dnrTV! featuring Peter Blum

Overview

Hello, I am Peter Blum. My expertise is in how users try to use web controls for data entry and what challenges they face. Being a developer of third party controls, my everyday job is to solve these problems within the controls I build.
In this dnrTV! broadcast, I will be showing some of the challenges in using the ASP.NET validation controls. I will reveal a few tricks using some undocumented javascript functions of the ASP.NET Validation Library.
I will also show you how I implemented solutions for these challenges in my own product, Professional Validation And More Suite ("VAM").
You are welcome to download a free license of VAM for use on non-production servers.

Projects overview

1) Four important but commonly overlooked features of the ASP.NET Validators
2) Requiring that at least one textbox has text
3) Enabling validators based on the state of other controls
4) Situations too complex for Validation Groups
5) Determining if at least one checkbox is marked in a GridView

Following along with a predefined web app

I have created an ASP.NET 2 web application with web forms for each study case in this tutorial. In addition, I have provided a PDF with much of the instructions found in this video.
To get the web application, go to http://www.peterblum.com/dnrtv.
That link will also provide you with access to your free Professional Validation And More license and a second web application that demonstrates how I implement each of these cases using VAM.

Case 1: Four important but commonly overlooked features of the ASP.NET Validators

Shows setting up a basic validation form with the things people overlook.

Example Form: Case 1 – Initial.aspx

This web form validates a textbox for date entry and offers OK and Cancel buttons.

1. How to validate a date using the CompareValidator

Users always miss the fact that by setting **Operator**=`DataTypeCheck`, it validates the format of the field based on the **Type** property.

```
<asp:CompareValidator ID="CompareValidator1" runat="server"
ControlToValidate="TextBox1"
ErrorMessage="Not a valid date"
Operator="DataTypeCheck"
Type="Date"></asp:CompareValidator>
```

When using VAM, choose the DataTypeCheckValidator.

2. CultureInfo determines the expected date and number formats

The ASP.NET validators evaluate the format using the current CultureInfo object on the page.
DEMO: Change the `<@ Page Culture= >` *property to a non-US format. Use French (Canada) whose date format is yyyy-MM-dd.*

```
<%@ Page Culture="fr-CA" %>
```

3. Cancel button needs CausesValidation=false

CausesValidation property is also used when you need to handle some types of validation on the server side alone by calling individual validator's `Validate()` method. We'll see an example of this later.

```
<asp:Button ID="Button2" runat="server" Text="Cancel" CausesValidation="False"
/>
```

4. Always test Page.IsValid or individual validator's IsValid property in postback event handler

This is the only way to properly implement server side validation. Server side validation is ESSENTIAL. Browsers that have javascript turned off or that don't support the level of scripting needed for ASP.NET Validators will bypass client-side validation. Hackers will take advantage of this loophole to launch SQL Injection attacks and Cross Site Scripting attacks.

```
protected void Button1_Click(object sender, EventArgs e)
{
   if (Page.IsValid) // part of server side validation
   {
      // Page is Valid. Save the contents of the form here
   }
}
```

For more of these commonly overlooked features

I wrote an article on the subject. Please visit http://aspalliance.com/699.

Case 2: Requiring that at least one textbox has text

Shows using a CustomValidator and two useful but undocumented validation scripting functions.
The RequiredFieldValidator only handles a single textbox. How do you require that at least one of two textboxes has text?
Use a CustomValidator because once the existing validators don't handle the desired rules, you must use a CustomValidator.

Example Form: Case 2 – Initial.aspx

This web form has two textboxes. We want to add a single validator that reports an error when both are blank.

How to do it:

1. Initially just assign the error message. Don't assign **ControlToValidate** to anything.

```
<asp:CustomValidator ID="CustomValidator1" runat="server" ErrorMessage="At least
one"
></asp:CustomValidator>
```

2. Add the ServerValidate event handler to the CustomValidator.

```
protected void CustomValidator1_ServerValidate(object source,
   ServerValidateEventArgs args)
{
// code pending
}
```

```
<asp:CustomValidator ID="CustomValidator1" runat="server" ErrorMessage="At least
one"
OnServerValidate="CustomValidator1_ServerValidate"
></asp:CustomValidator>
```

3. Determine the logic that determines if the page is valid or not.
   It is basically: TextBox1 has text OR TextBox2 has text. Create that logic in the ServerValidate method.
   Since spaces alone are not considered an entry, make sure you use the `Trim()` method on the **Text** property.

```
protected void CustomValidator1_ServerValidate(object source,
   ServerValidateEventArgs args)
{
   if ((TextBox1.Text.Trim().Length > 0) || (TextBox2.Text.Trim().Length > 0))
      args.IsValid = true;
   else
      args.IsValid = false;
}
```

4. Add the client side validation function. This code belongs after the `<form>` tag.

```
 <script language="javascript">
 function AtLeastOne(source, args)
 {
 // retrieve the text value of each textbox
   var vTextBox1Val = document.getElementById("TextBox1").value;
   var vTextBox2Val = document.getElementById("TextBox2").value;

 // trim using a built-in function of the ASP.NET Validation scripts
 // ValidatorTrim() takes one parameter, the string to trim.
```

```
// It returns the trimmed string.
  vTextBox1Val = ValidatorTrim(vTextBox1Val);
  vTextBox2Val = ValidatorTrim(vTextBox2Val);

  if ((vTextBox1Val.length > 0) || (vTextBox2Val.length > 0))
     args.IsValid = true;
  else
     args.IsValid = false;
}
</script>
```

Assign the function name to the **ClientValidationFunction** property.

```
<asp:CustomValidator ID="CustomValidator1" runat="server" ErrorMessage="At least
one"
OnServerValidate="CustomValidator1_ServerValidate"
ClientValidationFunction="AtLeastOne"
></asp:CustomValidator>
```

5. Make edits to each textbox fire the validator. First set **ControlToValidate** to TextBox1.

```
<asp:CustomValidator ID="CustomValidator1" runat="server" ErrorMessage="At least
one"
OnServerValidate="CustomValidator1_ServerValidate"
ClientValidationFunction="AtLeastOne"
ControlToValidate="TextBox1"
></asp:CustomValidator>
```

   At this point, neither textbox fires the validator after an edit. Why? The CustomValidator ignores a blank textbox when the **ControlToValidate** property is used unless you set the **ValidateEmptyText** property to `true`.

6. Set **ValidateEmptyText** to `true`

```
<asp:CustomValidator ID="CustomValidator1" runat="server" ErrorMessage="At least
one"
OnServerValidate="CustomValidator1_ServerValidate"
ClientValidationFunction="AtLeastOne"
ControlToValidate="TextBox1"
ValidateEmptyText="True"
></asp:CustomValidator>
```

   At this point, only textbox 1 fires the validator upon edit.

7. Make edits to TextBox2 fire the validator. Add this code before the `</form>` tag.

```
<script language="javascript">
ValidatorHookupControlID("TextBox2",
   document.getElementById("CustomValidator1"));  // MUST BE ClientID property
values
</script>
```


Case 2: How to do it in VAM

Download the VAM example from http://www.peterblum.com/dnrtv. Use Case 2
VAM's MultiConditionValidator creates boolean expressions from rules of other validators. In this case, you have a boolean expression of:
TextBox1 is required OR TextBox2 is required
You define the validator without coding and it creates both client and server side validation.

```
<VAM:MultiConditionValidator ID="MultiConditionValidator1" runat="server"
      Operator="OR" ErrorMessage="At least one" >
      <Conditions>
            <VAM:RequiredTextCondition ControlIDToEvaluate="TextBox1" />
            <VAM:RequiredTextCondition ControlIDToEvaluate="TextBox2" />
      </Conditions>
</VAM:MultiConditionValidator>
```

The RequiredTextCondition objects are the validation part of the RequiredTextValidator. They evaluate the textboxes and return a result indicating if the textbox is empty or not.
VAM includes 30 of these "condition objects", most associated with its 25 validators, but also some that look at attributes of fields like visibility, enabled, and styles.
By using the condition object of the MultiConditionValidator itself, you can create very powerful boolean expressions and often eliminate the need to create your own code using a CustomValidator.

Case 3: Enabling validators based on the state of other controls

Several ways to disable a validator because a checkbox is not checked.

Example Form: Case 3 – Initial(name).aspx

This web form has a checkbox and textbox. When the checkbox is marked, we want to validate that the textbox has an entry. Typically that uses a RequiredFieldValidator, but the RequiredFieldValidator alone doesn't handle this case.
There are separate forms for each case although they are identical.

Using the CustomValidator – InitialUsingCustomValidator.aspx

Advantages: Both client and server side support
Disadvantages: Have to recreate the logic of the existing validators. Especially tricky in validating something like a date.

How to do it:

1. Add the CustomValidator

```
<asp:CustomValidator ID="CustomValidator1" runat="server"
ControlToValidate="TextBox1"
ErrorMessage="Required"
ValidateEmptyText="True"
></asp:CustomValidator>
```

2. Add the ServerValidation event handler

```
protected void CustomValidator1_ServerValidate(object source,
ServerValidateEventArgs args)
{

}

<asp:CustomValidator ID="CustomValidator1" runat="server"
ControlToValidate="TextBox1"
ErrorMessage="Required"
ValidateEmptyText="True"
OnServerValidate="CustomValidator1_ServerValidate"
></asp:CustomValidator>
```

3. Write the logic of the validator it replaces (Compare, Range, or Required)

```
protected void CustomValidator1_ServerValidate(object source,
   ServerValidateEventArgs args)
{
   if (TextBox1.Text.Trim().Length > 0)
      args.IsValid = true;
   else
      args.IsValid = false;
}
```

4. Add an `if` statement at the beginning that evaluates the other control. If that control indicates disable, return args.IsValid = `true`

```
protected void CustomValidator1_ServerValidate(object source,
   ServerValidateEventArgs args)
{
   if (CheckBox1.Checked)
   {
```

```
        if (TextBox1.Text.Trim().Length > 0)
           args.IsValid = true;
        else
           args.IsValid = false;
     }
     else
        args.IsValid = true;
  }
```

5. Add the client-side validation function with the same logic. It goes inside the `<form>` tag.

```
<script language="javascript">
function RequiredWhenChecked(source, args)
{
  if (document.getElementById("CheckBox1").checked)
  {
     if (ValidatorTrim(args.Value).length > 0)
        args.IsValid = true;
     else
        args.IsValid = false;
  }
  else  // disabled - act like its valid
  {
     args.IsValid = true;
  }
}
</script>
```

   Assign the function name to the **ClientValidationFunction** property.

```
<asp:CustomValidator ID="CustomValidator1" runat="server"
ControlToValidate="TextBox1"
ErrorMessage="Required"
ValidateEmptyText="True"
OnServerValidate="CustomValidator1_ServerValidate"
ClientValidationFunction="RequiredWhenChecked"
></asp:CustomValidator>
```

Server side to disable using autopostback – InitialUsingAutoPostBack.aspx

Advantages: Uses an existing validator. You don't write any javascript.
Disadvantages: Requires a postback just to update the validator's state. Requires writing server side code. Postback also affects the scroll position of the page so be sure to set **Page.MaintainScrollPositionOnPostBack** = `true`.

How to do it:

1. Add the RequiredFieldValidator and attach it to the TextBox

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
ErrorMessage="Required" ControlToValidate="TextBox1">
</asp:RequiredFieldValidator>
```

2. Set the **AutoPostBack** property to `true` on the CheckBox

```
<asp:CheckBox ID="CheckBox1" runat="server" Text="Other" AutoPostBack="True" />
```

3. In Page_Load, test the state of the control that enables/disables the validator and apply the result to the Enabled property of the validator.

```
protected void Page_Load(object sender, EventArgs e)
{
// when checkbox is checked, the RequiredFieldValidator must be enabled
```

```
        RequiredFieldValidator1.Enabled = CheckBox1.Checked;

    }
```

4. In Page_Load, set **MaintainScrollPositionOnPostBack** to `true`

```
protected void Page_Load(object sender, EventArgs e)
{
// when checkbox is checked, the RequiredFieldValidator must be enabled
   RequiredFieldValidator1.Enabled = CheckBox1.Checked;

   Page.MaintainScrollPositionOnPostBack = true;
}
```

Client side code toggles the enabled state of the validator – InitialUsingClientSide.aspx

Advantages: Uses an existing validator
Disadvantages: Requires custom code both on the client and server side.
The server side gets a bit tricky because any change to the enabled state on the client-side is not transferred to the server.

How to do it:

1. Add the RequiredFieldValidator and attach it to the TextBox

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
ErrorMessage="Required" ControlToValidate="TextBox1">
</asp:RequiredFieldValidator>
```

2. In `Page_Load()`, setup the client-side onclick or onchange event to call the `ValidatorEnable()` function. For checkboxes and radiobuttons, use onclick. For textboxes and lists, use onchange.

```
protected void Page_Load(object sender, EventArgs e)
{
  // Client-side support
  // The state of the checked property on the checkbox determines if the
validator is enabled or not
  // Use the validation function ValidatorEnable() which has these parameters:
  // * validator - DHTML element for the Validator. Use
document.getElementById('ClientID of the validator')
  // * enabled - boolean. When true, its enabled.

  string vScript = "ValidatorEnable(document.getElementById('" +
RequiredFieldValidator1.ClientID + "'), this.checked);";
      CheckBox1.Attributes.Add("onclick", vScript);

}
```

3. In `Page_Load()`, test the state of the control that enables/disables the validator and apply the result to the **Enabled** property of the validator.

```
// Server-side support
// when checkbox is checked, the RequiredFieldValidator must be enabled
// This code has two effects:
// 1. When Page.Validate() is called by the Button.Click event, the validator
will be properly setup
// 2. When the page is output to the browser, all of the HTML for the validator
will be in place, reflecting its enabled or disabled state
```



```
RequiredFieldValidator1.Enabled = CheckBox1.Checked;
```

Case 3: How to do it in VAM

Download the VAM example from http://www.peterblum.com/dnrtv. Use Case 3
VAM provides the Enabler property on each of its validators. The Enabler provides an extensive list of rules to evaluate the state of controls on the page. It provides both client-side and server side functionality without writing any code.

```
<vam:RequiredTextValidator ID="RequiredTextValidator1" runat="server"
      ErrorMessage="Required" ControlIDToEvaluate="TextBox1">
      <EnablerContainer>
            <vam:CheckStateCondition ControlIDToEvaluate="CheckBox1"
                  Checked="true" EvaluateOnClickOrChange="False"/>
      </EnablerContainer>
</vam:RequiredTextValidator><br />
```

Like in Case 2, this control is using one of the many “condition objects”. This one evaluates the state of a checkbox or radiobutton.

Case 4: Situations too complex for Validation Groups

How to work around limitations of Validation Groups
Validation groups let a button or other submit control validate from a specific list of validators. You assign each group a unique name and assign it to the ValidationGroup property on the submit control, associated validators and even the ValidationSummary.
What happens when a validator may be part of several groups?
Validation Groups do not support this case and must be replaced by server side code.

Example Form: Case 4 – Initial.aspx

This form has two textboxes. The top one must be validated by both buttons. The bottom textbox is validated by the first button.

How to do it

1. Do not use Validation Groups. If you have assigned the validation group name to controls, you can leave it alone or clear the **ValidationGroup** property.

2. Turn off client-side validation on the submit control by setting its **CausesValidation** property to `false`.
```
<asp:Button ID="Button1" runat="server"
Text="Submit"
CausesValidation="False" />
<asp:Button ID="Button2" runat="server"
Text="Use the address you have on record"
CausesValidation="False" />
```

3. Add the Click event handler methods for both buttons.
```
protected void Button1_Click(object sender, EventArgs e)
{

}
protected void Button2_Click(object sender, EventArgs e)
{

}

<asp:Button ID="Button1" runat="server"
Text="Submit"
CausesValidation="False"
OnClick="Button1_Click"  />
<asp:Button ID="Button2" runat="server"
Text="Use the address you have on record"
CausesValidation="False"
OnClick="Button2_Click" />
```

4. In each post back event handler method, call the `Validate()` method on each validator that is associated with that button. Once done, test that all of those validators have **IsValid**=`true` before saving.
```
protected void Button1_Click(object sender, EventArgs e)
{
   // button1 requires both validators to be valid
   ContactNameRequired.Validate();
   AddressRequired.Validate();
   if (ContactNameRequired.IsValid && AddressRequired.IsValid)
```



```
      {
         // Page is Valid. Save the contents of the form here
      }
   }
   protected void Button2_Click(object sender, EventArgs e)
   {
      // button2 requires the contact validator to be valid
      ContactNameRequired.Validate();
      if (ContactNameRequired.IsValid)
      {
         // Page is Valid. Save the contents of the form here
      }

   }
```

Case 4: How to do it in VAM

Download the VAM example from http://www.peterblum.com/dnrtv. Use Case 4
VAM's Validation Group works like the ASP.NET validators, except it's Group property can handle multiple Validation Group names. The Validation Group name supports the "*" character to allow matching to all or a pipe delimited list of group names ("groupname1|groupname2"). It automatically works for client-side validation.

```
<asp:TextBox ID="ContactNameTextBox" runat="server" Width="200px"></asp:TextBox>
<vam:RequiredTextValidator ID="ContactNameRequired" runat="server"
      ControlIDToEvaluate="ContactNameTextBox"
      ErrorMessage="Required" Group="All|Contact">
 </vam:RequiredTextValidator>

<asp:TextBox ID="AddressTextBox" runat="server" Rows="4"
      TextMode="MultiLine" Width="200px"></asp:TextBox>
<vam:RequiredTextValidator ID="AddressRequired" runat="server"
      ControlIDToEvaluate="AddressTextBox"
      ErrorMessage="Required" Group="All">
</vam:RequiredTextValidator>

<vam:Button ID="Button1" runat="server" Text="Submit"
      OnClick="Button1_Click" Group="All" />
<vam:Button ID="Button2" runat="server" Text="Use the address you have on record"
      OnClick="Button2_Click" Group="Contact" />
```


Case 5: Determining if at least one checkbox is marked in a GridView

Using a CustomValidator with a list of controls

Always use a CustomValidator when there are no existing validators that handle the task. That's the case here. The challenge is to know the IDs of every checkbox in the grid. Once known, just iterate through those checkboxes to find one that is marked checked. The same idea can be applied to radiobuttons.

Example Form: Case 5 – Initial.aspx

This form has a GridView. Its first column is template with a CheckBox control. When the user clicks the button, it should report an error if none are checked.

How to do it

1. Create an ArrayList object as a field defined on your Page class.
```
private ArrayList fCheckBoxes = new ArrayList();
```

2. In the GridView's RowCreate event, get the checkbox object for that row and assign it to the ArrayList.
```
protected void GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
   // collect the CheckBox objects into fCheckBoxes
   CheckBox vCheckBox = e.Row.FindControl("CheckBox1") as CheckBox;
   if (vCheckBox != null)
      fCheckBoxes.Add(vCheckBox);
}
```

3. Add the CustomValidator below the GridView and attach a method to its **ServerValidate** event.
```
<asp:CustomValidator ID="CustomValidator1" runat="server"
ErrorMessage="Mark at least one checkbox"
OnServerValidate="CustomValidator1_ServerValidate"
></asp:CustomValidator>

protected void CustomValidator1_ServerValidate(object source,
   ServerValidateEventArgs args)
{
// code pending
}
```

4. Write the server side validation function to iterate through that list looking at the **Checked** property for one that is `true`. If none are found, return `false`.
```
protected void CustomValidator1_ServerValidate(object source,
   ServerValidateEventArgs args)
{
   // if any checkbox is checked, its done
   foreach (CheckBox vCheckBox in fCheckBoxes)
      if (vCheckBox.Checked)
      {
         args.IsValid = true;
         return;  // done
      }
   args.IsValid = false;
}
```

5. Override the Page's `OnPreRender()` method to output an array of ClientIDs associated with checkboxes into the javascript of the page. Use the method

`Page.ClientScript.RegisterArrayDeclaration()`. It will create a client-side array which we'll name GridViewCheckboxes.

```
protected override void OnPreRender(EventArgs e)
{
   base.OnPreRender(e);
   // provide an array of clientids to the client-side
   foreach (CheckBox vCheckBox in fCheckBoxes)
      Page.ClientScript.RegisterArrayDeclaration(
           "GridViewCheckboxes", "'" + vCheckBox.ClientID + "'");
}
```

6. Write the client-side evaluation function to iterate through the GridViewCheckboxes array. Convert its values to DHTML elements and evaluate their checked attribute.

```
<script language="javascript">
function AtLeastOneChecked(source, args)
{
   for (var vI = 0; vI < GridViewCheckboxes.length; vI++)
   {
      var vCheckBox = document.getElementById(GridViewCheckboxes[vI]);
      if (vCheckBox.checked)
      {
         args.IsValid = true;
         return;
      }
   }  // for
   args.IsValid = false;   // none found
}
</script>
```


Case 5: How to do it in VAM

Download the VAM example from http://www.peterblum.com/dnrtv. Use Case 5
VAM provides the CountTrueConditionsValidator to count the number of fields at a particular state, such as a checkbox being checked. You still need to do some setup in the GridView.**RowCreated** event, but the rest is automatic, including client side validation support.

```
<asp:GridView ID="GridView1" runat="server" OnRowCreated="GridView1_RowCreated">
      <Columns>
            <asp:TemplateField>
                  <ItemTemplate>
                        <asp:CheckBox ID="CheckBox1" runat="server" /> 
                  </ItemTemplate>
             </asp:TemplateField>
      </Columns>
</asp:GridView>

<VAM:CountTrueConditionsValidator ID="CountTrueConditionsValidator1" runat=server
      ErrorMessage="Mark at least one checkbox" Minimum="1" />

protected void GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
   // add the checkboxes into the CountTrueConditionsValidator
   // by creating a CheckStateCondition and adding it into the validator's
Conditions property.
   CheckBox vCheckBox = e.Row.FindControl("CheckBox1") as CheckBox;
   if (vCheckBox != null)
      CountTrueConditionsValidator1.Conditions.Add(
         new PeterBlum.VAM.CheckStateCondition(
                  CountTrueConditionsValidator1, vCheckBox));
}
```

