
FormAlchemy Documentation

Release 1.4.3dev

Alexandre Conrad

January 03, 2013

CONTENTS

1	Modules contents	3
1.1	formalchemistry – Imports	3
1.2	Models API	3
1.3	formalchemistry.fields – <i>Fields</i> and <i>Renderers</i>	4
1.4	formalchemistry.forms – <i>FieldSet</i> : Form generation	11
1.5	formalchemistry.tables – <i>Grid</i> : Rendering collections	18
1.6	formalchemistry.validators – Validation stuff	21
1.7	formalchemistry.i18n – Internationalisation	23
1.8	formalchemistry.config – Global configuration	23
1.9	formalchemistry.templates – Template engines	23
1.10	Other customizations	25
1.11	Pylons integration	26
1.12	formalchemistry.ext.couchdb – CouchDB support	27
1.13	formalchemistry.ext.fsblob – File system renderer	27
1.14	formalchemistry.ext.pylons – Pylons extensions	28
1.15	formalchemistry.ext.rdf – rdfalchemy support	34
1.16	formalchemistry.ext.zope – zope.schema support	34
2	Indices and tables	35
3	Changes	37
3.1	1.4.3dev	37
3.2	1.4.2	37
3.3	1.4.1	37
3.4	1.4	37
3.5	1.3.9	38
3.6	1.3.8	38
3.7	1.3.6	38
3.8	1.3.5	38
3.9	1.3.4	38
3.10	1.3.3	38
3.11	1.3.2	39
3.12	1.3.1	39
3.13	1.3	39
3.14	1.2.1	39
3.15	1.2	40
3.16	1.1.1	40
3.17	1.1	40
3.18	1.0.1	40

3.19	1.0	41
3.20	0.5.1	41
3.21	0.5	41
3.22	0.3.1	42
3.23	0.3	42
3.24	0.2	42
3.25	0.1	43
4	Copyright and License	45
Index		47

See Also:

If you use the trunk you may look at a more up to date version of the documentation at <http://docs.formalchemy.org/current/>.

MODULES CONTENTS

1.1 formalchemy – Imports

All *FormAlchemy*'s objects live under the *formalchemy* package

forms related classes:

```
>>> from formalchemy import FieldSet, Field
```

validators:

```
>>> from formalchemy import validators, ValidationError
```

For manual Field definition:

```
>>> from formalchemy import types
```

tables for collection rendering:

```
>>> from formalchemy import Grid
```

Advanced fields customization:

```
>>> from formalchemy import FieldRenderer
```

The above imports are equivalent to:

```
>>> from formalchemy import *
```

1.2 Models API

FormAlchemy is aware of the `__unicode__` and `__html__` methods:

```
class User(Base):
    """A User model"""
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    email = Column(Unicode(40), unique=True, nullable=False)
    password = Column(Unicode(20), nullable=False)
    name = Column(Unicode(30))
    def __unicode__(self):
        """This is used to render the model in a relation field. Must return an
        unicode string."""
```

```
    return self.name
def __html__(self):
    """This is used to render the model in relation field (readonly mode).
    You need need to clean up the html yourself. Use it at your own
    risk."""
    return '<a href="mailto:%s">%s</a>' % (self.email, self.name)
def __repr__(self):
    return '<User %s>' % self.name
```

You can also use the `formalchemy.Column()` wrapper to set some extra options:

1.3 sqlalchemy.fields – *Fields and Renderers*

1.3.1 Fields

1.3.2 Renderers

It is important to note that although these objects are called *renderers*, they are also responsible for deserialization of data received from the web and insertion of those (possibly mangled) values back to the SQLAlchemy object, if any.

They also have to take into consideration that the data used when displaying *can* come either from the *self.params* (the dict-like object received from the web) or from the model. The latter case happens when first displaying a form, and the former when validation triggered an error, and the form is to be re-displayed (and still contain the values you entered).

FieldRenderer

TextFieldRenderer

Render a string field:

```
>>> fs = FieldSet(One)
>>> fs.append(Field(name='text', type=types.String, value='a value'))
```

Edit mode:

```
>>> print fs.text.render()
<input id="One--text" name="One--text" type="text" value="a value" />
```

Read only mode:

```
>>> print fs.text.render_readonly()
a value
```

IntegerFieldRenderer

PasswordFieldRenderer

Render a string field:

```
>>> fs = FieldSet(One)
>>> fs.append(Field(name='passwd').with_renderer>PasswordFieldRenderer))
```


Edit mode:

```
>>> print fs.passwd.render()
<input id="One--passwd" name="One--passwd" type="password" />
```

Read only mode:

```
>>> print fs.passwd.render_readonly()
*****
```

TextAreaFieldRenderer

Render a string field:

```
>>> fs = FieldSet(One)
>>> fs.append(Field(name='text', value='a value').with_renderer(TextAreaFieldRenderer))
```

Edit mode:

```
>>> print fs.text.render()
<textarea id="One--text" name="One--text">a value</textarea>
```

Read only mode:

```
>>> print fs.text.render_readonly()
a value
```

HiddenFieldRenderer

Render a string field:

```
>>> fs = FieldSet(One)
>>> fs.append(Field(name='text', value='h').with_renderer(HiddenFieldRenderer))
```

Edit mode:

```
>>> print fs.render()
<input id="One--text" name="One--text" type="hidden" value="h" />
```

Read only mode:

```
>>> print fs.text.render_readonly()
```

HiddenFieldRendererFactory

CheckBoxFieldRenderer

FileFieldRenderer

DateFieldRenderer

Render a date field:

```
>>> date = datetime(2000, 12, 31, 9, 00)
>>> fs = FieldSet(One)
>>> fs.append(Field(name='date', type=types.Date, value=date))
```

Edit mode:

```
>>> print pretty_html(fs.date.render())
<span id="One--date">
  <select id="One--date__month" name="One--date__month">
    <option value="MM">
      Month
    </option>
    <option value="1">
      January
    </option>
    ...
    <option selected="selected" value="12">
      December
    </option>
  </select>
  <select id="One--date__day" name="One--date__day">
    <option value="DD">
      Day
    </option>
    <option value="1">
      1
    </option>
    ...
    <option selected="selected" value="31">
      31
    </option>
  </select>
  <input id="One--date__year" maxlength="4" name="One--date__year" size="4" type="text" value="2000" />
</span>
```

Read only mode:

```
>>> print fs.date.render_readonly()
2000-12-31
```

TimeFieldRenderer

Render a time field:

```
>>> time = datetime(2000, 12, 31, 9, 03, 30).time()
>>> fs = FieldSet(One)
>>> fs.append(Field(name='time', type=types.Time, value=time))
```

Edit mode:

```
>>> print pretty_html(fs.time.render())
<span id="One--time">
  <select id="One--time__hour" name="One--time__hour">
    <option value="HH">
      HH
    </option>
    <option value="0">
      0
    </option>
    ...
    <option selected="selected" value="9">
      9
```

```

    </option>
    ...
    <option value="23">
        23
    </option>
</select>
:
<select id="One--time__minute" name="One--time__minute">
    <option value="MM">
        MM
    </option>
    <option value="0">
        0
    </option>
    ...
    <option selected="selected" value="3">
        3
    </option>
    ...
    <option value="59">
        59
    </option>
</select>
:
<select id="One--time__second" name="One--time__second">
    <option value="SS">
        SS
    </option>
    <option value="0">
        0
    </option>
    ...
    <option selected="selected" value="30">
        30
    </option>
    ...
    <option value="59">
        59
    </option>
</select>
</span>

```

Read only mode:

```

>>> print fs.time.render_readonly()
09:03:30

```

DateTimeFieldRenderer

Render a datetime field:

```

>>> datetime = datetime(2000, 12, 31, 9, 03, 30)
>>> fs = FieldSet(One)
>>> fs.append(Field(name='datetime', type=types.DateTime, value=datetime))

```

Edit mode:

```
>>> print pretty_html(fs.datetime.render())
<span id="One--datetime">
  <select id="One--datetime__month" name="One--datetime__month">
    <option value="MM">
      Month
    </option>
    . . .
    <option selected="selected" value="12">
      December
    </option>
  </select>
  <select id="One--datetime__day" name="One--datetime__day">
    <option value="DD">
      Day
    </option>
    . . .
    <option selected="selected" value="31">
      31
    </option>
  </select>
  <input id="One--datetime__year" maxlength="4" name="One--datetime__year" size="4" type="text" value="" />
  <select id="One--datetime__hour" name="One--datetime__hour">
    <option value="HH">
      HH
    </option>
    . . .
    <option selected="selected" value="9">
      9
    </option>
  </select>
  :
  <select id="One--datetime__minute" name="One--datetime__minute">
    <option value="MM">
      MM
    </option>
    . . .
    <option selected="selected" value="3">
      3
    </option>
  </select>
  :
  <select id="One--datetime__second" name="One--datetime__second">
    <option value="SS">
      SS
    </option>
    . . .
    <option selected="selected" value="30">
      30
    </option>
  </select>
</span>
```

Read only mode:

```
>>> print fs.datetime.render_readonly()
2000-12-31 09:03:30
```

HiddenDateFieldRenderer

HiddenTimeFieldRenderer

HiddenDateTimeFieldRenderer

RadioSet

CheckBoxSet

SelectFieldRenderer

EscapingReadonlyRenderer

1.3.3 Custom renderer

You can write your own *FieldRenderer* s to customize the widget (input element[s]) used to edit different types of fields...

1. Subclass *FieldRenderer*.
 - (a) Override *render* to return a string containing the HTML input elements desired. Use *self.name* to get a unique name and id for the input element. *self._value* may also be useful if you are not rendering multiple input elements.
 - (b) If you are rendering a custom type (any class you defined yourself), you will need to override *deserialize* as well. *render* turns the user-submitted data into a Python value. (The raw data will be available in *self.field.parent.data*, or you can use *_serialized_value* if it is convenient.) For SQLAlchemy collections, return a list of primary keys, and *FormAlchemy* will take care of turning that into a list of objects. For manually added collections, return a list of values.
 - (c) If you are rendering a builtin type with multiple input elements, override *_serialized_value* to return a single string combining the multiple input pieces. See the source for *DateFieldRenderer* for an example.
2. Update *FieldSet.default_renderers*. *default_renderers* is a dict of *FieldRenderer* subclasses. The default contents of *default_renderers* is:

```
class DefaultRenderers(object):
```

```
    default_renderers = {
        fatypes.String: fields.TextFieldRenderer,
        fatypes.Unicode: fields.TextFieldRenderer,
        fatypes.Text: fields.TextFieldRenderer,
        fatypes.Integer: fields.IntegerFieldRenderer,
        fatypes.Float: fields.FloatFieldRenderer,
        fatypes.Numeric: fields.FloatFieldRenderer,
        fatypes.Interval: fields.IntervalFieldRenderer,
        fatypes.Boolean: fields.CheckBoxFieldRenderer,
        fatypes.DateTime: fields.DateTimeFieldRenderer,
        fatypes.Date: fields.DateFieldRenderer,
        fatypes.Time: fields.TimeFieldRenderer,
        fatypes.LargeBinary: fields.FileFieldRenderer,
        fatypes.List: fields.SelectFieldRenderer,
```

```
fatypes.Set: fields.SelectFieldRenderer,
'dropdown': fields.SelectFieldRenderer,
'checkbox': fields.CheckBoxSet,
'radio': fields.RadioSet,
'password': fields.PasswordFieldRenderer,
'textarea': fields.TextAreaFieldRenderer,
'email': fields.EmailFieldRenderer,
fatypes.HTML5Url: fields.UrlFieldRenderer,
'url': fields.UrlFieldRenderer,
fatypes.HTML5Number: fields.NumberFieldRenderer,
'number': fields.NumberFieldRenderer,
'range': fields.RangeFieldRenderer,
fatypes.HTML5Date: fields.HTML5DateFieldRenderer,
'date': fields.HTML5DateFieldRenderer,
fatypes.HTML5DateTime: fields.HTML5DateTimeFieldRenderer,
'datetime': fields.HTML5DateTimeFieldRenderer,
'datetime_local': fields.LocalDateTimeFieldRenderer,
'month': fields.MonthFieldRender,
'week': fields.WeekFieldRenderer,
fatypes.HTML5Time: fields.HTML5TimeFieldRenderer,
'time': fields.HTML5TimeFieldRenderer,
fatypes.HTML5Color: fields.ColorFieldRenderer,
'color': fields.ColorFieldRenderer,
}
```

For instance, to make *Boolean*s render as select fields with Yes/No options by default, you could write:

```
>>> from formalchemy.fields import SelectFieldRenderer
>>> class BooleanSelectRenderer(SelectFieldRenderer):
...     def render(self, **kwargs):
...         kwargs['options'] = [('Yes', True), ('No', False)]
...         return SelectFieldRenderer.render(self, **kwargs)

>>> FieldSet.default_renderers[types.Boolean] = BooleanSelectRenderer
```

Of course, you can subclass *FieldSet* if you don't want to change the defaults globally.

One more example, this one to use the [jQuery UI DatePicker](#) to render *Date* objects:

```
>>> from formalchemy.fields import FieldRenderer
>>> class DatePickerFieldRenderer(FieldRenderer):
...     def render(self):
...         value= self.value and self.value or ''
...         vars = dict(name=self.name, value=value)
...         return """
...             <input id="% (name)s" name="% (name)s"
...                 type="text" value="% (value)s">
...             <script type="text/javascript">
...                 $ ('#% (name)s').datepicker({dateFormat: 'yy-mm-dd'})
...             </script>
...             """ % vars
```

(Obviously the page template will need to add references to the jquery library and css.)

Another example to render a link field:

```
>>> class LinkFieldRenderer(FieldRenderer):
...     def render(self, **kwargs):
...         """render html for edit mode"""
...         from formalchemy import helpers as h
```

```

...         return h.text_field(self.name, value=self._value, **kwargs)
...     def render_readonly(self, **kwargs):
...         """render html for read only mode"""
...         kwargs = {'value':self.field.raw_value}
...         return '<a href="%s">%s</a>' % kwargs

```

Then bind it to a specific field:

```

>>> from formalchemy.tests import *
>>> fs = FieldSet(One)
>>> fs.append(Field('link', value='http://www.formalchemy.org'))
>>> fs.configure(include=[fs.link.with_renderer(LinkFieldRenderer)])

```

Here is the result for edit mode:

```

>>> print fs.render()
<div>
  <label class="field_opt" for="One--link">
    Link
  </label>
  <input id="One--link" name="One--link" type="text" value="http://www.formalchemy.org" />
</div>
<script type="text/javascript">
  //
document.getElementById("One--link").focus();
//]]&gt;
&lt;/script&gt;
</pre>
</div>
<div data-bbox="112 476 279 491" data-label="Text">
<p>And for read only mode:</p>
</div>
<div data-bbox="112 501 470 693" data-label="Text">
<pre>
&gt;&gt;&gt; fs.readonly = True
&gt;&gt;&gt; print fs.render()
&lt;tbody&gt;
  &lt;tr&gt;
    &lt;td class="field_readonly"&gt;
      Link:
    &lt;/td&gt;
    &lt;td&gt;
      &lt;a href="http://www.formalchemy.org"&gt;
        http://www.formalchemy.org
      &lt;/a&gt;
    &lt;/td&gt;
  &lt;/tr&gt;
&lt;/tbody&gt;
</pre>
</div>
<div data-bbox="112 728 723 750" data-label="Section-Header">
<h2>1.4 formalchemy.forms – <i>FieldSet</i>: Form generation</h2>
</div>
<div data-bbox="112 770 478 789" data-label="Section-Header">
<h3>1.4.1 Configuring and rendering forms</h3>
</div>
<div data-bbox="112 803 530 819" data-label="Text">
<p>In FormAlchemy, forms are rendered using the <i>FieldSet</i> object.</p>
</div>
<div data-bbox="112 825 870 842" data-label="Text">
<p>There are several operations that can be made on a <i>FieldSet</i>. They can be <i>bound</i>, <i>configured</i>, <i>validated</i>, and <i>sync'd</i>.</p>
</div>
<div data-bbox="140 849 649 887" data-label="List-Group">
<ul>
<li>• <i>Binding</i> attaches a model object to the <i>FieldSet</i>.</li>
<li>• <i>Configuration</i> tells the <i>FieldSet</i> which fields to include, in which order, etc.</li>
</ul>
</div>
<div data-bbox="112 931 545 948" data-label="Page-Footer">1.4. formalchemy.forms – <i>FieldSet</i>: Form generation</div>
<div data-bbox="858 931 888 947" data-label="Page-Footer">11</div>
```

- *Validation* checks the form-submitted parameters for correctness against the FieldSet's validators and field definitions.
- *Synchronization* fills the model object with values taken from the web form submission.

Binding

Binding occurs at first on `FieldSet` object creation.

The `FieldSet` object constructor takes its parameters and calls its base class's constructor.

Fields

Each `FieldSet` will have a `Field` created for each attribute of the bound model. Additional `Field` knows how to render itself, and most customization is done by telling a `Field` to modify itself appropriately.

Field-s are accessed simply as attributes of the `FieldSet`:

```
>>> fs = FieldSet(bill)
>>> print fs.name.value
Bill
```

If you have an attribute name that conflicts with a built-in `FieldSet` attribute, you can use `fs[fieldname]` instead. So these are equivalent:

```
>>> fs.name == fs['name']
True
```

Field Modification

Field rendering can be modified with the following methods:

- ***validate(self, validator)***: Add the *validator* function to the list of validation routines to run when the `FieldSet`'s *validate* method is run. Validator functions take one parameter: the value to validate. This value will have already been turned into the appropriate data type for the given `Field` (string, int, float, etc.). It should raise *ValidationException* if validation fails with a message explaining the cause of failure.
- ***required(self)***: Convenience method for *validate(validators.required)*. By default, NOT NULL columns are required. You can only add required-ness, not remove it.
- ***label(self)***: Change the label associated with this field. By default, the field name is used, modified for readability (e.g., 'user_name' -> 'User name').
- ***with_null_as(self, option)***: For optional foreign key fields, render null as the given option tuple of text, value.
- ***with_renderer(self, renderer)***: Change the renderer class used to render this field. Used for one-off renderer changes; if you want to change the renderer for all instances of a `Field` type, modify `FieldSet.default_renderers` instead.
- ***with_metadata(self, **attrs)***: Add/modify some metadata for the `Field`. Use this to attach any metadata to your field. By default, the *instructions* property is used to show additional text below or beside your rendered `Field`.
- ***disabled(self)***: Render the field disabled.
- ***readonly(self)***: Render the field readonly.
- ***hidden(self)***: Render the field hidden. (Value only, no label.)
- ***password(self)***: Render the field as a password input, hiding its value.

- ***textarea(self, size=None)***: Render the field as a textarea.
- ***radio(self, options=None)***: Render the field as a set of radio buttons.
- ***checkbox(self, options=None)***: Render the field as a set of checkboxes.
- ***dropdown(self, options=None, multiple=False, size=5)***: Render the field as an HTML select field. (With the *multiple* option this is not really a ‘dropdown’.)

Methods taking an *options* parameter will accept several ways of specifying those options:

- an iterable of SQLAlchemy objects; *str()* of each object will be the description, and the primary key the value
- a SQLAlchemy query; the query will be executed with *all()* and the objects returned evaluated as above
- an iterable of (description, value) pairs
- a dictionary of {description: value} pairs
- a callable that return one of those cases. Used to evaluate options each time.

Options can be “chained” indefinitely because each modification returns a new `Field` instance, so you can write:

```
>>> fs.append(Field('foo').dropdown(options=[('one', 1), ('two', 2)]).radio())
```

or:

```
>>> fs.configure(options=[fs.name.label('Username').readonly()])
```

Here is a callable example:

```
>>> def custom_query(fs):
...     return fs.session.query(User).filter(User.name=='Bill')
>>> fs3 = FieldSet(bill)
>>> fs3.configure(options=[fs3.name.dropdown(options=custom_query)])
>>> print fs3.name.render()
<select id="User-1-name" name="User-1-name">
<option value="">None</option>
<option value="1">Bill</option>
</select>
```

Manipulating Fields

You can add additional fields not in your SQLAlchemy model with the *append* method, which takes a `Field` object as parameter:

```
>>> fs3 = FieldSet(bill)
>>> fs3.configure(include=[fs3.name, fs3.email])
>>> fs3.append(Field('password', renderer='password'))
>>> fs3.render_fields.keys()
['name', 'email', 'password']
```

You can also *insert* fields. Here we add a country before the password field:

```
>>> fs3.insert(fs3.password, Field('country'))
>>> fs3.render_fields.keys()
['name', 'email', 'country', 'password']
```

And finally, you can *delete* fields:

```
>>> del fs3.country
>>> fs3.render_fields.keys()
['name', 'email', 'password']

>>> del fs3['password']
>>> fs3.render_fields.keys()
['name', 'email']
```

Here is *Field*'s constructor:

Fields to render

The *configure* method specifies a set of attributes to be rendered. By default, all attributes are rendered except primary keys and foreign keys. But, relations **based on** foreign keys **will** be rendered. For example, if an *Order* has a *user_id* FK and a *user* relation based on it, *user* will be rendered (as a select box of *User*'s, by default) but *user_id* will not.

See parameters in `FieldSet.configure()`.

Examples: given a `FieldSet` *fs* bound to a *User* instance as a model with primary key *id* and attributes *name* and *email*, and a relation *orders* of related *Order* objects, the default will be to render *name*, *email*, and *orders*. To render the orders list as checkboxes instead of a select, you could specify:

```
>>> fs2 = fs.bind(bill)
>>> fs2.configure(options=[fs.orders.checkbox()])
```

To render only name and email:

```
>>> fs2 = fs.bind(bill)
>>> fs2.configure(include=[fs.name, fs.email])
```

or:

```
>>> fs2 = fs.bind(bill)
>>> fs2.configure(exclude=[fs.orders])
```

Of course, you can include modifications to a field in the *include* parameter, such as here, to render name and options-as-checkboxes:

```
>>> fs2 = fs.bind(bill)
>>> fs2.configure(include=[fs.name, fs.orders.checkbox()])
```

Rendering

Once you've configured your `FieldSet`, just call the *render* method to get an HTML string suitable for including in your page:

```
>>> fs = FieldSet(bill)
>>> print fs.render()
<div>
  <label class="field_req" for="User-1-email">
    Email
  </label>
  <input id="User-1-email" maxlength="40" name="User-1-email" type="text" value="bill@example.com" />
</div>
<script type="text/javascript">
  //
document.getElementById("User-1-email").focus();</pre></div><div data-bbox="111 931 139 948" data-label="Page-Footer"><hr/>14</div><div data-bbox="644 931 889 949" data-label="Page-Footer">Chapter 1. Modules contents</div>
```

```
//]]>
</script>
<div>
  <label class="field_req" for="User-1-password">
    Password
  </label>
  <input id="User-1-password" maxlength="20" name="User-1-password" type="text" value="1234" />
</div>
<div>
  <label class="field_opt" for="User-1-name">
    Name
  </label>
  <input id="User-1-name" maxlength="30" name="User-1-name" type="text" value="Bill" />
</div>
<div>
  <label class="field_opt" for="User-1-orders">
    Orders
  </label>
  <select id="User-1-orders" multiple="multiple" name="User-1-orders" size="5">
    <option value="2">
      Quantity: 5
    </option>
    <option value="3">
      Quantity: 6
    </option>
    <option selected="selected" value="1">
      Quantity: 10
    </option>
  </select>
</div>
```

Note that there is no *form* element! You must provide that yourself.

You can also render individual fields for more fine-grained control:

```
>>> fs = FieldSet(bill)
>>> print fs.name.render()
<input id="User-1-name" maxlength="30" name="User-1-name" type="text" value="Bill" />
```

1.4.2 Custom FieldSet

You can customize your FieldSet, and create a ready-made derived version for when you need it in your application. For example, you could create one FieldSet per model object in your application.

In this example, we create a FieldSet to edit the *User* model object:

```
from formalchemy import validators
class UserFieldSet(FieldSet):
    """Used to edit users"""
    def __init__(self):
        """Pre-configuration"""
        FieldSet.__init__(self, model.User)

        self.add(Field('passwd1'))
        self.add(Field('passwd2'))
        inc = [self.username,
               self.passwd1.password().label(u'Password'),
               self.passwd2.password().label(u'Confirm') \
```

```
        .validate(validators.passwords_match('passwd1')),
        self.email,
        self.firstname,
        self.lastname,
    ]
    self.configure(include=inc)
```

Then you could use it in your framework controllers as:

```
fs = UserFieldSet().bind(my_user_object, data=request.POST or None)
if request.POST and fs.validate():
    fs.sync()
    fs.model.password = fs.passwd1.value
    ...
```

Another option would be to create a function that generates your FieldSet, perhaps at the top of your controller if it's not to be reused anywhere, otherwise in a central lib for your application. Then you would call your function instead of the `forms.UserFieldSet()` above.

You can use the `.insert`, `.insert_after`, `.append`, `.extend` functions to tweak your FieldSet's composition afterwards. You can also use the `del` keyword on `Field` attributes (like `fs.password`) to remove them from the FieldSet.

You'll probably want to modify the default behavior for fields using the `.set` function on the `Field` attributes directly. This will tweak the objects in-place.

1.4.3 Including data from more than one class

FormAlchemy only supports binding to a single class, but a single class can itself include data from multiple tables. Example:

```
>>> class Order__User(Base):
...     __table__ = join(Order.__table__, User.__table__).alias('__orders__users')
```

Such a class can then be used normally in a FieldSet.

See http://www.sqlalchemy.org/docs/05/mappers.html#advdatamapping_mapper_joins for full details on mapping multiple tables to a single class.

1.4.4 Non-SQLAlchemy forms

You can create a FieldSet from non-SQLAlchemy, new-style (inheriting from *object*) classes, like this:

```
>>> class Manual(object):
...     a = Field()
...     b = Field(type=types.Integer).dropdown([('one', 1), ('two', 2)])

>>> fs = FieldSet(Manual)
```

Field declaration is the same as for adding fields to a SQLAlchemy-based FieldSet, except that you do not give the Field a name (the attribute name is used).

You can still validate and sync a non-SQLAlchemy class instance, but obviously persisting any data post-sync is up to you.

You can also have a look at `formalchemy.ext.zope`.

1.4.5 A note on Sessions

FormAlchemy can save you the most time if you use contextual Sessions: <http://www.sqlalchemy.org/docs/05/session.html#contextual-thread-local-sessions>. Otherwise, you will have to manually pass Session objects when you bind *FieldSet* and *Grid* instances to your data.

1.4.6 Advanced Customization: Form Templates

There are three parts you can customize in a *FieldSet* subclass short of writing your own render method. These are *default_renderers*, and *prettify*. As in:

```
>>> from formalchemy import fields
>>> def myprettify(value):
...     return value

>>> def myrender(**kwargs):
...     return template % kwargs

>>> class MyFieldSet(FieldSet):
...     default_renderers = {
...         types.String: fields.TextFieldRenderer,
...         types.Integer: fields.IntegerFieldRenderer,
...         # ...
...     }
...     prettify = staticmethod(myprettify)
...     _render = staticmethod(myrender)
```

default_renderers is a dict of callables returning a *FieldRenderer*. Usually these will be *FieldRenderer* subclasses, but this is not required. For instance, to make Booleans render as select fields with Yes/No options by default, you could write:

```
>>> class BooleanSelectRenderer(fields.SelectFieldRenderer):
...     def render(self, **kwargs):
...         kwargs['options'] = [('Yes', True), ('No', False)]
...         return fields.SelectFieldRenderer.render(self, **kwargs)

>>> FieldSet.default_renderers[types.Boolean] = BooleanSelectRenderer
```

prettify is a function that, given an attribute name ('user_name') turns it into something usable as an HTML label ('User name').

_render should be a template rendering method, such as *Template.render* from a mako *Template* or *Template.substitute* from a Tempita *Template*.

_render should take as parameters:

- *fieldset* the *FieldSet* object to render

Your template will be particularly interested in these *FieldSet* attributes:

- *render_fields*: the list of fields the user has configured for rendering
- *errors*: a dictionary of validation failures, keyed on field. *errors[None]* are errors applying to the form as a whole rather than a specific field.
- *prettify*: as above
- *focus*: the field to focus

You can also override *prettify* and *_render* on a per-*FieldSet* basis:

```
fs = FieldSet(...)
fs.prettify = myprettify
fs._render = ...
```

The default template is *formalchemy.forms.template_text_tempita*.

1.4.7 Classes definitions

FieldSet

1.5 `formalchemy.tables` – *Grid*: Rendering collections

Besides `FieldSet`, *FormAlchemy* provides *Grid* for editing and rendering multiple instances at once. Most of what you know about `FieldSet` applies to *Grid*, with the following differences to accomodate being bound to multiple objects:

1.5.1 The *Grid* class

1.5.2 Creating

The *Grid* constructor takes parameters (*cls*, *instances=[]*, *session=None*, *data=None*). A significant difference from `FieldSet` is that the first argument must *_always_* be a mapped class, e.g., *User*. *instances* is the objects to render, which must all be of the given type. The other parameters are the same as in `FieldSet`.

1.5.3 Binding

Grid *bind* and *rebind* methods are similar to those methods in `FieldSet`, except they take an iterable *instances* instead of an instance *model*. Thus, the full signature is (*instances*, *session=None*, *data=None*).

1.5.4 Configuration

The *Grid* *configure* method takes the same arguments as `FieldSet` (*pk*, *exclude*, *include*, *options*, *readonly*), except there is no *focus* argument.

1.5.5 Validation and Sync

These are the same as in `FieldSet`, except that you can also call *sync_one(instance)* to sync a single one of the instances that are bound to the *Grid*.

The *Grid* *errors* attribute is a dictionary keyed by bound instance, whose value is similar to the *errors* from a `FieldSet`, that is, a dictionary whose keys are *Field*'s, and whose values are *ValidationError* instances.

1.5.6 Customizing *Grid*

Overriding *Grid* rendering is similar to `FieldSet`. The differences are:

- The default templates take a *collection* parameter instead of *fieldset*, which is the instance of *Grid* to render

- The instances given to the collection are available in *collection.rows*; to access the fields of each single row, call *_set_active(row)*, then access *render_fields*.

The default templates are *formalchemy.tables.template_grid_readonly* and *formalchemy.tables.template_grid*.

1.5.7 Usage

You need some imports:

```
>>> from formalchemy.tables import *
```

Then you can initialize a *Grid* and bind it to a list of row instance:

```
>>> tc = Grid(User, [bill])
>>> tc.configure(readonly=True)
```

This will render instances as an html table:

```
>>> print tc.render()
<thead>
<tr>
<th>
    Email
</th>
<th>
    Password
</th>
<th>
    Name
</th>
<th>
    Orders
</th>
</tr>
</thead>
<tbody>
<tr class="even">
<td>
    bill@example.com
</td>
<td>
    1234
</td>
<td>
    Bill
</td>
<td>
    Quantity: 10
</td>
</tr>
</tbody>
```

You can also add a field to the *Grid* manually:

```
>>> tc2 = Grid(User, [bill, john])
>>> tc2.append(Field('link', type=types.String, value=lambda item: '<a href=%d>link</a>' % item.id))
>>> tc2.configure(readonly=True)
>>> print tc2.render()
<thead>
```

```
<tr>
  <th>
    Email
  </th>
  <th>
    Password
  </th>
  <th>
    Name
  </th>
  <th>
    Orders
  </th>
  <th>
    Link
  </th>
</tr>
</thead>
<tbody>
<tr class="even">
  <td>
    bill@example.com
  </td>
  <td>
    1234
  </td>
  <td>
    Bill
  </td>
  <td>
    Quantity: 10
  </td>
  <td>
    <a href="1">
      link
    </a>
  </td>
</tr>
<tr class="odd">
  <td>
    john@example.com
  </td>
  <td>
    5678
  </td>
  <td>
    John
  </td>
  <td>
    Quantity: 5, Quantity: 6
  </td>
  <td>
    <a href="2">
      link
    </a>
  </td>
</tr>
</tbody>
```


You can provide a dict as new values:

```
>>> g = Grid(User, [bill, john], data={'User-1-email': 'bill@example.com', 'User-1-password': '1234'}
```

Validation work like *Fieldset*:

```
>>> g.validate()
True
```

Sync too:

```
>>> g.sync()
>>> session.flush()
>>> session.refresh(john)
>>> john.email == 'john@example.com'
True
>>> session.rollback()
```

1.6 formalchemy.validators – Validation stuff

To validate data, you must bind it to your *FieldSet* along with the SQLAlchemy model. Normally, you will retrieve *data* from a dict:

```
>>> from formalchemy.tests import User, bill
>>> from formalchemy.forms import FieldSet
>>> fs = FieldSet(User)
>>> fs.configure(include=[fs.name]) # we only use the name field here
>>> fs.rebind(bill, data={'User-1-name': 'Sam'})
```

Validation is performed simply by invoking *fs.validate()*, which returns *True* if validation was successful, and *False* otherwise. Validation functions are added with the *validate* method, described above.

If validation fails, *fs.errors* will be populated. *errors* is a dictionary of validation failures, and is always empty before *validate()* is run. Dictionary keys are attributes; values are lists of messages given to *ValidationException*. Global errors (not specific to a single attribute) are under the key *None*.

Rendering a *FieldSet* with errors will result in error messages being displayed inline. Here's what this looks like for a required field that was not supplied with a value:

```
<div>
  <label class="field_req" for="foo">
    Foo
  </label>
  <input id="foo" name="foo" type="text" value="" />
  <span class="field_error">
    Please enter a value
  </span>
</div>
```

If validation succeeds, you can have *FormAlchemy* put the submitted data back into the bound model object with *fs.sync()*. (If you bound to a class instead of an object, the class will be instantiated for you.) The object will be placed into the current session, if one exists:

```
>>> if fs.validate(): fs.sync()
>>> print bill.name
Sam
```

1.6.1 Exception

All validators raise a *ValidationError* if the validation failed.

exception `ValidationError`

1.6.2 Validators

formalchemy.validators contains two types of functions: validation functions that can be used directly, and validation function *_generators_* that *_return_* a validation function satisfying some condition. E.g., *validators.maxlength(30)* will return a validation function that can then be passed to *validate*.

```
>>> from sqlalchemy.validators import *
```

Validation Functions

A validation function is simply a function that, given a value, raises a *ValidationError* if it is invalid.

Function generators

1.6.3 Write your own validator

You can write your own validator, with the following function signature. The *field* parameter will be the *Field* object being validated (and though its *parent* attribute, the *FieldSet*:

```
>>> def negative(value, field):
...     if not (isinstance(value, int) and value < 0):
...         raise ValidationError('Value must be less than 0')
```

Then bind it to a field:

```
>>> from sqlalchemy import types
>>> fs = FieldSet(One)
>>> fs.append(Field('number', type=types.Integer))
>>> fs.configure(include=[fs.number.validate(negative)])
```

Then it should work:

```
>>> fs.rebind(One, data={'One--number': '-2'})
>>> fs.validate()
True

>>> fs.rebind(One, data={'One--number': '2'})
>>> fs.validate()
False
```

You can also use the *field* positional argument to compare with some other fields in the same *FieldSet* if you know this will be contained in a *FieldSet*, for example:

```
>>> def passwd2_validator(value, field):
...     if field.parent.passwd1.value != value:
...         raise validators.ValidationError('Password do not match')
```

The *FieldSet.errors* and *Field.errors* attributes contain your custom error message:

```
>>> fs.errors
{AttributeField(number): ['Value must be less than 0']}

>>> fs.number.errors
['Value must be less than 0']
```

1.7 formalchemy.i18n – Internationalisation

FormAlchemy is able to render error messages in your own language. You just need to provide a *lang* attribute to the render method:

```
>>> html_fr = fs.render(lang='fr')
```

If you use *Pylons* the language is retrieved from *pylons.i18n.get_lang()* so the *lang* attribute become optional.

At the moment only the french translation is available.

You have to checkout the source (<http://code.google.com/p/formalchemy/source/checkout>) and install *FormAlchemy* in develop mode to add a new translation:

```
$ cd FormAlchemy && python setup.py develop
```

Then install Babel with easy_install:

```
$ easy_install Babel
```

You are now able to initialize a new catalog

```
$ python setup.py init_catalog -l <lang>
```

Where *<lang>* is your language code. This will generate a new file named *formalchemy/i18n/<lang>/LC_MESSAGES/formalchemy.po*

Replace all the *msgstr* in the new *.po* file with your translated messages and compile the catalogs:

```
$ python setup.py compile_catalog
```

Now the new language is available. Last step, send your *.po* to the [<http://groups.google.com/group/formalchemy> project list] !

1.8 formalchemy.config – Global configuration

1.9 formalchemy.templates – Template engines

1.9.1 Available engines

1.9.2 Base class

1.9.3 Customize templates

You can override the default template by adding a directory for your project which will contain the templates. The engine will scan the directory and try to load templates from it. If he can't, the default templates are used.

Here is an example:

```
>>> ls(mako_templates_dir)
- fieldset.mako

>>> cat(mako_templates_dir, 'fieldset.mako')
<ul>
%for field in fieldset.render_fields.itervalues():
<li>${field.name}</li>
%endfor
</ul>
```

Then you can override the default mako templates:

```
>>> from formalchemy import config
>>> from formalchemy import templates
>>> config.engine = templates.MakoEngine(
...     directories=[mako_templates_dir],
...     input_encoding='utf-8', output_encoding='utf-8')
```

And see the result:

```
>>> print FieldSet(User).render()
<ul>
<li>email</li>
<li>password</li>
<li>name</li>
<li>orders</li>
</ul>
```

Same with genshi except that formalchemy don't provide default templates:

```
>>> cat(genshi_templates_dir, 'fieldset.html')
<ul xmlns:py="http://genshi.edgewall.org/">
<li py:for="field in fieldset.render_fields.itervalues()">${field.name}</li>
</ul>
```

```
>>> config.engine = templates.GenshiEngine(directories=[genshi_templates_dir])
```

And same the result of course:

```
>>> print FieldSet(User).render()
<ul>
<li>email</li><li>password</li><li>name</li><li>orders</li>
</ul>
```

1.9.4 Write your own engine

You need to subclass the TemplateEngine:

```
>>> class MyEngine(TemplateEngine):
...     def render(self, template_name, **kw):
...         return 'It works !'
```

You can use it for a specific FieldSet:

```
>>> fs = FieldSet(User)
>>> fs.engine = MyEngine()
>>> print fs.render()
It works !
```

You can also override the engine in a subclass:

```
>>> class MyFieldSet(FieldSet):  
...     engine = MyEngine()
```

Or set it as the global engine with formalchemy's config:

```
>>> from formalchemy import config  
>>> config.engine = MyEngine()
```

It should be available for all FieldSet:

```
>>> print FieldSet(User).render()  
It works !
```

1.10 Other customizations

1.10.1 Customization: CSS

FormAlchemy uses the following CSS classes:

- *fieldset_error*: class for a div containing a “global” error
- *field_error*: class for a span containing an error from a single *Field*
- *field_req*: class for a label for a required field
- *field_opt*: class for a label for an optional field
- *field_readonly*: class for the td of the ‘label’ for a field in a readonly *FieldSet* table
- *grid_error*: class for a span containing an error from a single *Field* in a *Grid*

Here is some basic CSS for aligning your forms nicely:

```
label {  
    float: left;  
    text-align: right;  
    margin-right: 1em;  
    width: 10em;  
}  
  
form div {  
    margin: 0.5em;  
    float: left;  
    width: 100%;  
}  
  
form input[type="submit"] {  
    margin-top: 1em;  
    margin-left: 9em;  
}
```

1.11 Pylons integration

1.11.1 Bootstrap your project

FormAlchemy come with a subclass of the *Pylons* template. If you have *Pylons* and *FormAlchemy* installed you should see that:

```
$ paster create --list-templates
Available templates:
  basic_package:    A basic setuptools-enabled package
  paste_deploy:    A web application deployed through paste.deploy
  pylons:          Pylons application template
  pylons_fa:       Pylons application template with formalchemy support
  pylons_minimal:  Pylons minimal application template
```

To bootstrap a new *Pylons* project with *FormAlchemy* support enable just run:

```
$ paster create -t pylons_fa pylonsapp
```

1.11.2 Using forms in controllers

Imagine you have a *Foo* model in your *model/__init__.py* then your controller can look like this:

```
import logging
from pylons import request, response, session, url, tmpl_context as c
from pylons.controllers.util import abort, redirect
from pylonsapp.lib.base import BaseController, render
from pylonsapp.model import meta
from pylonsapp import model
from pylonsapp.forms import FieldSet

log = logging.getLogger(__name__)

Foo = FieldSet(model.Foo)
Foo.configure(options=[Foo.bar.label('This is the bar field')])

class BasicController(BaseController):

    def index(self, id=None):
        if id:
            record = meta.Session.query(model.Foo).filter_by(id=id).first()
        else:
            record = model.Foo()
        assert record is not None, repr(id)
        c.fs = Foo.bind(record, data=request.POST or None)
        if request.POST and c.fs.validate():
            c.fs.sync()
            if id:
                meta.Session.update(record)
            else:
                meta.Session.add(record)
            meta.Session.commit()
            redirect(url.current(id=record.id))
        return render('/form.mako')
```

If you have a lot of fieldset and configuration stuff and want to use them in different controller, then you can use

the *forms/* module to put your fieldsets. This is a standard and allow you to use the `formalchemy.ext.pylons` extension

You can also have a look at the *RESTful Controller*

1.12 formalchemy.ext.couchdb – CouchDB support

This module provides a subclass of `FieldSet` to support `couchdbkit`'s schema.

1.12.1 Usage

1.12.2 Classes definitions

`FieldSet`

`Grid`

`Session`

`Query`

1.13 formalchemy.ext.fsblob – File system renderer

This extension is used to store binary files/images on filesystem and only store the file path in the database.

This page present a Pylons integration but it should work on most framework.

1.13.1 Renderers

1.13.2 Usage

You must override the *storage_path* attribute:

```
# -*- coding: utf-8 -*-
from pylons import config
from pylonsapp import model
from pylonsapp.forms import FieldSet
from formalchemy.ext.fsblob import FileFieldRenderer

Files = FieldSet(model.Files)
Files.configure(options=[Files.path.with_renderer(
    FileFieldRenderer.new(
        storage_path=config['app_conf']['storage_path'],
        url_prefix='/' ) )])
```

As you can see, you can use it like all *fields* in the *.configure* method.

1.13.3 Validators

Work like all validators.

1.13.4 Sample app

You can have a look at the complete [source](#) of the application used for FA's testing.

1.14 `formalchemy.ext.pylons` – Pylons extensions

1.14.1 Administration interface

Purpose

The Pylons administration interface provides a simple way to enable CRUD (create, retrieve, update, delete) operations on your SQLAlchemy models, with a high degree of customizability.

Sample model listing:









Sample model overview page:

Owner

Models

Modified Owner 2

Existing objects

Name	Animals	Edit	Delete
gawel	minou		
jbellis			
aconrad			

+

Related types

Animal

Sample model creation page:

New object Owner

Models

New object Owner

Name

gael

Animals

minou

Envoyer

Cancel

Setup

First, generate a controller in your application:

```
$ paster controller admin
```

Next, edit your controllers/admin.py, replacing pylonsapp with your application name:

```
import logging
from formalchemy.ext.pylons.controller import ModelsController
from webhelpers.paginate import Page
from pylonsapp.lib.base import BaseController, render
from pylonsapp import model
from pylonsapp import forms
from pylonsapp.model import meta

log = logging.getLogger(__name__)

class AdminControllerBase(BaseController):
    model = model # where your SQLAlchemy mappers are
    forms = forms # module containing FormAlchemy fieldsets definitions
    def Session(self): # Session factory
        return meta.Session

    ## customize the query for a model listing
    # def get_page(self):
    #     if self.model_name == 'Foo':
    #         return Page(meta.Session.query(model.Foo).order_by(model.Foo.bar)
    #     return super(AdminControllerBase, self).get_page()

AdminController = ModelsController(AdminControllerBase,
                                   prefix_name='admin',
                                   member_name='model',
                                   collection_name='models',
                                   )
```

Now you need to configure your routing. As an example here is the routing.py used for testing the UI. Check fa_static and the /admin part:

```
"""Routes configuration

The more specific and detailed routes should be defined first so they
may take precedent over the more generic routes. For more information
refer to the routes manual at http://routes.groovie.org/docs/
"""
from routes import Mapper

def make_map(config):
    """Create, configure and return the routes Mapper"""
    map = Mapper(directory=config['pylons.paths']['controllers'],
                  always_scan=config['debug'])
    map.minimization = False

    # The ErrorController route (handles 404/500 error pages); it should
    # likely stay at the top, ensuring it can always be resolved
    map.connect('/error/{action}', controller='error')
    map.connect('/error/{action}/{id}', controller='error')

    # CUSTOM ROUTES HERE
    # Map the /admin url to FA's AdminController
    # Map static files
    map.connect('fa_static', '/admin/_static/{path_info:.*}', controller='admin', action='static')
    # Index page
```

```

map.connect('admin', '/admin', controller='admin', action='models')
map.connect('formatted_admin', '/admin.json', controller='admin', action='models', format='json')
# Models
map.resource('model', 'models', path_prefix='/admin/{model_name}', controller='admin')

# serve couchdb's Pets as resource
# Index page
map.connect('couchdb', '/couchdb', controller='couchdb', action='models')
# Model resources
map.resource('node', 'nodes', path_prefix='/couchdb/{model_name}', controller='couchdb')

# serve Owner Model as resource
map.resource('owner', 'owners')

map.connect('/{controller}/{action}')
map.connect('/{controller}/{action}/{id}')

return map

```

All done! Now you can go to the `/admin/` url.

Customization

`ModelsController` creates a new class having `AdminControllerBase` and the internal FA models controller (`_ModelsController`) as its parent classes, in that order.

So, you can do simple customization just by overriding the `_ModelsController` methods in `AdminControllerBase`, e.g.,:

```

class AdminControllerBase(BaseController):
    ...

    @auth_required
    def edit(self, *args, **kwargs):
        return super(AdminControllerBase, self).edit(*args, **kwargs)

```

To customize the forms used to list and edit your objects, create a module *yourapp.forms* and specify that as the forms module in `AdminController`. In this module, create `FieldSet` (for create and edit forms) and `Grid` (for object lists) instances for the models you wish to customize. (The *Grids* will automatically get edit and delete links added, and be made readonly.)

See [forms](#) for details on form configuration.

API

Troubleshooting

If you don't see all your models on the top-level admin page, you'll need to import them into your model module, or tell *FormAlchemy* the correct module to look in (the "model =" line in the controller class you created). In particular, *FormAlchemy* does not recursively scan for models, so if you have models in e.g., *model/foo.py*, you will want to add *from foo import ** in *model/__init__.py*.

Sample app

You can have a look at the complete [source](#) of the application used for FA's testing.

1.14.2 RESTful controller

This module provide a RESTful controller for formalchemy's FieldSets.

You can use your fieldset as a REST resource. And yes, it's also work with JSON.

Usage

Use the FieldSetController to wrap your Pylons controller:

```
import logging

from pylons import request, response, session, url, tmpl_context as c
from pylons.controllers.util import abort, redirect

from pylonsapp.lib.base import BaseController, render
from pylonsapp import model
from pylonsapp.model import meta

from formalchemy.ext.pylons.controller import RESTController

log = logging.getLogger(__name__)

class OwnersController(BaseController):

    def Session(self):
        return meta.Session

    def get_model(self):
        return model.Owner

OwnersController = RESTController(OwnersController, 'owner', 'owners')
```

Add this to your config/routing.py:

```
map.resource('owner', 'owners')
```

Customisation

You can override the following methods:

Here is a customisation sample to use CouchDB as backend using the ModelsController (≈ API):

```
__doc__ = """This is an example on ow to setup a CRUD UI with couchdb as
backend"""
import os
import logging
import pylonsapp
from couchdbkit import *
from webhelpers.paginate import Page
from pylonsapp.lib.base import BaseController, render
from couchdbkit.loaders import FileSystemDocsLoader
```

```
from formalchemy.ext import couchdb
from formalchemy.ext.pylons.controller import ModelsController
```

```
log = logging.getLogger(__name__)
```

```
class Person(couchdb.Document):
    """A Person node"""
    name = StringProperty(required=True)
    def __unicode__(self):
        return self.name or u''
```

```
class Pet(couchdb.Document):
    """A Pet node"""
    name = StringProperty(required=True)
    type = StringProperty(required=True)
    birthdate = DateProperty(auto_now=True)
    weight_in_pounds = IntegerProperty(default=0)
    spayed_or_neutered = BooleanProperty()
    owner = SchemaListProperty(Person)
    def __unicode__(self):
        return self.name or u''
```

```
# You don't need a try/except. This is just to allow to run FA's tests without
# couchdb installed. Btw this have to be in another place in your app. eg: you
# don't need to sync views each time the controller is loaded.
```

```
try:
    server = Server()
    if server: pass
except:
    server = None
else:
    db = server.get_or_create_db('formalchemy_test')

    design_docs = os.path.join(os.path.dirname(pylonsapp.__file__), '_design')
    loader = FileSystemDocsLoader(design_docs)
    loader.sync(db, verbose=True)

    contain(db, Pet, Person)
```

```
class CouchdbController(BaseController):

    # override default classes to use couchdb fieldsets
    FieldSet = couchdb.FieldSet
    Grid = couchdb.Grid
    model = [Person, Pet]

    def Session(self):
        """return a formalchemy.ext.couchdb.Session"""
        return couchdb.Session(db)
```

```
CouchdbController = ModelsController(CouchdbController, prefix_name='couchdb', member_name='node', c
```

Helpers

1.15 `formalchemy.ext.rdf` – `rdfalchemy` support

1.15.1 Classes definitions

Field

FieldSet

Grid

1.16 `formalchemy.ext.zope` – `zope.schema` support

1.16.1 Classes definitions

Field

FieldSet

Grid

Utilities

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

CHANGES

3.1 1.4.3dev

- Allow fields to set their *readonly* status to False as well as True.

3.2 1.4.2

- WebOb1.2 compat
- Add some HTML5 renderer
- Improve fsblob deletion. issue 16
- Add support for fanstatic in pylons (thanks to Bruno Binet aka inneos)

3.3 1.4.1

- Implemented WebOb-like request passing to FieldSet directly.
- Also implemented request passing to Grid
- Added support to set *.html_options* with `Field.set(html={ 'some': 'thing' })`
- Added support for `set(validators=[validator1, validator2])` which adds the specified validators.
- Fixed the `set(null_as=...)`, was `nul_as` and badly wired in.
- Improved documentation for the `Field.set()` method
- Support `zope.schema.Password`
- Fix issues 9, 10, 11, 12

3.4 1.4

- Fix issue 5, 7
- Allow to binf form to a webob like request
- Add Column wrapper to store some form options in models
- Field label translation

3.5 1.3.9

- fix unicode issue with non webob based framework

3.6 1.3.8

- use webob.multidict as `data`. This will improve unicode handling in the future (eg: py3k migration). WebOb is now a dependency.
- add `to_dict()` method and `.bind(with_prefix=True/False)` to help to work with json data
- improve `ext.fsblob`. files are wrote on the file system using `shutil.copyfileobj` from the `cgi.FieldStorage` field
- Add a `HiddenFieldRendererFactory` and allow to hide Date/Time fields via `.hidden()` and `.set(hidden=True)` (Thanks to tarek to put this idea in my brain)
- added german translation (thanks @disko for pull request)
- fix issue 1, 2 (on github)

3.7 1.3.6

- fix issues 150, 151, 153, 161, 162
- Added `field.label()` and `field.label_tags()`
- Major refactoring. The `base` module no longer existe.

3.8 1.3.5

- No longer use Binary type. Use LargeBinary instead.
- fix issues 145, 147

3.9 1.3.4

- `renderer._value` is deprecated. Use `renderer.value`
- added `renderer.raw_value`
- `Model.__html__()` (if any) is used to render model in `read_only` mode.

3.10 1.3.3

- Added `.insert_after(field, new_field)` to the `FieldSet` object. Same as `.insert()`, except it adds it after the specified field.
- Docs improvements
- Fix 131 to 137

3.11 1.3.2

- Added *.value_objects* to both Field and FieldRenderer objects. Returns the objects instead of list of primary keys when working with ForeignKeys.
- add IntervalFieldRenderer
- switch back to WebHelpers
- add Hungarian translation (125)
- fix bug with latest version of couchdbkit
- update paster template to Pylons 1.0b1
- fix issues 123, 124, 127, 128

3.12 1.3.1

- include css in MANIFEST.in

3.13 1.3

- new controllers to generate CRUD interfaces based on pylons RESTController
- couchdb support improvement (allow to use RESTController)
- Experimental RDFAlchemy support
- Add date formats to config module.
- add fs.copy()
- zope.schema.List and zope.schema.Choice support (thanks to Christophe Combelles)
- fix issues 107, 113, 114, 117, 118
- css improvement for pylons admin interface

3.14 1.2.1

- Added fs.append(field) fs.insert(field, new_field) and del fs.field to Fieldset. fs.add() is deprecated.
- Added field.set() to modify the field inplace.
- bug fixes: issues 70, 80, 82, 97
- added spanish translation (thanks to robarago)
- added the *.with_html* method to *AbstractField* which will be passed to the renderers, allowing to add some HTML attributes to rendered HTML tags. Removed html_options from render method. (See issue #60)
- validators are now passed as second argument the *field* being validated. WARN: this will mean adding the parameter to your functions to be backwards compatible. The validator function signature changed from *myfunc(value)* to *myfunc(value, field=None)*.
- ext.couchdb now use couchdbkit instead of py-simplecouchdb

- added the `.with_metadata` method to *AbstractField* which allows you to add metadata to your field. The difference with `.with_html()` is that the attributes passed in will not be rendered in the HTML element, but are there only to be used in your templates, to tweak the output according to those properties. See docs/forms.txt

3.15 1.2

- add a paster template to bootstrap a pylons project with FA support enabled
- much sexier look for admin interface
- performance improvements
- non-SQLA Fields are no longer considered “experimental”
- `with_null_as` feature (see issue #52)
- `prefix` feature (see issue #59)
- when auto-querying for option values, the `order_by` given on the relation is used, if any
- synonym awareness (you don’t have to manually exclude the shadowed attribute)
- `ext.couchdb` (experimental)

3.16 1.1.1

- bug fixes: issues 36, 37, 38, 39, 40, 41, 42, 43, 45, 46, 47, 49
- added `EscapingReadonlyRenderer`
- add `Date*Renderer` translation

3.17 1.1

- `formalchemy.ext.pylons.admin` added; see <http://docs.formalchemy.org/ext/pylons.html>
- `formalchemy.ext.fsblob` added; see <http://docs.formalchemy.org/ext/fsblob.html>
- support for composite primary keys
- support for composite foreign keys of primitive types
- `model` argument now optional for `FieldSet.bind`
- apply i18n to Grid labels
- documentation improvement
- bug fixes

3.18 1.0.1

- Bug fixes

3.19 1.0

- i18n support (gael.pasgrimaud)
- file upload support (gael.pasgrimaud)
- mapper property alias support (gael.pasgrimaud)
- add *kwargs* to FieldSet and Grid render methods, which are passed on to the template. this allows easy custom template use w/o having to subclass. (lbruno)
- removed query_options. Just pass the query as the argument to the options parameter, and FA will turn it into (description, value) pairs. FA will also accept an iterable of objects as a value to the options parameter.
- unicode(object) is used as the default option description, not str(object). (Before, unicode was only used if the engine had convert_unicode turned on.) This is more consistent with normal SA behavior.
- added sanity checks to disallow getting into an inconsistent state. notably, binding to an object that belongs to a session but does NOT have a primary key set is not allowed. workaround: bind to the class, and FA will instantiate it and take it out of the session [until sync()]. Then you can pull that instance out as the .model attribute.
- sync() will save model to session, if necessary
- add Field.with_renderer
- allow manually-added fields to pull their value from the bound model
- fs.[field] returns the configured version of the field, not the unconfigured. fs.fields renamed to fs._fields. Added Field.reset() to deepcopy the unconfigured version.
- explicit renderers required for custom types (FieldRenderer.render removed)
- new documentation <http://docs.formalchemy.org> (gael.pasgrimaud)
- bug fixes

3.20 0.5.1

- Synonym support
- Bug fixes

3.21 0.5

- Composite field and custom type support
- Joined table support
- Grid (companion to FieldSet) renders and edits multiple instances at once.
- readonly support for FieldSet (replacing undocumented Table), Grid (replacing TableCollection)
- FieldSet can render Fields from a non-mapped class (experimental)
- Saner (backwards-incompatible, but easy port) widget (FieldRenderer) API
- FieldSet.render_fields is now an OrderedDict like FieldSet.fields. Use render_fields.[iter]values() to get an iterable like the old render_fields.

- Bug fixes

3.22 0.3.1

- Bug fixes
- Much better DateTime support
- Extensible widget API (want to use your favorite date picker instead? No problem.)
- *FieldRenderer* is now part of *from formalchemy import ** for use here
- Minor changes to template API (details in documentation). Does not affect you unless you already wrote a custom template
- order fields by declared order as much as possible, instead of alphabetical, when include= is absent
- Validator suite fleshed out (minlength, maxlength, regex, email, currency)
- Added doc sections on widget API and validation functions

3.23 0.3

- Completely new API, based on Fields instead of column names
- Support manually added Fields, not just attributes from the SA model
- Relations (a FK will be rendered with a dropdown of related objects)
- Validation + sync
- Template-based rendering for greater customizability. Tempita is included; Mako is detected and used if present
- WebHelpers is no longer a dependency; the small parts FA needs have been moved into helpers.py. (This was prompted by WebHelpers 0.6 breaking backwards compatibility in nontrivial ways.)
- Pervasive docstrings
- Preliminary SA 0.5 support
- Regression test suite

3.24 0.2

- Added 'disable', 'disable_pk', 'disable_fk' options.
- Fixed a bug where 'readonly*' options only worked for 'password' fields.
- Added 'date', 'time' and 'datetime' options for date/time fields formatting.
- Added 'bool_as_radio' option.
- Added a hack to force browsers to POST unchecked checkboxes.
- Fixed a bug where 'opts' from the 'dropdown' option is no longer rendered as an attribute of the <select> tag.
- Fixed a compatibility issue with SQLAlchemy 0.4.1. The 'foreign_key' Column attribute is now 'foreign_keys'.
- Added 'fieldset' option.

- Added ‘include’ option. Patch from Adam Gomaa.
- Added ‘textarea’ option. Additionnal patch provided by Adam Gomaa for passing native tuple of intergers as *size* argument value.
- Added new experimental, little customizable, ‘TableItem’ and ‘TableCollection’. TableItem renders a table from a bound model. TableCollection renders a table from a collection of items that are of the same class than the bound model: TableCollection(bind=client, collection=client_list). The bound model can be a non-instantiated mapped class.
- Removed NullType type column detection for now, as it seems to be a SA 0.4 only thing. What would a NullType HTML field represent anyway?
- FieldSet now returns fields embedded in <fieldset> HTML tags.
- Implemented the ‘legend’ option for FieldSet to provide an optional and customizable <legend> tag. FieldSet uses the bound model’s class name as the legend by default. The legend can be customized by passing a string to the ‘legend’ option: *legend=‘My legend’*. The fieldset can be legend-less by passing *legend=False*.
- Big core changes. Splitted the single formalchemy.py module into a formalchemy package. More classes, more flexibility. Plus, we’re now using model-level and column-level rendering engines: ‘ModelRenderer’ and ‘FieldRenderer’.
- ‘ModelRender’ and ‘FieldRender’ allows you to render a whole model (like FieldSet, but without the field-set/legend tags) or a single column.
- FieldSet now uses ‘ModelRenderer’.
- Added new experimental, little customizable, non-form related, ‘TableItem’ and ‘TableCollection’. TableItem renders a table from a bound model. TableCollection renders a table from a collection of items that are of the same class than the bound model: TableCollection(bind=client, collection=client_list). The bound model can be a non-instantiated mapped class.

3.25 0.1

- Initial release.

COPYRIGHT AND LICENSE

The MIT License

Copyright (c) 2007 Alexandre Conrad

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

INDEX

V

ValidationError, [22](#)