FLASK - HTTP Header Parsing

General Information & Licensing

Code Repository	http://github.com/pallets/flask
License Type	BSD-3
License Description	Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: ● Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. ● Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. ● Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
License Restrictions	Prohibits others from using the name of the copyright holder or its contributors to promote derived products without written consent.

HTTP Headers Parsing

Magic

In our project, we implement Flask as the web frame and Flask can deal with the http requests parsing. From the github repository of flask, there are functions to parse the header of the http requests. The requests handled in our project are only the GET request and POST request. And the purpose is to parse all these headers and grab required information from them.

- Once the request is received, a class named **Request** with parameter **RequestBase** is to used by default in Flask and handle the request. https://github.com/pallets/flask/blob/a03719b01076a5bfdc2c8f4024eda7b874614bc1/src/flask/wrappers.py#L15
- Inside the class, the function **endpoint** is to return the endpoint that matched the request URL. If the URL has not been performed yet, the function returns None. https://github.com/pallets/flask/blob/a03719b01076a5bfdc2c8f4024eda7b874614bc1/src/flask/wrappers.py#L53
- After the class is set up, flask starts parsing the header. And there are multiple functions to parse the header that can be used. This function **parse_list_header (value: str)** parses comma-separated lists where the elements of the list may include quoted-strings, which means **value** is a string with a list header. It returns a list that is unquoted at the end. https://github.com/pallets/werkzeug/blob/c7ae2fea4fb229ffd71187c2b665874c91b96277/src/werkzeug/http.py#L309
- It imports the function **parse_http_list** as **_parse_list_header** to parse the list above. This function takes a string as parameter, if there is '\\' inside the string, the next character of the string will append the list. If there is a comma or quote in the string, they are not counted if they are escaped and only double-quotes count. At the end, the function returns a list with only double-quotes. https://github.com/python/cpython/blob/64fe34371722d90448e0d1a0c04e7ed106f5f70a/Lib/urllib/request.py#L1434
- The function **parse_dict_header** is to parse lists of key, value pairs and convert them into a python dict. The function is used when the http request contains dict. The parameter **value** is a string with a dict header and **cls** is a callable to use for storage of parsed results. The function parse the **name** and **value**, makes each item in **value** unquoted, pair **name** as key with its corresponding **value** inside a dic and append to the **result** list. At the end, the function returns the **result** list. https://github.com/pallets/werkzeug/blob/c7ae2fea4fb229ffd71187c2b665874c91b96277/src/werkzeug/http.py#L339
- The function **parse_options_header** is to parse a 'Content-Type'-like header into a tuple with value and any options. The return type of the value will be ('text/html', {'charset': 'utf8'}). First, the **result** will be created as an empty tuple. Then in the while loop, the loop will break if the header didn't match the starting type. Otherwise, it will go to the nested loop to find the matched value and return as a tuple. https://github.com/pallets/werkzeug/blob/c7ae2fea4fb229ffd71187c2b665874c91b96277/src/werkzeug/http.py#L379
- The function **parse_accept_header** parses an HTTP Accept-* header. However, this function does not implement a complete valid algorithm but one that supports at least

value and quality. The parameters are **value**, the accept header string to be parsed, and **cls**, the wrapper class for the return value. At the end, it returns an instance of '**cls**'. If **cls** is none, it will cast the **Accept** type to the value(**cls**).https://github.com/pallets/werkzeug/blob/c7ae2fea4fb229ffd71187c2b665874c91b96277/src/werkzeug/http.py#L484

Otherwise, it will look for the quality match in **_accept_re.** https://github.com/pallets/werkzeug/blob/c7ae2fea4fb229ffd71187c2b665874c91b96277/src/werkzeug/http.py#L509

_accept_re is defined as a compiled repository, in which the **compile** function compiles a regular expression pattern, returning a Pattern object.

The above compile object is produced by **_compile** function. This function takes pattern and flags as the parameter and compiles the pattern. If flags belong to an instance, it will be changed into its value. Then try and except can either cache the pattern and flags or pass if keyerror exists. After that, the first compile result will be assigned to **p** and the function will pass if runtime error, key error or iteration stops. Otherwise, it cache to the next iteration of the cache and return the final compile **p**.https://github.com/python/cpython/blob/64fe34371722d90448e0d1a0c04e7ed106f5f70a/Lib/re/__init__.py#L272

- All the header parsing functions above call the function **unquote_header_value** to unquote a header value. This function takes **value**, the header value to unquote, and **is_filename,** the value represents a filename or path. If **value** is quoted by double-quotes, value will be changed to value[1:-1] which does not include the double-quotes. If this is a filename and the starting characters look like a UNC path, then just return the value without quotes. Otherwise, replace "\\\\" with "\\" and replace "\\"" with ""‘ before returning the **value**.
  https://github.com/pallets/werkzeug/blob/c7ae2fea4fb229ffd71187c2b665874c91b96277/src/werkzeug/http.py#L218

@app.route("/", method=['GET'])

To figure out how Flask establishes the routing, we first looked into the function **route()**.

- **route()** has 3 parameters: **self, rule (The URL rule string), options.** It helps decorate a view function to register it with the given URL rule and options. Inside the function, the **decorator** is created and returns to itself later. This decorator function takes in a route callable. The callable can be any function that we call right after the route(). Then, the endpoint will 'pop' out from the options. In the next line, the function **add_url_rule** is called and it will finally return the **view_func**. In this add_url_rule method, it registers a rule for routing incoming requests and building URLs. The 'route' decorator is equivalent to the 'view_func' here, which means the function that we try to call will be returned. If 'view_func' has a 'required_methods' attribute, those methods are added to the passed and automatic method. Remember, endpoint is the endpoint name to associate with the rule and view function, it's set by default as 'view_func.__name__'. The only thing that the add_url_rule does is to raise a NotImplementedError since the method is not implemented at this moment. An erro would be raised if a function has already been registered for the endpoint.https://github.com/pallets/flask/blob/main/src/flask/scaffold.py#L520
- Back to the **route()** method, it finally returns the decorator, which returns the function that we want to implement. The ''methods'' parameter is set to ["GET"] by default. In the example above, we want to only send the response to a request with the GET

heading.https://github.com/pallets/flask/blob/main/src/flask/scaffold.py#L487

return render_template(homepage.html)

- We use the **render_template()** to send the html or any action from the back-end. This method render a template by name with the given context.
- In the function, there are two parameters. One is the template_name_list, which is the name of the template to render. If a list is given, the first name to exist will be rendered. The other parameter is context, which are variables to make available in the template. At first, **app** is set to a Callable, and the template is extended by the **get_or_select_template**. This method can grab the exact form of the template. Then, the function returns to _render(app, template, context).
- The **_render** method takes "Flask" as the content of parameter app then update the template context. The called method updates the template context with some commonly used variables in order to inject everything template context processors want to inject. The variable **names** is set to an iterable in the options of all templates. If there is a update request, the template may be rendered outside a request context with reversed blueprints. In the loop of verifying all the element in **names**, if the element is in the processors of template context. The processor is a dict collecting blue print key and the callables of processor. After the context is updated to the function completely, the original copy will update the new sending context and add it into the dict. The new wrapped template will be sent out and return the render status of the template. In that case, the context inside homepage.html displays in the front-end part due to the updated template dict.