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

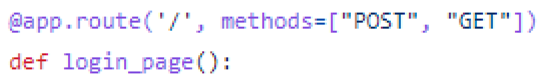
Code Repository	https://github.com/pallets/flask
License Type	MIT
License Description	● Gives users the permission to reuse code for any purpose. Is under the BSD-Style and permissive class of licenses. ● Doesn't require modifications to be open sourced ● Only condition required to use software is to 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

  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 login_page():
  ```

  (Link: https://github.com/vwong175/cse312_group/blob/main/server.py#L25)
  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.

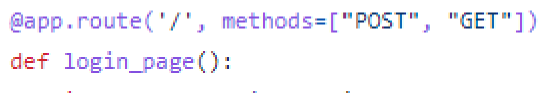
- Where is the specific code that does what you use the tech for? You ***must*** provide a link to the specific file in the repository for your tech with a line number or number range.
  - If there is more than one step in the chain of calls *(hint: there will be)*, you must provide links for the entire 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

Tracing:

—---------------------------------Parsing HTTP Request Headers—---------------------------------

There are multiple times of usage, so we will choose one to represent all the others.
We will use the function `login_page` in our code of server.py to start the tracing.



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

(Link: https://github.com/vwong175/cse312_group/blob/main/server.py#L25)

In this “`@app.route(‘/’, methods=[“POST”,GET])`”, we use library flask/scaffold function “`route`” to get the header target path and allow methods to be received.
Link:

(https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/scaffold.py#L423)

Since we set requirements for receiving methods POST and GET. We call the function "`decorator`" in the route. Which "`decorator`" calls "`setupmethod`" to set up methods allowed for this path "/".
(Link:https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/scaffold.py#L45)

"`setupmethod`" function will further call "`add_url_rule`" to set our required methods for target path, in this case will be POST and GET.
(Link:
https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/scaffold.py#L455)

`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)
During the "`add_url_rule`" function, if we give methods that will fit into the method's condition, it will call on those functions. For example, Get. 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.

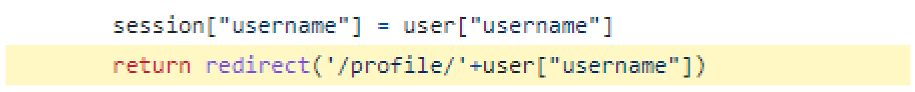
From lines 383 to 420 in the aforementioned link, are the other setup methods for the different request types, and they all follow the similar structure of the flask application calling the _method_route method on itself.

After it setting up the "`add_url_rule`" it returns multiple times to back to our code in server.py by following order: "`add_url_rule`", "`setupmethod`","`decorator`" and "`Route`". Finally return to "`@app.route('/', methods=["POST",GET])`" and run our following functions. Until that request HTTP header parsing for target path "/" is being set up that only allows POST and GET.

—-------------------------------Parsing HTTP Response Headers—----------------------------------

As above mentioned, there are multiple usages so we pick one as an example to represent the HTTP response headers parsing.

We will use ***redirect*** as an example of it.



```
        session["username"] = user["username"]
        return redirect('/profile/'+user["username"])
```

(Link: https://github.com/vwong175/cse312_group/blob/main/models.py#L12)

It called the function "`redirect`" in the flask/helpers library to take a string input as the target redirect location.
(Link:
https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/helpers.py#L266)

Then “`redirect`” will call on another “`redirect`” in the flask/app.py to call on “`_wz_redirect`”.
(Link for redirect in flask/app.py:
https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/app.py#L2041 ;
Link for _wz_redirect :
https://github.com/pallets/flask/blob/cc66213e579d6b35d9951c21b685d0078f373c44/src/flask/app.py#L2053
)

```
def redirect(location: str, code: int = 302, Response:
t.Optional[t.Type[“BaseResponse”]] = None) -> “BaseResponse”
```

(Link for redirect in _wz_redirect :
https://github.com/pallets/werkzeug/blob/main/src/werkzeug/utils.py#L244
)
In the “`_wz_redirect`” calls on another “`redirect`”. This redirect takes “location, code, and response” as input and returns HTTP response as result. In this function, it generates a new “`Response`” value with “***code***” equals 302 and “***location***” as location we are given.

(Link for class “`Response`”:
https://github.com/pallets/werkzeug/blob/main/src/werkzeug/wrappers/response.py#L67 ;
Link for “`__init__`” in “`Response`” :
https://github.com/pallets/werkzeug/blob/main/src/werkzeug/wrappers/response.py#L170)
In the generated new “`Response`”, it takes in input for “status”, “headers” , “mimetype”, and “content_type” and assigns it to belongs. After getting the new “`Response`” it will check its status, and depends on its status set “`self.response`”. This step is similar to what we did in Homework, it gives a response with a redirect message in bytes then returns all the functions back to our code models.py.

All other responses are similar like this one, they take given proper input then use function “`Response`” in werkzeug library to change it to corresponding response in bytes. Then send it to clients.