Open-Source Report

Proof of knowing your stuff in CSE312

Guidelines

Provided below is a template you must use to write your reports for 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 need 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.

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

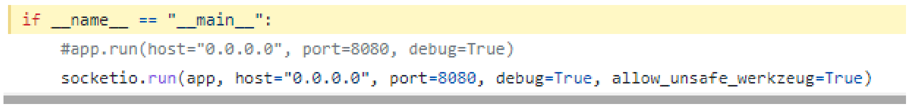
Code Repository	https://github.com/pallets/flask
License Type	MIT
License Description	● Gives users the permission to reuse code for any purpose. Is under the BSD-Style and permissive class of licenses. ● Doesn't require modifications to be open sourced ● Only condition required to use software is to include the same copyright notice in all copies or any substantial portions of the software
License Restrictions	● If the code is distributed, the license automatically becomes a GPL license. ● There are no warranties for the software and no attribution ● No patent protection and no copyleft

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? Please explain this in detail, starting from **after the TCP socket is created**

  After the TCP socket is created, this is when in the program it runs the line,

  ```
  if __name__ == "__main__":
      #app.run(host="0.0.0.0", port=8080, debug=True)
      socketio.run(app, host="0.0.0.0", port=8080, debug=True, allow_unsafe_werkzeug=True)
  ```

  (link: https://github.com/vwong175/cse312_group/blob/main/server.py#L159)
  To which it will start up the server and in the application python file, we have routes defined and functions those route decorators are attached to where the function will return the expected response when the client requests for the defined route.

  This is seen in all of the decorator lines where we have: `@app.route()` and thus function takes in some string as a parameter representative as the URL rule. The route function is used to register a view function for a given URL rule.

- 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.

`@setupmethod add_url_rule`: (line: https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/app.py#L1309)
For the `add_url_rule`, if a view function was passed to it, then that view function is registered with the endpoint. What this means is that it connects the URL rule that was defined in the decorator of the `route()` function to the name of that function the decorator is assigned with. In other words, the endpoint is the function name and the view function is the actual function body that we define under the @app.route() decorator that we want to be triggered for a specific endpoint request.

The add_url_rule is of type Rule and defaults to the class: werkzeug.routing.Rule

If the endpoint is not given, then there is another method that is called, _endpoint_from_view_funct(view_func), that will return a default name for the endpoint. Regardless, the endpoint is stored as the value to the key 'endpoint' in the dictionary named 'options', and we get the method associated with that endpoint.

The add_url_rule does not return anything, but rather in the process of it being called, it gets all of the required methods as defined in line: https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/app.py#L1352 .

`@setupmethod def endpoint(self, endpoint:str) -> t.Callable[[F], F]` link: (https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/scaffold.py#L523)
As mentioned previously, in both route() and add_url_rule(), both of these functions rely on also the idea of making an endpoint happen and putting that in the Flask app. As seen in the above permalink, what this function takes in is the endpoint as a string and returns back a t.Callable. In the body of this function, it similarly defines a decorator function and instead of it being the same code as the route() function, the decorator defined for an endpoint is assigned in the view functions with the endpoint as the key, and the associated value is f. The decorator function returns and so the endpoint function terminates and returns with the decorator function.

Trace:

1. Start from creating a Flask app called "`app`".
   (link: https://github.com/vwong175/cse312_group/blob/main/server.py#L159).
2. In trace 1 line, it will call on library function `__init__` in library *flask/app.py* to create a flask server base on line 553 to line 718. After this a flask server is done with the name "`app`".
   (link: https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/app.py#L553).
3. Then start from our server.py line 25 (link: https://github.com/vwong175/cse312_group/blob/main/server.py#L25), we call on "`@app.route`" to communicate with clients by using the TCP package. When looking at the `app.route()` decorator, it calls the setup method (https://github.com/vwong175/cse312_group/blob/main/server.py#L159) and in it, there is a function definition for wrapper_func. It needs to make the check if the setup for the route is finished through calling another function _check_setup_finished() (https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/app.py#L720). This will return back to wrapper_func as defined within the setup method and return f.

4. `route()`: (link: https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/scaffold.py#L423). After the setup method from step 3, the route function is executed since it is decorated with the @setup. The parameters to `route()` is the URL rule as a string, as well as options, and returns a decorator. In the route() function, it defines the decorator function as well where it takes a T_route and returns a T_route. Essentially, the decorator in this case is a function that uses the options that were passed into route, pops the "endpoint" if given, and then proceeds to run the `add_url_rule` method which adds the URL rule that was passed down to the Flask application.

5. The code stack jumps back to server.py where it is now at the line of the function definition header. It then jumps back to the scaffold.py, specifically back to the route() function definition in this module, and is at this line: https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/scaffold.py#L448 where it assigns to the variable named "endpoint" by doing options.pop("endpoint", None). Since in our code, we have our current options for the page as methods=["GET"], the options would be {'methods': ['GET']}, and thus the endpoint is None.

6. The next line to be executed in the stack trace is the following line: `self.add_url_rule(rule, endpoint, f, **options)` which is the line just below the assignment to the endpoint. It checks if the setup is finished, to which what follows after is actually executing the `@setupmethod add_url_rule` (https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/app.py#L1309) and specifically the line "if endpoint is None". Because in our case the endpoint was None, it reassigns the endpoint to be the endpoint from the view function. For the `add_url_rule`, if a view function was passed to it, then that view function is registered with the endpoint. What this means is that it connects the URL rule that was defined in the decorator of the `route()` function to the name of that function the decorator is assigned with. In other words, the endpoint is the function name and the view function is the actual function body that we define under the @app.route() decorator that we want to be triggered for a specific endpoint request.

7. In the _endpoint_from_view_func(view_func) from step 6, we assert that the view func is not None, to which it isn't. And because it passes the assertion, we return the view function name property back to the previous stack frame, meaning we are back at the lines

```
8. if endpoint is None:
    a. endpoint = _endpoint_from_view_func(view_func)  # type:
       ignore
```

(link: https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/app.py#L1317).
We follow through with the rest of the function which adds to options, which is a dictionary, as a key the string literal "endpoint" mapping to the variable that endpoint represents. It also assigns a variable named "method" which pops the value of "methods" from the dictionary named "options".

8. Using our specific case as an example, this would mean that "methods" is assigned ["GET"]. Continuing with the same add_url_rule function, it checks if the variable "methods" is None or if it is an instance of a string. If the method was None, it will default to be a GET request, and if somehow the programmer passed the methods as a string, then it will throw a TypeError saying that is not allowed. add_url_rule finishes with the last line of its function by making a dictionary of all the known request types in "methods" to be uppercased. As the stack trace continues, it makes provide_automatic_options to be True and has the variable required_methods assigned to {'OPTIONS'}.

9. The stack trace moves into Werkzeug code because we run the line: https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/app.py#L1354 and is within the initializer for making a Rule instance. It checks to make sure that the passed in URL rule starts with a slash ('/') and builds the state variables for the Rule. It additionally adds "HEAD" into the methods and returns back the Rule instance

10. It then adds the rule to its mapping to keep it known and compiles the rule by compiling the regular expression and storing it. After doing all the preprocessing with adding the rule and the route, the trace steps back out to return f, which is of type T_route.

11.