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/vwong175/cse312_group
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 ⋆★˚⋆.˚☽˚｡⋆✧˚.⋆★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? Please explain this in detail, starting from after the TCP socket is created

  For HTTP parsing, it starts off with the route() function, uses the endpoint that was specified as well as the view function to the given endpoint. In the route() function, you can specify the type of methods that you expect to receive and accommodate the view function to handle, specifically passing in the request method types as strings into the list for the parameter named 'methods'. For example:

```
@app.route('/', methods=['POST', 'GET'])
def user():
```

  This view function named 'user' has two methods that it should be expected to handle, specifically 'POST' and 'GET' but more can be included if needed. By default, if nothing is specified, the default request method type would be 'GET'.

  As seen here, methods in this case would be an option in the formal definition of the .route() setup method.

  This allows functionality of the server to handle different request types of the headers of a request.

- 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 will likely grow beyond the page

```
@setupmethod get(rule: str, **options: t.Any)->
t.Callable[[T_route], T_route]
```

Link: (https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/scaffold.py#L383)

Get is a setup method to provide functionality for the Flask app to handle GET requests. This works for when we use the route() decorator and specify GET as one of the methods in the list of possible request types for that specific route. As can be seen, it calls _method_route on itself, with the arguments being a string literal "GET", the passed in URL rule, as well as options.

From lines 383 to 420 in the aforementioned link, are the other setup methods for the different request types, and they all follow the similar structure of the flask application calling the _method_route method on itself.

`_method_route(self, method: str, rule: str, options: dict) -> t.Callable[[T_route], T_route]`
Link: (https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/scaffold.py#L371)
As seen in the link provided to the following function `_method_route()`, it takes in as parameters: method: str, rule: str, and options: dict. It will raise a TypeError if it sees the string literal "methods" in options to prevent the programmer from continuing further if they don't fix the error. This will only happen if they don't use the `route()` decorator to specify the request types for a given route. Otherwise, `_method_route` will return `self.route(rule, methods=[method], **options)`.

Mentioned previously in our TCP headers report (link to route() function: https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/scaffold.py#L423), it can be seen that if no request type is specified, that the methods parameter will default to "GET" in the route function. The same holds in the `add_url_rule` (link to add_url_rule: https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/app.py#L1309) (with details of how `route()` and `add_url_rule` work in conjunction is explained in our TCP Connections report), if methods is left empty when using the `route()` decorator, naturally it defaults the view function to be handled with GET requests.

`def redirect(location: str, code: int = 302, Response: t.Optional[t.Type["BaseResponse"]] = None) -> "BaseResponse"`
For redirect responses, in a similar fashion, if a user were to make a GET request to some path, we will have to respond with a redirect response. In Flask, there is a redirect function that is provided (link: https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/helpers.py#L266) and this shares similarity to how we handled redirect responses in the earlier homeworks of this class. We would handle the GET request to some path that the user may, and redirect them back elsewhere. For example, in our code, we'll implement something of the like when a user wants to log out:

```
@app.route("/logout")
def logout():
    session.clear()
    return redirect("/")
```

where the idea is the moment, they log out, they will be redirected to the default start page.

Taking a look at the redirect code itself as defined in Flask, it takes in the new URL location which is a string, response code which is defaulted to 302, and a Response. The Response is a Response class to use It will return a BaseResponse type. If there is a current_app, it will use the current_app's redirect method rather than _wz_redirect function. If it uses the current_app's redirect method, it passes the location and the code to it, and if it uses the _wz_redirect function, it passes location, code, as well as the Response.

`Current_app: "Flask" = LocalProxy(_cv_app, "app", unbound_message = _no_app_message_)`
https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/globals.py#L59

Current_app is defined in .globals and it is an instance of the Localproxy in the Werkzeug framework. It has access to information about the app including its configuration. Current_app “points to the application handling the running activity” (https://flask.palletsprojects.com/en/2.2.x/appcontext/).