Form Process Documentation

Release 0.4

Ian Wilson

April 01, 2011

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

Form processing can be horribly complicated. Each use case always has a weird quirk that differs from what has been done before. Rather than create another strongly coupled library I tried to create a reusable pattern that did not strongly depend on any web framework, form rendering method or form validation method. The pattern itself so far has not needed to change too much. When a developer wants to process a form it usually goes something like this: * The user is prompted with a form filled with defaults. * The user fills out and submits the form. * If there were errors then the form is redisplayed with the defaults and errors. * If there were not errors then the form input is processed and the user is either sent a message or redirected elsewhere.

That pretty much sums up all there is to it. I said earlier that it does not strongly depend on any specific tools but my examples will be using formencode and htmlfill. Also note that whichever *schema* is chosen for validation needs to provide a to_python method and raise formencode.Invalid exactly as formencode does upon finding errors. No strong coupling I swear!

CHAPTER

TWO

BASIC USAGE

This library is meant to be called directly from higher level functions. Form processing usually involves two stages. The first stage prompts the user with a form to fill in. This stage requires generating defaults and instructions on how to fill the form. The second stage processes the form that the user submits after filling it in. Not all applications that need validation will require the first stage. For example APIs don't usually send defaults to the developer before the developer makes a call to the API. We will discuss validation from a web form perspective in most examples unless otherwise noted.

2.1 Defining a form handler

The developer must at least define a schema. Here is a simple example of a handler:

```
email_form_handler = FormHandler(
        schema=BaseSchema(email=Email(not_empty=True)))
```

2.2 Prompting For Input

The developer should generate defaults and then pass those to the form handler in order to get back the form dict. This allows the handler to add more defaults, add filling arguments, encode defaults, setup the state and return a standardized dictionary to be used during form construction. Here is a simple example that populates a user's email for them to change:

```
def _render_change_email(form_dict):
    return {'email_form': form_dict}

def prompt_change_email(request):
    defaults = {
        'email': request.user.email
    }
    form_dict = email_form_handler.prompt(defaults=defaults)
    return _render_change_email(form_dict)
```

Note that if not defaults are given the defaults will just be initialized to an empty *UnicodeMultiDict*.

2.3 Processing The Form

The developer should give the unsafe form params sent by the user to the handler. Then the developer should check for success or not and act accordingly. Here is a simple example that updates the user's email.

```
def process_change_email(request):
    form_dict = email_form_handler.process(request.POST)
    if email_form_handler.was_success(form_dict):
        request.user.email = form_dict['defaults']['email']
        # redirect to something now.
    else:
        return _render_change_email(form_dict)
```

2.4 Using State

The state object provides a way for the developer to pass data to validators and hooks during form processing. Also the state object allows validators to pass additional information out to the caller of the handler after validation. For example a login validator might attach the actual user object to the state so that it can be retrieved by the caller.

All keyword arguments passed to *prompt* except for *defaults* will be set on the state object. All keyword arguments passed to *process* will be set on the state object. Calls to prompt and process where state is passed to validator might look like the following:

```
def prompt_change_username(request):
    # Pass the logged in user to the form handler.
    form_dict = form_handler.prompt(user=request.user)
    #...

def process_change_username(request):
    # Pass the logged in user to the form handler.
    form_dict = form_handler.process(request.POST, user=request.user)
    # ...
```

A call where state is retrieved from a validator might look like this:

```
def process_login(request):
    form_dict = form_handler.process(request.POST)
    if form_hander.was_success(form_dict):
        # User was set by a validator during processing.
        user = form_dict['state'].user
        # "Log in" the user.
        request.session['user_id'] = user.id
        request.session.save()
        # Redirect to something.
    else:
        # Re-render the login form.
        render_login_form(form_dict)
```

2.5 Contents of the *form_dict*

The form_dict object is just a plain python dictionary with the following keys: *defaults*, *errors*, *state* and *fill_kwargs*. The value of *errors* should evaluate to False when no errors have occurred. These have been mostly discussed before except for *fill_kwargs*.

The entry *fill_kwargs* is meant to be passed to something like formencode's *htmlfill()* function. It contains the defaults and errors but also might contain keyword arguments that tell htmlfill more information such as the encoding to use. This can be useful so you don't have to inject these all over your project.

2.6 Handler Hooks

Various hooks can be used to add more functionality to the handler. These hooks can be added by injecting them in __init__ during subclassing or by passing them in directly when creating a *FormHandler* instance. The hooks are described in the *Hooks* section.

CHAPTER

THREE

HOOKS

Hooks are functions that are used to add more functionality to the handler. There can be zero more of each type of hook except for *get_schema*.

3.1 update_defaults

This hook adds defaults to be used during the prompt stage of form handling. A hook function might look like this:

```
def set_user_email(handler_instance, defaults, state):
    defaults['email'] = state.user.email
    return defaults

form_handler = FormHandler(schema=schema,
        update_defaults_hooks=[set_user_email])
```

3.2 get_schema

This hooks lets the developer dynamically determine which schema to use for form validation. It should be obvious but just a reminder these params have received no validation and must be considered unsafe. Also there should only be zero or one of this type of hook. If this hook is not set the schema attribute of the handler is used for validation. An example of choosing a schema for a product's information based on the product's type might look like this:

```
def get_schema(handler_instance, unsafe_params, state):
    if unsafe_params.get('product_type', None) == 'advanced':
        return advanced_product_schema
    else:
        return simple_product_schema

form_handler = FormHandler(get_schema_hook=get_schema)
```

3.3 update_state

This hooks is called during both prompt and process and can be used to add attributes to state based on access to global variables. Here is a completely fabricated example that sets today's date on the state object.

```
def set_today(handler_instance, defaults, state):
    state.today = date.today()

form_handler = FormHandler(update_state_hooks=[set_today])
```

3.4 on_process_error

This hook is called when validation fails. It is useful to add or remove things before rendering the form again. This example removes the password:

```
def remove_password(handler_instance, defaults, errors, state):
    if defaults.has_key('password'):
        del defaults['password']
    return defaults

form_handler = FormHandler(on_process_error_hooks=[remove_password])
```

3.5 defaults_filter

This hook pre-processes the defaults passed to process. This is useful to remove/add/alter the defaults before validation. Here is an example that removed keys ending in '–system'.

```
def filter_system_params(handler_instance, unsafe_params, state):
    for k in unsafe_params.keys():
        if k.endswith('--system'):
            del unsafe_params[k]
    return unsafe_params

form_handler = FormHandler(defaults_filter_hooks=[filter_system_params])
```

3.6 customize_fill_kwargs

This hook is useful to inject kwargs used for filling the form. Here is an example that adds an error formatter and tells htmlfill to use it by default.

```
from formencode.htmlfill import html_quote, default_formatter_dict

error_class = 'std_error'

error_formatters = default_formatter_dict.copy()

def std_formatter(error):
    return '<span class="%s">%s</span>' % \
            (error_class, html_quote(error))

error_formatters['std'] = std_formatter

def add_std_error_formatter(handler_instance, defaults, errors, state,
        fill_kwargs):
    fill_kwargs.update({
        'error_class': error_class,
```

```
        'error_formatters': error_formatters,
        'auto_error_formatter': error_formatters['std'],
    })
    return fill_kwargs

form_handler = FormHandler(
        customize_fill_kwargs_hooks=[add_std_error_formatter])
```

CHAPTER

FOUR

FORMPROCESS API DOCUMENTATION

4.1 Modules

4.1.1 formprocess.handler

4.1.2 formprocess.utils

4.1.3 formprocess.pylonshandler

4.1.4 formprocess.pyramidhandler

CHAPTER

FIVE

INDICES AND TABLES

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