Formbar Documentation

Release 1.1.0+bewec1+bewec1

Torsten Irländer

Sep 27, 2017

Contents

CHAPTER 1

Getting Started

About

Formbar is a python library to layout, render and validate HTML forms in web applications. Formbar renders forms which are compatible with Twitter Bootstrap styles.

In contrast to many other form libraries forms with formbar are configured in XML files to separate the form definition form the implementation and handle it as configuration.

Formbar is the German word for "shapeable" and should emphasise the character of formbar which hopefully makes shaping your forms more easy.

Installation

Formbar is available as Pypi package. To install it use the following command:

```
<venv> pip install formbar
```

The source is availble on Bitbucket. You can check of the source and install the library with the following command:

```
(venv)> hg clone https://bitbucket.org/ti/formbar
(venv)> cd formbar
(venv)> python setup.py develop # use develop for development install
```

Tip: I recommend to install the library for testing issue in the virtual python environment. See Virtualenv documentation for more details.

Features

- Support for SQLAlchemy mapped items and plain forms.
- Expression bases rules
- Conditionals in forms
- Type conversation and validation
- XML based form definition
- i18n Support
- Row and column based layouts
- Different form layouts for the same model (Create, Edit, Read...)
- Twitter bootstrap support
- Custom CSS styling
- Error and warning messages
- Help texts
- Numbering of fields
- Extern defined renderers
- ...

Quickstart

See *Usage*

Demo-Server

Formbar comes with a very simple demo server to give you a impress on some features of formbar.

To run the server do the following:

```
cd examples
python serve.py
```

Licence

Authors

Torsten Irländer <torsten at irlaender dot de>

CHAPTER 2

Form configuration

The form will be configured using a XML definition. The configuration is basically splitted into two parts:

1. The definition of the datamodel in the *source* directive.
2. Definition and Layout of forms in *forms*.

The basic form configuration looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration>
   <source>
   <!-- Define different entity types -->
   </source>
   <form>
   <!-- Define and layout a form -->
   </form>
   <snippet>
   <!-- Container holdig parts of a form definition -->
   </snippet>
</configuration>
```

The *Snippet* element is optional and just a helper.

Datamodel

The `source` directive defines the *Entity* are available in your forms. An entity is defined only once in the source section. It will get referenced in the *FormbarFormEditor* directive later to build the forms.

Entity

A *Entity* is a field definition. The entity is used to configure aspects of the *datamodel* the *layout* and *behaviour* of the field in the form.

Here is an example of an entity definition:

```
<entity id="f1" name="age" label="Age" type="integer" css="field" required="true">
    <renderer type="text"/>
    <help>This is a help text</help>
    <rule expr="$age ge 21" msg="Age must be greater than 21"/>
    <validator src="a.b.external_validator" msg="Error message"/>
</entity>
```

Entities can be marked as *required* or *desired*. Formed will generate automatically a *Rule* for this field. Missing required fields will trigger an error on form validation. Desired fields will trigger a warning.

Entities can be marked as *readonly*. Readonly fields are renderer as simple text in the form displaying the current value of the field. Note, that readonly fields are not sent on submission! If you need the value if the form you will need to add an additional entity and render is with the hidden field renderer.

Each entity can optional have a *Renderer*, *Rule* or *Help* element.

Attribute	Description
id	Used to refer to this entity in the form. Requiered. Must be unique.
name	Used as name attribute in the rendered field. Defines the name of this attribute in the model. Name of the field must only contain characters which are valid in context of your database. So better stay with [**a-zA-Z0-9_**]
label	The field will be rendered with this label.
number	A small number which is rendered in front of the label.
type	Defines the python datatype which will be used on deserialisation of the submitted value. Defines the datatype of the model. Possible values are `string` (default), `text`, `integer`, `float`, `date`, `datetime`, `email`, `boolean`, `time`, `interval`. css Value will be rendered as class attribute in the rendered field.
expr	Expression which is used to calculate the value of the field.
value	Default value of the field. Supports expressions. The default value might get overwritten on rendering.
placeholder	Custom placeholder that overrides the default of a field. For now only usable for `interval`.
readonly	Flag to indicate that the field should be rendered as readonly field. Default is `false`.
required	Flag to indicate that the is a required field. Default is `false`.
autofocus	Flag to mark the field to be focused on pageload. Only one field per form can be focused. Default is `false`.
desired	Flag to indicate that the is a desired field. Default is `false`.
tags	Comma separated list of tags for this field.

Defaults

You can set a default for the field in case there is no value for the field. The default value can be set by using the `value` attribute of the entity.

You can provide a default value by

1. Given in plain string value
2. Accessing an attribute of the SA mapped item. This supports dot separated attribute names of the item to access related items:

```
... value="$foo.bar.baz"
```

"$" represents the current form item. So foo is an attribute of it and bar is an attribute of foo.

3. Using expressions. The default value can be calculated by using a expression:

```
... value="% date('today')"
```

The expample will set the value to the current date. "%" is used to say formbar that the following string must be considered as an expression. The Expression will evaluated with the values of the current form item.

Options

Options are used to define available options for a entity in case it is an selection. The options my be defined in different ways.

By defining every option per hand:

```
<options>
    <option value="1">Foo</option>
    <option value="2">Bar</option>
    ...
    <option value="99">Baz</option>
</options>
```

By setting the value attribute of the options. This should be the name of an attribute of the item which is used to get the available options:

```
<options value=""/>
```

By not defining options at all and letting the library load the options for you based on the entity name.

Attribute	Description
value	Optional. Name of an attribute of the item which will provide a list of items used for the options.

Rule

Rules are used to validate data in the form. Formed does already some basic validated on the submitted data depending on the configured data type in the *Entity*. These checks are often already sufficient for most basic forms.

If you need more validation rules can be used to define additional checks. There are two types of rules. Rules which triggers errors, and rules which trigger a warning if the evaluation of the rule fails.

Rules are evaluated in the process of validation the submitted data. On validation formed will collect warning and errors and will rerender the form displaying them. If the form has errors the validation fails. Warnings are ok for validation.

Validation of rules can be done in differen modes. Rules with the mode `pre` are evaluation before the deserialisation of the submitted value occurs into the python data type of the field. In contrast rules with mode `post` are evaluation after the deserialisation happened.

Here is a example rule:

```
<rule expr="$age ge 21" msg="Age must be greater than 21" mode="post" triggers=
↪"warning"/>
```

Here you can see a example rule. The rule will check the value of field "age" ($age) is greater or equal that the value 21. The rule is evaluated in post mode. And will trigger a warning if the evaluation fails.

Attribute	Description
expr	Expression which is used to validate the value if the field.
msg	The message which is displayed if the evaluation of the rule fails.
mode	Point in validation when this rules gets evaluations. `post` (default) means after the deserialisation of the value and `pre` is before deserialisation.
triggers	Flag which defines which type of message a the rule will trigger if the evaluation fails. Be be `error` (default) or `warning`.

Validator

A validator defines an external validator. See *Write external validators* for more details. Those validators are usally used if the validation become more complex or it is just not possible to express the rule with a *Rule* You can define a validator in the form configuration in a similar way like defining rules for an entity:

```
<validator src="a.b.external_validator" msg="Error message"/>
```

Attribute	Description
src	The *src* attribute is the modul path to the callable. The path is used to import the validator dynamically at runtime.
msg	The message which is displayed if the evaluation of the validation fails.

Help

The help block can be used to add some information to the field for the user. You can also define some HTML content for the help block to add links to external ressources for example:

```
<help display="text"><html>HTML content must be wrapped in <i>html</i>
tags></html></help>
```

To be able to use the HTML content the content of the help element must be wrapped in a html tag. But you can leave this out in case you just have ordinary text content.

Attribute	Description
display	Defines how and where to display the information on the field. Can be `tooltip` (default) or `text`.

Depending on the display attribute of the help the information is either shown as tooltip next to the label of the field or below the field as normal text.

Renderer

The renderer directive can be used to configure an alternative renderer to be used to render the field.

The default renderer is chosen depending on the datatype of the field and is a textfield for almost all normal datatypes. On relations (in SQLAlchemy mapped items) a selection field is used for the relations

Attribute	Description
type	Type of the renderer. See *Renderers*
indent	Style of indent of input elements. If set the field elements and help texts under the label will get an indent. This only applies if the label position is set to top. Defaults to no indent. Possible values are *empty*, *symbol* and *number*, *bg*. The style can be combined with the further attributes to define additional styling aspects linke border and width of the indent. Use *bordered* to get some additional visual indication of the indent and *sm*, *md*, *lg* to define the size of the indention.

There are different types of *Renderers* available coming with formed. You can define which renderer will be used by setting the *type* attribute:

```
<renderer type="checkbox" indent="number-borderd-lg"/>
```

But it is very easy to write your own custom renderer. See *Write custom renderes* for more details on writing custom renderes and *Use Custom renderers* on how to use them for rendering in your form.

Label

The lable tag can be used to have more options to configure the rendering of the fields label. The label tag can be seen as a configuration option of the renderer:

```
<renderer>
    <label position="left" align="right" width="4"/>
    ...
</renderer>
```

The label tag is only used to configure the position, alignment and the width of the label. The text of the label is still configured in the entitiy.

Attribute	Description
position	The position of the label realtive to the field element. Can be “left”, “top”, “right”. Defaults to “top”.
align	The alignment of the text in the label. This only applies for labels with position set to “left” or “right”. Can be “left” and “right”. Defaults to “left”.
width	The width of the label in cols. The whole field including the label can be deived into 12 cols. If the label has e.g 4 cols the field will automatically take the remaining 8 cols. This only applies for labels with position set to “left” or “right”.
number	The position of the small number (if set) in the label. Can be *left* or *right* Defaults to *left*.
background	Optional if set to true the label will get a light backgroud color.

Layout

The form directive is the place where the form definition and layout happens.

Hint: You can define more than one form in one configuration. This gets very handy if you want to define different forms for differen purposes. Example: You have a form to create a new item with a reduced set of fields. Another form which has all fields included can be used to edit the item.

Forms are built by using references to the defined entities packed in some layout directives:

```
<form id="create" css="fooish" autocomplete="off" method="POST" action="" enctype=
↪"multipart/form-data">
...
</form>
```

Attribute	Description
id	Unique id of the field.
css	The attribute will be added to the *class* attribute of the form.
autocomplete	Flag to indicate if the form should be autocompleted by the browser. Defaults to `on`.
method	HTTP method used to submit the data. Defaults to POST.
action	URL where is submitted data is sent to. Default to the current URL.
enctype	Encrytion used while sending the data. Defaults to `application/x-www-form-urlencoded`. Use `multipart/form-data` if you plan to submit file uploads.

Buttons

Optional directive within the form tag to configure custom buttons for the form. If not defined the default Submit Button is renderered. If the form has pages than an additional "Save and proceed" button is rendered.:

```
<buttons>
  <button type="submit" value="delete" name="_submit" class="warning" icon="glyphicon␣
↪glyphicon-delete">Delete</button>
  ...
</buttons>
```

Buttons are rendererd at the bottom of the form element. The first button in the definition will be the first button on the left side.

Attribute	Description
type	Optional. Type of action the button will trigger on the form (submit, reset). Defaults to `submit`
value	Optional. Value which is submitted in the form. Defaults to the buttons text.
name	Optional. Name under which the value will be available in the submitted data Defaults to `_$type`.
class	Optional. CSS class which will be added to the button.
icon	Optional. Definition of glyphicons which will be displayed before the buttons label.
ignore	Optional. If set the button will be ignored on rendering. This can be used to ignore rendering of buttons at all in a specific form.

Page

Use pages if you want to divide your form into multiple pages. Pages are rendered as a separate outline of the form on the left site to navigate through the form pages.

Row, Col

Used to layout the form:

```
<row>
  <col></col>
  <col></col>
</row>
<row>
  <col width="8"></col>
  <col width="2"></col>
  <col width="2"></col>
</row>
```

The form is divided into 12 virtual cols. The width of each col is calculated automatically. A single in a row will have the full width of 12. For 2 cols in a row each col will have a width of 6 cols. If you define 3 cols each col will have a width of 4 and so on.

You can alternatively define the *width* of the col. If you provide the width of the col you need to take care that the sum of all cols in the row is 12 to not mess up the layout.

Rows and cols can be mixed. So rows can be in cols again.

Attribute	Description
width	Width of the col (1-12).

Sections

Sections can be used to divide a page in logical sections. This is very similar to the fieldsets:

```
<section label="1. Section">
  <subsection label="1.1 Subsection">
    <row>
      <col></col>
      <col></col>
    </row>
    <subsubsection label="1.1.1 Subsubsection">
        ...
    </subsubsection>
  </subsection>
</section>
```

Every section will genereate a HTML header tag. Formbar supports up to three levels of sections.

Attribute	Description
label	Label of the fieldset rendered as header.

Fieldset

A fieldset can be used to group fields into a logical unit a fieldset will have a label which is rendered as a heading above the first field of the fieldset. Fieldsets can be nested to model some kind of hierarchy. Formbar supports up to three levels. The size of the font in the fieldset legend will be reduced a littlebit on every level.:

```
<fieldset label="1. Foo">
...
  <fieldset label="1.1 Bar">
    <row>
      <col></col>
      <col></col>
    </row>
```

```
  <fieldset>
<fieldset>
```

A fieldset can include almost all other directives.

Attribute	Description
label	Label of the fieldset rendered as header.

Text

Text can be used to add some simple text information in the form. It does not support any formatting of the text. If you need more formatting please use the html renderer:

```
<row>
  <col><text>Hello I'm Text</text></col>
  <col><text>Hello I'm a seconds Text</text></col>
</row>
```

Attribute	Description
color	Color of the text. Possible options: “muted”, “warning”, “danger”, “info”, “primary”, “success”. Defaults to no change of the current text color.
bg	Color of the background. Possible options: “warning”, “danger”, “info”, “primary”, “success”. Defaults to render no background.
em	Emphasis of the text. Possible options: “strong”, “small”, “em” (italic). Defaults to no emphasis.

Table

Important: Tables should not be used to layout the form!

Tables can be used to arrange your fields in a tabuluar form. This becomes handy in some situations e.g to build your own widget:

```
<table>
  <tr>
    <th>Criteria</th>
    <th>Male</th>
    <th>Female</th>
  </tr>
  <tr>
    <td width="70%">Number of humans in the world</td>
    <td><field ref="men"/></td>
    <td><field ref="women"/></td>
    <td><field ref="total"/></td>
  </tr>
</table>
```

Tables are usually used in the same way as *Field* is used. Tables will take 100% of the available space. You can set the `width` attribute of the <td> field to configure the width of the columns. The width of the column can be set to % or pixel.

The following attributes are supported for the `td` and `th` tags of the table: `width`, `class` , `rowspan`, `colspan`.

HTML

The html directive is used to insert custom html code. This is usefull if you want to render generic text sections icluding lists or other markup elements linke images. Images will need a external source for the image file.:

```
<html>
 <ul style="padding:15px">
   <li>List item 1</li>
   <li>List item 2</li>
   <li>List item 3</li>
 </ul>
</html>
```

The content of the html directive will be rendererd as defined so you are free to include whatever you want.

Field

A field in the form. The field only references an *Entity*:

```
<field ref="f1"/>
```

Attribute	Description
ref	id if the referenced *Entity*.

Conditional

Conditional can be used to hide, or render form elements like fields, tables, fieldsets and text elements within the conditional as readonly elements.

If the condition must evaluate to true or false. If true, the elements are rendered normal. If the condition is false the effect is determined by the type of the conditional. On default the elements will be hidden completely. As alternative you can set the type of the conditional to "readonly". Currently only the type "readonly" are supported. Expample:

```
<if type="readonly" expr="$fieldname == 4">
    <field ref="r1"/>
</if>
```

In the example above the referenced field will be shown if the field in the form with the name "fieldname" has the value of 4. Else the element will be set to readonly and the element will have a lowered opacity.

Attribute	Description
type	Effect of the conditional if the condition evaluates to false. Defaults to `hidden`.
expr	The expression which will be evaluated.
static	Flag disable dynamic clientsided evaluation of the conditional. Defaults to `false`.
reset-value	If *true* than the value of all fields with in the conditional will be removed . Defaults to `false`.

Conditionals are evaluated using JavaScript on the client side. Formbar also needs to evaluate the conditional internal on validation to determine which values will be taken into account while validating. As result validation rules will not be applied for "hidden" fields.

Snippet

Snippets are reusable parts of your form definiton. Snippets allow you to define parts of the form only once and use them in multiple forms. Example: If you want to use the same form to create and edit than you can define the form in a snippet and use it in the create and edit form:

```
<form id="foo">
  <snippet ref="s1"/>
</form>
<form id="bar">
  <snippet ref="s1"/>
</form>
<snippet id="s1">
  <row>...</row>
</snippet>
```

Snippet needs to be in a form to get rendered. Snippets can reference other snippets using the `ref` attribute. Snippets are of great help if you want to reduced the effort of rearranging groups of elements in the form. But on the other side the can make the form quite complicated if you use them too much. Use them with care.

Attribute	Description
id	Unique id of the snippet
ref	References the snippet with id.

Renderers

Usually the renderer for a field is chosen automatically from formbar based on the datatype. But you can define an alternative renderer. Below you can the the available default renderers in ringo. If you need custom renderers the refer to *Write custom renderes*

Textarea

Use this renderer if you want to render the field as a textfield:

```
<renderer type="textarea" rows="20"/>
```

Attribute	Description
rows	Number of rows of the texteare. Default is 3.
maxlength	Number of chars "allowed". If set a small indicator below the textarea is show indicating how many chars are left. Please note that this does **not** triggers any rules. Rules to enforce this maxlength must be defined too.

Infofield

The info field renderer is used to render the value of the entity as textual information. This renderer is usually used to display calculated values of the entity. See the `expr` attribute of the *Entity*. If you simply want to display a static value comming from on of the items attribute you can also use the `value` attribute. Appearance is same as a readonly field:

```
<renderer type="infofield"/>
```

Attribute	Description
showraw-value	If set to true the info field will return the "raw" value if the field which whithout any exapandation or conversion of the value. This becomes handy for relations if you want to show the related item instead of just its id. Default is false.

Selection

The selection renderer is used to render a selection list fields. Such a field is capable to select multiple options. The renderer defines also the options which should be available in the dropdown menu. For SQLAlchemy mapped items the options are automatically determined from the underlying data model:

```
<entity>
  <renderer type="selection"/>
  <!-- Note, that the options are part of the entity! -->
  <options>
     <option value="1">Option 1</option>
     <option value="2">Option 2</option>
     <option value="3">Option 3</option>
  </options>
</entity>
```

Attribute	Description
filter	Expression which must evaluate to True if the option should be shown in the Dropdown.
remove_filtered	Flag "true/false" to indicate that filtered items should not be rendered at all. On default filtered items will only be hidden and selection is still present.
sort	If set to "true" than the options will be alphabetically sorted. Defaults to no sorting.
sortorder	If set to "desc" the sorting will be descending (reversed) order. Default is ascending sorting.

Filtering can be done by defining a expression in the filter attribute. This expression is later evaluated by the rule system of formbar. The expression must evaluate to true and is evaluated for every option. The expression uses a two special variables begining with

1. `%`. Variables beginning with % marks the options of the selection. `%attr` will access a attribute named 'attr' in the option. A single `%` can be used on userdefined options to access the value of the option. For SQLAlchemy based options comming from the database `%` can be used to access a attribute of the option. E.g '%id' will access the id attribute of the option. The variable will be replaced by the value of the attribute of the current item in the option for every option before evaluating.

2. `@`. Varaible beginning with @ marks the name of an attribute of the parents form item.

3. `$`. Varaible beginning with $ marks the name of field in the form.

All variables support accessing related items through the dot-syntax:

```
<renderer type="selection" filter="%foo eq @bar.baz">
```

Dropdown

The dropdown renderer is used to render dropdown fields. The renderer defines also the options which should be available in the dropdown menu. For SQLAlchemy mapped items the options are automatically determined from the underlying data model:

```
<entity>
   <renderer type="dropdown"/>
   <options>
      <option value="1">Option 1</option>
      <option value="2">Option 2</option>
      <option value="3">Option 3</option>
   </options>
</entity>
```

Attribute	Description
filter	Expression which must evaluate to True if the option should be shown in the Dropdown.
remove_filtered	Flag "true/false" to indicate that filtered items should not be rendered at all. On default filtered items will only be hidden and selection is still present.
sort	If set to "true" than the options will be alphabetically sorted. Defaults to no sorting.
sortorder	If set to "desc" the sorting will be descending (reversed) order. Default is ascending sorting.

Note: Filtering is only possible for SQLAlchemy mapped items.

See filtering section of the *Selection* renderer.

Radio

The radio renderer is used to render radio fields based on the given options. Such a field is capable to select only one option. For SQLAlchemy mapped items the options are automatically determined from the underlying data model. The radionfields will be aligned in a horizontal row:

```
<entity>
  <renderer type="radio"/>
  <options>
     <option value="1">Option 1</option>
     <option value="2">Option 2</option>
     <option value="3">Option 3</option>
  </options>
</entity>
```

Attribute	Description
filter	Expression which must evaluate to True if the option shoul be shown in the Dropdown.
align	Alignment of the checkboxes. Can be "vertical" or "horizontal". Defaults to "horizontal".
sort	If set to "true" than the options will be alphabetically sorted. Defaults to no sorting.
sortorder	If set to "desc" the sorting will be descending (reversed) order. Default is ascending sorting.
selected	If set the renderer will select the n-th entry from the options (0 ist first, -1 is las, etc). The entry will only be seleced if the entity does **not** have a value or a default value. Default is to not select an entry.

See filtering section of the *Dropdown* renderer.

Checkbox

The checkbox renderer is used to render checkbox fields based on the given options. Such a field is capable to multiple options. For SQLAlchemy mapped items the options are automatically determined from the underlying data model. The checkboxes will be aligned in a horizontal row:

```
<entity>
  <renderer type="checkbox"/>
  <options>
     <option value="1">Option 1</option>
     <option value="2">Option 2</option>
     <option value="3">Option 3</option>
  </options>
</entity>
```

Attribute	Description
filter	Expression which must evaluate to True if the option shoul be shown in the Dropdown.
remove_filtered	Flag "true/false" to indicate that filtered items should not be rendered at all. On default filtered items will only be hidden and selection is still present.
align	Alignment of the checkboxes. Can be "vertical" or "horizontal". Defaults to "horizontal".
sort	If set to "true" than the options will be alphabetically sorted. Defaults to no sorting.
sortorder	If set to "desc" the sorting will be descending (reversed) order. Default is ascending sorting.

See filtering section of the *Dropdown* renderer.

Textoption

A textoption field is basically a selection field which can be used to set multible values. This type of renderer is often used for adding *tags*. In a textoption field the values can be entered in a textfield. The textfield has support for autocompletion which offers the available options:

```
<entity>
  <renderer type="textoption"/>
  <options>
     <option value="1">Option 1</option>
     <option value="2">Option 2</option>
     <option value="3">Option 3</option>
  </options>
</entity>
```

In this example the user can enter "Op" in the textfield and the autocompletion will offer all options beginning with "Op". If the users selects on or more options, the will be set in the background and submitted on form submission.

Attribute	Description
filter	Expression which must evaluate to True if the option shoul be shown in the Dropdown.
remove_filtered	Flag "true/false" to indicate that filtered items should not be rendered at all. On default filtered items will only be hidden and selection is still present.
sort	If set to "true" than the options will be alphabetically sorted. Defaults to no sorting.
sortorder	If set to "desc" the sorting will be descending (reversed) order. Default is ascending sorting.

See filtering section of the *Dropdown* renderer.

Datepicker

The datepicker renderer has some Javascript functionality which lets the used pick the date from a calender. It also only allows valid date entries per keyboard:

```
<renderer type="datepicker"/>
```

Password

The password renderer renderes a password field which hides the users input:

```
<renderer type="password"/>
```

Hidden

The hidden field renderer is used to render a hidden field for the entity. No labels, helptexts or error messages will be renderer. The hidden field will also take care on relations for SQLAlchemy mapped items:

```
<renderer type="hidden"/>
```

Html

The html renderer is used to render custom html code. This is usefull if you want to render generic text sections or insert images. Images will need a external source for the image file. The html renderer will render Javascript , Stylesheets and HTML code:

```
<renderer type="html">
 <div>
   <p>You can include all valid html including images, lists etc.</p>
   <p><strong>Warning:</strong>Also JS can be included.</p>
 </div>
</renderer>
```

Your custom code should be wrapped into a empty div node. Otherwise only the first child node of the renderer will be rendererd. The entity only needs the id attribute. If a label is provided, the label will be uses as some kind of header to the html part.

Warning: Use this renderer with caution as it may introduce a large security hole if users inject malicious javascript code into the form using the html renderer.

FormbarFormEditor

Use this renderer if you want to render a editor for formbar forms. The Editor will have a preview window which shows the result of the rendering of the form. If rendering fails, the preview will show the errors which happened while rendering:

```
<renderer type="formbareditor" url="foo/bar" rows="20"/>
```

Attribute	Description
rows	Number of rows of the textarea. Default is 3.
url	URL which is called to renderer the form.

Metadata (Specification)

You can add add metadata information to `configuration`, `entity`, `option`, `renderer`, `rule`, `form`, `snippet` elements of the form.

Metadata can be used to build some kind of specification of the form. This data can be used by the `formspec.py` command to generate a specification of the form.

Every metadata block will look like this:

```
<metadata>
    <meta attrib="example" date="YYYYMMDD"></meta>
</metadata>
```

Attribute	Description
attrib	Classification of the metaattribute.
label	Optional. Used for the *free* classification to provide a label.
date	Date of the entry

The following classification are available:

change Documentation of change made to the element (may appear multiple times)

comment Additional comments to the element.

Comments which are applicable to the whole document which will be printed at the top of the RST document (may appear multiple times).

desc General plain-language description of the element(unique).

free Required additional attributes: `label`

General purpose metadata field which allows custom labels (may appear multiple times).

intro An introductory text applicable to the whole document which will be printed at the top of the RST document (unique).

All meta items must contain a `date` attribute in the format `YYYYMMDD`.

Entities

Example:

```
<entity>
  <metadata>
    <meta attrib="change" date="20150820">Customer request: Changed label of field to␣
↪Foo</meta>
    <meta attrib="change" date="20150826">Customer request: Changed label of field to␣
↪Bar</meta>
  </metadata>
</entity>
```

Rules

Example:

```
<entity>
  <rule>
    <metadata>
      <meta attrib="desc" date="20150820">Is True when Foo is larger than Bar</meta>
      <meta attrib="change" date="20150826">Customer request: Added rule to check␣
↪value of Foo</meta>
    </metadata>
  </rule>
</entity>
```

Document metadata (<configuration>/Root Metadata)

The main <configuration> element may contain metadata (*root metadata*) which is relevant to the whole document. This information will be formatted as a preamble to the RST output

Example:

```
<configuration>
  <metadata>
    <meta attrib="intro" date="20150820">This text will be rendererd as preamble.</
↪meta>
    <meta attrib="comment" date="20150826">Adapted all labels to fullfill gender␣
↪mainstreaming requirements.</meta>
  </metadata>
  <source>
   ...
  </source>
  ...
</entity>
```

Write custom renderes

Formbar makes it easy to create a custom renderer. All you need to to is to overwrite the *FieldRenderer* class. In most cases you only need to provide a new Template for your field which handles to main rendering. As example see *InfoFieldRenderer* how to set a new template.

Write external validators

A external validator is a simple python callable of the following form:

```
def external_validator(field, data):
    return 16 == data[field]
```

The value 'data' is the converted value dictionary of the form and contains all values of the form. The value 'field' defines the name of the field for which this validation belongs to and also determines on which field the error message will be shown.

The function should return True in case the validation succeeds or either return False or raise an exception in case of validation errors. If the method raises an exception the message of the exception will be used as error message. The validator can be added in two differen ways.

In the formconfig

See *Validator* for more details.

In the view

Another way to add validator to the form is to add the form in the view after the form has been initialized:

```
validator = Validator('fieldname',
                      'Error message',
                      external_validator)
self.form.add_validator(validator)
```

Includes

New in version 0.17.0.

Includes are used to include the content of a different file into the current configuration. The included file may contain *Entity* definition or parts of the *Layout* like a single *Snippet*. The include will be replaced with the content of the of the included file.

A include can be placed at any location of the form configuration and looks like this:

```
<include src="path/to/form/config.xml"/>
```

Attribute	Description
src	Location of the configuration file which should be included
element	Only include a single element form the XML file defined in src. The element is referenced by its id.
entity-prefix	Prefix of the name of the entity fieldname

The include file must be a valid XML file. The content of the include file can be wrapped into a *configuration* tag:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration>
        ... Content ...
</configuration>
```

Supported URL formats

The location of the file can be defined in three ways:

1. As a path relatice to the current XML file.
2. As a absoulte path (Path is begining with an "/").
3. Package relative. Example: *@foo/path/to/form/config.xml*. Formbar will evaluate the path to the package *foo* and replaces the packagage location with the @foo placeholder

Examples

Include options

Includes can be handy to outsource parts of the form definition into its own file. This is especially useful when the outsourced parts are potentially reused in multiple places. Think of a long list of options within a entity:

```
<entity id="country" name="country" type="integer">
    <options>
        <include src="./countries.xml"/>
        <option value="4">Value 4</option>
```

```
    </option>
</entity>
```

The include file looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration>
    <option value="1">Value 1</option>
    <option value="2">Value 2</option>
    <option value="3">Value 3</option>
</configuration>
```

This way you can keep your form definition clean and short and maintain the countries in a separate file.

Inheritance

New in version 0.17.0.

Inheritance can be used to build a form based on another parent form. The inherited form will takeover all properties of the parent form, but can add or modify properties.

An inherited form looks like a usual form, but adds a *inherits* attribute in the *configuration* section:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration inherits="./parent.xml">
   <source>
   <!-- Add or modify entities -->
   </source>
   <form>
   <!-- Add or modify forms -->
   </form>
   <snippet>
   <!-- Add or modify snippets -->
   </snippet>
</configuration>
```

The *source*, *form* and *snippet* section is optional and are only needed if this section needs to be modified.

Inheritance can only be applied on elements in the form which have an *id*. This is because the id is used to identify to elements in the parent form.

To overwrite an element of the parent form you need to add an element with the same id in the inherited form. This will replace the element including all attributes and subelements.

To add new elements, you simply need to at a new element with an id which isn't already defined in the parent form. The new element will be appended at the end of the related section/part of the form.

Removing elements in the inherited form is not supported.

See *supported URL formats* for more information on how to refer to the inherited file.

CHAPTER 3

Usage

You will not need much code to include formbar in your application to be able to render nice forms. Only a few lines of code are needed:

```
from formbar.config import Config, load
from formbar.form import Form
# Simple rendering here, no data submission
# nor validation or saving.
config = Config(load('/path/to/formconfig.xml'))
form_config = config.get_form('example')
form = Form(form_config)
form.render()
```

This is of course just a very easy configuration. See sections below for more options on usage.

The configuration of the form happens in a XML configuration file. See *Form configuration* for more details on options to configura a form. Such a configuration file can be loaded by using the *load()* function to create a form configuration *Config* object.

As the form configuration usually contains more than one form configuration (different configurations for editing, reading or creating new items) you will need get the form configuration for a specific form by calling the `get_form()` method.

This configuration can be used to create a new *Form*.

Form configuration

There are some things which can be configured when initializing the form.

SQLAlchemy support

Formbar can work with mapped SQLAlchemy items. You can provide such an item as *item* attribute while initializing the form.

Translation

Formbar support translation of the following parts of the form:

- Labels
- Error and Warning messages
- Help
- Text

To make the translation work you will need to provide a translation function with the *translate* parameter while initializing a *Form* instance.

This translation function can be any function which behaves like a *gettext* method.

Use Custom renderers

To use custom renderers you will need to provide the classes of the renderes with the *renderer* parameter while initializing a *Form* instance.

The renderers are provided as a dictionary:

```
from your.app.renderer import FooFieldRenderer, BarFieldRenderer
renderers = {
        "foo": FooFieldRenderer
        "bar": BarFieldRenderer
}
```

The key of the dictionary is the name of the *type* form the entities renderer in the the form configuration.

See *Write custom renderes* for more details on how to create a custom renderer.

Use Custom validators

Write me.

Rule evaluation

Rule evaluation on client side is done by sending AJAX requests to a specific URL which takes care of evaluating the submitted rules and returning the correct respose. The URL to which those requests are sent can be provided with the *eval_url* parameter.

Hint: Formbar can be run as server (See serve.py for more details). This server provides such an URL under localhost:8080/evaluate.

CSRF Token

Formbar supports rendering a hidden field in its form which includes the string provided as the *csrf_token* parameter while initializing the form.

The generated field look like this:

```
<input type="hidden" name="csrf_token" value="fe84d264dc7b9f25cce309c275464c1a60f6074a
↪"/>
```

The value can be used on the server side to to some protection against CSRF Attacks.

If no parameter is provided no field will be generated.

Render

See `render()` for more details on options for rendering the form.

Validation

To validate the submitted form data you can use the `validate()` function:

```
if form.validate(request.POST):
    errors = form.get_errors()
    warnings = form.get_warnings()
    submitted = form.submitted_data
    # Handle Error
else:
    warnings = form.get_warnings()
    validated = form.data
    # Handle Success
```

The validation will take care of correct conversation into python types and rule checking. In case the validated succeeds, the *data* attribute of the form will hold the converted python data based on the fields data type.

Saving data

Saving of the converted data after validation is usually done in the application and **not** by formbar. Although formbar provides a `save()` method for mapped SQLAlchemy items but this method is deprecated.

Generate specification

You can generate a specification based on the form configuration and additional *Metadata (Specification)* by using the *formspec.py* command.

`formspec.py` parses Formbar XML configuration files in order to convert them to different formats. Its main purpose is to convert the XML data into a human-readable form specification in RST format.

A specification is generated per form. The command can be invoked like this:

```
python formbar/contrib/formspec.py --title Foo --form update /path/to/foo.xml > foo.
↪rst
```

The *–title* parameter is optional. It will set the topmost heading of the specification to the given titel. Otherwise the name of the form will be used.

The *–form* parameter is optional. On default the "update" form will be used to generate the specification.

CHAPTER 4

API

formbar.config.**load**(*path*)

Return the parsed XML form the given file. The function will load the file located in path and than returns the parsed content.

class formbar.config.**Config**(*tree*)

Class for accessing the form configuration file. It provides methods to get certain elements from the configuration.

get_form(*id*)

Returns a *Form* instance with the configuration for a form with id in the configuration file. If the form can not be found a KeyError is raised.

Id ID of the form in the configuration file

Returns FormConfig instance

class formbar.form.**Form**(*config*, *item=None*, *dbsession=None*, *translate=None*, *change_page_callback={}*, *renderers={}*, *request=None*, *csrf_token=None*, *eval_url=None*, *url_prefix=''*, *locale=None*, *values=None*)

Class for forms. The form will take care for rendering the form, validating the submitted data and saving the data back to the item.

The form must be instanciated with an instance of an Form configuration and optional an SQLAlchemy mapped item.

If an SQLAlchemy mapped item is provided there are some basic validation is done based on the defintion in the database. Further the save method will save the values directly into the database.

If no item was provided than a dummy item will be created with the attributes of the configured fields in the form.

get_errors(*page=None*)

Returns a dictionary of all errors in the form. If page parameter is given, then only the errors for fields on the given page are returned. This dictionary will contain the errors if the validation fails. The key of the dictionary is the fieldname of the field. As a field can have more than one error the value is a list.

Page Dictionary with errors

Returns Dictionary with errors

get_warnings(*page=None*)
Returns a dictionary of all warnings in the form. If page parameter is given, then only the warnings for fields on the given page are returned. This dictionary will contain the warnings if the validation fails. The key of the dictionary is the fieldname of the field. As a field can have more than one warning the value is a list.

Page Name of the page

Returns Dictionary with warnings

render(*values=None*, *page=0*, *buttons=True*, *previous_values=None*, *outline=True*)
Returns the rendererd form as an HTML string.

Values Dictionary with values to be prefilled/overwritten in the rendered form.

Previous_values Dictionary of values of the last saved state of the item. If provided a diff between the current and previous values will be renderered in readonly mode.

Outline Boolean flag to indicate that the outline for pages should be rendered. Defaults to true.

Returns Rendered form.

save()
Will save the validated data back into the item. In case of an SQLAlchemy mapped item the data will be stored into the database. :returns: Item with validated data.

validate(*submitted=None*, *evaluate=True*)
Returns True if the validation succeeds else False. Validation of the data happens in three stages:

1. Prevalidation. Custom rules that are checked before any datatype checks on type conversations are made. 2. Basic type checks and type conversation. Type checks and type conversation is done based on the data type of the field and further constraint defined in the database if the form is instanciated with an SQLAlchemy mapped item. 3. Postvalidation. Custom rules that are checked after the type conversation was done. Note: Postevaluation is only done for successfull converted values. 4. External Validators. External defined checks done on teh converted values. Note: Validators are only called for successfull converted values

All errors are stored in the errors dictionary through the process of validation. After the validation finished the values are stored in the data dictionary. In case there has been errors the dictionary will contain the origin submitted data.

Submitted Dictionary with submitted values.

Returns True or False

class formbar.renderer.**FieldRenderer**(*field*, *translate*)
Renderer for fields. The renderer will build the the HTML for the provided field.

class formbar.renderer.**InfoFieldRenderer**(*field*, *translate*)
A Renderer to render simple fa_field elements

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index