Umbraco Contour 1.1

Developer Documentation

Per Ploug Hansen

8/3/2010

Contains information for developers working with and extending Umbraco Contour

Table of Contents

Introduction

These developer documents covers working with Umbraco Contour from a developer standpoint. It covers retrieving data from contour, either via XSLT or via the normal API, it shows how to extend the system by hooking into the provider model, and finally it describes the available events and workflows you can use to extend or integrate Contour.

This document is divided into 2 main parts. The first part consists of small walkthroughs to perform common tasks. The second part is the reference chapter, which covers the different elements in more detail and shows the more advanced options.

Revision History

- Version 1.1, August, 2010, Author: Per Ploug Hansen
    - Added information on the new 1.1 features
    - RecordAction
    - RecordSetAction
    - Import / export / template xml schema
- Version 1.0.1 November 20th 2009, author: Per Ploug Hansen
    - Added information on the workflow bracket syntax
- Version 1.0 November 13th 2009, author: Per Ploug Hansen

Contour XSLT Library

Umbraco Contour includes an XSLT Extension library which is accessible through the XSLT Editor in the developer section.

Record Xml Format

The Library contains a number of methods which returns records as XML in the below format

```
<?xml version="1.0" encoding="utf-8"?>
<uformrecords>
  <uformrecord>
    <state>Approved</state>
    <created>2009-11-13T10:01:55</created>
    <updated>2009-11-13T10:01:55</updated>
    <id>119ecc43-df79-46e1-9020-b2e27e239175</id>
    <ip>127.0.0.1</ip>
    <pageid>0</pageid>
    <memberkey></memberkey>
    <fields>

      <name record="119ecc43-df79-46e1-9020-b2e27e239175" sortorder="0">
        <key>2295187e-0345-4260-a406-eabcc1e774e2</key>
        <fieldKey>e6157c93-0b54-4415-b7ba-5c7c2c953b70</fieldKey>
        <caption>Name</caption>
        <datatype>String</datatype>
        <values>
          <value><![CDATA[My Name]]></value>
        </values>
      </name>

      <email record="119ecc43-df79-46e1-9020-b2e27e239175" sortorder="1">
        <key>a92875a8-938d-4ba0-990a-59a3518ce62c</key>
        <fieldKey>d8b10ffb-c437-4a44-8df6-01e6af5ac26f</fieldKey>
        <caption>Email</caption>
        <datatype>String</datatype>
        <values>
          <value><![CDATA[pph@testdomain.com]]></value>
        </values>
      </email>

    </fields>
  </uformrecord>
</uformrecords>
```

All record nodes are contained in a `<uformrecords>` element. All records consist of a `<uformrecord>` element with some meta data on it. The `<uformrecord>` contains a field element which contains a collection of nodes reflecting the form data fields.

The naming of the child elements inside the `<fields>` element are named accordingly to the caption of the field converted to lower case and with all foreign characters removed.

The element contains a `<values>` element which contains all entered values in individual `<value>` elements. A field can have multiple values, a checkboxlist for instance can save multiple values.

Sample XPath statements

These samples are provided as an introduction to working with Contour xml data. It does however follow the XPath standard and the above xml format can work with any valid XPath. The below snippets needs the standard umbraco xslt file to work, so simply create a new xslt file and insert the snippets on that file.

Get all records from the current page

```
<ul>
<xsl:for-each select="umbraco.contour:GetRecordsFromPage($currentPage/@id)//uformrecord">
<xsl:sort select="created" order="ascending"/>
  <li>
    A record with the state set to <xsl:value-of select="state"/>
    was created on <xsl:value-of select="umbraco.library:LongDate(created)"/>
  </li>
</xsl:for-each>
</ul>
```

To select all fields on a record

```
<xsl:for-each select="umbraco.contour:GetRecord($id)/uformrecord/fields/child::*">
    <xsl:sort select="caption" order="ascending"/>
    <h4>
        <xsl:value-of select=" caption"/>
    </h4>
</xsl:for-each>
```

To Select a field with a specific caption

```
<xsl:variable name="record" select="umbraco.contour:GetRecord($id)"/>

<xsl:variable name="email" select="$record/uformrecord/fields/child::* [caption = 'Email']"/>

or

<xsl:variable name="email" select="$record/uformrecord/fields/email"/>
```

Contour Provider Model

Most parts of Umbraco Contour uses a provider model, which makes it easy to add new parts to the application.

The model uses the notion that everything must have type to exist. The type defines the capabilities of the item. For instance a Textfield on a form has a FieldType, this particular field type enables it to render an input field and save simple text strings. The same goes for workflows, which has a workflow type, datasources which have datasource type and so on. Using the model you can seamlessly add new types and thereby extend the application.

In the current version it is possible to add new Field types, Data Source Types, Prevalue Source Types, Export Types, and Workflow Types.

Field Types

A field type handles rendering of the UI for a field in a form. It renders a standard asp.net webcontrol and is able to return a list of values when the form is saved.

Data Source Types

A data source type enables Contour to connect to a custom source of data. A datasource can consist of any kind of storage as long as it possible to return a list of fields Contour can map values to. For exemple: a Database data source can return a list of columns Contour can send data to, which enables Contour to map a form to a data source. A data source type is responsible for connecting to the storage, retrieving available fields and sending a record to the data source and then saving it.

Prevalue Source types

A prevalue source type can connect to a 3rd party storage and retrieve a collection of values which can be used on fields which support prevalues. The prevalue source is responsible for connecting to the source and retrieving the collection of values. A prevalue sourc e type can also implement edit capabilities so new items can be added/updated/deleted directly from the form editor.

Workflow types

A workflow can be executed each time a form changes state (when it is submitted for instance). A workflow is responsible for executing simple logic which can modify the record or notify 3rd party systems.

Export Types

Export types are responsible for turning form records (which are xml) into any other data format, which is then returned as a file.

Adding a type to the provider model

To add a new type, no matter if it's a workflow, field, data source, etc, there is a number of tasks to perform to connect to the Contour provider model. This chapter walks through each step and describes how each part works. This chapter will reference the creation of a workflow type. It is however the same process for all types.

Preparations

Create a new asp.net or class project in Visual Studio 2005/2008 add references to the Umbraco.Forms.Core.dll.

Adding the type to Contour

The Contour api contains a collection of classes that the provider model automaticly registers. So to add a new type to Contour you simply inherit from the right class. In the sample below we use the class for the workflow type.

```
public class Class1 : Umbraco.Forms.Core.WorkflowType
    {
        public override WorkflowExecutionStatus Execute(Umbraco.Forms.Core.Record record)
        {
            throw new NotImplementedException();
        }

        public override List<Exception> ValidateSettings()
        {
            throw new NotImplementedException();
        }
    }
```

When you implement this class you get two methods added. One of them is Execute which performs the execution of the workflow and the other is a method which validates the workflow settings, we will get back to these settings later on.

Even though we have the class inheritance in place, we still need to add a bit of default information.

Setting up basic type information

Even though we have the class inheritance in place, we still need to add a bit of default information. This information is added in the class's empty constructor like this:

```
public Class1() {
      this.Name = "The logging workflow";
      this.Id = new Guid("D6A2C406-CF89-11DE-B075-55B055D89593");
      this.Description = "This will save an entry to the log";
}
```

All three are mandatory and the ID must be unique, otherwise the type might conflict with an existing one.

Adding settings to a type

Now that we have a basic class setup, we would like to pass setting items to the type. So we can reuse the type on multiple items but with different settings. To add a setting to a type, we simply add a property to the class, and give it a specific attribute like this:

```
[Umbraco.Forms.Core.Attributes.Setting("Log Header",
                        description = "Log item header",
                        control = "Umbraco.Forms.Core.FieldSetting.TextField")]
public string LogHeader { get; set; }
```

The `Umbraco.Forms.Core.Attributes.Setting` registers the property in Umbraco Contour and there will automatically be UI and storage generated for it. In the attribute a name, description and the control to be rendered is defined.

With the attribute in place, the property value is set every time the class is instantiated by Umbraco Contour. This means you can use the property in your code like this:

```
[Umbraco.Forms.Core.Attributes.Setting("Document ID",
          description = "Node the log entry belongs to",
          control = "Umbraco.Forms.Core.FieldSetting.Pickers.Content")]
public string document { get; set; }

public override Enums.WorkflowExecutionStatus Execute(Record record)
{
   Log.Add(LogTypes.Debug, int.Parse(document), "record submitted from: " + record.IP);
}
```

For all types that uses the provider model, settings work this way. By adding the Setting attribute Contour automatically registers the property in the UI and sets the value when the class is instantiated.

Validating type settings with ValidateSettings()

The `ValidateSettings()` method which can be found on all types supporting dynamic settings, is used for making sure the data entered by the user is valid and works with the type.

```
public override List<Exception> ValidateSettings()
        {
            List<Exception> exceptions = new List<Exception>();

            int docId = 0;
            if (!int.TryParse(document, out docId))
                exceptions.Add(new Exception("Document is not a valid integer"));

            return exceptions;
        }
```

Registering the class with Umbraco and Contour

Finally compile the project and copy the .dll to your website /bin folder or copy the .cs file to the app_code directory. The website will now restart and your type will be registered automatically, no configuration

needed. Also look in the reference chapter for complete class implementations of workflows, fields and export types

Adding a field type to Umbraco Contour

This builds on the "adding a type to the provider model" chapter

Add a new class to the visual studio solution and make it inherit from `Umbraco.Forms.Core.FieldType` and override the `Editor` property.

In the empty constructor add the following information:

```
public Textfield() {
            //Provider
            this.Id = new Guid("D6A2C406-CF89-11DE-B075-55B055D89593 ");
            this.Name = "Textfield";
            this.Description = "Renders a html input fieldKey";

            //FieldType
            this.Icon = "textfield.png";
            this.DataType = FieldDataType.String;
        }
```

In the constructor we specify the standard provider information (remember to set the ID to a unique ID)

And then we set the field type specific information. In this case a preview Icon for the form builder UI and what kind of data it will return, this can either be `string, longstring,integer,datetime` or `boolean.`

Then we will start building the editor and the values it returns

```
public System.Web.UI.WebControls.TextBox tb;
public List<Object> _value;

public override WebControl Editor
        {
            get
            {
                tb.TextMode = System.Web.UI.WebControls.TextBoxMode.SingleLine;
                tb.CssClass = "text";

                if (_value.Count > 0)
                    tb.Text = _value[0].ToString();

                return tb;
            }
            set
            {
                base.Editor = value;
            }
        }
```

The editor simply takes care of generating the UI control and setting its value. The `List<object>` is what is later returned by the field type.

The reference chapter contains the full class implementation

Adding settings to field types

New in Contour 1.1, is the option to add settings to a field type, please follow the chapter on adding settings to types to see sample code on setting types

Adding a workflow type to Umbraco Contour

This builds on the "adding a type to the provider model" chapter.

Add a new class to your project and have it inherit from `Umbraco.Forms.Core.WorkflowType`, implement the class. For this sample we will focus on the `execute` method. This method process the current record (the data submitted by the form) and have the ability to change data and state.

```
public override WorkflowExecutionStatus Execute(Record record, RecordEventArgs e)
{
//first we log it
Log.Add(LogTypes.Debug, -1, "the IP " + record.IP + " has submitted a record");

//we can then iterate through the fields
foreach(RecordField rf in record.RecordFields.Values){
   //and we can then do something with the collection of values on each field
   List<object> vals = rf.Values;

   //or just get it as a string
   rf.ValuesAsString();
}

//If we altered a field, we can save it using the record storage
Umbraco.Forms.Data.Storage.RecordStorage store = new RecordStorage();
store.UpdateRecord(record, e.Form);
store.Dispose();


//we then invoke the recordservice which handles all record states
//and make the service delete the record.
Umbraco.Forms.Core.Services.RecordService rs = new RecordService(record);
rs.Delete();
rs.Dispose();

return WorkflowExecutionStatus.Completed;}
```

The `Execute()` method gets a `Record` and a `RecordEventArgs` argument. These 2 arguments contains all information related to the workflow. The record contains all data and meta data submitted by the form. The `RecordEventArgs` contains references to what form the record is from, what state it is in and a reference to the current `HttpContext`

The sample above uses 2 different areas to work with the record. The first is the `RecordStorage` class which handles all storage of the record data. The second is the `RecordService`, this handles record state changes and record events. Both the `RecordService` and `RecordStorage` is described in detail in the reference chapter.

The reference chapter contains the full class implementation

Record and Recordset actions

This functionality is only available in Contour 1.1 and above

A "Record action" is a utillity method you can execute in the context of a record after it has been submitted and stored. This means you can add additional options for processing a Record or a Collection of records (a Recordset)

This functionaly uses the same plugin model as the rest of Contour, so it is truly easy to extend the entries viewer UI with your own tools.

Sample Record Action

For this quick intro, we will setup a simple action that will convert the Record to XML and then set it to an email address. This is done from the entries viewer in contour.

Setting up the action type

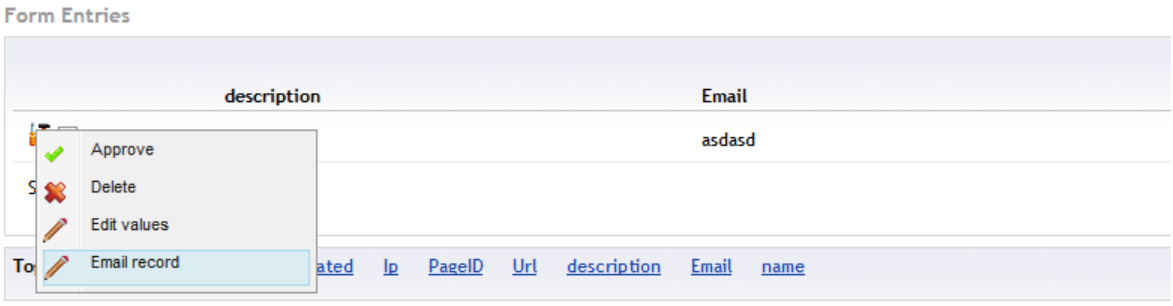
Like everything else in Contour, all logic is defined in a type. To get more in-depth information on types, please read the chapter "adding a type to the provider model"

```
public class SendToEmail : RecordActionType
{
public SendToEmail()
{
    this.Description = "Sends the record data to an email addresse of your choice";
    this.Icon = "edit.png";
    this.Id = new Guid("cdb3fa20-9e2f-11df-981c-0800200c9a66");
    this.Name = "Email record";
}

public override Enums.RecordActionStatus Execute(Record record, Form form)
{
    string Email = "my@email.com";
    string recordXML = record.ToXml(new System.Xml.XmlDocument()).OuterXml;
    string sender = umbraco.UmbracoSettings.NotificationEmailSender;

    umbraco.library.SendMail(sender, Email, "Contour Record", recordXML, false);
    return Enums.RecordActionStatus.Completed;
}
}
```

This adds a new item to the record context menu on the entries viewer like so:

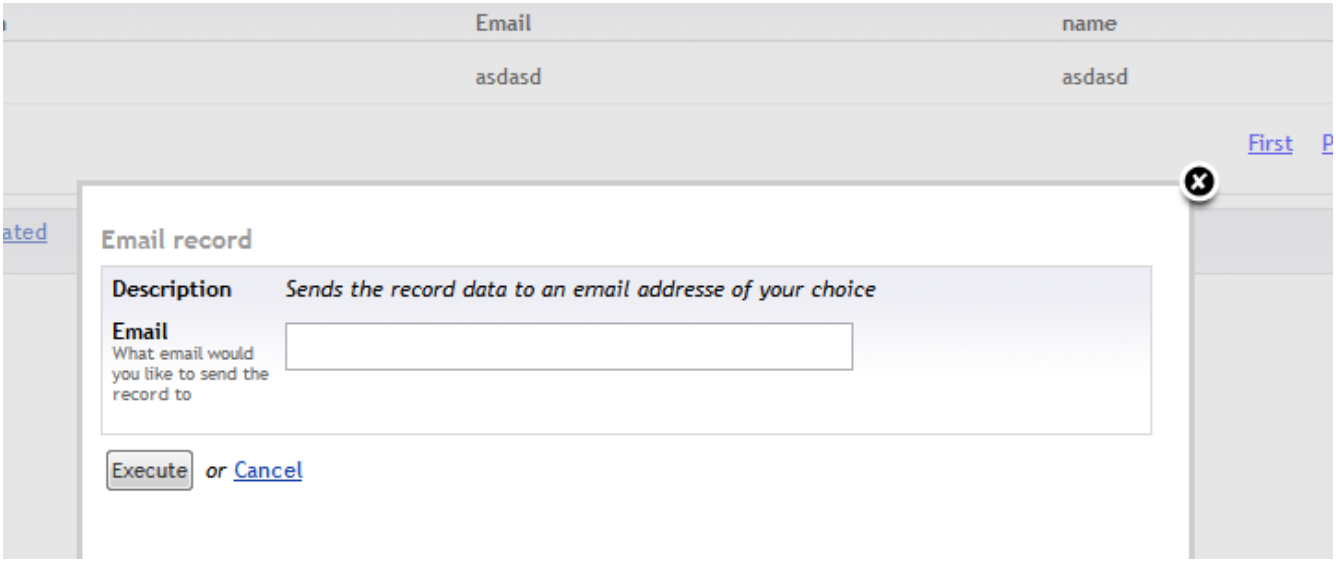


When the item is clicked, the `Execute()`method is run with the record and form as parameters.

If you wish, you can also add settings to these types, making them more flexible and reusable, adding a setting type like this to the class:

```
[Attributes.Setting("Email", description = "What email would you like to send the record to",
control = "Umbraco.Forms.Core.FieldSetting.TextField")]
public string Email { get; set; }
```

Will open a dialog like this instead of just executing the code right away:

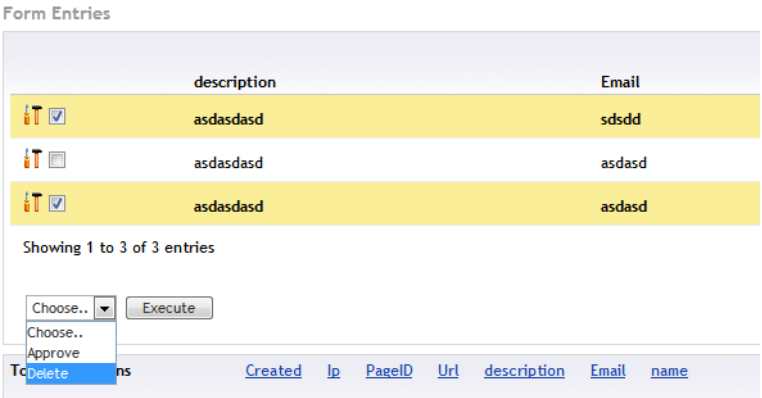


The public property "Email" can then be used in the `execute()` method as Contour will load these properties before executing.

Sample Recordset action

Recordset actions are almost the same as a Record action. They are implemented the same way and have the same options in regard to settings and customization.

The only difference is that they can process an entire collection of selected records from the entries viewer, so they are available from a dropdown box when records are selected in the entries viewer:



A sample for deleting the collection of selected records in the entries viewer:

```
public override Enums.RecordActionStatus Execute(List<Record> records, Form form)
{
    RecordService s = new RecordService(form);
    foreach (Record r in records)
    {
            s.Record = r;
            s.Delete();
    }
    s.Dispose();
    return Enums.RecordActionStatus.Completed;
}
```

Adding form templates to Contour

Contour 1.1 comes with a complete XML schema to represent a form. This XML can be exported and imported as .ucf files which contains a full XML representation of the form.

However, these .ucf files can also be placed in the form templates directory to make them reusable by end-users.

Adding an existing form as a template

Export the form you wish to use as a template, this will download a .ucf file to your local machine.

Copy or ftp the file to the **/umbraco/plugins/umbracoContour/templates/forms** directory

There is nothing that needs to be renamed or modified.

Right click the forms folder in the contour tree and select create, your form should now be available as a template.

Enabling advanced Macro properties

When installing Umbraco Contour, it by default only enables seleting a form, when inserting a macro. The macro does however a couple of other options to configure if needed.

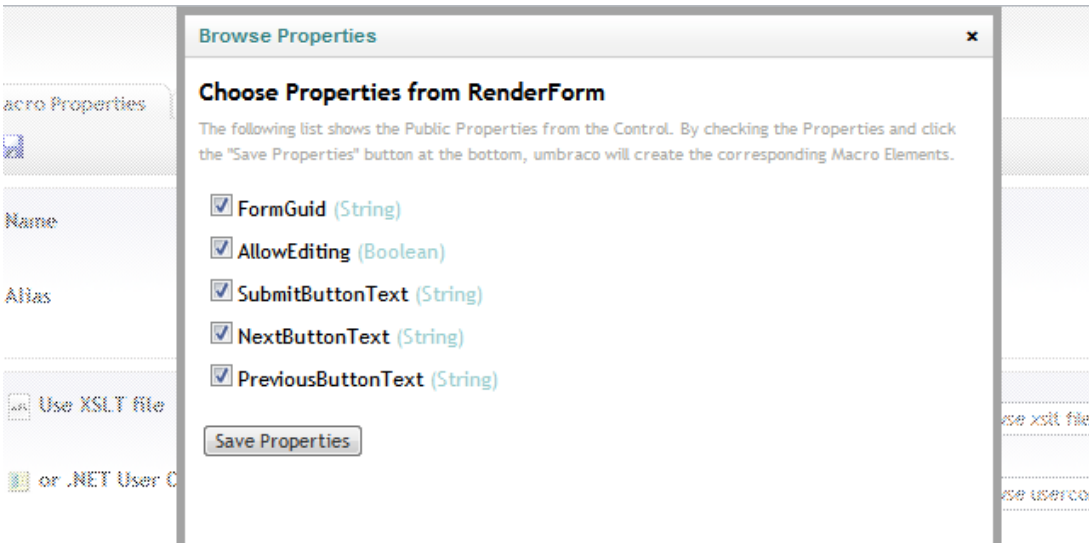
Available Macro properties

Name	Description
FormGuid	Sets the guid of the form to render, expects a valid GUID
AllowEditing	Enables the macro to open previously saved records and edit them, by default set to FALSE, expects a boolean value
SubmitButtonText	Override the text on the submit button, expects a string
NextButtonText	Override the text on the Next button, expects a string
PreviousButtonText	Override the text on the Previous button. expects a string

If the button texts are not set, Contour will use either #submit, #next, #prev to set the value. these will use dictionary items if available or fall-back to an english translation.

To enable these additional parameters, go to the developer section in Umbraco, expand the "macros" folder, and select the "insert macro from umbraco contour"

On the macro edit screen, click "Browse properties" you will then be presented with this screen which add the selected the properties when you click "save properties"



Reference

List of classes, interfaces, UI controls available in the current version

Umbraco Contour Library methods

Umbraco Contour includes a libary for easy access to record data in the xml format. The library is located in the class `Umbraco.Forms.Library`[1]

GetApprovedRecordsFromPage

```
XPathNodeIterator GetApprovedRecordsFromPage(int pageId)
```

Returns All records with the state set to approved from all forms on the umbraco page with the id = pageId as a `XPathNodeIterator`

GetApprovedRecordsFromFormOnPage

```
XPathNodeIterator GetApprovedRecordsFromFormOnPage(int pageId, string formId)
```

Returns All records with the state set to approved from the form with the id = formId on the umbraco page with the id = pageId as a `XPathNodeIterator`

GetRecordsFromPage

```
XPathNodeIterator GetRecordsFromPage(int pageId)
```

Returns All records from all forms on the umbraco page with the id = pageId as a `XPathNodeIterator`

GetRecordsFromFormOnPage

```
XPathNodeIterator GetRecordsFromFormOnPage(int pageId, string formId)
```

Returns All records from the form with the id = formId on the umbraco page with the id = pageId as a `XPathNodeIterator`

GetRecordsFromForm

```
XPathNodeIterator GetRecordsFromForm(string formId)
```

Returns All records from the form with the ID = formId as a `XPathNodeIterator`

GetRecord

```
XPathNodeIterator GetRecord(string recordId)
```

Returns the specific record with the ID = recordId as a `XPathNodeIterator`

[1] Umbraco.Forms is the internal class naming for Umbraco Contour

Bracket syntax for workflow settings

When adding settings to a workflow, it can be handy to map record values to the setting fields. This can be done using the build-in bracket syntax, and also by using the standard umbraco syntax for adding page, session and cookie values.

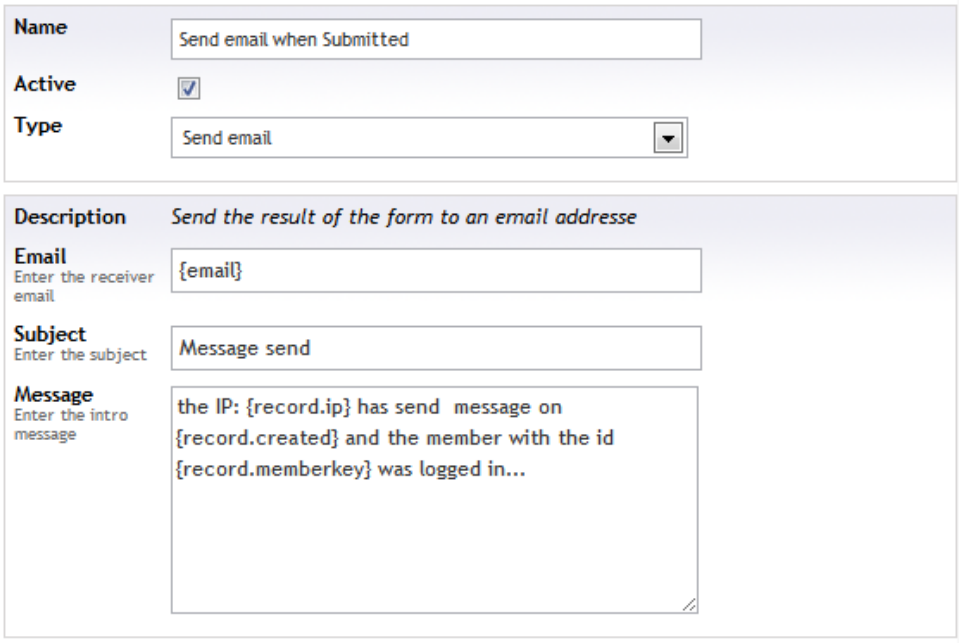
Record values syntax

Individual fields can be referenced by their caption in lowercase and with whitespace and special characters removed:

Sample field caption	Bracket syntax
Email	{email}
Enter your employment ID	{enteryouremploymentid}
First & last name	{firstlastname}
Your #1 job	{your1job}

it is also possible to reference core record attributes like current member id, when the record was created, page id, etc. These are all prefixed with "record."[2]

Record attribute name	Bracket syntax
Creation Date	{record.created}
Update Date	{record.updated}
Member Id	{record.memberkey}
Page Id	{record.umbracopageid}
Client IP	{record.ip}



[2] Notice the record attributes was added to version 1.0.5 of Umbraco Contour so was not available in 1.0 or any of the beta releases

Page, Session and cookie values syntax

To insert values from the current http context like request collection data, cookie values or values from the current umbraco page, we can use the standard umbraco bracket syntax. This is detailed on the umbraco wiki

Type of value	Bracket syntax	Example
Insert page value	[#propertyAlias]	[#bodyText]
Insert recursive page value	[$propertyAlias]	[$metaDescription]
Insert cookie value	[%cookieValue]	[%sessionKey]
Insert value from request collection	[@requestKey]	[@formField]

The provider model

Inheritable classes for the provider model

Classes the provider model automatically registers.

Type	Description	Class
DataSource Type	Adds a new datasource type	`Umbraco.Forms.Core.DataSourceType`
Export Type	Adds a new export type	`Umbraco.Forms.Core.ExportType`
Workflow Type	Adds a new workflow type	`Umbraco.Forms.Core.WorkflowType`
Field Type	Adds a new field type	`Umbraco.Forms.Core.FieldType`
Prevalue Source Type	Adds a new prevalue source type	`Umbraco.Forms.Core.PrevalueSourceType`
Record Action Type	Adds a new Record Action*	`Umbraco.Forms.Core.RecordActionType`
RecordSet Action Type	Adds a new RecordSet Action*	`Umbraco.Forms.Core.RecordsetActionType`

The classes are used to register new types in Umbraco Contour. To use, simply inherit the class and set some meta information.

```
namespace ClassLibrary1
{
    public class Class1 : Umbraco.Forms.Core.WorkflowType
    {
        public Class1() {
            this.Name = "My new workflow type";
            this.Id = new Guid("D6A2C406-CF89-11DE-B075-55B055D89593");
            this.Description = "This is a sample";
        }
    }
}
```

A complete fieldtype class

The below class shows a complete field type implemention using the `FieldType` class. This is the sourcecode of the `Fieldtype` "Textfield" included in Umbraco Contour.

```
public class Textfield : FieldType
    {
        public System.Web.UI.WebControls.TextBox tb;
        public List<Object> _value;

        public Textfield() {
            //Provider
            this.Id = new Guid("3F92E01B-29E2-4a30-BF33-9DF5580ED52D");
            this.Name = "Textfield";
            this.Description = "Renders a html input fieldKey";

            this.Icon = "textfield.png";
            this.DataType = FieldDataType.String;

            tb = new TextBox();
            _value = new List<object>();
        }

        public override WebControl Editor {
            get{
                tb.TextMode = System.Web.UI.WebControls.TextBoxMode.SingleLine;
                tb.CssClass = "text";

                if (_value.Count > 0)
                    tb.Text = _value[0].ToString();

                return tb; }
            set{
                base.Editor = value;
            }
        }


        public override List<Object> Values{
            get{
                if (tb.Text != "") {
                    _value.Clear();
                    _value.Add(tb.Text);
                }
                return _value;
            }
            set{
                _value = value;
            }
        }

        public override string RenderPreview(){
            return "<input type=\"text\" class=\"textfield\" />";
        }

        public override string RenderPreviewWithPrevalues(List<object> prevalues){
            return RenderPreview();
        }


        public override bool SupportsRegex {
            get{return true;}
        }
```

A complete workflow type class

Shows a complete workflow type implementation using the WorkflowType class. This is the actual source code of the "perform filtering" workflow included in Umbraco Contour.

```
public class ChangeRecordState : WorkflowType
    {
        [Attributes.Setting("Dirty Words",
          description = "Comma seperated list of forbidden words",
          control = "Umbraco.Forms.Core.FieldSetting.TextField")]
        public string DirtyWords { get; set; }

        [Attributes.Setting("Action", prevalues = "Delete Record,Approve Record",
          description = "What to do if it matches",
          control = "Umbraco.Forms.Core.FieldSetting.Dropdownlist")]
        public string Action { get; set; }

        public ChangeRecordState() {
            this.Id = new Guid("4C40A092-0CB5-481d-96A7-A02D8E7CDB2H");
            this.Name = "Perform filtering";
            this.Description = "Changes the state of the record being processed";
        }

        public override List<Exception> ValidateSettings() {
            return new List<Exception>();
        }

        public override WorkflowExecutionStatus Execute(Record record, RecordEventArgs e) {
            string content = "";
            string[] words = DirtyWords.Split(',');
            bool dirty = false;

            foreach (RecordField rf in record.RecordFields.Values){
                content += rf.ValuesAsString();
            }

            foreach(string s in words) {
                if (content.Contains(s)) {
                    dirty = true;
                    break;
                }
            }

            if (dirty) {
                Services.RecordService rs = new Services.RecordService(record);
                if (Action == "Delete Record")
                    rs.Delete();
                else
                    rs.Approve();
                rs.Dispose();
            }

            return WorkflowExecutionStatus.Completed;
        }
    }
```

Setting Types

UI components for adding setting controls to workflow,datas ource,prevalue and export types. Notice Field Types does not currently support setting components. The settings types are used as attributes on a public string property. This will auto generate the UI and set the property automatically every time the type is instantiated.

```
 [Umbraco.Forms.Core.Attributes.Setting("Method",
          description = "POST or GET",
          prevalues = "POST,GET,PUT,DELETE",
          control = "Umbraco.Forms.Core.FieldSetting.Dropdownlist")
]
public string Method { get; set; }
```

Parameter	Description
Name	Label text for the UI
Description	Tooltip information for the UI
Prevalues	Contains a comma-seperated list of prevalues
control	Contains the full class name of the control to render

Available field setting types

The below classes are the UI controls currently available in Umbraco Contour. These can only be used for setting property values of the Type "string"

Description	Class
Renders a checkbox	`Umbraco.Forms.Core.FieldSetting.Checkbox`
Renders a UI component for mapping a record to an umbraco document type	`Umbraco.Forms.Core.FieldSetting.DocumentMapper`
Renders a dropdownlist	`Umbraco.Forms.Core.FieldSetting.Dropdownlist`
Renders a UI component for mapping a record to named fields	`Umbraco.Forms.Core.FieldSetting.FieldMapper`
Renders a file upload	`Umbraco.Forms.Core.FieldSetting.File`
Render a password field	`Umbraco.Forms.Core.FieldSetting.Password`
Renders a text area	`Umbraco.Forms.Core.FieldSetting.TextArea`
Renders a input field	`Umbraco.Forms.Core.FieldSetting.TextField`
Renders a umbraco content picker	`Umbraco.Forms.Core.FieldSetting.Pickers.Content`
Renders a content picker with a XPath search field*	`Umbraco.Forms.Core.FieldSetting.Pickers.ContentWithXPath`
Renders an list of umbraco datatypes	`Umbraco.Forms.Core.FieldSetting.Pickers.DataType`
Renders a list of umbraco document types	`Umbraco.Forms.Core.FieldSetting.Pickers.DocumentType`
Renders a list of umbraco media types	`Umbraco.Forms.Core.FieldSetting.Pickers.MediaType`
Renders a list of umbraco member groups	`Umbraco.Forms.Core.FieldSetting.Pickers.MemberGroup`
Renders a list of umbraco member types	`Umbraco.Forms.Core.FieldSetting.Pickers.MemberType`

The RecordService

`Umbraco.Forms.Core.Services.RecordService` handles changing record states, starting workflows and triggering events on a record. So to trigger any events on a record, it is not enough to change the record's state property. Instead the change must be triggered through the `RecordService`.

The `RecordService` is instantiated using either a reference to a Form, a Record or both.

```
//Instantiate using a record reference
Umbraco.Forms.Core.Services.RecordService rs = new
Umbraco.Forms.Core.Services.RecordService(record);

//perform state changes
rs.Approve();

//Dispose of the service object
rs.Dispose();
```

Record-State changing methods

State	Description
`RecordService.Open()`	Triggers the open event, and workflows that reacts on "Opened"
`RecordService.Resume()`	Triggers the resume event, and workflows that reacts on "Resumed"
`RecordService.NextPage()`	Triggers the open event, and workflows that reacts on "Partially Submitted"
`RecordService.PreviousPage()`	Does not trigger any events or workflows
`RecordService.Submit()`	Triggers the submit event, and workflows that reacts on "Submitted"
`RecordService.Approve()`	Triggers the approve event, and workflows that reacts on "Approved"
`RecordService.Delete()`	Triggers the delete event, and workflows that reacts on "Deleted"
`RecordService.Dispose()`	Disposes of the record service and releases all resources

Available Record Events

The record service also exposes static traditional .net events which can be subscribed to like normal events. The events triggered by record changes are below.

Event	Description
`RecordService.RecordOpened`	Triggers when a form is displayed the first time
`RecordService.RecordPartiallySubmitted`	Trigges when user goes from one form step to another
`RecordService.RecordResumed`	Triggers when the form is resumed after inactivity
`RecordService.RecordSubmitted`	Triggers when a form is submitted
`RecordService.RecordApproved`	Triggers when the record is approved
`RecordService.RecordDeleted`	Triggers when the record is deleted

Subscribing to record events using umbraco.buinesslogic.ApplicationBase

Umbraco has build in support for connecting to events without any extra work: simply inherit from the `ApplicationBase` class.

```
using Umbraco.Forms.Core;
using Umbraco.Forms.Core.Services;
using umbraco.BusinessLogic;
public class Class1 : umbraco.BusinessLogic.ApplicationBase  {
    public Class1(){
        RecordService.RecordApproved += new EventHandler<RecordEventArgs>(RecordApproved);
    }
    void RecordApproved(object sender, Umbraco.Forms.Core.RecordEventArgs e) {
        Record r = (Record)sender;
        Log.Add(LogTypes.Debug, r.UmbracoPageId, "record submitted by " + r.IP);
    }
}
```

The RecordStorage and RecordsViewer

The Umbraco.Forms.Data.Storage.RecordStorage provides crudding capabilities on the Record object.

Instantiating the record storage

```
//either connect to the current storage
RecordStorage rs = new Umbraco.Forms.Data.Storage.RecordStorage();

//or connect to a remote storage using a connection string
ISqlHelper sqlhelper = umbraco.DataLayer.DataLayerHelper.CreateSqlHelper( connectionstring);
RecordStorage remoteRs = new Umbraco.Forms.Data.Storage.RecordStorage(sqlhelper);
```

All storage classes are disposable so remember to call Dispose() after use.

Import / Export / Form Templates XML schema

Contour has a unified XML schema to describe a form and its fields. This XML Schema is used for exporting, importing and storing templates of forms. The schema follows the class structure of the Umbraco.Forms.Core.Form class:

```
<?xml version="1.0" encoding="utf-8"?>
<Form>
    <Name>sdasda</Name>
    <Created>2010-08-02T13:58:07.25</Created>
    <FieldIndicationType>NoIndicator</FieldIndicationType>
    <Indicator />
    <ShowValidationSummary>false</ShowValidationSummary>
    <HideFieldValidation>false</HideFieldValidation>
    <RequiredErrorMessage>{0} is mandatory</RequiredErrorMessage>
    <InvalidErrorMessage>{0} is not valid</InvalidErrorMessage>
    <MessageOnSubmit>sdasdasdasd</MessageOnSubmit>
    <GoToPageOnSubmit>0</GoToPageOnSubmit>
    <ManualApproval>false</ManualApproval>
    <Archived>false</Archived>
    <StoreRecordsLocally>true</StoreRecordsLocally>
    <DisableDefaultStylesheet>false</DisableDefaultStylesheet>
    <Pages>
        <Page>
            <FieldSets>
                <FieldSet>
                    <Fields>
                        <Field>
                            <PreValues />
                            <Caption>description</Caption>
                            <ToolTip />
                            <SortOrder>2</SortOrder>
                            <PageIndex>0</PageIndex>
                            <FieldsetIndex>0</FieldsetIndex>
                            <Id>00000000-0000-0000-0000-000000000000</Id>
                            <FieldSet>e075574b-47fd-453a-9b54-2fd7025e3b95</FieldSet>
                            <Form>19891658-5d4a-4f29-8606-6a8ad89cc58a</Form>
                            <FieldTypeId>3f92e01b-29e2-4a30-bf33-9df5580ed52c</FieldTypeId>
                            <Mandatory>false</Mandatory>
                            <RegEx />
                            <RequiredErrorMessage />
                            <InvalidErrorMessage />
                            <PreValueSourceId></PreValueSourceId>
                            <Settings />
                        </Field>
                    </Fields>
                    <Caption>sdasda</Caption>
                    <SortOrder>0</SortOrder>
                    <Id>00000000-0000-0000-0000-000000000000</Id>
                    <Page>8ea44696-608e-45c9-b59d-a4c302d551bc</Page>
                </FieldSet>
            </FieldSets>
            <Caption>sdasda</Caption>
            <SortOrder>0</SortOrder>
            <Id>00000000-0000-0000-0000-000000000000</Id>
            <Form>19891658-5d4a-4f29-8606-6a8ad89cc58a</Form>
        </Page>
    </Pages>
    <DataSource>00000000-0000-0000-0000-000000000000</DataSource>
    <Id>19891658-5d4a-4f29-8606-6a8ad89cc58a</Id>
</Form>
```