Forms module
Setup Guide

CodeDroids Forms module for OpenCms
Version 1.0

Forms module Setup Guide

CodeDroids ApS
Møsvråvej 86
DK-6051 Almind
Denmark
support@codedroids.com

For more information please visit: http://www.codedroids.com

For more information on OpenCms, please visit http://www.opencms.org

Table of Contents

1 Introduction

The Forms module provides a number of different functions.

To major aspects are the forms-handling and the persistence-handling.

Forms

The forms-handling is intended to put an end to the numerous getParameter()-calls plus hacking HTML form-stuff. Instead one defines a XML file specifying what fields the form should include and then let the module render the appropriate HTML and, upon the user submits the form, handle the request parameters and set the values on the form.

The focus of the development has been the typical types of forms that you have on web-sites, such as contact forms and the like. I.e fairly “flat” data structures – something that typically would be put into en email or pushed to a single table in the database.

The forms-handling is moderately extensible as new renders can be plugged in.

Persistence

In case the values collected by the form need to be stored in a database, the included persistence layer will ease this as by providing more or less “code-free” persistence (once you have described the database entity in an XML file).

While similar in concept it is certainly not a Hibernate/Cayenne/TopLink/Enterprise Objects (insert your own ORM favorite here) replacement. Again the focus has been on simple data structures. It does not support relations, so you will have to manage them by hand. But still, compared to fiddling with JDBC and SQL, this simple persistence layer is still a time saver.

Examples

The module includes a couple of JSP that are functional and provides an easy way to work with forms, but they also serve as example on how to use the above mentioned functions.

2 Usage

We first describe how to use the two JSP's that are included in the module, then we will look at using the forms and persistence API.

But first things first, you need to upload the module to OpenCms through the Module Administration, and then restart the application instance.

Once that is done you are ready to start using it.

2.1 Form.jsp

This JSP-file is intended to be included by another JSP-file stored in OpenCms' virtual filesystem. This could be an OpenCms template, or a simple JSP as the one shown here:

```
<%@ page session="false" %>
<%@ taglib prefix="cms" uri="http://www.opencms.org/taglib/cms" %>
<html>
  <head>
    <title>Form.jsp test</title>
  </head>
  <body>
    <cms:include file=”/system/modules/com.codedroids.oc.forms/elements/
Form.jsp” />
  </body>
</html>
```

Obviously you would have quite a lot more HTML markup in a real template-file, and in case that you make a JSP to be accessible on the homepage, you would probably include the head and foot from some other template.

Anyway, here we assume that you have saved this JSP in the root of the site under the name **register.jsp**

If you try display the page you will get an exception telling you that you must set the Form-property. The Form.jsp reads a number of properties from the page-file (the JSP file in this example or the XMLPage in case the Form.jsp is used from a template).

Here is a list of the properties that the Form.jsp reads:

Property	**Description**
Form	XML file in VFS that defines the form.
FormIdent	If given overrides the one from the form definition, if it is not defined there either an exception is thrown
FinishPage	Where to go when finished succesfully. If FinishTemplate/FinishFormat/FinishForm is set then this automated forwarding is suppressed otherwise their output would not be shown
FinishForm	Values: "hidden", "visible" or "readonly". Determines if the form is shown once finished or it is hidden. If readonly then it is shown but the fields are disabled. If not set it acts like "hidden" but differs as this won't suppress forwarding to the FinishPage in case that is defined as setting the property explicitly to "hidden" would.
FinishTemplate	Filename for the template. If set FinishFormat determines various settings. If only the filename is given it is looked up relative to the request path. If

	only a path is given a file is looked up in that where the filename is based in the name of the request path/page - extension + type extension
FinishFormat	If FinishTemplate is set this provides further settings for that. Value is a list with JSON format or for the simple case: key:value. At least the type must be given, eg. "type:ftl" or "type:xsl", other settings depends on type. If the FinishTemplate is not set, this format it treated like simple template using ${...}, where \n and \t are treated as special newline and tab characters. The values that can be referenced are the form values, i.e. if the form has an “email” field, then the value can be used as in the following example of an property value: “Thank you, a confirmation has been sent to ${email}.”
FormValues	Option for setting default values in fields, form: <field-name>:<field-value>[,<field-value>]*; Files properties can be accessed as ${...}
Entity	If given a new record of the entity is created and saved with the values from the submitted form. As a convenience, if the entity defines fields like "creator", "creationDate" and "lastIp" these will be populated
Persistence	Provides a more flexible approach than using the Entity-variation. Spec is on JSON format. It must define either the "class" or "jsp" entries, e.g. {"jsp":"/foo/bar/save.jsp"}. If "class" is given then the class is instantiated (must implement I_PersistFormToRecord interface). In this case the instantiated class is responsible for interpreting the rest of the config info. If "jsp" is given (but no "class") then the com.codedroids.oc.forms.PersistViaJsp is instantiated. This class imports the JSP file into the current request context (after setting two pageContext attributes: the com.codedroids.oc.forms.PersistViaJsp.form and the com.codedroids.oc.forms.PersistWithJsp.config.

At the very least the Form property must be set, but you probably also need to set either Entity or Persistence, otherwise nothing will be saved from the form submission.

Assuming that you have a form definition file and and record definition file placed somewhere in the VFS you can set the following properties on the register.jsp file to have a working example:

```
Form          =/system/modules/com.codedroids.oc.forms/docs/example.form.xml
FinishFormat  =Thank you, we have registered the following email address: ${email}.
Entity        =/system/modules/com.codedroids.oc.forms/docs/example.rec.xml
```

See 3.1 Forms examples p. 8 and 4.1 Records example p.20 for examples on the form and entity files.

Since the above setup is all about saving values to the database an appropriate table should be created in the database before we are ready to test the form submission.

Once that is in place, then the values submitted by the user will be stored in the table in the database and the user sees the messages stored in the FinishFormat property (assuming that the data is validated otherwise the user will be returned to the form with indication of which fields could not be validated)

2.2 MailForm.jsp

This JSP shares a lot of functionality with the previous one as it too can save the submitted data to the database. But it also enables email the submitted data to one or more designated recipients, as well as generating an acknowledge email for the submitter.

The acknowledge email can be given a specific format different from the format used for the email send to the designated recipients.

Note that in order to send emails you need to set up the SMTP settings in the opencms-system.xml configuration file (and restart the application instance after the changes)

It reads the same properties as the Form.jsp plus a few more:

Property	**Description**
MailSubject	Subject line for the mails send to the designated recipients
MailFrom	Mandatory, the sender of the emails. The value must be a valid RFC822 format.
MailReplyTo	If set the reply-to address is set specifically. It must specify a field name where an email address is stored (typically the submitters email so that the designated recipient can use the reply button in the mail program as a convenience instead of having to copy the field value manually in case a reply should be send to the submitter). If field is empty or contains an empty string then reply-to is not set. The value of the field pointed by this value must be a valid RFC822 format.
MailTo	Mail address of the designated recipient(s), possibly a comma-separated list of more than one - if a “MailTo” request attribute is given this takes precedence over the property. The value should only contain the email address (the full RFC822 is not supported).
MailTemplate	Filename for the template. If set mail format determines various settings. If only the filename is given it is looked up relative to the request path. If only a path is given a file is looked up in that where the filename is based in the name of the request path/page - extension + type extension
MailFormat	If MailTemplate is set this provides further settings for that. Value is a list with JSON format or for the simple case: key:value. At least the type must be given, eg. "type:ftl" or "type:xsl", other settings depends on type. If the MailTemplate is not set, this format it treated like simple template using ${...}, where \n and \t are treated as special newline and tab characters. The values that can be referenced are the form values, i.e. if the form has an “email” field, then the value can be used. If neither MailTemplate nor MailFormat is set, then the contents of the email is on the format: Field label : field value with one line per field in the form. If a field does not have a label the field name is used instead. Note that field with field names starting with “_” (underscore) will not be included in the email.
MailValues	Option for setting default values in fields, format: <field-name>:<field-value>[,<field-value>]*; Files properties can be accessed as ${...}

MailAckSubject	Defaults to the MailSubject if not given.
MailAck	Send an acknowledge mail to the user that entered data. It must specify a field name where the users mail address is stored.
MailAckTemplate	Same as for MailTemplate, though this is specific for the acknowledge mail.
MailAckFormat	Same as for MailFormat though this is specific for the acknowledge mail.

```
<%@ page session="false" %>
<%@ taglib prefix="cms" uri="http://www.opencms.org/taglib/cms" %>
<html>
  <head>
    <title>MailForm.jsp test</title>
  </head>
  <body>
    <cms:include file=”/system/modules/com.codedroids.oc.forms/elements/
MailForm.jsp” />
  </body>
</html>
```

If used from the simple JSP shown above the following properties on the page will make a working setup:

```
Form                =/system/modules/com.codedroids.oc.forms/docs/example.form.xml
FinishFormat        =Thank you, we have registered the following email address: ${email}.
MailSubject         =Registration from homepage
MailFrom            =no-reply@codedroids.com
MailTo              =foo@codedroids,com,bar@codedroids.com
MailReplyTo         =email
MailAck             =email
MailAckFormat  =Thank you, this a confirmation of your registration for our newsletter.
```

When the user submits the form an email will be sent to foo@codedroids.com and bar@codedroids.com. This email will look like:

```
active.............: true
newsletterType.....: cd-developer
Your email address*: foobar@somemail.com
action.............: Subscribe
```

An acknowledge email is send to the foobar@somemail.com, in the example it looks like:

```
Thank you, this a confirmation of your registration for our newsletter.
```

In this example nothing is saved in the database as neither the Persistence nor the Entity property are set.

2.3 Using the Forms API

The Form.jsp and MailForm.jsp can be used as examples on how to use the API.

Below is a simpler example:

```
    CmsJspActionElement cms = new CmsJspActionElement(...);
    String uri = cms.getRequestContext().getUri();
    String formFile = “/some/where/example.form.xml”;
    StringBuffer errors = new StringBuffer();
1)  OCForm form = OCForm.getInstance(cms, formFile);
2)  form.getLayout().setFormAction(cms.link(uri));
3)  form.setLocale(cms.getRequestContext().getLocale().toString());
    boolean done = false;
4)  if(form.isSubmitted(request)) {
5)    form.collectValues(request);
6)    int errorCnt = form.validateValues();
      if(errorCnt > 0) {
7)      Iterator i = form.collectErrorMessages().iterator();
        while(i.hasNext())
          errors.append("<li>"+i.next());
      } else {
8)      // do something to the collected data...
        done=true;
      }
    }
    if(!done) {
      if(errors.length() > 0)
        out.println(errors);
9)    out.println(form.toHTML());
    } else {
      ...
    }
```

The first step is to create a Form-instance from a form.xml-file by means of factory-method getInstance() (1). The format of that file is described in the next chapter.

Usually you want to set the action URI dynamically which happens in (2). And if you have a multi-language setup and have defined labels for multiple languages in the form.xml file then you should set the locale as well (3).

For simple cases as mail forms and the like you typically just have one JSP does both request-handling as well as display-handling. To support that the form keeps track on whether or not it has been submitted. You only want to handled the form input if it has been submitted – this is checked in (4).

The first step is to have the form collect the values it need from the request (5) and then you most likely want to validate the input (6). This will return 0 in case no problems were encountered otherwise a number indicating how many of the validations that failed. If there are any errors the error messages can be retrieved as in (7).

If there are no errors go ahead and do your thing with the data (8) – see the next section for an example.

Finally you call the toHTML() method as in (9) to render the form.

2.4 Using the Records API

The Form.jsp and MailForm.jsp are also examples on using the Records API. In the modules “doc” folder there is a further example on how to use the API – the persist-example is intended to be used with the Persistence-property as used by the Form.jsp and MailForm.jsp.

The example below assumes that you some data as hand that need to be persisted, in this case that you have a validated form (as if the code appeared at (8) in the previous code example).

Also, you have to create the relevant tables yourself before it will work. The table and column names must match the ones defined in the rec.xml file.

```
    CmsJspActionElement cms = new CmsJspActionElement(...);
    Form form = ... // submitted and validated form.
    String entityFile = “/some/where/example.rec.xml”
1)  RecordFactory factory = →
        OCRecordFactory.registerFactory(cms.getCmsObject(), entityFile);
2)  I_Record cd = factory.getNewInstance();
3)  form.pushToRecord(cd);
4)  if(cd.hasField("creationDate")) {
      Timestamp now = new Timestamp(System.currentTimeMillis());
5)    cd.takeValueForKey("creationDate", now);
    }
6)  factory.saveRecord(cd);
```

In (1) the factory is created for the entity defined in the rec.xml file. In (2) a record is created for that entity definition. The getNewInstance() method returns a class as defined in the rec.xml file, but they all must implement the I_Record interface.

Next you must set the data on the record. In this example the data is taken from a form by using the pushToRecord() method on the form (3). This works by setting the values in the record that has the same field names as the ones in the form, the string values from the form are converted into the appropriate values as defined by the entity definition (in the rec.xml file). Note, that this step may throw conversion exceptions.

You can manually set the values on the record as shown in (5). In the example it is tested to see if the record defines the field before it is attempted to set it.

Finally the data is persisted in (6).

There is a query API as well. The example below assumes that there is an entity that defines the following fields: newsletterType as a string and active as a boolean type.

```
    CmsJspActionElement cms = new CmsJspActionElement(...);
    String entityFile = “/some/where/example.rec.xml”
    RecordFactory factory = →
        OCRecordFactory.registerFactory(cms.getCmsObject(), entityFile);
1)  AndClause where = new AndClause();
2)  where.addClause( →
            new GenericClause(factory, "newsletterType", "=", “news”) );
3)  where.addClause( →
            new GenericClause(factory, "active", "=", Boolean.TRUE) );
4)  A_OrderBy orderBy = new DescendingOrderBy(factory, "creationDate");
5)  List result = factory.fetchRecords(where, orderBy);
```

In (1) an AND clause is created, to which two clauses are added in (2) and (3). The former requires the newsletter type to be “news” and the latter requires that active is true.

In (4) the sorting of the result is set up such that it is sorted by creationDate (assuming that such a column exists).

Finally the result is fetched in (5). The entries are record instances (I.e. that implements I_Record).

3 Forms file format

The format files are files stored in OpenCms' virtual filesystem. The files are dynamically read so it is not necessary to restart the application instance in order for changes to take effect.

3.1 Forms examples

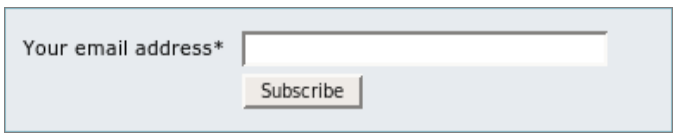
A couple of examples:

```
<?xml version="1.0" encoding="UTF-8"?>
<form>
  <ident>DevNews</ident>
  <fields>
    <private>
      <name>active</name>
      <value>true</value>
    </private>
    <private>
      <name>newsletterType</name>
      <value>cd-developer</value>
    </private>
    <text>
      <name>email</name>
      <label>Your email address*</label>
      <validators>
        <required>
          <error>Please fill in the field.</error>
        </required>
        <email>
          <error>The email address is not valid.</error>
        </email>
      </validators>
    </text>
    <submit>
      <name>action</name>
      <label>Subscribe</label>
    </submit>
  </fields>
</form>
```

This example shows a form with one text field and a submit button. It also includes two private fields. The private fields are server-side fields in the sense that they are not pushed to the client as hidden fields. Typically used for setting values that is needed but should never be changed. Often the same thing is done with hidden fields, but hidden fields can be altered by a crafty user.

The example also shows how to setup up validation. The “email” text fields is marked as required and that it most be on a valid email format. The sequence of the validators define which one is checked first. In the example it is first checked that the fields is not empty, then it is checked that the email address is on the proper format.

The form render like this:




```
<?xml version="1.0" encoding="UTF-8"?>
<form>
  <ident>Newsletter</ident>
  <fields>
    <text>
      <name>releaseDate</name>
      <label>Send date*</label>
      <label locale="da">Afsend. dato*</label>
      <render>date-selector</render>
      <layout>
        <cols>30</cols>
      </layout>
      <validators>
        <required>
          <error>Please fill in the field</error>
          <error locale="da">Udfyld venligst feltet</error>
        </required>
      </validators>
    </text>
    <popuplist>
      <name>newsletterType</name>
      <label>Type*</label>
      <value>-none-</value>
      <ignore>-none-</ignore>
      <option>
        <label><![CDATA[ ]]></label>
        <value>-none-</value>
      </option>
      <option>
        <label>Newsletters</label>
        <value>cd-newsletter</value>
      </option>
      <option>
        <label>Press releases</label>
        <value>cd-pressrelease</value>
      </option>
      <validators>
        <options>
          <min>1</min>
          <max>1</max>
          <error>Please select one value from the list</error>
          <error locale="da">Vælg en værdi fra listen</error>
        </options>
      </validators>
    </popuplist>
    <text>
      <name>subject</name>
      <label>Subject</label>
      <label locale="da">Emne</label>
    </text>
    <text>
      <name>document</name>
      <label>Document*</label>
      <label locale="da">Dokument*</label>
      <render>file-selector</render>
      <validators>
        <required>
          <error>Please fill in the field</error>
          <error locale="da">Udfyld venligst feltet</error>
        </required>
      </validators>
    </text>
    <submitlist>
      <name>action</name>
      <label></label>
      <option>
        <label>Save</label>
```



```
        <label locale="da">Gem</label>
        <value>save</value>
      </option>
      <option>
        <label>Cancel</label>
        <label locale="da">Afbryd</label>
        <value>cancel</value>
      </option>
    </submitlist>
  </fields>
  <layout>
    <formName>Newsletter</formName>
    <styleName>ToolForm</styleName>
  </layout>
  <paramEnc>ISO-8859-1</paramEnc>
</form>
```

This example demonstrates a couple of things. First of all labels and error messages are localized in English and Danish (assuming that the default is English). By setting the locale on the form instance before rendering, it will render itself according to that locale.

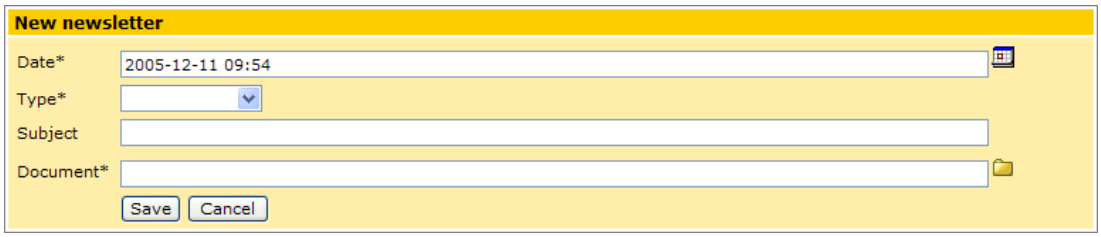
A couple of the text fields has a <render> setting defined. This will invoke a different method when the field is rendered to HTML. The render-classes must be registered before the form is rendered, but a couple special renders are available per default. Date-selector and file-selector renders the text fields with an extra button for selecting a date and a file selector respectively.

The <popuplist> renders as a popup field in the form. In the example three options are given, by setting the <value> this is used as the default, and by setting <ignore> the form will ignore the selection when validating. So if the user chooses the “-none-” selection, then it is not counted during the validation and the hence the <option>-validator will fail since none were selected.

The <submitlist> is recommended in case more than one submit button is required since it is easier to check for in the code afterwards.

The final thing to mention about the example is the <paramEnc> tag. This is useful in setups where the environment (in particular the servlet container setup) is not able to guarantee that submitted values are handled as the correct encoding. There can be many reasons for this, but by using the <paramEnc> option, the form processing will test the value of a hidden field to see if the encoding has been mangled. If it detects that the encoding is not correct it will assume that it is of the encoding given in the <paramEnc>-tag and automatically convert it to the requested encoding. In the example that would mean that garbled data would be assumed to be in ISO-8859-1 encoding, and it would then be converted to UTF-8.

The form is rendered like this (the title is provided by surrounding code):



The default date and the locale are set by the program before the form is displayed.

3.2 Reference

The valid tags are listed alphabetically below:

<banner [locale=]>

Child of: `<message>`.

Using this instead of <label>/<ps> in a <message>-field causes the contents to be output across label, spacing and postscript parts of a row (acting as a banner). Can be localized.

<button>

Child of: `<fields>`.

Children: `<ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>`.

Renders a button element of type “button”.

<checkbox>

Child of: `<fields>`.

Children: `<ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>, <labelPos>, <onValue>, <offValue>`.

Holds a single value, renders a checkbox-type input-field. If using onValue/offValue make sure they are set before the value tag. LabelPos determines wether the label appears to the left or right of checkbox.

<checklist>

Child of: `<fields>`.

Children: `<ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>*, <option>*`.

Multiple checkboxes, zero, one or more values can be selected from multiple options. The default render places option label to the right of the checkbox. Options are listed on one line, either use the stylesheet or
`<layout><optionEnd><![CDATA[<br/>]]></optionEnd></layout>`
to make each option appear on a separate line.

<cols>

Child of: `<layout>`.

Sets the size or number of columns on the fields that supports it.

<email>

Child of: `<validators>`.

Children: `<error>`.

Makes a simple syntax check to see if the value looks like an email – no check of domain or the like.

<error [locale=]>

Child of: `<required>, <email>, <regex>, <options>`.

Error text (possible localized) for the various validators.

<fieldBegin>

Child of: `<layout>`.

Default: ”`\n<td class=\"Field\">`”

Markup output before the field is rendered.

<fieldEnd>

Child of: `<layout>`.

Default: ”`</td></tr>`”

Markup output after the field is rendered.

<fields>

Child of: `<form>`.

Children: `<raw>, <message>, <text>, <hidden>, <password>, <textarea>, <checkbox>, <radiolist>, <popuplist>, <private>, <selectlist>, <checklist>, <submitlist>, <submit>, <reset>, <button>`.

Contains one or more fields which is inlcuded in the form.

<form>

Children: `<ident>, <readonly>, <fields>, <layout>, <paramEnc>`.

Top level tag in forms file.

<formAction>

Child of: `<layout>`.

Submit action for the form. Only relevant for layout-section of form, not relevant for the individual fields.

<formBegin>

Child of: `<layout>`.

Default: ”`\n<table border=\"0\" cellspacing=\"0\" cellpadding=\"2\">`”

Markup inserted after the form definition is rendered, but before any fields are rendered. Only relevant for layout-section of form, not relevant for the individual fields.

<formEnd>

Child of: `<layout>`.

Default: ”`\n</table>`”

Markup inserted after the last field is rendered but before the enclosing form-tag is rendered. Only relevant for layout-section of form, not relevant for the individual fields.

<formMethod>

Child of: `<layout>`.

Default: ”`post`”

Method for the form, POST or GET. Only relevant for layout-section of form, not relevant for the individual fields.

<formName>

Child of: `<layout>`.

Name-attribute on form, if missing the <ident> on the form is used instead. Only relevant for layout-section of form, not relevant for the individual fields.

<format>

Child of: all fields except `<raw>`.

Children: `<type>, <pattern>`.

Currently not in use.

<group>

Child of: `<option>`.

Currently not in use.

<hidden>

Child of: `<fields>`.

Children: `<ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>`.

Holds a single value, renders as hidden input-field. Hidden fields are rendered as the first in the form, without any additional layout markup. Layout- and label-tag has no effect here.

<ident>

Child of: `<form>,` and all fields except `<raw>`.

In HTML output this value is set on the id-attribute. In case of the form itself this is used for the name-attribute as well, if there is no form name defined in the form layout. For fields the name is used as ident in case that is set and the ident is not.

<ignore>

Child of: `<radiolist>, <popuplist>`.

If the input value is equals the value given as ignore, then it is considered a no-selection choice. The value is then null.

<label [locale=]>

Child of: all fields except `<raw>`.

Text used as label for the field. Default render output a label-element

<labelBegin>

Child of: `<layout>`.

Default: ”`\n<tr><td class=\"Label\">`”

Markup output before rendering of a given field starts.

<labelEnd>

Child of: `<layout>`.

Default: ”`</td><td width=\"10\"></td>`”

Markup output after the label has been rendered, but before the field itself is rendered.

<labelPos>

Child of: `<checkbox>`.

Value: `before` (default) or `after`.

Place the label either before (normal procedure) or after the checkbox.

<layout>

Child of: `<form>`, and all fields except `<raw>`.

Children: `<styleName>`, `<style>`, `<formAction>`, `<formMethod>`, `<formName>`, `<formBegin>`, `<formEnd>`, `<labelBegin>`, `<labelEnd>`, `<fieldBegin>`, `<fieldEnd>`, `<optionBegin>`, `<optionEnd>`, `<stylePrefix>`, `<rows>`, `<cols>`.

Controls how the form is rendered (typically as HTML, which is what is used for the default values of the various children). Note that the layout-values defined at <form>-level will override the default for the values used at field-level. Fields must define their own layout-values to override inherited or default value.

The tags named form-something are only relevant for the layout of the form itself, not the fields.

<max>

Child of: `<options>`.

Maximum number of options that can be selected from a multi-choice field. Used by the options validator.

<message>

Child of: `<fields>`.

Children: `<ident>`, `<name>`, `<label>`, `<validators>`, `<layout>`, `<readonly>`, `<ps>`, `<format>`, `<render>`, `<value>`, `<banner>`.

Not really a value field, just a convenience for printing banners messages. Default renderer places label as usual and banner as if it where the value. The banner localized which the value cannot. If the banner text contains "`${value}`" then this will be replaced by the value of `<value>`.

If `<label>` and `<ps>` are not given, then the banner will strech all the way across the entire row.

<min>

Child of: `<options>`.

Minimum number of options that must be selected from a multi-choice field. Used by the options validator.

<name>

Child of: all fields except `<raw>`.

Name of the field. If no name is given the ident is used as name.

<offValue>

Child of: `<checkbox>`.

Provide a specific value indicating that the checkbox is unchecked/off..

<onValue>

Child of: `<checkbox>`.

Provide a specific value indicating that the checkbox is checked/on. Defaults to `true`.

<option>

Child of: `<radiolist>, <checklist>, <popuplist>, <selectlist>`.

Children: `<label>, <value>`.

Used together with fields where the user can select from more than one value. The <label> can be localized. If the `<label>` is missing, the value is used as label. If the `<value>` is missing the label is used as value. If both are missing, the contents of `<option>`-tag is read as text and used both as label and value.

<optionBegin>

Child of: `<layout>`.

Default: ””

For elements that uses options, this markup is output before the option is rendered.

<optionEnd>

Child of: `<layout>`.

Default: ”` `”

For elements that uses options, this markup is output after the option is rendered. If options should appear on seperate lines one could insert a `<br>` here (or better yet, control it via stylesheet).

<options>

Child of: `<validators>`.

Children: `<error>, <min>, <max>`.

Validates the users selection from fields where more than one item can be selected. Minimum and maximum specifies the valid range. If `<min>` is not given minimum default to 1. If `<max>` is not

given, then the selection may hold any number.

This validator also ensures that the selected values actually are values given by the <option>s for the field.

<paramEnc>

Child of: <form>.

This is useful in setups where the environment (in particular the servlet container setup) is not able to guarantee that submitted values are handled with the correct encoding. There can be many reasons for this, but by using the <paramEnc> option, the form processing will test the value of a hidden field to see if the encoding has been mangled somewhere along the line. If it detects that the encoding is not correct it will assume that it is of the encoding given in the <paramEnc>-tag and automatically convert it before storing the value.

<password>

Child of: <fields>.

Children: <ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>.

Holds a single value, renders as password input-field.

<pattern>

Child of: <regex>.

Regular expression used by the <regex> validator.

<popuplist>

Child of: <fields>.

Children: <ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>, <option>*.

Renders a multi-choice popup-list, with zero or one selected (based on the given <value>s). If value matches the <ignore>-value this is considered as if nothing was selected. Multiple <option>s can be given.

<private>

Child of: <fields>.

Children: <ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>.

This value is only set internally in the form instance, it will not be send to the client and the client cannot set it. This is safer than using <hidden> for fields that should stay fixed.

<ps [locale=]>

Child of: all fields except <raw>.

Text to be rendered after the field element (postscript). Typically a comment , explanation, unit of the value in the field or what ever. Can be localized.

<psBegin>

Child of: `<layout>`.

Default: ””

Markup to mark the start of field postscript

<psEnd>

Child of: `<layout>`.

Default: ””

Markup to mark the end of field postscript

<radiolist>

Child of: `<fields>`.

Children: `<ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>, <option>*`.

Renders a multichoice list, with zero or one selected (based on value). If value matches the ignore-value this is considered as if nothing was selected. Multiple options can be given.

<readonly>

Child of: `<form>`, and all fields except `<raw>`.

Values: `true` or `false` (default).

Marks a field read only. If set on the form all fields inherit the setting unless they have a value set explicitly.

<regex>

Child of: `<validators>`.

Children: `<error>, <pattern>`.

Validates the value against a regilar expression given by `<pattern>`.

<render>

Child of: and all fields except `<raw>`.

Use the given render to render the field instead of the default render. Renders must be registered with the form instance before parsing a form that request special renders.

The following field renders are available:

default (Form) – works with all fields.
none (Form) – works with all fields (generates no output).
file-selector (OCForm) – works with text-fields (inserts the VFS fileselector).
date-selector (OCForm) – works with text-fields (inserts a Javascript date-selector).

<required>

Child of: `<validators>`.

Children: `<error>`.

Report error if the value is null, or the selection is empty (the selection is considered empty if a `<ignore>` is set for the field and the selection matches it).

<raw [locale=]>

Child of: `<fields>`.

Raw text, concatenates text/nodes. Output only – carries no value.

<reset>

Child of: `<fields>`.

Children: `<ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>`.

Renders a button element of type “reset”.

<rows>

Child of: `<layout>`.

Sets the number of rows for those fields that supports it.

<selectlist>

Child of: `<fields>`.

Children: `<ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>*, <option>*`.

Multiple lines, zero, one or more values can be selected from multiple options.

<style>

Child of: `<layout>`.

Sets the given CSS directly on element via the style-attribute.

<styleName>

Child of: `<layout>`.

Sets the value as class-attribute of the element (besides the default style classes set by the render).

<stylePrefix>

Child of: `<layout>`.

If defined the default style-classes (set on the class-attributes by render) gets prefixed with this value. Useful for pages with more than one form and where the forms need to have different look

<submit>

Child of: `<fields>`.

Children: `<ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>`.

Renders a button element of type “submit”.

<submitlist>

Child of: <fields>.

Children: <ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>.

Renders a group of submit-buttons (one per option) in such a manner that the value of the field is set to that of the option that was clicked. Default render uses <input type="submit"> rather than <button type="submit"> (as opposed to the submit-field above)

<text>

Child of: <fields>.

Children: <ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>.

Holds a single value, renders as text-type input-field.

<textarea>

Child of: <fields>.

Children: <ident>, <name>, <label>, <validators>, <layout>, <readonly>, <ps>, <format>, <render>, <value>.

Holds a single value, renders as multiline input-field.

<type>

Child of: <format>.

Currently not in use.

<validators>

Child of: all fields except <raw>.

Children: <required>, <email>, <regex>, <options>.

One or more validators. The sequence of the validators inside this tag defines the sequence in which they are applied.

<value>

Child of: <text>, <hidden>, <textarea>, <checkbox>, <radiolist>, <checklist>, <popuplist>, <selectlist>, <submit>, <reset>, <option>.

The value of the field (or in case it is used inside <option> the value of the option).

Some fields such as <checklist> and <selectlist> can have multiple values.

4 Records file format

The format files are files stored in OpenCms' virtual filesystem. The files are dynamically read so it is not necessary to restart the application instance in order for changes to take effect, but you will have to update the database table to reflect changes to the fields. If your entity class is not dynamic, you will have to reload the instance in order for the newly compiled class to be reloaded.

4.1 Records example

The example uses the Map wrapper record class which allows any fields, I.e. it is fully dynamic. All you have to do is to add the fields you want to the rec.xml file and to the appropriate table in the database and you are ready to use it.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<entity>
  <name>Subscription</name>
  <class>com.codedroids.records.MapRecord</class>
  <table>cd_subscriptions</table>
  <ident scheme="sequence">id</ident>
  <fields>
    <field>
      <name>id</name>
      <type>Integer</type>
      <column>id</column>
    </field>
    <field>
      <name>email</name>
      <type>String</type>
      <column>email</column>
    </field>
    <field>
      <name>newsletterType</name>
      <type>String</type>
      <column>newsletter_type</column>
    </field>
    <field>
      <name>active</name>
      <type>Boolean</type>
      <column>active</column>
    </field>
    <field>
      <name>creationDate</name>
      <type>Timestamp</type>
      <column>creation_date</column>
    </field>
  </fields>
</entity>
```

Other than defining the data class, the definition defines which table to use and which field is used as ident. The ident also defines the scheme which defines how the idents are generated. The example uses sequenc which basically uses OpenCms' “old” numeric ident generator.

Next a number of fields are defined. Each one defines the name (field name), the Java data type and the column name.

4.2 Reference

<bag>

Child of: `<entity>`.

Currently not in use.

<class>

Child of: `<entity`.

Name of class to be instantiated, one instance per record.

<column>

Child of: `<field>`.

Name of column in database.

<entity [version=]>

Children: `<bag>, <class>, <ident>, <name>, <table>, <fields>`.

Top-level tag.

<field>

Child of: `<fields>`.

Children: `<column>, <name>, <type>`.

Defines a field.

<fields>

Child of: `<entity>`.

Children: `<field>*`.

The list of fields that are mapped to columns in the table.

<ident [scheme=]>

Child of: `<entity>`.

Defines the ident (primary key) field, must hold the name of a field.

Scheme must be one of the following: `sequence` (default), `uuid`, `random`, `none`.

<name>

Child of: `<entity>` or `<field>`.

Name of the entity or the field.

<table>

Child of: `<entity>`.

Name of the table in the database,

<type>

Child of: `<field>`.

The Java type name corresponding to the column value, such as `Integer`, `Boolean` and so forth.

