uiro Documentation

Release 0.2

Hiroki KIYOHARA

November 08, 2013

Contents

le Web framework.

A simple Uiro application will be like this.

```
from wsgiref.simple_server import make_server
from matcha import Matching, make_wsgi_app
from uiro.controller import BaseController
from uiro.view import view_config


class Controller(BaseController):
    @view_config(method='get')
    def get_view(self, request, context):
        return 'Hello {name}!'.format(**request.matched_dict)

 matching = Matching('/hello/{name}', Controller())

if __name__ == '__main__':
    app = make_wsgi_app(matching)
    server = make_server('0.0.0.0', 8888, app)
    server.serve_forever()
```

And setup.

```
pip install uiro
python hello.py
```

Now, you can visit http://localhost:8888/hello/world in a browser, you will see the text 'Hello world!'.

CHAPTER 1

Next step

Above example is too tiny to create a common-sensible Web application. On next step, you can create your first project through *Starting your first package* documentation. In this doc, you can create an application package, not just for a example.

To learn more about Uiro browse these topics:

1.1 Starting your first package

1.1.1 Installing

Create your python env and install it from PyPI:

```
pip install uiro
```

1.1.2 First project

After installing Uiro, 'gearbox' command will be available on your env. You can manage projects and applications by using this command. Now let's create your first Uiro package:

```
gearbox create -n packagename
```

Above 'packagename' string should be replaced to some another name you want.

And then install created package:

```
cd packagename
python setup.py develop
```

This action makes your created package available on your env.

Then create SQLite DB as a file named 'default.db' to current dir:

```
gearbox initdb
```

The setting for DB is written in development.ini, and some another setting too, check it out.

Finally, you can serve your application, by *serve* command:

```
gearbox serve
```

Then, you can run your web browser and access to localhost with port 8888 to confirm automatically created package is running.

1.1.3 What files are in your package?

Your package must contain these packages:

```
.
-- packagename
|   -- __init__.py
|   -- views.py
|   -- matching.py
|   -- models.py
|   -- templates
|   |   -- top.mako
|   -- static
|       -- uiro.css
-- development.ini
-- setup.py
-- README.rst
-- CHANGES.txt
-- MANIFEST.in
```

views.py Module to store Controllers which bundles each Views.

See *Writing your views* documentation.

matching.py Entry point for each Controllers, specifying which Controller should be called corresponds to URL gave by clients.

See *URL Dispatching* documentation.

models.py Module to store Models, which is abstraction layer for structuring and manipulating the database. Uiro framework depends on SQLAlchemy, you can write your own Models by SQLAlchemy more easily.

See *Connectiong to RDBs using SQLAlchemy* documentation.

templates Directory to store mako templates. Templates will be collected automatically. The top.mako template can be picked up as a signature like 'packagename:top.mako'.

See *Using mako templates* documentation.

static Directory to store static files. Static files will be also collected automatically and served, so you should not handle them manually. The uiro.css file will be in /static/packagename/uiro.css, and the link can be generated, using request.matching.

See *Serving static files* documentation.

development.ini Configation file for WSGI application. It specifies that which Matching to use, which Database to connect and so on.

See *Application configuration* documentation.

setup.py Builder your package, which usually tells you that the module/package you are about to install have been packaged and distributed.

See setuptools documentation.

MANIFEST.in File to specify which file should be contained in package.

CHANGES.txt Text file to describe your package's change logs.

README.rst README file for your package.

1.2 Writing your views

"Views" are methods to encapsulate the logic responsible for...:

- Processing a user's request and context determined data by request.
- Returning the response, such as response object, dictionary or string.

Commonly views will look like this:

```
from uiro.controller import BaseController
from uiro.view import view_config

class MyController(BaseController):
    @view_config(method='get')
    def get_view(self, request, context):
        return 'Hello world!'
```

This MyController class is WSGI application returns response containing text 'Hello world!' in it's body.

- Controller is a WSGI application.
- view_config is decorator to construct methods as view.

Views are logic about interfaces. Storing business logic in views is not recommended.

1.2.1 Testing views without decorators

On Unit testing, you should test target logic without any other things. View methods must be applied view_config decorator, so It seems that it is difficult to test views without any decorators.

Don't worry, Uiro provide a feature allowing you to write tests without decorators. You can test views like this:

```
>>> target = MyController().get_view
>>> assert target('dummy_request', 'dummy_context') == 'Hello world!'
```

1.2.2 Responsibility of controllers

Constructing a WSGI application from views. It checks which view should be called and dispatching. You don't need to write any logic for controllers. All of them have been determined by Uiro framework. It is only used as a container for views like above example.

You can change logic in controller, specifying some values to interface provided by it's own. The best example of this is *resource*, it is object to some resources on app determined by a request. For more detail, see Apply resources for views.

Note: The number of APIs provided by Controllers should be as little as possible. Uiro should not force users to remember a lot of APIs. it will be labor for users and generally it will be difficult to use. and what is worth, changing APIs may be hard work so Uiro will become inflexible increasingly.

1.2.3 Apply resources for views

For many cases, necessary data for one view can be determined by only a request. And It should be separate from views, to increase testability and readability:

- Separating logic to collect data form views.
- Allowing to dispatch views corresponding to collected data in context.

It will be applied request object and you can write logic to collect data in it. It can be specified *resource* attribute in your Controller. A controller apply request to class in *resource* attribute and pass it to each view methods.

You can use this behavior like this:

```
from .models import Page

class PageResource(object):
    def __init__(self, request):
        self.request = request

    @property
    def page(self):
        return Page.query.filter_by(id=request.matched_dict['id']).one()

class Controller(BaseController):
    resource = PageResource

    @view_config(method='get'):
    def get_view(request, context):
        return {'page': context.page}
```

Hereby, you separated collection logic and view (user interface). When you test each views, you can pass dummy request and context easily. you can focus writing tests for about interfaces.

Note: It's better to store resource classes in a separated module to correspond to each models. Above example, The PageResource class in *page.py* seems better. Then of cause, you will store another logic for the Page model in *page.py* too.

1.3 URL Dispatching

1.3.1 Registering Controller

You can write dispatcher in *yourpacake/matching.py* like this:

```
from matcha import Matching as m, bundle
from .views import DashbordController, PageController

matching = bundle(
    m('/', DashbordController(), name='dashbord'),
    m('/page/{slug}', PageController(), name='page'),
)
```

Then, client accessing:

localhost/ DashboardController will be dispatched

localhost/page/hello_word PageController will be dispatched

localhost/page/about_ritsu also PageController will be dispatched

Uiro is using matcha dispatcher. For more details about dispatching, watch matcha documentation.

1.3.2 Getting URL arguments from request

When accessing '/page/hello_word', you can that 'slug' value in your views.

```
>>> # In your views
>>> request.matched_dict['slug']
'hello_world'
```

1.3.3 Getting URL for each controllers

```
request.matching.reverse('page', slug='hello_world')
```

request.matching is actually same value with matching object constructed by above example. Uiro watches matching object and call controllers, then it assigns taken matching object to request object.

1.3.4 Actually behavior

To choice which Controller should be called, Uiro is using matcha dispatcher. Uiro will automatically construct application from a matching object of matcha specified by configuration .ini file.

In .ini file, the *uiro.root_matching* is key to specify core matching object for your application. the value should be splittable by colon (':'), then, the left value is path to module to store your root matching object, and the right value is it's name.

If you write a setting like this:

```
uiro.root_matching = path.to.yourmodule:matchingobject
```

follow matching object will be used as root matching to construct your application.

```
# In path.to.yourmodule module.
matchingobject = Matching('/', SomeWSGIApp())
```

Constructed application will call SomeWSGIApp when the PATH_INFO is '/'.

1.4 Connectiong to RDBs using SQLAlchemy

Uiro depends on SQLAlchemy, a powerful ORM for python. It only provides these features:

- Base class and Session for SQLAlchemy to write models
- Specifying zope.sqlalchemy extention
- A command to create tables
- Setting about database by .ini file

1.4.1 Creating your Model

It's same with SQLAlchemy. Just using Base/Session provided by Uiro to creating tables automatically.

```
import sqlalchemy as sa
from uiro.db import Base, Session

class MyModel(Base):
    __tablename__ = 'mymodel'
    query = Session.query_property()

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String(255))

    def __init__(self, name):
        self.name = name
```

And using like this:

```
>>> import transaction
>>> with transaction.manager:
...     Session.add(MyModel(name='spam'))
```

You can see SQLAlchemy and zope.sqlalchemy documentation to learn more.

1.4.2 Create tables by initdb command

You can initialize DB with following gearbox command:

```
gearbox initdb
```

This command is to create tables to specified database.

1.4.3 Setting about database to use

To change database to use, specify that URL to *sqlalchemy.url* setting in .ini file:

```
sqlalchemy.url = sqlite:///default.db
```

1.5 Using mako templates

Uiro depends on mako template, and provide some shortcuts to use templates for applications. Templates for an application should store in *yourpackage/templates* directory. Uiro will collect templates and correlate the app and templates.

You learn how to use shortcuts here, but not about mako template usage. you should see mako documentation and learn about it.

1.5.1 Easiest way, by view

Most easiest way to get and render a template is specifying it by *view_config*, like this.

```
@view_config(method='get',
             template_name='blog:entry.mako')
def get_view(self, request, context):
    entry = context['blog_entry']
    return {'entry': entry}
```

If views return a dictionary, *view_config* will handle it as context dictionary for a template specified by *template_name*. *view_config* will render it and use it as Response body.

You can specify a template by passing a template name that is string splittable by a colon. The left value is package name, and the right value is template name, like 'blog:entry.mako'.

The template used in above example must be here:

```
blog/templates/entry.mako
```

It was collected automatically, so you can use it only specifying the signature.

1.5.2 Simplest way

You can get template by a getter function *uiro.template.get_template*. It simply returns mako template object, so you can render it by *.render* method.

```
from uiro.template import get_template
get_template('blog:entry.mako').render(entry='blog entry')
```

1.6 Serving static files

Uiro prepares a feature to collect static files and serve it. It will collect static files as same way for templates (*Using mako templates*).

It will create an app for serving static files and register in in URL dispatcher. Static files will be collected from directory named 'static' under your application:

```
./blog/static/
```

This example is with an application named *blog*. URLs for static files in static directory will begin with */static/app_name/*. so in blog app case, if the directory has css/main.css file, the file will be published like this:

```
yoursite.com/static/blog/css/main.css
```

1.6.1 Getting URL for a static files

You can get this URL by reversing form matching object

```
request.matching.reverse('blog:static', path=['css', 'main.css'])
```

1.7 Application configuration

In this documentation, sometime you may write same configurations to .ini file.

1.7.1 Uiro-limited configurations

uiro.root_matching Specifying which object to use as root matching to construct URL dispatcher.

uiro.installed_apps Uiro packages to initialize, collecting templates, static files, DBs and so on.

More about detail, you see PasteDeploy documentation and learn.

1.8 Uiro API reference

package API reference.

1.8.1 uiro package

Subpackages

uiro.commands package

uiro.commands.initdb module

class `uiro.commands.initdb.`**`InitDBCommand`**(*app*, *app_args*)

Bases: `uiro.commands.LoadAppCommand`

Creating database tables.

`take_action`(*self*, *parsed_args*)

uiro.commands.shell module

class `uiro.commands.shell.`**`ShellCommand`**(*app*, *app_args*)

Bases: `uiro.commands.LoadAppCommand`

Running python shell after building up an uiro application.

`make_default_shell`(*self*, *interact=<function interact at 0x7f4b6f9429e0>*)

`take_action`(*self*, *parsed_args*)

uiro.commands.create module

class `uiro.commands.create.command.`**`CreateCommand`**(*app*, *app_args*)

Bases: `gearbox.command.TemplateCommand`

`CLEAN_PACKAGE_NAME_RE` **= <_sre.SRE_Pattern object at 0x7f4b6f062e10>**

`get_description`(*self*)

`get_parser`(*self*, *prog_name*)

`take_action`(*self*, *opts*)

Module contents

class `uiro.commands.`**`LoadAppCommand`**(*app*, *app_args*)

Bases: `gearbox.command.Command`

Base class for creating uiro commands.

You can override this class and call *loadadd* method to get WSGI application built by paste.app_factory. While building the application, almost necessary setup will be done (for example setup databases, template lookups and so on), so then you can run some application-dependent scripts

get_parser(*self*, *prog_name*)

loadapp(*self*, *parsed_args*)

uiro.controller module

class uiro.controller.**BaseController**
Bases: builtins.object

Base WSGI application class to handle Views.

Controllers try to call methods wrapped by *uiro.view.view_config*. But actually it will call _wrapped attribute of each Views:

- Original View methods can be called without any decorators. This behavior is provided for ensuring depending-less tests.
- When wrapped View raised ViewNotMatched, it will try next one.
- All of views are not matched, it will return 404 response.

You can inherit this class and register views. Then, decorate views with uiro.view.view_config to apply configation to each views, such as witch views will be call or witch template to use.

```
class DashboardController(BaseController):
    @view_config(method='get')
    def get_view(self, request):
        return "Hello guys"

    @view_config(method='post')
    def post_view(self, request):
        return "Posted something"
```

Check the behavior of view_config for more detail.

resource(*s*, *x*)

views = []

class uiro.controller.**ControllerMetaClass**
Bases: builtins.type

exception uiro.controller.**NotFound**
Bases: builtins.Exception

Error for notifying the resource was not found.

uiro.db module

uiro.db.**initdb**(*config*)
Initializing database settings by using config from .ini file.

uiro.request module

class uiro.request.**Request**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)

Bases: webob.request.BaseRequest

matched_dict

matching

uiro.static module

uiro.static.**generate_static_matching**(*app*, *directory_serve_app=<class 'webob.static.DirectoryApp'>*)

Creating a matching for WSGI application to serve static files for passed app.

Static files will be collected from directory named 'static' under passed application:

```
./blog/static/
```

This example is with an application named *blog*. URLs for static files in static directory will begin with /static/app_name/. so in blog app case, if the directory has css/main.css file, the file will be published like this:

```
yoursite.com/static/blog/css/main.css
```

And you can get this URL by reversing form matching object:

```
matching.reverse('blog:static', path=['css', 'main.css'])
```

uiro.static.**get_static_app_matching**(*apps*)

Returning a matching containing applications to serve static files correspond to each passed applications.

uiro.template module

uiro.template.**get_app_template**(*name*)

Getter function of templates for each applications.

Argument *name* will be interpreted as colon separated, the left value means application name, right value means a template name.

get_app_template('blog:dashboarb.mako')

It will return a template for dashboard page of *blog* application.

uiro.template.**get_lookups**()

Returning the lookups

The global variable _lookups should not be imported directory by another modules. By importing directory, the value will not change evenif setup_lookup

uiro.template.**setup_lookup**(*apps*, *lookup_class=<class 'mako.lookup.TemplateLookup'>*)

Registering template directories of apps to Lookup.

Lookups will be set up as dictionary, app name as key and lookup for this app will be it's value. Each lookups is correspond to each template directories of apps._lookups. The directory should be named 'templates', and put under app directory.

uiro.view module

class uiro.view.**MethodPredicate**(*method*)
Bases: builtins.object

Predicate class to checking Method of request object.

MethodPredicate is preserve views when the request method was not same with applied in instantiate.

exception uiro.view.**ViewNotMatched**
Bases: builtins.Exception

Called view was not apposite. This exception is to notify Controllers that called view was not apposite to the applied rquest.

uiro.view.**get_base_wrappers**(*method='get'*, *template_name=''*, *predicates=()*, *wrappers=()*)
basic View Wrappers used by view_config.

uiro.view.**preserve_view**(**predicates*)
Raising ViewNotMatched when applied request was not apposite.

preserve_view calls all Predicates and when return values of them was all True it will call a wrapped view. It raises ViewNotMatched if this is not the case.

Predicates: This decorator takes Predicates one or more, Predicate is callable to return True or False in response to inputted request. If the request was apposite it should return True.

uiro.view.**render_template**(*template_name*, *template_getter=<function get_app_template at 0x7f4b6f05ea70>*)
Decorator to specify which template to use for Wrapped Views.

It will return string rendered by specified template and returned dictionary from wrapped views as a context for template. The returned value was not dictionary, it does nothing, just returns the result.

uiro.view.**view_config**(*method='get'*, *template_name=''*, *predicates=()*, *wrappers=()*, *base_wrappers_getter=<function get_base_wrappers at 0x7f4b6ea55320>*)
Creating Views applied some configurations and store it to _wrapped attribute on each Views.

- _wrapped expects to be called by Controller (subclasses of uiro.controller.BaseController)
- The original view will not be affected by this decorator.

Module contents

uiro.**import_module_attribute**(*path*, *splitter=':'*)

uiro.**main**(*global_conf*, *root*, ***settings*)
Entry point to create Uiro application.

Setup all of necessary things:

- Getting root matching
- Initializing DB connection
- Initializing Template Lookups
- Collecting installed applications
- Creating apps for serving static files

and will create/return Uiro application.

Python Module Index

u