Open-Source Technology Use Report

Proof of knowing your stuff in CSE312

# Flask

## General Information & Licensing

|  |  |
| --- | --- |
| Code Repository | https://github.com/pallets/flask |
| License Type | BSD-3 |
| License Description | * Any source code must contain the license and copyright statement * Documentation for binaries must contain the license and copyright statement * Redistribution for commercial purposes is fine as long as Flask is clearly listed as the copyright holder |
| License Restrictions | * We are not allowed to use the name of the copyright holder or any contributors for endorsing products made using Flask without written permission |
| Who worked with this? |  |

*Use as many of the sections below as needed, or create more, to explain every function, method, class, or object type you used from this library/framework.*

## Purpose

|  |
| --- |
| Replace this text with some that answers the following questions for the above tech:   * This tech provides us a backbone for serving content to the users and receiving data back from them via HTTP/TCP connections. It also has good templating features for dynamically generating content given some sort of input. * The majority of our code uses flask in some way, especially if it relates directly to content seen by the user. |

## *Magic* ⋆★͎۪۫｡˚۰˚☽˚⁀➷｡˚★彡͎۪۫⋆ ༄

|  |
| --- |
| *Dispel the magic of this technology. Replace this text with some that answers the following questions for the above tech:*   * *How does this technology do what it does for you in the* ***Purpose*** *section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.* * *Where is the specific code that does what you use the tech for? You* ***must*** *provide a link to the specific file in the repository for your tech with a line number or number range.*   + *If there is more than one step in the chain of calls (hint: there will be), you must provide links for the entire chain of calls from your code, to the library code that actually accomplishes the task for you.*   + *Example: If you use an object of type HttpRequest in your code which contains the headers of the request, you must show exactly how that object parsed the original headers from the TCP socket. This will often involve tracing through multiple libraries and you must show the entire trace through all these libraries with links to all the involved code.*   *\*This section may grow beyond the page for many features.*  *We are using the flask framework since it contains tools and libraries used to develop a web application efficiently. Objects such as requests we implement is based on Werkzeug. This WSGi toolkit enables us to build a web frame on it. Jinjia2 template allows us to pass Python variable into HTML template easier, and also enable to render a dynamic web page with a specific data source when combines a template.* |

First, we use “@app.route()” to set up a route for the home requests.

@app.route('/') # Serving home page and associated content  
def home():

# handle home request

The @app.route('/') will call the `route` method. Then again it calls the `add\_url\_rule` method to add the url rule into the system.

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L1309>

@setupmethod  
def add\_url\_rule(  
 self,  
 rule: str,  
 endpoint: t.Optional[str] = None,  
 view\_func: t.Optional[ft.RouteCallable] = None,  
 provide\_automatic\_options: t.Optional[bool] = None,  
 \*\*options: t.Any,  
) -> None:  
 if endpoint is None:  
 endpoint = \_endpoint\_from\_view\_func(view\_func) # type: ignore  
 options["endpoint"] = endpoint  
 methods = options.pop("methods", None)  
  
 # if the methods are not given and the view\_func object knows its  
 # methods we can use that instead. If neither exists, we go with  
 # a tuple of only ``GET`` as default.  
 if methods is None:  
 methods = getattr(view\_func, "methods", None) or ("GET",)  
 if isinstance(methods, str):  
 raise TypeError(  
 "Allowed methods must be a list of strings, for"  
 ' example: @app.route(..., methods=["POST"])'  
 )  
 methods = {item.upper() for item in methods}  
  
 # Methods that should always be added  
 required\_methods = set(getattr(view\_func, "required\_methods", ()))  
  
 # starting with Flask 0.8 the view\_func object can disable and  
 # force-enable the automatic options handling.  
 if provide\_automatic\_options is None:  
 provide\_automatic\_options = getattr(  
 view\_func, "provide\_automatic\_options", None  
 )  
  
 if provide\_automatic\_options is None:  
 if "OPTIONS" not in methods:  
 provide\_automatic\_options = True  
 required\_methods.add("OPTIONS")  
 else:  
 provide\_automatic\_options = False  
  
 # Add the required methods now.  
 methods |= required\_methods  
  
 rule = self.url\_rule\_class(rule, methods=methods, \*\*options)  
 rule.provide\_automatic\_options = provide\_automatic\_options # type: ignore  
  
 self.url\_map.add(rule)  
 if view\_func is not None:  
 old\_func = self.view\_functions.get(endpoint)  
 if old\_func is not None and old\_func != view\_func:  
 raise AssertionError(  
 "View function mapping is overwriting an existing"  
 f" endpoint function: {endpoint}"  
 )  
 self.view\_functions[endpoint] = view\_func

First, it checks the “methods” parameter and set it to a tuple of only `GET` if it was not specified. Then, if string “OPTIONS” is not in the “methods”, then it adds string “OPTIONS” to the “required\_methods”.

And it will merge the “required\_methods” into the “methods” tuple.

Next, it uses the “url\_rule\_class” to convert the url(or endpoint) that handles the request and the accepted methods to a “Rule” object.

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L1354>

And then this “Rule” object will be added to the “url\_map”. It will also check if “view\_func”(the function that handles the request) None, if not, it checks if the endpoint has a handle function in the “view\_func” map already, If not again, it will add this “view\_func” to the map “view\_functions” with the endpoint as the key.

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L1358>

This is how the flask handles the routes.

In the beginning, We used “app.run” to start the web server.

if \_\_name\_\_ == "\_\_main\_\_":  
 app.run(debug=True, host='0.0.0.0')

The “run” methods will handle the parameters first(such as setting the default web server port), and it calls “run\_simple” with host, port, options and self as the application parameter.

Later in our project, we used Flask and the Flask-SocketIO to handle http request and websocket connections.

app = Flask()

socketio = SocketIO(app)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 socketio.run(app)

We uses “socketio.run” to start the web and websocket server.

if \_\_name\_\_ == "\_\_main\_\_":  
 socketio.run(app, debug=True, host='0.0.0.0')

We pass the Flask application to the Flask-SocketIO class, in the \_\_init\_\_ function of the socketio, it stores the Flask Application in the “wsgi\_app” of the SockekIO object.

def \_\_init\_\_(self, app=None, \*\*kwargs):  
 if app is not None:  
 # here we attach the SocketIO middleware to the SocketIO object so  
 # it can be referenced later if debug middleware needs to be  
 # inserted  
 self.sockio\_mw = \_SocketIOMiddleware(self.server, app,  
 socketio\_path=resource)  
 app.wsgi\_app = self.sockio\_mw

<https://github.com/miguelgrinberg/Flask-SocketIO/blob/91b5ddc31bebeb6241d281252c711b160550ce01/src/flask_socketio/__init__.py#L250>

The socketio.run() function encapsulates the start up of the web server and replaces the app.run() standard Flask development server start up (according to [Flask-SocketIO documentation](https://flask-socketio.readthedocs.io/en/latest/getting_started.html#initialization)). And SocketIO adds a middleware to the Flask App to handle WebSocket connections, leaving the normal http request handle by the Flask App.

The “run” methods of Flask will handle the parameters first(such as setting the default web server port), and it calls “run\_simple” with host, port, options and self as the application parameter.

Line 1191 in flask/app.py

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L1191>

try:  
 run\_simple(t.cast(str, host), port, self, \*\*options)  
finally:

def run\_simple(  
 hostname: str,  
 port: int,  
 application: "WSGIApplication",  
 use\_reloader: bool = False,  
 use\_debugger: bool = False,  
 use\_evalex: bool = True,  
 extra\_files: t.Optional[t.Iterable[str]] = None,  
 exclude\_patterns: t.Optional[t.Iterable[str]] = None,  
 reloader\_interval: int = 1,  
 reloader\_type: str = "auto",  
 threaded: bool = False,  
 processes: int = 1,  
 request\_handler: t.Optional[t.Type[WSGIRequestHandler]] = None,  
 static\_files: t.Optional[t.Dict[str, t.Union[str, t.Tuple[str, str]]]] = None,  
 passthrough\_errors: bool = False,  
 ssl\_context: t.Optional[\_TSSLContextArg] = None,  
) -> None:  
 *"""Start a development server for a WSGI application. Various  
 optional features can be enabled.  
  
 .. warning::  
  
 Do not use the development server when deploying to production.  
 It is intended for use only during local development. It is not  
 designed to be particularly efficient, stable, or secure.* ***:param*** *hostname: The host to bind to, for example ``'localhost'``.  
 Can be a domain, IPv4 or IPv6 address, or file path starting  
 with ``unix://`` for a Unix socket.* ***:param*** *port: The port to bind to, for example ``8080``. Using ``0``  
 tells the OS to pick a random free port.* ***:param*** *application: The WSGI application to run.* ***:param*** *use\_reloader: Use a reloader process to restart the server  
 process when files are changed.* ***:param*** *use\_debugger: Use Werkzeug's debugger, which will show  
 formatted tracebacks on unhandled exceptions.* ***:param*** *use\_evalex: Make the debugger interactive. A Python terminal  
 can be opened for any frame in the traceback. Some protection is  
 provided by requiring a PIN, but this should never be enabled  
 on a publicly visible server.* ***:param*** *extra\_files: The reloader will watch these files for changes  
 in addition to Python modules. For example, watch a  
 configuration file.* ***:param*** *exclude\_patterns: The reloader will ignore changes to any  
 files matching these :mod:`fnmatch` patterns. For example,  
 ignore cache files.* ***:param*** *reloader\_interval: How often the reloader tries to check for  
 changes.* ***:param*** *reloader\_type: The reloader to use. The ``'stat'`` reloader  
 is built in, but may require significant CPU to watch files. The  
 ``'watchdog'`` reloader is much more efficient but requires  
 installing the ``watchdog`` package first.* ***:param*** *threaded: Handle concurrent requests using threads. Cannot be  
 used with ``processes``.* ***:param*** *processes: Handle concurrent requests using up to this number  
 of processes. Cannot be used with ``threaded``.* ***:param*** *request\_handler: Use a different  
 :class:`~BaseHTTPServer.BaseHTTPRequestHandler` subclass to  
 handle requests.* ***:param*** *static\_files: A dict mapping URL prefixes to directories to  
 serve static files from using  
 :class:`~werkzeug.middleware.SharedDataMiddleware`.* ***:param*** *passthrough\_errors: Don't catch unhandled exceptions at the  
 server level, let the serve crash instead. If ``use\_debugger``  
 is enabled, the debugger will still catch such errors.* ***:param*** *ssl\_context: Configure TLS to serve over HTTPS. Can be an  
 :class:`ssl.SSLContext` object, a ``(cert\_file, key\_file)``  
 tuple to create a typical context, or the string ``'adhoc'`` to  
 generate a temporary self-signed certificate.  
  
 .. versionchanged:: 2.1  
 Instructions are shown for dealing with an "address already in  
 use" error.  
  
 .. versionchanged:: 2.1  
 Running on ``0.0.0.0`` or ``::`` shows the loopback IP in  
 addition to a real IP.  
  
 .. versionchanged:: 2.1  
 The command-line interface was removed.  
  
 .. versionchanged:: 2.0  
 Running on ``0.0.0.0`` or ``::`` shows a real IP address that  
 was bound as well as a warning not to run the development server  
 in production.  
  
 .. versionchanged:: 2.0  
 The ``exclude\_patterns`` parameter was added.  
  
 .. versionchanged:: 0.15  
 Bind to a Unix socket by passing a ``hostname`` that starts with  
 ``unix://``.  
  
 .. versionchanged:: 0.10  
 Improved the reloader and added support for changing the backend  
 through the ``reloader\_type`` parameter.  
  
 .. versionchanged:: 0.9  
 A command-line interface was added.  
  
 .. versionchanged:: 0.8  
 ``ssl\_context`` can be a tuple of paths to the certificate and  
 private key files.  
  
 .. versionchanged:: 0.6  
 The ``ssl\_context`` parameter was added.  
  
 .. versionchanged:: 0.5  
 The ``static\_files`` and ``passthrough\_errors`` parameters were  
 added.  
 """* if not isinstance(port, int):  
 raise TypeError("port must be an integer")  
  
 if static\_files:  
 from .middleware.shared\_data import SharedDataMiddleware  
  
 application = SharedDataMiddleware(application, static\_files)  
  
 if use\_debugger:  
 from .debug import DebuggedApplication  
  
 application = DebuggedApplication(application, evalex=use\_evalex)  
  
 if not is\_running\_from\_reloader():  
 s = prepare\_socket(hostname, port)  
 fd = s.fileno()  
 # Silence a ResourceWarning about an unclosed socket. This object is no longer  
 # used, the server will create another with fromfd.  
 s.detach()  
 os.environ["WERKZEUG\_SERVER\_FD"] = str(fd)  
 else:  
 fd = int(os.environ["WERKZEUG\_SERVER\_FD"])  
  
 srv = make\_server(  
 hostname,  
 port,  
 application,  
 threaded,  
 processes,  
 request\_handler,  
 passthrough\_errors,  
 ssl\_context,  
 fd=fd,  
 )  
  
 if not is\_running\_from\_reloader():  
 srv.log\_startup()  
 \_log("info", \_ansi\_style("Press CTRL+C to quit", "yellow"))  
  
 if use\_reloader:  
 from .\_reloader import run\_with\_reloader  
  
 run\_with\_reloader(  
 srv.serve\_forever,  
 extra\_files=extra\_files,  
 exclude\_patterns=exclude\_patterns,  
 interval=reloader\_interval,  
 reloader\_type=reloader\_type,  
 )  
 else:  
 srv.serve\_forever()

Then, inside the “run\_simple” method, it calls “make\_server” method to create WSGIServer with a few parameters such as the hostname, port and the flask application object. The “make\_server” method will create different types of WSGIServer depends on the parameters “threaded” and “processes”.

<https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/serving.py#L1037>

class BaseWSGIServer(HTTPServer):  
 *"""A WSGI server that that handles one request at a time.  
  
 Use :func:`make\_server` to create a server instance.  
 """* multithread = False  
 multiprocess = False  
 request\_queue\_size = LISTEN\_QUEUE  
  
 def \_\_init\_\_(  
 self,  
 host: str,  
 port: int,  
 app: "WSGIApplication",  
 handler: t.Optional[t.Type[WSGIRequestHandler]] = None,  
 passthrough\_errors: bool = False,  
 ssl\_context: t.Optional[\_TSSLContextArg] = None,  
 fd: t.Optional[int] = None,  
 ) -> None:  
 if handler is None:  
 handler = WSGIRequestHandler  
  
 # If the handler doesn't directly set a protocol version and  
 # thread or process workers are used, then allow chunked  
 # responses and keep-alive connections by enabling HTTP/1.1.  
 if "protocol\_version" not in vars(handler) and (  
 self.multithread or self.multiprocess  
 ):  
 handler.protocol\_version = "HTTP/1.1"  
  
 self.host = host  
 self.port = port  
 self.app = app  
 self.passthrough\_errors = passthrough\_errors  
  
 self.address\_family = address\_family = select\_address\_family(host, port)  
 server\_address = get\_sockaddr(host, int(port), address\_family)  
  
 # Remove a leftover Unix socket file from a previous run. Don't  
 # remove a file that was set up by run\_simple.  
 if address\_family == af\_unix and fd is None:  
 server\_address = t.cast(str, server\_address)  
  
 if os.path.exists(server\_address):  
 os.unlink(server\_address)  
  
 # Bind and activate will be handled manually, it should only  
 # happen if we're not using a socket that was already set up.  
 super().\_\_init\_\_(  
 server\_address, # type: ignore[arg-type]  
 handler,  
 bind\_and\_activate=False,  
 )  
  
 if fd is None:  
 # No existing socket descriptor, do bind\_and\_activate=True.  
 try:  
 self.server\_bind()  
 self.server\_activate()  
 except BaseException:  
 self.server\_close()  
 raise  
 else:  
 # Use the passed in socket directly.  
 self.socket = socket.fromfd(fd, address\_family, socket.SOCK\_STREAM)  
 self.server\_address = self.socket.getsockname()  
  
 if address\_family != af\_unix:  
 # If port was 0, this will record the bound port.  
 self.port = self.server\_address[1]  
  
 if ssl\_context is not None:  
 if isinstance(ssl\_context, tuple):  
 ssl\_context = load\_ssl\_context(\*ssl\_context)  
 elif ssl\_context == "adhoc":  
 ssl\_context = generate\_adhoc\_ssl\_context()  
  
 self.socket = ssl\_context.wrap\_socket(self.socket, server\_side=True)  
 self.ssl\_context: t.Optional["ssl.SSLContext"] = ssl\_context  
 else:  
 self.ssl\_context = None

(werkzeug/serving.py)

The “BaseWSGIServer” is inherts the built in “HTTPServer”. The “make\_server” will pass the “host”, ‘port” and the flask application object as the “application” parameter to create the “BaseWSGIServer” object.

<https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/serving.py#L651>

Since we didn’t pass the “handler”, it will be None, and in the first 2 lines of the “\_\_init\_\_” function of the “BaseWSGIServer” class, “handler” is assigned to “WSGIRequestHandler” if it’s None. So in our case, “handler” is assigned to “WSGIRequestHandler”.

Then the remaining part of “\_\_init\_\_” function process the server address and passing the “server\_address” and handler to the “\_\_init\_\_” function of its parent class, the “HTTPServer”. It also loads the SSL certificate information here, and the rest are the TCP related code.

After creating the “BaseWSGIServer” class, the “run\_simple” method will call the “serve\_forever” method of the “BaseWSGIServer”.

def serve\_forever(self, poll\_interval: float = 0.5) -> None:  
 try:  
 super().serve\_forever(poll\_interval=poll\_interval)  
 except KeyboardInterrupt:  
 pass  
 finally:  
 self.server\_close()

and the “serve\_forever” method will call the “serve\_forever” method of its parent class, the “HTTPServer”. Now the HTTPServer starts the TCP service.

<https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/serving.py#L764>

def serve\_forever(self, poll\_interval=0.5):  
 *"""Handle one request at a time until shutdown.  
  
 Polls for shutdown every poll\_interval seconds. Ignores  
 self.timeout. If you need to do periodic tasks, do them in  
 another thread.  
 """* self.\_\_is\_shut\_down.clear()  
 try:  
 # XXX: Consider using another file descriptor or connecting to the  
 # socket to wake this up instead of polling. Polling reduces our  
 # responsiveness to a shutdown request and wastes cpu at all other  
 # times.  
 with \_ServerSelector() as selector:  
 selector.register(self, selectors.EVENT\_READ)  
  
 while not self.\_\_shutdown\_request:  
 ready = selector.select(poll\_interval)  
 # bpo-35017: shutdown() called during select(), exit immediately.  
 if self.\_\_shutdown\_request:  
 break  
 if ready:  
 self.\_handle\_request\_noblock()  
  
 self.service\_actions()  
 finally:  
 self.\_\_shutdown\_request = False  
 self.\_\_is\_shut\_down.set()

In the “serve\_forever” method of the “HTTPServer”, it uses the “\_ServerSelector()” and “selector.select” in a while loop to continuously check if a new request comes in. If a new request comes in, it calls the “\_handle\_request\_noblock” method to process the request.

<https://github.com/python/cpython/blob/7a0f3c1d92ef0768e082ace19d970b0ef12e7346/Lib/socketserver.py#L216>

<https://github.com/python/cpython/blob/7a0f3c1d92ef0768e082ace19d970b0ef12e7346/Lib/socketserver.py#L229>

<https://github.com/python/cpython/blob/7a0f3c1d92ef0768e082ace19d970b0ef12e7346/Lib/socketserver.py#L238>

def \_handle\_request\_noblock(self):  
 *"""Handle one request, without blocking.  
  
 I assume that selector.select() has returned that the socket is  
 readable before this function was called, so there should be no risk of  
 blocking in get\_request().  
 """* try:  
 request, client\_address = self.get\_request()  
 except OSError:  
 return  
 if self.verify\_request(request, client\_address):  
 try:  
 self.process\_request(request, client\_address)  
 except Exception:  
 self.handle\_error(request, client\_address)  
 self.shutdown\_request(request)  
 except:  
 self.shutdown\_request(request)  
 raise  
 else:  
 self.shutdown\_request(request)

Then, the “\_handle\_request\_noblock” method get “request” and “client\_address” by the method “get\_request”, which should be part of the TCP API.

<https://github.com/python/cpython/blob/7a0f3c1d92ef0768e082ace19d970b0ef12e7346/Lib/socketserver.py#L303> <https://github.com/python/cpython/blob/7a0f3c1d92ef0768e082ace19d970b0ef12e7346/Lib/socketserver.py#L311>

Then it calls the “process\_request” to handle the TCP request.

<https://github.com/python/cpython/blob/7a0f3c1d92ef0768e082ace19d970b0ef12e7346/Lib/socketserver.py#L316>

def process\_request(self, request, client\_address):  
 *"""Call finish\_request.  
  
 Overridden by ForkingMixIn and ThreadingMixIn.  
  
 """* self.finish\_request(request, client\_address)  
 self.shutdown\_request(request)

In the “process\_request” method, it calls the “finish\_request”

<https://github.com/python/cpython/blob/7a0f3c1d92ef0768e082ace19d970b0ef12e7346/Lib/socketserver.py#L341>

<https://github.com/python/cpython/blob/7a0f3c1d92ef0768e082ace19d970b0ef12e7346/Lib/socketserver.py#L347>

def finish\_request(self, request, client\_address):  
 *"""Finish one request by instantiating RequestHandlerClass."""* self.RequestHandlerClass(request, client\_address, self)

In the “finish\_request”, it builds a new “RequestHandlerClass” with request and client\_address as parameters. The “RequestHandlerClass” is the “WSGIRequestHandler” that was passed int the “\_\_init\_\_” method of “BaseWSGIServer”.

<https://github.com/python/cpython/blob/7a0f3c1d92ef0768e082ace19d970b0ef12e7346/Lib/socketserver.py#L358>

<https://github.com/python/cpython/blob/7a0f3c1d92ef0768e082ace19d970b0ef12e7346/Lib/socketserver.py#L360>

class WSGIRequestHandler(BaseHTTPRequestHandler):  
 *"""A request handler that implements WSGI dispatching."""* server: "BaseWSGIServer"

(Line 148 of werkzeug/serving.py)

We can see that the WSGIRequestHandler inherts the BaseHTTPRequestHandler.

<https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/serving.py#L148>

Then, according to the RequestHandler, the “handle” method of the “WSGIRequestHandler” will be called when the request comes in.

def handle(self) -> None:  
 *"""Handles a request ignoring dropped connections."""* try:  
 super().handle()  
 except (ConnectionError, socket.timeout) as e:  
 self.connection\_dropped(e)  
 except Exception as e:  
 if self.server.ssl\_context is not None and is\_ssl\_error(e):  
 self.log\_error("SSL error occurred: %s", e)  
 else:  
 raise

It calls the “handle” method of its parent class “BaseHTTPRequestHandler” to handle the request.

<https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/serving.py#L358>

def handle(self):  
 *"""Handle multiple requests if necessary."""* self.close\_connection = True  
  
 self.handle\_one\_request()  
 while not self.close\_connection:  
 self.handle\_one\_request()

(Line 423 of http/server.py)

<https://github.com/python/cpython/blob/748c6c0921ee02a19e01a35f03ce5f4d9cfde5a6/Lib/http/server.py#L428>

def handle\_one\_request(self):  
 *"""Handle a single HTTP request.  
  
 You normally don't need to override this method; see the class  
 \_\_doc\_\_ string for information on how to handle specific HTTP  
 commands such as GET and POST.  
  
 """* try:  
 self.raw\_requestline = self.rfile.readline(65537)  
 if len(self.raw\_requestline) > 65536:  
 self.requestline = ''  
 self.request\_version = ''  
 self.command = ''  
 self.send\_error(HTTPStatus.REQUEST\_URI\_TOO\_LONG)  
 return  
 if not self.raw\_requestline:  
 self.close\_connection = True  
 return  
 if not self.parse\_request():  
 # An error code has been sent, just exit  
 return  
 mname = 'do\_' + self.command  
 if not hasattr(self, mname):  
 self.send\_error(  
 HTTPStatus.NOT\_IMPLEMENTED,  
 "Unsupported method (%r)" % self.command)  
 return  
 method = getattr(self, mname)  
 method()  
 self.wfile.flush() #actually send the response if not already done.  
 except socket.timeout as e:  
 #a read or a write timed out. Discard this connection  
 self.log\_error("Request timed out: %r", e)  
 self.close\_connection = True  
 return

Then it treats the socket as a stream and reads data from it. But since it calls “readline”, it will only read the first line, which is the requestline.

<https://github.com/python/cpython/blob/748c6c0921ee02a19e01a35f03ce5f4d9cfde5a6/Lib/http/server.py#L391>

<https://github.com/python/cpython/blob/748c6c0921ee02a19e01a35f03ce5f4d9cfde5a6/Lib/http/server.py#L400>

Then it will call “parse\_request” to parse the request.

<https://github.com/python/cpython/blob/748c6c0921ee02a19e01a35f03ce5f4d9cfde5a6/Lib/http/server.py#L410>

As the function name states, the following is about paring the HTTP request, and it will be covered in the next report about paring the HTTP request.

For now, the headers are fully processed by the “parse\_request” method.

Back to the “handle\_one\_request” of the HTTPRequestHandler, it now knows the headers, method, path and the HTTP version number of the HTTP request.

Then it calls a method called “’do\_’+ self.command” of itself. For example, if this request is a “GET” request, it will call “do\_get” of itself.

mname = 'do\_' + self.command  
if not hasattr(self, mname):  
 self.send\_error(  
 HTTPStatus.NOT\_IMPLEMENTED,  
 "Unsupported method (%r)" % self.command)  
 return  
method = getattr(self, mname)  
method()  
self.wfile.flush() #actually send the response if not already done.

So, the “do\_get”, “do\_post”, … (of the SWGIRequestHandler) will be called depends on the method of the request.

<https://github.com/python/cpython/blob/748c6c0921ee02a19e01a35f03ce5f4d9cfde5a6/Lib/http/server.py#L413>

In the “SWGIRequestHandler”, it defines a magic handler, the “\_\_getattr\_\_”

<https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/serving.py#L377>

def \_\_getattr\_\_(self, name: str) -> t.Any:  
 # All HTTP methods are handled by run\_wsgi.  
 if name.startswith("do\_"):  
 return self.run\_wsgi  
  
 # All other attributes are forwarded to the base class.  
 return getattr(super(), name)

Now, calls started by “do\_” will eventually call “self.run\_wsgi” method

<https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/serving.py#L380>

def execute(app: "WSGIApplication") -> None:  
 application\_iter = app(environ, start\_response)  
 try:  
 for data in application\_iter:  
 write(data)  
 if not headers\_sent:  
 write(b"")  
 if chunk\_response:  
 self.wfile.write(b"0\r\n\r\n")  
 finally:  
 if hasattr(application\_iter, "close"):  
 application\_iter.close() # type: ignore  
  
try:  
 execute(self.server.app)  
except (ConnectionError, socket.timeout) as e:  
 self.connection\_dropped(e, environ)  
except Exception as e:  
 if self.server.passthrough\_errors:  
 raise

In the “self.run\_wsgi” method, it calls the “execute” function with the “app” parameter(which is the flask application).

<https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/serving.py#L319>

It calls the builds flask application object with the “start\_response”.

<https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/serving.py#L320>

Then in the “\_\_call\_\_” function of the Flask application object, it calls the “wsgi\_app” method.

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L2551>

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L2498>

def \_\_call\_\_(self, environ: dict, start\_response: t.Callable) -> t.Any:  
 *"""The WSGI server calls the Flask application object as the  
 WSGI application. This calls :meth:`wsgi\_app`, which can be  
 wrapped to apply middleware.  
 """* return self.wsgi\_app(environ, start\_response)

def wsgi\_app(self, environ: dict, start\_response: t.Callable) -> t.Any:  
 *"""The actual WSGI application. This is not implemented in  
 :meth:`\_\_call\_\_` so that middlewares can be applied without  
 losing a reference to the app object. Instead of doing this::  
  
 app = MyMiddleware(app)  
  
 It's a better idea to do this instead::  
  
 app.wsgi\_app = MyMiddleware(app.wsgi\_app)  
  
 Then you still have the original application object around and  
 can continue to call methods on it.  
  
 .. versionchanged:: 0.7  
 Teardown events for the request and app contexts are called  
 even if an unhandled error occurs. Other events may not be  
 called depending on when an error occurs during dispatch.  
 See :ref:`callbacks-and-errors`.* ***:param*** *environ: A WSGI environment.* ***:param*** *start\_response: A callable accepting a status code,  
 a list of headers, and an optional exception context to  
 start the response.  
 """* ctx = self.request\_context(environ)  
 error: t.Optional[BaseException] = None  
 try:  
 try:  
 ctx.push()  
 response = self.full\_dispatch\_request()  
 except Exception as e:  
 error = e  
 response = self.handle\_exception(e)  
 except: # noqa: B001  
 error = sys.exc\_info()[1]  
 raise  
 return response(environ, start\_response)  
 finally:  
 if "werkzeug.debug.preserve\_context" in environ:  
 environ["werkzeug.debug.preserve\_context"](\_cv\_app.get())  
 environ["werkzeug.debug.preserve\_context"](\_cv\_request.get())  
  
 if error is not None and self.should\_ignore\_error(error):  
 error = None  
  
 ctx.pop(error)

It gets the HTTP context by “full\_dispatch\_request” method.

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L1801>

def full\_dispatch\_request(self) -> Response:  
 *"""Dispatches the request and on top of that performs request  
 pre and postprocessing as well as HTTP exception catching and  
 error handling.  
  
 .. versionadded:: 0.7  
 """* # Run before\_first\_request functions if this is the thread's first request.  
 # Inlined to avoid a method call on subsequent requests.  
 # This is deprecated, will be removed in Flask 2.3.  
 if not self.\_got\_first\_request:  
 with self.\_before\_request\_lock:  
 if not self.\_got\_first\_request:  
 for func in self.before\_first\_request\_funcs:  
 self.ensure\_sync(func)()  
  
 self.\_got\_first\_request = True  
  
 try:  
 request\_started.send(self)  
 rv = self.preprocess\_request()  
 if rv is None:  
 rv = self.dispatch\_request()  
 except Exception as e:  
 rv = self.handle\_user\_exception(e)  
 return self.finalize\_request(rv)

It calls the “preprocess\_request” to process the request before dispatching it, then it calls the “dispatch\_request”.

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L1821>

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L1823>

def dispatch\_request(self) -> ft.ResponseReturnValue:  
 *"""Does the request dispatching. Matches the URL and returns the  
 return value of the view or error handler. This does not have to  
 be a response object. In order to convert the return value to a  
 proper response object, call :func:`make\_response`.  
  
 .. versionchanged:: 0.7  
 This no longer does the exception handling, this code was  
 moved to the new :meth:`full\_dispatch\_request`.  
 """* req = request\_ctx.request  
 if req.routing\_exception is not None:  
 self.raise\_routing\_exception(req)  
 rule: Rule = req.url\_rule # type: ignore[assignment]  
 # if we provide automatic options for this URL and the  
 # request came with the OPTIONS method, reply automatically  
 if (  
 getattr(rule, "provide\_automatic\_options", False)  
 and req.method == "OPTIONS"  
 ):  
 return self.make\_default\_options\_response()  
 # otherwise dispatch to the handler for that endpoint  
 view\_args: t.Dict[str, t.Any] = req.view\_args # type: ignore[assignment]  
 return self.ensure\_sync(self.view\_functions[rule.endpoint])(\*\*view\_args)

Not it takes the url to build a rule object, then at the end, it tries to look for the route for this endpoint from “self.view\_functions”, and then dispatch this request to the corresponding route.

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L1799>

After our Application returns a Response, the variable “rv” will be assigned to the returned response of dispatch\_request(), the returned to the “full\_dispatch\_request”.

And then the Flask app calls “return response(environ, start\_response)” to build the response.

So the \_\_call\_\_ function of the Response will be call with the environ and start\_response

def \_\_call\_\_(  
 self, environ: "WSGIEnvironment", start\_response: "StartResponse"  
) -> t.Iterable[bytes]:  
 *"""Process this response as WSGI application.* ***:param*** *environ: the WSGI environment.* ***:param*** *start\_response: the response callable provided by the WSGI  
 server.* ***:return****: an application iterator  
 """* app\_iter, status, headers = self.get\_wsgi\_response(environ)  
 start\_response(status, headers)  
 return app\_iter

In the \_\_call\_\_function it uses “start\_response” to send the response.

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L2551>

def write(data: bytes) -> None:  
 nonlocal status\_sent, headers\_sent, chunk\_response  
 assert status\_set is not None, "write() before start\_response"  
 assert headers\_set is not None, "write() before start\_response"  
 if status\_sent is None:  
 status\_sent = status\_set  
 headers\_sent = headers\_set  
 try:  
 code\_str, msg = status\_sent.split(None, 1)  
 except ValueError:  
 code\_str, msg = status\_sent, ""  
 code = int(code\_str)  
 self.send\_response(code, msg)  
 header\_keys = set()  
 for key, value in headers\_sent:  
 self.send\_header(key, value)  
 header\_keys.add(key.lower())  
  
 # Use chunked transfer encoding if there is no content  
 # length. Do not use for 1xx and 204 responses. 304  
 # responses and HEAD requests are also excluded, which  
 # is the more conservative behavior and matches other  
 # parts of the code.  
 # https://httpwg.org/specs/rfc7230.html#rfc.section.3.3.1  
 if (  
 not (  
 "content-length" in header\_keys  
 or environ["REQUEST\_METHOD"] == "HEAD"  
 or (100 <= code < 200)  
 or code in {204, 304}  
 )  
 and self.protocol\_version >= "HTTP/1.1"  
 ):  
 chunk\_response = True  
 self.send\_header("Transfer-Encoding", "chunked")  
  
 # Always close the connection. This disables HTTP/1.1  
 # keep-alive connections. They aren't handled well by  
 # Python's http.server because it doesn't know how to  
 # drain the stream before the next request line.  
 self.send\_header("Connection", "close")  
 self.end\_headers()  
  
 assert isinstance(data, bytes), "applications must write bytes"  
  
 if data:  
 if chunk\_response:  
 self.wfile.write(hex(len(data))[2:].encode())  
 self.wfile.write(b"\r\n")  
  
 self.wfile.write(data)  
  
 if chunk\_response:  
 self.wfile.write(b"\r\n")  
  
 self.wfile.flush()

It use “self.send\_response(code, msg)” to send the response first, then use a for-loop to send the headers

for key, value in headers\_sent:  
 self.send\_header(key, value)  
 header\_keys.add(key.lower())

<https://github.com/pallets/werkzeug/blob/3115aa6a6276939f5fd6efa46282e0256ff21f1a/src/werkzeug/serving.py#L262>

def send\_header(self, keyword, value):  
 *"""Send a MIME header to the headers buffer."""* if self.request\_version != 'HTTP/0.9':  
 if not hasattr(self, '\_headers\_buffer'):  
 self.\_headers\_buffer = []  
 self.\_headers\_buffer.append(  
 ("%s: %s\r\n" % (keyword, value)).encode('latin-1', 'strict'))

In send\_header, it concats the header stirng by “%s: %s \r\n” format.

And then add an empty line in the “write” function by “self.wfile.write(b"\r\n")” And since this empty line separates the headers and the body, so the the empty line follows by the body(data).

<https://github.com/python/cpython/blob/748c6c0921ee02a19e01a35f03ce5f4d9cfde5a6/Lib/http/server.py#L516>

And this is how Flask works.

We also used other methods in the “app.py” of our project, such as “redirect” and “render\_template””.

Redirect:

def redirect(  
 location: str, code: int = 302, Response: t.Optional[t.Type["BaseResponse"]] = None  
) -> "BaseResponse":  
 *"""Create a redirect response object.  
  
 If :data:`~flask.current\_app` is available, it will use its  
 :meth:`~flask.Flask.redirect` method, otherwise it will use  
 :func:`werkzeug.utils.redirect`.* ***:param*** *location: The URL to redirect to.* ***:param*** *code: The status code for the redirect.* ***:param*** *Response: The response class to use. Not used when  
 ``current\_app`` is active, which uses ``app.response\_class``.  
  
 .. versionadded:: 2.2  
 Calls ``current\_app.redirect`` if available instead of always  
 using Werkzeug's default ``redirect``.  
 """* if current\_app:  
 return current\_app.redirect(location, code=code)  
  
 return \_wz\_redirect(location, code=code, Response=Response)

redirect method calls a function “\_wz\_redirect” with the parameters from the “redirect” method, which is the following method.

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/helpers.py#L287>

def redirect(  
 location: str, code: int = 302, Response: t.Optional[t.Type["Response"]] = None  
) -> "Response":  
 *"""Returns a response object (a WSGI application) that, if called,  
 redirects the client to the target location. Supported codes are  
 301, 302, 303, 305, 307, and 308. 300 is not supported because  
 it's not a real redirect and 304 because it's the answer for a  
 request with a request with defined If-Modified-Since headers.  
  
 .. versionadded:: 0.6  
 The location can now be a unicode string that is encoded using  
 the :func:`iri\_to\_uri` function.  
  
 .. versionadded:: 0.10  
 The class used for the Response object can now be passed in.* ***:param*** *location: the location the response should redirect to.* ***:param*** *code: the redirect status code. defaults to 302.* ***:param*** *class Response: a Response class to use when instantiating a  
 response. The default is :class:`werkzeug.wrappers.Response` if  
 unspecified.  
 """* if Response is None:  
 from .wrappers import Response # type: ignore  
  
 display\_location = escape(location)  
 if isinstance(location, str):  
 # Safe conversion is necessary here as we might redirect  
 # to a broken URI scheme (for instance itms-services).  
 from .urls import iri\_to\_uri  
  
 location = iri\_to\_uri(location, safe\_conversion=True)  
  
 response = Response( # type: ignore  
 "<!doctype html>\n"  
 "<html lang=en>\n"  
 "<title>Redirecting...</title>\n"  
 "<h1>Redirecting...</h1>\n"  
 "<p>You should be redirected automatically to the target URL: "  
 f'<a href="{escape(location)}">{display\_location}</a>. If'  
 " not, click the link.\n",  
 code,  
 mimetype="text/html",  
 )  
 response.headers["Location"] = location  
 return response

In this method, it will build a Response by creating a “Response” Object with a default html content telling the page is redirecting.

And then set the header of “Location” to the url that it’s going to redirect, then this Response object will be returned to the app and it will be send to the browser by the “make\_response” method.

And render\_template is similar.

def \_render(app: "Flask", template: Template, context: t.Dict[str, t.Any]) -> str:  
 app.update\_template\_context(context)  
 before\_render\_template.send(app, template=template, context=context)  
 rv = template.render(context)  
 template\_rendered.send(app, template=template, context=context)  
 return rv  
  
  
def render\_template(  
 template\_name\_or\_list: t.Union[str, Template, t.List[t.Union[str, Template]]],  
 \*\*context: t.Any  
) -> str:  
 *"""Render a template by name with the given context.* ***:param*** *template\_name\_or\_list: The name of the template to render. If  
 a list is given, the first name to exist will be rendered.* ***:param*** *context: The variables to make available in the template.  
 """* app = current\_app.\_get\_current\_object() # type: ignore[attr-defined]  
 template = app.jinja\_env.get\_or\_select\_template(template\_name\_or\_list)  
 return \_render(app, template, context)

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/templating.py#L127>

It uses jinja2 to render the template, and then return the rendered page to the Flask App, and then returned data will be used to make response in “finalize\_response” method by the highlighted part in the code below:

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L1828>

def finalize\_request(  
 self,  
 rv: t.Union[ft.ResponseReturnValue, HTTPException],  
 from\_error\_handler: bool = False,  
) -> Response:  
 *"""Given the return value from a view function this finalizes  
 the request by converting it into a response and invoking the  
 postprocessing functions. This is invoked for both normal  
 request dispatching as well as error handlers.  
  
 Because this means that it might be called as a result of a  
 failure a special safe mode is available which can be enabled  
 with the `from\_error\_handler` flag. If enabled, failures in  
 response processing will be logged and otherwise ignored.  
  
 :internal:  
 """* response = self.make\_response(rv)  
 try:  
 response = self.process\_response(response)  
 request\_finished.send(self, response=response)  
 except Exception:  
 if not from\_error\_handler:  
 raise  
 self.logger.exception(  
 "Request finalizing failed with an error while handling an error"  
 )  
 return response

Then it will be also sent to the browser by the “make\_response” method.

<https://github.com/pallets/flask/blob/066a35dd322f689ec07d7c0e82b19eacadac3c6b/src/flask/app.py#L1847>