RedBarrel Documentation

Release 1.0

2011, Tarek Ziadé

August 08, 2011

CONTENTS

Warning: The RedBarrel project is an experiment. Do not use in production !

Contents:

CHAPTER

ONE

WHAT'S REDBARREL ?

tldr; **RedBarrel is a DSL and Micro-Framework to write Web Services**

RedBarrel allows you to describe your Web Services via a Domain Specific Language (DSL) and organize your code in a very simple way. The DSL replaces the code usually needed to route you request to the right piece of code, and let you describe all the pre- and post- steps.

RedBarrel can be used to write any kind of web application, but is specifically designed to write Web Services.

1.1 Anatomy of a Web Service

A Web Service is basically doing these four steps:

1. **[pre-processing]** Check the request body and headers, and potentially reject it. Rejection can be due to a Basic Authentication failure, an unexpected value for the request body, etc.
2. **[routing]** Find what code or application should be called to build the response. This is usually computed with the path information and sometimes some headers.
3. **[execution]** Invoke some code to build the response
4. **[post-processing]** Return the response built and maybe do some post-processing or post-assertions like converting the content-type etc.

In Python, when you build web services using a WSGI framework like Pylons, Pyramid, or simply Routes + WebOb, all of these steps happen in your code. You define the routing using Routes descriptions, or using more clever dispatching systems like what Pyramid offers, then delegate the execution to a controller class or a simple function, after a potential pre-processing. Although the pre-processing part is often merged with the execution part because they are closely related.

For instance, if you have a web service that requires a JSON mapping in the request body, you could write something that looks like:

```
def my_webservice(request):
    try:
        data = json.loads(request.body)
    except ValueError:
        # this raises a 400
        raise HTTPBadRequest("Unknown format -- we want JSON")

    ... do something ...
```

Of course you can always generalize this by using a decorator to clearly separate the pre-processing part:

```
@if_not_json(400)
def my_webservice(request):
    ... do something ...
```

Same thing for the post-processing step:

```
@if_not_json(400)
@convert_output('application/json')
def my_webservice(request):
    ... do something ...
```

And in some frameworks, the routing itself is expressed as a decorator:

```
@route('/here/is/my/webservice')
@if_not_json(400)
@convert_output('application/json')
def my_webservice(request):
    ... do something ...
```

It turns out that there are a lot of pre/post steps that can be pushed to a **meta level**.

With RedBarrel, Steps 1., 2. and 4. are described in a Domain Specific Language (DSL) instead of decorators or other meta-level code.

1.2 The RBR DSL

RedBarrel pushes all the pre- and post- processing descriptions in a DSL called **RBR**. The RBR file also describes in detail every web service, like the path to invoke them, which HTTP method should be used, what possible status codes will be returned, etc.

The first benefit is a natural separation of the code that actually does the job, from the code that checks or converts inputs and outputs. When you are developing a web service, you don't have to worry anymore about checking the request or setting the response content-type. You can just focus on the feature. This separation favors code reuse and reduces the boiler-plate code of your application: all the post- and pre- conditions can be grouped in a library that's reused accross all your web services.

A DSL also allows a possible delegation of pre/post tasks to a higher layer: if the RedBarrel DSL was to be implemented as an Nginx module all the pre- and post- processing could happen at the web server level, allowing an interesting leverage of resources. Rejecting bad request in NGinx makes more sense than doing it in the Python backend.

Last, expressing web services in a DSL facilitates introspection: the RedBarrel default implementation for instance, creates a HTML view of all available webservices.

1.3 Writing an application with RedBarrel

Writing a web application with RedBarrel consists of describing the logic of every web service in a RBR file, then worry about the implementation of every piece. There is no MVC paradigm or anything similar that the developer must follow. You can write your code in functions, classes or whatever, and just hook them into the RBR file.

For example, the RBR for a simple Hello world app can look like this:

```
path hello_world (
    description "Simplest application: Hello World!",
    method GET,
```

```
    url /,
    use python:somemodule.hello
);
```

With a function *hello* located in *somemodule*:

```
def hello(request):
    return 'Hello World'
```

Have a look at a full example: *Full example: an URL shortener*.

1.4 What do you get with RedBarrel ?

The current version includes:

- a parser for the RBR DSL.
- a RBR syntax checker.
- a pure Python implementation of a WSGI Web Server that can be launched with a RBR file.
- a documentation generator.
- a wizard to quick-start a project

What I'd like to have in a future revision:

- a NGinx Module implementation
- an Apache implementation
- more backends than Python, so web services can execute JS, Erlang, etc
- contributors

1.5 How to contribute ?

If you like the idea, you can contribute to the project, by:

- using RedBarrel in one of your project and providing feedback
- contributing code and/or documentation
- etc.

Contact me at tarek_at_ziade.org

CHAPTER

TWO

INSTALLING REDBARREL

First, make sure you have Pip, Distribute and Virtualenv installed.

You can quickly set things up with:

```
$ curl -O http://python-distribute.org/distribute_setup.py
$ python distribute_setup.py
$ easy_install pip virtualenv
```

Installing the latest stable RedBarrel release is then done with:

```
$ pip install redbarrel
```

If you want to try the cutting-edge version, grab the repository and create a local virtualenv:

```
$ hg clone https://bitbucket.org/tarek/redbarrel
$ cd redbarrel
$ virtualenv --distribute .
$ bin/python setup.py develop
```

In this setup, all RedBarrel scripts will be located in the **bin** directory.

You can also setup bash completion:

```
$ sudo cp bash_completion.d/redbarrel /etc/bash_completion.d/
```

CHAPTER

THREE

QUICKSTART

A RedBarrel-based application is composed of:

- a RBR file: a text file that contains a description of the application.
- a Python package containing the code that's going to be executed

3.1 Creating a project

To create the initial structure, create an empty directory and run the **rb-quickstart** command. It will create for you an example sample RBR file and a Python package containing the code used by the example – a simple hello world.

The script will ask you a few questions

```
$ rb-quickstart
Name of the project: Hello World !
Description: A simple app
Version [1.0]:
Home page of the project: http://example.com
Author: Tarek
Author e-mail: tarek@ziade.org
App generated. Run your app with "rb-run shortme.rbr"
```

Once the files are generated, run your application using **rb-run**:

```
$ bin/rb-run hello-wo.rbr
Generating the Web App...
=> 'hello' hooked for '/'
...ready

Serving on port 8000...
```

Then, visit *http://localhost:8000/__doc__*. You should see the documentation page of your application.

3.2 Adding web services

Adding a web service consists of describing it in the RBR file (see *RBR specification*), then adding the required code for every pre- or post- step, or for the service itself.

A possible convention is to add one Python module per service, and eventually group all pre- and post- functions in an `util.py` module.

Check out a full example at *Full example: an URL shortener* .

CHAPTER

FOUR

FULL EXAMPLE: AN URL SHORTENER

Warning: We make the assumption here that you are familiar with WebOb and Routes

You can find a more complex version of this demo , and other demos at: http://bitbucket.org/tarek/redbarrel/src/tip/redbarrel/demos

Let's create an URL shortener with RedBarrel. The shortener has the following features:

- an api to generate a shortened URL, given an URL
- a redirect to the target URL
- an api to get visit statistics for every redirect
- an HTML page with a form to create a shortener

There's no authentication at all for the sake of simplicity.

4.1 Step 1: the RBR file

Let's create a RBR file with the **rb-quickstart** command:

```
$ rb-quickstart
Name of the project: ShortMe
Description: An URL Shortener
Version [1.0]:
Home page of the project: http://example.com
Author: Tarek
Author e-mail: tarek@ziade.org
App generated. Run your app with "rb-run shortme.rbr"
```

The RBR file produced provides a simple hello world API:

```
# generated by RedBarrel's rb-quickstart script
meta (
    description """An URL Shortener""",
    version 1.0
);

path hello (
    description "Simplest service",
    method GET,
    url /,
    response-body (
```

```
        description "Returns an Hello word text"
    ),
    response-headers (
        set Content-Type "text/plain"
    ),
    use python:shortme.hello.hello
);
```

Let's remove the **hello** definition and add ours. We want to add:

- a POST for the URL shortener
- a GET for the stats
- the redirect
- the HTML page and its corresponding action

Let's describe the POST first. We want the URL to shorten in the request body, and get the result back in the response body:

```
path shorten (
    description "Shortens an URL",
    method POST,
    url /shorten,
    request-body (
        description "Contains the URL to shorten"
    ),
    response-body (
        description "The shortened URL"
    ),
    response-headers (
        set Content-Type "text/plain"
    ),
    response-status (
        describe 200 "Success",
        describe 400 "I don't like the URL provided"
    ),

    use python:shortme.shorten.shorten
);
```

The description is quite simple:

- the service is mapped at */shorten*
- the url to be shortened is provided in a POST body
- the response is a plain text with the shortened url
- the server returns a 200 or a 400
- the code will be located in the `shorten()` function in the *shorten* module in the *shortme* package.

The way the shortener works is:

1. a unique id is created on the server for every new URL, and kept in a dict
2. **/shorten** returns a URL that looks like : */r/ID*
3. When the redirect API is called, the server looks for the URL in the dict and redirects to it with a 303. In case it's not present in the dict, a 404 is returned.

The redirect is expressed as follows:

```
path redirect (
    description "Redirects",
    method GET,
    url /r/{redirect},
    response-status (
        describe 303 "Redirecting to original",
        describe 404 "Not Found"
    ),
    use python:shortme.shorten.redirect
);
```

The stats service computes statistics and returns them in json:

```
path stats (
    description "Returns a number of hits per redirects",
    method GET,
    url /stats,
    response-status (
        describe 200 "OK",
        describe 503 "Something went wrong"
    ),
    response-body (
        description "A mapping of url/hits",
        unless type is json return 503
    ),
    response-headers (
        set Content-Type "application/json"
    ),
    use python:shortme.shorten.stats
);
```

Stats are just simple counters for every URL, that get incremented everytime a redirect is done.

For the HTML page that let people create shortened URL, we return a page created with a template:

```
path shorten_html (
    description "HTML view to shorten an URL",
    method GET,
    url /shorten.html,
    response-headers (
        set Content-Type "text/html"
    ),
    response-status (
        describe 200 "Success"
    ),
    use python:shortme.shorten.shorten_html
);
```

Last, a second page is displayed for the shortening result:

```
path shortened_html (
    description "HTML page that displays the result",
    method GET,
    url /shortened.html,
    response-headers (
        set Content-Type "text/html"
    ),
    response-status (
        describe 200 "Success"
    ),
```

```
    use python:shortme.shorten.shortened_html
);
```

Let's save the file then verify its syntax:

```
$ rb-check shortme.rbr
Syntax OK.
```

The syntax checker just controls that your file is RBR compliant, by parsing it. It's useful to catch any syntax error, like a missing comma.

Notice that the checker does not check that:

1. the code and file locations are valid
2. there are duplicate definitions

Those are checked when the application gets initialized, and will generate errors.

4.2 Step 2: the code

Let's create the code now !

Now we can add a *shorten* module in our *shortme* package and add our functions in it. We have simple functions for this application but you can use classes or whatever you want to organize your application.

RedBarrel does not impose anything here.

The `shorten()` function gets the URL to shorten in the request's POST or body, depending if it was called directly or via the HTML form:

```
from webob.exc import HTTPNotFound, HTTPSeeOther

_SHORT = {}
_DOMAIN = 'http://localhost:8000/'

def shorten(globs, request):
    if 'url' in request.POST:
        # form
        url = request.POST['url']
        redirect = True
    else:
        # direct API call
        url = request.body
        # no redirect
        redirect = False

    next = len(_SHORT)
    if url not in _SHORT:
        _SHORT[next] = url

    shorten = '%sr/%d' % (_DOMAIN, next)

    if not redirect:
        return shorten
    else:
        location = '/shortened.html?url=%s&shorten=%s' \
                % (url, shorten)
        raise HTTPSeeOther(location=location)
```

Warning: This is a toy implementation. Do not run a shortener with this code ;-)

If the call was made from the html page, the API redirects to the result HTML page, otherwise it returns the shortened URL.

Once the URL is created, `redirect()` may be called via */r/someid*:

```
_HITS = defaultdict(int)

def redirect(globs, request):
    """Redirects to the real URL"""
    path = request.path_info.split('/r/')
    if len(path) < 2 or int(path[-1]) not in _SHORT:
        raise HTTPNotFound()
    index = int(path[-1])
    location = _SHORT[index]
    _HITS[index] += 1
    raise HTTPSeeOther(location=location)
```

Every call increments a hit counter, that's used in `stats()`:

```
def stats(globs, request):
    """Returns the number of hits per redirect"""
    res = [(url, _HITS[index]) for index, url in _SHORT.items()]
    return json.dumps(dict(res))
```

Last, the two HTML pages are simple string templates:

```
def shortened_html(globs, request, url='', shorten=''):
    """HTML page with the shortened URL result"""
    tmpl = os.path.join(os.path.dirname(__file__), 'shortened.html')
    with open(tmpl) as f:
        tmpl = f.read()
    return tmpl % {'url': url, 'shorten': shorten}


def shorten_html(globs, request, url=''):
    """HTML page to create a shortened URL"""
    tmpl = os.path.join(os.path.dirname(__file__), 'shorten.html')
    with open(tmpl) as f:
        tmpl = f.read()
    return tmpl % url
```

Notice that the GET params are passed through keywords arguments.

That's it !

To run the application, just execute the RBR file with *rb-run*:

```
$ ../bin/rb-run shortme.rbr
Generating LALR tables
Initializing the globs...
Generating the Web App...
=> 'shorten' hooked for '/shorten'
=> 'shorten_html' hooked for '/shorten.html'
=> 'shortened_html' hooked for '/shortened.html'
=> 'redirect' hooked for '/r/{redirect}'
=> 'stats' hooked for '/stats'
...ready
```

```
Serving on port 8000...
```

You can visit the API documentation at */__doc__*, which is generated automatically for you.

4.3 Step 3: release & deploy

To release and deploy applications, RedBarrel uses the existing standards:

- distutils
- WSGI

If you are familiar with those, this section should not be surprising.

4.3.1 Creating releases

The wizard creates a `setup.py` file and a `setup.cfg` file, you can use to create a release with Distutils.

With Distutils1:

```
$ python setup.py sdist
```

With Distutils2:

```
$ pysetup run sdist
```

The `setup.py` file is just a wrapper around the `setup.cfg` file so distutils-based installers are made happy.

4.3.2 Running behind a Web Server

Running the application via *rb-run* is just for development and tests usage. In production, we want to use a real Web Server like Nginx or Apache.

Since RedBarrel produces a WSGI application, it's very easy to provide a script for mod_wsgi or Gunicorn or any WSGI-compatible server. There's one default `wsgiapp.py` script provided when you run the wizard, located in the package created:

```
# generated by RedBarrel's rb-quickstart
from redbarrel.wsgiapp import WebApp
application = WebApp('shortme.rbr')
```

Running it with GUnicorn is as simple as:

```
$ gunicorn shortme.wsgiapp
2011-07-15 14:50:42 [14316] [INFO] Starting gunicorn 0.11.2
2011-07-15 14:50:42 [14316] [INFO] Listening at: http://127.0.0.1:8000 (14316)
2011-07-15 14:50:42 [14319] [INFO] Booting worker with pid: 14319
Initializing the globs...
Generating the Web App...
=> 'index' hooked for '/'
=> 'shorten' hooked for '/shorten'
=> 'shorten_html' hooked for '/shorten.html'
=> 'shortened_html' hooked for '/shortened.html'
=> 'redirect' hooked for '/r/{redirect}'
=> 'stats' hooked for '/stats'
...ready
```

Congrats, you now have a RedBarrel app that scales ;)

CHAPTER

FIVE

RBR SPECIFICATION

Version 0.9

The RBR (RedBaRell) Language is a syntax used to describe web services.

It's organized into:

- a meta section containing common definitions
- globs - values that can be used everywhere
- path definition sections for every web service

5.1 General syntax

RBR is organized into definitions containing variables or operations, and global definitions.

The general syntax is:

```
global name "value";

type binary python:Binary;

meta (
    title "this",
    version 1.0,
    description "That"
);

path somepath (
    method   GET,
    url /,
    use python:somecode,
);
```

Every section or global ends with a semi-colon, and every definitin is separated by a comma.

There's a special unique **meta** section that's used to define a few metadata:

```
meta (
    title "this",
    version 1.0,
    description "That"
);
```

In sections, for some variables, you can define multiple elements:

```
path Foo (
    variable1 (
        field1 value1,
        operation,
        field2 value2
    )
);
```

Here's a full example of a valid RBR file that defines an API to shorten URLs:

```
meta (
    description """An URL Shortener""",
    version 1.0
);

path shorten (
    description "Shortens an URL",
    method POST,
    url /shorten,

    request-body (
        description "Contains the URL to shorten"
    ),
    response-body (
        description "The shortened URL"
    ),
    response-headers (
        set Content-Type "text/plain"
    ),
    response-status (
        describe 200 "Success",
        describe 400 "I don't like the URL provided"
    ),

    use python:shortme.shorten.shorten
);
```

5.2 Types

RBR recognizes these types:

- "string" – characters wrapped into ".
- """text""" – a string that may contain end of lines and ".
- status code – an integer representing a status code, like 401
- verb – an HTTP verb. Possible values: GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT, OPTIONS. You can provide several verbs, separated by "|". Example: GET|POST.
- path – a path on the server, starting with /
- code location – a fully qualified name that points to some callable.
- file location – a path on the system

5.2.1 More on paths

Paths are described using the Routes syntax (see XXX), and always start with /.

XXX more on paths.

5.2.2 More on locations

Every location is suffixed with a type. Right now RBR recognizes:

- **python:fqn** – some python callable. The callable is expressed as a fully qualified name.
- **location:path** – a path to a static file.
- **proxy:url** – an http or https uri on which the call will be proxied. note that this will only work for the **use** option of paths.

Examples:

- location:/var/www/index.html
- python:package.module.func
- proxy:http://localhost:5000

5.3 List of variables

Every path definition can contain these variables:

- **description** – description of the section, can be a string or a text [1]
- **method** – the HTTP verb(s) used for the service [1] [2]
- **url** – the path matching the service [1] [3]
- **use** – the code or file location [1] [3]
- **response-body** – the response body [3]
- **response-headers** – the response headers [2]
- **response-status** – the response status [2]
- **request-body** – the request body [2]
- **request-headers** – the request headers [2]
- **request-status** – the request status [2]

The meta section can contain:

- **version** – the version of the API [1]
- **description** – a global description [1]
- **title** – a title for the application [1]

Every section can contain extra custom fields, as long as they are suffixed by *x-* so they don't conflict with a future version of the RBR DSL. Examples: **x-author**, **x-request-max-size**, etc. The reference implementation is not interpreting those fields, but they are loaded in the AST.

[1] Single value
[2] Mandatory
[3] Multiple values in a subsection. Subsections can contain a description, some variables and some operations.

5.4 Operations

Each section can contain one or several operations. Operations can be used to:

- check the type of a value, like the request body or one of its header.
- convert a value
- set a header
- describe response status codes
- document the web services

5.4.1 Check a type

You can check a request or response header or body, using one of these expressions:

1. unless type is [not] *name type* return code
2. unless *field name* type is [not] *name type* return code
3. unless *field name* validates with *prefixed name* return code

The first form can be used to validate a body. For example, to check that the request body is json and return 400 if not, you can write:

```
request-body (
    unless type is json return 400
)
```

The second form is to be used for headers:

```
request-headers (
    unless X-Back-Off is int return 400
)
```

RBR provides a very few pre-defined types for these operations:

- **json**
- **int**
- **float**

But you can define your own types. See *Defining custom types*.

The last form can be used to call some custom function. Basic authentication example:

```
request-headers (
    unless Authorization validates with python:auth return 401
)
```

Will return a 401 unless `auth()` returns True.

5.4.2 Convert a value

You can alter the value of a header or body using *alter with code*, where `code()` is a callable that will get the value to alter, and return the result.

For example, if you want to return a compressed version of a response that contains a CSS stylesheet, you can write:

```
response-body (
    alter with python:somemodule.compress_css
)
```

Where `compress_css()` is a function that returns a compressed version of the body.

5.4.3 Set a header

You can directly set a header, using *set header value*. For instance, if you want to set the Content-Type of a response to "text/css":

```
response-headers (
    set content-type "text/css"
)
```

5.4.4 Describe a status code

describe code text will let you describe every status code for the response.

Example:

```
response-status (
    describe 200 "Success",
    describe 400 "The request is probably malformed",
    describe 401 "Authentication failure"
)
```

5.4.5 Descriptions

As explained earlier, every section and subsection in the DSL file can contain a description. Descriptions are useful to document the web services:

```
path capitalize (
    description "A web service with several post/pre processing",
    ...

    request-body (
        description "Send a string in json and the server returns it Capitalized.",
    ),

    response-body (
        description "The string, Capitalized !",
    )
);
```

5.5 Defining custom types

RBR provides a very few pre-defined types for check operations:

- **json**
- **int**
- **float**

To define a new type, you can use a **type name value** definition, where name is the **name** of the type and **value** a code location.

The code location is instanciated, then invoked everytime a type needs to be chacked. It receives the value and must return True or False.

Example:

```
type blob python:Blob;
```

Corresponding code:

```
class Blob:
    def __call__(self, value):
        return value.startswith('blob:')
```

5.6 The meta section

The meta section allows you to define a title, a description and a version for your application.

Example:

```
meta (
    title "RedBarrel Application",
    version 1.1,
    description """
    This is a RedBarrel App !
    """
);
```

5.7 Proxying

Requests to a given url can be proxied to another server.

Example:

```
path shorten (
    description "Shortens an URL",
    method POST,
    url /shorten,

    request-body (
        description "Contains the URL to shorten"
    ),
    response-body (
        description "The shortened URL"
    ),
    response-headers (
        set Content-Type "text/plain"
    ),
    response-status (
        describe 200 "Success",
        describe 400 "I don't like the URL provided"
    ),
```

```
    use proxy:http://localhost:5000
);
```

The request and response can be checked as usual, and the request is eventually proxied to **http://localhost:5000** then the response returned.

This is useful if you want to use another server to build the response for a given service.

CHAPTER

SIX

RESERVED KEYWORDS

- path
- global
- meta
- unless
- validates
- with
- not
- is
- on
- error
- return
- type
- set
- alter
- describe
- description