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	git@github.com:weidajiang/CSE312.git
License Type	BSD(flask), MIT(flask-sock)
License Description	According to the site, this basically means that as long as the copyright and name of Flask still exists, and the disclaimer exists, we can use it however we want. This means we can actually reuse it any way we give them credit.
License Restrictions	We can actually reuse it anyway we give them credit.
Who worked with this?	Reference from Flask's

Flask - app.run (Start TCP Server)

Purpose

- Instead of using what we did in the homework to hold the TCP server, the flask uses WSGIServer to hold the server. Flask is based on werkzeug, and werkzeug is a WSGI utility library in Python. (WSGI is abbreviation of Web Server Gateway Interface)
  Below is codes from homework to hold TCP server:

```
host = "0.0.0.0"

port = 8000

with socketserver.ThreadingTCPServer((host,port),MyTCPHandler) as server:

        server.serve_forever()
```

- app.run( ) runs the application on a local development server, we can see below:
- https://github.com/weidajiang/CSE312/blob/main/webapp/app.py

```
270  if __name__ == '__main__':
271      app.run(host="0.0.0.0", port = "8000")
272      #app.run()
```

Magic

When we start the Flask project, the main entry below:
https://github.com/weidajiang/CSE312/blob/main/webapp/app.py
Line 12: `app = Flask(__name__)`
Line 270: `if __name__ == '__main__':`
Line 271: `app.run(host="0.0.0.0", port = "8000")`

Inside the run() flask/app.py from lines range (836 - 956), we see the host, and port.
https://github.com/pallets/flask/blob/main/src/flask/app.py

Line 951: `run_simple(t.cast(str, host), port, self, **options)`

run_simple() from werkzeug/serving.py lines (933 - 1091), a WSGI server, calls make_server() and serve_forever() to start the server.
https://github.com/pallets/werkzeug/blob/main/src/werkzeug/serving.py

make_server() from the same file lines (825 - 867):

```
    """
    if threaded and processes > 1:
        raise ValueError("Cannot have a multi-thread and multi-process server.")

    if threaded:
        return ThreadedWSGIServer(
            host, port, app, request_handler, passthrough_errors, ssl_context, fd=fd
        )

    if processes > 1:
        return ForkingWSGIServer(
            host,
            port,
            app,
            processes,
            request_handler,
            passthrough_errors,
            ssl_context,
            fd=fd,
        )

    return BaseWSGIServer(
        host, port, app, request_handler, passthrough_errors, ssl_context, fd=fd
    )
```

make_server will depend on the number of thread/processor to return WSGI server. By default, that returns BaseWSGIServer(extend from HTTPServer).

serve_forever() from the same file lines (737 - 743):

```
721     def serve_forever(self, poll_interval: float = 0.5) -> None:
722         try:
723             super().serve_forever(poll_interval=poll_interval)
724         except KeyboardInterrupt:
725             pass
726         finally:
727             self.server_close()
```

serve_forever call parent's(HTTPServer) serve_forever() function.

Flask - Basic request parameters

Purpose

- The Request has the ability to read the user input and translate the data into usable data for our code. With this ability we can use Request to help us to retrieve more accurate and security data from the user.

- After Request is called the data we get from the users will be translated into datas with MIME types, headers, x-content-type-options and Path management/routing accordingly, and by having those properties it can help us by letting us have a more effect way of accessing the data.

Magic

- The function will be activated when users send the request in. We can get those data we want such as MIME types, headers, x-content-type-options and Path management/routing etc by getting into the _SansIORequest to find the base of the function.
- The request function located in app.py in Flask, refers to the RequestBase and inherits from _SansIORequest. Then call the codes from line 260-267 to get the content_type:
- https://github.com/pallets/werkzeug/blob/bb21bf90b0b121e3ed45b9950b823e4b43a81fd8/src/werkzeug/sansio/request.py#L260

```
260     content_type = header_property[str](
261         "Content-Type",
262         doc="""The Content-Type entity-header field indicates the media
263         type of the entity-body sent to the recipient or, in the case of
264         the HEAD method, the media type that would have been sent had
265         the request been a GET.""",
266         read_only=True,
267     )
```

From line 270-286 to get content length from the same file:

```
270     def content_length(self) -> t.Optional[int]:
271         """The Content-Length entity-header field indicates the size of the
272         entity-body in bytes or, in the case of the HEAD method, the size of
273         the entity-body that would have been sent had the request been a
274         GET.
275         """
276         if self.headers.get("Transfer-Encoding", "") == "chunked":
277             return None
278
279         content_length = self.headers.get("Content-Length")
280         if content_length is not None:
281             try:
282                 return max(0, int(content_length))
283             except (ValueError, TypeError):
284                 pass
285
286         return None
```

From line 351-358 to get mimetype from the same file:

```
351     def mimetype(self) -> str:
352         """Like :attr:`content_type`, but without parameters (eg, withou
353         charset, type etc.) and always lowercase.  For example if the co
354         type is ``text/HTML; charset=utf-8`` the mimetype would be
355         ``'text/html'``.
356         """
357         self._parse_content_type()
358         return self._parsed_content_type[0].lower()
359
```

From line 119-150 to get headers, methods, path, etc from the same file:

```python
    def __init__(
        self,
        method: str,
        scheme: str,
        server: t.Optional[t.Tuple[str, t.Optional[int]]],
        root_path: str,
        path: str,
        query_string: bytes,
        headers: Headers,
        remote_addr: t.Optional[str],
    ) -> None:
        #: The method the request was made with, such as ``GET``.
        self.method = method.upper()
        #: The URL scheme of the protocol the request used, such as
        #: ``https`` or ``wss``.
        self.scheme = scheme
        #: The address of the server. ``(host, port)``, ``(path, None)``
        #: for unix sockets, or ``None`` if not known.
        self.server = server
        #: The prefix that the application is mounted under, without a
        #: trailing slash. :attr:`path` comes after this.
        #: trailing slash. :attr:`path` comes after this.
        self.root_path = root_path.rstrip("/")
        #: The path part of the URL after :attr:`root_path`. This is the
        #: path used for routing within the application.
        self.path = "/" + path.lstrip("/")
        #: The part of the URL after the "?". This is the raw value, use
        #: :attr:`args` for the parsed values.
        self.query_string = query_string
        #: The headers received with the request.
        self.headers = headers
        #: The address of the client sending the request.
        self.remote_addr = remote_addr

```

About the Flask parsing, with the args function in sansio/request.py, getting the parsed url:(line 170~185)
https://github.com/pallets/werkzeug/blob/main/src/werkzeug/sansio/request.py

```python
    @cached_property
    def args(self) -> "MultiDict[str, str]":
        """The parsed URL parameters (the part in the URL after the question
        mark).

        By default an
        :class:`~werkzeug.datastructures.ImmutableMultiDict`
        is returned from this function.  This can be changed by setting
        :attr:`parameter_storage_class` to a different type.  This might
        be necessary if the order of the form data is important.
        """
        return url_decode(
            self.query_string,
            self.url_charset,
            errors=self.encoding_errors,
            cls=self.parameter_storage_class,
        )

```

In the url parsing, inherits from the url_decode function in urls.py line 819-861:

```
818
819     def url_decode(
820         s: t.AnyStr,
821         charset: str = "utf-8",
822         include_empty: bool = True,
823         errors: str = "replace",
824         separator: str = "&",
825         cls: t.Optional[t.Type["ds.MultiDict"]] = None,
826     ) -> "ds.MultiDict[str, str]":
827         """Parse a query string and return it as a :class:`MultiDict`.
828
829         :param s: The query string to parse.
830         :param charset: Decode bytes to string with this charset. If not
831             given, bytes are returned as-is.
```

With the cookies function in sansio/request.py, getting the cookies for later authentication:(line 247~256)
https://github.com/pallets/werkzeug/blob/main/src/werkzeug/sansio/request.py

```
246     @cached_property
247     def cookies(self) -> "ImmutableMultiDict[str, str]":
248         """A :class:`dict` with the contents of all cookies transmitted with
249         the request."""
250         wsgi_combined_cookie = ";".join(self.headers.getlist("Cookie"))
251         return parse_cookie(  # type: ignore
252             wsgi_combined_cookie,
253             self.charset,
254             self.encoding_errors,
255             cls=self.dict_storage_class,
256         )
257
```

From the request function located in app.py in Flask, refer to the wrappers\request.py, we can find the files function to use the load_from_data function parse the data for later using:(line 413~431 for files function; line 251~282 for load_from_data function)

```
461     def files(self) -> "ImmutableMultiDict[str, FileStorage]":
462         """:class:`~werkzeug.datastructures.MultiDict` object containing
463         all uploaded files.  Each key in :attr:`files` is the name from the
464         ``<input type="file" name="">``.  Each value in :attr:`files` is a
465         Werkzeug :class:`~werkzeug.datastructures.FileStorage` object.
466
467         It basically behaves like a standard file object you know from Python,
468         with the difference that it also has a
469         :meth:`~werkzeug.datastructures.FileStorage.save` function that can
470         store the file on the filesystem.
471
472         Note that :attr:`files` will only contain data if the request method was
473         POST, PUT or PATCH and the ``<form>`` that posted to the request had
474         ``enctype="multipart/form-data"``.  It will be empty otherwise.
475
476         See the :class:`~werkzeug.datastructures.MultiDict` /
477         :class:`~werkzeug.datastructures.FileStorage` documentation for
478         more details about the used data structure.
479         """
480         self._load_form_data()
481         return self.files
```

```
251     def _load_form_data(self) -> None:
252         """Method used internally to retrieve submitted data.  After calling
253         this sets `form` and `files` on the request object to multi dicts
254         filled with the incoming form data.  As a matter of fact the input
255         stream will be empty afterwards.  You can also call this method to
256         force the parsing of the form data.
257
258         .. versionadded:: 0.8
259         """
260         # abort early if we have already consumed the stream
261         if "form" in self.__dict__:
262             return
263
264         if self.want_form_data_parsed:
265             parser = self.make_form_data_parser()
266             data = parser.parse(
267                 self._get_stream_for_parsing(),
268                 self.mimetype,
269                 self.content_length,
270                 self.mimetype_params,
271             )
272         else:
```

At the same location of files function, we can find form function here:

```
413     def form(self) -> "ImmutableMultiDict[str, str]":
414         """The form parameters.  By default an
415         :class:`~werkzeug.datastructures.ImmutableMultiDict`
416         is returned from this function.  This can be changed by setting
417         :attr:`parameter_storage_class` to a different type.  This might
418         be necessary if the order of the form data is important.
419
420         Please keep in mind that file uploads will not end up here, but instead
421         in the :attr:`files` attribute.
422
423         .. versionchanged:: 0.9
424
425             Previous to Werkzeug 0.9 this would only contain form data for POST
426             and PUT requests.
427         """
428         self._load_form_data()
429         return self.form
```

Get into the request.form method, you will notice in the form function _load_form_data function.
(Line 251-282 wrappers\request.py)
https://github.com/pallets/werkzeug/blob/bb21bf90b0b121e3ed45b9950b823e4b43a81fd8/src/werkzeug/wrappers/request.py#L251-282

```
def _load_form_data(self) -> None:
    """Method used internally to retrieve submitted data.  After calling
    this sets `form` and `files` on the request object to multi dicts
    filled with the incoming form data.  As a matter of fact the input
    stream will be empty afterwards.  You can also call this method to
    force the parsing of the form data.

    .. versionadded:: 0.8
    """
    # abort early if we have already consumed the stream
    if "form" in self.__dict__:
        return

    if self.want_form_data_parsed:
        parser = self.make_form_data_parser()
        data = parser.parse(
            self._get_stream_for_parsing(),
            self.mimetype,
            self.content_length,
            self.mimetype_params,
        )
```

In the same file Line 236- 249, the function make_form_data_parser will take the role of parser, and the make_form_data_parser will return a form_data_parser_class.

```
    def make_form_data_parser(self) -> FormDataParser:
        """Creates the form data parser. Instantiates the
        :attr:`form_data_parser_class` with some parameters.

        .. versionadded:: 0.8
        """
        return self.form_data_parser_class(
            self._get_file_stream,
            self.charset,
            self.encoding_errors,
            self.max_form_memory_size,
            self.max_content_length,
            self.parameter_storage_class,
        )
```

That is where the flask does the parsing. For example in the Line(270-291 wergkeug\formparser.py) .There is a multipart parser that split the input multipart form by using the boundary from the header, and extract the important message just like we did in HW2. Beside multiform there are also a lot of functions that help to parse the different kinds of forms.
https://github.com/pallets/werkzeug/blob/bb21bf90b0b121e3ed45b9950b823e4b43a81fd8/src/werkzeug/formparser.py#L270

```python
@exhaust_stream
def _parse_multipart(
    self,
    stream: t.IO[bytes],
    mimetype: str,
    content_length: t.Optional[int],
    options: t.Dict[str, str],
) -> "t_parse_result":
    parser = MultiPartParser(
        self.stream_factory,
        self.charset,
        self.errors,
        max_form_memory_size=self.max_form_memory_size,
        cls=self.cls,
    )
    boundary = options.get("boundary", "").encode("ascii")

    if not boundary:
        raise ValueError("Missing boundary")

    form, files = parser.parse(stream, boundary, content_length)
    return stream, form, files
```

Also, Flask accomplish the parse in _line_parse function from werkzeug/formparser.py as what we did in homework: (line 327~337)

```python
326
327 def _line_parse(line: str) -> t.Tuple[str, bool]:
328     """Removes line ending characters and returns a tuple (`stripped_line`,
329     `is_terminated`).
330     """
331     if line[-2:] == "\r\n":
332         return line[:-2], True
333
334     elif line[-1:] in {"\r", "\n"}:
335         return line[:-1], True
336
337     return line, False
338
```

In our project code, we use request function to take the data we need: (app.py)
https://github.com/weidajiang/CSE312/blob/main/webapp/app.py

- Taking the username and password form user already register before and now need to login:

```python
71      else:
72          AuthenticationToken = generate_token()
73          username = request.form.get("username")
74          password = request.form.get("password")
75          user_info = db.findInfo(username)
76
```

- Taking the username and password we need about the user new registration:

```
app.secret_key = salt
db = MongoDB.mongoDB()
username = request.form.get("NewUsername")
password = request.form.get("NewPassword")
user_info = db.findInfo(username)
if user_info is not None:
    return render_template("signup.html", failed=1)
```

```
if request.form.keys().__contains__("NewUsername"):
    username = request.form.get("NewUsername")
    password = request.form.get("NewPassword")
    hashed_password = hashlib.sha224(password.encode() + salt).hexdi
    db.addInfo(username, hashed_password, salt)
    db.addProfile(username, "N/A", "N/A", "N/A", "N/A", "N/A", "N/A"
```

- Taking the cookies from headers for authentication:

```
cookie = request.cookies.get("userToken")
username = db.findUsernameByCookie(cookie)['username']
info = db.findInfo(username)
salt = info["salt"]
filename = request.files['filename']
file = filename.read()
```

```
        return render_template('Profile.html')
    cookie = request.cookies.get("userToken")
    username = db.findUsernameByCookie(cookie)['username']
```

```
def chat():
    db = MongoDB.mongoDB()
    cookie = request.cookies.get("userToken")
    username = db.findUsernameByCookie(cookie)['username']
    clients[username] = ""
    render_text = []
```

```
def allevents():
    db = MongoDB.mongoDB()
    cookie = request.cookies.get("userToken")
    username = db.findUsernameByCookie(cookie)['username']
    render_text2 = []
```

```
def allusers():
    db = MongoDB.mongoDB()
    cookie = request.cookies.get("userToken")
    username = db.findUsernameByCookie(cookie)['username']
    user = db.findProfile(username)
```

```python
def about():
    db = MongoDB.mongoDB()
    cookie = request.cookies.get("userToken")
    username = db.findUsernameByCookie(cookie)['username']
```

```python
def websocket(socket):
    db = MongoDB.mongoDB()
    cookie = request.cookies.get("userToken")
    username = db.findUsernameByCookie(cookie)['username']
    clients[db.findUsernameByCookie(cookie)['username']] = sock
    while True:
```

- Taking the type of the sending request from headers by using request.method:

```python
    db = MongoDB.mongoDB()
    if request.method == 'GET':
        return render_template('signin.html')
```

```python
    db = MongoDB.mongoDB()
    if request.method == 'GET':
        return render_template('Profile.html')
```

```python
@app.route('/user', methods=["GET", "POST"])
def userPage():
    if request.method == 'GET':
        return render_template("UserPage.html")
    print(1)
```

- Taking the filename and split it with “.” and decode file in order to receive the file data which upload from users and show it as a icon:

```python
    salt = info["salt"]
    filename = request.files['filename']
    file = filename.read()
    if len(file) == 0:
        response = redirect(f"profilePage?username={username}")
        return response
    else:
        type_temp = filename.filename.split(".")[-1]
        salt = salt.decode().replace(".", "").replace("/","")
        name = salt+'.'+type_temp
        f = open("static/user_photo/" + name, 'wb')
        f.write(file)
        f.close()
        db.Update_photo(username, name)
        response = redirect(f"profilePage?username={username}")
```

- Taking email, bio, address… etc information of the user for the profile updated:

```python
username = db.findUsernameByCookie(cookie)['username']
print("update")
db.UpdateProfile(username, request.form.get("email"), request.form.get("sex"), request.form.get("dob"), request.form.get("address"), request.form
response = redirect(f"profilePage?username={username}")
return response
```

- Using request.args to phase url and get the username:

```
116     stored_username = db.findUsernameByCookie(cookie)['usern
117     db = MongoDB.mongoDB()
118     username = request.args.get('username')
119     print(username)
120     info = db.findProfile(username)
```

Flask - Route

Purpose

- To map URL and processing function.
- When the browser client sends a URL request to the webserver, the routing in the server will immediately find the corresponding function to process. It is used throughout the server for declaring path routes and no need to navigate from the homepage.
- Example : (http://127.0.0.1:5000/), this url will take us the hello_word function

```
@app.route('/')
def hello_world():
  return 'Hello World!'
```

Magic

Here is the sample code for registering a path.
https://github.com/weidajiang/CSE312/blob/main/webapp/app.py#L56-87

```python
@app.route('/', methods=["GET", "POST"])
def login():
    salt = bcrypt.gensalt()
    app.secret_key = salt
    db = MongoDB.mongoDB()
    if request.method == 'GET':
        return render_template('signin.html')
    if request.form.keys().__contains__("NewUsername"):
        username = request.form.get("NewUsername")
        password = request.form.get("NewPassword")
        hashed_password = hashlib.sha224(password.encode() + salt).hexdigest()
        db.addInfo(username, hashed_password, salt)
        db.addProfile(username, "N/A", "N/A", "N/A", "N/A", "N/A", "N/A", "bird.gif")
        return render_template("signin.html", successfully="Your account has been created

    else:
        AuthenticationToken = generate_token()
        username = request.form.get("username")
        password = request.form.get("password")
        user_info = db.findInfo(username)

        if user_info is None:
            return render_template("signin.html", failed=1)
        salt, stored_password = user_info["salt"], user_info["password"]
        hashed_password = hashlib.sha224(password.encode() + salt).hexdigest()
        if hashed_password == stored_password:...
        else:
            return render_template("signin.html", failed=2)

```

In Flask, they use decorator to add a url rule to the requestContext (flask.scaffold.py)
https://github.com/pallets/flask/blob/main/src/flask/scaffold.py

```python
    def route(self, rule: str, **options: t.Any) -> t.Callable[[F], F]:
        """Decorate a view function to register it with the given URL
        rule and options. Calls :meth:`add_url_rule`, which has more
        details about the implementation.

        .. code-block:: python

            @app.route("/")
            def index():
                return "Hello, World!"

        See :ref:`url-route-registrations`.

        The endpoint name for the route defaults to the name of the view
        function if the ``endpoint`` parameter isn't passed.

        The ``methods`` parameter defaults to ``["GET"]``. ``HEAD`` and
        ``OPTIONS`` are added automatically.

        :param rule: The URL rule string.
        :param options: Extra options passed to the
            :class:`~werkzeug.routing.Rule` object.
        """

        def decorator(f: F) -> F:
            endpoint = options.pop("endpoint", None)
            self.add_url_rule(rule, endpoint, f, **options)
            return f

        return decorator
```

Inside the add_url_rule (flask/app.py 1038 - 1094). In line 1086,

```
1079
1080        # Add the required methods now.
1081        methods |= required_methods
1082
1083        rule = self.url_rule_class(rule, methods=methods, **options)
1084        rule.provide_automatic_options = provide_automatic_options  # type: ignore
1085
1086        self.url_map.add(rule)
```

flask will set the route rule(Werkzeug.routing.Rule) to the url_map(werkzeug.routing:Map) endpoint, and view_function(python dict)
https://github.com/pallets/flask/blob/main/src/flask/app.py
In the flask.app (1755 - 1790), this creates a url adapter based on how many endpoints we create in the project.

```
754
755     def create_url_adapter(
756         self, request: t.Optional[Request]
757     ) -> t.Optional[MapAdapter]:
758         """Creates a URL adapter for the given request. The URL adapter
759         is created at a point where the request context is not yet set
760         up so the request is passed explicitly.
761
762         .. versionadded:: 0.6
763
764         .. versionchanged:: 0.9
765            This can now also be called without a request object when the
766            URL adapter is created for the application context.
767
768         .. versionchanged:: 1.0
769             :data:`SERVER_NAME` no longer implicitly enables subdomain
770             matching. Use :attr:`subdomain_matching` instead.
771         """
772         if request is not None:
773             # If subdomain matching is disabled (the default), use the
774             # default subdomain in all cases. This should be the default
775             # in Werkzeug but it currently does not have that feature.
776             if not self.subdomain_matching:
777                 subdomain = self.url_map.default_subdomain or None
778             else:
779                 subdomain = None
780
781             return self.url_map.bind_to_environ(
782                 request.environ,
783                 server_name=self.config["SERVER_NAME"],
784                 subdomain=subdomain,
785             )
786         # We need at the very least the server name to be set for this
```

The main loinc of the routing is implemented in the werkzeug module. In werkzeug.routing.py line (2209 - 2332), build function binds the routing path to a specific

endpoint through m.bind and matches the URL through urls.match. Under normal circumstances, the corresponding endpoint name and parameter dictionary are returned, and redirection or 404 exceptions may be reported.
https://github.com/pallets/werkzeug/blob/bb21bf90b0b121e3ed45b9950b823e4b43a81fd8/src/werkzeug/routing.py#L2209-2332

```
def build(
    self,
    endpoint: str,
    values: t.Optional[t.Mapping[str, t.Any]] = None,
    method: t.Optional[str] = None,
    force_external: bool = False,
    append_unknown: bool = True,
    url_scheme: t.Optional[str] = None,
) -> str:
    """Building URLs works pretty much the other way round.  Instead of
    `match` you call `build` and pass it the endpoint and a dict of
    arguments for the placeholders.

    The `build` function also accepts an argument called `force_external`
    which, if you set it to `True` will force external URLs. Per default
    external URLs (include the server name) will only be used if the
    target URL is on a different subdomain.

    >>> m = Map([
    ...     Rule('/', endpoint='index'),
    ...     Rule('/downloads/', endpoint='downloads/index'),
    ...     Rule('/downloads/<int:id>', endpoint='downloads/show')
    ... ])
    >>> urls = m.bind("example.com", "/")
    >>> urls.build("index", {})
    '/'
    >>> urls.build("downloads/show", {'id': 42})
    '/downloads/42'
    >>> urls.build("downloads/show", {'id': 42}, force_external=True)
    'http://example.com/downloads/42'
```

Then, in the match function, werkzeug use regular expression to match the path.
https://github.com/pallets/werkzeug/blob/bb21bf90b0b121e3ed45b9950b823e4b43a81fd8/src/werkzeug/routing.py#L903

```
        # a \n by WSGI.
        regex = rf"^{''.join(regex_parts)}{tail}$\Z"
        self._regex = re.compile(regex)

    def match(
        self, path: str, method: t.Optional[str] = None
    ) -> t.Optional[t.MutableMapping[str, t.Any]]:
        """Check if the rule matches a given path. Path is a string in the
        form ``"subdomain|/path"`` and is assembled by the map.  If
        the map is doing host matching the subdomain part will be the host
        instead.

        If the rule matches a dict with the converted values is returned,
        otherwise the return value is `None`.

        :internal:
        """
        if not self.build_only:
            require_redirect = False

            m = self._regex.search(path)
            if m is not None:
                groups = m.groupdict()
                # we have a folder like part of the url without a trailing
                # slash and strict slashes enabled. raise an exception that
                # tells the map to redirect to the same url but with a
                # trailing slash
                if (
                    self.strict_slashes
                    and not self.is_leaf
                    and not groups.pop("__suffix__")
                    and (
                        method is None or self.methods is None or method in self.methods
                    )
                ):
                    path += "/"
                    require_redirect = True
                # if we are not in strict slashes mode we have to remove
                # a __suffix__
                elif not self.strict_slashes:
                    del groups["__suffix__"]
```

Flask - render_template

Purpose

- In order to send the html file back to the frontend and use the template engine(we learn in the class), we decided to use the render_template function in the flask framework.

Magic

Usage:
https://github.com/weidajiang/CSE312/blob/main/webapp/app.py

```
190     @app.route("/about")
191     def about():
192         db = MongoDB.mongoDB()
193         cookie = request.cookies.get("userToken")
194         username = db.findUsernameByCookie(cookie)['username']
195         return render_template("AboutUs.html",username=username)
196
197
```

https://github.com/weidajiang/CSE312/blob/main/webapp/templates/AboutUs.html

```
30      <li class="nav-item">
31          <a class="nav-link" href="{{ url_for('profilePage')}}?username={{username}}">Profile</a>
32      </li>
33      <li class="nav-item">
```

In the about function (path:/about), this will load the AboutUs.html under the template folder, and assign username to a value “username”. In the AboutUs.html, based on the template engine, the about us page will replace the username with actual value.

Magic behind the scene:
Flask uses Jinja2 as its template engine. In flask/templating.py, the render_template function provides a very detailed comment, which will render a template from the template folder with given context.
https://github.com/pallets/flask/blob/main/src/flask/templating.py

```python
def render_template(
    template_name_or_list: t.Union[str, Template, t.List[t.Union[str, Template]]],
    **context: t.Any
) -> str:
    """Renders a template from the template folder with the given
    context.

    :param template_name_or_list: the name of the template to be
                                  rendered, or an iterable with template names
                                  the first one existing will be rendered
    :param context: the variables that should be available in the
                    context of the template.
    """
    ctx = _app_ctx_stack.top
    ctx.app.update_template_context(context)
    return _render(
        ctx.app.jinja_env.get_or_select_template(template_name_or_list),
        context,
        ctx.app,
    )

```

In the Flask (flask/app.py lines 396 - 539), line 404 specifies the source folder.
https://github.com/pallets/flask/blob/main/src/flask/app.py

```python
    def __init__(
        self,
        import_name: str,
        static_url_path: t.Optional[str] = None,
        static_folder: t.Optional[t.Union[str, os.PathLike]] = "static",
        static_host: t.Optional[str] = None,
        host_matching: bool = False,
        subdomain_matching: bool = False,
        template_folder: t.Optional[str] = "templates",
        instance_path: t.Optional[str] = None,
        instance_relative_config: bool = False,
        root_path: t.Optional[str] = None,
    ):
        super().__init__(
```

We also see that Flask object assign a jinja environment. (line 218)

```python
    #: The class that is used for the Jinja environment.
    #:
    #: .. versionadded:: 0.11
    jinja_environment = Environment
```

In Jinja2/Environment.py, line 1084 calls select_template.
https://github.com/pallets/jinja/blob/main/src/jinja2/environment.py#L1066-1084

```
@internalcode
def get_or_select_template(
    self,
    template_name_or_list: t.Union[
        str, "Template", t.List[t.Union[str, "Template"]]
    ],
    parent: t.Optional[str] = None,
    globals: t.Optional[t.MutableMapping[str, t.Any]] = None,
) -> "Template":
    """Use :meth:`select_template` if an iterable of template names
    is given, or :meth:`get_template` if one name is given.

    .. versionadded:: 2.3
    """
    if isinstance(template_name_or_list, (str, Undefined)):
        return self.get_template(template_name_or_list, parent, globals)
    elif isinstance(template_name_or_list, Template):
        return template_name_or_list
    return self.select_template(template_name_or_list, parent, globals)
```

In the same file, line 1047 - 1064, calls load_template.

```
1035            .. versionadded:: 2.3
1036            """
1037            if isinstance(names, Undefined):
1038                names._fail_with_undefined_error()
1039
1040            if not names:
1041                raise TemplatesNotFound(
1042                    message="Tried to select from an empty list of templates."
1043                )
1044
1045            for name in names:
1046                if isinstance(name, Template):
1047                    return name
1048                if parent is not None:
1049                    name = self.join_path(name, parent)
1050                try:
1051                    return self._load_template(name, globals)
1052                except (TemplateNotFound, UndefinedError):
1053                    pass
1054            raise TemplatesNotFound(names)  # type: ignore
```

Here is the load_template function in the same file from line 951-973

```python
@internalcode
def _load_template(
    self, name: str, globals: t.Optional[t.MutableMapping[str, t.Any]]
) -> "Template":
    if self.loader is None:
        raise TypeError("no loader for this environment specified")
    cache_key = (weakref.ref(self.loader), name)
    if self.cache is not None:
        template = self.cache.get(cache_key)
        if template is not None and (
            not self.auto_reload or template.is_up_to_date
        ):
            # template.globals is a ChainMap, modifying it will only
            # affect the template, not the environment globals.
            if globals:
                template.globals.update(globals)

            return template

    template = self.loader.load(self, name, self.make_globals(globals))

    if self.cache is not None:
        self.cache[cache_key] = template
    return template
```

The load_template() method will first check whether there is a cache, if the cache is available, use the cache; if the cache is not available, use the loader to load the template.

Also, a lot of the functions are implemented in the jinja2/parser.py function, https://github.com/pallets/jinja/blob/main/src/jinja2/parser.py such as parse_if, parse_tuple, parse_list, parse_statement and so on. We can use these keywords on the html page.

Flask_Sock

Purpose

- The purpose of using the library is to build up the websocket handshake, after the handshake the clients and server can send messages to each other through the flask_socketio.
- Live chat interaction

Line 204 in app.py
https://github.com/weidajiang/CSE312/blob/main/webapp/app.py#L204

```
@sock.route('/websocket')
def websocket(socket):
    db = MongoDB.mongoDB()
    cookie = request.cookies.get("userToken")
    username = db.findUsernameByCookie(cookie)['username']
    clients[db.findUsernameByCookie(cookie)['username']] = socket
```

- Get into the route function of the flask_sock, you can find there is a variable name ws in the decorator function.
line(53-92 in flask_sock__init__.py):
https://github.com/miguelgrinberg/flask-sock/blob/a390d5705537b0cb127280fe2a41b59e92990d66/src/flask_sock/__init__.py#L53-92

```
        """
        def decorator(f):
            @wraps(f)
            def websocket_route(*args, **kwargs):  # pragma: no cover
                ws = Server(request.environ, **current_app.config.get(
                    'SOCK_SERVER_OPTIONS', {}))
```

- Take a deep look into the Server which is from ws.py, there is a handshake function, the handshake uses the instance of class name self.ws to receive data.
Line(340-351, simple_websocket\ws.py):
https://github.com/miguelgrinberg/simple-websocket/blob/main/src/simple_websocket/ws.py#L340-351

```
    def handshake(self):
        out_data = self.ws.send(Request(host=self.host, target=self.path))
        self.sock.send(out_data)

        in_data = self.sock.recv(self.receive_bytes)
        self.ws.receive_data(in_data)
        event = next(self.ws.events())
        if isinstance(event, RejectConnection):  # pragma: no cover
            raise ConnectionError(event.status_code)
        elif not isinstance(event, AcceptConnection):  # pragma: no cover
            raise ConnectionError(400)
        self.connected = True
```

- Keep exploring what exactly is the self.ws, you can find out that there is an instance name handshake in the init function of the class WSconnection.

Line(22-31, wsporoto__init__.py):
https://github.com/python-hyper/wsproto/blob/master/src/wsproto/__init__.py#L22-31

```
def __init__(self, connection_type: ConnectionType) -> None:
    """
    Constructor

    :param wsproto.connection.ConnectionType connection_type: Controls
        whether the library behaves as a client or as a server.
    """
    self.client = connection_type is ConnectionType.CLIENT
    self.handshake = H11Handshake(connection_type)
    self.connection: Optional[Connection] = None
```

- In the H11handShake class you will notice that you will feel familiar, because it is exactly what we learned in the class, the 101 Switching Protocols, sec-websocket-extensions, upgrade, etc. That all we need to build up the websocket handshake
  Line(137-148, wsporoto\handshake.py):
  https://github.com/python-hyper/wsproto/blob/master/src/wsproto/handshake.py#L137-148

```
            break

        if self.client:
            if isinstance(event, h11.InformationalResponse):
                if event.status_code == 101:
                    self._events.append(self._establish_client_connection(ev
                else:
                    self._events.append(
                        RejectConnection(
                            headers=list(event.headers),
                            status_code=event.status_code,
                            has_body=False,
                        )
                    )
```

- Line(196-215, wsporoto\handshake.py)
  https://github.com/python-hyper/wsproto/blob/master/src/wsproto/handshake.py#L196-215

```
        for name, value in event.headers:
            name = name.lower()
            if name == b"connection":
                connection_tokens = split_comma_header(value)
            elif name == b"host":
                host = value.decode("ascii")
                continue  # Skip appending to headers
            elif name == b"sec-websocket-extensions":
                extensions = split_comma_header(value)
                continue  # Skip appending to headers
            elif name == b"sec-websocket-key":
                key = value
            elif name == b"sec-websocket-protocol":
                subprotocols = split_comma_header(value)
                continue  # Skip appending to headers
            elif name == b"sec-websocket-version":
                version = value
            elif name == b"upgrade":
                upgrade = value
            headers.append((name, value))
```

- For the communication you can find in the wsproto.frame_protocol, in the file there is a lot of bit manipulation going on, for example getting the fin bits, mask, decoding the message and buffer just like what we did in HW3.

  Line(25-32, wsporoto\frame_protocol.py)
  https://github.com/python-hyper/wsproto/blob/master/src/wsproto/frame_protocol.py#L25-32

```
def process(self, data: bytes) -> bytes:
    if data:
        data_array = bytearray(data)
        a, b, c, d = (_XOR_TABLE[n] for n in self._masking_key)
        data_array[::4] = data_array[::4].translate(a)
        data_array[1::4] = data_array[1::4].translate(b)
        data_array[2::4] = data_array[2::4].translate(c)
        data_array[3::4] = data_array[3::4].translate(d)
```

- Line(253-258, wsporoto\frame_protocol.py)

```
class Buffer:
    def __init__(self, initial_bytes: Optional[bytes] = None) -> None:
        self.buffer = bytearray()
        self.bytes_used = 0
        if initial_bytes:
            self.feed(initial_bytes)
```

- Also there is the packing part, after decoding the message from the user we have to encode the message and send back to the front-end, and you find the send method in the same file.
- https://github.com/python-hyper/wsproto/blob/master/src/wsproto/frame_protocol.py#L25-32
  Line((591-612, wsporoto\frame_protocol.py)

```
    def send_data(
        self, payload: Union[bytes, bytearray, str] = b"", fin: bool = True
    ) -> bytes:
        if isinstance(payload, (bytes, bytearray, memoryview)):
            opcode = Opcode.BINARY
        elif isinstance(payload, str):
            opcode = Opcode.TEXT
            payload = payload.encode("utf-8")
        else:
            raise ValueError("Must provide bytes or text")

        if self._outbound_opcode is None:
            self._outbound_opcode = opcode
        elif self._outbound_opcode is not opcode:
            raise TypeError("Data type mismatch inside message")
        else:
            opcode = Opcode.CONTINUATION

        if fin:
            self._outbound_opcode = None

        return self._serialize_frame(opcode, payload, fin)
```

- It will return the _serilize_frame function, this is the function that appends the payload length and payload into the websocket frame and returns.
- https://github.com/python-hyper/wsproto/blob/master/src/wsproto/frame_protocol.py#L25-32
  Line(294-324, wsporoto\frame_protocol.py)

```
    def _serialize_frame(
        self, opcode: Opcode, payload: bytes = b"", fin: bool = True
    ) -> bytes:
        rsv = RsvBits(False, False, False)
        for extension in reversed(self.extensions):
            rsv, payload = extension.frame_outbound(self, opcode, rsv, payload

        fin_rsv_opcode = self._make_fin_rsv_opcode(fin, rsv, opcode)

        payload_length = len(payload)
        quad_payload = False
        if payload_length <= MAX_PAYLOAD_NORMAL:
            first_payload = payload_length
            second_payload = None
        elif payload_length <= MAX_PAYLOAD_TWO_BYTE:
            first_payload = PAYLOAD_LENGTH_TWO_BYTE
            second_payload = payload_length
        else:
            first_payload = PAYLOAD_LENGTH_EIGHT_BYTE
            second_payload = payload_length
            quad_payload = True
```

Flask - redirect

Purpose

- The purpose for using the flask.redirect is to redirect from one url to other. In our project we usually use redirect to redirect the signup page to our login page, and login page to user page.

Magic

- Line 267 in app.py
- https://github.com/weidajiang/CSE312/blob/main/webapp/app.py#L267

```
    response = redirect("/")
    response.set_cookie("userToken", "InvalidCookie", max_age=3600)
    return response
```

- Get into the redirect function, you will notice the redirect function take a location as the input(target url), and set the status code to 302 Found for the redirection
- Then it will create a response class, and set the status code into 302, and location will be the input location by edit the header, and lastly return the response.
  Line(325 -331, sansio\response.py)
  https://github.com/pallets/werkzeug/blob/main/src/werkzeug/sansio/response.py#L325-331

```
location = header_property[str](
    "Location",
    doc="""The Location response-header field is used to redirect
    the recipient to a location other than the Request-URI for
    completion of the request or identification of a new
    resource.""",
)
```

- And lastly, return the response to the browser.
  Line(288-289, werkzeug\utilis.py)
  https://github.com/pallets/werkzeug/blob/main/src/werkzeug/utils.py#L288

```
response.headers["Location"] = location
return response
```