**[Asynchronous Form in ASP.NET MVC](http://tpeczek.blogspot.co.uk/2010/01/asynchronous-form-in-aspnet-mvc.html)**

Author:Tomasz Pęczek on Monday, January 18, 2010 / Tags:[ajax](http://tpeczek.blogspot.co.uk/search/label/ajax), [antiforgerytoken](http://tpeczek.blogspot.co.uk/search/label/antiforgerytoken), [asp.net mvc](http://tpeczek.blogspot.co.uk/search/label/asp.net%20mvc), [jquery validation](http://tpeczek.blogspot.co.uk/search/label/jquery%20validation)

This time I will create an asynchronous *form* with validation (using [jQuery Validation plugin](http://bassistance.de/jquery-plugins/jquery-plugin-validation/) for client-side) and AntiForgeryToken. I haven't dig into new ASP.NET MVC 2 model validation features yet, so the sample will not use any of them (when I take a closer look at them I might write a new version of this post). You should also know, that this solution doesn't perfectly follow DRY principle - we will have to write validation rules twice (this is where ASP.NET MVC 2 might make the biggest difference).  
We will start by creating the form:

<div id="dvRegister">  
<% using (Html.BeginForm("AsynchronousForm", "Home", FormMethod.Post, new { id = "frmRegister" })) { %>  
<%= Html.AntiForgeryToken() %>  
<div>  
<fieldset>  
<legend>Account Information</legend>  
<p>  
<label for="userName">Username:</label>  
<%= Html.TextBox("UserName") %>  
</p>  
<p>  
<label for="email">Email:</label>  
<%= Html.TextBox("Email") %>  
</p>  
<p>  
<label for="password">Password:</label>  
<%= Html.Password("Password") %>  
</p>  
<p>  
<label for="confirmPassword">Confirm password:</label>  
<%= Html.Password("ConfirmPassword") %>  
</p>  
<p>  
<input type="submit" value="Register" />  
</p>  
</fieldset>  
</div>  
<% } %>  
</div>

Someone may ask why I'm not using *Ajax.BeginForm*. That's because it doesn't work out-of-the-box with jQuery validation. We would have to add some code, so the plugin can prevent form from submitting. Instead of that we will handle submitting within validation plugin and avoid using two javascript libraries:

<script src="<%= Url.Content("~/Scripts/jquery-1.3.2.min.js") %>" type="text/javascript"></script>  
<script src="<%= Url.Content("~/Scripts/jquery.validate.min.js") %>" type="text/javascript"></script>  
<script type="text/javascript">  
$(document).ready(function() {  
$("#frmRegister").validate({  
rules: {  
UserName: {  
required: true,  
minlength: 5,  
remote: '<%=Url.Action("ValidateUserName", "Home") %>'  
},  
Email: {  
required: true,  
email: true  
},  
Password: {  
required: true,  
remote: '<%=Url.Action("ValidatePassword", "Home") %>'  
},  
ConfirmPassword: {  
required: true,  
equalTo: "#Password"  
}  
},  
messages: {  
UserName: {  
required: "Please enter username",  
minlength: $.format("Please enter at least {0} characters"),  
remote: $.format("{0} is already in use")  
},  
Email: {  
required: "Please enter email address",  
email: "Please enter valid email address"  
},  
Password: {  
required: "Please enter password"  
},  
ConfirmPassword: {  
required: "Please repeat password",  
equalTo: "Please enter the same password as above"  
}  
},  
submitHandler: function() {  
var registerData = $("#frmRegister").serialize();  
$.ajax({  
type: 'POST',  
url: '<%=Url.Action("AsynchronousForm", "Home") %>',  
dataType: 'json',  
data: registerData,  
success: function(registerResult) {  
if (registerResult.Success) {  
$('#dvRegister').empty().text('User successfully registered.');  
}  
else {  
var errorsContainer = $('#registerErrors');  
if (errorsContainer.length > 0) {  
errorsContainer.empty();  
}  
else {  
$('#frmRegister').after('<ul id="registerErrors" class="validation-summary-errors"></ul>');  
errorsContainer = $('#registerErrors');  
}  
for (error in registerResult.Errors) {  
errorsContainer.append('<li>' + registerResult.Errors[error] + '</li>');  
}  
}  
}  
});  
}  
});  
});  
</script>

Let's take a look at *validate* options.  
First we define validation *rules* for our inputs. The complete list of available rules can be found [here](http://docs.jquery.com/Plugins/Validation#List_of_built-in_Validation_methods). All rules are pretty obvious.  
Next we set error messages. Two things are worth mentioning here:  
- in some cases you can use additional parameters to create your message (like for *remote* and *minlength* in our sample)  
- for *remote* validators, you can return message from server (we will do that for password)  
Last thing we provide is *submitHandler*. This is where we make our form asynchronous. To achieve this, we will use [jQuery.ajax](http://docs.jquery.com/Ajax/jQuery.ajax). First we serialize our form. Then we set parameters for our request (*type*, *url*, *dataType*, *data*) and function which will be called when the request succeed. We should use this function for providing some feedback to user. In our sample we do one of two things. If operation was successful, the form is cleared and user gets success message. If there was an error on the server, we create the same markup as *Html.ValidationSummary* would.  
Now we should add some controller actions:

/// <summary>  
/// Handles the initial GET request  
/// </summary>  
/// <returns>AsynchronousForm view</returns>  
public ViewResult AsynchronousForm()  
{  
return View();  
}  
  
/// <summary>  
/// Handles the asynchronous POST request  
/// </summary>  
/// <param name="user">The user data from the form</param>  
/// <returns>SubmitResult</returns>  
[AcceptVerbs(HttpVerbs.Post)]  
[ValidateAntiForgeryToken]  
public JsonResult AsynchronousForm(UserRegisterData user)  
{  
SubmitResult result = new SubmitResult() { Success = true };  
  
List<string> validationErrors = user.Validate(\_usersRepository);  
if (validationErrors.Count == 0)  
{  
try  
{  
\_usersRepository.RegisterUser(user);  
}  
catch (Exception ex)  
{  
result.Success = false;  
result.Errors = new string[] { ex.Message };  
}  
}  
else  
{  
result.Success = false;  
result.Errors = validationErrors.ToArray();  
}  
return Json(result);  
}  
  
/// <summary>  
/// Validates username  
/// </summary>  
/// <param name="UserName">The username</param>  
/// <returns>true or false</returns>  
public JsonResult ValidateUserName(string UserName)  
{  
return Json(\_usersRepository.ValidateUserNameUnique(UserName));  
}  
  
/// <summary>  
/// Validates password  
/// </summary>  
/// <param name="Password">The password</param>  
/// <returns>true or error message</returns>  
public JsonResult ValidatePassword(string Password)  
{  
string passwordInvalidMessage = \_usersRepository.ValidatePassword(Password);  
if (String.IsNullOrEmpty(passwordInvalidMessage))  
return Json(true);  
else  
return Json(passwordInvalidMessage);  
}

Action for initial GET request is very simple. Validation actions aren't too complicated either. You should only notice, that *ValidatePassword* action returns *true* if password is valid, or *error message* if not. The real place of interest is action for our asynchronous POST request. ASP.NET MVC will put our post data into *UserRegisterData* object:

public class UserRegisterData  
{  
public string UserName { get; set; }  
public string Email { get; set; }  
public string Password { get; set; }  
public string ConfirmPassword { get; set; }  
  
public List<string> Validate(IUsersRepository repository)  
{  
List<string> validationErrors = new List<string>();  
  
if (String.IsNullOrEmpty(UserName) || UserName.Trim().Length == 0)  
validationErrors.Add("Username is required");  
else if (UserName.Length < 5)  
validationErrors.Add("Username should have at least 5 characters");  
else if (!repository.ValidateUserNameUnique(UserName))  
validationErrors.Add(String.Format("Username {0} is already in use", UserName));  
  
if (String.IsNullOrEmpty(Email) || Email.Trim().Length == 0)  
validationErrors.Add("Email is required");  
else if (!Regex.IsMatch(Email, @"^...$"))  
validationErrors.Add("Email is invalid");  
  
if (String.IsNullOrEmpty(Password) || Password.Trim().Length == 0)  
validationErrors.Add("Password is required");  
else  
{  
string passwordInvalidMessage = repository.ValidatePassword(Password);  
if (!String.IsNullOrEmpty(passwordInvalidMessage))  
validationErrors.Add(passwordInvalidMessage);  
}  
  
if (String.IsNullOrEmpty(ConfirmPassword) || ConfirmPassword.Trim().Length == 0)  
validationErrors.Add("Confirm password is required");  
else if (!ConfirmPassword.Equals(Password))  
validationErrors.Add("Confirm password does not match password");  
  
return validationErrors;  
}  
}

As you can see, we were forced to repeat our validation logic (you can never trust client-side validation). After validating our data and performing required operation, we send *SubmitResult* object in response (this is the object we are using in our *submitHandler*):

public class SubmitResult  
{  
/// <summary>  
/// Success flag  
/// </summary>  
public bool Success { get; set; }  
  
/// <summary>  
/// Errors array  
/// </summary>  
public string[] Errors { get; set; }  
}