iktomi Documentation

Release 0.4.3

Denis Otkidach

Sep 27, 2017

Contents

1 Routing **3**
- 1.1 Basic Practices . . . 3
- 1.2 Advanced Practices . . . 8
- 1.3 Web Routing Reference . . . 10

2 Forms **11**
- 2.1 Overview: Form Abstraction Layers . . . 11
- 2.2 Form Class . . . 15
- 2.3 Fields . . . 16
- 2.4 Converters . . . 21
- 2.5 Widgets . . . 24
- 2.6 Forms Reference . . . 26

3 Cli **27**
- 3.1 Cli Reference . . . 27

4 Templates **31**
- 4.1 Template . . . 31
- 4.2 Templates Reference . . . 32

5 Utilities **33**
- 5.1 Various utilities . . . 33

Python Module Index **37**

Iktomi is a python package providing some basic tools for creating web applications. Iktomi is built on top of webob and supports Python 2.7 and Python 3.5.

It contains few independent subpackages, you can use both, or just one of them:

- **iktomi.web**, a flexible and extensible routing tool, suitable to dispatch HTTP requests between various views (or controllers).
- **iktomi.forms**, a web forms validation and rendering tool.
- **iktomi.cli**, a layer for building command-line utilities.
- **iktomi.templates**, an adaptation layer for template engines, in particular, for jinja2.
- **iktomi.db**, database utilities, in particular, sqlalchemy types, collections, declarative mixins.

Some things are dedicated to package **iktomi.unstable**. This means the interfaces are unstable or unclear in some point, and we do not want to gurarantee their permanence for a long time.

CHAPTER 1

Routing

Basic Practices

Hello, World

Iktomi produces WSGI application from a couple of own web handlers. It wraps environment into webob.Request and accepts the result as webob.Response object. In most cases, web handler is represented by function.

Here is the common interface of web handlers:

```
def handler(env, data):
    ...
    return response
```

env is an iktomi application's current environment. Basically it contains only one significant attribute: *webob.Request* object in *env.request*. *data* will be described below.

A handler returns *webob.Response* or *None* (this case will be described below), also it can raise *webob.exc.HTTPException* subclasses or call other (in most cases, next) handlers and return their result.

So, here is an example for very basic web handler:

```
import webob

def hello_world(env, data):
    name = env.request.GET.get('name', 'world')
    return webob.Response('Hello, %s!' %name)
```

Any handler can be converted to WSGI app:

```
from iktomi import web
from iktomi.web.app import Application

wsgi_app = Application(hello_world)
```

Here it is! You can use the given object as common WSGI application, make server, for example, using *Flup*. Implementation of development server can be found at Development server. Now we can create *manage.py* file with the following content:

```
#!/usr/bin/python2.7
import sys
from iktomi.cli import manage
from iktomi.cli.app import App

def run():
    manage(dict(
        # dev-server
        app = App(wsgi_app),
    ), sys.argv)

if __name__ == '__main__':
    run()
```

And now we can run the server:

```
./manage.py app:serve
```

Basic Routing

There are a couple of handlers to match different url parts or other request properties: *web.match*, *web.prefix*, *web.methods*, *web.subdomain*, etc.

Iktomi routing is based on *web.cases* and *web.match* class. Constructor of *web.cases* class accepts a couple of other handlers. When called the *web.cases* instance calls each of handlers until one of them returns *webob.Response* object or raises *webob.exc.HTTPException*.

If any handler returns *None*, it is interpreted as "request does not match, the handler has nothing to do with it and *web.cases* should try to call the next handler".

Constructor of *web.match* class accepts URL path to match and a handler name to be used to build an URL (see below). If the request has been matched, *web.match* calls next handler, otherwise returns *None*. Let's see an example:

```
web.cases(
    web.match('/', 'index') | index,
    web.match('/contacts', 'contacts') | contacts,
    web.match('/about', 'about') | about,
)
```

As we see, | operator chains handlers and makes second handler next for the first.

Note: handlers are stateful, they store their next and nested handlers in the attributes. Therefore, they can be reused (i.e. can be included in application in two places), because '|' operator copies instances of handlers.

And here's how it works. For request:

```
GET /contacts
```

1. *web.cases* is called, it calls the first *web.match* handler
2. *web.match('/', 'index')* does not accept the request and returns None.
3. *web.cases* gets *None* from first handler and calls the next.
4. *web.match('/contacts', 'contacts')* accepts the request, calls next handler (contacts) and returns it's result.

5. *web.cases* gets not-None result from handler, stops iteration over handlers and returns the result.

Note that execution of chain can be cancelled by every handler. For example, if *contacts* handler returns *None*, *web.cases* does not stop iteration of handlers and *web.match('/about', 'about')* is called.

URL parameters

If URL contains values that should be used in handlers (object ids, slugs, etc), *werkzeug*-style URL parameters are used:

```
web.match('/user/<int:user_id>')
```

Where *int* is name of an url converter, and *user_id* is attribute name. All url-matching handlers use common url parsing engine. They get parameters' values from url and put them to *data* object by *__setattr__*.

Iktomi provides some basic url converters: *string* (default), *int*, *bool*, *any*. It also allows you to create and use own ones (see below).

Nested handlers and URL Namespaces

There is very handy way to logically organize your url map: namespaces:

```
web.cases(
    web.prefix('/api', name="api") | web.cases(...),
    # this is equal to:
    # web.prefix('/api') | web.namespace('api') | web.cases(...),
    web.prefix('/user/<int:user_id>', name='user')  | web.cases(...),
)
```

For more complex projects a simple combinations of *web.cases* and *web.match* does not satisfy. Iktomi provides some handlers to create complex routing rules and allows to create your own handlers. And you can combine handlers as you want. Here is an example:

```
web.cases(
    web.prefix('/api', name="api") | web.methods(['GET']) | web.cases(
        web.match('/users', 'users') | users_list,
        web.match('/comments', 'comments') | comments_list
    ) | to_json,

    web.match('/', 'index') | index,
    web.prefix('/user/<int:user_id>', name="user") | web.cases(
        web.match('', 'profile') | user_profile,
        web.match('/comments', 'comments') | user_comments,
    )
)
```

URL namespacing is useful to include similar app parts to many places in your app, or for plug-in any reusable app from outside without warry about name clashes.:

```
def handler(env, data):
    curr_namespace = env.namespace if hasattr(env, 'namespace') else None
    en_url = env.root.build_url('en.index')
    curr_url = env.root.build_url('.index')
    return webob.Response('%s %s %s' % (curr_namespace,
                                        en_url, curr_url))
```

```
part = web.match('/index', 'index') | handler

web.cases(
    # first renders "en /en/index /en/index"
    web.prefix('/en', name='en') | part,
    # second renders "ru /en/index /ru/index"
    web.prefix('/ru', name='ru') | part,
)
```

Building URLs

Iktomi provides url building (or reversing) engine.

URL reverse object is a callable that can be created for any handler:

```
root = web.Reverse.from_handler(app)
```

or the same object can be found in *env.root* attribute during the request handling.

There are two ways of using *Reverse* object. Attribute-based one:

```
root.user(user_id=5).as_url
root.user(user_id=5).comments.as_url
```

or string-based method:

```
root.build_url('user', user_id=5)
root.build_url('user.comments', user_id=5)
```

Note: string-based API is just a shortcut layer on top of attribute-based one Note: attribute-based API returns a subreverse object (also 'Reverse' instance), while string-based API returns 'web.URL' instances. If you want to get subreverse, use 'root.build_subreverse('user', user_id=5)'

Controlling execution flow

Iktomi allows to natively implement many use cases without any extra essences like Django-middlewares, etc.

For example, to implement "middleware" you can do something like:

```
@web.request_filter
def wrapper(env, data, next_handler):
    do_something()
    result = next_handler(env, data)
    do_something_else(result)
    return result

wrapped_app = wrapper | web.cases(..)
```

Note: 'web.request_filter' is decorator transforming function to regular WebHandler, this allows to chain other handlers after given. The chained handler is passed as third argument into the handler.

It is transparent, obvious and native way. Also, it is possible to use try...except statements with next_handler:

```
@web.request_filter
def wrapper(env, data, next_handler):
    try:
```

```
        return next_handler(env, data)
    except MyError:
        return exc.HTTPNotFound()
```

or even something like that:

```
@web.request_filter
def wrapper(env, data, next_handler):
    with open_db_connection() as db:
        env.db = db
        return next_handler(env, data)
```

Scopes of environment and data variables

env and *data* objects does not just store a data, also they delimitate data between handlers from differrent app parts. *web.cases* handler is responsible for this delimitation. For each nested handler call it "stores" the state of *env* and *data* objects and restores it after handler execution.

Each nested handler can change *env* and *data* objects and these changes will not affect other routing branches. So you don't worry about the data you've added to *data* and *env* will involve any unexpected problems in other part of your app. Therefore, be careful with this feature, it can lead to design mistakes.

Smart URL object

URL build functions does not return actually *str* object, but it's *web.URL* subclass'es instance. It allows to make common operations with queryString parameters (add, set, delete) and also has method returning URL as human-readable unicode string:

```
>>> print(URL('/').set(q=1))
/?q=1
>>> print(URL('/').set(q=1).add(q=2))
/?q=1&q=2
>>> print(URL('/').set(q=1).set(q=3))
/?q=3
>>> print(URL('/').set(q=1).delete('q'))
/
>>> print(URL('/', host=u".").set(q=u''))
http://xn--80abnh6an9b.xn--p1ai/?q=%D0%BE%D0%BA
>>> print(URL('/', host=u".").set(q=u'').get_readable())
http://./?q=
```

Throwing HTTPException

Iktomi allows *webob.HTTPException* raising from inside a handler:

```
from webob import exc

@web.request_filter
def handler(env, data, next_handler):
    if not is_allowed(env):
        raise exc.HTTPForbidden()
    return next_handler(env, data)
```

Also you can use *HTTPException* instances in route map:

```
web.cases(
    web.match('/', 'index') | index,
    web.match('/contacts', 'contacts') | contacts,
    web.match('/about', 'about') | about,
    exc.HTTPNotFound(),
)
```

Advanced Practices

Advanced routing tools

Iktomi provides some additional filters.

A **subdomain** filter allows to select requests with a given domain or subdomain:

```
web.cases(
    web.subdomain('example.com') | web.cases(
        web.match('/', 'index1') | index1,
    ),
    web.subdomain('example.org') | web.cases(
        web.match('/', 'index2') | index2,
    ),
)
```

You can use multiple subdomain filters in a line to select lower-level subdomains. To specify a base domain chain one subdomain filter before:

```
web.subdomain('example.com') | web.cases(
    # all *.example.com requests get here
    web.subdomain('my') | web.cases(
        # all *.my.example.com requests get here
        ...
    ),
    ...
)
```

A **static_files** handles static files requests and also provides a reverse function to build urls for static files:

```
static = web.static_files(cfg.STATIC_PATH, cfg.STATIC_URL)

@web.request_filter
def environment(env, data, next_handler):
    ...
    env.url_for_static = static.construct_reverse()
    ...

app = web.request_filter(environment) | web.cases(
    static,
    ...
)
```

Handling files is provided for development and testing reasons. You can use it to serve static file on development server, but it is strictly not recommended to use it for this purpose on production (use your web server configuration requests instead of it). Surely, reverse function is recommended to use on both production and development servers.

Custom URL converters

You can add custom URL converters by subclassing *web.url.Converter*. A subclass should provide *to_python* and *to_url* methods. First accepts **unicode** url part and returns any python object. Second does reverse transformation. Note, that url parts are escaped automatically outside URL converter:

```
class MonthConv(url.Converter):
    def to_python(self, value, **kwargs):
        try:
            return int(value)
        except ValueError:
            raise ConvertError(self.name, value)

    def to_url(self, value):
        return str(value)
```

To include URL converter, pass *convs* argument to handler constructor:

```
prefix('/<month:month_num>', convs={'month': MonthConv})
```

Make an application configurable

Configuring *env* object:

```
class FrontEnvironment(web.AppEnvironment):
    cfg = cfg
    cache = memcache_client

    def __init__(self, *args, **kwargs):
        super(FrontEnvironment, self).__init__(*args, **kwargs)
        self.template_data = {}

    @cached_property
    def url_for(self):
        return self.root.build_url

    @storage_cached_property
    def template(storage):
        return BoundTemplate(storage, template_loader)

    @storage_method
    def render_to_string(storage, template_name, _data, *args, **kwargs):
        _data = dict(storage.template_data, **_data)
        result = storage.template.render(template_name, _data, *args, **kwargs)
        return Markup(result)

    @storage_method
    def render_to_response(self, template_name, _data,
                           content_type="text/html"):
        _data = dict(self.template_data, **_data)
        return self.template.render_to_response(template_name, _data,
                                                content_type=content_type)

    @storage_method
    def redirect_to(storage, name, qs, **kwargs):
        url = storage.url_for(name, **kwargs)
        if qs:
```

```
            url = url.qs_set(qs)
        return HTTPSeeOther(location=str(url))

    def json(self, data):
        return webob.Response(json.dumps(data),
                              content_type="application/json")

    @cached_property
    def db(self):
        return db_maker()

wsgi_app = Application(app, env_class=FrontEnvironment)
```

Describe differences between *storage_method*, *storage_property*, *storage_cached_property*, *cached_property* here.

- *BoundTemplate* subclassing
- *environment* handler

Web Routing Reference

Basic handlers

Builtin filters

Url converters

Reversing urls

URL object

WSGI application

CHAPTER 2

Forms

Overview: Form Abstraction Layers

Form is abstraction designed to validate user form data and convert it to inner representation form.

Iktomi forms can accept data in *webob.MultiDict*-like form (basically *request.POST* and *request.GET* objects), and return it in form of any python objects hierarchy, depending only on implementation. Basically it is structured combination of python dicts and lists containing atomic values.

The most basic usage of forms is the following:

```
from iktomi.forms.form import Form
from iktomi.forms.fields import Field, FieldList

class UncleForm(Form):
    fields = [
        Field('name'),
        FieldList('nephews', field=FieldSet(None, fields=[
            Field('name'),
        ])),
    ]

def process_form(request):
    form = UncleForm()
    if form.accept(request.POST):
        render_somehow(form.python_data)
        # {"name": "Scrooge",
        #  "nephews": [{"name": "Huey"},
        #              {"name": "Dewey"},
        #              {"name": "Louie"}]}
    else:
        render_somehow(form, form.errors)
```

Iktomi form implementation contains of few abstraction layers:

Forms

iktomi.forms.form.Form subclasses contain a scheme of validated data as list of fields. Instances of these classes provide an interface to work with entire form, such as: validate the data (*Form.accept*), render the entire form (*Form.render*). Also they store common form data: initial, raw and resulting values, environment, errors occured during validation.

See more.

Fields

iktomi.forms.fields.BaseField instances represent one node in data scheme. It can be atomic data (string, integer, boolean) or data aggregated from collection of other fields (*FieldList* or *FieldSet*, see below). Atomic values correspond to *Field* class.

Each field has a name. Name is a key in resulting value dictionary.

Also there are a few auxillary attributes like *label*, *hint*.

Finally, the main options of *BaseField* instances are converter and widget objects.

See more.

Converters

iktomi.forms.convs.Converter instances are responsible for all staff related to data validation and convertation. Converter subclasses should define methods for transformations in two directions:

- *to_python* method accepts unicode value of user info, and returns value converted to python object of defined type. If the value can not be converted, it raises *iktomi.forms.convs.ValidationError*.
- *from_python* method accepts python object and returns corresponding unicode string.

Examples of converters are *Int*, *Char*, *Html*, *Bool*, *Date*, etc.

Converters support few interesting additional features.

The most used feature is **require** check. If the converter has *require* attribute set to *True*, it checks whether *to_python* result is an empty value:

```
Field('name',
      conv=convs.Char(required=True))
```

Multiple values are implemented by *ListOf* converter:

```
class MyForm(Form):

    fields = [
        Field('ids',
              conv=ListOf(Int()))
    ]

# ids=1&ids=2 =>
# {"ids" [1, 2]}
```

Additional validation and simple one-way convertation can be made by **validators**:

```
Field('name',
      Char(strip, length(0, 100), required=True))
```

See more.

Widgets

iktomi.forms.widget.Widget instances are responsible for visual representation of an item.

The main method of widget is *render*, which is called to get HTML code of field with actual value.

Widget can do some data preparations and finally it is rendered to template named *widget.template* (by default, *jinja2* is used).

Examples of widgets are *TextInput*, *Textarea*, *Select*, *CheckBox*, *HiddenInput*, etc.

See more.

Aggregate Fields

Iktomi forms are very useful to validate and convert structured data with nested values.

There are three basic subclasses of *BaseField*. Combining fields of those classes, you can describe a scheme for nested JSON-like data (containing lists and dictionaries). And you can easily describe any tree-like python objects structure using custom *Converter* subclasses.

These classes are:

- *FieldSet* represents a collection of various fields with different names, converters and widgets. Purpose of *FieldSet* is to combine values into a dictionary or object (you can get an object of whatever type you want by defining your own converter for *FieldSet* with transformation rules to/from dictionary):

```
class MyForm(Form):
    fields = [
        FieldSet('name',
                 fields=[
                    Field('first_name'),
                    Field('last_name'),
                 ])
    ]

# {"name": {'first_name': 'Jar Jar', 'last_name': "Binks"}}
```

- *FieldBlock* is like *FieldSet*, but it does not form separate object. Instead, it adds it's own values to parent field's value, as if they are not wrapped in separate field. *FieldBlock* is used for visually group fields or for purposes of combined validation of those fields:

```
class MyForm(Form):
    fields = [
        FieldBlock(None,
                 fields=[
                    Field('first_name'),
                    Field('last_name'),
                 ])
    ]

# {'first_name': 'Jar Jar', 'last_name': "Binks"}
```

- *FieldList* represent a list (basically infinite) of identical fields:

```
class MyForm(Form):
    fields = [
        FieldList(
            'characters',
            field=FieldSet(None,
                 fields=[
                    Field('first_name'),
                    Field('last_name'),
                 ]))
    ]

# {'characters': [{'first_name': 'Jar Jar', 'last_name': 'Binks'},
#                 {'first_name': 'Jabba', 'last_name': 'Hutt'}]}
```

See more.

File Handling

Readonly Fields, Permissions

Iktomi forms have a customizable permission layer. Two permissions supported by default are read (*r*) and write (*w*).

Each field can have it's own permissions, but the common rule is that child field permissions are subset of the parent field's (or form's) ones:

```
class MyForm(Form):

    fields = [
        Field('name', permissions="rw")
    ]

form = MyForm(permissions="r")
```

Permissions can be calculated dinamically based on environment (request, logged in user roles, etc.).

See more.

Copy Interface

Some classes (fields, widgets, converters) implement copy by *__call__*. This is very useful when making widely customizable interfaces.

You do not need to create a subclass every time you want reuse your widgets or converters. From other side, there is no need to instantiate a class every time with all the options.

Instead, you can just create an object once and then copy it redefining only options you want:

```
char = Char(length(0,100), NoUpper, required=False)

field1 = Field(conv=conv)
field2 = Field(conv=conv(required=True))
```

or even:

```
field1 = Field(conv=Char(length(0, 100)))
field2 = field1(conv=field1.conv(required=True))
```

Form Class

iktomi.forms.form.Form is the most top-level class in iktomi forms objects hierarchy. Instances of this class encapsulate all the data needed to validate a form and a result of the validation: field hierarchy with converters and forms-widgets, initial data, raw data which is converted and validated, resulting value, errors occured during validation, environment including all the data and context related to current request.

Form instances are usually the only objects user interacts with on runtime (during a request).

Form class is designed to serve on several purposes.

Form Validation

Form validation is done by *form.accept* method. This is main interface method of the form.

It accepts a *webob.MultiDict*-like object as argument and returns boolean value whether that value passes validation or not. In particular, it can be *GET* and *POST* properties of *webob.Request* object:

```
form = MyForm()
if form.accept(request.POST):
    do_something(form.python_data)
```

Forms are stateful, and *accept* method sets form state regarding given value. Here are some variables representing form set:

- *form.is_valid*, a boolean value: wheter form validation was successful or not.
- *form.python_data*, a dictionary, result of form, actual converted value. Can be inconsistent if form is not valid.
- *form.raw_data* is a copy of input MultiDict possibly mutated in converting process. It contains all the values from all form's fields, but can also contain an unrelated values if they existed in the source MultiDict. To get canonical and clear raw value of the actual state of the form, use *Form.get_data* method.
- *form.errors* is a dictionary containing errors occured during validation. Key of the dict is field.input_name, and value is error message related to that field.

Rendering to HTML

Form provide an interface to be rendered to HTML. This is *render* method. It takes no parameters and renders a template with name equal to *form.template* passing form as a variable.

For example, if you have *form* variable in jinja2 template, you can call:

```
<table>
    {{ form.render() }}
</table>
```

In that template forms fields are iterated and each field is rendered by *field.widget.render()*.

If you have non-trivial HTML layout, it is OK to ignore *form.render* interface and call directly *field.widget.render* method:

```
{{ form.get_field('field_input_name').widget.render() }}
```

And finally, for sure, you can redefine a template name in your Form subclass:

```
class MyForm(Form):

    template = "custom-form.html"
    fields = [...]
```

For details of rendering engine, see Widgets section.

Filling Initial Data

Form may have initial value. This is useful, for example, for object editing forms:

```
initial = as_dict(obj)
form = ObjForm(initial=obj)
```

Initial value is set to forms'python data, and then re-filled with each field's loaded initial value. At the same time form's raw value is updated to be in accordance with initial value.

To learn how each field loads an initial value, see *Fields: Setting initial value* section.

Providing Access to the Environment

Form instances have one more purpose. They store *env* object, a request-level iktomi environment, and provide an access to this environment for all other objects in form hierarchy: fields, convs, widgets:

```
form = MyForm(env)

form.env # same as
form.get_field(field_name).env # same as
form.get_field(field_name).widget.env # same as
form.get_field(field_name).conv.env
```

The environment can be used to acces a database, template engine, webob.Request, configuration, etc.

Fields

Field in iktomi are basic concept representing a single item in data model.

It can be atomic data (string, integer, boolean) or data aggregated from collection of other fields (*FieldList* or *FieldSet*, see below). Atomic values correspond to *Field* class.

Fields implement *copy* interface.

Field Naming

Each field has a name. Name is used to get a value from raw data and as key in resulting value dictionary. Name used to lookup in raw data is *Field.input_name*, it is calculated as input_name of parent field joined by a dot with the name of the field.

Data that has no related field, is not present in python value.

Here is example of form without convertation and validation just to show how field naming works:

```
class MyForm(Form):

    fields = [
        Field('name'),
        FieldSet('info', fields=[
            Field('address'),
            Field('birth_date'),
        ]),
    ]

raw_value = MultiDict([
    ('name', 'John'),
    ('info.address', 'Moscow'),
    ('info.birth_date', '19.05.1986'),
    ('info.more', 'This value is ignored'),
])

form = MyForm()
form.accept(raw_value)
print form.python_data
# {"name": "John",
#  "info": {"address": "Moscow",
#           "birth_date": "19.05.1986"}}
```

Also field name can be used to retrieve a field object from form or from parent fields. If there are nested fields, those values are joined by dot:

```
name_field = form.get_field('name')
address_field = form.get_field('info.address')
birth_field = form.get_field('info').get_field('birth_date')
```

Name of field should be unique throught it's neighbours.

Converters and Widgets

Two main properties of the field are *BaseField.conv*, defining convertation and validation rules, and *BaseField.widget*, defining how widget is rendered to HTML.

See more in Converters and Widgets sections.

Scalar and Multiple Fields

Iktomi forms have a way to MultiDict feature of having multiple values on the same key. It is implementes by *ListOf* converter.

Fields having *ListOf* converter are marked as multiple. This means they always return a list, each value of this list is converted by *ListOf.conv* converter.

Empty and defult values of multiple fields is empty list, while for scalar fields it is *None*.

See *ListOf* for details.

Setting Initial Value

Initial value of the field is calculated as follows:

- If the key equal to field's name is present in parent's initial value, it is used.
- If *BaseField.get_initial* is redefined, it is called and the result is used.
- If *BaseField.initial* is defined, it is used.
- Otherwise field initial value is set to empty value: *None* for scalar field and empty list for multiple field.

Aggregate Fields

The most significant feature of iktomi forms is ability to work with structured data with nested values.

Using iktomi you can easily work with deep JSON-like structures (containing lists and dictionaries), generate ORM objects, ORM object collections, and even ORM object collection inside other ORM object.

There are three common aggregate classes implemented in iktomi.

FieldSet

FieldSet is representation of dictionary (or object with named attribute).

FieldSet contains a collection of various fields with different names, converters and widgets. *FieldSet* is to combines values converted with child fields into a dictionary or object:

```
class MyForm(Form):
    fields = [
        FieldSet('name',
                 fields=[
                    Field('first_name'),
                    Field('last_name'),
                 ])
    ]

raw_value = MultiDict([
    ('name.first_name', 'Jar Jar'),
    ('name.last_name', 'Binks'),
])

form = MyForm()
form.accept(raw_value)
print form.python_data
# {"name": {'first_name': 'Jar Jar', 'last_name': "Binks"}}
```

FieldSet adds it's input name as prefix for child fields, joined with a dot.

There is a way to get object of custom type as a result of *FieldSet*. See *Custom FieldSet Value Type*.

And, of course, you can add extra validation rules for FieldSet, including combined common validation of child values. See *Collective validation*.

FieldBlock

FieldBlock is like *FieldSet*, but it does not form separate object. Instead, it adds it's own key-value pairs to parent field's value, as if they are not wrapped in separate field.

FieldBlock is used for visually group fields or for purposes of combined validation of those fields:

```
class MyForm(Form):
    fields = [
        FieldBlock(None,
                  fields=[
                     Field('first_name'),
                     Field('last_name'),
                  ])
    ]

raw_value = MultiDict([
    ('first_name', 'Jar Jar'),
    ('last_name', 'Binks'),
])

form = MyForm()
form.accept(raw_value)
print form.python_data
# {'first_name': 'Jar Jar', 'last_name': "Binks"}
```

Combined validation of nested fields is also easy to implement:

```
def validate(field_block, value):
    if not (value['first_name'] or value['last_name']):
        raise convs.ValidationError('specify first or last name')
    return value

FieldBlock(None,
           fields=[
               Field('first_name'),
               Field('last_name'),
           ],
           conv=FieldBlock.conv(validate))
```

FieldBlock does not affect on input names of child fields. It is named as if they are children of FieldBlock's parent.

FieldList

FieldList represent a list (basically infinite) of identical fields.

FieldList creates instances of child field for each value list item. Their input name is equal to FieldList's input name joined by a dot with value index in a list.

FieldList stores indexes of it's values in raw data, to use them to find data of nested fields. The order of values in python_data depends on order of indices of values in raw data.

Here is an example:

```
class MyForm(Form):
    fields = [
        FieldList(
            'characters',
            field=FieldSet(None,
                  fields=[
                     Field('first_name'),
                     Field('last_name'),
                  ]))
```

```
    ]

raw_value = MultiDict([
    ('characters.1.first_name', 'Jar Jar'),
    ('characters.2.last_name', 'Binks'),
])

form = MyForm()
form.accept(raw_value)
print form.python_data
# {'characters': [{'first_name': 'Jar Jar', 'last_name': 'Binks'},
#                 {'first_name': 'Jabba', 'last_name': 'Hutt'}]}
```

Access to Converted and Raw Values

Access to current field value is provided by two properties: *raw_value* - actual field raw (unconverted, result of *from_python*) and *clean_value* - actual field converted value.

Raw data is stored in *Form* instance and actual clean value is stored directly in the field.

Field instances are responsible for raw and clean value consistency with current form state.

They fill *raw_data* with initial value reflection on form initialization and they fill *raw_data* with actual validated value reflection during validation process. Raw data is managed by *set_raw_value* method.

And *clean_value* is managed by *accept* method, the result of converter call is set to *self.clean_value*.

These methods are already implemented for all fields provided by default and done automatically. But if you want to implement your own field class with specific data flow, you should carefully handle data consistency.

Field permissions

Iktomi provides a simple but flexible permission system. Permissions can be set in UNIX-like way by string where every single letter defines a permission:

```
Field('name', permissions="rw")
```

Permissions propagate from parent fields (or form) to their children: child field permissions are subset of it's parent permissions.

Two permissions supported by default are read (*r*) and write (*w*).

Read permission allows field to be rendered.

Write permission allows assign a field value to convertation result. If the field has no *'w'* permission, it can not be changed by *form.accept* method.

Permission can be set explicitly by passing *permissions* argument to *Field* or by defining a custom permission getter object. For example, if you want a field to be accessible only for several users, you can define your own subclass of *FieldPerm* and pass it to the field:

```
Field('name', perm_getter=UserBasedFieldAuth())
```

See more about permission customization.

Converters

iktomi.forms.convs.Converter instances are responsible for all staff related to data validation and convertation.

Converters implement *copy* interface.

Value convertation

Converter subclasses should define methods for transformations in two directions:

- *to_python* method accepts unicode value of user info, and returns value converted to python object of defined type. If the value can not be converted, it raises *iktomi.forms.convs.ValidationError*.
- Filters and validators provide extra validation and are called after *to_python* method. *See below* for details.
- *from_python* method accepts python object and returns corresponding unicode string.

ValidationError

The common way for converter to indicate that given value can not be accepted is to raise *ValidationError* from *to_python* method:

```
def to_python(self, value):
    if not self.match(value):
        raise ValidationError(error_message)
    return value
```

If *ValidationError* was raised, the error is added to *form.errors* dictionary. The key is current field's input name and the value is

In the case *ValidationError* occured, converter returns field's initial/last valid value. In other words, the value is reverted to it's last valid state, basically it is initial state.

Raise error for other field

Sometimes you need to show error not on a field which is validating, or show error on multiple fields on the same condition. In this case *ValidationError.by_field* argument can be used.

Just pass in *by_field* kwarg a dict where a key is input_name of any field in the form and a value is error message:

```
raise convs.ValidationError(by_field={
        'name': 'Provide a name or a nickname',
        'nickname': 'Provide a name or a nickname'})
```

Relative field input names can be used. If a name starts with a dot, *conv.field.get_field* will be used to get target field. If it starts with two dots, *conv.field.parent.get_field* will be used, three dots - *conv.field.parent.parent.get_field*, etc.

Why not to set *form.errors[field.input_name]* directly? Trust me, it is not good idea! One reason is *conv.accept* silent mode, used to fill in initial values of the field and sometimes can be used to call converter as a function without form attached and error handling. **Other reasons?**

Error messages redefinition

Default converters support redefinition of default error messages.

Set *error_<type>* parameter to your own message template, for example:

```
convs.Char(required=True,
           regex="\d{2}\.\d{2}\.\d{2}",
           error_regex='Should match YY.MM.DD',
           error_required='Do not forget this field!')
```

Require Check

The most used feature is **require** check. If the converter has *require* attribute set to *True*, it checks whether *to_python* result is an empty value:

```
Field('name',
      conv=convs.Char(required=True))
```

Empty values are empty string, emty list, empty dict and *None*. If the result value is equal to one of these values, *ValidationError* with *conv.error_required* is raised.

ListOf and Multiple Values

Multiple values with same key in raw data are supported by *ListOf* converter.

ListOf gets all values by field's key from raw data *MultiDict*, applies *ListOf.conv* to each of them, and collects non-empty results into a list. If ValidationError was raised in *ListOf.conv*, the value is also ignored:

```
class MyForm(Form):

    fields = [
        Field('ids',
              conv=ListOf(Int()))
    ]

# ids=1&ids=2&ids=x =>
# {"ids" [1, 2]}
```

The *multiple* property of fields and widgets having ListOf converter, is equal to *True*.

Filters and validators:

Additional validation and simple one-way convertation can be made by validators:

```
Field('name',
      Char(strip, length(0, 100), required=True))
```

Filters are functions performing additional validation and convertation after `to_python()` method. The interface of filters is following:

```
def filter_value(conv, value):
    if wrong(value):
        raise ValidationError(..)
```

```
    new_value = do_smth(value)
    return new_value

convs.Char(filter_value, required=True)
```

Validators are shortcuts to filters that do no convertations, but only do assertions:

```
@validator(error_message)
def validate(conv, value):
    return is_valid(value)
```

Both filters and validators can be passed to converter as positional arguments and will be applied after `to_python()` method and *required* check in order they are mentioned.

Internationalization

Iktomi implements very basic internationalization support. There are *N_* and *M_* markers for single and plural translatable strings respectively.

There is no complex mechanics with threadlocals or other things allowing to transparently "in place" and lazy translate these strings. Iktomi by default supports only translation of *ValidationError* messages before they are added in *form.errors*.

For single messages *env.gettext* is called, and for plural ones *env.ngettext* is called. You must provide these methods in your *Application* subclass.

Here is an example of how plural merker works. Dictionary formatting with % is used and a key *M_.count_field* is used as count indicator to *ngettext*:

```
message = M_(u'must be less than %(max)d symbol',
             u'must be less than %(max)d symbols',
             count_field="max")
def validate(conv, value):
    max_length = get_max_length(conv)
    if len(value) > max_length:
        message = message % dict(max=max_length)
        raise convs.ValidationError(message)
    return value
```

Converters for Aggregate Fields

Collective validation

FieldSet and *FieldBlock* converters are good place to implement a complex validation rules, including data from more than one field.

You can implement them in to_python method of converter or in a validator. To access a value of child field just get it from actual dict by a field name:

```
#def validate(conv, value):
def to_python(self, value):
    if value['field1'] == value['field2']:
        raise ValidationError('values must not be equal')
    return value
```

ValidationError.by_field feature also can be useful here.

Custom FieldSet Value Type

To get a custom object as a clean value of FieldSet, you can define own *Converter* subclass implementing transformations from an object to dictionary (in *from_python* method) and from dictionary to an object (in *to_python*).

The most basic example of converter of this kind:

```
class ObjConv(Converter):

    def from_python(self, value):
        result = {}
        # in case there are nested FieldBlock fields, always use field.field_names
        # to get a list of fields directly contained in the value
        field_names = sum([x.field_names for x in self.field.fields], [])
        for field_name in field_names:
            result[field_name] = getattr(value, field_name)
        return result

    def to_python(self, value):
        return self.model(** value)
```

You can see *iktomi.unstable.forms.convs.ModelDictConv* as an example of custom FieldSet converter.

Converter implementations

Examples of converters are *Int*, *Char*, *Html*, *Bool*, *Date*, etc.

Widgets

iktomi.forms.widget.Widget instances are responsible for visual representation of an item.

Widgets implement *copy* interface.

Rendering

General way to render a form to HTML is to call *Form.render* method. This method iterates over all top-level fields and calls their widgets' *render* methods. This method is called to get HTML code of field with actual value.

Widget can do some data preparations and finally it is rendered to template named *widget.template* (by default, *jinja2* is used). You can redefine template name by passing it to the widget:

```
class MyWidget(TextInput):

    template = 'widgets/my-widget.html'

widget = MyWidget(template="widgets/my-widget-2.html")
```

Widget class is considered to render a widget itself, without labels, hints, form layout, etc. These things are rendered in parent instance (form or parent field's widget)

Render Types

As we just have learned above, widget labels, form layout, etc are rendered in parent template. But there is some inconsistense, because different widgets can expect different different layout. For example, checkboxes usually should be rendered on the left to the label, while ordinary field's widget should be on the right.

Iktomi provides a way to make this trick. There is a widget attribute called *render_type*, parent instance can use it to figure out how to render the widget, the implementation of expected layout is completely on parent instance:

```
widget = MyWidget(render_type="full-width")
```

There are a few render types supported by default, and you are free to implement own one:

- *'default'*: label is rendered in usual place;
- *'checkbox'*: label and widget are rendered close to each other;
- *'full-width'*: for table-like templates, a widget takes a full row of the form, the label can be rendered above the widget;
- *'hidden'*: label is not rendered, and the widget is rendered in hidden HTML element.

Data Preparations

If you need to pass extra data to template, you can extend *Widget.prepare_data* method:

```
class MyWidget(TextInput):
    def prepare_data(self):
        template_data = TextInput.prepare_data(self)
        value = template_data['value']
        var1 = get_var1(value)
        return dict(template_data,
                    var1=var1)
```

Multiple and Redonly options

The important thing, widget implementation should carry about, is to support *readonly* option, and, optionally, support *multiple* option.

Readonly fields can not be changed by user, and it should be represented in user interface. For example, <*select readonly="readonly"*> or <*input disabled="disabled"*> can be used. It is recommended to still submit readonly widget's value, so duplicating disabled input (which is not submitted) with hidden input containing actual value is fine.

Multiple options indicate that the field has *conv.ListOf* instance as it's converter. Fields of this kind accept multiple values under the same name. From HTML point of view it can be implemented, for example, as <*select multiple="multiple"*> or as multiple checkboxes with the same name.

Widget implementations

Examples of widgets are *TextInput*, *Textarea*, *Select*, *CheckBox*, *HiddenInput*, etc.

Forms Reference

Forms

Fields

Converters

Validators and filters

Widgets

CHAPTER 3

Cli

Cli Reference

Base

manage

iktomi.cli.base.**manage**(*commands*, *argv=None*, *delim=':'*)

Parses argv and runs neccessary command. Is to be used in manage.py file.

Accept a dict with digest name as keys and instances of `Cli` objects as values.

The format of command is the following:

```
./manage.py digest_name:command_name[ arg1[ arg2[...]]][ --key1=kwarg1[...]]
```

where command_name is a part of digest instance method name, args and kwargs are passed to the method. For details, see `Cli` docs.

Cli

class iktomi.cli.base.**Cli**

Base class for all command digests

__call__(*command_name*, **args*, ***kwargs*)

description(*argv0='manage.py'*, *command=None*)

Description outputed to console

class iktomi.cli.lazy.**LazyCli**(*func*)

Wrapper for creating lazy command digests.

Sometimes it is not needed to import all of application parts to start a particular command. LazyCli allows you to define all imports in a function called only on the command:

```
@LazyCli
def db_command():
    import admin
    from admin.environment import db_maker

    from models import initial
    from iktomi.cli import sqla
    return sqla.Sqla(db_maker, initial=initial.install)

# ...

def run(args=sys.argv):
    manage(dict(db=db_command, ), args)
```

Development Server

class iktomi.cli.app.**App**(*app*, *shell_namespace=None*, *extra_files=None*, *bootstrap=None*)

Development application

Parameters

- **app** – iktomi app
- **shell_namespace** – dict with initial namespace for shell command
- **extra_files** – extra files to watch and reload if they are changed
- **bootstrap** – bootstrap function before called dev server is being runned

command_serve(*host=''*, *port='8000'*, *level='debug'*)

Run development server with automated reload on code change:

```
./manage.py app:serve [host] [port] [level]
```

command_shell()

Shell command:

```
./manage.py app:shell
```

Executed with *self.shell_namespace* as local variables namespace.

FCGI Server

class iktomi.cli.fcgi.**Flup**(*app*, *bind=''*, *logfile=None*, *pidfile=None*, *cwd='.'*, *umask=2*, *fastcgi_params=None*)

Flup FastCGI server

Parameters

- **app** – iktomi app
- **bind** – socket file
- **logfile** – log file
- **pidfile** – PID file
- **cwd** – current working directory

- **umask** –
- **fastcgi_params** (*dict*) – arguments accepted by flup *WSGIServer*, plus *preforked*

command_start(*daemonize=False*)
Start a server:

```
./manage.py flup:start [--daemonize]
```

command_stop()
Stop a server:

```
./manage.py flup:stop
```

SQLAlchemy

CHAPTER 4

Templates

Template

iktomi.templates.Template class is originnaly designed to unify template interface for forms, but can be used in anywhere else.

Template object provides *render*, *render_to_response* methods and *render_to* handler factory. The constructor accepts a list of directories for search temlates in (as *args) and following keyworg arguments:

- *globs*.
- *cache*.
- *engines*.

Engine is class providing *render* method, which accepts template name and template arguments as keyword args, and returns rendered string. The constructor shoul accept templates paths list and option switching template cache on/off:

```
class MyEngine(object):
    def __init__(self, paths, cache=False):
        self.engine = MyTemplateEngine(paths, cache=cache)

    def render(self, template_name, **kw):
        template = self.engine.get_template(template_name)
        return template.render(kw)
```

Iktomi supports *jinja2* engine by default.

Now we can instantiate *Template* object with engines we have:

```
from iktomi.templates import jinja2, Template
from iktomi import web

jinja_loader = jinja2.TemplateEngine(cfg.TEMPLATES,
                                     extensions=[])
template = Template(engines={'html': jinja_loader,
```

```
                                'my': MyEngine},
                      *cfg.TEMPLATES)
```

To bound a template object to the iktomi *env*, to add request-specific values to template variables, *BoundTemplate* is used:

```
class BoundTemplate(BaseBoundTemplate):

    constant_template_vars = dict(template_vars)

    def get_template_vars(self):
        lang = self.env.lang
        d = dict(
            lang = self.env.lang,
            url = self.env.root,
            url_for_object = self.env.url_for_object,
            url_for_static = self.env.url_for_static,
            now = datetime.now(),
        )
        return d
```

It is recommended to put it into *env.template* object. Particularly, this is required for correct form rendering. And it may be useful to define *env.render_to_string* and *env.render_to_response* shortcuts:

```
class MyAppEnvironment(web.AppEnvironment):

    # ...

    @storage_cached_property
    def template(storage):
        return BoundTemplate(storage, template_loader)

    @storage_method
    def render_to_string(storage, template_name, _data, *args, **kwargs):
        _data = dict(storage.template_data, **_data)
        result = storage.template.render(template_name, _data, *args, **kwargs)
        return Markup(result)

    @storage_method
    def render_to_response(self, template_name, _data,
                           content_type="text/html"):
        _data = dict(self.template_data, **_data)
        return self.template.render_to_response(template_name, _data,
                                                content_type=content_type)
```

Templates Reference

Template

Jinja2

CHAPTER 5

Utilities

Unsorted stuff for make iktomi working.

Various utilities

Property sugar

iktomi.utils.**cached_property**(*method*, *name=None*)
Turns decorated method into caching property (method is called once on first access to property).

iktomi.utils.**cached_class_property**(*method*, *name=None*)
Turns decorated method into caching class property (method is called once on first access to property of class or any of its instances).

Versioned Storage

Versioned storage, a classes for *env* and *data* objects.

class iktomi.utils.storage.**VersionedStorage**(*cls=<class 'iktomi.utils.storage.StorageFrame'>*, **args*, ***kwargs*)
Storage implements state managing interface, allowing to safely set attributes for *env* and *data* objects.

Safity means that state of the storage is rolled back every time routing case returns ContinueRoute signal (*None*).

Be very accurate defining methods or properties for storage, choose correct method or property type depending on what do you want to achieve.

Regular methods will hold the state of storage frame they are added to. If you want to have an access to actual value, use storage property and method decorators.

as_dict()
Returns attributes of storage as dict

class iktomi.utils.storage.**StorageFrame**(*_parent_storage=None*, ***kwargs*)
A single frame in the storage

class iktomi.utils.storage.**storage_property**(*method*, *name=None*)
Turns decorated method into storage property (method is called with VersionedStorage as self).

class iktomi.utils.storage.**storage_cached_property**(*method*, *name=None*)
Turns decorated method into storage cached property (method is called only once with VersionedStorage as self).

iktomi.utils.storage.**storage_method**(*func*)
Calls decorated method with VersionedStorage as self

Internationalization

iktomi.utils.**N_**(*msg*)
Single translatable string marker. Does nothing, just a marker for *.pot file compilers.

Usage:

```
n = N_('translate me')
translated = env.gettext(n)
```

class iktomi.utils.**M_**(*single*, *plural*, *count_field='count'*, *format_args=None*)
Marker for translatable string with plural form. Does not make a translation, just incapsulates a data about the translatable string.

Parameters

- **single** – a single form
- **plural** – a plural form. Count can be included in %-format syntax
- **count_field** – a key used to format

Usage:

```
message = M_(u'max length is %(max)d symbol',
             u'max length is %(max)d symbols',
             count_field="max")
m = message % {'max': 10}
trans = env.ngettext(m.single,
                     m.plural,
                     m.count
                     ) % m.format_args
```

count
A count based on *count_field* and *format_args*.

Paginator

Other

iktomi.utils.**quoteattr**(*value*)
Works like quoteattr from saxutils (returns escaped string in quotes), but is safe for HTML

iktomi.utils.**quoteattrs**(*data*)
Takes dict of attributes and returns their HTML representation

iktomi.utils.**quote_js**(*text*)
Quotes text to be used as JavaScript string in HTML templates. The result doesn't contain surrounding quotes.

iktomi.utils.**weakproxy**(*obj*)
Safe version of weakref.proxy.

iktomi.utils.deprecation.**deprecated**(*comment=None*)
This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used. Usage:

```
@deprecated()
def foo():
    pass
```

or:

```
@deprecated('Use bar() instead.')
def foo():
    pass
```

iktomi.utils.dt.**strftime**(*dt*, *fmt*)
strftime implementation working before 1900

Utils

Python Module Index

i

iktomi.cli.base, 27
iktomi.forms.convs, 26
iktomi.forms.fields, 26
iktomi.forms.form, 26
iktomi.forms.widgets, 26
iktomi.templates, 32
iktomi.templates.jinja2, 32
iktomi.utils, 33
iktomi.utils.storage, 33
iktomi.web.app, 10
iktomi.web.core, 10
iktomi.web.filters, 10
iktomi.web.reverse, 10
iktomi.web.url, 10
iktomi.web.url_converters, 10

Index

Symbols

__call__() (iktomi.cli.base.Cli method), 27

A

App (class in iktomi.cli.app), 28
as_dict() (iktomi.utils.storage.VersionedStorage method), 33

C

cached_class_property() (in module iktomi.utils), 33
cached_property() (in module iktomi.utils), 33
Cli (class in iktomi.cli.base), 27
command_serve() (iktomi.cli.app.App method), 28
command_shell() (iktomi.cli.app.App method), 28
command_start() (iktomi.cli.fcgi.Flup method), 29
command_stop() (iktomi.cli.fcgi.Flup method), 29
count (iktomi.utils.M_ attribute), 34

D

deprecated() (in module iktomi.utils.deprecation), 35
description() (iktomi.cli.base.Cli method), 27

F

Flup (class in iktomi.cli.fcgi), 28

I

iktomi.cli.base (module), 27
iktomi.forms.convs (module), 26
iktomi.forms.fields (module), 26
iktomi.forms.form (module), 26
iktomi.forms.widgets (module), 26
iktomi.templates (module), 32
iktomi.templates.jinja2 (module), 32
iktomi.utils (module), 33
iktomi.utils.storage (module), 33
iktomi.web.app (module), 10
iktomi.web.core (module), 10
iktomi.web.filters (module), 10
iktomi.web.reverse (module), 10
iktomi.web.url (module), 10
iktomi.web.url_converters (module), 10

L

LazyCli (class in iktomi.cli.lazy), 27

M

M_ (class in iktomi.utils), 34
manage() (in module iktomi.cli.base), 27

N

N_() (in module iktomi.utils), 34

Q

quote_js() (in module iktomi.utils), 34
quoteattr() (in module iktomi.utils), 34
quoteattrs() (in module iktomi.utils), 34

S

storage_cached_property (class in iktomi.utils.storage), 34
storage_method() (in module iktomi.utils.storage), 34
storage_property (class in iktomi.utils.storage), 34
StorageFrame (class in iktomi.utils.storage), 33
strftime() (in module iktomi.utils.dt), 35

V

VersionedStorage (class in iktomi.utils.storage), 33

W

weakproxy() (in module iktomi.utils), 35