pyramid_simpleform

Version 0.7.dev0

Dan Jacob

April 22, 2022

Contents

pyramid_simpleform, as the name implies, is a simple form validation and rendering library. It's intended to replace the old `@validate` decorator from Pylons with a form handling pattern inspired by Django forms, WTForms and Flatland. However it's also intended for use with the Pyramid framework and uses FormEncode for most of the heavy lifting. It's therefore assumed you are already familiar with FormEncode.

Part I

Installation

Install using **pip install pyramid_simpleform** or **easy_install pyramid_simpleform**.

If installing from source, untar/unzip, cd into the directory and do **python setup.py install**.

The source repository is on GitHub. Please report any bugs, issues or queries there.

Part II

Getting started

pyramid_simpleform doesn't require any special configuration using ZCML or otherwise. You just create FormEncode schemas or validators for your application as usual, and wrap them in a special **Form** class. The **Form** provides a number of helper methods to make form handling as painless, and still flexible, as possible.

Here is a typical (truncated) example:

```
from formencode import Schema, validators
from pyramid_simpleform import Form
from pyramid_simpleform.renderers import FormRenderer

class MySchema(Schema):

    allow_extra_fields = True
    filter_extra_fields = True

    name = validators.UnicodeString(max=5)
    value = validators.Int()

class MyModel(object):
    """
    Mock object.
    """

    pass


@view_config(name='edit',
             renderer='edit.html')
def edit(self):

    item_id = request.matchdict['item_id']
    item = session.query(MyModel).get(item_id)

    form = Form(request,
                schema=MySchema(),
                obj=item)

    if form.validate():

        form.bind(item)

        # persist model somewhere...
        return HTTPFound(location="/")

    return dict(item=item, form=FormRenderer(form))
```

(continues on next page)

(continued from previous page)

```
@view_config(name='add',
             renderer='submit.html')
def add(self):

    form = Form(request,
                defaults={"name" : "..."},
                schema=MySchema())

    if form.validate():

        obj = form.bind(MyModel())

        # persist model somewhere...

        return HTTPFound(location="/")

    return dict(renderer=FormRenderer(form))
```

In your template (using a Mako template in this example):

```
${renderer.begin(request.resource_url(request.root, 'add'))}
${renderer.csrf_token()}
<div class="field">
    ${renderer.errorlist("name")}
    ${renderer.text("name", size=30)}
</div>
....
<div class="buttons">
    ${renderer.submit("submit", "Submit")}
</div>
${renderer.end()}
```

The steps are:

1. Create a **Form** instance with the Request and your schema. You can optionally pass in default values or an object instance.

2. Call **validate()**. This will check if the form should be validated (in most cases, if the request method is HTTP POST), and validates against the provided schema. It returns **True** if the form has been validated and there are no errors. Any errors get dumped into the **errors** property as a dict.

3. If the form is valid, use **bind()** to bind the form fields to your object. If you would rather do this manually (or you don't have an object) you can access the validated data (as a dict) from the **data** property of the form.

4. If the form hasn't been validated yet, or contains errors, pass it to your template. The form can optionally be wrapped in a **FormRenderer** which makes it easier to output individual HTML widgets.

Note the use of the *allow_extra_fields* and *filter_extra_fields* attributes. These are recommended in order to remove unneeded fields (such as the CSRF) and also to prevent extra field values being passed through.

For a complete working example, look at the "examples" directory in the source repository.

Part III

Validation

The **validate()** method does two things. First, it checks if the form should be validated. Second, it performs the validation against your schema or validators.

By default, validation will run if the request method is HTTP POST. This is set by the *method* argument to the constructor.

The validated values, or values from the request, are passed to the **data** property. Any errors are passed to the **errors** property.

Part IV

Working with models

pyramid_simpleform makes it easier to work with your models (be they SQLAlchemy or ZODB models, or something else).

First, you can pass the *obj* argument to the constructor, which can be used instead of *defaults* to set default values in your form:

```
form = Form(request, MySchema(), obj=MyModel(name="foo"))
assert form.data['name'] == 'foo'
```

Second, the **bind()** method sets object properties from your form fields:

```
if form.validate():

    obj = form.bind(MyModel())
```

Some care should be taken when using **bind()**. You can use the parameters **include** and **exclude** to filter any unwanted data:

```
if form.validate():
    obj = form.bind(MyModel(), include=['name', 'value'])
```

Note that running **bind()** on a form that hasn't been validated yet, or where the form contains errors, will raise a **RuntimeError**.

Also note that attributes starting with an underscore will not be explicitly bound. In order to bind such properties you must do so manually from the `data` property of your form instance.

Part V

Form rendering

Form rendering can be done completely manually if you wish, or using webhelpers, template macros, or other methods. The **FormRenderer** class contains some useful helper methods for outputting common form elements. It uses the WebHelpers library under the hood.

The widget methods automatically bind to the relevant field in the parent **Form** class. For example:

```
form = Form(request, MySchema(), defaults={"name" : "foo"})
renderer = FormRenderer(form)
```

If this is output using the **text()** method of your renderer:

```
${renderer.text("name", size=30)}
```

will result in this HTML snippet:

```
<input type="text" name="name" id="name" value="foo" size="30" />
```

It is expected that you will want to subclass **FormRenderer**, for example you might wish to generate custom fields with JavaScript, HTML5 fields, and so on.

Part VI

CSRF Validation

pyramid_simpleform doesn't do CSRF validation: this is left up to you. One useful pattern is to use a **NewRequest** event to handle CSRF validation automatically:

```
def csrf_validation(event):

    if event.request.method == "POST":

        token = event.request.POST.get("_csrf")
        if token is None or token != event.request.session.get_csrf_
↪token():
            raise HTTPForbidden("CSRF token is missing or invalid")
```

However the **FormRenderer** class has a couple of helper methods for rendering the CSRF hidden input. **csrf()** just prints the input tag, while **csrf_token()** wraps the input in a hidden DIV to keep your markup valid.

Part VII

State

[TBD]

Part VIII

API

class Form(*request*, *schema=None*, *validators=None*, *defaults=None*, *obj=None*, *extra=None*, *include=None*, *exclude=None*, *state=None*, *method='POST'*, *variable_decode=False*, *dict_char='.'*, *list_char='-'*, *multipart=False*, *from_python=False*)

Legacy class for validating FormEncode schemas and validators.

request : Pyramid request instance

schema : FormEncode Schema class or instance

validators : a dict of FormEncode validators i.e. { field : validator }

defaults : a dict of default values

obj : instance of an object (e.g. SQLAlchemy model)

state : state passed to FormEncode validators.

method : HTTP method

variable_decode : will decode dict/lists

dict_char : variabledecode dict char

list_char : variabledecode list char

Also note that values of `obj` supercede those of `defaults`. Only fields specified in your schema or validators will be taken from the object.

all_errors()

Returns all errors in a single list.

bind(*obj*, *include=None*, *exclude=None*)

Binds validated field values to an object instance, for example a SQLAlchemy model instance.

include : list of included fields. If field not in this list it will not be bound to this object.

exclude : list of excluded fields. If field is in this list it will not be bound to the object.

Returns the *obj* passed in.

Note that any properties starting with underscore "_" are ignored regardless of `include` and `exclude`. If you need to set these do so manually from the `data` property of the form instance.

Calling bind() before running validate() will result in a RuntimeError

default_state
alias of *State*

errors_for(*field*)
Returns any errors for a given field as a list.

htmlfill(*content*, ***htmlfill_kwargs*)
Runs FormEncode **htmlfill** on content.

is_error(*field*)
Checks if individual field has errors.

render(*template*, *extra_info=None*, *htmlfill=True*, ***htmlfill_kwargs*)
Renders the form directly to a template, using Pyramid's **render** function.

template : name of template

extra_info : dict of extra data to pass to template

htmlfill : run htmlfill on the result.

By default the form itself will be passed in as *form*.

htmlfill is automatically run on the result of render if *htmlfill* is **True**.

This is useful if you want to use htmlfill on a form, but still return a dict from a view. For example:

```
@view_config(name='submit', request_method='POST')
def submit(request):

    form = Form(request, MySchema)
    if form.validate():
        # do something
    return dict(form=form.render("my_form.html"))
```

validate(*force_validate=False*, *params=None*)
Runs validation and returns True/False whether form is valid.

This will check if the form should be validated (i.e. the request method matches) and the schema/validators validate.

Validation will only be run once; subsequent calls to validate() will have no effect, i.e. will just return the original result.

The errors and data values will be updated accordingly.

force_validate : will run validation regardless of request method.

params : dict or MultiDict of params. By default will use **request.json_body** (if JSON body), **request.POST** (if HTTP POST) or **request.params**.

class State(***kwargs*)
Default "empty" state object.

Keyword arguments are automatically bound to properties, for example:

```
obj = State(foo="bar")
obj.foo == "bar"
```

class FormRenderer(*form*, *csrf_field='_csrf'*, *id_prefix=None*)
A simple form helper. Uses WebHelpers to render individual form widgets: see the WebHelpers library for more information on individual widgets.

begin(*url=None*, ***attrs*)
Creates the opening <form> tags.

By default URL will be current path.

csrf(*name=None*)
Returns the CSRF hidden input. Creates new CSRF token if none has been assigned yet.

The name of the hidden field is **_csrf** by default.

csrf_token(*name=None*)
Convenience function. Returns CSRF hidden tag inside hidden DIV.

end()
Closes the form, i.e. outputs </form>.

hidden_tag(**names*)
Convenience for printing all hidden fields in a form inside a hidden DIV. Will also render the CSRF hidden field.

Versionadded 0.4

Index