LEARNING Flask

#flask

Table of Contents

About

Chapter 1: Getting started with Flask

Remarks

Flask is a Python web application micro-framework built on top of the Werkzeug WSGI library. Flask may be "micro", but it’s ready for production use on a variety of needs.

The "micro" in micro-framework means Flask aims to keep the core simple but extensible. Flask won’t make many decisions for you, such as what database to use, and the decisions that it does make are easy to change. Everything is up to you, so that Flask can be everything you need and nothing you don't.

The community supports a rich ecosystem of extensions to make your application more powerful and even easier to develop. As your project grows you are free to make the design decisions appropriate for your requirements.

Versions

Version	Code Name	Release Date
0.12	Punsch	2016-12-21
0.11	Absinthe	2016-05-29
0.10	Limoncello	2013-06-13

Examples

Installation - Stable

Use pip to install Flask in a virtualenv.

```
pip install flask
```

Step by step instructions for creating a virtualenv for your project:

```
mkdir project && cd project
python3 -m venv env
# or `virtualenv env` for Python 2
source env/bin/activate
pip install flask
```

Never use `sudo pip install` unless you understand exactly what you're doing. Keep your project in a local virtualenv, do not install to the system Python unless you are using the system package manager.

Hello World

Create `hello.py`:

```
from flask import Flask

app = Flask(__name__)


@app.route('/')
def hello():
    return 'Hello, World!'
```

Then run it with:

```
export FLASK_APP=hello.py
flask run
 * Running on http://localhost:5000/
```

Adding the code below will allow running it directly with `python hello.py`.

```
if __name__ == '__main__':
    app.run()
```

Installation - Latest

If you want to use the latest code, you can install it from the repository. While you potentially get new features and fixes, only numbered releases are officially supported.

```
pip install https://github.com/pallets/flask/tarball/master
```

Installation - Development

If you want to develop and contribute to the Flask project, clone the repository and install the code in development mode.

```
git clone ssh://github.com/pallets/flask
cd flask
python3 -m venv env
source env/bin/activate
pip install -e .
```

There are some extra dependencies and tools to be aware of as well.

sphinx

Used to build the documentation.

```
pip install sphinx
cd docs
make html
firefox _build/html/index.html
```

py.test

Used to run the test suite.

```
pip install pytest
py.test tests
```

tox

Used to run the test suite against multiple Python versions.

```
pip install tox
tox
```

Note that tox only uses interpreters that are already installed, so if you don't have Python 3.3 installed on your path, it won't be tested.

Read Getting started with Flask online: https://riptutorial.com/flask/topic/790/getting-started-with-flask

Chapter 2: Accessing request data

Introduction

When working with an web application it's sometimes important to access data included in the request, beyond the URL.

In Flask this is stored under the global **request** object, which you can access in your code via `from flask import request`.

Examples

Accessing query string

The query string is the part of a request following the URL, preceded by a `?` mark.

Example: `https://encrypted.google.com/search`**`?hl=en&q=stack%20overflow`**

For this example, we are making a simple echo webserver that echos back everything submitted to it via the `echo` field in `GET` requests.

Example: `localhost:5000/echo`**`?echo=echo+this+back+to+me`**

Flask Example:

```
from flask import Flask, request

app = Flask(import_name=__name__)

@app.route("/echo")
def echo():

    to_echo = request.args.get("echo", "")
    response = "{}".format(to_echo)

    return response

if __name__ == "__main__":
    app.run()
```

Combined form and query string

Flask also allows access to a CombinedMultiDict that gives access to both the `request.form` and `request.args` attributes under one variable.

This example pulls data from a form field `name` submitted along with the `echo` field in the query string.

Flask Example:

```
from flask import Flask, request

app = Flask(import_name=__name__)


@app.route("/echo", methods=["POST"])
def echo():

    name = request.values.get("name", "")
    to_echo = request.values.get("echo", "")

    response = "Hey there {}! You said {}".format(name, to_echo)

    return response

app.run()
```

Accessing form fields

You can access the form data submitted via a `POST` or `PUT` request in Flask via the `request.form` attribute.

```
from flask import Flask, request

app = Flask(import_name=__name__)


@app.route("/echo", methods=["POST"])
def echo():

    name = request.form.get("name", "")
    age = request.form.get("age", "")

    response = "Hey there {}! You said you are {} years old.".format(name, age)

    return response

app.run()
```

Read Accessing request data online: https://riptutorial.com/flask/topic/8622/accessing-request-data

Chapter 3: Authorization and authentication

Examples

Using flask-login extension

One of the simpler ways of implementing an authorization system is using the flask-login extension. The project's website contains a detailed and well-written quickstart, a shorter version of which is available in this example.

General idea

The extension exposes a set of functions used for:

- logging users in
- logging users out
- checking if a user is logged in or not and finding out which user is that

What it doesn't do and what you have to do on your own:

- doesn't provide a way of storing the users, for example in the database
- doesn't provide a way of checking user's credentials, for example username and password

Below there is a minimal set of steps needed to get everything working.

I would recommend to place all auth related code in a separate module or package, for example `auth.py`. That way you can create the necessary classes, objects or custom functions separately.

Create a `LoginManager`

The extension uses a `LoginManager` class which has to be registered on your `Flask` application object.

```
from flask_login import LoginManager
login_manager = LoginManager()
login_manager.init_app(app) # app is a Flask object
```

As mentioned earlier `LoginManager` can for example be a global variable in a separate file or package. Then it can be imported in the file in which the `Flask` object is created or in your application factory function and initialized.

Specify a callback used for loading users

A users will normally be loaded from a database. The callback must return an object which represents a user corresponding to the provided ID. It should return `None` if the ID is not valid.

```
@login_manager.user_loader
def load_user(user_id):
    return User.get(user_id) # Fetch the user from the database
```

This can be done directly below creating your `LoginManager`.

A class representing your user

As mentioned the `user_loader` callback has to return an object which represent a user. What does that mean exactly? That object can for example be a wrapper around user objects stored in your database or simply directly a model from your database. That object has to implement the following methods and properties. That means that if the callback returns your database model you need to ensure that the mentioned properties and methods are added to your model.

- `is_authenticated`

  This property should return `True` if the user is authenticated, i.e. they have provided valid credentials. You will want to ensure that the objects which represent your users returned by the `user_loader` callback return `True` for that method.

- `is_active`

  This property should return True if this is an active user - in addition to being authenticated, they also have activated their account, not been suspended, or any condition your application has for rejecting an account. Inactive accounts may not log in. If you don't have such a mechanism present return `True` from this method.

- `is_anonymous`

  This property should return True if this is an anonymous user. That means that your user object returned by the `user_loader` callback should return `True`.

- `get_id()`

  This method must return a unicode that uniquely identifies this user, and can be used to load the user from the `user_loader` callback. Note that this must be a unicode - if the ID is natively an int or some other type, you will need to convert it to unicode. If the `user_loader` callback returns objects from the database this method will most likely return the database ID of this particular user. The same ID should of course cause the `user_loader` callback to return the same user later on.

If you want to make things easier for yourself (**it is in fact recommended) you can inherit from `UserMixin` in the object returned by the `user_loader` callback (presumably a database model). You can see how those methods and properties are implemented by default in this mixin here.

Logging the users in

The extension leaves the validation of the username and password entered by the user to you. In fact the extension doesn't care if you use a username and password combo or other mechanism. This is an example for logging users in using username and password.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    # Here we use a class of some kind to represent and validate our
    # client-side form data. For example, WTForms is a library that will
    # handle this for us, and we use a custom LoginForm to validate.
    form = LoginForm()
    if form.validate_on_submit():
        # Login and validate the user.
        # user should be an instance of your `User` class
        login_user(user)

        flask.flash('Logged in successfully.')

        next = flask.request.args.get('next')
        # is_safe_url should check if the url is safe for redirects.
        # See http://flask.pocoo.org/snippets/62/ for an example.
        if not is_safe_url(next):
            return flask.abort(400)

        return flask.redirect(next or flask.url_for('index'))
    return flask.render_template('login.html', form=form)
```

In general logging users in is accomplished by calling login_user and passing an instance of an object representing your user mentioned earlier to it. As shown this will usually happen after retrieving the user from the database and validating his credentials, however the user object just magically appears in this example.

I have logged in a user, what now?

The object returned by the `user_loader` callback can be accessed in multiple ways.

- In templates:

  The extension automatically injects it under the name `current_user` using a template context processor. To disable that behaviour and use your custom processor set `add_context_processor=False` in your `LoginManager` constructor.

  ```
  {% if current_user.is_authenticated %}
    Hi {{ current_user.name }}!
  {% endif %}
  ```

- In Python code:

  The extension provides a request-bound object called `current_user`.

```
from flask_login import current_user

@app.route("/hello")
def hello():
    # Assuming that there is a name property on your user object
    # returned by the callback
    if current_user.is_authenticated:
        return 'Hello %s!' % current_user.name
    else:
        return 'You are not logged in!'
```

- Limiting access quickly using a decorator A `login_required` decorator can be used to limit access quickly.

```
from flask_login import login_required

@app.route("/settings")
@login_required
def settings():
    pass
```

Logging users out

Users can be logged out by calling `logout_user()`. It appears that it is safe to do so even if the user is not logged in so the `@login_required` decorator can most likely be ommited.

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(somewhere)
```

What happens if a user is not logged in and I access the `current_user` object?

By defult an AnonymousUserMixin is returned:

- `is_active` and `is_authenticated` are `False`
- `is_anonymous` is `True`
- `get_id()` returns `None`

To use a different object for anonymous users provide a callable (either a class or factory function) that creates anonymous users to your `LoginManager` with:

```
login_manager.anonymous_user = MyAnonymousUser
```

What next?

This concludes the basic introduction to the extension. To learn more about configuration and additional options it is highly recommended to read the official guide.

Timing out the login session

Its good practice to time out logged in session after specific time, you can achieve that with Flask-Login.

```
from flask import Flask, session
from datetime import timedelta
from flask_login import LoginManager, login_require, login_user, logout_user

# Create Flask application

app = Flask(__name__)

# Define Flask-login configuration

login_mgr = LoginManager(app)
login_mgr.login_view = 'login'
login_mgr.refresh_view = 'relogin'
login_mgr.needs_refresh_message = (u"Session timedout, please re-login")
login_mgr.needs_refresh_message_category = "info"


@app.before_request
def before_request():
    session.permanent = True
    app.permanent_session_lifetime = timedelta(minutes=5)
```

Default session lifetime is 31 days, user need to specify the login refresh view in case of timeout.

```
app.permanent_session_lifetime = timedelta(minutes=5)
```

Above line will force user to re-login every 5 minutes.

Read Authorization and authentication online: https://riptutorial.com/flask/topic/9053/authorization-and-authentication

Chapter 4: Blueprints

Introduction

Blueprints are a powerful concept in Flask application development that allow for flask applications to be more modular and be able to follow multiple patterns. They make administration of very large Flask applications easier and as such can be used to scale Flask applications. You can reuse Blueprint applications however you cannot run a blueprint on its own as it has to be registered on your main application.

Examples

A basic flask blueprints example

A minimal Flask application looks something like this:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def index():
    return "Hello World!"
```

A large Flask application can separate one file into multiple files by `blueprints`.

Purpose

Make it easier for others to maintain the application.

Folder Structure of Large Application

```
/app
    /templates
    /static
    /views
        __init__.py
        index.py
    app.py
```

views/index.py

```
from flask import Blueprint, render_template

index_blueprint = Blueprint('index', __name__)

@index_blueprint.route("/")
def index():
    return "Hello World!"
```

app.py

```
from flask import Flask
from views.index import index_blueprint

application = Flask(__name__)
application.register_blueprint(index_blueprint)
```

Run application

```
$ export FLASK_APP=app.py
$ flask run
```

Read Blueprints online: https://riptutorial.com/flask/topic/6427/blueprints

Chapter 5: Class-Based Views

Examples

Basic example

With Class-Based Views, we use classes instead of methods to implement our views. A simple example of using Class-Based Views looks as follows:

```
from flask import Flask
from flask.views import View

app = Flask(__name__)


class HelloWorld(View):

    def dispatch_request(self):
        return 'Hello World!'


class HelloUser(View):

    def dispatch_request(self, name):
        return 'Hello {}'.format(name)

app.add_url_rule('/hello', view_func=HelloWorld.as_view('hello_world'))
app.add_url_rule('/hello/<string:name>', view_func=HelloUser.as_view('hello_user'))

if __name__ == "__main__":
    app.run(host='0.0.0.0', debug=True)
```

Read Class-Based Views online: https://riptutorial.com/flask/topic/7494/class-based-views

Chapter 6: Custom Jinja2 Template Filters

Syntax

- {{ my_date_time|my_custom_filter }}
- {{ my_date_time|my_custom_filter(args) }}

Parameters

Parameter	Details
value	The value passed in by Jinja, to be filtered
args	Extra arguments to be passed into the filter function

Examples

Format datetime in a Jinja2 template

Filters can either be defined in a method and then added to Jinja's filters dictionary, or defined in a method decorated with `Flask.template_filter`.

Defining and registering later:

```
def format_datetime(value, format="%d %b %Y %I:%M %p"):
    """Format a date time to (Default): d Mon YYYY HH:MM P"""
    if value is None:
        return ""
    return value.strftime(format)

# Register the template filter with the Jinja Environment
app.jinja_env.filters['formatdatetime'] = format_datetime
```

Defining with decorator:

```
@app.template_filter('formatdatetime')
def format_datetime(value, format="%d %b %Y %I:%M %p"):
    """Format a date time to (Default): d Mon YYYY HH:MM P"""
    if value is None:
        return ""
    return value.strftime(format)
```

Read Custom Jinja2 Template Filters online: https://riptutorial.com/flask/topic/1465/custom-jinja2-template-filters

Chapter 7: Deploying Flask application using uWSGI web server with Nginx

Examples

Using uWSGI to run a flask application

The built-in `werkzeug` server certainly is not suitable for running production servers. The most obvious reason is the fact that the `werkzeug` server is single-threaded and thus can only handle one request at a time.

Because of this we want to use the uWSGI Server to serve our application instead. In this example we will install uWSGI and run a simple test application with it.

Installing uWSGI:

```
pip install uwsgi
```

It is as simple as that. If you are unsure about the python version your pip uses make it explicit:

```
python3 -m pip install uwsgi  # for python3
python2 -m pip install uwsgi  # for python2
```

Now let's create a simple test application:

app.py

```
from flask import Flask
from sys import version

app = Flask(__name__)

@app.route("/")
def index():
    return "Hello uWSGI from python version: <br>" + version

application = app
```

In flask the conventional name for the application is `app` but uWSGI looks for `application` by default. That's why we create an alias for our app in the last line.

Now it is time to run the app:

```
uwsgi --wsgi-file app.py --http :5000
```

You should see the message "Hello uWSGI ..." by pointing your browser to `localhost:5000`

In order not to type in the full command everytime we will create a `uwsgi.ini` file to store that configuration:

uwsgi.ini

```
[uwsgi]
http = :9090
wsgi-file = app.py
single-interpreter = true
enable-threads = true
master = true
```

The `http` and `wsgi-file` options are the same as in the manual command. But there are three more options:

- `single-interpreter`: It is recommended to turn this on because it might interfere with the next option
- `enable-threads`: This needs to be turned on if you are using additional threads in your application. We don't use them right now but now we don't have to worry about it.
- `master`: Master mode should be enable for various reasons

Now we can run the app with this command:

```
uwsgi --ini uwsgi.ini
```

Installing nginx and setting it up for uWSGI

Now we want to install nginx to serve our application.

```
sudo apt-get install nginx  # on debian/ubuntu
```

Then we create a configuration for our website

```
cd /etc/nginx/site-available  # go to the configuration for available sites
# create a file flaskconfig with your favourite editor
```

flaskconfig

```
server {
    listen 80;
    server_name localhost;

    location / {
        include uwsgi_params;
        uwsgi_pass unix:///tmp/flask.sock;
    }
}
```

This tells nginx to listen on port 80 (default for http) and serve something at the root path (`/`). There we tell nginx to simply act as a proxy and pass every request to a socket called `flask.sock` located in `/tmp/`.

Let's enable the site:

```
cd /etc/nginx/sites-enabled
sudo ln -s ../sites-available/flaskconfig .
```

You might want to remove the default configuration if it is enabled:

```
# inside /etc/sites-enabled
sudo rm default
```

Then restart nginx:

```
sudo service nginx restart
```

Point your browser to `localhost` and you will see an error: `502 Bad Gateway`.

This means that nginx is up and working but the socket is missing. So lets create that.

Go back to your `uwsgi.ini` file and open it. Then append these lines:

```
socket = /tmp/flask.sock
chmod-socket = 666
```

The first line tells uwsgi to create a socket at the given location. The socket will be used to receive requests and send back the responses. In the last line we allow other users (including nginx) to be able to read and write from that socket.

Start uwsgi again with `uwsgi --ini uwsgi.ini`. Now point your browser again to `localhost` and you will see the "Hello uWSGI" greeting again.

Note that you still can see the response on `localhost:5000` because uWSGI now serves the application via http **and** the socket. So let's disable the http option in the ini file

```
http = :5000  # <-- remove this line and restart uwsgi
```

Now the app can only be accessed from nginx (or reading that socket directly :)).

Enable streaming from flask

Flask has that feature which lets you stream data from a view by using generators.

Let's change the `app.py` file

- add `from flask import Response`
- add `from datetime import datetime`
- add `from time import sleep`

- create a new view:

```
@app.route("/time/")
def time():
    def streamer():
        while True:
            yield "<p>{}</p>".format(datetime.now())
            sleep(1)

    return Response(streamer())
```

Now open your browser at `localhost/time/`. The site will load forever because nginx waits until the response is complete. In this case the response will never be complete because it will send the current date and time forever.

To prevent nginx from waiting we need to add a new line to the configuration.

Edit `/etc/nginx/sites-available/flaskconfig`

```
server {
    listen 80;
    server_name localhost;

    location / {
        include uwsgi_params;
        uwsgi_pass unix:///tmp/flask.sock;
        uwsgi_buffering off;  # <-- this line is new
    }
}
```

The line `uwsgi_buffering off;` tells nginx not to wait until a response it complete.

Restart nginx: `sudo service nginx restart` and look at `localhost/time/` again.

Now you will see that every second a new line pops up.

Set up Flask Application, uWGSI, Nginx - Server Configurations boiler template (default, proxy and cache)

This is a porting of set up sourced from DigitalOcean's tutorial of *How To Serve Flask Applications with uWSGI and Nginx on Ubuntu 14.04*

and some useful git resources for nginx servers.

Flask Application

This tutorial assume you use Ubuntu.

1. locate `var/www/` folder.
2. Create your web app folder `mkdir myexample`
3. `cd myexample`

optional You may want to set up virtual environment for deploying web applications on production server.

```
sudo pip install virtualenv
```

to install virtual environment.

```
virtualenv myexample
```

to set up virtual environment for your app.

```
source myprojectenv/bin/activate
```

to activate your environment. Here you will install all python packages.

end optional but recommended

Set up flask and gateway uWSGI

Install flask and uSWGI gateway:

```
pip install uwsgi flask
```

Example of flask app in myexample.py:

```
from flask import Flask
application = Flask(__name__)

@application.route("/")
def hello():
    return "<h1>Hello World</h1>"

if __name__ == "__main__":
    application.run(host='0.0.0.0')
```

Create file to communicate between your web app and the web server: gateway interface [https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface]

```
nano wsgi.py
```

then import your webapp module and make it run from the gateway entry point.

```
from myexample import application

if __name__ == "__main__":
    application.run()
```

To test uWSGI:

```
uwsgi --socket 0.0.0.0:8000 --protocol=http -w wsgi
```

To configure uWSGI:

1. Create a configuration file `.ini`

   ```
   nano myexample.ini
   ```

2. Basic configuration for gateway uWSGI

```
# include header for using uwsgi
[uwsgi]
# point it to your python module wsgi.py
module = wsgi
# tell uWSGI to start a master node to serve requests
master = true
# spawn number of processes handling requests
processes = 5
# use a Unix socket to communicate with Nginx. Nginx will pass connections to uWSGI through a
socket, instead of using ports. This is preferable because Nginx and uWSGI stays on the same
machine.
socket = myexample.sock
# ensure file permission on socket to be readable and writable
chmod-socket = 660
# clean the socket when processes stop
vacuum = true
# use die-on-term to communicate with Ubuntu versions using Upstart initialisations: see:
# http://uwsgi-docs.readthedocs.io/en/latest/Upstart.html?highlight=die%20on%20term
die-on-term = true
```

optional if you are using virtual env You can `deactivate` your virtual environment.

Nginx configuration We are gonna use nginx as:

1. default server to pass request to the socket, using uwsgi protocol
2. proxy-server in front of default server
3. cache server to cache successful requests (as example, you may want to cache GET requests if your web application)

Locate your `sites-available` directory and create a configuration file for your application:

```
sudo nano /etc/nginx/sites-available/myexample
```

Add following block, in comments what it does:

```
server {


    # setting up default server listening to port 80
    listen 8000 default_server;
    server_name myexample.com; #you can also use your IP

    # specify charset encoding, optional
    charset utf-8;

    # specify root of your folder directory
    root /var/www/myexample;
```

```
    # specify locations for your web apps.
    # here using /api endpoint as example
    location /api {
        # include parameters of wsgi.py and pass them to socket
        include uwsgi_params;
        uwsgi_pass unix:/var/www/myexample/myexample.sock;
    }

}

# Here you will specify caching zones that will be used by your virtual server
# Cache will be stored in /tmp/nginx folder
# ensure nginx have permissions to write and read there!
# See also:
# http://nginx.org/en/docs/http/ngx_http_proxy_module.html

proxy_cache_path /tmp/nginx levels=1:2 keys_zone=my_zone:10m inactive=60m;
proxy_cache_key "$scheme$request_method$host$request_uri";

# set up the virtual host!
server {
    listen   80  default_server;

    # Now www.example.com will listen to port 80 and pass request to http://example.com
    server_name www.example.com;

    # Why not caching responses

    location /api {
        # set up headers for caching
        add_header X-Proxy-Cache $upstream_cache_status;

        # use zone specified above
        proxy_cache my_zone;
        proxy_cache_use_stale updating;
        proxy_cache_lock on;

        # cache all responses ?
        # proxy_cache_valid 30d;

        # better cache only 200 responses :)
        proxy_cache_valid 200 30d;

        # ignore headers to make cache expire
        proxy_ignore_headers X-Accel-Expires Expires Cache-Control;

        # pass requests to default server on port 8000
        proxy_pass http://example.com:8000/api;
    }
}
```

Finally, link the file to `sites-enabled` directory. For an explanation of available and enabled sites, see answer: [http://serverfault.com/a/527644]

```
sudo ln -s /etc/nginx/sites-available/myproject /etc/nginx/sites-enabled
```

You are done now with nginx. However, you may want to check out this very precious boiler template: [https://github.com/h5bp/server-configs-nginx]

Very useful for fine tuning.

Now test Nginx:

```
sudo nginx -t
```

Launch Nginx:

```
sudo service nginx restart
```

Automate Ubuntu to start uWSGI The last thing is to make Ubuntu start the wsgi gateway communicating with your application, otherwise you should do it manually.

1. Locate directory for initialisation scripts in Ubuntu, and create a new script:

```
sudo nano /etc/init/myexample.conf
```

2. Add following block, comments in line to explain what it does

```
# description for the purpose of this script
description "uWSGI server instance configured to serve myproject"

# Tell to start on system runtime 2, 3, 4, 5. Stop at any other level (0,1,6).
# Linux run levels: [http://www.debianadmin.com/debian-and-ubuntu-linux-run-levels.html]
start on runlevel [2345]
stop on runlevel [!2345]

# Set up permissions! "User" will be the username of your user account on ubuntu.
setuid user
# Allow www-data group to read and write from the socket file.
# www-data is normally the group Nginx and your web applications belong to.
# you may have all web application projects under /var/www/ that belongs to www-data
group
setgid www-data

# tell Ubunutu which environment to use.
# This is the path of your virtual environment: python will be in this path if you
installed virtualenv. Otherwise, use path of your python installation
env PATH=/var/www/myexample/myexample/bin
# then tell to Ubuntu to change and locate your web application directory
chdir /var/www/myexample
# finally execute initialisation script, that load your web app myexample.py
exec uwsgi --ini myexample.ini
```

Now you can activate your script: sudo start myexample

Read Deploying Flask application using uWSGI web server with Nginx online: https://riptutorial.com/flask/topic/4637/deploying-flask-application-using-uwsgi-web-server-with-nginx

Chapter 8: File Uploads

Syntax

- request.files['name'] # single required file
- request.files.get('name') # None if not posted
- request.files.getlist('name') # list of zero or more files posted
- CombinedMultiDict((request.files, request.form)) # combine form and file data

Examples

Uploading Files

HTML Form

- Use a `file` type input and the browser will provide a field that lets the user select a file to upload.
- Only forms with the `post` method can send file data.
- Make sure to set the form's `enctype=multipart/form-data` attribute. Otherwise the file's name will be sent but not the file's data.
- Use the `multiple` attribute on the input to allow selecting multiple files for the single field.

```
<form method=post enctype=multipart/form-data>
    <!-- single file for the "profile" field -->
    <input type=file name=profile>
    <!-- multiple files for the "charts" field -->
    <input type=file multiple name=charts>
    <input type=submit>
</form>
```

Python Requests

Requests is a powerful Python library for making HTTP requests. You can use it (or other tools) to post files without a browser.

- Open the files to read in binary mode.
- There are multiple data structures that `files` takes. This demonstrates a list of `(name, data)` tuples, which allows multiple files like the form above.

```
import requests

with open('profile.txt', 'rb') as f1, open('chart1.csv', 'rb') as f2, open('chart2.csv', 'rb')
as f3:
    files = [
        ('profile', f1),
        ('charts', f2),
        ('charts', f3)
    ]
    requests.post('http://localhost:5000/upload', files=files)
```

This is not meant to be an exhaustive list. For examples using your favorite tool or more complex scenarios, see the docs for that tool.

Save uploads on the server

Uploaded files are available in `request.files`, a `MultiDict` mapping field names to file objects. Use `getlist` — instead of `[]` or `get` — if multiple files were uploaded with the same field name.

```
request.files['profile']  # single file (even if multiple were sent)
request.files.getlist('charts')  # list of files (even if one was sent)
```

The objects in `request.files` have a `save` method which saves the file locally. Create a common directory to save the files to.

The `filename` attribute is the name the file was uploaded with. This can be set arbitrarily by the client, so pass it through the `secure_filename` method to generate a valid and safe name to save as. This doesn't ensure that the name is *unique*, so existing files will be overwritten unless you do extra work to detect that.

```
import os
from flask import render_template, request, redirect, url_for
from werkzeug import secure_filename

# Create a directory in a known location to save files to.
uploads_dir = os.path.join(app.instance_path, 'uploads')
os.makedirs(uploads_dir, exists_ok=True)

@app.route('/upload', methods=['GET', 'POST'])
def upload():
    if request.method == 'POST':
        # save the single "profile" file
        profile = request.files['profile']
        profile.save(os.path.join(uploads_dir, secure_filename(profile.filename)))

        # save each "charts" file
        for file in request.files.getlist('charts'):
            file.save(os.path.join(uploads_dir, secure_filename(file.name)))

        return redirect(url_for('upload'))

    return render_template('upload.html')
```

Passing data to WTForms and Flask-WTF

WTForms provides a `FileField` to render a file type input. It doesn't do anything special with the uploaded data. However, since Flask splits the form data (`request.form`) and the file data (`request.files`), you need to make sure to pass the correct data when creating the form. You can use a `CombinedMultiDict` to combine the two into a single structure that WTForms understands.

```
form = ProfileForm(CombinedMultiDict((request.files, request.form)))
```

If you're using Flask-WTF, an extension to integrate Flask and WTForms, passing the correct data will be handled for you automatically.

Due to a bug in WTForms, only one file will be present for each field, even if multiple were uploaded. See this issue for more details. It will be fixed in 3.0.

PARSE CSV FILE UPLOAD AS LIST OF DICTIONARIES IN FLASK WITHOUT SAVING

Developers often need to design web sites that allow users to upload a CSV file. Usually there is **no reason** to **save** the actual CSV file since the data will processed and/or stored in a database once uploaded. However, many if not most, PYTHON methods of parsing CSV data requires the data to be read in as a file. This may present a bit of a headache if you are using **FLASK** for web development.

Suppose our CSV has a header row and looks like the following:

```
h1,h2,h3
'yellow','orange','blue'
'green','white','black'
'orange','pink','purple'
```

Now, suppose the html form to upload a file is as follows:

```
<form action="upload.html" method="post" enctype="multipart/form-data">
    <input type="file" name="fileupload" id="fileToUpload">
    <input type="submit" value="Upload File" name="submit">
</form>
```

Since no one wants to reinvent the wheel you decide to **IMPORT csv** into your **FLASK** script. There is no guarantee that people will upload the csv file with the columns in the correct order. If the csv file has a header row, then with the help of the **csv.DictReader** method you can read the CSV file as a list of dictionaries, keyed by the entries in the header row. However, **csv.DictReader** needs a file and does not directly accept strings. You may think you need to use **FLASK** methods to first save the uploaded file, get the new file name and location, open it using **csv.DictReader**, and then delete the file. Seems like a bit of a waste.

Luckily, we can get the file contents as a string and then split the string up by terminated lines. The csv method **csv.DictReader** will accept this as a substitute to a file. The following code demonstrates how this can be accomplished without temporarily saving the file.

```
@application.route('upload.html',methods = ['POST'])
def upload_route_summary():
    if request.method == 'POST':
        # Create variable for uploaded file
        f = request.files['fileupload']

        #store the file contents as a string
        fstring = f.read()

        #create list of dictionaries keyed by header row
        csv_dicts = [{k: v for k, v in row.items()} for row in
csv.DictReader(fstring.splitlines(), skipinitialspace=True)]

        #do something list of dictionaries
    return "success"
```

The variable **csv_dicts** is now the following list of dictionaries:

```
   csv_dicts =
    [
        {'h1':'yellow','h2':'orange','h3':'blue'},
        {'h1':'green','h2':'white','h3':'black'},
        {'h1':'orange','h2':'pink','h3':'purple'}
    ]
```

In case you are new to PYTHON, you can access data like the following:

```
csv_dicts[1]['h2'] = 'white'
csv_dicts[0]['h3'] = 'blue'
```

Other solutions involve importing the **io** module and use the **io.Stream** method. I feel that this is a more straightforward approach. I believe the code is a little easier to follow than using the **io** method. This approach is specific to the example of parsing an uploaded CSV file.

Read File Uploads online: https://riptutorial.com/flask/topic/5459/file-uploads

Chapter 9: Flask on Apache with mod_wsgi

Examples

WSGI Application wrapper

Many Flask applications are developed in a *virtualenv* to keep dependencies for each application separate from the system-wide Python installation. Make sure that *mod-wsgi* is installed in your *virtualenv*:

```
pip install mod-wsgi
```

Then create a wsgi wrapper for your Flask application. Usually it's kept in the root directory of your application.

my-application.wsgi

```
activate_this = '/path/to/my-application/venv/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
import sys
sys.path.insert(0, '/path/to/my-application')

from app import app as application
```

This wrapper activates the virtual environment and all of its installed modules and dependencies when run from Apache, and makes sure the application path is first in the search paths. By convention, WSGI application objects are called `application`.

Apache sites-enabled configuration for WSGI

The advantage of using Apache over the builtin werkzeug server is that Apache is multi-threaded, meaning that multiple connections to the application can be made simultaneously. This is especially useful in applications that make use of *XmlHttpRequest* (AJAX) on the front-end.

/etc/apache2/sites-available/050-my-application.conf (or default apache configuration if not hosted on a shared webserver)

```
<VirtualHost *:80>
        ServerName my-application.org

        ServerAdmin admin@my-application.org

        # Must be set, but can be anything unless you want to serve static files
        DocumentRoot /var/www/html

        # Logs for your application will go to the directory as specified:

        ErrorLog ${APACHE_LOG_DIR}/error.log
        CustomLog ${APACHE_LOG_DIR}/access.log combined
        # WSGI applications run as a daemon process, and need a specified user, group
        # and an allocated number of thread workers. This will determine the number
        # of simultaneous connections available.
        WSGIDaemonProcess my-application user=username group=username threads=12

        # The WSGIScriptAlias should redirect / to your application wrapper:
        WSGIScriptAlias / /path/to/my-application/my-application.wsgi
        # and set up Directory access permissions for the application:
        <Directory /path/to/my-application>
                WSGIProcessGroup my-application
                WSGIApplicationGroup %{GLOBAL}

                AllowOverride none
                Require all granted
        </Directory>
</VirtualHost>
```

Read Flask on Apache with mod_wsgi online: https://riptutorial.com/flask/topic/6851/flask-on-apache-with-mod-wsgi

Chapter 10: Flask-SQLAlchemy

Introduction

Flask-SQLAlchemy is a Flask extension that adds support for the popular Python object relational mapper(ORM) SQLAlchemy to Flask applications. It aims to simplify SQLAlchemy with Flask by providing some default implementations to common tasks.

Examples

Installation and Initial Example

Installation

```
pip install Flask-SQLAlchemy
```

Simple Model

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80))
    email = db.Column(db.String(120), unique=True)
```

The code example above shows a simple Flask-SQLAlchemy model, we can add an optional tablename to the model declaration however it is often not necessary as Flask-SQLAlchemy will automatically use the class name as the table name during database creation.

Our class will inherit from the baseclass Model which is a configured declarative base hence there is no need for us to explicitly define the base as we would when using SQLAlchemy.

Reference

- Pypi URL: [https://pypi.python.org/pypi/Flask-SQLAlchemy][1]
- Documentation URL: [http://flask-sqlalchemy.pocoo.org/2.1/][1]

Relationships: One to Many

```
    class User(db.Model):
        id = db.Column(db.Integer, primary_key=True)
        name = db.Column(db.String(80))
        email = db.Column(db.String(120), unique=True)
        posts = db.relationship('Post', backref='user')


    class Post(db.Model):
        id = db.Column(db.Integer, primary_key=True)
        content = db.Column(db.Text)
        user_id = db.Column(db.Integer, db.ForeignKey('user.id')
```

In this example we have two class the User class and the Post class, the User class will be our parent and the Post will be our post as only post can belong to one user but one user can have multiple posts. In order to achieve that we place a Foreign key on the child referencing the parent that is from our example we place a foreign key on Post class to reference the User class. We then use `relationship()` on the parent which we access via our SQLAlchemy object `db`. That then allows us to reference a collection of items represented by the Post class which is our child.

To create a bidirectional relationship we use`backref`, this will allow the child to reference the parent.

Read Flask-SQLAlchemy online: https://riptutorial.com/flask/topic/10577/flask-sqlalchemy

Chapter 11: Flask-WTF

Introduction

It is a simple integration of Flask and WTForms. It allows for the easier creation and management of web forms, it automatically generates a CRSF token hidden field in your templates. It also features easy form validation functions

Examples

A simple Form

```
from flask_wtf import FlaskForm
from wtforms import StringField, IntegerField
from wtforms.validators import DataRequired

class MyForm(FlaskForm):
    name = StringField('name', validators=[DataRequired()])
    age = InterField('age', validators=[DataRequired()])
```

To render the template you will use something like this:

```
<form method="POST" action="/">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(size=20) }}
    <br/>
    {{ form.age.label }} {{ form.age(size=3) }}
    <input type="submit" value="Go">
</form>
```

The above simple code will generate our very simple flask-wtf web form with a hidden CRSF token field.

Read Flask-WTF online: https://riptutorial.com/flask/topic/10579/flask-wtf

Chapter 12: Message Flashing

Introduction

Flashing message to the template by `flash()` function.

Syntax

- flash(message, category='message')
- flash('hello, world!')
- flash('This is a warning message', 'warning')

Parameters

message	the message to be flashed.
category	the message's category, the default is `message`.

Remarks

- Template Inheritance
- API

Examples

Simple Message Flashing

Set `SECKET_KEY`, then flashing message in view function:

```
from flask import Flask, flash, render_template

app = Flask(__name__)
app.secret_key = 'some_secret'

@app.route('/')
def index():
    flash('Hello, I'm a message.')
    return render_template('index.html')
```

Then render the messages in `layout.html` (which the `index.html` extended from):

```
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
```

```
      <li>{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
{% block body %}{% endblock %}
```

Flashing With Categories

Set second argument when use `flash()` in view function:

```
flash('Something was wrong!', 'error')
```

In the template, set `with_categories=true` in `get_flashed_messages()`, then you get a list of tuples in the form of `(message, category)`, so you can use category as a HTML class.

```
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    <ul class=flashes>
    {% for category, message in messages %}
      <li class="{{ category }}">{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

Read Message Flashing online: https://riptutorial.com/flask/topic/10756/message-flashing

Chapter 13: Pagination

Examples

Pagination Route Example with flask-sqlalchemy Paginate

In this example we use a parameter in the route to specify the page number. We set a default of 1 in the function parameter `page=1`. We have a `User` object in the database and we query it, ordering in descending order, showing latest users first. We then use the `paginate` method of the `query` object in flask-sqlalchemy. We then pass this to `render_template` to be rendered.

```
@app.route('/users')
@app.route('/users/page/<int:page>')
def all_users(page=1):
    try:
        users_list = User.query.order_by(
            User.id.desc()
        ).paginate(page, per_page=USERS_PER_PAGE)
    except OperationalError:
        flash("No users in the database.")
        users_list = None

    return render_template(
        'users.html',
        users_list=users_list,
        form=form
    )
```

Rendering pagination in Jinja

Here we use the object that we passed to `render_template` to display the pages, the current active page, and also a previous and next buttons if you can go to the previous/next page.

```
<!-- previous page -->
{% if users_list.has_prev %}
<li>
    <a href="{{ url_for('users', page=users_list.prev_num) }}">Previous</a>
</li>
{% endif %}

<!-- all page numbers -->
{% for page_num in users_list.iter_pages() %}
    {% if page_num %}
        {% if page_num != users_list.page %}
            <li>
                <a href="{{ url_for('users', page=page_num) }}">{{ page_num }}</a>
            </li>
        {% else %}
            <li class="active">
                <a href="#">{{ page_num }}</a>
            </li>
        {% endif %}
    {% else %}
```

```
        <li>
            <span class="ellipsis" style="white-space; nowrap; overflow: hidden; text-overflow:
ellipsis">…</span>
        </li>
    {% endif %}
{% endfor %}

<!-- next page -->
{% if users_list.has_next %}
<li>
    <a href="{{ url_for('users', page=users_list.next_num) }}">Next</a></li>
{% endif %}
{% endif %}
```

Read Pagination online: https://riptutorial.com/flask/topic/6460/pagination

Chapter 14: Redirect

Syntax

- redirect(location, code, Response)

Parameters

Parameter	Details
location	The location the response should redirect to.
code	(Optional) The redirect status code, 302 by default. Supported codes are 301, 302, 303, 305, and 307.
Response	(Optional) A Response class to use when instantiating a response. The default is werkzeug.wrappers.Response if unspecified.

Remarks

The location parameter must be a URL. It can be input raw, such as 'http://www.webpage.com' or it can be built with the url_for() function.

Examples

Simple example

```
from flask import Flask, render_template, redirect, url_for

app = Flask(__name__)

@app.route('/')
def main_page():
    return render_template('main.html')

@app.route('/main')
def go_to_main():
    return redirect(url_for('main_page'))
```

Passing along data

```
# ...
# same as above

@app.route('/welcome/<name>')
def welcome(name):
    return render_template('main.html', name=name)

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # ...
        # check for valid login, assign username
        if valid:
            return redirect(url_for('main_page', name=username))
        else:
            return redirect(url_for('login_error'))
    else:
        return render_template('login.html')
```

Read Redirect online: https://riptutorial.com/flask/topic/6856/redirect

Chapter 15: Rendering Templates

Syntax

- `render_template(template_name_or_list, **context)`

Examples

render_template Usage

Flask lets you use templates for dynamic web page content. An example project structure for using templates is as follows:

```
myproject/
    /app/
        /templates/
            /index.html
        /views.py
```

`views.py`:

```
from flask import Flask, render_template


app = Flask(__name__)

@app.route("/")
def index():
    pagetitle = "HomePage"
    return render_template("index.html",
                            mytitle=pagetitle,
                            mycontent="Hello World")
```

Note that you can pass dynamic content from your route handler to the template by appending key/value pairs to the render_templates function. In the above example, the "pagetitle" and "mycontent" variables will be passed to the template for inclusion in the rendered page. Include these variables in the template by enclosing them in double braces: `{{mytitle}}`

`index.html`:

```
<html>
    <head>
        <title>{{ mytitle }}</title>
    </head>
    <body>
        <p>{{ mycontent }}</p>
    </body>
</html>
```

When executed same as the first example, `http://localhost:5000/` will have the title "HomePage" and a paragraph with the content "Hello World".

Read Rendering Templates online: https://riptutorial.com/flask/topic/1641/rendering-templates

Chapter 16: Routing

Examples

Basic Routes

Routes in Flask can be defined using the `route` decorator of the Flask application instance:

```
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello Flask'
```

The `route` decorator takes a string which is the URL to match. When a request for a URL that matches this string is received by the application, the function decorated (also called a *view function*) will be invoked. So for an about route we would have:

```
@app.route('/about')
def about():
    return 'About page'
```

It's important to note that these routes are **not** regular expressions like they are in Django.

You can also define *variable rules* to extract URL segment values into variables:

```
@app.route('/blog/posts/<post_id>')
def get_blog_post(post_id):
    # look up the blog post with id post_id
    # return some kind of HTML
```

Here the variable rule is in the last segment of the URL. Whatever value is in the last segment of the URL will be passed to the view function (`get_blog_post`) as the `post_id` parameter. So a request to `/blog/posts/42` will retrieve (or attempt to retrieve) the blog post with an id of 42.

It is also common to reuse URLs. For example maybe we want to have `/blog/posts` return a list of all blog posts. So we could have two routes for the same view function:

```
@app.route('/blog/posts')
@app.route('/blog/posts/<post_id>')
def get_blog_post(post_id=None):
    # get the post or list of posts
```

Note here that we also have to supply the default value of `None` for the `post_id` in `get_blog_post`. When the first route is matched, there will be no value to pass to the view function.

Also note that by default the type of a variable rule is a string. However, you can specify several different types such as `int` and `float` by prefixing the variable:

```
@app.route('/blog/post/<int:post_id>')
```

Flask's built-in URL-converters are:

```
string | Accepts any text without a slash (the default).
int    | Accepts integers.
float  | Like int but for floating point values.
path   | Like string but accepts slashes.
any    | Matches one of the items provided
uuid   | Accepts UUID strings
```

Should we try to visit the URL `/blog/post/foo` with a value in the last URL segment that cannot be converted to an integer, the application would return a 404 error. This is the correct action because there is not a rule with `/blog/post` and a string in the last segment.

Finally, routes can be configured to accept HTTP methods as well. The `route` decorator takes a `methods` keyword argument which is a list of string representing the acceptable HTTP methods for this route. As you might have assumed, the default is `GET` only. If we had a form to add a new blog post and wanted to return the HTML for the `GET` request and parse the form data for the `POST` request, the route would look something like this:

```
@app.route('/blog/new', methods=['GET', 'POST'])
def new_post():
    if request.method == 'GET':
        # return the form
    elif request.method == 'POST':
        # get the data from the form values
```

The `request` is found in the `flask` package. Note that when using the `methods` keyword argument, we must be explicit about the HTTP methods to accept. If we had listed only `POST`, the route would no longer respond to `GET` requests and return a 405 error.

Catch-all route

It may be useful to have one catch-all view where you handle complex logic yourself based on the path. This example uses two rules: The first rule specifically catches `/` and the second rule catches arbitrary paths with the built-in `path` converter. The `path` converter matches any string (including slashes) See Flask Variable-Rules

```
@app.route('/', defaults={'u_path': ''})
@app.route('/<path:u_path>')
def catch_all(u_path):
    print(repr(u_path))
    ...
```

```
c = app.test_client()
c.get('/')  # u_path = ''
c.get('/hello')  # u_path = 'hello'
c.get('/hello/stack/overflow/')  # u_path = 'hello/stack/overflow/'
```

Routing and HTTP methods

By default, routes only respond to `GET` requests. You can change this behavior by supplying the `methods` argument to the `route()` decorator.

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

You can also map different functions to the same endpoint based on the HTTP method used.

```
@app.route('/endpoint', methods=['GET'])
def get_endpoint():
    #respond to GET requests for '/endpoint'


@app.route('/endpoint', methods=['POST', 'PUT', 'DELETE'])
def post_or_put():
    #respond to POST, PUT, or DELETE requests for '/endpoint'
```

Read Routing online: https://riptutorial.com/flask/topic/2415/routing

Chapter 17: Sessions

Remarks

Sessions are derived from dictionaries which means they will work with most common dictionary methods.

Examples

Using the sessions object within a view

First, ensure you have imported sessions from flask

```
from flask import session
```

To use session, a Flask application needs a defined **SECRET_KEY**.

```
app = Flask(__name__)
app.secret_key = 'app secret key'
```

Sessions are implemented by default using a **cookie** signed with the secret key. This ensures that the data is not modified except by your application, so make sure to pick a secure one! A browser will send the cookie back to your application along with each request, enabling the persistence of data across requests.

To use a session you just reference the object (It will behave like a dictionary)

```
@app.route('/')
def index():
    if 'counter' in session:
        session['counter'] += 1
    else:
        session['counter'] = 1
    return 'Counter: '+str(session['counter'])
```

To release a session variable use **pop()** method.

```
session.pop('counter', None)
```

Example Code:

```
from flask import Flask, session

app = Flask(__name__)
app.secret_key = 'app secret key'

@app.route('/')
```

```
def index():
    if 'counter' in session:
        session['counter'] += 1
    else:
        session['counter'] = 1
    return 'Counter: '+str(session['counter'])

if __name__ == '__main__':
    app.debug = True
    app.run()
```

Read Sessions online: https://riptutorial.com/flask/topic/2748/sessions

Chapter 18: Signals

Remarks

Flask supports signals using Blinker. Signal support is optional; they will only be enabled if Blinker is installed.

```
pip install blinker
```

http://flask.pocoo.org/docs/dev/signals/

Signals are not asynchronous. When a signal is sent, it immediately executes each of the connected functions sequentially.

Examples

Connecting to signals

Use a signal's `connect` method to connect a function to a signal. When a signal is sent, each connected function is called with the sender and any named arguments the signal provides.

```
from flask import template_rendered

def log_template(sender, template, context, **kwargs):
    sender.logger.info(
        'Rendered template %(template)r with context %(context)r.',
        template=template, context=context
    )

template_rendered.connect(log_template)
```

See the documentation on built-in signals for information about what arguments they provides. A useful pattern is adding a `**kwargs` argument to catch any unexpected arguments.

Custom signals

If you want to create and send signals in your own code (for example, if you are writing an extension), create a new `Signal` instance and call `send` when the subscribers should be notified. Signals are created using a `Namespace`.

```
from flask import current_app
from flask.signals import Namespace

namespace = Namespace()
message_sent = namespace.signal('mail_sent')

def message_response(recipient, body):
    ...
    message_sent.send(
        current_app._get_current_object(),
        recipient=recipient,
        body=body
    )

@message_sent.connect
def log_message(app, recipient, body):
    ...
```

Prefer using Flask's signal support over using Blinker directly. It wraps the library so that signals remain optional if developers using your extension have not opted to install Blinker.

Read Signals online: https://riptutorial.com/flask/topic/2331/signals

Chapter 19: Static Files

Examples

Using Static Files

Web applications often require static files like CSS or JavaScript files. To use static files in a Flask application, create a folder called `static` in your package or next to your module and it will be available at `/static` on the application.

An example project structure for using templates is as follows:

```
MyApplication/
    /static/
        /style.css
        /script.js
    /templates/
        /index.html
    /app.py
```

app.py is a basic example of Flask with template rendering.

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')
```

To use the static CSS and JavaScript file in the template index.html, we need to use the special 'static' endpoint name:

```
{{url_for('static', filename = 'style.css')}}
```

So, **index.html** may contain:

```
<html>
    <head>
        <title>Static File</title>
        <link href="{{url_for('static', filename = 'style.css')}}" rel="stylesheet">
        <script src="{{url_for('static', filename = 'script.js')}}"></script>
    </head>
    <body>
        <h3>Hello World!</h3>
    </body>
</html>
```

After running app.py we will see the webpage in http://localhost:5000/.

Static Files in Production (served by frontend webserver)

Flask's built-in webserver is able to serve static assets, and this works fine for development. However, for production deployments that are using something like uWSGI or Gunicorn to serve the Flask application, the task of serving static files is one that is typically offloaded to the frontend webserver (Nginx, Apache, etc.). This is a small/easy task with smaller apps, especially when all of the static assets are in one folder; for larger apps though, and/or ones that are using Flask plugin(s) that provide static assets, then it can become difficult to remember the locations of all of those files, and to manually copy/collect them into one directory. This document shows how to use the Flask-Collect plugin to simplify that task.

Note that the focus of this documentation is on the collection of static assets. To illustrate that functionality, this example uses the Flask-Bootstrap plugin, which is one that provides static assets. It also uses the Flask-Script plugin, which is used to simplify the process of creating command-line tasks. Neither of these plugins are critical to this document, they are just in use here to demonstrate the functionality. If you choose not to use Flask-Script, you will want to review the Flask-Collect docs for alternate ways to call the `collect` command.

Also note that configuration of your frontend webserver to serve these static assets is outside of the scope of this doc, you'll want to check out some examples using Nginx and Apache for more info. Suffice it to say that you'll be aliasing URLs that start with "/static" to the centralized directory that Flask-Collect will create for you in this example.

The app is structured as follows:

```
/manage.py - The app management script, used to run the app, and to collect static assets
/app/ - this folder contains the files that are specific to our app
    | - __init__.py - Contains the create_app function
    | - static/ - this folder contains the static files for our app.
        | css/styles.css - custom styles for our app (we will leave this file empty)
        | js/main.js - custom js for our app (we will leave this file empty)
    | - templates/index.html - a simple page that extends the Flask-Bootstrap template
```

1. First, create your virtual environment and install the required packages: (your-virtualenv) $ pip install flask flask-script flask-bootstrap flask-collect

2. Establish the file structure described above:

   $ touch manage.py; mkdir -p app/{static/{css,js},templates}; touch app/{**init**
   .py,static/{css/styles.css,js/main.js}}

3. Establish the contents for the `manage.py`, `app/__init__.py`, and `app/templates/index.html` files:

```
# manage.py
#!/usr/bin/env python
import os
from flask_script import Manager, Server
from flask import current_app
from flask_collect import Collect
from app import create_app
```

```
class Config(object):
   # CRITICAL CONFIG VALUE: This tells Flask-Collect where to put our static files!
   # Standard practice is to use a folder named "static" that resides in the top-level of the
project directory.
   # You are not bound to this location, however; you may use basically any directory that you
wish.
   COLLECT_STATIC_ROOT = os.path.dirname(__file__) + '/static'
   COLLECT_STORAGE = 'flask_collect.storage.file'

app = create_app(Config)

manager = Manager(app)
manager.add_command('runserver', Server(host='127.0.0.1', port=5000))

collect = Collect()
collect.init_app(app)

@manager.command
def collect():
  """Collect static from blueprints. Workaround for issue: https://github.com/klen/Flask-
Collect/issues/22"""
  return current_app.extensions['collect'].collect()

if __name__ == "__main__":
   manager.run()
```

```
# app/__init__.py
from flask import Flask, render_template
from flask_collect import Collect
from flask_bootstrap import Bootstrap

def create_app(config):
  app = Flask(__name__)
  app.config.from_object(config)

  Bootstrap(app)
  Collect(app)

  @app.route('/')
  def home():
    return render_template('index.html')

  return app
```

```
# app/templates/index.html
{% extends "bootstrap/base.html" %}
{% block title %}This is an example page{% endblock %}

{% block navbar %}
<div class="navbar navbar-fixed-top">
  <!-- ... -->
</div>
{% endblock %}

{% block content %}
  <h1>Hello, Bootstrap</h1>
{% endblock %}
```

4. With those files in place, you can now use the management script to run the app:

```
$ ./manage.py runserver # visit http://localhost:5000 to verify that the app works correctly.
```

5. Now, to collect your static assets for the first time. Before doing this, it's worth noting again that you should NOT have a `static/` folder in the top-level of your app; this is where Flask-Collect is going to place all of the static files that it's going to be collecting from your app and the various plugins you might be using. If you *do* have a `static/` folder in the top level of your app, you should delete it entirely before proceeding, as starting with a clean slate is a critical part of witnessing/understanding what Flask-Collect does. Note that this instruction isn't applicable for day-to-day usage, it is simply to illustrate the fact that Flask-Collect is going to create this directory for you, and then it's going to place a bunch of files in there.

With that said, you can run the following command to collect your static assets:

```
$ ./manage.py collect
```

After doing so, you should see that Flask-Collect has created this top-level `static/` folder, and it contains the following files:

```
$ find ./static -type f # execute this from the top-level directory of your app, same dir that
contains the manage.py script
static/bootstrap/css/bootstrap-theme.css
static/bootstrap/css/bootstrap-theme.css.map
static/bootstrap/css/bootstrap-theme.min.css
static/bootstrap/css/bootstrap.css
static/bootstrap/css/bootstrap.css.map
static/bootstrap/css/bootstrap.min.css
static/bootstrap/fonts/glyphicons-halflings-regular.eot
static/bootstrap/fonts/glyphicons-halflings-regular.svg
static/bootstrap/fonts/glyphicons-halflings-regular.ttf
static/bootstrap/fonts/glyphicons-halflings-regular.woff
static/bootstrap/fonts/glyphicons-halflings-regular.woff2
static/bootstrap/jquery.js
static/bootstrap/jquery.min.js
static/bootstrap/jquery.min.map
static/bootstrap/js/bootstrap.js
static/bootstrap/js/bootstrap.min.js
static/bootstrap/js/npm.js
static/css/styles.css
static/js/main.js
```

And that's it: use the `collect` command whenever you make edits to your app's CSS or JavaScript, or when you've updated a Flask plugin that provides static assets (like Flask-Bootstrap in this example).

Read Static Files online: https://riptutorial.com/flask/topic/3678/static-files

Chapter 20: Testing

Examples

Testing our Hello World app

Introduction

In this minimalist example, using `pytest` we're going to test that indeed our Hello World app does return "Hello, World!" with an HTTP OK status code of 200, when hit with a GET request on the URL `/`

First let's install `pytest` into our virtualenv

```
pip install pytest
```

And just for reference, this our hello world app:

```
# hello.py
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, World!'
```

Defining the test

Along side our `hello.py`, we define a test module called `test_hello.py` that is going to be discovered by `py.test`

```
# test_hello.py
from hello import app

def test_hello():
    response = app.test_client().get('/')

    assert response.status_code == 200
    assert response.data == b'Hello, World!'
```

Just to review, at this point our project structure obtained with the `tree` command is:

```
.
├── hello.py
└── test_hello.py
```

Running the test

Now we can run this test with the `py.test` command that will automatically discover our `test_hello.py` and the test function inside it

```
$ py.test
```

You should see some output and an indication that 1 test has passed, e.g.

```
=== test session starts ===
collected 1 items
test_hello.py .
=== 1 passed in 0.13 seconds ===
```

Testing a JSON API implemented in Flask

This example assumes you know how to test a Flask app using pytest

Below is an API that takes a JSON input with integer values `a` and `b` e.g. `{"a": 1, "b": 2}`, adds them up and returns sum `a + b` in a JSON response e.g. `{"sum": 3}`.

```
# hello_add.py
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/add', methods=['POST'])
def add():
    data = request.get_json()
    return jsonify({'sum': data['a'] + data['b']})
```

Testing this API with `pytest`

We can test it with `pytest`

```
# test_hello_add.py
from hello_add import app
from flask import json

def test_add():
    response = app.test_client().post(
        '/add',
        data=json.dumps({'a': 1, 'b': 2}),
        content_type='application/json',
    )

    data = json.loads(response.get_data(as_text=True))

    assert response.status_code == 200
    assert data['sum'] == 3
```

Now run the test with `py.test` command.

Accessing and manipulating session variables in your tests using Flask-Testing

Most of the web applications use the session object to store some important information. This examples show how you can test such application using Flask-Testing. Full working example is also available on github.

So first install Flask-Testing in your virtualenv

```
pip install flask_testing
```

To be able to use the session object you have to set the secret key

```
app.secret_key = 'my-seCret_KEy'
```

Let's imagine you have in your application function that need to store some data in session variables like this

```
@app.route('/getSessionVar', methods=['GET', 'POST'])
def getSessionVariable():
  if 'GET' == request.method:
    session['sessionVar'] = 'hello'
  elif 'POST' == request.method:
    session['sessionVar'] = 'hi'
  else:
    session['sessionVar'] = 'error'

  return 'ok'
```

To test this function you can import flask_testing and let your test class inherit flask_testing.TestCase. Import also all the necessary libraries

```
import flask
import unittest
import flask_testing
from myapp.run import app

class TestMyApp(flask_testing.TestCase):
```

Very important before you start testing is to implement the function **create_app** otherwise there will be exception.

```
  def create_app(self):
    return app
```

To test your application is working as wanted you have a couple of possibilities. If you want to just assure your function is setting particular values to a session variable you can just keep the context around and access **flask.session**

```
def testSession1(self):
    with app.test_client() as lTestClient:
      lResp= lTestClient.get('/getSessionVar')
      self.assertEqual(lResp.status_code, 200)
      self.assertEqual(flask.session['sessionVar'], 'hello')
```

One more useful trick is to differentiate between *GET* and *POST* methods like in the next test function

```
def testSession2(self):
    with app.test_client() as lTestClient:
      lResp= lTestClient.post('/getSessionVar')
      self.assertEqual(lResp.status_code, 200)
      self.assertEqual(flask.session['sessionVar'], 'hi')
```

Now imagine your function expects a session variable to be set and reacts different on particular values like this

```
@app.route('/changeSessionVar')
def changeSessionVariable():
  if session['existingSessionVar'] != 'hello':
    raise Exception('unexpected session value of existingSessionVar!')

  session['existingSessionVar'] = 'hello world'
  return 'ok'
```

To test this function you have to use so called *session transaction* and open the session in the context of the test client. This function is available since **Flask 0.8**

```
def testSession3(self):
    with app.test_client() as lTestClient:
      #keep the session
      with lTestClient.session_transaction() as lSess:
        lSess['existingSessionVar'] = 'hello'

      #here the session is stored
      lResp = lTestClient.get('/changeSessionVar')
      self.assertEqual(lResp.status_code, 200)
      self.assertEqual(flask.session['existingSessionVar'], 'hello world')
```

Running the tests is as usual for unittest

```
if __name__ == "__main__":
    unittest.main()
```

And in the command line

```
python tests/test_myapp.py
```

Another nice way to run your tests is to use unittest Discovery like this:

```
python -m unittest discover -s tests
```

Read Testing online: https://riptutorial.com/flask/topic/1260/testing

Chapter 21: Working with JSON

Examples

Return a JSON Response from Flask API

Flask has a utility called `jsonify()` that makes it more convenient to return JSON responses

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/get-json')
def hello():
    return jsonify(hello='world') # Returns HTTP Response with {"hello": "world"}
```

Try it with `curl`

```
curl -X GET http://127.0.0.1:5000/api/get-json
{
  "hello": "world"
}
```

Other ways to use `jsonify()`

Using an existing dictionary:

```
person = {'name': 'Alice', 'birth-year': 1986}
return jsonify(person)
```

Using a list:

```
people = [{'name': 'Alice', 'birth-year': 1986},
          {'name': 'Bob', 'birth-year': 1985}]
return jsonify(people)
```

Receiving JSON from an HTTP Request

If the mimetype of the HTTP request is `application/json`, calling `request.get_json()` will return the parsed JSON data (otherwise it returns `None`)

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/echo-json', methods=['GET', 'POST', 'DELETE', 'PUT'])
```

```
def add():

    data = request.get_json()
    # ... do your business logic, and return some response
    # e.g. below we're just echo-ing back the received JSON data
    return jsonify(data)
```

Try it with `curl`

The parameter `-H 'Content-Type: application/json'` specifies that this is a JSON request:

```
 curl -X POST -H 'Content-Type: application/json' http://127.0.0.1:5000/api/echo-json -d
'{"name": "Alice"}'
{
  "name": "Alice"
}
```

To send requests using other HTTP methods, substitute `curl -X POST` with the desired method e.g. `curl -X GET`, `curl -X PUT`, etc.

Read Working with JSON online: https://riptutorial.com/flask/topic/1789/working-with-json

Credits

S. No	Chapters	Contributors
1	Getting started with Flask	arsho, bakkal, Community, davidism, ettanany, Martijn Pieters, mmenschig, Sean Vieira, Shrike
2	Accessing request data	RPi Awesomeness
3	Authorization and authentication	boreq, Ninad Mhatre
4	Blueprints	Achim Munene, Kir Chou, stamaimer
5	Class-Based Views	ettanany
6	Custom Jinja2 Template Filters	Celeo, dylanj.nz
7	Deploying Flask application using uWSGI web server with Nginx	Gal Dreiman, Tempux, user305883, wimkeir, Wombatz
8	File Uploads	davidism, sigmasum
9	Flask on Apache with mod_wsgi	Aaron D
10	Flask-SQLAlchemy	Achim Munene, arsho, Matt Davis
11	Flask-WTF	Achim Munene
12	Message Flashing	Grey Li
13	Pagination	hdbuster
14	Redirect	coralvanda
15	Rendering Templates	arsho, atayenel, Celeo, fabioqcorreia, Jon Chan, MikeC
16	Routing	davidism, Douglas Starnes, Grey Li, junnytony, Luke Taylor, MikeC, mmenschig, sytech
17	Sessions	arsho, PsyKzz, this-vidor

18	Signals	davidism
19	Static Files	arsho, davidism, MikeC, YellowShark
20	Testing	bakkal, oggo
21	Working with JSON	bakkal, g3rv4