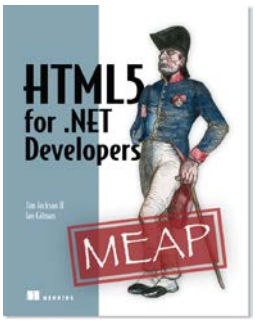
MANNING PUBLICATIONS



HTML5 for .NET Developers
By Jim Jackson II and Ian Gilman

This article based on chapter 4 of HTML5 for .NET Developers shows you some important MVC framework features that are relevant to HTML application builders.

An MVC 3 Application in Action

To show you some important MVC framework features that are relevant to HTML application builders, we will build out a real application using a sample application called MvcNewsletter. These features are:

- Posting form data to a specific controller action.
- Client and server business rule validation.
- Redirecting a controller to a different view.

Building a data entry form

The simplest means of data entry in any web application is to post an HTML form to some location on a web server and use the posted data to enter data into storage. In our sample application, we will post a user's name, email address and confirmation that he or she agrees to fictitious terms and conditions. The page that hosts our form will include a client-side business rule implementation and the same rules will be enforced on the server.

The form that presents the appropriate screen to the user is in listing 1.

Listing 1 The newsletter signup form in the Home\Index.cshtml view

```
@{}                                                          #A
<section>
@using (Html.BeginForm(                                      #B
   "SignupNow",                                              #B
   "Home",                                                   #B
   FormMethod.Post,                                          #B
   new { id = "signup" })) {                                 #C
   <fieldset>
      <legend>Sign Up to Receive Our Free Newsletter!</legend>
      <div>
         <label for="emailAddr">Email Address:</label>
         <input id="emailAddr" name="emailAddr" type="text" />   #D
      </div>
      <div>
         <label for="fastName">First Name:</label>
         <input id="firstName" name="firstName" type="text" />
      </div>
      <div>
         <label for="lastName">Last Name:</label>
         <input id="lastName" name="lastName" type="text" />
      </div>
      <div>
         <input type="checkbox" id="chkAgree" name="chkAgree" />
         <label for="chkAgree" id="lblAgree" class="lblCheck" >
            I agree with all terms and conditions</label>
```

```
        </div>
        <div>
            <br />
            <input type="submit" value="Join Now" />                    #E
        </div>
    </fieldset>
}
    <script type="text/javascript"                                      #F
        src="@Url.Content("~/Scripts/htm5Net.js")" ></script>           #F
    <script type="text/javascript">
        $(document).ready(function () {                                 #G
            _app.init();
        });
    </script>
</section>
```

Our view does not take a model as a parameter (#A), although that would be an alternate way of automatically passing an object to the controller. Our HTML form is created using Razor syntax (#B) and, when it renders in the browser, will be directed to the SignupNow method of the Home controller by means of a `post` method. A collection of HTML attributes can also be passed in as a dynamic object (#C) that will allow our form to get an id value for validation purposes. Each element in the form will be posted using its name (#D), and a simple submit button (#E) will post the form with no additional code. We will also include our custom JavaScript library (#F) so that we can validate business rules prior to the form's posting. These rules are wired up when the page is ready (#G) using jQuery.

Element identifiers and form properties

When our form is presented on the server, it will have an attributed action that points to "/Home/SignupNow", a "`signup`" id and a `post` method. Each input object has a name value that will be posted to the server along with the value in the text box. The checkbox will return a value that can be converted to a Boolean but is usually a string of "on" or "off". The names of elements are important because these names will be automatically parsed into the same parameter names as the controller method. If the names do not match, null values will automatically be used for parameter values and your data will be lost.

Using the controller to handle posted form data

A normal `ActionResult` method in an MVC controller will handle post operations without a specific attribute. If you have an overloaded method though, you can use attributes to specify methods for `get` and `post`. If the name of the controller method is unique, MVC will automatically route the posted data to it and no additional attributes are necessary on the controller.

Validating posted data on the server

At this point, we are going to deviate a little from the standard MVC pattern. Normally, a model object would be used to validate business rules with data annotations and coded rules. In our case, because our goal is to show how to move valid data between the server and the browser, we will forgo having a strongly typed model and just perform the validation in the controller. You will also notice that in the templated code earlier there are a number of calls to the ModelState property of the controller. This will also be circumvented for brevity.

So, in this snippet of code, we will just check value directly and then verify that all rules are valid. The purpose here is to compare a simple check in C# with the same check using jQuery.Validation coming up in the next section.

```
var validEmail = Regex.IsMatch(emailAddr,
@"^[a-zA-Z][\w\.-]*[a-zA-Z0-9]@[a-zA-Z0-9][\w\.-]*
    [a-zA-Z0-9]\.[a-zA-Z][a-zA-Z\.]*[a-zA-Z]$"
);
var validFirstName = (firstName.Length >= 3 && firstName.Length <= 80);
var validLastName = (lastName.Length >= 3 && lastName.Length <= 80);
var validAgree = (chkAgree == "on");
if (validEmail && validFirstName && validLastName && validAgree)
    // Valid!
```

How to get MVC to use a different view

Next, we will take a look at the entire `SignupNow` method of the Home controller. It adds the user to the repository and then returns a view named "Confirmation." This view has a model that is just a simple value type (Boolean). The repository will return a string message indicating that a duplicate email address was entered and the controller responds accordingly.

Listing 2 The ActionResult for a posted form using convention-based routes

```
[HandleError]
public ActionResult SignupNow(string emailAddr,
   string firstName, string lastName, string chkAgree)
{
   // Test Business Rules again before adding to the store
   var validEmail = Regex.IsMatch(emailAddr,
   @"^[a-zA-Z][\w\.-]*[a-zA-Z0-9]@[a-zA-Z0-9][\w\.-]*[a-zA-Z0-9]\.
      [a-zA-Z][a-zA-Z\.]*[a-zA-Z]$");
   var validFirstName =
      (firstName.Length >= 3 && firstName.Length <= 80);
   var validLastName =
      (lastName.Length >= 3 && lastName.Length <= 80);
   var validAgree = (chkAgree == "on");
   if (validEmail && validFirstName &&
      validLastName && validAgree)
   {
      var tst = CacheRepo.AddUser(
         HttpContext.Cache, emailAddr,
         firstName, lastName);
      if (tst["message"].ReadAs<string>() ==
         "Email already exists")
      {
         return View("Confirmation", false);             #1
      }
      else
      {
         return View("Confirmation", true);              #2
      }
   }
   else
   {
      throw new Exception("Business Rule Failure");      #3
   }
}
```

#1 If the email was duplicated in the repository, the confirmation view will be passed a model value of false.

#2 If everything worked, the model view will receive a true value.

#3 If business rules are invalid, the repository will never be called and an exception will be returned from the controller.

Note that in this scenario, we do not return any information about business rule failures. That is because while we expect our form to be the only thing to ever post to this controller, we cannot guarantee it. If we wanted to handle business rules more gracefully on the server, we would respond with more information or a JSON object containing exception information.

Adding the view

Our controller is performing all the business rule validations and sending necessary information on to the repository for storage but the result, the Confirmation view does not yet exist to resolve this at any position in the `SignupNow` method of the Home controller, right-click and select Add View. Set it up similar to what you see in figure 1 and your new view should be ready to go.

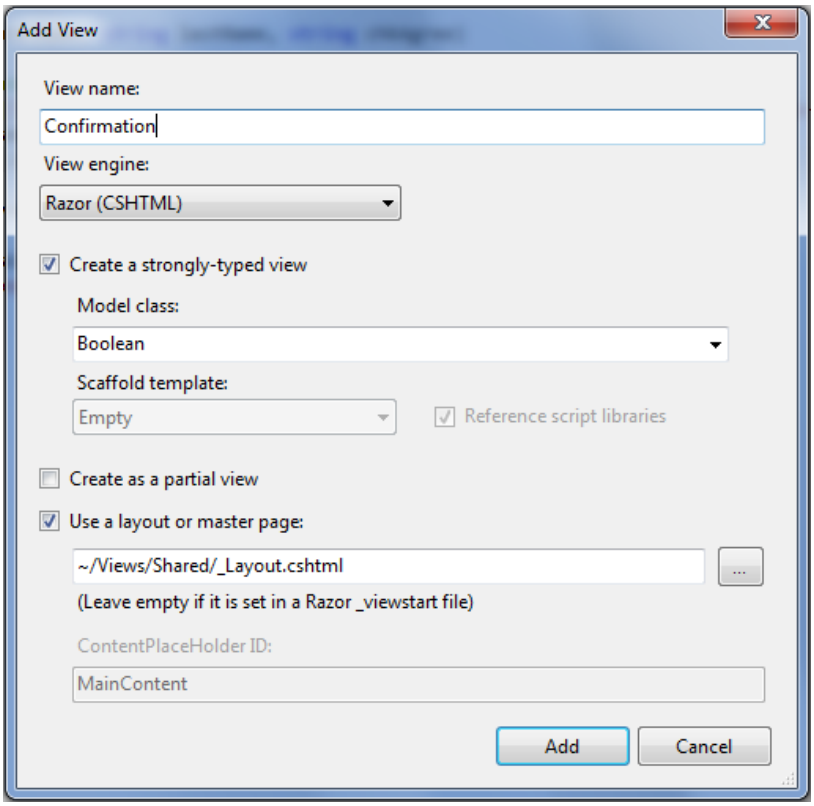


Figure 1 Adding a confirmation view to the SignupNow controller action

The resulting view will have almost nothing in it so we will build it out a bit using HTML and Razor to provide different messages based on the model's value. We will also add a little script that will be useful later for running our web services.

Listing 3 The Confirmation view based on a boolean model

```
@model Boolean
@{
   ViewBag.Title = "Confirmation";
   Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Confirmation</h2>
@{ if (Model)                                              #1
   {
      <p>Thanks for signing up!</p>
   }
   else
   {
      <p>The email address already exists in our database.       #2
         Please try again.</p>
   }
}
<div class="waiting">
   <img
      src="@Url.Content("~/Content/waiting.gif")"
      alt="Please stand by" />
</div>
<button id="getAllUsers">Get All Users</button>
<script
   src="@Url.Content("~/Scripts/htm5Net.js")"
   type="text/javascript" ></script>
<script type="text/javascript">
   $(document).ready(function () {
```

```
        _app.initConfirm();
    });
</script>
```

#1 A true value indicates success and the user gets a thank you message.

#2 The only reason for a false value in our scenario is a duplicate email address so this message is set up appropriately.

Responding to errors without throwing exceptions

Returning enough information to the client to let them know everything went right or wrong is important but normal security operations dictate that too much data can be as damaging to the user experience as not enough data. To that end, we opted to just describe to the user in the view that their attempt was successful or not based on the controller model's value of true or false.

```
if (tst["message"].ReadAs<string>() == "Email already exists")
{
    return View("Confirmation", false);
}
else
{
    return View("Confirmation", true);
}
```

We could also have assigned header data, passed a custom JSON object or not notified the user of any error at all. Each case is different and should be handled based on application requirements.

Validating data entry using jQuery

In order to get our data to validate properly on the client using jQuery, we will be using the jQuery validation library and we will get it from the Microsoft CDN. Remember that the _Layout.cshtml file is used as the master page for our entire site so open that file remove the existing script tag that references the local copy of jQuery and replace it with the following two tags:

```
<script type="text/javascript"
    src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.js" ></script>
<script type="text/javascript"
    src="http://ajax.aspnetcdn.com/ajax/jquery
        .validate/1.8.1/jquery.validate.js" ></script>
```

Be sure to use the latest version of jQuery available and in every page that uses _Layout.cshtml as its master page, both jQuery and jQuery validation will be available.

jQuery Validation basics

In order to validate business rules on the client, jQuery needs to know which elements to validate and what rules to use. The methodology is similar to data annotations on the server in that you can use special rules keywords when building business logic but the similarities end there. In jQuery validation, you create a business rule validation object that has property objects corresponding to the names of elements. Default rules are available for simple validations and you can create your own rules for more complex scenarios. Listing 4 shows how to assert that the `emailAddr` field in the `signup` form will be required and must be a valid email address format. Note that you use a jQuery selector and then call the `validate` method passing in a validation rules object.

Listing 4 Basic jQuery.Validation syntax

```
var validationOptions = {
    rules: {
        emailAddr: {                        #1
            required: true,                 #2
            email: true
        }
    }
}
$("#signup").validate(validationOptions); #3
```

#1 Individual rules objects are nested inside a specifically named rules property which is in turn nested in a validation object.

#2 Basic rule validations can be handled using the built-in functionality or with custom rule functions.

#3 Calling validate() on a form and passing in the validation object will verify the rules against the current form's data.

Believe it or not, that is all you need to do in order to both validate the rules on the client and display error messages when something goes wrong. This is because jQuery.Validate will automatically inject new HTML with exception messages inside as rules are evaluated. Figure 2 shows what happens when all the business rules we are about to implement fail. No additional markup was added manually. jQuery.Validate did all the work for us!

Figure 2 Business rule validation errors are automatically injected into your markup when rules fail. Messages, message placement, and the styles used to display messages can all be manipulated but the out-of-the-box experience is a suitable starting point.

Building the rules

You can see that there are quite a few rules that we want to build to validate our form. Listing 5 shows the complete listing of these rules. We start out after declaring our validation object by stating that the checkbox element should get its error indicated after the label, and all other elements should get their messages after the data value. This is because the checkbox label comes after the element. Then we go through each element in the form and validate its value.

Listing 5 JavaScript business rules, custom rules and custom error messages

```
_app.validationOptions = {
   errorPlacement: function (error, element) {
      if (element.attr("name") == "chkAgree")
         error.insertAfter("#lblAgree");                    #1
      else
         error.insertAfter(element);                        #2
   },
   rules: {
      emailAddr: {
         required: true,
         email: true
      },
      firstName: {
         required: true,
         minlength: 3,
         maxlength: 80
      },
      lastName: {
         required: true,
         rangelength: [3, 80]                               #3
      },
      chkAgree: {
         terms: true                                        #4
      }
   },
   messages: {
      firstName: {                                          #5
         minlength: "Name is too short",
         maxlength: "Name is too long"
      }
   }
```

```
};
```

#1 We can use errorPlacement and custom placement values to add the exception messages anywhere we like.

#2 While the checkbox label comes after the data element, all the rest of the inputs have a label first so we will place the label after the element throwing the exception.

#3 There are quite a few default validation rules built into the validation library with some having custom attributes and some with simple value types.

#4 We can also add custom rules to validate along with the default rules.

#5 Messages can also be customized on a per-field basis and applied based on the rule name.

If you are following along in Visual Studio, you will have noticed that there are no email or terms validation rules in the jQuery.Validation library. These are custom rules, appended to the validation library with a custom function. Listing 6 shows how this is done.

Listing 6 Custom rules in jQuery.Validate can coexist or overwrite default rules

```
_app.setupRules = function () {
   $.validator.addMethod(
      "email",                                                    #1
      function (value) {                                          #2
         return /^[a-zA-Z][\w\.-]*[a-zA-Z0-9]@[a-zA-Z0-9]
            [\w\.-]*[a-zA-Z0-9]\.[a-zA-Z]
            [a-zA-Z\.]*[a-zA-Z]$/.test(value);                    #3
      },
      "Please enter a valid email address."                       #4
   );
   $.validator.addMethod(
      "terms",
      function (value, element) {                                 #5
         if (element.checked)
            return true;
         return false;
      },
      "You must agree to terms and conditions to continue."
   );
};
```

#1 Each rule must have a unique name in the validation library or the original rule will be overwritten.

#2 Rules must have at least a value parameter to validate against input data.

#3 Any kind of rule (including regex) can be executed inside the validation function.

#4 Each custom rule must also have a message to display if the validation returns false.

#5 Optionally, a rule can receive an element as an inbound parameter.

Activating validation on the form using the ready handler

Once the rules are in place, all you need is to call the method that appends them to the `validator` object. Then, when all rules are ready, you call the validate method on the form. This does not have to happen on the `submit` action of the form although that is also an option. Performing the initial validation as soon as the screen is ready, though, will allow you to validate rules immediately. The code in listing 7 will continuously validate the form but will not present a message until a field has either been touched or the user attempts to submit it. If the user submits the form, jQuery.Validate will verify the form's data at which time you can check the `valid()` value to see if the form is ready to submit.

Listing 7 The application init function fired from jQuery ready

```
_app.init = function () {
   _app.setupRules();                                             #1
   $("#signup").validate(_app.validationOptions);                 #2
   $("#signup").submit(function () {                              #3
      if ($("#signup").valid()) {                                 #4
```

```
            return true;                                    #5
         }
         else {
            return false;                                   #6
         }
      });
   };
```

#1 First, we initialize all the rules objects for the form.

#2 Next, we bind the rules from step 1 to the form.

#3 We then bind the form's submit method to an anonymous function.

#4 The form.valid() result will be automatically updated as rules are validated during data entry.

#5 Returning true from the form submit will post the form to the server.

#6 Returning false from the form submit will cancel the form's submission.

Stylesheet edits to pretty up the page

It is worth a quick look at the styles we used to create our custom interface values. Each listed style will be useful for a different area of our application to add pop and, in circumstances like the wait screen, help the user understand that progress is being made.

```
label { width: 100px; display: inline-block; }

label#lblAgree { width: 235px; }

label.error { color: Maroon; font-style: italic; margin-left: 25px; width: 350px; }

div.waiting { background-color: Gray; width: 100%; height: 100%; opacity: 0.95;

   position: absolute; top: 0px; left: 0px; z-index: 100; display: block;

   text-align: center; }

div.waiting img { position: absolute; left: 300px; top: 100px;

fieldset div { margin-top: 10px; }

div.showName { background-color: Gray; color: White;

   border: 1px, solid, Maroon; margin-left: 20px; }
```

Confirmation view ready handler to show progress

Our final touch is to make the confirmation screen go dim for a second after submit the form. At the same time, we will display a waiting icon in the center of the screen. This is an effect that, while contrived and unnecessary for this screen, gives you a quick peek at what can be done in other situations when you may need the user to wait for longer periods. A spinning icon is annoying if you show it for 10 seconds, but it just might buy you three seconds in a pinch!

Listing 8 Ready handler to show processing on page load

```
_app.initConfirm = function () {
   $("div.waiting img").css("top",
      ($(document.body).height() / 2) + "px");
   $("div.waiting img").css("left",
      ($(document.body).width() / 2) + "px");
   setTimeout(function () {
      $("div.waiting").toggle("slow");
   }, 1000);
};
```

You will notice that we are manually centering the `<div>` element and then calling `toggle` on it. `toggle` is a built-in jQuery effect that will show or hide an element.

Summary

We discussed some general highlights of how you can integrate your HTML application's JavaScript code and semantic markup using the MVC framework.

Here are some other Manning titles you might be interested in:

Quick & Easy HTML5 and CSS3
Rob Crowther

Sass and Compass in Action
Wynn Netherland, Nathan Weizenbaum, and Chris Eppstein

Secrets of the JavaScript Ninja
John Resig and Bear Bibeault

Last updated: December 2, 2011