artichoke Documentation

Release 0.2

Alessandro Molina

Sep 27, 2017

Contents

Artichoke is a lightweight WSGI Python framework for rapid prototyping of web applications. Artichoke has been developed by AXANT with a syntax similar to the one of the Turbogears2 framework to permit to develop fast and small web applications which can be quickly switched to a full stack framework when necessary.

CHAPTER 1

Getting Started with Artichoke

Get Artichoke installed, learn how to create a new Artichoke application and explore examples.

How to install Artichoke

You can install artichoke both from an official release trough setuptools or by fetching the development version from the repositories

Installing stable version with setuptools

Artichoke releases require Python and setuptools to be installed

Python

Artichoke works with any version of python between 2.5 and 2.7. The most widely deployed version of python at the moment of this writing is version 2.5.

Setuptools

```
$ wget http://peak.telecommunity.com/dist/ez_setup.py | sudo python
```

You may also use your system's package for setuptools.

Artichoke

```
$ sudo easy_install artichoke
```

This should install the last stable release of artichoke and download all the required dependencies including Genshi, Paste and WebOb.

Upgrading Artichoke

```
$ sudo easy_install -U artichoke
```

Quickstarting with artichoke

Creating an artichoke application

Artichoke applications can be created from the *artichoke.application.Application* by passing a root *artichoke.controller.Controller* and a path where to find the templates exposed by the controller methods.

You can serve your application both using mod_wsgi or by using the internal artichoke wsgi server. The internal server will also reload application if any file of the application itself is changed, this comes at a high performance cost so it is only intended for development and not for production deploy.

```
import artichoke
from artichoke.server import serve

app = artichoke.Application(root=RootController, templates_path='views')

if __name__ == '__main__':
    serve(app)
```

Creating a controller for your application

Each artichoke application requires a root controller which will serve requests sent to the application itself. The root controller can have any number of sub controllers as instance variables to serve nested urls.

```
import artichoke
from artichoke import expose
from artichoke.server import serve

class RootController(artichoke.Controller):
    @expose(content_type='text/plain')
    def index(self, args, params):
        return 'Hi from my first artichoke application'

    @expose()
    def hello(self, args, params):
        return '<html><head></head><body>Hello World</body></html>'

serve(artichoke.Application(root=RootController, templates_path='views'))
```

Each method exposed with the *@expose* decorator will be served as an url inside the root of the application. In this case */hello* will be served by the hello method which doesn't expose any template and so it returns the html to be served as a string with the default content type (which is text/html).

An exception to this rule are the *index* and *not_found* methods. The first will be served with the last part of the url is the controller itself (for example the *index* of your root controller will serve the */* url). While the *not_found* one is called when no method or subcontroller has been found to serve the requested url. This method by default returns a 404 with a standard html but can be overridden by the user.

Apart from exposed methods the controllers will inherit some utility methods. Those include:

- **render(self, template, params)** *which will render the given template file (requires extension) from the template_path*

Serving Templates

Being able to serve content isn't really useful if you can serve only strings. For this reason the expose decorator supports declaring both a content_type and a template. The first will be useful when you need to serve JSON or files and the latter will be used frequently to serve web pages.

Controller Example

When exposing a template your method should return a dict. Each entry inside the dict will be exposed as a variable inside the template

```
import artichoke
from artichoke import request, response, expose, redirect, url, flash
from artichoke.server import serve

class RootController(artichoke.Controller):
    @expose('index')
    def index(self, args, params):
        who = params.get('who', 'World')
        return dict(who=who)

serve(artichoke.Application(root=RootController, templates_path='views'))
```

View Example

Save the following code as **index.choke** inside the *views* directory (the one passed to the Application as **templates_path** argument) and it will be served when calling the */index* url as the *@expose* decorator declared that the index method should serve the index template.

```
<html>
    <head>
        <title>Hello ${who}</title>
    </head>

    <body>
        Welcome ${who}
    </body>
</html>
```

Serving Nested Urls

Is it possible to create controllers inside controllers, this will permit to serve nested urls. To perform this just allocate more controllers inside the **__init__** of the root controller. Each controller will serve the url equal to the name of the variable it has been assigned to.

In the following example we the **/sub/hello** url will be served by the *hello* method of the *SubController* class as it has been created inside the RootController.

```
import artichoke
from artichoke import request, response, expose, redirect, url, flash
from artichoke.server import serve

class SubController(artichoke.Controller):
    @expose()
    def hello(self, args, params):
        return 'Hello World'

class RootController(artichoke.Controller):
    def __init__(self, templates_path, helpers):
        super(RootController, self).__init__(templates_path, helpers)
        self.sub = SubController(os.path.join(templates_path, 'sub'), helpers)

    @expose('index')
    def index(self, args, params):
        who = params.get('who', 'World')
        return dict(who=who)

serve(artichoke.Application(root=RootController, templates_path='views'))
```

Utility Functions

Artichoke Exposes a set of functions to help you create your application:

- **redirect(where)** which will redirect the user to another url
- **url(path, params=dict)** which will generate an url with the given parameters
- **flash('message', 'class') will inject inside the response object of the** current call (or next call after a redirect) the **flash_obj** dictionary which will expose the *msg* and *class* keys specified inside the *response.flash* call.

  As both the request and response objects are available inside the template context you can display the flash message inside the template with something like:

```
${%if response.flash_obj:}
    <div>
        <div class="${response.flash_obj['class']}">${response.flash_obj['msg']}</div>
    </div>
${%end}
```

The Request and Response objects

`artichoke.request` and `artichoke.response` objects are automatically created by artichoke itself for each request. For documentation about the request and response objects you can refer to the WebOb documentation.

The not_found method

not_found method of a controller will be called when each other url resolution method has failed to find a valid callable.

The default implementation of the method will set the *response.status* to **404**, *response.headers['Content-Type']* to **text/html** and will return a simple error message as an html page.

You can override this method to serve a different error page, implement different dispatching mechanisms or rest urls.

Advanced topics

This section will cover some advanced functions of Artichoke like helpers, authentication and form builder

Exposing Helpers inside templates

By default when you render a template from artichoke **response**, **request** and the **h** variables will be available inside the template. The last one will expose a collection of helpers useful while creating templates.

By default this collection is empty, but you can override it by passing a different object to the **config** parameter of the artichoke *Application*

```
from datetime import datetime
import artichoke
from artichoke.server import serve

class AppHelpers(object):
    def copyright(self):
        return 'Copyright 2010-%s' % datetime.now().strftime('%Y')

class RootController(artichoke.Controller):
    @expose('index')
    def index(self, args, params):
        return dict()

serve(artichoke.Application(root=RootController, templates_path='views'))
```

```
<html>
     <head>
         <title>Hello World</title>
     </head>

     <body>
         Welcome, this page is ${h.copyright()} MySelf
     </body>
 </html>
```

It might be useful inside your application to use WebHelpers to implement your helpers.

Authentication

By default the Artichoke framework will enable a simple authentication layer which will make possible to login users by saving session cookies inside their browsers.

You will have just to implement your **/login** and **/logout** methods to save and delete the credentials and permit to your users to login and logout

You can login an user by saving inside the **response.identity** variable a dictionary containing the **user** key pointing to an object exposing at least a **user_name** and **password** properties.

When the user comes back the *response.identity* object will contain the data of the user that came back. You can then logout the user by setting response.identity to **None**

```
from artichoke import redirect, response, flash

@expose()
def login(self, args, params):
    class FakeUser(object):
        pass

    user = FakeUser()
    user.user_name = params['user']
    user.password = params['password']

    response.identity = {'user':user}
    flash('Welcome back!')
    return redirect('/index')

@expose()
def logout(self, args, params):
    response.identity = None
    return redirect('/index')
```

Form Builder

The **artichoke.forms.FormBuilder** class permits to quickly create forms inside your web pages.

The first parameter of the constructor is the url where to submit the form data, the second parameter is a dictionary with the field to expose inside the form and the third and optional one is the order of the fields inside the form (omitting it will cause random order).

Each entry inside the fields dict will need a key with the same name of the parameter and a value which must be a dictionary itself. The dictionary value can specify a label and a type for the field (valid types are *textarea*, *password*, *text*, *file*). If nothing is specified it will default to a text field with a label equal to the field key capitalized.

```
new_project_form = FormBuilder('/add_project', dict(name={},
                                                    download_url={},
                                                    short_desc={'label':'Short␣
↪Description:'},
                                                    long_desc={'label':'Long␣
↪Description:',
                                                               'type':'textarea'},
                                                    icon={'type':'file'}),
                                fields_order=['name', 'download_url', 'icon', 'short_
↪desc',
                                              'long_desc'])
```

To display the form inside the template you must pass the form to the template and call the **form.render**() method

Custom Middlewares

Since version 0.3.1 Artichoke supports middlewares. Registering middlewares is quite simple, just passing a list of middleware to create to the `middlewares` configuration variable is enough.

Each middleware will receive the current application:`app`, artichoke core:`core` and configuration options:`config` at construction

You can for example create a middleware that handles database models with sqlalchemy:

```
import sqlalchemy as sqla
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import scoped_session, sessionmaker

DeclarativeBase = declarative_base()
metadata = DeclarativeBase.metadata
maker = sessionmaker(autoflush=True, autocommit=False)
DBSession = scoped_session(maker)

class SQLAMiddleware(object):
    def __init__(self, app, core, config):
        self.app = app

        self.engine = sqla.create_engine(config.get('sqlalchemy.url'), echo=False)
        self.session = config.get('sqlalchemy.session')

        metadata.create_all(self.engine)
        self.session.configure(bind=self.engine)

    def __call__(self, environ, start_response):
        self.session.begin()
        try:
            ans = self.app(environ, start_response)
            self.session.flush()
            self.session.commit()
        except:
            self.session.rollback()
            raise
        return ans

app = artichoke.Application(root=RootController, templates_path='views',
                            config={'sqlalchemy.url':'sqlite:///devdata.db',
                                    'sqlalchemy.session':DBSession,
                                    'middlewares':[SQLAMiddleware]})
```

Application Configuration

Apart from the *root* and *templates_path* parameters the **Application** class constructor accepts a thir parameter called **config**. This parameter contains a dictionary with various configuration options about the application itself:

- **helpers** (*default: an empty object*) The application helpers object
- **statics** (*default: 'public'*) The application static files path (will be available inside a controller as self.application.statics)
- **middlewares** (*default: []*) List of middlewares to allocate around the application
- **autoreload** (*default: False*) The application should disable the templates cache reloading them at each request
- **authenticator** (*default: CookieAuthenticator*) The authenticator class to be used to authenticate users
- **mail_errors_to** (*default: None*) Mail crash tracebacks to the specified address
- **mail_errors_from** (*default: 'artichoke@localhost'*) The *From* field of mailed tracebacks
- **traceback** (*default: False*) On crash print traceback inside the web browser (you should disable this on production)