Flastik – Specifications & Syntax Patterns

Table of Contents

Scope

In with words, Flastik is a tiny-framework for static website design inspired by Flask micro-framework (see http://flask.pocoo.org/docs/1.0/). It provides tools for designing simple static website project using Flask-like syntax and project architecture as well as leveraging Jinja2 templating system and Bootstrap "beautyfying" capability. Additionally, Flastik aims to easy the porting to Flask if extra functionality becomes needed further down your website life cycle. Basic knowledge of Jinja2 templating and Flask1.0 microwebframework will be assumed from here on (see http://flask.pocoo.org/docs/1.0/ and http://jinja.pocoo.org/docs/2.10/).

In addition classes and functions have been designed in order to ease the management and templating of images, downloads and other static files (see StaticFile, Image and Download classes as well as collect_static_files function).

Some basic templates are also provided, namely "base.html", "navbar.html" and "footer.html". They are located in the "base_templates" folder and will be copied at the installation location of the package. Similarly, a default Python icon is provided as "favicon.ico".

Flastik is python 3.6+ compatible only and requires the jinja2 python library to be installed.

Bootstrap 4.3.1 (see https://getbootstrap.com), JQuery 3.4.1 (see https://jquery.com) as well as Popper 1.0 (see https://popper.js.org) are provided with the package.

Flastik is distributed under a GNU General Public License v3.0 (see LICENSE.txt in package).

Installation

The installation of Flastik is standard:

- Change directory to the FlastiK code base
- Run "python setup.py install" to install the package (or "sudo python setup.py install" if root permission is required)
- Finally run "python setup.py test" to test the sanity of the package installation (or "sudo python setup.py test" if root permission is required)

Classes, Decorators, Functions and Environment Methods

Their sole purposes is to achieve near-seamless compatibility with Flask syntax and framework.

Builder Class

Designed to be used like the App Class of Flask, this class defines the overall project environment as well as provides functions, methods and decorators for templating, routing and building static websites.

Roles:

- set-up the Jinja templating environment
- create the “website-folder-architecture” and *.html files (see above)
- avoid duplicate folder and *.html by keeping track of every new instance in a dict like so {“function_name”+”var-1”+...+”var-n”: “path/filename.html”}...
- ...by leveraging the @website.route decorator (see below)

Features:

- Based on “singleton” and “factory” software patterns.

Options:

- templates: path to user’s template folder. User’s templates are added to the "base templates" provided in the package.
- use_package_templates: boolean switch. If True (default): "base templates" will be available in the template environment. If False: they won't.
- bootstrap_folder: path to bootstrap folder. If None (default): a complete distribution of Bootstrap 4.3.1 will be used and copied in static_website_root/static during built/deployment. Otherwise: specified Bootstrap distribution will be used and linked to.
- css_style_sheet: path to *.css stylesheet file. If None (default): a blank stylesheet will be provided, used and copied in static_website_root/static during built/deployment. Otherwise: the specified stylesheet will be used and linked to.
- favicon: path to web browser tab icon. if None (default): a generic python icon will be used and copied in static_website_root/static during built/deployment. Otherwise: specified will be used and copied to static.
- meta: dictionary providing meta information.
- description: web site's description (meta info.).
- author: web site's author(s) (meta info.).
- log_level: logging level. It can be ‘CRITICAL', 'ERROR', 'WARNING', 'INFO' or 'DEBUG'.

StaticFile Class

Dedicated Python class for static files, the StaticFile class keeps track of of all of its instances and takes care of copying (or making symlinks to) their sources to the static_webiste_root at the building/deployment phase.

Roles:

- move/copy static files to their destinations (“files” folder by default or anywhere else if specified) during building phase.
- Avoid duplication by keeping track of every new instance in a dict

Features:

- based on a “factory” pattern
- all generated paths are relative to static_website_root for deployment flexibility
- StaticFile instances have a “url” attribute to facilitate the templating in Jinja. This attribute returns the relative path of the static file during template rendering.

Options:

- dest: path to destination. The file will be copied to os.path.join('website_root/files/', dest).
- handle_duplicate: boolean switch. If True: the class will automatically take care of duplicated destinations. If False (default): file destination must be unique or it will raise an error.

Image Class

Dedicated Python class for image files.

Roles:

- move/copy static files to their destinations (“images” folder by default or anywhere else if specified) during building phase.
- Avoid duplication by keeping track of every new instance in a dict

Features:

- This class is a child class from StaticFile and therefore has the same functionality plus some additional like…
- ...a “html_image” providing pre-formatted html for images

Options:

- same as StaticFile class

Download Class

Dedicated Python class for downloadable files.

Roles:

- move/copy static files to their destinations (“downloads” folder by default or anywhere else if specified) during building phase.
- Avoid duplication by keeping track of every new instance in a dict

Features:

- This class is a child class from StaticFile and therefore has the same functionality plus some additional like…
- ...a “html_download” providing pre-formatted html for downloads

Options:

- same as StaticFile class

@Builder.route Decorator

Inspired by “@app.route” Flask decorator, @Builder.route decorator is an elegant way to specify which *.html file(s) will be created and where should it (they) go. Furthermore, the ability to specify the values of the route variable has been added. Similarly to Frozen-Flask, this new functionality allows the user to specify, beforehand, the client’s url requests and build a static website accordingly (see “Example” section below)

Roles:

- Defines url patterns and associated static website directory tree
- Defines html file names

Features:

- Based on a decorator pattern from https://realpython.com/primer-on-python-decorators/#more-real-world-examples
- Designed to decorate “views” only (see below)
- The pattern variables’ values can only be defined via list(s) of values or dictionary(s) of lists of values (see pattern 3).
- There are three ways (or patterns) to use this decorator:

Pattern 1 creates one *.html file:

```
@website.route('path/pattern1/')
def view_1():
   # python code
   ...
   return render_template("template_2.html", **context)
```

Pattern 2 generates len(var-1) *...* len(var-n) *html templates:

```
@website.route('another/path/pattern/<type:var1>/.../<type:varn>/', var1=[val1, …, valn], …,varn=[val1, …, valn])
def view_2(var1,…,varn):
   # Logic/Mining/etc
   ...
   context = {“templating_var_name1”: mined_value1,…,
                “templating_var_nameN”: mined_valueN}
   ...
   return render_template("template_2.html", **context)
```

Pattern 3 generates a specific number of *html templates and directory ramification depending what the user specified through its dictionary of list:

```
@website.route('/<type:var1>/<type:var2>/', var1=[val1, val2], var2={'val1': [valA,], 'val2': [valX, valY]})
def view_2(var1, var2):
   # Logic/Mining/etc
      …
   return render_template("template_2.html", **context)
```

The previous example would create 3 html files, namely “/val1/valA/index.html”, “/val2/valX/index.html” and “/val2/valY/index.html”

Builder.build Method

This method essentially builds and deploys the static website project. In sequence it makes the web site folder tree (to destination if specified), copies the Bootstrap Suite (i.e. bootstrap, java scripts, css) if needed , renders the templates and make *.html files and applies various u-masks to directories and files.

Roles:

- Handles the static website's deployment and re-built

Options:

- dest: path to deployment destination. If None (default): the web site will be built in "./build".
- views: list of views to be built. Note: All views are built by default.
- overwrite: boolean switch. If True (default): pre-existing *.html files and Bootstrap suite will be overwritten.
- static_umask: U-mask for Bootstrap suite. Default value is 0o655 (i.e. Operating-system mode bitfield).
- html_umask: U-mask for *.html files. Default value is 0o644 (i.e. Operating-system mode bitfield).
- dir_umask: U-mask for directories. Default value is 0o755 (i.e. Operating-system mode bitfield).

url_for() Environment Method

Flask-lookalike templating function, the url_for is an indexing method designed to facilitate the templating by using calls like {{ url_for('static', filename='style.css') }} or {{ url_for('view_1') }} directly in the Jinja templates.

Roles:

- Returns relative path (here equivalent to url in a static website scenario) from template being rendered to requested *.html file or static file

Features:

- It is an Jinja environment method hence it can be used inside templates as well as inside views

Options:

- **kwargs: view arguments as kwargs or 'filename' for static files. Note that the order of the **kwargs needs to match the order of the view's arguments.

render_template Function

Similar to Flask's render_template function, it fetches the requested template in the "templates" folder, passes the "context" and renders it.

Collect_static_files Function

Collects all StaticFile's (and Child classes') instances and deploy them at the web site root directory.

Options:

- static_root: path to site root directory.
- overwrite_static: boolean switch. If True (default): existing static files will be overwritten. If False: they won't.
- copy_locally: boolean switch. If True: static files will copied locally. If False (default): symlinks will be used instead.
- file_umask: u-mask for files (Operating-system mode bitfield). Default = 0o644.
- folder_umask: u-mask for static folders (Operating-system mode bitfield). Default = 0o755.

add_XXX_arguments Functions

There are three “add_XXX_arguments” functions provided in the package, namely add_Builder_arguments, add_build_arguments and add_collect_static_files_arguments. They permit to quickly set-up the command-line key-arguments of the “project_builder.py” python file (see “Example” below). They respectively enable to define, via the command line, all the options for your Builder class instance, associated “build” method and “collect_static_files” function.

Project Architecture Template

The “Project Architecture Template” is essentially a light version of the Flask project architecture plus a “project_builder.py” file usually named “__init__.py” in Flask. The “build” folder is the resulting static website once “website.build()” is ran (see “Coding Pattern” below):

Project Folder: Development phase

|_project_builder.py: python file containing views and defining routing)
|_templates: folder (optional) containing custon Jinja templates
|_style.css: CSS style sheet (optional) containing custom CSS styling code
|_.../bootstrap (folder. Optional, could be replaced by a central)
    |_bootstrap widget, java snippets

Build Folder: Post-build/Pre-deployment phase

|_build(folder, aka “static_website_root”)
|_static_website: essentially a folder-architecture containing *.html
|_static: folder (optional). Only if user did not provide an existing bootstrap distribution
    |_stylesheet.css (Copy from above)
    |_css: folder containing Bootstrap CSS
    |_ jquery: folder containing JQuery javascript
    |_ js: folder containing Bootstrap javascript
    |_ popper: folder containing Popper javascript
|_files: folder (optional) contains all the files defined via StaticFile instances
|_images: folder (optional) contains all the images defined via Image instances
|_downloads: folder (optional) contains all the download defined via Download instances

Coding Pattern/Template

Purposely inspired by standard Flask __init__.py file, here is the “project_builder.py” template:

```
from flastik import Builder, Static, render_template

if __name__ == '__main__':
   from argparse import ArgumentParser
   # Templating imports
   from flastik import Builder, render_template
   # Static files imports
   from flastik import Image, Download, collect_static_files
   # Argument parsers imports
   from flastik import (add_Builder_arguments, add_build_arguments,
                        add_collect_static_files_arguments)

   # Define Argument parser
   arg_parser = ArgumentParser()
   # - add Builder's arg
   arg_parser = add_Builder_arguments(arg_parser)
   # - add build's arg
   arg_parser = add_build_arguments(arg_parser)
   # - add collect_static_files' arg
   arg_parser = add_collect_static_files_arguments(arg_parser)

   # Parse & format command-line args.
   arglist = sys.argv[1:]
   options = vars(arg_parser.parse_args(args=arglist))

   website = Builder(**options)

   # Define views
   @website.route(‘path/pattern/’)
   def simple_view():
      return render_template("template_1.html")

   @website.route(‘other/path/pattern/’)
   def view_with_data_mining():
      # Logic/Mining/etc
      context = {}
      context[‘title’] = blabla
      context[‘body’] = blabla
      …
      im1 = Static(/path/to/image, dest=/path/to/go/otherwise/goes/to/static)
      context[‘im1’] = im1
      ...
      return render_template("template_2.html", **context)

   @website.route(‘another/path/pattern/<type:var1>/.../<type:varn>/’, var1=[val1, …, valn],
      …,varn=[val1, …, valn])
   def view_with_data_mining_and_variables(var1,…,varn):
      from random import random_function
      # Logic/Mining/etc
      context = {}
      context[‘title’] = blabla
```

```
    context['body'] = blabla
    context['value'] = random_function(var1, var2)
    …
    im = Static("/path/to/image"+ var1, dest="/path/to/go/otherwise/goes/to/static"+ var1)
    return render_template("template_3.html", **context)

@website.route('another/path/pattern/<type:var1>/<type:var2>/', var1=[valA, valB],  var2={'valA':[…], 'valB':
[...])
  def view_with_data_mining_and_variables(var1,var2):
    from random import random_function
    # Logic/Mining/etc
    context = {}
    context['title'] = blabla
    context['body'] = blabla
    context['value'] = random_function_2(var1, var2)
    …
    im = Static("/path/to/image"+ var1, dest="/path/to/go/otherwise/goes/to/static"+ var1)
    return render_template("template_4.html", **context)

  website.build(**options)
  collect_static_files(**options)
```

Example

The following example is based on the test unit provided with Flastik:

```
import os
import sys
from argparse import ArgumentParser
# Templating imports
from flastik import Builder, render_template
# Static files imports
from flastik import Image, Download, collect_static_files
# Argument parsers imports
from flastik import (add_Builder_arguments, add_build_arguments,
                add_collect_static_files_arguments)

if __name__ == '__main__':
  # Define Argument parser
  arg_parser = ArgumentParser()
  # - add Builder's arg
  arg_parser = add_Builder_arguments(arg_parser)
  # - add build's arg
  arg_parser = add_build_arguments(arg_parser)
  # - add collect_static_files' arg
  arg_parser = add_collect_static_files_arguments(arg_parser)

  # Parse commend line args.
  arglist = sys.argv[1:]
  options = vars(arg_parser.parse_args(args=arglist))

  # Website Builder
  website = Builder(**options)
```

```
# Global vars.
context = {
'project_name': 'project_name',
'navbar_links': [
   {'name': 'home', 'url': "?"},
   {'name': 'test', 'url': 'https://www.surfline.com'},
],
'footer_link': {'name': 'Flastik - Copyright 2019', 'url': 'https://www.surfline.com'},
}

img = Image("Default Icon",
         os.path.join(website.package_path, "base_templates/default_icon.png"),
         dest="test/something_else.png")

dwnld = Download("README",
            os.path.join(website.package_path, "README.pdf"))

ship_list = ["Shippy-MacShipface", "Boatty-MacBoatface"]
cruise_dict = {"Shippy-MacShipface": [1, 2], "Boatty-MacBoatface": [3,]}

@website.route("/hello_world.html")
def hello_world():
   context['img'] = img
   context['dwnld'] = dwnld
   context['title'] = "Hello World !"
   context['body_text'] = '<h2>Hello World !</h2>'
   pattern = "\n<br><a href='%s/cruise/%s/report/index.html'>%s: report for cruise %s</a>"
   for ship in ship_list:
      cruises = cruise_dict[ship]
      for cruise_id in cruises:
         context['body_text'] += pattern % (ship, cruise_id, ship, cruise_id)
   return render_template('test.html', **context)

@website.route("/<string:ship>/cruise/<int:cruise_id>/", ship=ship_list, cruise_id=cruise_dict)
def cruise_report(ship, cruise_id):
   context['dwnld'] = ""
   context['title'] = "%s: Cruise %s" % (ship, cruise_id)
   # Testing "url_for" call from view
   context['navbar_links'][0]['url'] = website.url_for('hello_world')
   context['body_text'] = '<h2>This cruise %s. Hail to the %s !</h2>' % (cruise_id, ship)
   return render_template('test.html', **context)

@website.route("/<string:ship>/cruise/<int:cruise_id>/<string:folder_name>/",
          ship=ship_list, cruise_id=cruise_dict, folder_name=['data', 'report'])
def cruise_n_data(ship, cruise_id, folder_name):
   context['dwnld'] = ""
   context['title'] = "%s - %s" % (folder_name, ship)
   # Testing "url_for" call from view
   context['navbar_links'][0]['url'] = website.url_for('hello_world')
   context['body_text'] = "<h2>Welcome to the %s folder for the %s cruise of the %s</h2>" % (
      folder_name, cruise_id, ship)
   return render_template('test.html', **context)

website.build(**options)
collect_static_files(**options)
```

Assuming that this code is copied in “test_flastik.py”, running “python test_flastik.py” would result in the creation of a ‘build’ folder containing the following:

```
build
    |_hello_world.html
    |_Shippy-MacShipface
    |   |_cruise
    |      |_1
    |      |   |_index.html
    |      |_2
    |         |_index.html
    |_Boatty-MacBoatface
    |   |_cruise
    |      |_3
    |         |_index.html
    |_ downloads
    |  |_README.pdf
    |_images
    |  |_test
    |     |_something_else.png
    |_static
       |_ favicon.ico
       |_ stylesheet.css
       |_ css...
       |_ jquery...
       |_ js...
       |_ popper...
```