Open-Source Technology Use Report

Proof of knowing your stuff in CSE312

Guidelines

Provided below is a template you must use to write your report for each of the technologies you use in your project.

Here are some things to note when working on your report, specifically about the **General Information & Licensing** section for each technology.

- **Code Repository**: Please link the code and not the documentation. If you'd like to refer to the documentation in the **Magic** section, you're more than welcome to, but we'd like to see the code you're referring to as well.
- **License Type**: Three letter acronym is fine.
- **License Description**: No need for the entire license here, just what separates it from the rest.
- **License Restrictions**: What can you *not* do as a result of using this technology in your project? Some licenses prevent you from using the project for commercial use, for example.
- **Who worked with this?**: It's not necessary for the entire team to work with every technology used, but we'd like to know who worked with what.

Also, feel free to extend the cell of any section if you feel you need more room.

If there's anything we can clarify, please don't hesitate to reach out! You can reach us using the methods outlined on the course website or see us during our office hours.

Flask

General Information & Licensing

Code Repository	https://github.com/pallets/flask
License Type	BSD 3-Clause "New" or "Revised" License
License Description	"*A permissive license similar to the BSD 2-Clause License, but with a 3rd clause that prohibits others from using the name of the project or its contributors to promote derived products without written consent.*" - Github description
License Restrictions	• Is not liable for damages • Does not come with a software warranty
Who worked with this?	[Contributors] Everyone in the group worked with this framework. It is the base framework for an http server where everything is build around it.

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.

[app.route(rule, **options)]

Purpose

Replace this text with some that answers the following questions for the above tech:

- What does this tech do for you in your project?
    - This tech adds a route to our flask app which registers a user. It can accept either HTTP GET or HTTP POST.
- Where specifically is this tech used in your project? Give us some details like file location and line number, if applicable. If too cumbersome, a general description of where it's used for a given purpose is fine as well.
    - We use this tech in a lot of locations in our code. One example of where it is used is in /auth/auth.py. In line 24.
    - The purpose of using this is so that it adds routes to our application. For example in our repo on line 24 of /auth/auth.py, we added the route for /register so users are able to go to the /register route to register for an account to use on our application

Magic

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.
    - In our case, we are using this so that we can create paths for different purposes in our application. On line 24 in our repo in the file /auth/auth.py, we created a register path on the path /register so users can sign up.
        - On line 43 in /auth/auth.py we created a login path on the path /login which will allow users to login with an account that is already registered in our system.
    - Flask builds the path by using information given from the user. For example, if they were to call app.route("/") flask will be building the path for the home directory, and will incorporate whatever was implemented inside of the users function for their app.route("/")
- 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..

In src/flask/app.py
Lines 1037 to 1094 is involved with creating the functionality of app.route. When the user attempts to create a path in their application using app.route("/*insert_path_here*") add_url_rule() is called to help registers rules for routing incoming requests and building URLs. Calling the ".route" is a shortcut to calling this function so many users opt with calling it by app.route().

In src/flask/scaffold.py
Lines 745 to 750. This is just a helper function that returns the default endpoints for a given function. It is used in src/flask/app.py on line 1047 to set up an endpoint.

*This section may grow beyond the page for many features.

[Blueprint]

Purpose

Replace this text with some that answers the following questions for the above tech:

- What does this tech do for you in your project?
    - A blueprint is an object that allows defining application functions without requiring an application object ahead of time. It uses the same decorators as :class:`~flask.Flask`, but defers the need for an application by recording them for later registration.
    - Link: https://github.com/pallets/flask/blob/main/src/flask/blueprints.py
- Where specifically is this tech used in your project? Give us some details like file location and line number, if applicable. If too cumbersome, a general description of where it's used for a given purpose is fine as well.
    - This is the backbone of our auth.py and chat.py apps. We do not use two separate flask apps, rather blueprints to abstract the web server functionality and only focus on content. We then join these two applications together with register_blueprint.

Magic ★★ ° ˚ ☽ ˚ ⌒ ˚ ★ ≡ ★ ≈

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.
    - A blueprint is a way to store static routes. It does not parse nor get anything from the TCP socket. It basically defines the structure of the flask app. The flask app can then import this blueprint with a url prefix for all routes of the blueprint by calling register_blueprint.
    - This was useful to us for separating functionality and adding encapsulation to our code and avoiding putting everything into one file.
- 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.
    - The entire code is in https://github.com/pallets/flask/blob/main/src/flask/blueprints.py
    - The route function is the same as the route function in the previous section.

[render_template]

Purpose

Replace this text with some that answers the following questions for the above tech:

- What does this tech do for you in your project?
    - This tech allows us to render html from a folder for pages we need.
- Where specifically is this tech used in your project? Give us some details like file location and line number, if applicable. If too cumbersome, a general description of where it's used for a given purpose is fine as well.
    - It is used in /auth/auth.py on lines 39, 61, and 87.
    - It is also used in /chat/chat.py on line 22

Magic

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.
    - Render template does this for us by integrating the Jinja library with it. We feed it the information it needs, which in our case would be the html page and any variables that should be available in the context of the template. This information is then fed into the Jinja library and the html will then be rendered on our page.
    - The function used in the Jinja library is the get_or_select_template() function. It can return one of two functions as the name suggests. get_template() or select_template()
- 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.

In the flask library src/flask/templating.py
Lines 130 to 148 is involved with creating the render_template functionality for when the user calls render_template(html_page, context). As the name of the function may suggest

In the jinja library /src/jinja2/enviroment.py
Lines 1057 to 1074 contains the code that is used in the render_template() code in the flask templating.py file.

In the jinja library /src/jinja2/enviroment.py
Lines 966 to 1000 contains the code used for get_template()

In the jinja library /src/jinja2/enviroment.py
Lines 1003 to 1054 contains the code used for select_template()

*This section may grow beyond the page for many features.

[run(host=None, port=None, debug=None, load_dotenv=True, **options)]

Purpose

Replace this text with some that answers the following questions for the above tech:

- What does this tech do for you in your project?
    - At a high level, this starts the flask application on the host and port.
    - This will run a multithreaded TCP server, handling each request as a separate thread.
    - It waits until a user types the host name on their browser, then parses the HTTP headers and serves them the content they need.
- Where specifically is this tech used in your project? Give us some details like file location and line number, if applicable. If too cumbersome, a general description of where it's used for a given purpose is fine as well.
    - This is used in our app.py file to start the server.

Magic

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.
    - Starting from the TCP socket, the BaseHTTPRequest Handler is used to parse the HTTP request, parsing one request at a time.
    - After the request is parsed, the function run_wsgi is called, which handles processing the parsed request and generating the response needed.
    - There is a function in run_wsgi that is called execute. This function interfaces with the Flask application (or any WSGI application) and generates a response.
    - In the execute function, the application is probed for its routes in a function called debug_application.
        - In this function, an iterator is generated that produces data as bytes to append to a response that will eventually be sent to the end user.
    - In this debug_application, flask is called like a function, and returns its self.wsgi_app function output
    - Inside of the wsgi_app function, another function called full_dispatch_request is called, which its description says
        - Dispatches the request and on top of that performs request pre and post processing as well as HTTP exception catching and error handling.
    - Inside of the full_dispatch_request, a dispatch_request sub function is called, which retrieves the url rule, and calls the flask route mapping to get the function needed to handle the particular request. In our case, it's chat.index. It also passes in any payload if any payload is there via req.view_args.
    - In our index function, render_template is called, which is described thoroughly in another section of this report. It returns an html file as a string.
        - This value is saved in a variable called rv in the full_dispatch_request sub function.
    - At the end of full_dispatch_request, finalize_request is called, which accepts rv calls make_response to determine the type of rv and construct the headers around it. In this case, rv is type str to represent an html file.
        - In the make_response function, the exact type of rv is determined and loaded into a response class, which will construct the headers automatically for the given data rv. This new rv, which wraps around the original response to include the headers is returned.
    - In finalize_request, response is now a variable that holds rv and the response headers. It then calls a function process_response for additional post processing.
        - In this case, it doesn't do anything because post processing isn't used in our project.
    - After this, finalize_request calls the send method of the signal library request_finished (which doesn't do anything). It then returns back to wsgi_app
    - Back in wsgi_app, the response variable now has the response for the get request to root. This response variable is callable:
        - Inside itself, it gets the response headers and an application iterator and returns that and the status (which is 200 OK)
    - In this callable response, the parameter start_response function is called,

which leads to the start_response in the WSGIRequestHandler being called.
        - In this function, the status and the headers are collected, and the write function of the WSGIRequestHandler is returned back to the wsgi_app function. The wgsi_app function is now executed, lending control back to the debug_application function and returning the response to the request.
    - Back in the debug_application_function, the data in the response along with the headers are set as global variables, and the control is returned to the execute function.
    - In the execute function, the write method is called. In this write method, the data, along with the stored headers are retrieved and sent via the socket with various functions described below in code. The handler is now reset and waiting for the next request to repeat this process.
- 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.

In werkzeug -> serving.py -> WSGIRequestHandler:

(def handle(self) -> None) - Line(s) 344 - 355. Calls super().handle and function gives control to server.py

In http -> server.py -> BaseHTTPRequestHandler:

(def handle(self):) - Line(s) 423 - 429. Calls handle_one_request in the same server.py

(def handle_one_request(self):) - Line(s) 386 - 421. Calls parse_request in the same server.py, and requests the top HTTP line from the raw socket.

(def parse_request(self):) - Line(s) 269 - 366. Reads the raw TCP socket and parses the HTTP method, the route and the HTTP version. Calls http.client.parse_headers to parse the rest of the headers. Also does error checking, and raises an error if any of the headers are invalid. Errors are handled by send_error in the same file line(s) 431 - 482. Returns true iff the headers were able to be parsed and are valid and false otherwise.

In http -> client.py

(def parse_headers:) - Line(s) 224 - 236. Calls _read_headers of the same file to read from the raw socket and joins the headers separated by '\r\n;. Also calls email.parser.Parser(_class=_class).parsestr for additional processing.

(def _read_headers(fp):) - Line(s) 206 - 222. Reads at most _MAXHEADERS from the socket. If the line is '\r\n', then that marks the end of the headers and the beginning of the payload, which means the function returns.

In lib/email/parser.py

(def parsestr(self, text, headersonly=False):) - Line(s) 59 - 67. Parses and creates message structure from string. Calls parse of the same file.

(def parse(self, fp, headersonly=False):) - Line(s) 41 - 57. Reads and returns the message structure from a file pointer.

Back in In http -> server.py -> BaseHTTPRequestHandler:
(def parse_request(self):) - Line(s) 269 - 366. After parsing the rest of the headers, the function returns to handle_one_request.

(def handle_one_request(self):) - Line(s) 386 - 421. After parse_request returns, gets the method matched by mname, which is do_GET. Regardless of mname however, the function run_wsgi in werkzeug -> serving.py -> WSGIRequestHandler is called and it handles the rest of the request.

Back in werkzeug -> serving.py -> WSGIRequestHandler:

(def run_wsgi(self) -> None:) - Line(s) 239 - 342. This is the hotspot of this HTTP server. All functions and methods are routed through this. Starts off calling (def make_environ(self) -> "WSGIEnvironment":) of the same file to parse the headers into a form it understands easier.

(def make_environ(self) -> "WSGIEnvironment":) - Line(s) 159 - 237. Parses the headers further. All of the info this function creates is saved into the self.environ variable.

(def run_wsgi(self) -> None:) - Line(s) 239 - 342. After the headers are processed, this function defines 3 functions - write, start_response and execute. Immediately, execute is called with the flask application.

(def execute(app: "WSGIApplication") -> None:) - Line(s) 305 - 316. Generates an application iterable which has the response content and the headers. Calls debug_application to generate the iterator in werkzeug -> debug -> __init__.py passing in environ and a function called start_response.

In werkzeug -> debug -> __init__.py

(def debug_application(
        self, environ: "WSGIEnvironment", start_response: "StartResponse"
) -> t.Iterator[bytes]:
) - Line(s) 305 - 345. Calls the callable WSGI Flask app to generate the response content to send.

In flask -> app.py

(def __call__(self, environ: dict, start_response: t.Callable) -> t.Any:) - Line(s) 2090 - 2095. Middleware and calls wsgi_app of the same file.

(def wsgi_app(self, environ: dict, start_response: t.Callable) -> t.Any:) - Line(s) 2047 -2088. Calls full_dispatch_request of the same file and retrieves its return value which is a response object.

(def full_dispatch_request(self) -> Response:) - Line(s) 1511 - 1526. Calls dispatch_request to deal with the current request.

(def dispatch_request(self) -> ResponseReturnValue:) - Line(s) 1487 - 1509. Gets the url path, and matches it with a flask view function (the functions declared below the @route decorator). It then passes the body if one exists into the function. The return value is stored in rv. Then, self.finalize_request(rv) is called of the same file.

*** This is where our body function is ***

(def finalize_request(
        self,
        rv: t.Union[ResponseReturnValue, HTTPException],
        from_error_handler: bool = False,
) -> Response:
) - Line(s) 1528 - 1555. Generates a response with the correct mime_type using make_response. The resulting object is stored in the variable response. The response is then processed further with process_response. Then, the response is sent to a dud signal using the send method from request_finished.

(def process_response(self, response: Response) -> Response:) - Line(s) 1868 - 1894. In most cases, this does nothing and does not modify the parameter response. Returns the response

In flask -> signals.py

(def send(self, *args: t.Any, **kwargs: t.Any) -> t.Any:) - Line(s) 25 - 26. Does nothing.

Back in flask -> app.py

(def wsgi_app(self, environ: dict, start_response: t.Callable) -> t.Any:) - Line(s) 2047 -2088. After a response has been generated, it is callable and called, with the parameter environ and start_response being passed in.

In werkzeug -> wrappers -> response.py

(def __call__(
        self, environ: "WSGIEnvironment", start_response: "StartResponse"
) -> t.Iterable[bytes]:
) - Line(s) 619 - 631. Gets the application iterator, the headers and the status from get_wsgi_response function (passing in the environ parameter) and calls start_response with the status and the headers.

Back in werkzeug -> serving.py -> WSGIRequestHandler:

(def start_response(status, headers, exc_info=None):) - Line(s) 291 - 303. Sets the global variables status_set and headers_set for when the response is written to the socket, and returns the write function of the same file. This return function won't be used however.

Back in werkzeug -> wrappers -> response.py

(def __call__(
        self, environ: "WSGIEnvironment", start_response: "StartResponse"
) -> t.Iterable[bytes]:
) - Line(s) 619 - 631. Returns the application iterator.

Back in werkzeug -> debug -> __init__.py

(def debug_application(
        self, environ: "WSGIEnvironment", start_response: "StartResponse"
) -> t.Iterator[bytes]:
) - Line(s) 305 - 345. Yields the application iterator back to the execute function in werkzeug -> debug -> __init__.py . Execute then calls write for every bit of data in the application iterator.

(def write(data: bytes) -> None:) - Line(s) 250 - 289. Aggregates all the response headers, and sends them via the raw socket in end_headers. It then also sends the data over the raw socket at line 284. The headers are located in _headers_buffer, and the end_headers function adds an additional '\r\n' before joining all of the headers together and sending them over the raw socket. After this, a long sequence of return statements back to the handle method in the werkzeug -> serving.py -> WSGIRequestHandler happens, and the process repeats for all future requests.

For errors concerning 404, self.dispatch_request() will throw an exception, which is handled at line 1524 in flask -> app.py.

[url_for]

Purpose

Replace this text with some that answers the following questions for the above tech:

- What does this tech do for you in your project?
    - It takes a string which represents a function to call or folder, and a bunch of optional arguments if the route has optional parameters.
        - If a route is defined with /route/<a>, url_for can use 'route', and say a=something.
- Where specifically is this tech used in your project? Give us some details like file location and line number, if applicable. If too cumbersome, a general description of where it's used for a given purpose is fine as well.
    - This is used in base.html to generate the links for the css files, html files, etc…

Magic

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.

WORK IN PROGRESS.

[app.register_blueprint(auth, url_prefix="/")

app.register_blueprint(chat, url_prefix="/")]

Purpose

Replace this text with some that answers the following questions for the above tech:

- What does this tech do for you in your project?
    - It takes a sub flask app with routes already made and it puts it into the current app
- Where specifically is this tech used in your project? Give us some details like file location and line number, if applicable. If too cumbersome, a general description of where it's used for a given purpose is fine as well
    - This is used in app.py on line(s) 21 - 22.

Magic

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.
    - This only adds user routes to an internal registry, meaning all the work is done before the server starts up.
    - This moves all the routes and configurations (in our case the auth and chat routes) to the flask app and allows the app to use both.
- 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.

In flask -> app.py:

Line(s) 1003 - 1028 involved registering the blueprint to the flask app. Using the url prefix "/" indicates that all the blueprint routes will have a url prefix of "/". For instance, if we have '/chat' as a blueprint route with a url prefix of '/hi', the route in app.py will be /hi/chat.

This calls blueprint.register(self, options), which will be described below.

In flask -> blueprints.py

Line(s) 271 - 391 involved dissecting the blueprint and adding all the routes of the blueprint to the app. It checks to see if the blueprint is already present in the flask app, and if it is not, will add the blueprint to the list of blueprints in the flask app. A blueprint can also register another blueprint, which can register another blueprint, so this function can be recursive to add all the chained blueprints to the flask app.