Form Builder - Developer Guide

2017-02-10

Orckestra, Europe
Nygårdsvej 16
DK-2100 Copenhagen
Phone +45 3915 7600

www.orckestra.com

Contents

1 Introduction

This guide is intended for Web and C# developers who want to customize or extend the Form Builder add-on.

With the Form Builder you can quickly build a well-designed, mobile-friendly and accessible web form with validation and semantics like e-mail fields in place by simply using "drag-and-drop" and add it to a page. The data via this form can be saved to files, exported to various formats (XML files, Excel spreadsheets). You can also integrate the data with various back-end systems, for example, a CRM.

For information about using the Form Builder add-on, please see the “Form Builder User Guide”.

The Form Builder is highly customizable and extendible. And in this guide, you’ll learn how to:

- Localize the front-end and back-end
- Work with styling
- Change the existing fields and create custom ones
- Extend the submit actions with your own
- Handle secure data submission

To customize and extend, you should be proficient in C# / .NET and Razor and be able to work with XHTML and CSS / LESS.

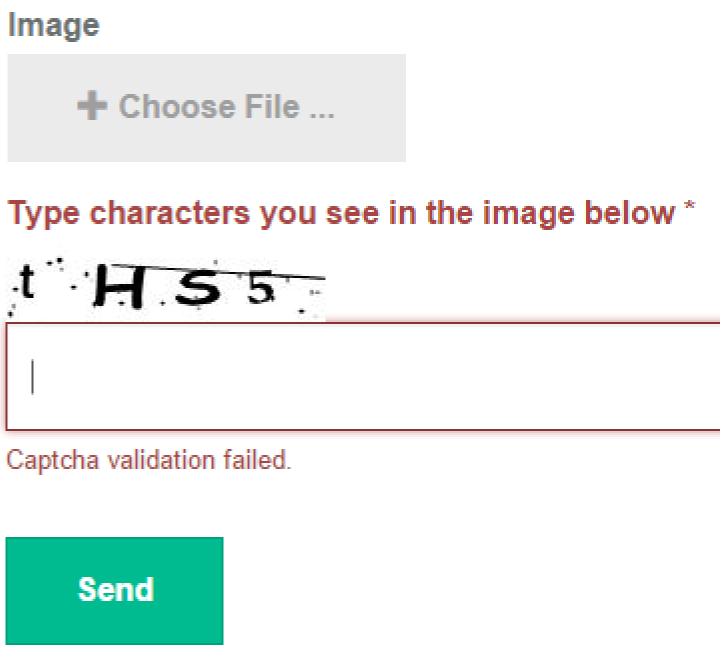
1.1 Before You Start

You should make sure that you have the following prerequisites in place before you start the guide:

- You have installed and configured the latest version of C1 CMS
- You are using a test C1 CMS installation that you can safely experiment with.
- You have installed and configured the latest version of the Form Builder add-on on this test C1 CMS installation instance.

2 Localizing the front-end

The Form Builder uses text strings in English for texts that appear on forms, for example, validation messages.



Captcha validation failed.

Figure 1: CAPTCHA validation message in the default language (English)

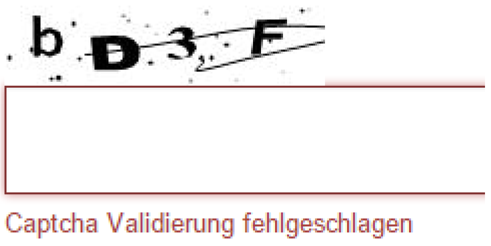
You can find these strings in ~/App_GlobalResources/Composite/Forms/FormBuilder.resx.

If your website's default language is other than English, this is where you can translate the strings into that language.

If your website is multi-lingual, you need to localize these strings for each language of your website. (We assume that the default language of your website is English in this case.)

To localize the strings for other language versions of your website:

1. Make a copy of the FormBuilder.resx in ~/App_GlobalResources/Composite/Forms/.
2. Rename this file so that it matches the website language you want to translate strings into. For this, append the corresponding language culture name to the file name, for example, *FormBuilder.de-de.resx* for German, where "de-de" is the language culture name for German.
3. Open the file in Visual Studio or other resource file editor and translate the strings.
4. Repeat Steps 1-3 for as many languages as you use on your website.



Captcha Validierung fehlgeschlagen

Figure 2: Localized CAPTCHA validation message

If you are interested in localizing the Form Builder GUI in the CMS Console, see "How to localize the admin UI".

3 Working with styling

The Form Builder is Bootstrap 3 ready out-of-the-box and requires LESS to work properly.

The forms created with the Form Builder use styles of the Bootstrap 3 based starter sites such as “Open Cph” or “Tiny Cph”.

These styles are defined in ~/Frontend/Styles/style.less and other LESS files, which come with the starter site, and applied across the website on all pages with, and without forms.

Please note that if you are using the Form Builder on a website other than a C1 CMS starter site, you may need to edit:

- ~/Frontend/Composite/Forms/FormBuilder/FormBuilderPreview.less

Here, you should set a correct @import directive because the default reference to "../../../styles/style.less" may not reflect your situation.

The styles specific to the Form Builder are defined in:

- ~/Frontend/Composite/Forms/FormBuilder/Composite.Forms.FormBuilder.less

This style sheet also includes the minimum set of styles for the Form Builder to work on sites not based on the C1 CMS Starter sites.

4 Changing the HTML or behavior of existing fields

Each form field and other form component are available in the Component Library and the Text & Decorations lists.

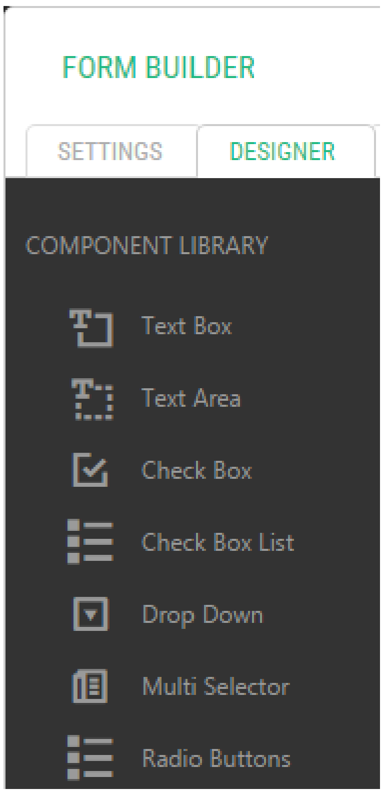
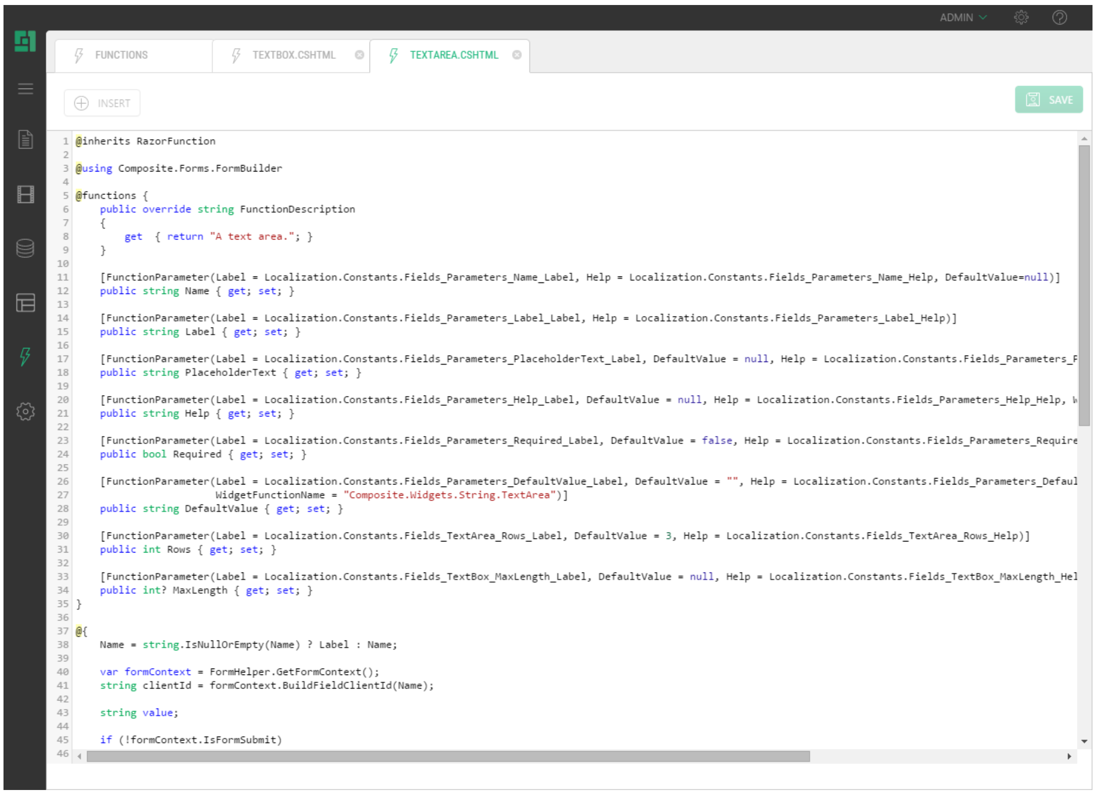


Figure 3: Form fields in the from designer

These are regular Razor functions and available from the Function perspective as:

- Composite.Forms.FormBuilder.Fields.*
- Composite.Forms.FormBuilder.Fields.Decoration.*



```
@inherits RazorFunction

@using Composite.Forms.FormBuilder

@functions {
    public override string FunctionDescription
    {
        get  { return "A text area."; }
    }

    [FunctionParameter(Label = Localization.Constants.Fields_Parameters_Name_Label, Help = Localization.Constants.Fields_Parameters_Name_Help, DefaultValue=null)]
    public string Name { get; set; }

    [FunctionParameter(Label = Localization.Constants.Fields_Parameters_Label_Label, Help = Localization.Constants.Fields_Parameters_Label_Help)]
    public string Label { get; set; }

    [FunctionParameter(Label = Localization.Constants.Fields_Parameters_PlaceholderText_Label, DefaultValue = null, Help = Localization.Constants.Fields_Parameters_F
    public string PlaceholderText { get; set; }

    [FunctionParameter(Label = Localization.Constants.Fields_Parameters_Help_Label, DefaultValue = null, Help = Localization.Constants.Fields_Parameters_Help_Help, W
    public string Help { get; set; }

    [FunctionParameter(Label = Localization.Constants.Fields_Parameters_Required_Label, DefaultValue = false, Help = Localization.Constants.Fields_Parameters_Require
    public bool Required { get; set; }

    [FunctionParameter(Label = Localization.Constants.Fields_Parameters_DefaultValue_Label, DefaultValue = "", Help = Localization.Constants.Fields_Parameters_Defaul
                       WidgetFunctionName = "Composite.Widgets.String.TextArea")]
    public string DefaultValue { get; set; }

    [FunctionParameter(Label = Localization.Constants.Fields_TextArea_Rows_Label, DefaultValue = 3, Help = Localization.Constants.Fields_TextArea_Rows_Help)]
    public int Rows { get; set; }

    [FunctionParameter(Label = Localization.Constants.Fields_TextBox_MaxLength_Label, DefaultValue = null, Help = Localization.Constants.Fields_TextBox_MaxLength_Hel
    public int? MaxLength { get; set; }
}

@{
    Name = string.IsNullOrEmpty(Name) ? Label : Name;

    var formContext = FormHelper.GetFormContext();
    string clientId = formContext.BuildFieldClientId(Name);

    string value;

    if (!formContext.IsFormSubmit)
```

Figure 4: Form fields as Razor functions

By editing these Razor functions, you can change the way the existing fields look and behave:

1. Form the "Functions" perspective, expand "Razor Functions", then Composite.Forms.FormBuilder.Fields, or Composite.Forms.FormBuilder.Fields.Decoration
2. Edit a function that stands for the field you want to change.
3. Edit the markup and / or code up to your requirements
4. Save the function.

Note please note that the appearance and behavior of the field will change on each existing form.

If you want to edit these Razor function in your preferred code editor, for example, Visual Studio, find the corresponding .cshtml files here:

- ~/App_Data/Razor/Composite/Forms/FormBuilder/Fields/
- ~/App_Data/Razor/Composite/Forms/FormBuilder/Fields/Decoration/

5 How to create a new field type

Each field available in the Form Builder in the Component Library or Text & Decorations are Razor functions.

To create a custom field for the Form Builder, you need to:

1. Create a Razor function that will render the filed on the form.
2. Then register it in the Form Builder's configuration file. See "The configuration file".

5.1 Creating a custom field

To create a sample field:

1. From the "Functions" perspective, add a new Razor function. Alternatively, you can create a Razor function in Visual Studio 2012. (Please see "Creating Razor Functions in Visual Studio").
2. Add necessary code and markup.
3. Save the function.
4. Now edit *~/App_Data/Composite/Configuration/Composite.Forms.FormBuilder.xml*
5. Add a field within one of the field groups, setting the attributes where needed and specifying the function created in Steps 1-3 for the field.
6. Save the file.

Please see the following samples for illustration of a function's code and markup and a field registration settings in Composite.Forms.FormBuilder.xml.

5.2 A simple field

The simplest example of the field is the existing Separator field that comes with the Form Builder.

```
@inherits RazorFunction
@using Composite.Forms.FormBuilder;
@functions {
	public override string FunctionDescription
	{
		get { return "A visual separator"; }
	}
}
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
</head>
<body>
	<hr />
</body>
</html>
```

Listing 1: Composite.Forms.FormBuilder.Fields.Decoration.Separator

As you can see in the code above, along with the regular Razor Function's code (the inherit directive, description and the markup, the Razor function that stands for a field in the Form Builder should at minimum:

- Include a using directive that references "Composite.Forms.FormBuilder".

If the field has properties they should be declared as input parameters in this Razor function (see the following sample for illustration).

It is registered in the Form's Builder's configuration file like this:

C1 CMS

```
<FormBuilder xmlns:f="http://www.composite.net/ns/function/1.0">
  <Fields>
    ...
    <FieldGroup 
label="${Composite.Forms.FormBuilder,FieldGroup.TextAndDecorations}">
         ...
         <Field name="decoration-separator" 
label="${Composite.Forms.FormBuilder,Fields.Decoration.Separator}" 
icon="icon-minus">
        <f:function 
name="Composite.Forms.FormBuilder.Fields.Decoration.Separator" />
      </Field>
         ...
    </FieldGroup>
  </Fields>
  ...
</FormBuilder>
```

Listing 2: The Separator registered in the configuration

In the configuration above, you can see that:

The field is declared with the Field element, which is place within a FieldGroup element (“Text and Decorations” in this example)

It uses the name attribute to be uniquely identified in the system.

It also uses the label and icon attributes to specify its corresponding elements in the Form Builder GUI.

The Field element nests the Razor function (here, Composite.Forms.FormBuilder.Fields.Decoration.Separator) that renders the field on a form.

The function may include no, one or more parameters.

5.3 A field with properties and validation

In the following sample, a text box is created, which only accepts numbers between 1 and 10 as the value. If the user enters anything else, a validation message is displayed.

The field comes with two properties: Label (used on the frontend) and Name (used programmatically).

```
@inherits RazorFunction
@using Composite.Forms.FormBuilder
@functions {
    [FunctionParameter(Label =
Localization.Constants.Fields_Parameters_Name_Label, Help =
Localization.Constants.Fields_Parameters_Name_Help)]
    public string Name { get; set; }
    [FunctionParameter(Label =
Localization.Constants.Fields_Parameters_Label_Label, DefaultValue = null,
Help = Localization.Constants.Fields_Parameters_Label_Help)]
    public string Label { get; set; }
}
@{
    string label = string.IsNullOrEmpty(Label) ? Name : Label;
    var formContext = FormHelper.GetFormContext();
    string value = "";
    if (formContext.IsFormSubmit)
    {
        value = Request.Unvalidated[Name];
        if (!string.IsNullOrEmpty(value))
        {
            int number;
            if (!int.TryParse(value, out number))
            {
                formContext.Errors.Raise(string.Format("Field '{0}' should
contain an integer value.", value));
            }
            else if (number < 0 || number > 10)
            {
                formContext.Errors.Raise(string.Format("Field '{0}' should
contain an value from [1, 10] range.", value));
            }
            else
            {
                formContext.Result.Add(Name, label, typeof(int),
int.Parse(value));
            }
        }
    }
}
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
</head>
<body>
    <div class="form-group">
        <label for="form@(Name)" class="control-label">@label</label>
        <input type="number" class="form-control" id="form@(Name)"
name="@Name"  value="@value" min="1" max="10" />
    </div>
</body>
</html>
```

Listing 3: A custom field as a Razor function

In the example above, you can see that:

- 2 properties are declared: “Label” and “Name”.
- The label can be set explicitly via the Label property. Otherwise, it will be copied form the Name property’s value.
- The validation logic for controlling the range of the numbers in the value is implemented that prevents users from entering invalid numbers or illegal characters.
- Two HTML elements - <label/> and <input/> of the type “number” - are used to display the field on the form.
- The field is nested within the <div> element of the class “form-group” and each HTML element has its own class applied to: “control-label” and “form-control”

This field should be registered in the Form Builder’s configuration file:

```
<FormBuilder xmlns:f="http://www.composite.net/ns/function/1.0">
  <Fields>
    <FieldGroup label="${Composite.Forms.FormBuilder,FieldGroup.Standard}">
         ...
      <Field name="onetoten" label="One To Ten" icon="icon-textbox">
        <f:function name="Composite.Forms.FormBuilder.Fields.OneToTen">
        </f:function>
      </Field>
         </FieldGroup>
  </Fields>
  ...
</FormBuilder>
```

Listing 4: A custom field registered in the configuration

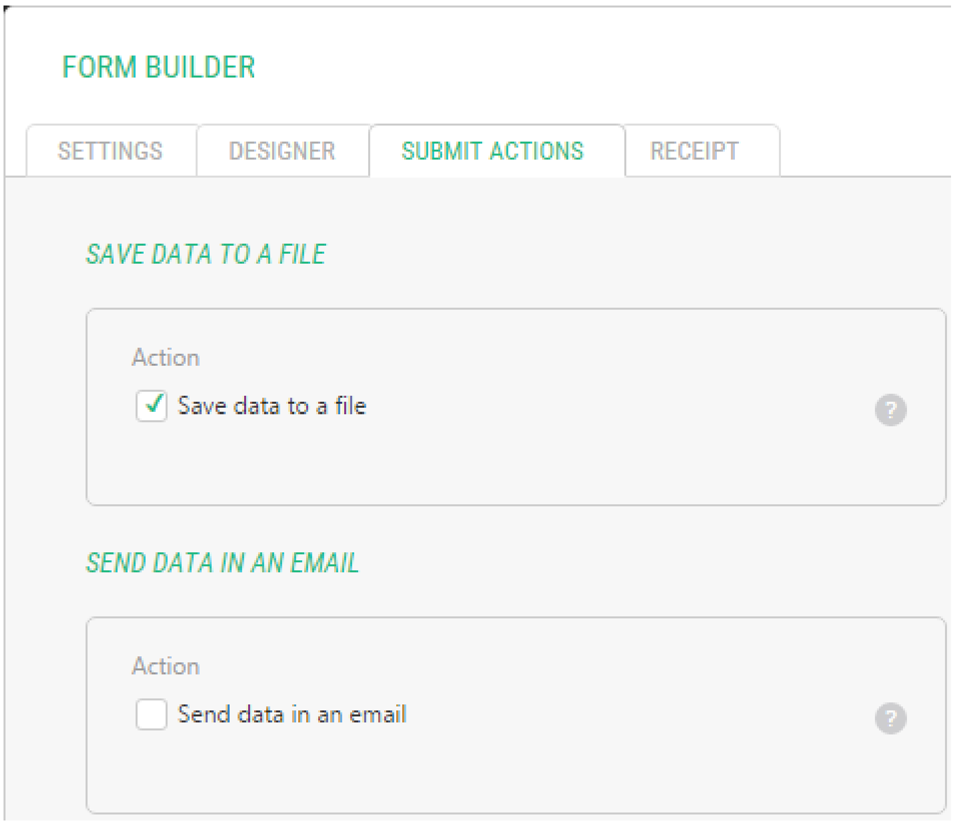
6 How to create a new submit handler

By default, when the user submits a form, the submitted data are saved in an XML file.

This is handled by a submit handler “SaveToFile” hooked in to the submit event and enabled out-of-the-box. Another submit handler you can enable in a form is “SendEmail” that sends the submitted data to a specified email address.

Each submit handler is a Razor function in the “Composite.Forms.FormBuilder.Handler” namespace. Normally, these functions make use of the FormBuilder’s GetFormData method to access the data submitted via the form.

Each submit handler must be registered in the Composite.Forms.FormBuilder.xml configuration file to appear in the Form Builder on the Submit Action tab and enabled on this tab to execute on the submit event.



☑ Save data to a file

☐ Send data in an email

Figure 5: Default submit actions

6.1 Creating a custom submit handler

To create a Razor function that will serve as a submit handler:

1. From the “Functions” perspective, add a new Razor function using, for example, “Composite.Forms.FormBuilder.Handler” for its namespace. Alternatively, you can create a Razor function in Visual Studio 2012. (Please see “Creating Razor Functions in Visual Studio”).
2. Add necessary code and markup. To access the submitted data use the Composite.Forms.FormBuilder.FormHelper.GetFormData method
3. Save the function.

For illustration, you can check “SaveToFile”, “SendEmail” and “ListValues” functions or the sample that follows this section.

To register the submit handler in the FormBuilder’s configuration file:

1. Edit ~/App_Data/Composite/Configuration/Composite.Forms.FormBuilder.xml.
2. Within the **<Actions/>** element, add a new **<Action/>** element.
3. Set its attributes such as name, label and help.
4. Within this **<Action/>** element, specify the function created earlier.
5. Save the file.

To enable the submit handler on the form:

1. Edit the form you want to enable another submit action on.
2. On the “Submit Actions” tab, select the respective action.
3. Fill the related fields if necessary. (Enabling some actions may display additional fields to fill.)
4. Save the form.

6.2 A custom submit handler sample

In the following example, we’ll created a submit handler that will save the submitted data in the CSV (comma-separated values) file and register it in the FormBuilder’s configuration file.

1. From the “Functions” perspective, add a new Razor function named “Composite.Forms.FormBuilder.Handler.SaveAsCsv”.
2. Replace the content with the code below.
3. Save the function.

```
@inherits RazorFunction
@using Composite.Forms.FormBuilder
@functions {
    public override string FunctionDescription
    {
        get  { return "Saves submitted values into the specified data
type."; }
    }
}
@{
    // get form data
    var formData = FormHelper.GetFormData();
    if (formData == null)
    {
        return;
    }
    // ensure a valid path to the CSV file
    string dirPath =
HttpContext.Current.Server.MapPath("~/App_Data/FormData");
    if (!Directory.Exists(dirPath))
    {
        Directory.CreateDirectory(dirPath);
    }
    string filePath = Path.Combine(dirPath, formData.FormName + ".csv");
    // add the CVS header on the first submit
    string csvData = string.Empty;
    if (!File.Exists(filePath))
    {
        csvData +=  string.Format("{0}\r\n", string.Join(",",
formData.Fields.Select(d=>d.Label)));
    }
    // add the form data
    csvData += string.Format("{0}\r\n", string.Join(",",
formData.Fields.Select(d => d.Value)));
    // save the data to the CSV file
    StreamWriter file = new StreamWriter(filePath, true);
    file.Write(csvData);
    file.Close();
}
```

Listing 5: Custom submit handler as a Razor function

In the above sample, we:

1. Get the form data with the **GetFormData** method
2. Build a path to the file from the pre-defined directory and the from name as the filename
3. Add the header if this is the first submit
4. Add the form data
5. Save the resulting string to the CSV file

Now that we created a custom submit handler, we need to register it in the Form Builder's configuration file:

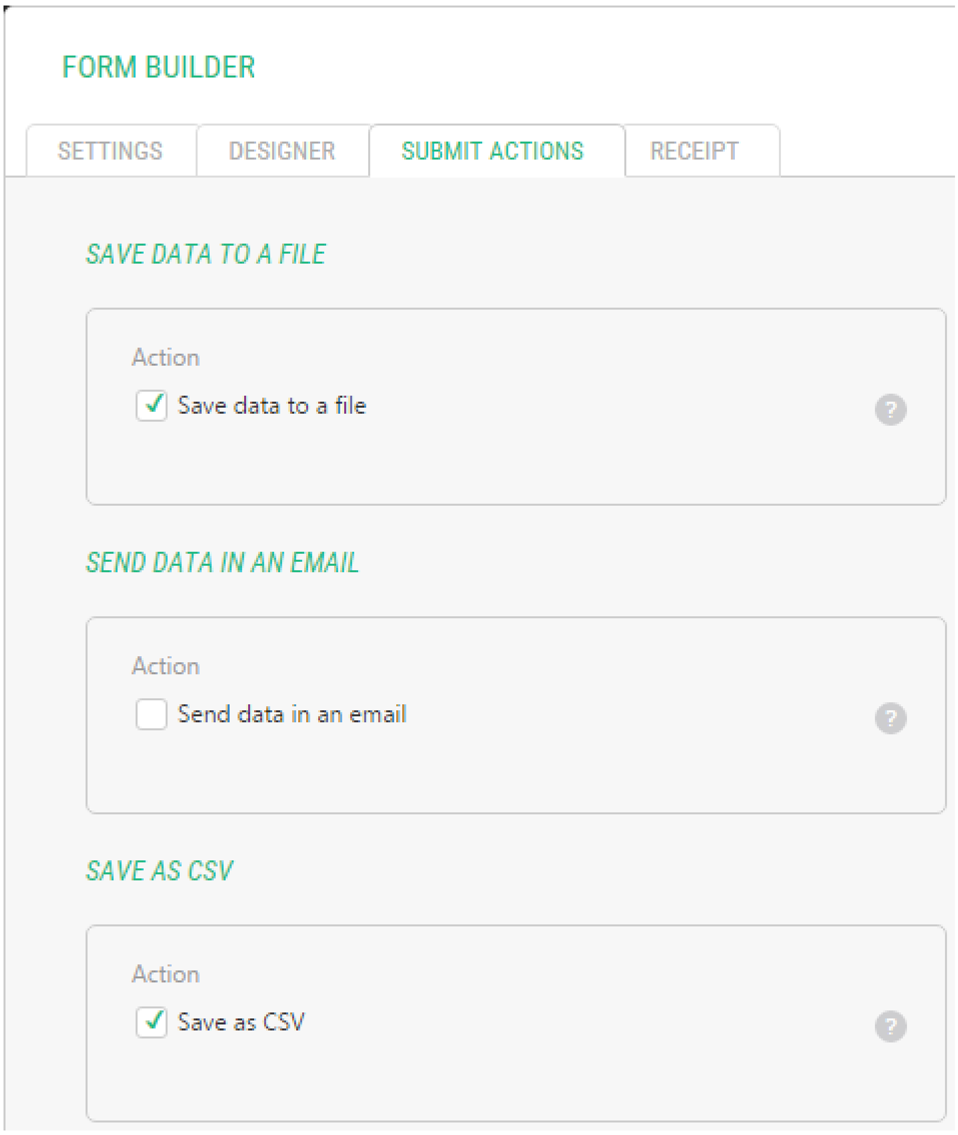
1. Edit ~/App_Data/Composite/Configuration/Composite.Forms.FormBuilder.xml.
2. Following the latest **<Action/>** element, add the markup below.
3. Save the file.

```
  <Actions>
       <!-- other Action elements -->
    <Action name="SaveAsCsv" label="Save as CSV" help="Saves form data to a
CSV file">
      <f:function name="Composite.Forms.FormBuilder.Handlers.SaveAsCsv">
      </f:function>
    </Action>
  </Actions>
```

Listing 6: Custom submit handler registered in the configuration

Finally, you should enable the new submit handler on your form.

1. Edit the form you want to enable the “Save as CSV” submit action on.
2. On the “Submit Actions” tab, select the “Save as CSV”.
3. Save the form.



SAVE DATA TO A FILE

Action

[✓] Save data to a file

SEND DATA IN AN EMAIL

Action

[ ] Send data in an email

SAVE AS CSV

Action

[✓] Save as CSV

Figure 6: Custom submit action in the Form Builder

When the user submits the form, the submitted data will be now added to a CSV file (named after the form) at ~/App_Data/FormData.

7 How to create a new security handler

To ensure data submissions via secure connections, you can create a security handler and use it on forms built with the Form Builder.

A security handler is a Razor Function that handles the security connection.

The Form Builder comes with its own security handler, available as the Razor function “Composite.Forms.FormBuilder.Handlers.HttpsCheck”.

It is disabled by default in the Form Handler’s configuration file.

You can use this security handler for secure data submissions if your website has HTTPS support.

The general procedure for creating and using a security handler is as follows:

1. Make sure your website has the HTTPS support.
2. Create a Razor function that will serve as a security handler, i.e. handle the secure form submission.
3. Register the security handler in the Form Handler’s configuration file.
4. Select the secure connection in the form’s settings.

7.1 Configuring HTTPS

Before using a security handler on forms, make sure:

1. The Certificate Server is installed on the target server with IIS.
2. A proper certificate is installed on the server. You might need to request the certificate with a certificate issuing company such as VeriSign.
3. SSL is configured on IIS for your website.

For testing purposes, you can generate your own certificate by using ad-hoc utilities such as openssl or SelfSSL. For example, see “Enabling SSL on IIS 7.0 Using Self-Signed Certificates”.

7.2 Creating and a security handler

To create a security handler:

1. From the Functions perspective, add a new Razor function using, for example, “Composite.Forms.FormBuilder.Handler” for its namespace. Alternatively, you can create a Razor function in Visual Studio 2012. (Please see “Creating Razor Functions in Visual Studio”).
2. Add necessary code and markup.
3. Save the function.

For illustration, you can use the security handler that comes with the Form Builder “Composite.Forms.FormBuilder.Handlers.HttpsCheck”.

```
@inherits RazorFunction
@using Composite
@using Composite.Forms.FormBuilder
@functions {
    public override string FunctionDescription
    {
        get  { return "Redirects to https url if http connection is not
already secured."; }
    }
}

@{
    var formContext = FormHelper.GetFormContext();
    if (formContext == null)
    {
        return;
    }
    if (Request.IsSecureConnection || Data.CurrentPublicationScope ==
PublicationScope.Unpublished)
    {
        return;
    }
    formContext.FailPrerequisites();
    var url = Request.Url.AbsoluteUri;
    Verify.That(url.StartsWith("http://",
StringComparison.OrdinalIgnoreCase), "Incorrect schema");
    string secureUrl = "https" + url.Substring(4);
    Response.Redirect(secureUrl, false);
}
```

Listing 7: HttpsCheck security handler as a Razor function

7.3 Registering the security handler

If you choose to use the out-of-the-box security handler of the Form Builder:

1. Edit ~/App_Data/Composite/Configuration/Composite.Forms.FormBuilder.xml.
2. Locate and uncomment the security handler already available there within the **<SecurityHandlers/>** element.
3. Save the file.

```
  <SecurityHandlers>
    <SecurityHandler name="Https"
label="${Composite.Forms.FormBuilder,SecurityHandlers.Https.Label}">
      <Prerequisites>
        <f:function name="Composite.Forms.FormBuilder.Handlers.HttpsCheck">
        </f:function>
      </Prerequisites>
      <Postrequisites>
      </Postrequisites>
    </SecurityHandler>
  </SecurityHandlers>
```

Listing 8: HttpsCheck security handler registered in the configuration

To register your own security handler:

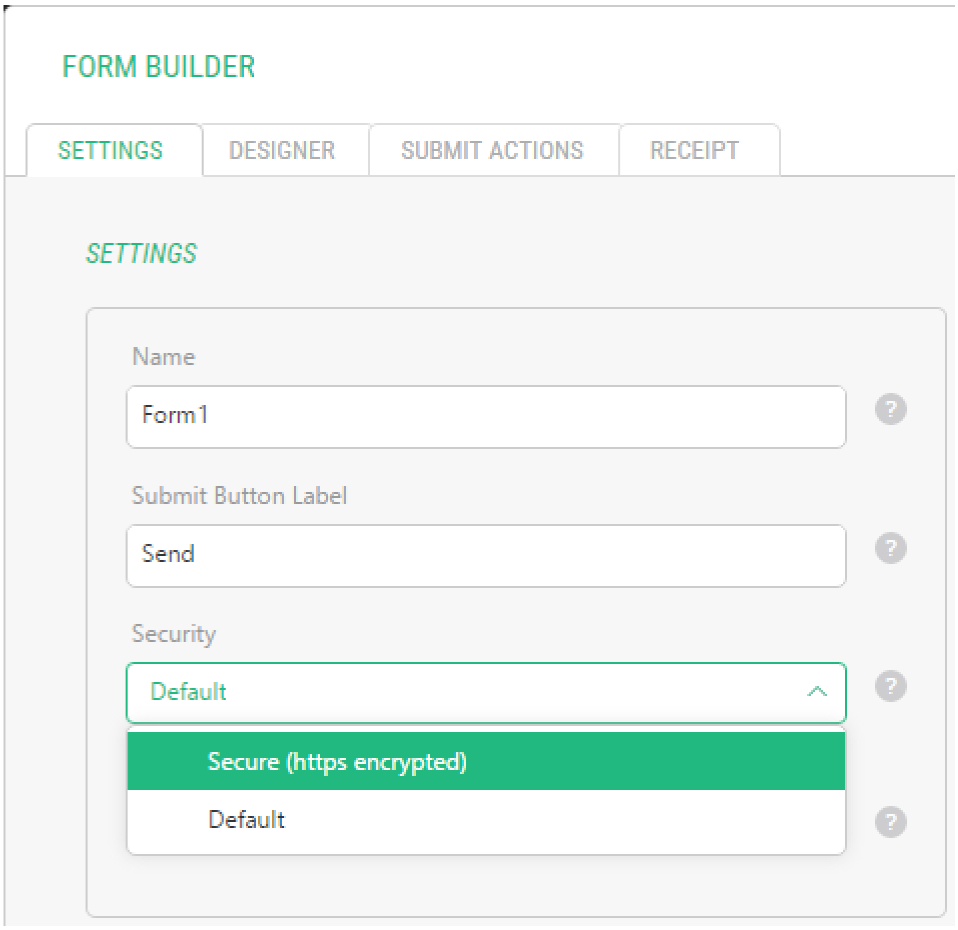
1. Edit ~/App_Data/Composite/Configuration/Composite.Forms.FormBuilder.xml.
2. Within the **<SecurityHandlers/>** element, add a **<SecurityHandler/>** element and specify values in its name and label attributes.
3. Add a **<Prerequisites/>** element and a **<Postrequisites/>** element as child elements of the **<SecurityHandler/>** element.
4. Within the **<Prerequisites/>** element add one or more Razor functions that serve as the security handlers, which must be executed before the form submission.

5. Within the **<Postrequisites/>** element add one or more Razor functions that serve as the security handlers, which must be executed after the form submission.
6. Save the file.

7.4 Using security handlers on forms

To enable a specific security handler on a form:

1. Edit the form you want to enable a security handler on.
2. On the Settings tab, select the security handler in the “Security” drop-down list.
3. Save the form.



Name: Form1

Submit Button Label: Send

Security: Default

- Secure (https encrypted)
- Default

Figure 7: A security handler available in the Form Builder

8 How to localize the admin UI

If your CMS Console GUI is localized and is in a language other than English, you may want to localize the Form Builder, too.

The text strings used in the Form Builder are stored in:

~/Composite/InstalledPackages/localization/Composite.Forms.FormBuilder.en-us.xml

To localize the Form Builder GUI into your language:

1. Make a copy of the Composite.Forms.FormBuilder.en-us.xml in ~/Composite/InstalledPackages/localization/.
2. Rename this file so that it matches the language of your CMS Console GUI. For this, append the corresponding language culture name to the file name, for example, Composite.Forms.FormBuilder.de-de.xml for German, where “de-de” is the language culture name for German.
3. Open the file in your preferred XML editor and translate the strings.

You may need to reload the CMS Console to see the changes.

9 The configuration file

The Form Builder’s configuration file allows you to control what tools are available for you when creating forms with the Form Builder. The file is located on your website at:

- ~/App_Data/Composite/Configuration/Composite.Forms.FormBuilder.xml

The tools available for configuration here are:

- Form fields and other form components (see “How to create a new field type”)
- Submit actions (see “How to create a new submit handler”)
- Secure data submission (see “How to create a new security handler”)
- Form preview styles (see “Form preview styles”)

These are all reflected in the section structure of the configuration file:

```
<FormBuilder xmlns:f="http://www.composite.net/ns/function/1.0">
  <Fields>
       <!-- field groups and fields -->
  </Fields>
  <Actions>
       <!-- submit actions -->
  </Actions>
  <SecurityHandlers>
       <!-- security handlers -->
  </SecurityHandlers>
  <FieldEditor>
    <Styles>
       <!—links to style sheets -->
    </Styles>
  </FieldEditor>
</FormBuilder>
```

Listing 9: Sections in the Form Builder’s configuration file

Each form field or handler you want to use when creating forms must be “registered” in this configuration file. Otherwise, they won’t be available in the Form Builder GUI.

9.1 Form preview styles

The appearance of the fields in the form preview (the middle area on the “Designer” tab) is controlled via the styles defined in stylesheets.

By default, the form preview uses:

You can reference other stylesheets along with or in place of, the default style sheet in the **<FieldEditor/>** section within the **<Styles/>** element.

```
  <FieldEditor>
    <Styles>
      <Style
file="~/Frontend/Composite/Forms/FormBuilder/FormBuilderPreview.less" />
    </Styles>
  </FieldEditor>
```

Listing 10: Stylesheet reference in the configuration