Rückenwind Documentation

Release 0.4.0-alpha

Florian Ludwig

May 24, 2018

Contents

Contents:

CHAPTER 1

Getting started

1.1 Design Principles and Goals

- **KISS** Keept it simple, stupid.
- **Convention over configuration** But don't take it too far.
- **Reusablity** I will not repeat myself.
- **Don't try to please everyone**

Note: Rückenwind is in **alpha** stage of development, backwards compatibility will break. And actually did: 0.3.x and 0.4.x are quite different.

1.2 Install

Use pip:

```
virtualenv my_playground
. ./my_playground/bin/activate
pip install rueckenwind
```

1.3 Quick start

It is highly recommended to develop and deploy within a virtualenv. Always. So this documentation just assumes you do without further mentioning it. After installing rückenwind you got a new command at your disposal: *rw*. You can use it to generate a new rückendwind project skeleton:

```
rw skel --name my_new_project
```

To start your new project:

```
rw serv my_new_project.http
```

Go and visit 127.0.0.1:8000 to grab your *hello world* while it is hot.

1.4 Modules

The most basic entity in rückenwind is a module. A rückenwind module is a normal python module with some added convention.

The basic structure:

```
module/
        __init__.py
        http.py
        static/
        templates/
        locale/
```

The obvious: all your static files go into the static/ folder and your jinja2 templates into the templates/ folder.

The http.py must contain a Module named root. It is the entry point for all http requests to your app.

Note: Rückenwind does not try to be a framework for everyone and everything. As one of the consequences only a single templating engine is supported. This keeps rückendwind code KISS. Don't like jinja? Sorry, rückenwind is not for you, switch to one of the many other frameworks.

1.5 Routing

At the heart of rückenwind there is routing of http requests. It draws inspiration from several other projects, like Flask .

Note:

- **HTTP methods** are strictly seperated. You cannot have one python function answering GET and POST.

An example RequestHandler

```
import time

import rw.http
import rw.gen


registration = rw.http.Module('my_playground')
```

(continues on next page)

(continued from previous page)

```
@registration.get('/')
def register(handler):
    handler.render_template(template='register.html')


@registration.post('/')
def register_post(handler):
    u = User(email=handler.get_argument('email'),
             username=handler.get_argument('password'))
    handler.redirect(url_for(main))


root = rw.http.Module('my_playground')


@root.get('/')
def main(handler):
    handler['time'] = time.strftime("%a, %d %b %Y %H:%M:%S +0000", time.gmtime())
    root.render_template('index.html')


@root.get('/entry/<id>')
@rw.gen.coroutine
def entry(self, id):
    self['entries'] = yield get_entry_from_id(id)
    self.finish(template='my_playground/main.html')


root.mount('/register', registration)
```

For details see: *Basic HTTP and Services*

1.6 Templates

For documentation about the Jinja templating engine please look at its beautiful online documentation .

Assigning variables:

```
@root.get('/')
def main(handler):
    handler['time'] = strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
    root.finish(template='my_playground/index.html')
```

Within the template:

```
The current time is {{ time }}.
```

If you refer to another resource there are two helper functions for creating URIs. For static files use:

```
{{ static('main.css') }}
```

This will insert an URI to your *main.css*, assuming there is one in your modules static folder.

If you want to link to another page there is:

```
{{ url_for(handler.login) }}
```

Same routes as before:

```
class Main(RequestHandler):
    @get('/')
    def main(self):
        # ...

    @get('/register')
    def register(self):
        # ...

    @get('/entry/<id>')
    def entry(self, id):
        # ...
```

command	result
url_for(handler.login)	/
url_for(handler.register)	/register
url_for(handler.entry, id=1)	/entry/1

CHAPTER 2

Basic HTTP and Services

CHAPTER 3

Static Content

You probably got content that is for every user the same on every request. Typically this is content like images, css or javascript. Downloads (as in content that is not directly displayed from html) or for this matter large files in generall should treated a little different.

You probably remember that there is a function called “static” to refer to your static content:

```
{{ static('main.css') }}
```

The url it generates looks something like this:

```
/static/simpleblog/main.css?v=a37b
```

All static content is below /static/. Followed by the modules name (simpleblog here). Appended is version string to ensure the browser got the right thing in its cache. The version is the first few bytes of the hexdecimal representation of the md5 of the content of the main.css.

3.1 Dynamic Static Content

The definition of “static content” in rückenwind is content that is the same on every request for every user.

3.2 Deployment

CHAPTER 4

Configuration Management

For your convenience Rückenwind comes with simple (as in KISS) configuration management. When you start up your project Rückenwind will try to load its configuration and you might see logging like:

```
rw[INFO] reading config: /my_virtualenv/src/my_cool_project_git/myproject/myproject.
↪cfg
rw[INFO] reading config: /etc/myproject.cfg
rw[INFO] reading config: /home/joe/.myproject.cfg
rw[INFO] reading config: /my_virtualenv/etc/myproject.cfg
```

The order the files appear is of importance. Values in the later overwrite values in the former. So you can specify default values for your configuration values inside your project that gets overwritten by system wide configuration in /etc that gets overwritten by user configuration in their homes that gets overwritten by configuratin inside your virtual_envs /etc.

The tornado server can be configured with httpserver:

```
httpserver:
  xheaders: true
```

Note:

```
TODO: Configs are now yaml format. Document.
```

For example if your your project features the following config file:

```
[mongodb]
host = 127.0.0.1
db = some_db
```

And your ~/.myproject.cfg is:

```
[mongodb]
host = db.local
```

You will get the following configuration dict:

```
{'mongodb': {
    'host': 'db.local'
    'db': 'some_db'
}}
```

Actually it is not a standard python dict but a ConfigObj and therefor provides some extra methods like as_bool. It can be accessed via `rw.cfg`. Please note that the dict gets populated when rw setups your project. So if you use:

```
rw serv mycoolproject
```

The dict is populated before your module is loaded - meaning you can access `rw.cfg` without worries. In circumstances were you are not running through the rw start command you might want to either use `rw.setup('mycoolproject')` to load your module (which is what `rw serv` does). In other circumstances you might just want the config you can use:

Design Decisions: - JSON does not support comments - ini is not strictly typed

CHAPTER 5

TODO

The documentation is far from completion, here are some TODOs

- plugin
- debugging
- i18n
- deployment

CHAPTER 6

Indices and tables

- genindex
- modindex
- search