Open-Source Report

Flask

An Overview of the Flask Web Framework

Student Group:

Jesse Clapper, Joseph Naro, Matthew Wiewiorski

Teacher:

Jesse Hartloff

Course:

CSE312

# What is Flask?

Flask is an open source web framework meant to provide its users with the tools and libraries necessary to create a variety of web applications. It is a micro-framework with minimal reliance on any external libraries, needing only the Werkzeug utility library and the Jinja2 template engine to function; and making it a light and economical option for web development.

## What does this do for us?

For our purposes we only require most basic usage of Flask in order to handle webserver tasks such as serving files. Flask automatically creates a WSGI webserver that will function as the central object of the webserver when run, with its own functions, templates, and more. There is a lot of objects Flask automatically manages and creates for its own purposes but for our project we are mainly utilizing Flask’s application context, which keeps track of application-level data and functionality for us.

When our program is run it imports from a variety of files, including flask and flask\_socketio, which further calls on the relevant Werkzeug iles. When we generate a Flask application (A WSGI application using the Werkzeug code detailed below) which will open connections to our external MariaDB database and any external clients. Flask handles all of the file handling necessary according to the app.route specifications we define in our code; with each of these direct the control flow of our application when receiving certain input from the client, and handle both GET and POST operations.

We also use Flask to perform all necessary back-and-forth with our MariaDB server. Operating on port 3306 (the typical MySQL port) Flask both commits data to the database and grabs copies of it for it to output to the clients. It also uses Jinja2, which has built in HTML templates in order to quickly output formatted data which we use to output images, comments, likes, and other database information. Both of these are their own programs which we only use part of in order to accomplish our task, but they’re used directly *through* the Flask application instead of alongside it.  
  
A breakdown of how our Flask accomplishes its operations is detailed below:

Werkzeug

Once Flask runs the WSGI application it automatically performs webserver tasks, starting with opening a TCP socket on port 5000 (the default value for Flask) and listening for client-side requests. It then takes these requests, parses them, and (if capable) returns an appropriate HTTP response to the client through the same socket connection. Let us break down each section of this:

**It is imperative to understanding that Flask uses a proxy object called request**. This object is where the current HTTP request on the current thread is stored. All operations that call on request prompt Flask to grab the request immediately as it is on that thread and act on it.

Serving.py

<https://github.com/pallets/werkzeug/blob/master/src/werkzeug/serving.py>

This is main file that handles serving the static files of our website. It calls on HTTP.py explicitly to parse or assemble HTTP requests.

* After the initialization of the socket on port 5000, application (self) listens for a HTTP client request
* Once a request is logged the function handle\_one\_request extracts the entire request into a variable called self.rawrequestline using the readline function
* If the self.rawrequestline variable is filled the function parse\_request is then called and returns self.run\_wsgi
* run\_wsgi writes a premade header in bytes to a file and then uses key value pairs generated as it parses the request to assemble its HTTP response
* Once completed the function runs end\_headers and terminates the control flow around sending the request back.
* The application will resume waiting for another HTTP request just as it had before sending the response.

http.py

<https://github.com/pallets/werkzeug/blob/master/src/werkzeug/http.py>  
<https://werkzeug.palletsprojects.com/en/1.0.x/http/>

This file contains all functions and information pertaining to HTTP requests. It covers both GET and POST requests and generates information pertaining to the request object.

* When an HTTP request is received, Flask (through Werkzeug) uses a variety of functions to parse incoming HTTP headers.
* parse\_options\_header takes a content type like header and places it into a tuple of its type and options, and parse\_set\_header will extract items without regards to cases, placing them in order. This returns a HeaderSet object.
* Other possible functions that perform similar parsing are parse\_list\_header and parse\_dict\_header. These are context dependent based on what is received, but all return a HeaderSet object (albeit formatted differently)
* All HeaderSet objects set by parse\_list/parse\_dict/parse\_set functions can be “dumped” (reassembled back into it’s previous HTTP format”) via dump\_header. Explicit parts of the HeaderSet object can be pulled, ignoring others.
* All of the above form the HTTP parsing section of the code, which use either HeaderSet objects in their functions or assemble HTTP responses using functions from HTTP.py
* These therefore assemble some core components of the request object when it is generated for the current thread.
* This file closely associates with formparser.py. They are separated so that the form data can be accessed from the WSGI environment rather than through HeaderSet objects.

formparser.py

<https://github.com/pallets/werkzeug/blob/master/src/werkzeug/formparser.py>

<https://werkzeug.palletsprojects.com/en/1.0.x/http/>

A file that interacts mainly with HTTP.py. It handles all form parsing at the WSGI environment level; and as such allows us to take relevant form data such as images or usernames and send them *directly* to our database.

* formparser.py mainly uses another import, cStringIO’s StringIO library.
* parse\_form\_data is the main function used here. Once the appropriate form data is placed into the environment it can be called to return a tuple in the form of the stream, form, and files.
* It also has a function parse\_multipart\_headers used to parse explicitly multipart headers from a newline terminated iterable.
* This tuple’s data is what is sent directly to our request object. This data is then accessed via request.form[“key”] and is placed into our database via the db\_cursor.

routing.py

<https://github.com/pallets/werkzeug/blob/master/src/werkzeug/routing.py>

<https://werkzeug.palletsprojects.com/en/1.0.x/routing/>

This file is the dispatcher for our Flask application. It handles multiple client connections and worker-thread management. Werkzeug generates a Map of Rules, where Rule is an object that represents pertinent information about the client connection, including its endpoint. This section requires less coverage than the others, just know that routing.py handles multiple client connections via the rules set in the Map object.

socket.io

<https://github.com/miguelgrinberg/python-socketio/tree/master/socketio>

socket.io is actually not a Werkzeug library. Instead, it is a flask package that handles all of the bi-directional communications between our server and the clients. It mainly revolves around events, with handlers being put in place to deal with specific occurrences in the client/server communication.

In our code we only use socket.io to generate a SocketIO object new\_app. Calling on the emit function of SocketIO we can send our image data from the server to the client side via websockets.

Jinja2

<https://flask.palletsprojects.com/en/1.1.x/tutorial/templates/>

Jinja is the templating language used by Flask to generate HTML information to be sent to the client. It is two primary applications in our project is the automatic formatting of given data such that it can be sent as HTML and viewed by the client, and for forming the “security” preventing us from injection attacks.

Jinja2 in Flask revolves around the render\_template function. All templates written for our application are stored within the flaskr package, and these templates are able to hold static data and input our dynamically generated database data for easy output. Our final documents are rendered by having this template rendered with this database data.

Jinja2 notable autoescapes any and all data that is rendered in HTML templates and does so for all user input. This means that no matter what is placed in the incoming HTML everything will be escaped with safe values that have no unwanted effects. As it does this it renders the Flask application safe from injection attacks.

## Licensing

Flask operates under the BSD 3-Clause source license that dictates that all content within the Flask repository or source distribution must be accompanied with the copyright “Copyright 2010 Pallets” and are not used to advertise personal projects. This doesn’t really affect our code in anyways be we do not redistribute or modifying the source code in any way. Additional information concerning this license can be found here: <https://github.com/pallets/flask/blob/master/LICENSE.rst>