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, which makes it a light and economical option for web development.

What does this do for us?

We primarily use Flask to handle and serve web requests. When run, Flask automatically creates a WSGI application that will function as the central object of the webserver.

There are many objects that Flask automatically manages and creates for its own purposes. For our project, we are mainly utilizing Flask's application context. Flask's application context is different for each incoming request. Internally, it is implemented as a stack-like data structure that contains various information about the current request and response. Specifically, the application context contains, among many other things, the 'request' object, the 'response' object and the 'g' object. The 'request' object contains information about the current request, like the request method and submitted form data. The 'response' object contains information about the response to send to the client (information like response code and headers). The 'g' object is used to store temporary information during a request (like a database connection for instance). The 'ctx.py' and 'globals.py' files show the application context.

https://github.com/pallets/flask/blob/master/src/flask/ctx.py

https://github.com/pallets/flask/blob/master/src/flask/globals.py

When our program is run it imports a variety of packages, including flask and flask_socketio. Our Flask application communicates with an external MariaDB database server (over the standard port 3306) and with clients. Most of the WSGI application setup occurs in the following file (app.py):
https://github.com/pallets/flask/blob/master/src/flask/app.py

Flask handles incoming requests according to the rules we set with the '@app.route' decorator. These rules specify a URL endpoint and HTTP methods (like GET, POST, etc.). If the URL and method of an incoming request matches a '@app.route' decorator, the function under the decorator is executed. These functions handle the request and end with a return statement; the data in the return statement is returned back to the client

who made the request. If the incoming request does not match any '@app.route' decorator, Flask tries to serve a file from the 'static' directory. If there is still not a match, Flask returns a 404 response. The code for the '@app.route' decorator is in the 'route' function in the previously mentioned 'app.py' file.

Our project also uses functionality from the 'helpers.py' file. We use the 'url_for' function to generate URLs for endpoints; this function helps prevent the hardcoding of URLs in the Flask application itself and in template files. We use the 'safe_join' function to join file paths. We, or rather Flask, use the 'send_static_file' function to automatically send files located within the 'static' folder.

https://github.com/pallets/flask/blob/master/src/flask/helpers.py

Our project makes use of Flask's configuration object to store important constants for our web application. For example, we store the name of our upload directory. The configuration object, which is basically a Python dictionary is defined in the 'config.py' file: https://github.com/pallets/flask/blob/master/src/flask/config.py

Much of the remaining functionality of Flask is provided by two other packages: Werkzeug and Jinja2. These two packages are described in further detail below.

Werkzeug

Werkzeug is a simple web server that handles HTTP requests and responses. When Flask runs, it fires up Werkzeug by calling the 'run_simple' function. Werkzeug handles most of the low-level HTTP request and response details for Flask. Specifically, Werkzeug parses incoming HTTP requests and sends data about the incoming request to Flask. Flask takes this data and places most of it into the request object. Then, Flask finds a function to handle the request. Finally, Flask sends data back to Werkzeug, which uses this data to generate a HTTP response.

Werkzeug uses Python's built-in 'HTTPServer' class to handle the basic connection details. The 'HTTPServer' uses python's built-in 'TCPServer' class to handle the server's TCP listening socket and other socket connections with the 'socket' class. By default, Flask instructs Werkzeug to open a TCP listening socket on port 5000. The code for these classes are here: https://github.com/python/cpython/blob/3.8/Lib/http/server.py & https://github.com/python/cpython/blob/3.8/Lib/socketserver.py

The main HTTP functionality of Werkzeug appears in the following main files:

serving.py

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

This is main file that handles server setup and basic HTTP request and response tasks. It calls on http.py (discussed below) to parse or assemble HTTP requests and responses. The 'run_simple' function resides in this file.

- After the initialization of the socket, the application listens for HTTP client requests.
- Once a request is logged the function 'handle_one_request' extracts the entire request into a variable called 'rawrequestline' using the 'readline' function.
- When the 'rawrequestline' variable is filled the function 'parse_request' is then called and returns 'run_wsgi'.
- 'run_wsgi' writes a premade header in bytes to a file and then uses key-value pairs as it parses the request in order to assemble the resultant HTTP response.
- When finished, the 'make_response' function handles sending the response to the client.
- 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 Flask's request object.

- When an HTTP request is received, Werkzeug uses a variety of functions to parse incoming HTTP headers. A few of these functions are described below.
- '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.
- Other possible functions that perform similar parsing are 'parse_list_header' and 'parse_dict_header'. These are context dependent based on what is received.
- The file also contains cookie parsing functions like 'parse_cookie'.
- This file also contains important constants like HTTP status codes and messages.
- Flask's request object is filled with some of the data generated by these functions.

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 process them as desired.

- '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 '(stream, form, and files)'. This function uses two classes for parsing: 'FormDataParser' and 'MultiPartParser'.
- The function 'parse_multipart_headers' is used to parse multipart form data headers.
- Flask's request object received this data in the 'request.form' object.

routing.py

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

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

This file handles the routing for incoming requests. The routing is handled by creating a set of rules. Each rule specifies an endpoint and available request methods for that endpoint. The rules themselves are stored in the 'Map' class. Whenever an incoming request comes in, it is checked against the 'Map' of rules and if there is a match, the function associated with the rule is executed.

base_request.py & base_response.py

https://github.com/pallets/werkzeug/blob/master/src/werkzeug/wrappers/base_request.py

https://github.com/pallets/werkzeug/blob/master/src/werkzeug/wrappers/base_response.py

These two files handle information about the current request and the current response. For example, the classes in 'base_request.py' hold information like the request method, submitted form data, and client cookie data; the classes in 'base_response.py' hold information like the response headers, the response status code, the response mime type, and the response content length. Flask directly interacts with the classes in these two files via Flask's own 'request' and 'response' objects. This can be seen Flask's 'wrapper.py' file: https://github.com/pallets/flask/blob/master/src/flask/wrappers.py

Jinja2

Jinja is the templating engine used by Flask. We use Jinja in our project in order to dynamically generate HTML content based on information present in our database server. Jinja is invoked by Flask via the 'render_template' function, which is located in the 'templating.py' file: https://github.com/pallets/flask/blob/master/src/flask/templating.py

Jinja's internal 'render' template function is located in the 'environment.py' file. The 'environment.py' file also includes information about the templating environment and contains helper functions that preprocess, lex, parse, and compile templates. A compiled template is placed into the 'Template' class where it is "run" and evaluated into its final output. https://github.com/pallets/jinja/blob/master/src/jinja2/environment.py

Most of the template parsing occurs in the 'lexer.py', 'parser.py', 'compiler.py' files. The compiled template is run and evaluated via functions in the 'runtime.py' file.

https://github.com/pallets/jinja/blob/master/src/jinja2/lexer.py

https://github.com/pallets/jinja/blob/master/src/jinja2/parser.py

https://github.com/pallets/jinja/blob/master/src/jinja2/compiler.py

https://github.com/pallets/jinja/blob/master/src/jinja2/runtime.py

Jinja2 automatically escapes all data that is sent to it. This means that no matter what is placed into the HTML template, it will be escaped with safe values that will have no unwanted effects. Since we rely on Jinja2 when working with all user input, our Flask application is safe from injection attacks. The escaping itself is done by the 'escape' function in the 'markupsafe' module:
https://github.com/pallets/markupsafe/blob/master/src/markupsafe/_native.py

Licensing

Flask as well as all of its components operate 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

Flask-SocketIO

Flask-SocketIO is a Flask extension that implements the popular 'socket.io' library. We use Flask-SocketIO in our application to push data to clients that are currently viewing certain pages on our website; the clients then use this data to update the page in close to real-time. Due to limitations with Werkzeug, the Flask-SocketIO extension uses long-polling to connect clients to the server.

Flask-SocketIO itself mainly revolves around events, with handlers being put in place to deal with specific event occurrences between the client and server. We primarily use the 'emit' and 'join_room' functions in the code. In addition we also use the '@.on' decorator as well. These functions are defined in the Flask-SocketIO's '__init__.py' file:
https://github.com/miguelgrinberg/Flask-SocketIO/blob/master/flask_socketio/__init__.py

Flask-SocketIO is a high-level interface for the 'python-socketio' module, which is an interface for the 'python-engineio' module. The main code for these libraries appear the 'server.py' files:

https://github.com/miguelgrinberg/python-socketio/blob/master/socketio/server.py

https://github.com/miguelgrinberg/python-engineio/blob/master/engineio/server.py

https://github.com/miguelgrinberg/python-engineio/blob/master/engineio/socket.py

The client code to establish the long-polling is available here:
https://github.com/socketio/socket.io-client/blob/master/dist/socket.io.dev.js

Licensing

Flask-SocketIO is licensed under the MIT License. The MIT License is an incredibly permissive license that requires only the license text and the author's copyright notice to be displayed: Copyright (c) 2014 Miguel Grinberg

https://github.com/miguelgrinberg/Flask-SocketIO/blob/master/LICENSE