as advanced and we just want the RequiredIf validator to check for the presence of *any* value in another field. In this case, we would not need a custom validation function at all and would only be required to write an adapter. JQuery.validate's built-in required function takes in a dependency expression. Typically we pass true which means that a field is always required but we can optionally pass a selection filter:

```
jQuery.validator.unobtrusive.adapters.add("requiredif", ["other"], function (options) {
        var $other = $('#' + options.params.other);
        options.rules['required'] = "#" + options.params.other + (($other.attr('type').toUpperCase() == "CHECKBOX") ?
        ":checked" : ":filled");
        options.messages['required'] = options.message;
  });
```

In this situation, the adapter needs to map the server generated data-val-requiredif-other attribute to the selection filter that the required validator can accept. We check if the other field is a checkbox. If so, we generate a selection filter in the format

information about selection filters. I'll leave it as an exercise for the reader to make this a bit more sophisticated!

Exploring alternatives with IValidatableObject

ASP.NET MVC 3 also introduces an alternative mechanism for validation involving multiple properties: IValidatableObject. This interface contains a single method:

```
IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
```

Given that this interface is implemented by the viewmodel itself, you have direct access to all model properties making your code even more straightforward than the subclassing of ValidationAttribute approach that we used above. Instead of using our custom NotEqualToAttribute, we can achieve the same validation functionality on the server by implementing IValidatableObject:

```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
  if (UserName == Password)
  {
        yield return new ValidationResult("Password cannot be the same as username",
              new[] { "Password" });
  }
}
```

This definitely results in more readable and more concise code. IValidatableObject looks especially useful for one-off validation situations where re-use is never going to happen - where creating a specific validation attribute seems like overkill.

Unfortunately, there are a couple of reasons why you might want to think twice before using IValidatableObject:

Two stage validation

your IValidatableObject validation will only be called if your data annotations validation validates successfully. This can result in a rather poor user experience:

1. user submits form
2. a single validation error is displayed (data annotations)
3. user corrects mistake and resubmits.
4. a different, totally unrelated error is displayed (IValidatableObject).
5. user corrects second mistake and resubmits
6. success

When this experience is compared with the alternative (ValidationAttribute) approach where all validation errors are returned in one go, you may find it somewhat lacking. You may be wondering if this is relevant if you have enabled client-side validation. More bad news I am afraid...

Client-side validation

Unlike the simple IClientValidatable implementation required for the ValidationAttribute method, I have not found any way of enabling client-side validation when using IClientValidatable. This was recently confirmed by a couple of guys at Microsoft. Without client-side validation, I simply cannot see IValidatableObject being used for the vast majority of validation scenarios. If you can replicate the validation logic on the client, you would subclass ValidationAttribute. Otherwise, you would use the RemoteAttribute that we discussed in part 2 of this series. Both of these approaches result in immediate feedback for the user which is simply not possible with IValidatableObject.

Code Download

You can now download all the code from this article here.

Conclusion

(using JavaScript) and at the server. We have dealt with scenarios where we have utilised existing validation functions and adapters. We have also developed totally custom validators where we were required to write everything ourselves. We also briefly looked at an alternative to custom validation attributes - IValidatableObject. We loved the clean server side code that we could use with this approach, but were sorely disappointed by the two-stage validation process and the lack of client-side integration. Have I missed anything out? Almost certainly, so there is something related to MVC 3 validation that I did not cover, please get in touch and let me know and I will address it in a future post.

Useful or Interesting?

If you liked the article, I would really appreciate it if you could share it with your Twitter followers.

SHARE ON TWITTER

Comments

DAVE WROTE ON 11 MAR 2011

Awesome articles. Do you have a code download?

RAJKUMAR WROTE ON 22 MAR 2011

Very nice post. Even regular programmer can understand this post.
Thank you.

Is it possible to scroll to a validation summary upon errors in form, or does it always scroll to the first input ?

PAUL HILES WROTE ON 07 APR 2011

@Cter - ASP.NET MVC does not scroll at all upon errors. ASP.NET WebForms on the other hand uses JavaScript via the MaintainScrollPositionOnPostBack property. The preferred option in MVC is to submit the form using AJAX, avoiding a page reload and maintaining your scroll position. If you want to scroll to the ValidationSummary, you could always hook into the ajax onSuccess event and use something like jQuery.ScrollTo plugin.

MARTY WROTE ON 07 APR 2011

I'm trying to modify your **NotEqual**ToAttribute class to allow multiple attributes (I've got a property value that must be different from 2 other properties in order to be valid).

So I set [AttributeUsage(AllowMultiple = true)] and added add two instances of the **NotEqual**To attribute to a property like so:

```
[NotEqualTo("OldPassword", ErrorMessage = "The new password cannot be the same as your old password.")]
[NotEqualTo("UserName", ErrorMessage = "The password cannot be the same as your Username.")]
[Display(Name = "New password")]
public string NewPassword { get; set; }
```

But client-side validation doesn't work, as only the second occurrence of the **NotEqual**To property was used in rendering the markup.

Any clues how to overcome this?

Thanks

PAUL HILES WROTE ON 08 APR 2011

@Marty - The mechanism for mapping the validation attributes into jQuery.validate functions (using HTML-5 attributes) does not support having multiple validation attributes of the same type on the same property. If you look at all the built-in validators, you will see that they are all marked as AllowMultiple = false. However, all is not lost. We can approach the problem slightly differently and change the NotEqualTo validator to take in a list of properties instead of just one. Hopefully, this will suit your needs. I have added a code download that includes this new change at the bottom of the article.

ANDERS WROTE ON 13 APR 2011

Really useful article, thanks a lot!

TWEEZZ WROTE ON 10 MAY 2011

Hi,

If only I would have bumped into your 2 asp mvc validation posts a week ago :)
There's one issue I'm still struggling with.. How does one validate properties of a complex type?

Manu.

PAUL HILES WROTE ON 11 MAY 2011

@TweeZz - I tend to keep my view models very simple (and flat) which facilitates easy client and server side validation using data annotations. Once the view model has been validated successfully, I typically use Automapper to map from the view model to the domain entities which is then passed on to the next layer.

EDDIE GROVES WROTE ON 01 JUN 2011

Hi Paul

Fantastic post, this is really useful for clearing up a lot of the confusion regarding ASP.NET MVC validation. I really feel that this common use case has been made much harder due to the numerous changes in each version of MVC and the lack of definitive documentation regarding best practices.

Just having a look at the number of questions on Stackoverflow reveals many devs struggling with this as well.

I also appreciate the opinionated summary of IValidatableObject.

Regarding complex type scenarios - I have been seriously burnt by trying to use out of the box validation and model binding with complex types. I think your comment above about mapping complext models to simple flat view models is extremely understated and I wish there was more guidance around this for new comers by prominent bloggers like Phil Haack and Scott Gu.

Great article Paul, very useful. Much more useful than anything I can find on SO.

However, I can get server side validation fine, but my client side validation isn't working. I copied your code down to the tee with only some variable name changes. I made the jscript and I referenced it in my View. Not sure what else I can do.

Any ideas as to why it wouldn't be working? Client side validation has worked for everything else so far.

Thanks

PAUL HILES WROTE ON 24 JUN 2011

@Hayden - I have just tried the download and it is working ok for me. Here are some things to try: View the page source and check that your custom validation is being rendered as HTML5 data attributes. For example, search for 'data-val-**notequal**to' if you are using the NotEqualTo validation attribute from the example. If this not present, then there is a problem with your IClientValidatable code.

It is more likely to be a JavaScript error though. First check that you are referencing jquery, jquery.validate, jquery.validate.unobtrusive and the file containing your custom validators (devtrends.validation.js in the download). They need to be registered in this order. If you based your solution on the Internet application template, you will find that some of the script references have been added to the actual pages rather than the layout. Search the solution for script tags and remove duplicates. I would just put all four references in your layout (master page).

If this isn't you problem, then you probably have an error in your client-side validation js code. Look out for JavaScript errors when the page loads and also when you tab out of the field with the custom validation. Tools | Error Console in FireFox will list all JS errors.

Hope this helps.

HAYDEN WROTE ON 24 JUN 2011

I tried doing the things you said and I initially noticed that Firefox gave me a JScript error: "$.validator is undefined". I fixed this with some new includes. However, it still won't work. The scripts are added in the right order, like you specified. And my js code matches yours pretty closes

The HTML renders as data attributes fine and the code lines up with what it should also. Any other ideas?

STANK WROTE ON 28 JUN 2011

Excellent article, very helpful.

I have struck on problem though. The object I am validating is a property of the model for the view.

Where this is the case, the jquery line

var $other = $('#' + params.other);

will not retrieve the correct element, as the actual element ID will be something like 'ModelName_PropertyName'.

Any ideas how to handle this scenario?

Actually, I guess something like
var $other = $('#' + params.other);
if (!$other.length) {
$other = $('#' + $(element).attr('id').substring(0, $(element).attr('id').lastIndexOf('_') + 1) + params.other);
}

would work

PAUL HILES WROTE ON 28 JUN 2011

@StanK - yes, that would work. Microsoft actually has two functions (getModelPrefix and appendModelPrefix) in jquery.validate.unobtrusive that do the same thing but unfortunately they are not exposed so cannot be used for custom validators. You can of course, copy them or recreate the logic as you have done.

PAUL HILES WROTE ON 28 JUN 2011

@Hayden - difficult to debug without seeing the code. Try posting on stackoverflow with example code and I will take a look.

BHUVIN WROTE ON 27 JUL 2011

but Paul could you help me with a scenario , actually there is a complex model which has various collections as properties. Now these has various business rules i need to validate , now the custom validators i have written works well and pushes Error results in ValidationResult from IsValid method . Now the error is not getting pushed into the collection properties when there is more than one , Secondly how to take this client side without adding a pseudo property ?
Hope you would help me out !

TOM WROTE ON 29 JUL 2011

You mentioned using the Reflector to examine Mvc3.dll, but there is an easier way. Microsoft has released the source code to MVC3 under the MS-PL. You can get it from the download page for the MVC3 RTM at http://www.microsoft.com/download/en/details.aspx?id=4211.

TAKI WROTE ON 15 SEP 2011

Hello, I'm just a newbie in jQuery, I have downloaded your code but I don't know how to use the validation , using $("form").validate({....}) ? Or I must use server-side validate along with jquery client-side ? Please write more details . Thanks

PAUL HILES WROTE ON 15 SEP 2011

@taki - This article is concerned with using jQuery validate in conjunction with ASP.NET MVC3. in this scenario, server side validation attributes and used to automatically validate user input at the client and at the server. No explicit call to validate is required in this case. If you are using MVC3, take a look at part 1 of this article (see link at top of page) and follow along.

KALORY WROTE ON 29 SEP 2011

Thank you for the excellent article. I spent much time trying to see if Compare could do a validation for not equal instead of equal. I do have a special situation that I wonder if it is possible to remedy.

I have two dropdown lists with security questions. I am using your **notequal**to validator on the second dropdown to ensure the two questions are not the same and everything works perfectly according to the design.

when I do get a clientside error by selecting the same item in the second dropdown list as the first, the error is cleared only if the selected item in the second dropdown is changed. Is there a way to trigger the validation on the second dropdown list when the selection in the first dropdown list changes so that the error would go away?

This would remedy what appears to be a bug to the user when the error shows up and the first dropdown selection is changed and yet the error continues to state that the two cannot be the same when they are not the same.

Thanks

PAUL HILES WROTE ON 02 OCT 2011

@Kalory - All you need to do is use call the valid() method on the control with the validator on.

e.g. Add this in a jQuery document ready handler (untested):

$('#SecurityQuestion1').change(function() {

});

ANDRÉ LIMA WROTE ON 20 OCT 2011

I came across the need to use a Boolean field in radiobox
Then fix the validation:

```
var $other = $('#' + params.other);
var elems = $other.length;
var otherVal = "";

if (elems > 0) {
otherVal = ($other.attr('type').toUpperCase() == "CHECKBOX") ?
($other.attr("checked") ? "true" : "false") : $other.val();

} else
otherVal = $("input[name='" + params.other + "']:checked").val().toLowerCase();
```

I hope it was useful.

JON WROTE ON 02 DEC 2011

Thanks André your radio button fix helped me, cheers

Hi
I have a little problem with creating my custom validator. Maybe someone here can help me with this.

This is a part of my Edit action:
if (TryUpdateModel(category))
{
cmsDb.SaveChanges();
return RedirectToAction("All");
}

...and my first question is why validation is executed twice first on TryUpdateModel and then on SaveChanges ?
I'm asking becouse when I'm calling TryUpdateModel:

in my isValid function:
protected override ValidationResult IsValid(object value, ValidationContext validationContext)
{
var scat = validationContext.ObjectInstance.GetType().GetProperty("subcategories");
var scatVal = scat.GetValue(validationContext.ObjectInstance,null);
(...)
}
validationContext.ObjectInstance has correct "Category" type - so I have access to the property, but when validation is called second time by SaveChanges()
validationContext.ObjectInstance type is System.Data.Entity.Internal.InternalEntityEntry, and

validationContext.ObjectInstance.GetType().GetProperty("subcategories");

returns null...

PAUL HILES WROTE ON 05 DEC 2011

@wnk - It is firing twice because Entity Framework also uses data annotations to validate entities before saving to the db. Off the top of my head, I can think of two options for you:

(1) Think about moving away from using domain entities directly in MVC views/binding. I would have a separate view model that has all your validation. Use the view model within your view and controller action and then map to your domain entity before saving.

(2) Turn off validation within Entity Framework. Assuming that you are using code first, override OnModelCreating in your DbContext

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
Configuration.ValidateOnSaveEnabled = false;
}
```

WNK WROTE ON 05 DEC 2011

Yes.. I think moving validation to view model is good idea.
Thanks for help

Thank you for putting this together.

RITIKA WROTE ON 20 DEC 2011

Awesome post..very helpful..

Instead of show/hide div..how can i enable/disable the div?

Thanks

R BRADLEY WROTE ON 01 JAN 2012

This is really a good article. I'm making the conversion from Java (UX) to MVC (MS). The thing the seem to be missing is the 'AccountModels.cs', and maybe others. I downloaded the code from http://www.devtrends.co.uk/downloads/devtrends.validation.zip and the file(s) are not in the zip.

Thanks

PAUL HILES WROTE ON 03 JAN 2012

template. The download just includes the client and server code for the validators which you can then add to your project. Part one has some more information that may help you.

http://www.devtrends.co.uk/blog/the-complete-guide-to-validation-in-asp.net-mvc-3-part-1

RAUL VEJAR WROTE ON 18 JAN 2012

Awesome post, the best resource out there for MVC unobstrusive validation

Thanks

RAUL VEJAR WROTE ON 18 JAN 2012

One suggestion though, on your RequiredIf validation, there is a problem when the value you are checking against has any uppercase characters.
A simple solution is to use instead in the js:
var otherVal = (($other.attr('type').toUpperCase() == "CHECKBOX") ? ($other.attr("checked") ? "true" : "false") : $other.val()).toString().toLowerCase();

Or remove the ".ToLower()" on the ModelClientValidationRequiredIfRule function for the "value" parameter

I've opted for the 2nd one

Best Article.....

KATERINA WROTE ON 31 JAN 2012

Thank you for this post. Very useful!

KARIBE WROTE ON 17 FEB 2012

How do you use the RequiredOr in models?

VANDANA CHADHA WROTE ON 21 MAR 2012

Thanks for the detailed article.

I have defined the CustomValidator and it also both ValidationAttribute and IClientValidatable. Still when this validation fails, the controller gets called. Whereas when any of the other validations fail for example Required, the controller method does not get called.
What could I be doing wrong.

I have also added the foll. to the .chtml page:

<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

```
<script type="text/javascript">
// we add a custom jquery validation method
jQuery.validator.addMethod('greaterThan', function (value, element, params) {
if (!/Invalid|NaN/.test(new Date(value))) {
return new Date(value) > new Date($(params).val());
}
return isNaN(value) && isNaN($(params).val()) || (parseFloat(value) > parseFloat($(params).val()));
}, '');

// and an unobtrusive adapter
jQuery.validator.unobtrusive.adapters.add('dategreaterthanstart', {}, function (options) {
options.rules['greaterThan'] = true;
options.messages['greaterThan'] = options.message;
});
</script>
```

VINCENT WROTE ON 21 MAR 2012

The most complete and usefull post i have ever read. Contains hints that helped me solve a problem i found nowhere else.

MANISH KUMAR WROTE ON 17 APR 2012

I am facing a issue in rendering Hyperlink inside a Validation Message For a Text Box.

My Scenario is I want to display Help Link for rectifying Error, so user can input correct data. For Example,

User input Password during registration. As per business rule, password should be minimum of 4 characters and contain 1 capital and 1 number. Sample of Password is "Amaze007".

Error Message is

"Please enter password with our minimum requirement. Sample are link Password Help start link close

This message doesn't render HTML Anchor tag. Your help is appreciated.

BRIAN MCCLEARY WROTE ON 19 APR 2012

Paul, I appreciate your post and I have been using it extensively. However, I have one issue where I need the RequiredIf attribute to validate against a complex type. Please see my post at http://stackoverflow.com/questions/10236198/get-full-name-of-complex-type-from-modelclientvalidationrequiredifrule-method-in and let me know if you know of a solution. Thanks!

BRIAN MCCLEARY WROTE ON 25 APR 2012

If it helps anyone, I came up with a working solution to validating against complex types. I simply modified the following in the customValidation.js file:

```
jQuery.validator.addMethod("requiredif", function (value, element, params) {
if ($(element).val() != '') return true;
var prefix = getModelPrefix(element.name); // NEW LINE
var $other = $('#' + prefix + params.other); // MODIFIED LINE
var otherVal = ($other.attr('type').toUpperCase() == "CHECKBOX") ? ($other.attr("checked") ? "true" : "false") : $other.val();
return params.comp == 'isequalto' ? (otherVal != params.value) : (otherVal == params.value);
});
```

I also added the following method to that file (within the JQuery block so as to be only privately accessible):

```
function getModelPrefix(fieldName) {
return fieldName.substr(0, fieldName.lastIndexOf(".") + 1).replace(".","_");
}
```

BOON WROTE ON 30 APR 2012

Your explanation is far better than the examples in MVC 3 In Action, nice work.

JOHNNY IV YOUNG WROTE ON 26 SEP 2012

thank you very much...useful your validation **NotEqual**To!!!! i used in the project actual MVC4.

;)

Hi Paul,

Thank you for the excellent article.

Just have a query about the name of the otherProperty appearing as the variable name of the property in the model. So in my case, if the otherProperty is a string named OldPassword, then that is how it shows up in the error message as well.

Is there a way to make the error message use the display name(i.e. Old Password), rather than the variable name(OldPassword)?

Cheers,
Hitarth

Comments are now closed for this article.

About

DevTrends is owned by Paul Hiles who has over 15 years of experience developing Microsoft based applications. MORE »

Contact

Email me at **paul@devtrends.co.uk**

Search

enter search term

SEARCH

Latest Blog Posts

Fast Free Geolocation in .NET with freegeoip.net

3 Ways To Avoid An Anemic Domain Model In Entity Framework

Installing the ASP.NET Core 2.0 runtime store on Linux

Dependency Injection in action filters in ASP.NET Core

Custom response caching in ASP.NET Core (with cache invalidation)

A guide to caching in ASP.NET Core

Hashing, Encryption and Random in ASP.NET Core

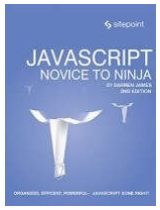
Create a Free Private NuGet Server with Continuous Deployment using VSTS

Creating your first shared library in .NET Core

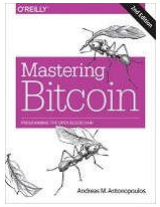
Conditional Middleware based on request in ASP.NET Core

VIEW ALL BLOG POSTS »

Reading List

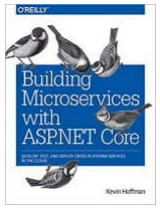


JavaScript: Novice to Ninja 2nd Edition (2017) - Darren Jones

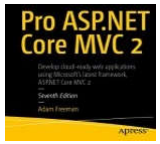


Mastering Bitcoin: Programming the Open Blockchain - Andreas M. Antonopoulos

Clean Architecture: A Craftsman's Guide to Software Structure and Design - Robert C. Martin



Building Microservices with ASP.NET Core - Kevin Hoffman


Pro ASP.NET
Core MVC 2
Seventh Edition
Adam Freeman
apress