15

Validation

This chapter covers

* Implementing Data Annotations
* Extending the ModelMetaDataProvider
* Enabling ASP.NET Ajax client-side validation

The ASP.NET MVC 1.0 release provided a lot of out-of-the-box functionality, but one common piece was missing: user input validation. Integrating validation frameworks with the 1.0 release was quite difficult, as the hooks to put in validation were not fully formed. With ASP.NET MVC 2 comes full support for validation frameworks, as well as built-in support for Microsoft's Data Annotations library. From the initial login screen in many web applications, some level of easy validation is needed for productive application development. In this chapter, we will examine the built-in validators from the Data Annotations library. Next, we will look at extending the model metadata providers with richer, more convention-driven behavior. Finally, we will look at enabling client-side validation support.

15.1 Validation with Data Annotations

Data Annotations, introduced with the .NET 3.5 SP1 release, are a set of attributes and classes that allow you to decorate your classes with metadata. While .Net 3.5 SP1 does not provide validation execution, this support has been added in .Net 4. The Data Annotations attributes, part of the System.ComponentModel.DataAnnotations assembly, rely on other libraries to inspect the annotation information. Since a validation library for thick-client applications will have very different needs than one for an MVC application, this is not an oversight but rather an explicit design decision.

The Data Annotation attributes control more than validation. Some are used for the new templating features, as we saw in Chapter 3 with the DisplayName and DataType attributes. The attributes controlling specifically validation are listed in table 1.1 below.

Table 1.1 The Data Annotations attributes used for validation

|  |  |
| --- | --- |
| Attribute | Description |
| RequiredAttribute | Specifies that a data field value is required. |
| RangeAttribute | Specifies the numeric range constraints for the value of a data field. |
| RegularExpressionAttribute | Specifies that a data field value must match the specified regular expression. |
| StringLengthAttribute | Specifies the maximum length of characters that allowed in a data field. |

As part of ASP.NET MVC 2 are a set of backing validation classes to provide validation for the metadata indicated. To demonstrate the validation attributes, let's first look at a screen that might need some validation. In figure 15.1 below, we see an edit screen that includes a company name and email address.

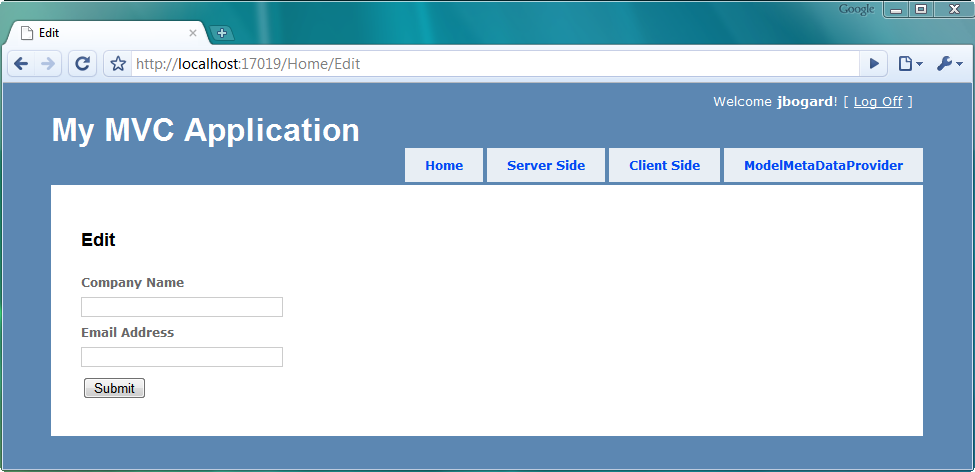


Figure 15.1 An edit screen with a required field

In our application, Company Name is a required field and email address is optional. To indicate that the Company Name field is required, we use the RequiredAttribute as shown in listing 15.1 below.

Listing 15.1 Decorating our model with data annotation attributes

public class CompanyInput

{

[Required] #1

public string CompanyName { get; set; }

[DataType(DataType.EmailAddress)] #2

public string EmailAddress { get; set; }

}

We decorate the CompanyName property with the RequiredAttribute (1). Additionally, we can also decorate the EmailAddress attribute with the DataTypeAttribute if we want to take advantage of custom email address templates. In our view, we need to display potential validation error messages. We can accomplish this in several ways. If we are using the model templates, the validation messages are already included in the template, as shown in listing 15.2.

Listing 15.2 The edit view using editor templates for displaying validation messages

<h2>Edit</h2>

<% using (Html.BeginForm()) { %>

<%= Html.EditorForModel() %> #1

<button type="submit">Submit</button>

<% } %>

The default editor model templates (1) include side-by-side input elements and validation messages. For finer-grain control of output, we can use the HtmlHelper extension methods for validation. The ValidationSummary extension provides a summary list of validation errors, usually displayed at the top of the form. For validation errors for specific model properties, we can use the ValidationMessage and expression-based ValidationMessageFor methods.

With our validation messages in place, we need to actually check that our model is valid in the resultant POST action in our controller. We can decorate our model with validation attributes all we like, but it is still up to us to handle validation errors in our controller action, shown in listing 15.3.

Listing 15.3 Handling validation errors in our controller action

[HttpPost]

public ActionResult Edit(CompanyInput input)

{

if (ModelState.IsValid) #1

{

return View("Success"); #2

}

return View(new CompanyInput()); #3

}

In our Edit POST action, we first check to see if there are any ModelState errors (1). The MVC validation engine places validation errors in ModelState, aggregating the existence of any errors into the IsValid property. If there are no errors, we show the Success view (2). Otherwise, we display the original Edit view, now with validation errors inline (3). To display our validation errors, we simply need to post our form without the company name filled out. The resulting page is shown in figure 15.2 below.

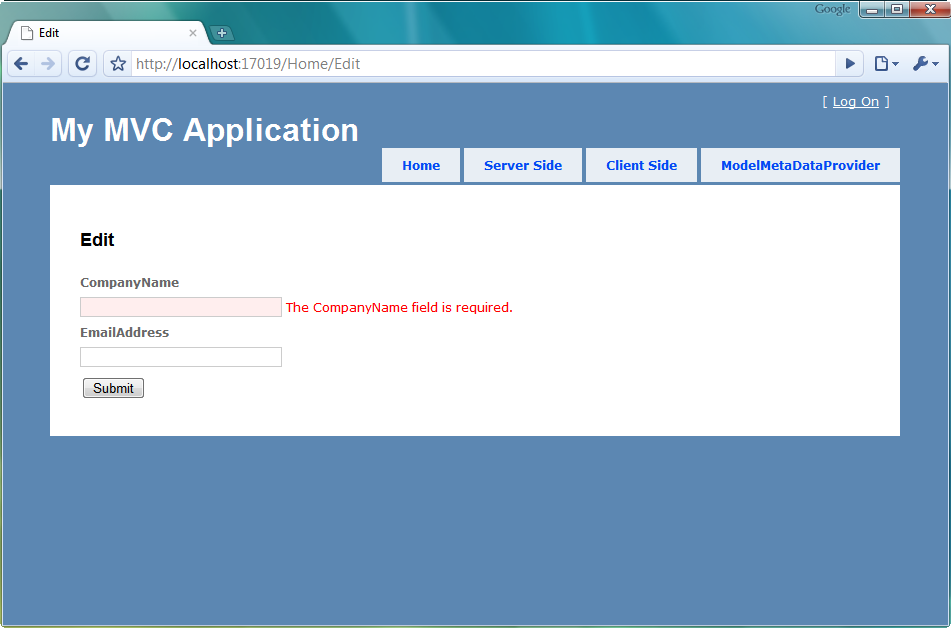


Figure 15.2 Validation error from the missing company name field

By submitting a form with the missing company name field, our validation message showed up correctly. To display the validation message, we needed to first decorate our model with the Data Annotations validation attribute. Next, we added code in our controller action to handle validation errors. Finally, we used the appropriate HtmlHelper extensions to display our validation errors. In the figure above, there is still a problem with our screen and the validation error message. Both the validation error message and input label are displayed as "CompanyName" with no space. However, we would like to always include spaces between words in our labels. One way of fixing the label would be to include a DisplayNameAttribute (part of the System.ComponentModel namespace). Since it is a common occurrence to simply display the property name with spaces between words, we will examine extending the built-in ModelMetadataProvider class to automatically include spaces.

15.2 Extending the ModelMetadataProvider

As we saw in the previous section, many new features in ASP.NET MVC 2 use model metadata. Templates use model metadata to display input elements and display text, and the validation providers use model metadata to execute validation. The model metadata is populated from an implementation of a ModelMetadataProvider class, which by default is the DataAnnotationsModelMetadataProvider class.

If we want our model metadata to be populated from sources other than Data Annotations, we would need to create a ModelMetadataProvider implementation, shown in listing 15.4 below.

Listing 15.4 The abstract ModelMetadataProvider class

public abstract class ModelMetadataProvider {

public abstract IEnumerable<ModelMetadata> |#1

GetMetadataForProperties(object container, |#1

Type containerType); |#1

public abstract ModelMetadata |#2

GetMetadataForProperty(Func<object> modelAccessor, |#2

Type containerType, string propertyName); |#2

public abstract ModelMetadata |#3

GetMetadataForType(Func<object> modelAccessor, |#3

Type modelType); |#3

}

The ModelMetadataProvider class includes methods to get ModelMetadata for each member in the type (1), ModelMetadata for a specific property (2), and ModelMetadata for a particular type (3). However, we only need to override specific behavior of the existing DataAnnotationsModelMetadataProvider class. To assist in model metadata scenarios where the metadata is pulled from traditional classes, properties and attributes, the AssociatedMetadataProvider class provides some common functionality. Derived classes, such as the DataAnnotationsModelMetadataProvider class, only need to build ModelMetadata from already-discovered attributes. In our case, we want to modify the behavior of the DisplayName model metadata. In the built-in case, the ModelMetadata's DisplayName property comes from the DisplayNameAttribute if supplied. We may want to still supply the DisplayName value through an attribute.

In listing 15.5 below, we extend the built-in DataAnnotationsModelMetadataProvider with this more useful display name behavior.

Listing 15.5 Our custom, conventions-based model metadata provider

public class ConventionProvider :

DataAnnotationsModelMetadataProvider #1

{

protected override ModelMetadata CreateMetadata( #2

IEnumerable<Attribute> attributes,

Type containerType,

Func<object> modelAccessor,

Type modelType,

string propertyName)

{

var meta = base.CreateMetadata(attributes, containerType, modelAccessor, modelType, propertyName); #3

if (meta.DisplayName == null) #4

meta.DisplayName = meta.PropertyName.ToSeparatedWords(); #5

return meta; #6

}

}

To build our convention-based display name scheme, we first create a class that inherits from the DataAnnotationsModelMetadataProvider class (1). This class provides quite a lot of functionality out of the box, but we only need to override the CreateMetadata method (2). Since the base class provides a lot of behavior we want to keep, we first call the base class method (3) and store its results in a local variable. Since we might override the display name with an attribute, we only want to modify its behavior if the display name was not already set (4). If that value was not set, we want to separate the property name into individual words, with the ToSeparatedWords extension method (5). Finally, we return the ModelMetadata object containing the modified display name (6).

The ToSeparatedWords extension method, shown in listing 15.6 below, is a rather naïve regular expression separating out Pascal cased identifiers into individual words.

Listing 15.6 The ToSeparatedWords extension method

public static class StringExtensions

{

public static string ToSeparatedWords(this string value)

{

if (value != null)

return Regex.Replace(value, "([A-Z][a-z]?)", " $1").Trim();

return value;

}

}

With our custom ModelMetadataProvider built, we need to configure ASP.NET MVC to use our new provider. The typical location for this customization is in the Global.asax file, shown in listing 15.7 below.

Listing 15.7 Configuring the new ModelMetadataProvider

protected void Application\_Start()

{

RegisterRoutes(RouteTable.Routes);

ModelMetadataProviders.Current = new ConventionProvider(); #1

}

To override the model metadata provider used, we set the ModelMetadataProviders.Current property and supply our custom provider. With our custom provider in place, the labels displayed on both the input and validation messages have a much friendlier look, shown in figure 15.3.

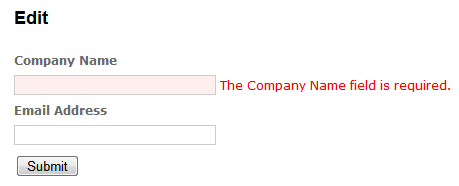


Figure 15.3 The edit screen with friendlier input labels and error messages

With our convention-based modification to the built-in DataAnnotationsModelMetadataProvider, we can rely on our property names more for displaying labels and error messages. Otherwise, we would either need to avoid using the editor and display templates, or supply the display name in attribute form in many, many more places. In the examples so far, we have used strictly server side validation. However, ASP.NET MVC 2 includes support for dual client and server side validation also, as we will see in the next section.

15.3 Client-side validation with ASP.NET Ajax

With the advent of modern browsers and rich client behavior, client-side validation in the form of JavaScript has become more popular. The feedback from a client-side validation is much quicker than server side validation as the round-trip from client to server is avoided. Many client-side validation frameworks also include advanced functionality such as executing validation when input element focus is lost, so that a user tabbing through form elements gets dynamic validation messages.

Building this behavior from scratch is most often cost-prohibitive and wasteful, as many client validation frameworks have been under development and in production for years. However, the real trick with integrating client-side validation has been linking client and server-side validation, without repeating a lot of code. With ASP.NET MVC 2, the potential duplication is greatly reduced. ASP.NET MVC 2 ships with support for using the Microsoft ASP.NET Ajax library for performing client-side validation. There is also integration with jQuery available as part of the MvcFutures project which can be found at http://aspnet.codeplex.com.

First, we need to make sure that our application includes both the ASP.NET Ajax script library as well as the MVC validation support library, shown in figure 15.4.

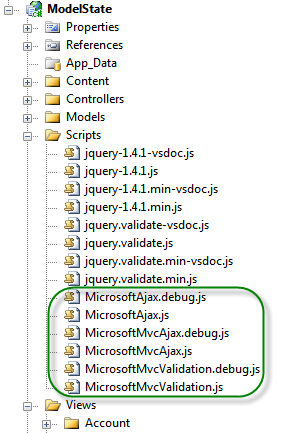


Figure 15.4 The ASP.NET AJAX client libraries and supporting debug files

With our JavaScript libraries included in the project, we now need to include them in our pages. This can be done in the master page, shown in listing 15.8.

Listing 15.8 The master page with script files included

<head runat="server">

<title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>

<link href="../../Content/Site.css" rel="stylesheet" type="text/css" />

<script src="../../Scripts/MicrosoftAjax.js" type="text/javascript"></script> #1

<script src="../../Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>

<script src="../../Scripts/MicrosoftMvcValidation.js" type="text/javascript"></script> #2

</head>

Since each JavaScript library build off others, it is important that the above files are included in the correct order. We first register the ASP.NET Ajax library (1) and later register the MVC validation support library (2). If we are using jQuery as our validation framework, we will include the MicrosoftMvcJQueryValidation file instead (included with MvcFutures). With our client libraries included in the master page, we can selectively opt-in to validation on individual pages. This is as simple as the EnableClientValidation HtmlHelper extension method, shown in listing 15.9 below.

Listing 15.9 Enabling client validation in our view

<h2>Client Validation</h2>

<% Html.EnableClientValidation(); %> #1

<% using (Html.BeginForm("Edit", "Home")) { %> #2

<%= Html.EditorForModel() %>

<button type="submit">Submit</button>

<% } %>

The EnableClientValidation method (1) merely turns on a flag in ViewContext. It is the form helper method BeginForm (2) that emits the pertinent client-side scripts to enable validation. The EnableClientValidation needs to be placed before the BeginForm method in your view to correctly enable scripts. In our original screen with company name and email address, the model metadata is emitted as a set of JSON objects. This JSON, shown in figure 15.5 below, includes the model metadata information, validation information and model information in the form a well-structured JSON object.

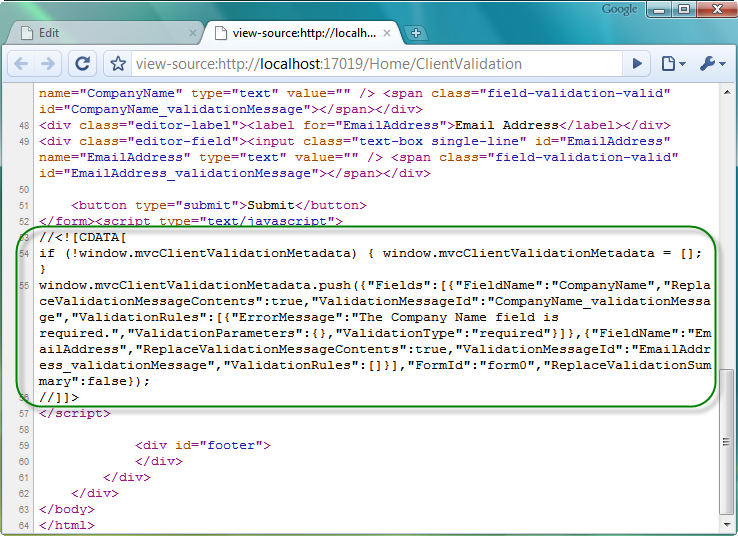


Figure 15.4 The generated metadata and validation information.

The generated validation information combines with the MVC validation library to act as a bridge between the client-side validation framework and the server-side model metadata emitted as JSON. For example, we can see above that there seems to be some information about the CompanyName field, as well as a validation message for the required field validation. With our custom validators in place, we can now exercise client-side validation by submitting our form with missing company name information. The result does not post back, as shown in figure 15.5.

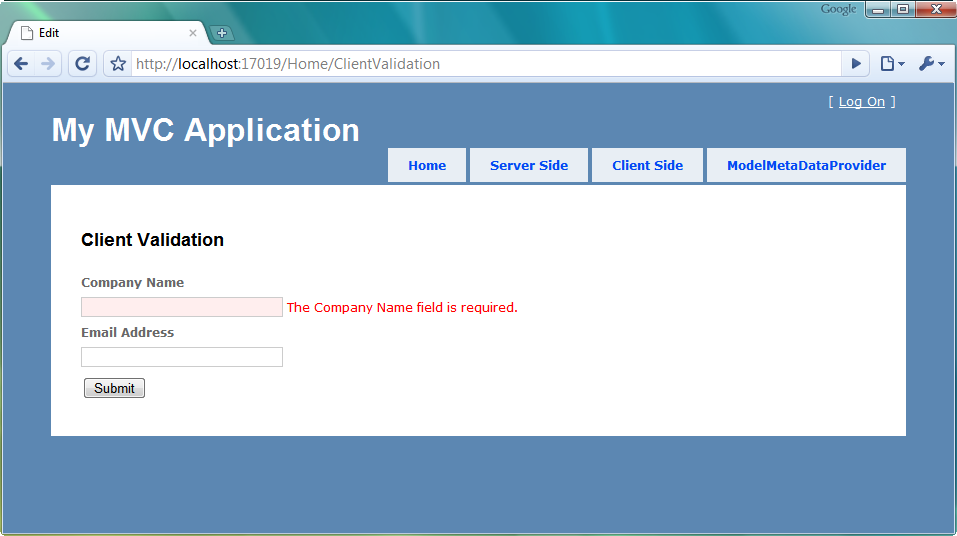


Figure 15.5 The client side validation in action

Because our server-side validation is still in place, we can be confident that even browsers without JavaScript available or enabled will still have validation executed. ASP.NET MVC 2 also supports custom validators, with plugins for both server and client-side behavior. It is just up to the developers to decide how much richness is needed in the client-side behavior.

15.4 Summary

With the release of ASP.NET MVC 2, a large gap was closed in much needed functionality for validation. Rich, extensible server-side validation in the form of Data Annotations and support for popular client-side validation helps remove much of the custom-built validation solutions prevalent in MVC 1.0 applications. The integration of a metadata model allowed validation and HTML generation tools to share metadata information for displaying labels, generating input elements, and executing and displaying validation errors. Since many applications demand a rich client-side experience, MVC 2 also includes support for two popular client-side validation libraries, jQuery and ASP.NET Ajax. Validation is now as simple as decorating our models with attributes. In the next chapter, we move into the advanced ASP.NET MVC topics, starting with routing.