An Advanced Guide to Data Types

2017-02-14

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

www.orckestra.com

Contents

1 Introduction

Custom content types called “data types” here are an essential part of C1 CMS. When using data types, it is possible to keep and access data required for a website in a structured way, similar to storing data in database tables.

If you are new to C1 CMS data types, we recommend that you first read *A Guide to Creating Data Types*, which introduces you to the basics of data types.

This guide focuses on more advanced topics related to data types as well as forms based on these data types.

It includes topics about localizing data types and input forms as well as customizing the data item editor by editing a data type’s form markup.

Besides, it discusses ways of ensuring appropriate values in the input fields with validation rules and lists of predefined values supplied by referenced data types.

Widgets are presented in a greater detail and some practical information is given on grouping data items into tree-like structures.

1.1 Who Should Read This Guide?

This guide is intended for designers and developers who want to learn how to effectively edit and use data types in web design and web development in C1 CMS.

We assume that you know how to work in the Data perspective of C1 CMS and have permissions to create and edit data types. We also assume that you have an idea of structured data such as tables in databases.

Some topics might require that you should know XML and have an idea of .NET resource files.

1.2 Getting Started

Getting started with advanced topics on data types implies that you learn one or more related activities.

Getting Started	
Activity	**Chapter or section**
How to localize data types and their data items	*Localizing Data Types*
How to localize forms to match the language used on your website	*Localizing Forms*
How to use and fine-tune advanced widgets for data type fields	*Using Advanced Widgets*
How to customize the data item forms by editing the form markup	*Editing Form Markup*
How to ensure that the user should enter proper values by using validation rules on fields	*Applying Validation Rules to Fields*
How to ensure that the user should enter proper values by using data types as lists of predefined values	*Creating Data Types That Reference Other Data Types*
How to present data items hierarchically by grouping them by data reference fields	*Grouping by Data Reference Fields*

In the following few chapters, you will learn more about these and other activities.

1.3 Terms and Abbreviations

The following is the list of terms and their definitions used throughout this guide.

Terms and Definitions	
Term	**Definition**
Data type	An entity used to store and reuse structured data in C1 CMS
Validation rule	A criterion that ensures that input data meets the criterion and is thus correct
Form markup	An XML-based representation of the editor form used in C1 CMS to allow adding or editing data items
Widget	In C1 CMS, a control used for getting input values from users and storing them in corresponding data type fields

2 Localizing Data Types

Localization of data types follows the general localization rules in C1 CMS. Localization of forms created with Forms Renderer has its specifics. In the following sections we will cover both types of localization in detail.

Global data types and page datafolders are localized in the same manner; page metatypes are localized as part of localization of pages when used on the latter.

The focus of this guide is global data types; that is why the following procedures will be illustrated with these data types.

Localizing data types in C1 CMS implies localizing the data items these data types contain. If you add an item to a localized data type, you can switch to the other language in C1 CMS, translate the item and use it on its own on the localized website from now on.

The item in the main language and the item in the language you localize to will now co-exist. Hence, changes in the item in one language will not affect the item in the other language.

Before you can localize data items, you should enable localization on data types.

2.1 Enabling Localization on Data Types

To enable localization on data types and localize data items, at least two languages must be added to C1 CMS.

By default, data types created in C1 CMS are not localizable. Enabling localization on existing data types is different from enabling localization on data types being created.

2.1.1 Localizing Existing Data Types

To enable localization on existing data types:

1. Select the data type.
2. On the toolbar, click **Enable Localization**.

Figure 1: Enable Localization button on the toolbar

3. In the **Enable Localization** wizard, select the “language” (locale) you want to “move” existing data items to.

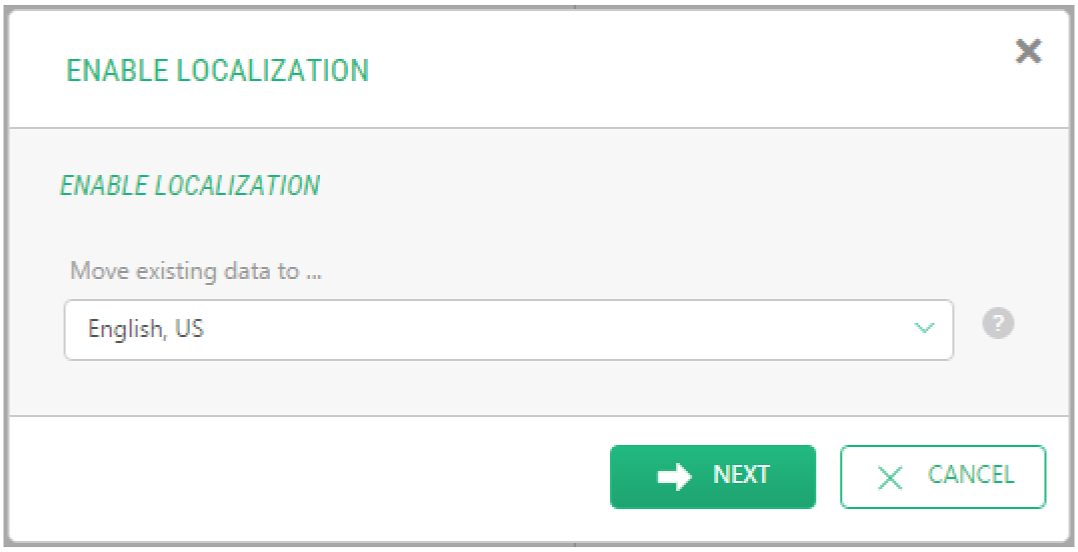


Figure 2: Move existing data to a proper locale

When you enable localization on a data type, you should choose *where* to keep the existing data items. We recommend that you keep ("move") the data items in the language you have created them in, which is often the default language in the system.

4. Click **Next** and then click **Finish** to confirm your settings.

Now the data type has been localized, and the button on the toolbar for this data type is replaced with the **Disable Localization** button.

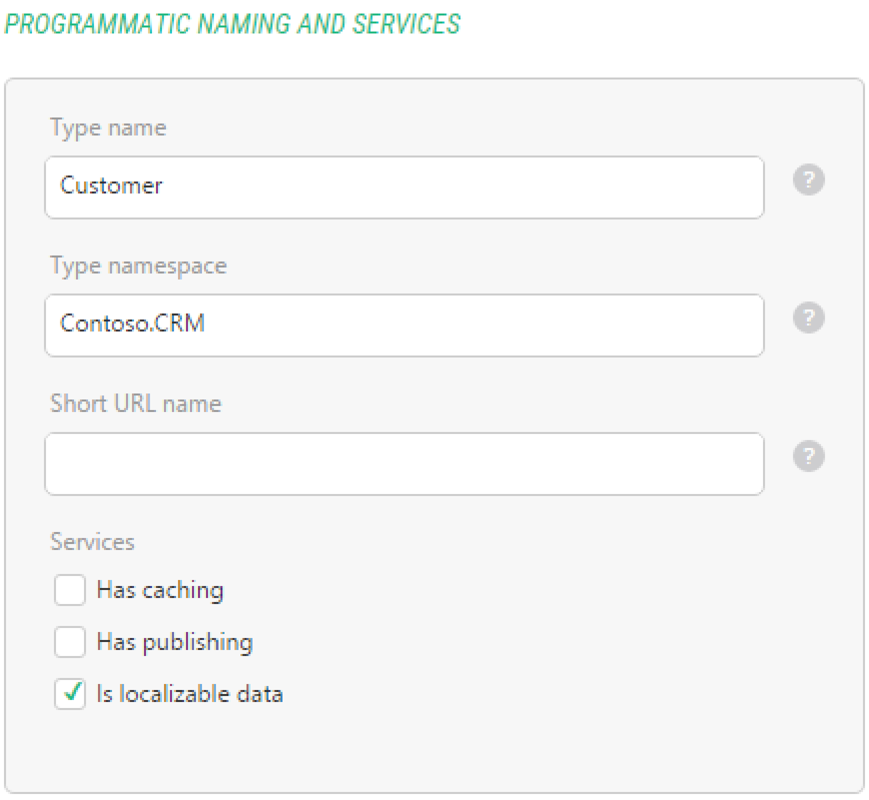
Figure 3: Disable Localization button on the toolbar

Before you go on to localize data items, learn how you can localize a data type you are creating.

2.1.2 Localizing Data Types Being Created

You can also enable localization on a data type while creating it:

1. In the **Data** perspective, select **Global data types** and click **Add Datatype** on the toolbar.
2. On the **Settings** tab, place a check mark in the **Is localizable data** check box.


PROGRAMMATIC NAMING AND SERVICES

Type name: Customer

Type namespace: Contoso.CRM

Short URL name:

Services
- [ ] Has caching
- [ ] Has publishing
- [✓] Is localizable data

Figure 4: Localizing a data type being currently created

3. Continue to create your data type.

Alternatively, you can first create your data type and then enable localization as described in the previous procedure.

2.2 Localizing Data Items

Localization of data items is similar to localization of web pages. Before you localize an item, you should make sure that the item has the *published* status.

To localize a data item:

1. Switch to the language you localize to.
2. Expand the data type with the target item.
3. Select the item and click **Translate** on the toolbar. The item editor form opens in the working area.
4. Replace the values in the original language with those in the target language where necessary.
5. Save the item.
6. Repeat Steps 3-5 for as many items as you need.

2.3 Troubleshooting

- If there is no Enable Localization on the toolbar or in the context menu of a data type, make sure that you have at least two languages added in the system.
- If the item you want to localize is grayed (disabled), make sure you have the item published in the original language. This might be the case if the data type requires that items should be manually published.
- If you have created the item in Language A, and it does not exist in Language B, make sure that the translation language is set to Language A (**View** > **Translation** > **From language**). In other words, to see the item created in one language, you should select the latter as the translation language when switching to another language.

3 Localizing Forms

When you localize your website (for example, from English to Danish), by default, the input forms rendered by Forms Renderer will only display their *labels* and *validation messages* in the main language (for example, English).

To have the form labels and validation messages in the language of the localized website, you should:

1. Create a .NET resource file for each language used.
2. Translate corresponding strings in each file.
3. Set specific properties of the data type used by the Forms Renderer to these strings.

Note: The system validation messages that appear when validation fails on the user's input in the field cannot be localized. However, you can have them automatically replaced with your user-friendly messages by providing Help texts for the data type fields used in your forms. These user-friendly messages can be localized.

3.1 Creating Resource File for Main Language

To create a resource file for the main language:

1. Open your website in Visual Studio.
2. Open or add the **App_GlobalResources** folder in the root of your website.
3. Add a new .NET resource file (.resx) there. (Right-click **App_GlobalResources** in Visual Studio's Solution Explorer > **Add** > **Resource File**).
4. In Visual Studio's Resource Editor add as many strings as you need for localization.

Alternatively, you can create the file locally with the minimum XML (see below) or already with the localized strings and upload it to the **App_GlobalResources** folder.

1. Locally create a .NET resource file (.resx) and name it something appropriate, for example, "**MainContactForm.resx**".
2. Copy the following XML into this file:

```
<root>
  <resheader name="resmimetype">
    <value>text/microsoft-resx</value>
  </resheader>
  <resheader name="version">
    <value>2.0</value>
  </resheader>
  <resheader name="reader">
    <value>System.Resources.ResXResourceReader, System.Windows.Forms,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <resheader name="writer">
    <value>System.Resources.ResXResourceWriter, System.Windows.Forms,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <!-- strings go here -->
  <data name="NameLabel" xml:space="preserve">
    <value>Name</value>
    <comment>This is a sample comment, not displayed on the site. This is
optional.
    </comment>
  </data>
  <data name="NameHelp" xml:space="preserve">
    <value>Enter your first and last name.</value>
  </data>
</root>
```

Listing 1: .NET resource file content

This file will serve as a resource file for your main language. When properly configured, the string values in this file will appear as labels and validation messages on forms in the main language.

3. Edit the file and add as many **<data>** elements as you need for each string you want to localize.

   As the file is XML-formatted, you can edit it in any XML editor. We recommend that you use Visual Studio to edit this file because it has a more convenient built-in resource editor.

   As you can see in the above XML, each string is represented by a **<data>** element, the **name** attribute of which specifies the name of the string you will refer to when localizing forms. As the name suggests, the **<value>** element specifies the value of this string, i.e. the string itself, which the user will see on a form.

4. Log into the CMS Console.
5. From the System perspective, expand the root of the website (***/***) and then, open or create the **App_GlobalResources** folder in the root.
6. Upload your resource file there. (Select **App_GlobalResources**, click **Upload File** on the toolbar and follow the steps in the wizard.)

Important! Please avoid creating an *empty* .resx file in this folder because it may cause an error on your website. Either create it in Visual Studio, or upload a local file with valid XML.

3.2 Creating Resource Files for Other Languages

For each language the website is localized to, you should create a localized copy of the resource file in the main language.

Use the following file naming pattern for each localized resource file:

<resource_name>.<culture_name>.resx

For example, if you have a Danish version of your English website, its resource file must be named "MainContactForm.da-DK.resx".

For information on culture names, please see https://msdn.microsoft.com/en-us/library/system.globalization.cultureinfo(vs.71).aspx.

3.3 Translating Strings in Localized Resource Files

Once you have created localized resource files, you should translate the strings.

For this, you should edit the file and replace the string values in the main language with the strings in the target language.

Note: Changes in the **App_GlobalResources** folder make ASP.NET (and hence C1 CMS) restart, which can make the application feel unresponsive right after updates.

3.4 Using Resource Strings in Data Type Field Properties

Once your resource files for the main and other languages are ready, you should use the resource strings in the **Label** and **Help** properties of the data type fields.

As the name suggests, the **Label**'s value is used in the labels of forms.

If provided, the **Help** text is used instead of the system validation messages for fields.

The reference to a resource string in those properties must comply with a specific format:

${Resource, Resources.<resource_file_name>.<string_name>}

<resource_file_name> stands for the name of the resource file without the culture name (e.g. “en-us”) and extension (“.resx”)

- **<string_name>** stands for the string name attached to the resource file name and separated from the resource file name with a period (“.”).

For example, if you want to refer to the string named **NameLabel** in the **MainContactForm.da-dk.resx** resource file, you should build the following value:

${Resources.MainContactForm.NameLabel}

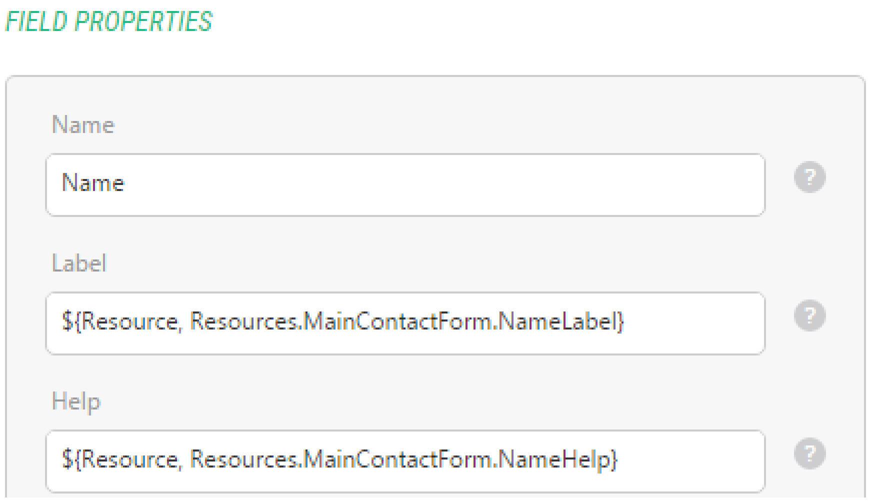
To specify resource strings for labels and messages on forms:

1. Edit the data type used by the Forms Renderer for the form you want to localize.
2. Open the **Fields** tab and select a field under the **Datatype fields**.
3. Refer to the string from the resource file in proper format in the field’s **Label** property, for example:

${Resource, Resources.MainContactForm.NameLabel}

4. Likewise, refer to the corresponding resource string in the **Help** property of the field, for example:

${Resource, Resources.MainContactForm.NameHelp}


FIELD PROPERTIES

Name: Name

Label: ${Resource, Resources.MainContactForm.NameLabel}

Help: ${Resource, Resources.MainContactForm.NameHelp}

Figure 5: Using resource strings in Label and Help properties

5. Repeat Steps 3-4 for the **Label** and **Help** properties of each data type field.

Now you can open a page with a form on your localized website and see the labels and messages translated into the language used.

3.5 Troubleshooting

If you are experiencing an ASP.NET error screen (for example, “your resource cannot be located”), make sure that you have:

- Upgraded to the latest version of the Forms Renderer add-on.
- Correctly named the file for the main language (without the culture name) and other languages (with the culture name)
- Correctly referred to the resource file and the string name in the data type field properties (Label and Help)

4 Using Advanced Widgets

Each type of a data type field has one or more widgets associated with it. If more than one widget can be used with a field, one of them is assigned to the field by default.

Some widgets have no parameters and can be used out-of-the-box when assigned. Other widgets have parameters and can be therefore configured.

Normally, the configurable widgets have reasonable default values for their parameters. You can thus use them out-of-the-box as well. However, if you have some specific requirements, you can override these defaults.

The following widgets have no parameters:

- DateSelector
- DateTimeSelector
- Selector (data reference)
- Optional Selector
- MediaFileFolderSelector

The configurable widgets are as follows:

- TextBox
- TextArea
- CheckBox
- VisualXhtmlEditor
- ImageSelector
- MediaFileSelector
- BoolSelector
- DataIdMultiSelector
- Selector (string)

Note: You can have a quick overview of all the widgets in *A Guide to Creating Data Types*.

In this chapter we will focus on the configurable widgets. You will learn in detail how to set up parameters of these widgets.

Each parameter is listed by its name used in the *Form XML* (“XML Name”), which may be different from its name visible to the user in the **Field Widget Configuration** window (“GUI Name”).

Each parameter has a short description provided and its type specified. It is indicated whether the parameter is required or optional. Its possible values and its default value are specified.

4.1 TextBox

The **TextBox** widget has allows you to choose whether the text entered will be spell-checked (Firefox only) based on the current language of the website. You should explicitly disable spell checking on fields that contain e-mails, code values etc. not suitable for spell checking.

SpellCheck

XML Name	SpellCheck
GUI Name	Spell check
Type	Boolean
Required	No
Description	If true (“Allow spell checking”), the text entered in the field will be spell-checked (Firefox and IE10 only).
Possible values	True (“Allow spell checking”) / False (“Do not allow spell checking”)
Default value	True

4.2 TextArea

The **TextArea** widget has allows you to choose whether the text entered will be spell-checked (Firefox only) based on the current language of the website. You should explicitly disable spell checking on fields that contain e-mails, code values etc. not suitable for spell checking.

SpellCheck

XML Name	SpellCheck
GUI Name	Spell check
Type	Boolean
Required	No
Description	If true (“Allow spell checking”), the text entered in the field will be spell-checked (Firefox and IE10 only).
Possible values	True (“Allow spell checking”) / False (“Do not allow spell checking”)
Default value	True

4.3 CheckBox

The **CheckBox** widget has a label to its left copied from its corresponding property. You can also specify the label for the check box item. It will appear to its right.

ItemLabel

XML Name	ItemLabel
GUI Name	Sub label
Type	String
Required	No
Description	Text to the right of the check box on a form
Possible values	Any valid string that can serve as a check box item label
Default value	(none)

4.4 BoolSelector

The **BoolSelector** widget always specifies two options to choose from, which read "*True*" and "*False*" by default. You can override these defaults by modifying its corresponding parameters.

TrueLabel

XML Name	TrueLabel
GUI Name	True label
Type	String
Required	Yes
Description	A label that describes one of the two possible options. Use this parameter to specify a custom label for the True option.
Possible values	Any valid string that can serve as a label for an option
Default value	True

FalseLabel

XML Name	FalseLabel
GUI Name	False label
Type	String
Required	Yes
Description	A label that describes one of the two possible options. Use this parameter to specify a custom label for the False option.
Possible values	Any valid string that can serve as a label for an option
Default value	False

4.5 VisualXhtmlEditor

The **VisualXhtmlEditor** widget allows the user to create and edit XHTML formatted content for the field. It comes with a predefined set of class names used for formatting XHTML elements ("common"). If you have defined your own set of class names, you can specify it instead.

ClassConfigurationName

XML Name	ClassConfigurationName
GUI Name	Class configuration name
Type	String
Required	No
Description	A string used to configure the visual editor and offer the editor a special set of class names for formatting XHTML elements.
Possible values	Any valid string that serve as the name of a predefined set of class names
Default value	common

4.6 DataIdMultiSelector

With the **DataIdMultiSelector** widget you can use items in a data type as options in a list to choose from. Since the widget has two views – *verbose* for longer lists and *compact* for shorter ones, you can configure how to present the options by selecting the proper view.

OptionsType

XML Name	OptionsType
GUI Name	Data type to select from
Type	Type
Required	Yes
Description	A data type whose items will serve as the list of options to choose from
Possible values	Any valid data type
Default value	(none)

CompactMode

XML Name	CompactMode
GUI Name	Compact UI
Type	Boolean
Required	No
Description	If true (“Compact”), a compact representation of long option lists will be used; otherwise, the full representation.
Possible values	True (Compact) / False (Verbose)
Default value	False (Verbose)

4.7 Selector (String)

Unlike the Selector widget used with other field types, the Selector widget used with the String type has a number of parameters to configure.

First of all, you can specify the option list to choose from by using a specific *function* that will return a simple enumerable list of such options (for example, **Composite.Utils.String.Split**) or an object with multiple properties (fields), for example a *data type*.

If the function returns an object with multiple properties, you should also choose which of the properties will supply values for options in the list (for example, “ID”) and which will supply labels for options in the list (for example, “Name”).

You can also make a selection of an option in a list required or optional. By allowing multiple selections and selecting the **Compact** or **Verbose** mode, you can change the way the options are presented to the user.

Options

XML Name	Options
GUI Name	Options
Type	IEnumerable (simple lists), Dictionary
Required	Yes
Description	A list of elements to use as options.
Possible values	A call to a function that can return a list of options
Default value	(none)

KeyFieldName

XML Name	KeyFieldName
GUI Name	Source key field name
Type	String
Required	No
Description	A name of the property in the option source, whose values will be used as key values. In other words, if your option source returns a list of objects or XElements, this field specifies the name of the field (property) to use for key values. Leave this field empty to use the source option value as a string.
Possible values	A valid string that specifies the name of the property in the option source, or an empty string (none)
Default value	(none)

LabelFieldName

XML Name	LabelFieldName
GUI Name	Source key field label
Type	String
Required	No
Description	A name of the property in the option source, whose values will be used as labels for options in the list. In other words, if your option source returns a list of objects or XElements, this field specifies the name of the field (property) to use for labels. Leave this field empty to use the source option value as a string.
Possible values	A valid string that specifies the name of the property in the option source, or an empty string (none)
Default value	(none)

Required

XML Name	Required
GUI Name	Selection required
Type	Boolean
Required	No
Description	If true, the user is forced to select a value in the list; otherwise, no selection is allowed ("<NONE>"). This parameter has no effect if the **Multiple selection** parameter is set to *True*.
Possible values	True / False
Default value	True

Multiple

XML Name	Multiple
GUI Name	Multiple selection
Type	Boolean
Required	No
Description	If *true*, the user can select none, one or more than one value in the list. The selected values will be joined in a comma-separated string like 'A,B,C'. If set to *true*, this feature disables the Selection required parameter. If set to *true*, it also changes the view of the widget from drop-down list to the compact or verbose list (as defined in the Compact mode parameter).
Possible values	True / False
Default value	False

Compact

XML Name	Compact
GUI Name	Compact mode
Type	Boolean
Required	No
Description	If *true*, the UI element will be presented in a compact view; otherwise, the verbose view is used. This parameter has effect only when the **Multiple selection** parameter is set to *True*; otherwise, a drop-down list is used.
Possible values	True (Compact) / False (Verbose)
Default value	False (Verbose/Show all options)

4.8 ImageSelector

By default, the **ImageSelector** widget allows the user to select the images from the entire Media Archive ("All Media Files").

To narrow the scope, you can choose a folder, from which the images will be only selected. You can also configure the widget so that the user can choose not to select any image in the list ("<NONE>").

MediaFileFolderReference

XML Name	MediaFileFolderReference
GUI Name	Image folder
Type	DataReference<C1 Media Folder>
Required	Yes
Description	A media folder to choose images from. It also includes images from all subfolders.
Possible values	Any valid path to a media folder
Default value	All Media Files

Required

XML Name	Required
GUI Name	Selection required
Type	Boolean
Required	No
Description	If *true*, selecting an image will be required; otherwise, optional ("<NONE>").
Possible values	True / False
Default value	True

4.9 MediaFileSelector

By default, the **MediaFileSelector** widget allows the user to select the media from the entire Media Archive ("All Media Files").

To narrow the scope, you can choose a specific folder, from which the media files will be only selected. You can further narrow the list of available media files by filtering the media files by their extensions as well as by indicating whether to include files from subfolders.

You can also configure the widget so that the user can choose not to select any media file in the list ("<NONE>").

MediaFileFolderReference

XML Name	MediaFileFolderReference
GUI Name	Media Folder
Type	DataReference<C1 Media Folder>
Required	Yes
Description	A media folder to choose files from.
Possible values	Any valid path to a media folder
Default value	All Media Files.

FileExtensionFilter

XML Name	FileExtensionFilter
GUI Name	File extension filter
Type	String
Required	No
Description	This parameter limits the list to files which have the specified extension.
Possible values	A string that specifies any valid file extension
Default value	(none)

IncludeSubfolders

XML Name	IncludeSubfolders
GUI Name	Include Subfolders
Type	Boolean
Required	No
Description	If *true*, files from subfolders will be included in the list; otherwise, only files in the specified folder will be listed
Possible values	True / False
Default value	True

Required

XML Name	Required
GUI Name	Selection required
Type	Boolean
Required	No
Description	If *true*, selecting a media file will be required; otherwise, optional (“<NONE>”).
Possible values	True / False
Default value	True

5 Editing Form Markup

When a user adds an item to a data type, C1 CMS automatically generates a XML-formatted form for entering values. By default, the input form in C1 CMS presents all the defined fields with matching widgets in an untitled field group box.

You can however customize the view of this form by editing its markup (XML). For example, you can make a text box read-only, add the title to a group box, group the widgets between two or more group boxes or tabs, or even use custom widgets for the fields defined in the data type.

In the following few sections you will learn how to edit a form's markup and modify the form's view.

5.1 Editing Form Markup

You can change a data item form's view and behavior by editing its markup (XML).

To edit the form markup:

1. Select a data type.
2. On the toolbar, click **Edit Form Markup**. The form markup opens in XHTML Source Editor in the working area.
3. Edit the XML contents. (The form XML structure and editing basics will be discussed shortly.)
4. Save the form markup.

5.1.1 XHTML Validation

The built-in XHTML validation mechanism will protect you from making mistakes while editing the form markup.

You will not be able to save poorly-formed XML. A message will inform you of an error if you try to save the form markup.

You can also force validation of the XML by clicking **Format** on the Source Editor's toolbar. Again, a message will inform you of any error in the XML.

Normally, the message describes what is wrong in the content and specifies the line and the position where the poorly-formed XML starts. Hence, you can easily locate and correct the error.

5.1.2 Form Markup File

When you save the modified form markup for the first time, a corresponding file is created on your hard disk as:

~/App_Data/Composite/DynamicTypeForms/<Namespace>/<DatatypeName>.xml

<Namespace> stands for the namespace of the data type

- **<DatatypeName>** stands for the name of the data type

For example, if you have a data type called **Samples.Users**, the file with modified markup can be found at:

~/App_Data/Composite/DynamicTypeForms/Samples/Users.xml

You can directly edit this file in an XML editor of your choice.

Important: *Once the form markup file is generated, any changes to the data type will not be automatically synchronized to the form markup. You should update your form markup manually to match the changes. For example, if you have added a new field to the data type, it will not appear in the form markup, you should add this field manually as described in this chapter.*

5.2 Overview of Form XML Structure

The auto-generated form markup has a default XML-based structure which only differs from form to form in fields and widgets. The elements used in the markup have their specific XML namespaces and defined in standard XML Schema definition files.

Some elements in the form markup are required, while others are optional.

Let's first review the XML namespaces required for elements in the form markup and then have a look at the default form's XML structure.

5.2.1 Required XML Namespaces

The form markup must specify a number of XML namespaces for XML elements to be valid and XML itself to be well-formed.

The following namespaces are present in the original form markup by default:

```
<cms:formdefinition
xmlns:cms="http://www.composite.net/ns/management/bindingforms/1.0"
xmlns="http://www.composite.net/ns/management/bindingforms/std.ui.controls.
lib/1.0"
xmlns:f="http://www.composite.net/ns/management/bindingforms/std.function.l
ib/1.0" >
<!-- skipped -->
</cms:formdefinition>
```

Listing 2: Required XML namespaces

Important: *You should never remove these namespaces.*

You can access and view the XML definitions by typing the following URL in your browser:

http://localhost/Composite/schemas

replacing **localhost** with the name of the server where your C1 CMS is running

If you use elements not covered by the above namespaces, you should specify additional namespace that will do. This might be the case with custom widgets.

5.2.2 Default Form XML Structure

Each form has the root element **<cms:formdefinition>** and two child elements: **<cms:bindings>** and **<cms:layout>**.

Bindings Section

Within the **<cms:bindings>** element, data type fields are listed, which are or can be bound to widgets.

Each field is introduced in the **<cms:binding>** element and has 3 attributes to set:

- name: (Required) It specifies the name of the field and corresponds to a data type field's **Name** property in the GUI
- type: (Required) It specifies the type of the field and corresponds to the **Field type** property

- **optional**: (Optional) It indicates whether the binding is optional or required.

```
<cms:formdefinition
xmlns:cms="http://www.composite.net/ns/management/bindingforms/1.0"
xmlns="http://www.composite.net/ns/management/bindingforms/std.ui.controls.
lib/1.0"
xmlns:f="http://www.composite.net/ns/management/bindingforms/std.function.l
ib/1.0">
  <cms:bindings>
    <cms:binding name="Id" type="System.Guid" optional="true" />
    <cms:binding name="Name" type="System.String" optional="true" />
    <cms:binding name="Choice" type="System.Boolean" optional="true" />
  </cms:bindings>
  <!-- skipped -->
</cms:formdefinition>
```

Listing 3: Sample elements in <cms:bindings>

Important: *The* **<cms:bindings>** *section is filled by C1 CMS automatically based on the data type used with this form. You are not supposed to change anything here.*

The fields listed here are referred to in the **<cms:layout>** section when used in or bound to widgets.

Layout Section

The layout of the form is handled in the **<cms:layout>** element. This is where you can add, modify or remove widget elements and change the layout of the widgets on the editor form in general.

The **<cms:layout>** element has two optional attributes, also mirrored in its respective child elements:

- **label** (**<cms:label>**): specifies the label of the form, which appears as a title on the tab in the working area. If no value is provided, the name of the data type is displayed instead.
- **iconhandle** (**<cms:iconhandle>**): specifies the icon to use with the form's label. If no value is provided, the default icon is used.

In the form markup, the label is set to the data type field, which is configured to serve as a title field in lists.

5.2.3 Root Element in Layout Section

Within the **<cms:layout>** element, a root element must be defined, which will serve as a container for other elements representing widgets or other containers.

By default, C1 CMS places all the elements under the **<FieldGroup>** element that serves as a container and the root element here. However, you can use other container elements available for the form instead of **<FieldGroup>**.

5.2.4 Widget Elements

Under the root element in the **<cms:layout>** section, C1 CMS places the widget elements binding them to the data type fields specified in the **<cms:bindings>** section.

Note: Along with widget elements you can insert other elements in this section such as container elements and functions. In this guide we will only focus on widgets and containers.

As a rule, a widget element has a number of attributes that in most cases match the widget's properties you can set for a data type field in the GUI. However, some widgets may have attributes that can be set only in the form markup.

The attributes are normally mirrored in respective child elements, which are more preferable for use in a number of cases. For example, you can set the Help text for a widget by using the **Help** attribute of the widget element:

```
<TextBox Label="Name" Help="Enter your name here" />
```

Listing 4: Setting the Help property in the Help attribute

When you, however, bind the widget's property to a data type field ("Text", in the example below), you will most likely use its corresponding child element. In the example below, it is accounted for by using another element for the value:

```
<cms:formdefinition
xmlns:cms="http://www.composite.net/ns/management/bindingforms/1.0"
xmlns="http://www.composite.net/ns/management/bindingforms/std.ui.controls.
lib/1.0"
xmlns:f="http://www.composite.net/ns/management/bindingforms/std.function.l
ib/1.0">
  <!-- skipped -->
  <cms:layout>
  <!-- skipped -->
    <TextBox Label="Name" Help=" Enter your name here ">
      <TextBox.Text>
        <cms:bind source="Name" />
      </TextBox.Text>
    </TextBox>
    <!-- skipped -->
  </cms:layout>
</cms:formdefinition>
```

Listing 5: Setting the Text property in the Text child element

5.2.5 Common Attributes

Most widget or container elements have at least two common attributes: **Label** and **Help**.

As the names suggest, these attributes specify the label and the help text for a widget or a container. They correspond to the **Label** and **Help** properties of a data type field specified in the GUI.

5.2.6 Initializing Widgets

Each widget that has a bindable property can be initialized with a specific value.

3 options are available for simple values:

- Specifying a value in the corresponding element's attribute
- Placing a value between a widget element's start and end tags and thus referring to the bindable property implicitly
- Placing a value between the start and end tags of a widget's bindable property element (explicitly)

For instance, you have a **TextBox** widget named and labeled "*Country*" to be initialized with "*Denmark*" as a value. The **TextBox** widget has a bindable property called **Text** available both as an attribute and as a child element. Your options could be:

```
<TextBox Label="Country" Text="Denmark" />
```

Listing 6: Initializing the value in an attribute (Option 1)

```
<TextBox Label="Country">Denmark</TextBox>
```

Listing 7: Initializing the value of the bindable property implicitly (Option 2)

```
<TextBox Label="Country">
```

```
  <TextBox.Text>Denmark</TextBox.Text>
</TextBox>
```

Listing 8: Initializing the value of the bindable property explicitly (Option 3)

You can also use a value that the corresponding data type field might have, especially if the field has a default value. In this case, you should use the **<cms:read />** element and specify the field's name in its **source** attribute. You can use the **<cms:read />** element in the last 2 options (as a child element).

For example, your data type has a field named "*Country*" and you want to set the widget to whatever the value is in this field by default. You can do one of the following:

```
<cms:formdefinition
xmlns:cms="http://www.composite.net/ns/management/bindingforms/1.0"
xmlns="http://www.composite.net/ns/management/bindingforms/std.ui.controls.
lib/1.0"
xmlns:f="http://www.composite.net/ns/management/bindingforms/std.function.l
ib/1.0">
  <!-- skipped -->
  <cms:layout>
  <!-- skipped -->
    <TextBox Label="Country">
      <cms:read source="Country" />
    </TextBox>
    <!-- skipped -->
  </cms:layout>
</cms:formdefinition>
```

Listing 9: Initializing the value with <cms:read> implicitly (Option 2)

```
<cms:formdefinition
xmlns:cms="http://www.composite.net/ns/management/bindingforms/1.0"
xmlns="http://www.composite.net/ns/management/bindingforms/std.ui.controls.
lib/1.0"
xmlns:f="http://www.composite.net/ns/management/bindingforms/std.function.l
ib/1.0">
  <!-- skipped -->
  <cms:layout>
  <!-- skipped -->
    <TextBox Label="Country">
      <TextBox.Text>
        <cms:read source="Country" />
      </TextBox.Text>
    </TextBox>
    <!-- skipped -->
  </cms:layout>
</cms:formdefinition>
```

Listing 10: Initializing the value with <cms:read> explicitly (Option 3)

When C1 CMS auto-generates a form, it normally uses Option 3 - the <cms:read /> element to bind a data type field to a matching widget.

5.2.7 Binding Fields to Widgets

When you bind a data type field to a widget, not only is the value of the field copied to the widget but also the value entered by the user in the widget is copied to the data type field.

To bind a data type field to the widget property, you should use the **<cms:bind />** element.

```
<cms:formdefinition
xmlns:cms="http://www.composite.net/ns/management/bindingforms/1.0"
xmlns="http://www.composite.net/ns/management/bindingforms/std.ui.controls.
lib/1.0"
xmlns:f="http://www.composite.net/ns/management/bindingforms/std.function.l
ib/1.0">
```

```
  <!-- skipped -->
  <cms:layout>
  <!-- skipped -->
    <TextBox.Text>
      <cms:bind source="Name" />
    </TextBox.Text>
    <!-- skipped -->
  </cms:layout>
</cms:formdefinition>
```

Listing 11: Binding a filed to a widget's property

In the **source** attribute of this element, you specify the field you bind to the widget. One field can be only bound to one widget. The field bound to a widget must be listed in the **<cms:bindings>** section.

5.2.8 Commonly Used Widget Elements

Each widget used with a data type field is represented with widget elements in the form markup.

To see definition of all the available elements:

1. Type **http://localhost/Composite/schemas** in your browser replacing **localhost** with the name of the server where your C1 CMS is running.
2. Click the namespace URL **http://www.composite.net/ns/management/bindingforms/std.ui.controls.lib/1.0**

The following is the list of widgets matched against their widget elements:

Widgets Matched Against Elements

Field Widget	**Field Type**	**Element in Form XML**
TextBox	String, Integer, Decimal, GUID	<TextBox />
TextArea	String	<TextArea />
DateSelector	Date	<DateSelector />
DateTimeSelector	Date	<DateTimeSelector />
CheckBox	Boolean	<CheckBox />
BoolSelector	Boolean	<BoolSelector />
VisualXhtmlEditor	String	<InlineXhtmlEditor />
DataIdMultiSelector	String	<MultiKeySelector />
Selector	String	<KeySelector />
Selector	String	<MultiKeySelector />
ImageSelector	Data Reference: C1 Image	<KeySelector />
MediaFileSelector	Data Reference: C1 Media File	<KeySelector />
MediaFolderFileSelector	Data Reference: C1 Media Folder	<DataReferenceSelector />
Selector	Data Reference (all types)	<KeySelector />
OptionalSelector	Data Reference (all types)	<KeySelector />

As you can see in the table above, many data type field widgets are implemented with fewer widget elements configured in the properties to be widget-specific.

For example, **TextBox** widgets for different field types are implemented with the same **<TextBox>** element and differ in the value of its implicit **Type** attribute. You will learn about using the **TextBox**’s **Type** attribute shortly.

Various selectors are implemented in most cases with the **<KeySelector>** element. In some specific cases, two additional selector elements are used: **<MultiKeySelector>** and **<DataReferenceSelector>**.

5.2.9 Specific Widget Element Attributes

Apart from the two properties, **Label** and **Help**, common to all the widgets and available as attributes and child elements of widget elements, the latter may have their specific attributes.

Normally, you can set them either in the **Field Widget Configuration** window or in the form markup. In some cases, you can only access their attributes in the form markup (for example, **Type** of **<TextBox>**).

The following is the list of the correspondences between the widget property and the widget element's attribute or child attribute. If the attribute has no corresponding widget property, the attribute is element-specific and can be only set in the form markup. A bindable property can be normally configured as the field's **Default value** property.

(The **Label** and **Help** attributes are omitted as common to all the elements below.)

<TextBox>		
Widget Element Attribute	**Widget Property**	**Bindable**
Text		Yes
Type		

<TextArea>		
Widget Element Attribute	**Widget Property**	**Bindable**
Text		Yes
Type		

<DateSelector>		
Widget Element Attribute	**Widget Property**	**Bindable**
Date		Yes

<DateTimeSelector>		
Widget Element Attribute	**Widget Property**	**Bindable**
Text		Yes

<CheckBox>		
Widget Element Attribute	**Widget Property**	**Bindable**
Checked		Yes
ItemLabel	Sub label	

<BoolSelector>		
Widget Element Attribute	**Widget Property**	**Bindable**
IsTrue		Yes
TrueLabel	True label	
FalseLabel	False label	

<InlineXhtmlEditor>		
Widget Element Attribute	**Widget Property**	**Bindable**
Xhtml		Yes
ClassConfigurationName	Class configuration name	
EmbedableFieldsTypes		

<KeySelector>		
Widget Element Attribute	**Widget Property**	**Bindable**
Selected		Yes
Options	Options	
OptionsKeyField	Source key field name	
OptionsLabelField	Source label field name	
Required	Selection required	
BindingType		
SelectedIndexChangedEventHandler		

<MultiKeySelector>		
Widget Element Attribute	**Widget Property**	**Bindable**
Selected		Yes
Options	Options	
OptionsKeyField	Source key field name	
OptionsLabelField	Source label field name	
Required	Selection required	
CompactMode	Compact mode	
BindingType		
SelectedIndexChangedEventHandler		

<DataReferenceSelector>		
Widget Element Attribute	**Widget Property**	**Bindable**
Selected		Yes
DataType		

It depends on the type of selector widget whether **<KeySelector>** and **<MultiKeySelector>** have their properties available for configuration in the **Field Widget Configuration** window. The names of the properties might also differ in the GUI from selector to selector.

5.2.10 Other Widget Elements

If you examine the XML schema definition for form UI controls (**http://www.composite.net/ns/management/bindingforms/std.ui.controls.lib/1.0**), you will notice that it has a number of other elements, for example, **<Heading>** or **<MarkupEditor>**.

Normally, they are not used in regular data type forms and are out of the scope in this guide.

5.3 Customizing TextBox View

The **TextBox** widget can be customized in both its view and its behavior by setting the **Type** attribute of the **<TextBox>** element in the form markup.

The following values are possible:

<TextBox> Type Attribute Values

Value	**Description**
ReadOnly	Makes the text box read-only
Password	Hides the input string behind a string of special characters as used on password fields
String	Accepts strings only
Integer	Accepts integers only
Decimal	Accepts decimals only
Guid	Accepts GUIDs only
ProgrammingIdentifier	Accepts Programming Identifiers only; normally not used in data type forms
ProgrammingNamespace	Accepts Programming Namespaces only; normally not used in data type forms

Using these values, you can make the text box widget read-only, hide passwords behind a string of specific characters, or link it to a specific field type such as an integer or a GUID leading to validation of its input values.

For example, if you want the text box widget to be read-only:

1. Edit the form markup.
2. Add the **Type** attribute to the **TextBox** element.
3. Specify "*ReadOnly*" as the value in the attribute.
4. Save the form.

When you add a data item, you will see that the text box is now read-only.

Note: The **TextArea** widget also has the **Type** attribute, but it can only accept the "ReadOnly" value; other values are only **TextBox**-specific.

5.4 Using Container Elements

By default, C1 CMS places all the widgets in a field group box. In the form markup, the latter is represented with the **<FieldGroup>** element. This is a container element, and therefore can contain other elements.

In addition to the **<FieldGroup>** element, you can use two more container elements in the form markup: **<PlaceHolder>** and **<TabPanels>**.

5.4.1 FieldGroup Element

The **<FieldGroup>** container element arranges its elements one below another and places them in a box. Its primary purpose is - as the name suggests – to group related field widgets, enhancing usability of the form, as a result.

It has two attributes: **Label** and **Help**. By default, it has no attributes specified in the auto-generated form markup. However, you can use its **Label** attribute to have a titled group box.

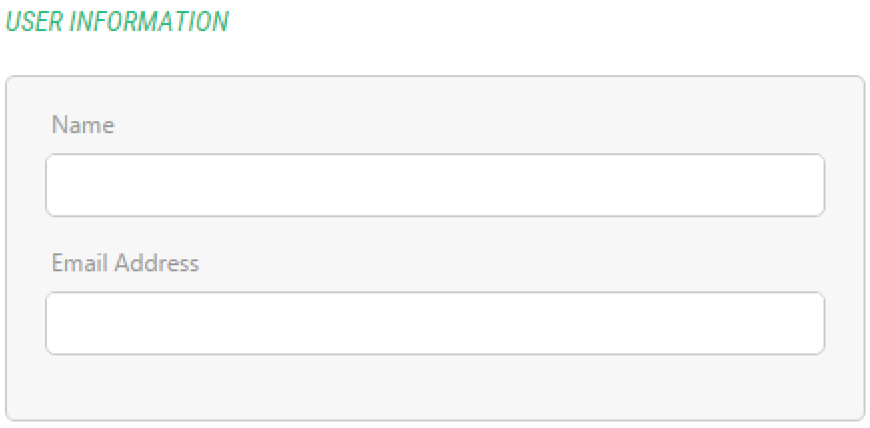


Figure 6: A field group with the label

5.4.2 PlaceHolder Element

The **<PlaceHolder>** container element is the simplest out of the three. As **<FieldGroup>**, it also arranges its elements one below another, but without placing them in a box. It practically has no visual representation.

It has two attributes: Label and Help. When used alone, the attribute values are not used. They might become visible when used in combination with other container elements.

<PlaceHolder> keeps the elements within its borders and can be used as a root element in the <cms:layout> section to arrange several field group boxes.

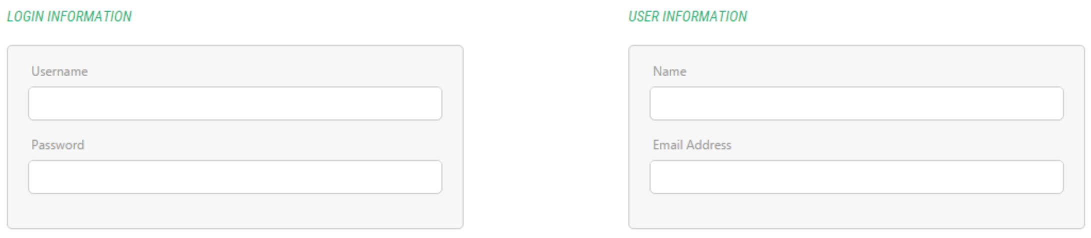


Figure 7: <PlaceHolder> used to lay out two field groups

You can also use it as a single tab in **<TabPanels>.**

5.4.3 TabPanels Element

The **<TabPanels>** container element groups widgets on tabs handling a form overloaded with numerous fields and enhancing its usability, as a result.

It uses its child elements as individual tabs so the best option to choose is a number of **<PlaceHolder>** elements as its child elements, each with the actual widgets or group of widgets within these “tab” place holders.

Figure 8: <TabPanels> used

Apart from its **Label** and **Help** attributes, usually not visible in the form GUI, it has its own specific attribute – **PreSelectedIndex** – which allows you to pre-select the tab when the form opens.

The index is zero-based, so if you have 3 tabs on your form, you should set this attribute to “2” if you want the 3rd tab to be pre-selected.

5.5 Customizing Form Layout

By default, C1 CMS generates a form that displays all its widgets in a group box. To enhance usability of the form, you can group related widgets using multiple field groups or you can place groups of field widgets on multiple tabs on a form.

5.5.1 Grouping Fields between Multiple Field Groups

To be able to use multiple field groups in the same form, you should change the root element in the **<cms:layout>** section of the form markup. The best option is **<PlaceHolder>**.

Also using two or more field groups implies naming each of the group for easier reference and better user experience.

1. Edit the form markup.
2. Insert the **<PlaceHolder>** element in **<cms:layout>**.
3. Place the current **<FieldGroup>** element with all its child elements within **<PlaceHolder>**.
4. Based on your requirements, add one or more **<FieldGroup>** elements within **<PlaceHolder>**
5. Move the necessary widget elements from the original **<FieldGroup>** to those you have just added where appropriate.
6. Add the **Label** attribute to each **<FieldGroup>** and use it to name each field group.
7. Save your form.

```
<PlaceHolder>
      <FieldGroup Label="Required Settings">
            <!-- widget elements go here -->
      </FieldGroup>
      <FieldGroup Label="Optional Settings">
            <!-- widget elements go here -->
      </FieldGroup>
</PlaceHolder>
```

Listing 12: Multiple field groups

Once the form is saved, you can add an item to the data type it represents and see the changes in its view.

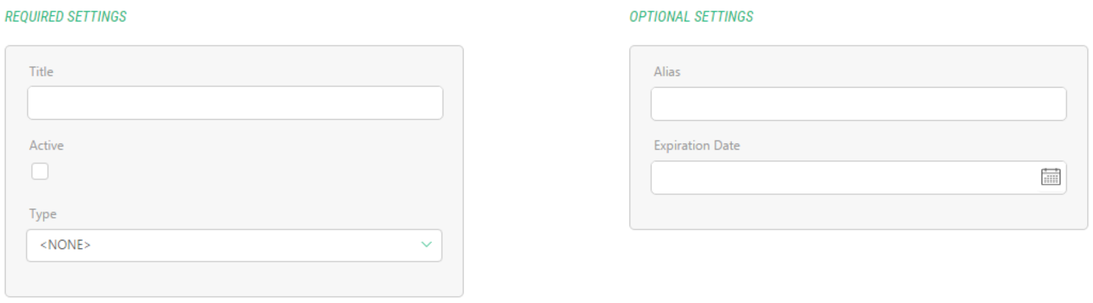


Figure 9: Multiple field groups in the GUI

5.5.2 Grouping Fields on Multiple Tabs

If you want to group the widgets using tabs, you should change the root element in the **<cms:layout>** section of the form markup to the **<TabPanels>**. Besides, you should create each tab individually by using **<PlaceHolder>** elements within **<TabPanels>**.

As tabs normally have titles, you should also name them. Optionally, you can pre-select a certain tab when the form opens.

1. Edit the form markup.
2. Insert the **<TabPanels>** element within **<cms:layout>**.
3. Within **<TabPanels>**, insert as many **<PlaceHolder>** elements as many tabs you need.
4. Add the **Label** attribute to each **<PlaceHolder>** and use it to name each tab.
5. Place the current **<FieldGroup>** element with all its child elements within the first **<PlaceHolder>**.
6. Insert a **<FieldGroup>** element into the empty **<PlaceHolder>**.
7. Based on your requirements, move the necessary widget elements from the original **<FieldGroup>** in the first **<PlaceHolder>** to the **<FieldGroup>** within the empty **<PlaceHolder>**.
8. If you have more than two tabs, repeat Steps 6-7.
9. Optionally, you can add the **PreSelectedIndex** attribute to the **<TabPanels>** element and specify its value.
10. Save your form.

```
<TabPanels>
<PlaceHolder Label="Main">
             <FieldGroup >
                    <!-- widget elements go here -->
             </FieldGroup>
</PlaceHolder>
<PlaceHolder Label="Optional">
             <FieldGroup >
                    <!-- widget elements go here -->
             </FieldGroup>
</PlaceHolder>
</TabPanels>
```

Listing 13: Multiple tabs

Once the form is saved, you can add an item to the data type it represents and see the changes.

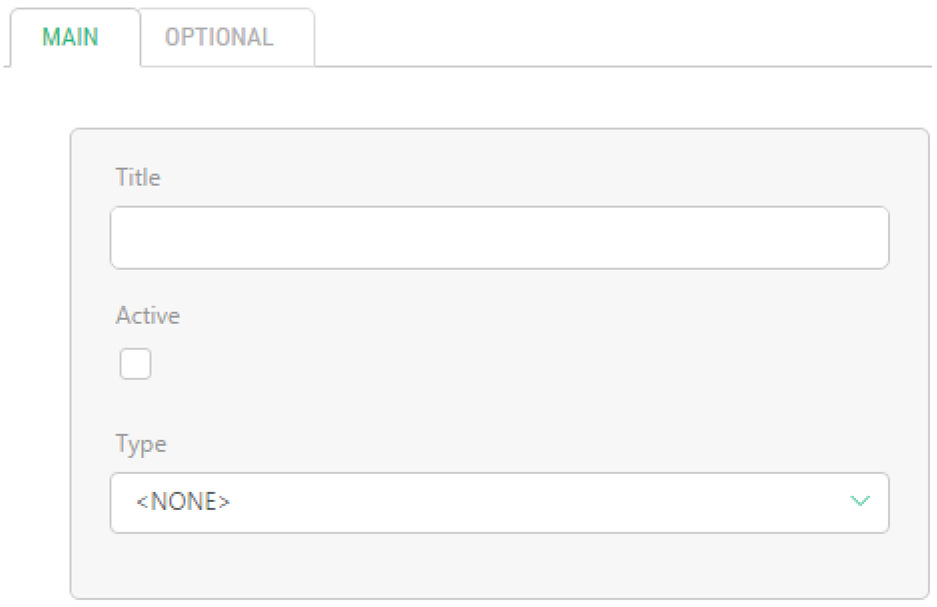


Figure 10: Multiple tabs in the GUI

You are not limited to one field group on one tab. You can create as many field groups as you need.

Besides, you can have child tabs on a tab. You just need to add another **<TabPanels>** element within one of the **<PlaceHolder>** elements and repeat the above procedure.

5.6 Example of Editing Form Markup

Let's assume that you have data type that holds information about user account. Each user account must have the following fields:

Required personal information

- Username
- Password
- Name
- Email Address (Email)

Optional personal information

- Age
- City
- Country

Service information

- User ID (UserId)
- Account Created (Created)

The requirements to the form are as follows:

- The fields must be distributed between 3 tabs (as it is grouped in the list above): "Main", "Optional", and "Service".
- The **Username** and **Password** fields must be grouped in the **Login** field group, the **Name** and **Email Address** fields must be grouped in the **User Information** field group.
- The password in the **Password** field should be hidden.

- The **User ID** must be read-only and show the GUID of the user account generated and assigned when the account was created.
- The **Account Created** field must be read-only and show the date when the account was created.

Before changing the form markup, you should specify the default value for the Created field:

1. Select the **Created** field.
2. On the **Advanced** tab, click in the **Default value** field.
3. In the **Select Function** dialog, select the **Composite.Utils.Date.Now** function.
4. Click **OK** in the **Select Function** dialog and then in the **Field Default Value Configuration**.

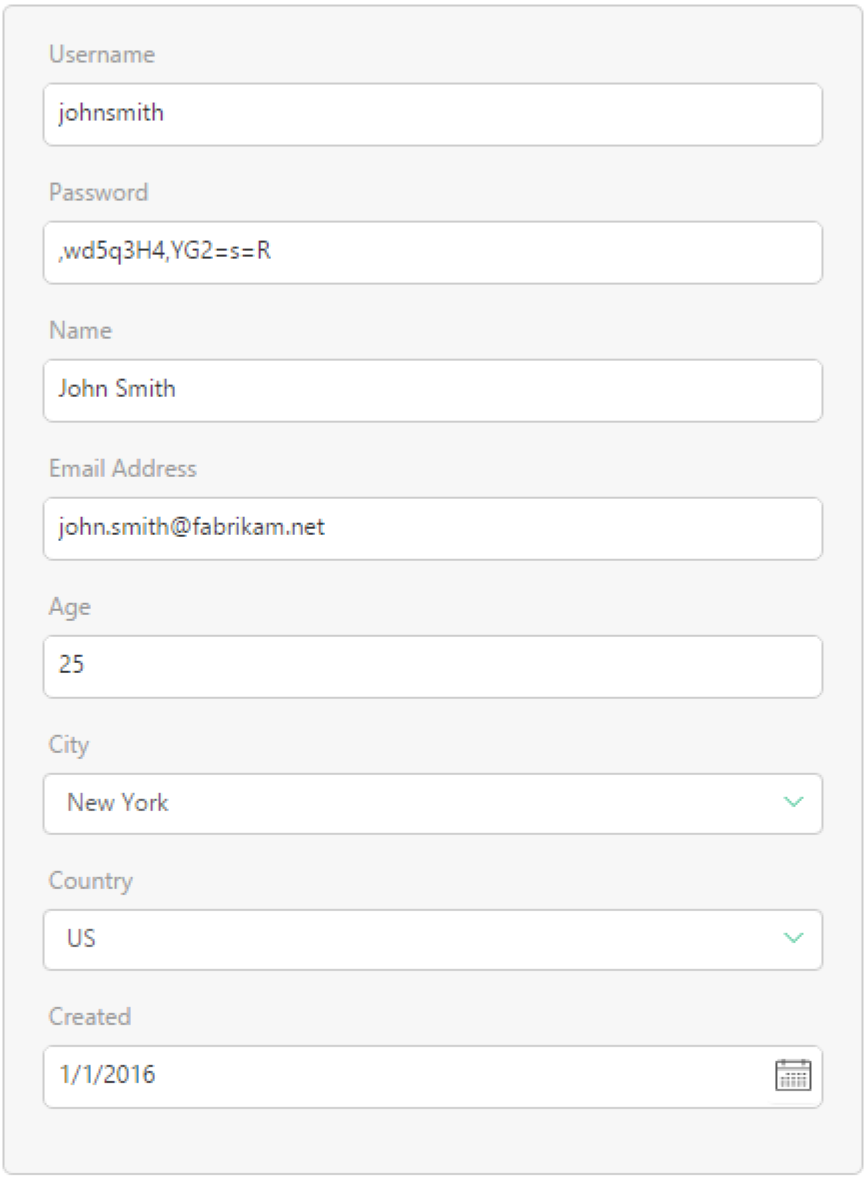
When the user opens the form to enter values in the fields, the Account Created field will be already initialized to the current date.

First, let's see what markup C1 CMS generated by default:

```
<cms:formdefinition
xmlns:cms="http://www.composite.net/ns/management/bindingforms/1.0"
xmlns="http://www.composite.net/ns/management/bindingforms/std.ui.controls.
lib/1.0"
xmlns:f="http://www.composite.net/ns/management/bindingforms/std.function.l
ib/1.0">
  <cms:bindings>
    <cms:binding name="Id" type="System.Guid" optional="true" />
    <cms:binding name="Username" type="System.String" optional="true" />
    <cms:binding name="Password" type="System.String" optional="true" />
    <cms:binding name="Name" type="System.String" optional="true" />
    <cms:binding name="Email" type="System.String" optional="true" />
    <cms:binding name="Age" type="System.Int32" optional="true" />
    <cms:binding name="City" type="System.Guid" optional="true" />
    <cms:binding name="Country" type="System.Guid" optional="true" />
    <cms:binding name="Created" type="System.DateTime" optional="true" />
  </cms:bindings>
  <cms:layout>
    <cms:layout.label>
      <cms:read source="Name" />
    </cms:layout.label>
      <FieldGroup>
        <TextBox Label="Username" Help="">
          <TextBox.Text>
            <cms:bind source="Username" />
          </TextBox.Text>
        </TextBox>
        <TextBox Label="Password" Help="">
          <TextBox.Text>
            <cms:bind source="Password" />
          </TextBox.Text>
        </TextBox>
        <TextBox Label="Name" Help="">
          <TextBox.Text>
            <cms:bind source="Name" />
          </TextBox.Text>
        </TextBox>
        <TextBox Label="Email Address" Help="">
          <TextBox.Text>
            <cms:bind source="Email" />
          </TextBox.Text>
        </TextBox>
        <TextBox Label="Age" Help="" Type="Integer">
          <TextBox.Text>
            <cms:bind source="Age" />
          </TextBox.Text>
        </TextBox>
        <KeySelector Label="City" Help="" OptionsKeyField="Key"
OptionsLabelField="Label" Required="false">
          <KeySelector.Selected>
            <cms:bind source="City" />
          </KeySelector.Selected>
          <KeySelector.Options>
            <f:StaticMethodCall Type="&lt;t
n=&quot;Composite.StandardPlugins.Functions.WidgetFunctionProviders.Standar
dWidgetFunctionProvider.DataReference.NullableDataReferenceSelectorWidgetFu
nction`1, Composite, Version=1.2.3610.26719, Culture=neutral,
PublicKeyToken=null&quot;&gt;&#xD;
  &lt;t n=&quot;DynamicType:Demo.Cities&quot; /&gt;&#xD;
&lt;/t&gt;" Method="GetOptions" Parameters="DynamicType:Demo.Cities" />
          </KeySelector.Options>
        </KeySelector>
        <KeySelector Label="Country" Help="" OptionsKeyField="Key"
OptionsLabelField="Label" Required="false">
          <KeySelector.Selected>
            <cms:bind source="Country" />
          </KeySelector.Selected>
          <KeySelector.Options>
            <f:StaticMethodCall Type="&lt;t
n=&quot;Composite.StandardPlugins.Functions.WidgetFunctionProviders.Standar
dWidgetFunctionProvider.DataReference.NullableDataReferenceSelectorWidgetFu
nction`1, Composite, Version=1.2.3610.26719, Culture=neutral,
PublicKeyToken=null&quot;&gt;&#xD;
  &lt;t n=&quot;DynamicType:Demo.Countries&quot; /&gt;&#xD;
&lt;/t&gt;" Method="GetOptions" Parameters="DynamicType:Demo.Countries" />
          </KeySelector.Options>
        </KeySelector>
        <DateSelector Label="Account Created" Help="">
          <DateSelector.Date>
            <cms:bind source="Created" />
          </DateSelector.Date>
        </DateSelector>
      </FieldGroup>
  </cms:layout>
</cms:formdefinition>
```

Listing 14: Form markup auto-generated by C1 CMS

All the fields are placed together within one field group.



Username	johnsmith
Password	,wd5q3H4,YG2=s=R
Name	John Smith
Email Address	john.smith@fabrikam.net
Age	25
City	New York
Country	US
Created	1/1/2016

Figure 11: Data item editor form auto-generated by C1 CMS

Now let's implement the requirements to the form.

```
<cms:formdefinition
xmlns:cms="http://www.composite.net/ns/management/bindingforms/1.0"
xmlns="http://www.composite.net/ns/management/bindingforms/std.ui.controls.
lib/1.0"
xmlns:f="http://www.composite.net/ns/management/bindingforms/std.function.l
ib/1.0">
  <cms:bindings>
    <cms:binding name="Id" type="System.Guid" optional="true" />
    <cms:binding name="Username" type="System.String" optional="true" />
    <cms:binding name="Password" type="System.String" optional="true" />
    <cms:binding name="Name" type="System.String" optional="true" />
    <cms:binding name="Email" type="System.String" optional="true" />
    <cms:binding name="Age" type="System.Int32" optional="true" />
    <cms:binding name="City" type="System.Guid" optional="true" />
    <cms:binding name="Country" type="System.Guid" optional="true" />
    <cms:binding name="Created" type="System.DateTime" optional="true" />
  </cms:bindings>
  <cms:layout>
    <cms:layout.label>
      <cms:read source="Name" />
    </cms:layout.label>
    <TabPanels>
      <PlaceHolder Label="Main">
        <FieldGroup Label="Login">
          <TextBox Label="Username" Help="">
            <TextBox.Text>
              <cms:bind source="Username" />
            </TextBox.Text>
          </TextBox>
          <TextBox Label="Password" Help="" Type="Password">
            <TextBox.Text>
              <cms:bind source="Password" />
            </TextBox.Text>
          </TextBox>
        </FieldGroup>
        <FieldGroup Label="User Information">
          <TextBox Label="Name" Help="">
            <TextBox.Text>
              <cms:bind source="Name" />
            </TextBox.Text>
          </TextBox>
          <TextBox Label="Email Address" Help="">
            <TextBox.Text>
              <cms:bind source="Email" />
            </TextBox.Text>
          </TextBox>
        </FieldGroup>
      </PlaceHolder>
      <PlaceHolder Label="Optional">
        <FieldGroup>
          <TextBox Label="Age" Help="" Type="Integer">
            <TextBox.Text>
              <cms:bind source="Age" />
            </TextBox.Text>
          </TextBox>
          <KeySelector Label="City" Help="" OptionsKeyField="Key"
OptionsLabelField="Label" Required="false">
            <KeySelector.Selected>
              <cms:bind source="City" />
            </KeySelector.Selected>
            <KeySelector.Options>
              <f:StaticMethodCall Type="&lt;t
n=&quot;Composite.StandardPlugins.Functions.WidgetFunctionProviders.Standar
dWidgetFunctionProvider.DataReference.NullableDataReferenceSelectorWidgetFu
nction`1, Composite, Version=1.2.3610.26719, Culture=neutral,
PublicKeyToken=null&quot;&gt;&#xD;
  &lt;t n=&quot;DynamicType:Demo.Cities&quot; /&gt;&#xD;
&lt;/t&gt;" Method="GetOptions" Parameters="DynamicType:Demo.Cities" />
            </KeySelector.Options>
          </KeySelector>
          <KeySelector Label="Country" Help="" OptionsKeyField="Key"
OptionsLabelField="Label" Required="false">
            <KeySelector.Selected>
              <cms:bind source="Country" />
            </KeySelector.Selected>
            <KeySelector.Options>
              <f:StaticMethodCall Type="&lt;t
n=&quot;Composite.StandardPlugins.Functions.WidgetFunctionProviders.Standar
dWidgetFunctionProvider.DataReference.NullableDataReferenceSelectorWidgetFu
nction`1, Composite, Version=1.2.3610.26719, Culture=neutral,
PublicKeyToken=null&quot;&gt;&#xD;
  &lt;t n=&quot;DynamicType:Demo.Countries&quot; /&gt;&#xD;
&lt;/t&gt;" Method="GetOptions" Parameters="DynamicType:Demo.Countries" />
            </KeySelector.Options>
          </KeySelector>
        </FieldGroup>
      </PlaceHolder>
      <PlaceHolder Label="Service">
```

```
        <FieldGroup>
          <TextBox Label="User ID" Help="" Type="ReadOnly">
            <TextBox.Text>
              <cms:read source="Id" />
            </TextBox.Text>
          </TextBox>
          <TextBox Label="Account Created" Help="" Type="ReadOnly">
            <TextBox.Text>
              <cms:read source="Created" />
            </TextBox.Text>
          </TextBox>
        </FieldGroup>
      </PlaceHolder>
    </TabPanels>
  </cms:layout>
</cms:formdefinition>
```

Listing 15: Customized form markup

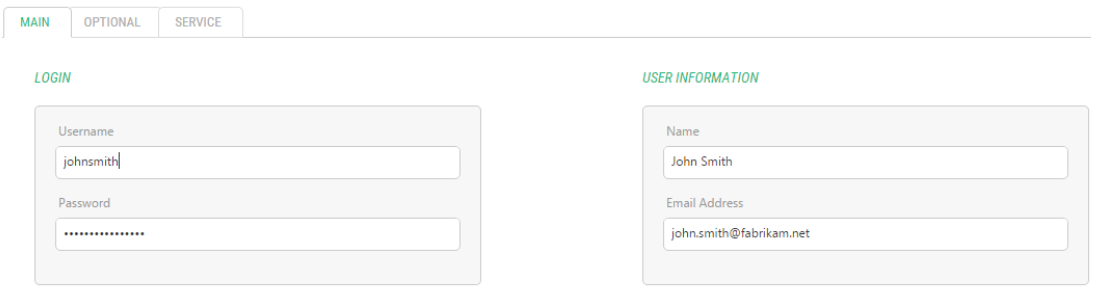
Let’s highlight some of the changes in the markup above. First of all, we have created tabs and field groups and distributed the existing widgets among them:

1. We have added a **<TabPanels>** element in the **<cms:layout>** and inserted 3 child **<PlaceHolder>** elements within the **<TabPanels>** element. We have also added the **Label** attribute to each **<PlaceHolder>** element and set its value to “*Main*”, “*Optional*” and “*Service*”.
2. We have moved the default **<FieldGroup>** element with all its child elements to the *Main* tab. We have also added the **Label** attribute to this **<FieldGroup>** element and set its value to “*Login*”.
3. We have added another **<FieldGroup>** element onto the *Main* tab, set its **Label** attribute to “*User Information*” and moved 3 widget elements in for these fields: **FirstName**, **LastName**, **Email**.
4. We have added another **<FieldGroup>** element onto the *Optional* tab and moved 3 widget elements in for these fields: **Age**, **City**, and **Country**.
5. We have added another **<FieldGroup>** element onto the *Service* tab and moved 1 widget element in for this field: **Created**.

Next, we have modified the widget elements so that they meet the requirements:

1. We have added the **Type** attribute to the **Password** widget element and set its value to “*Password*”.
2. We have replaced the **<DateSelector>** widget element for the **Created** field with a **<TextBox>**. We do not want users to select the date in this widget, and we want this field to be read-only. That is why we have chosen the **<TextBox>** for displaying the value. We have also added the **Type** attribute to this element and set it to “*ReadOnly*”.
3. We have added another **<TextBox>** element above the **Created** field, set its **Type** attribute to “*ReadOnly*” and initialized it to the **Id** field available in the **<cms:bindings>**. The Id field is always added by C1 CMS to the data type and listed in the **<cms:bindings>**; however, it is never used in the **<cms:layout>** by default.

When the user adds a data item and opens this form, it will now look exactly as required.



LOGIN

Username: johnsmith

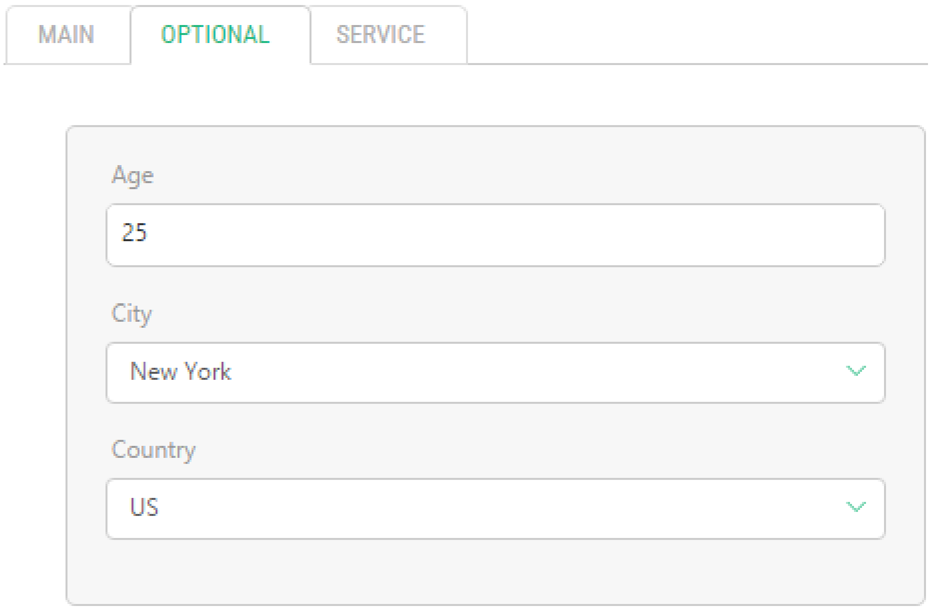
Password: ••••••••••••••••

USER INFORMATION

Name: John Smith

Email Address: john.smith@fabrikam.net

Figure 12: Customized data item editor form (Main tab)

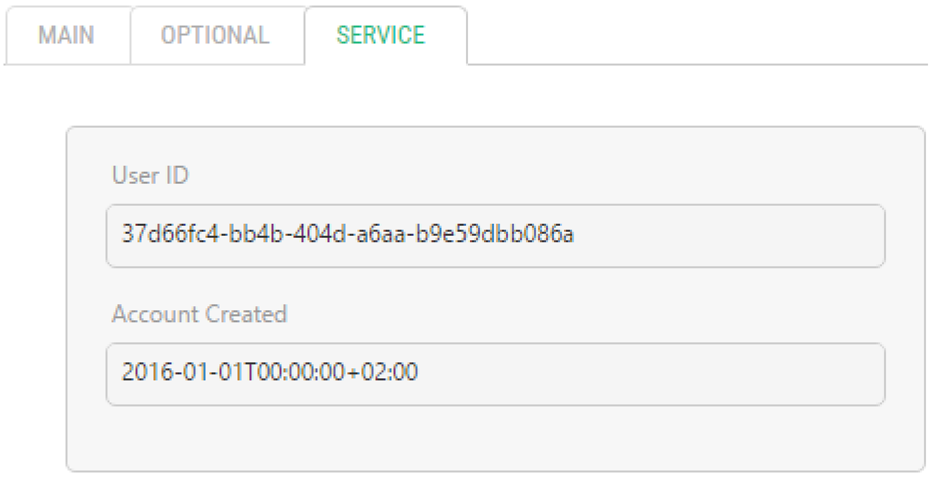


Age: 25

City: New York

Country: US

Figure 13: Customized data item editor form (Optional tab)



User ID: 37d66fc4-bb4b-404d-a6aa-b9e59dbb086a

Account Created: 2016-01-01T00:00:00+02:00

Figure 14: Customized data item editor form (Service tab)

6 Applying Validation Rules to Fields

A validation rule is a criterion to ensure that input data, for example, entered by a user in a form field is correct, that is, meets this criterion.

You can apply one or more validation rules to a data type field to control the user's input.

For example, you want to ensure that the user:

- Entered an email address in a corresponding field
- Entered a well-formed email address

In this case, you can apply two validation rules to the field:

- Checking that the field has a value (i.e. "is not null")
- Validating the string as a well-formed email address with a regular expression

C1 CMS comes with a predefined set of validation rules that you can apply to data type fields, too.

All the validation rules are available as functions in **Composite.Utils.Validation**.

Each type of field has its own set of validation rules.

6.1 Adding Validation Rule to Field

To add a validation rule to a field:

1. Edit a data type, and on the **Fields** tab, select a field to add a validation rule to.
2. On the **Advanced** tab, click **Add Validation Rules**. The **Field Validation Rules Configuration** window appears.

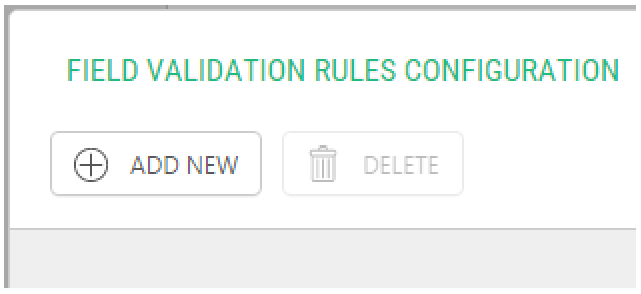


Figure 15: Field Validation Rules Configuration window

Note: If at least one validation rule has already been added to the field, the button will read "Edit Validation Rules".

3. Click **Add New**. The **Select Function** window appears.

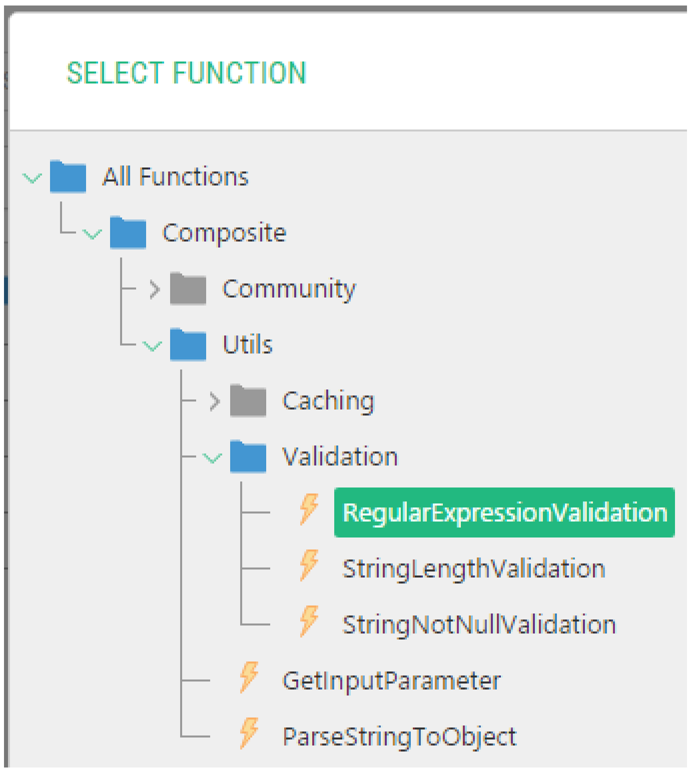
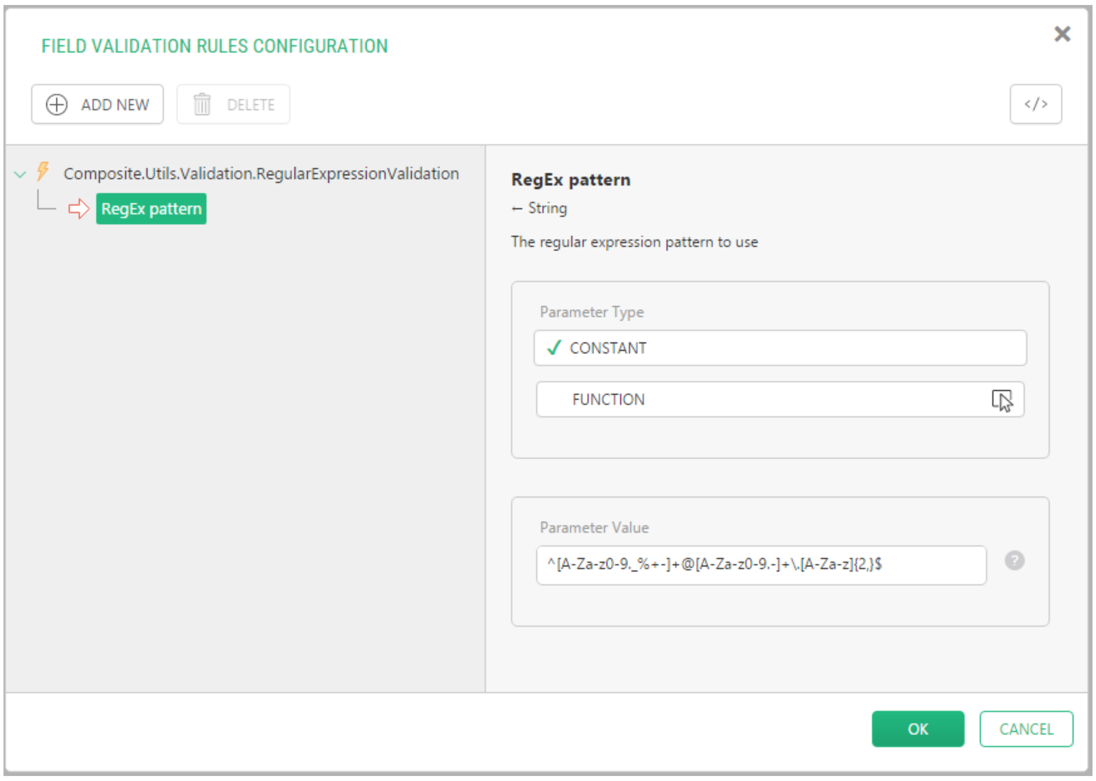


Figure 16: Select Function window

Note: If no validation rules have been previously added to the field, this window will pop up automatically after Step 3.

4. Expand **All Functions** > **Composite** > **Utils** > **Validation**, select the validation rule you want to use on the field and click **OK**.
5. If the validation rule requires that its parameters should be configured, enter the proper values.



Composite.Utils.Validation.RegularExpressionValidation
RegEx pattern

RegEx pattern
← String
The regular expression pattern to use

Parameter Type
✓ CONSTANT
FUNCTION

Parameter Value
`^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$`

Figure 17: Setting parameters of a validation rule

6. Click **OK** to save changes and close the **Field Validation Rules Configuration** window.

You have just added a validation rule to the field.

6.1.1 Editing Validation Rule

Note: You can only edit a validation rule that has parameters.

To edit a validation rule:

1. Edit a data type and on the **Fields** tab, select a field for which you want to edit a validation rule.
2. On the **Advanced** tab, click **Edit Validation Rules**. The **Field Validation Rules Configuration** window appears.
3. Select the validation rule you want to edit and modify its parameters where necessary.
4. Click **OK** to save changes and close the **Field Validation Rules Configuration** window.

You have just modified a validation rule for the field.

6.1.2 Deleting Validation Rule

To delete a validation rule:

1. Edit a data type and on the **Fields** tab, select a field from which you want to delete a validation rule.
2. On the **Advanced** tab, click **Edit Validation Rules**. The **Field Validation Rules Configuration** window appears.
3. Select the validation rule you want to delete and click **Delete**.

4. Click **OK** to save changes and close the **Field Validation Rules Configuration** window.

You have just deleted a validation rule from the field.

6.2 Overview of Validation Rules

Except the **Boolean** and **Data Reference** types, each field type has its own set of validation rules.

The **Boolean** type has no validation rules. The **Data Reference** type reuses the **String**'s validation rules.

Most data types has one common validation rule – the one that ensures that the value is not null.

6.2.1 DateTime

DateTimeNotNullValidation

This rule ensures that the input value has been entered and is therefore not null.

6.2.2 Decimal

DecimalNotNullValidation

This rule ensures that the input value has been entered and is therefore not null.

DecimalPrecisionValidation

This rule validates the precision of digits, that is, the number of decimals the user has specified). It has one required parameter:

MaxDigits

An Int32 value that specifies the maximum number of digits allowed on the decimal.

6.2.3 GUID

GuidNotNullValidation

This rule ensures that the input value has been entered and is therefore not null.

6.2.4 Int32

Int32NotNullValidation

This rule ensures that the input value has been entered and is therefore not null.

IntegerRangeValidation

This rule validates that an integer lies within a range of values specified in its two required parameters:

min

An Int32 value that specifies the minimum number allowed in this field.

max

An Int32 value that specifies the maximum number allowed in this field.

6.2.5 String and Data References

StringNotNullValidation

This rule ensures that the input value has been entered and is therefore not null.

RegularExpressionValidation

This rule validates that a string conforms to the regular expression specified in its required parameter:

Pattern

A string value that holds the regular expression pattern to use.

StringLengthValidation

This rule validates that the length of a string lies within a range specified in its two required parameters:

min

An Int32 value that specifies the minimum number of characters allowed in this field.

max

An Int32 value that specifies the maximum number of characters allowed in this field.

6.3 Example of Using Validation Rules

Let’s assume that you have a simple registration form with two fields: Name and Email Address.

You want the name the user should enter in the corresponding field to be no longer than 8 characters. You also want the email address to be well-formed.

You have a global data type called “Users” with the respective two string fields. Let’s see how you should meet the requirements by applying validation rules to both fields.

First, you have to limit the number of characters in the **Name** field to 8:

1. Edit a data type and on the **Fields** tab select the **Name** field.
2. On the **Advanced** tab, click **Add Validation Rules**.
3. In the **Select Function** window expand **All Functions** > **Composite** > **Utils** > **Validation**.
4. Select **StringLengthValidation** and click **OK**.
5. In the **Field Validation Rules Configuration** window, set the **Minimum length** parameter to *1* and the **Maximum length** parameter to *8* and click **OK**.

Note: If the **Select Function** window has not popped up over the **Field Validation Rules Configuration** window in Step 3, you should click **Add New**.

Next, you have to ensure that an email address entered is well-formed.

1. Select the **Email Address** field and on the **Advanced** tab, click **Add Validation Rules**.
2. In the **Select Function** window expand **All Functions** > **Composite** > **Utils** > **Validation**.
3. Select **RegularExpressionValidation** and click **OK**.

4. In the **Field Validation Rules Configuration** window, set the **RegEx pattern** parameter to "**^([a-zA-Z0-9_\-\.]+)@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,3})$**" (without quotation marks) and click **OK**.
5. Save the data type.

Now try to enter values in the Registration form by using the values that both meet and do not meet the requirements and see how the validation works.

6.4 Making Validation Messages on Forms User-Friendly

Note: The following information is applicable to the web forms generated by Forms Renderer.

The field validation on a form may fail because the value in the field does not meet requirements. You specify the requirements for the value when you either set properties of the field (for example, **Field type** or **Required**) or apply validation rules.

In both cases, if the validation fails, a validation error message appears on the form. By default, the Forms Renderer uses built-in system messages, which may be quite technical and not always appropriate on the form.

You can make the message user-friendly by replacing the system message with your own message. In this case, if the field validation fails, the user-friendly message will be displayed on the form instead.

When you specify the Help text for a field of a data type, the Forms Renderer uses this text for a validation message on a form.

To make a validation message for a field user-friendly:

1. Edit the data type used by the Forms Renderer.
2. Open the **Fields** tab and select a field under the **Datatype fields**.
3. Type some user-friendly text in the **Help** property.
4. Repeat Steps 2-3 for each field.
5. Save the data type.

7 Creating Data Types That Reference Other Data Types

One of the ways to make data entry error-proof and faster is to use lists of predefined values in fields instead of having users type their own values.

C1 CMS allows you to create data types with fields that reference other data types for a set of fixed values.

First, you should create a “referenced” data type, that is, a data type that will hold values for another data type.

Next, you should add data items to this data type. The data items will serve as values to select from in a field.

Finally, you should create or edit a “referencing” data type, that is, a data type that will use the values from the “referenced” data type, and add a special field for these values.

For example, you can have a data type for information about car makes and models (“Cars”). One of its fields (“Manufacturer”) stores the name of the company that makes a specific car.

When information about a car is added to the data type, instead of typing the company name in the field, the user selects the existing name from the drop-down list. And these company names are supplied by another data type.

In the following sections you will learn how to create such data types.

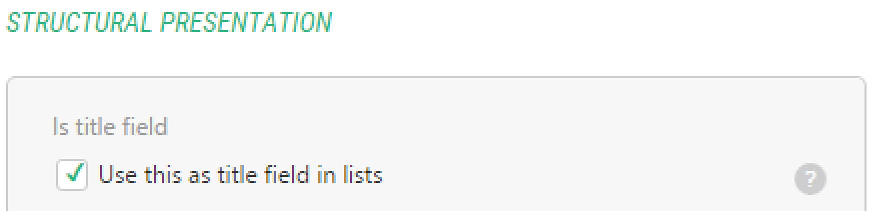
7.1 Creating Referenced Data Types

A referenced data type is the data type that supplies values for a Data Reference field in another data type.

The procedure of creating a referenced data type is the same as the general procedure for creating a data type. (Pease see *A Guide to Creating Data Types*.)

This data type may have only one field, the values of which will be used as values in a referencing data type.

If the referenced data type has more than one field, one of the fields will be used to provide values. This is normally a field set to be a “”title” field for a data type.


STRUCTURAL PRESENTATION

Is title field

✓ Use this as title field in lists

Figure 18: Assigning a title field

Once you have created the referenced data type, you should add data items to it.

7.2 Creating Referencing Data Types

A referencing data type is the data type that uses data items of another data type as values in one of its field. This field must be of the Data Reference type and its reference type must be set to the referenced data type.

To create a referencing data type:

1. Create or edit a data type (e.g. “Cars”).
2. Add a field of the Data Reference type (for example, “Manufacturer”).
3. In the **Reference Type** field, select the data type that will supply predefined values for this field.
4. Save the data type.

If this field is required, the **Selector** widget is used. If the field is optional, the **OptionalSelector** is used instead. The latter allows the user not to choose any value in this field. It appears on the drop-down list as the <NONE> option and is pre-selected by default.

When the user adds a data item to the data type, he or she will be able to choose a value from the selector. Each value is a separate data item in the referenced data type.

7.3 Grouping by Data Reference Fields

When using data types that reference other data types, you can create hierarchies of data items within these data types. These hierarchies can significantly help users in both locating existing items and adding new items.

Locating a data item in a hierarchy implies locating it within a specific group. Adding an item under a specific group within a data type initializes its values to those pre-defined by the group.

For example, the Cars data type might have many data items already added. In a “flat” view, each data item representing a car model will be placed immediately under the Cars data type in the Data navigator.

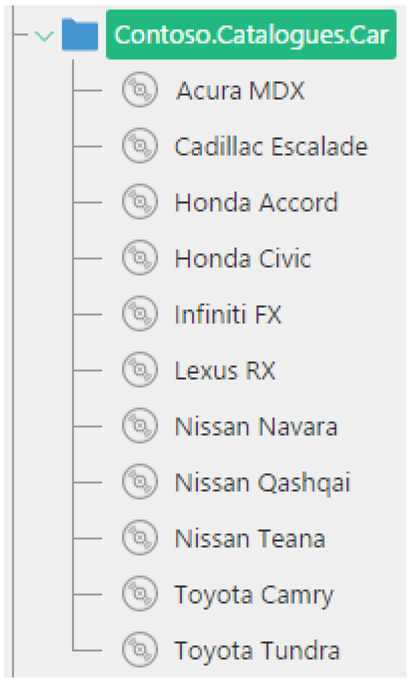


Figure 19: Data items listed in a “flat” view

You can however group the car models first by their manufacturers, and then by their brand names.

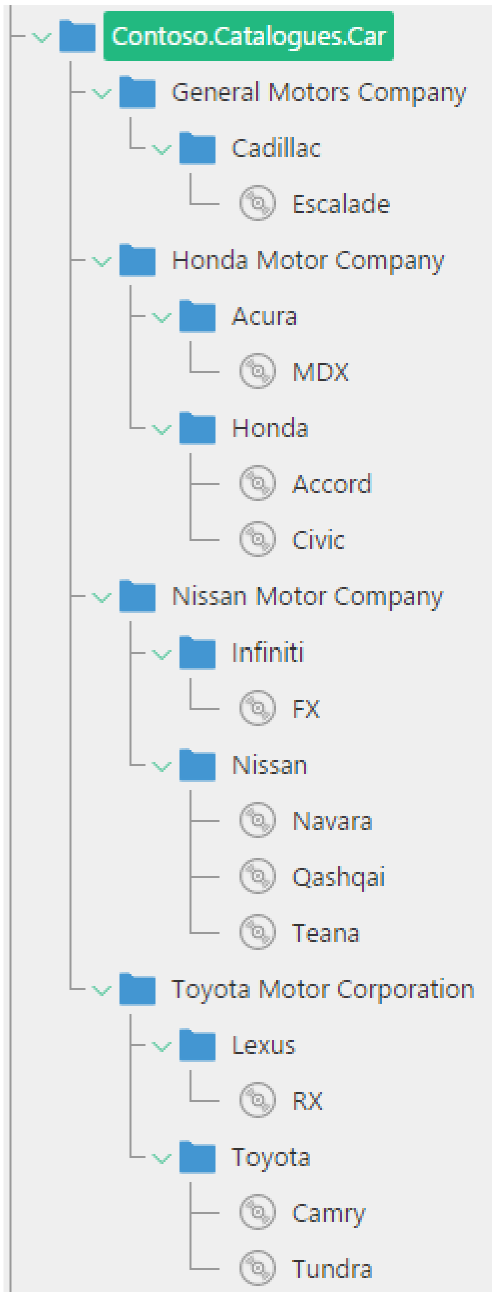


Figure 20: Data items listed hierarchically

To group data items within data types:

1. Edit a data type.
2. Select a field you want to serve as *Level 1* grouping category for data items. This should be a field that references a data type, which serves as a category or type for a data item. For example, in the Cars data type, it might be the Manufacturer field that references the Manufacturers data type.
3. On the **Advanced** tab, in the **Field grouping** dropdown list, select *Group by this field*.
4. Save the data type.

If you need more levels in the hierarchy, repeat the above procedure. When setting a field to serve as the *Level 2* grouping category, select *Group as 2. priority*. For example, in the Cars data type, it might be the Brand field that references the Brands data type.

8 Test Your Knowledge

8.1 TASK 1

1. Create a data type called “Contacts”
2. Add the following fields: *First Name*, *Last Nam*e, *Age*, *Marital Status*, *Street*, *City*, *Phone*, *Email*, *Website*.
3. Add a few items.
4. Localize the data type. Localize the data items where necessary.

8.2 TASK 2

1. Apply a validation rule on the *Age* field that allows integers between *21* and *75*.
2. Apply a validation rule on the *Email* field to allow only well-formed email addresses.
3. Apply a validation rule on the *Website* field to allow only well-formed URLs (starting with “*http://*” and including valid top-level domains such as “*.com*”)

8.3 TASK 3

1. Create another data type called “*Instant Messaging*”.
2. Add the field and name it “*Application*”.
3. Add the following data items to the data type: *ICQ*, *MSN*, *Skype*, *Yahoo*, *AIM*.
4. In the data type created in Task 1, add another field of the String type: *IM*.
5. Reference the *Instant Messaging* data type as its field type.

8.4 TASK 4

1. Edit the *Contacts* data type and select the *IM* field.
2. Change its type from *Data Reference* to *String*.
3. Replace the default *TextBox* widget with *DataIdMultiSelector*.
4. Set the *Data type to select from* property to the *Instant Messaging* data type.
5. Save the data type, add an item and see what widget is used for the *IM* field.

8.5 TASK 5

1. Edit the widget for the *IM* field again and set its *Compact UI* property to *Compact*.
2. Save the data type, add an item and see what widget is used for the IM field now.

8.6 TASK 6

1. Use the Forms Renderer add-on to create a web form based on the *Contact* data type.
2. Create.NET resource files for the form for the main language and another language.
3. Add strings to the resource files for form labels and help texts in a proper language.
4. Use the resource string in the data type's corresponding field properties.
5. Open the page with the form and see if the form's labels are in the main languages
6. Switch to the localized version of the website and see if the form's labels are in the other language.

8.7 TASK 7

1. Edit the form markup of the *Contacts* data type.
2. Place the *First Name*, *Last Name* fields in the *Name* field group and *Age* and *Marital Status* in the *Personal Information* field group.
3. Place these 4 fields on the *Person* tab.
4. Place the *Street* and *City* in the *Address* field group and *Phone*, *Email*, *Website*, *IM* in the *Communication* field group.
5. Place these 6 fields on the *Contact Information* tab.