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.

Framework: Flask

General Information & Licensing

Code Repository	https://github.com/pallets/flask
License Type	Flask is distributed under the 3-clause BSD license.
License Description	• The BSD license places minimal restrictions on future behavior. This allows BSD code to remain Open Source or become integrated into commercial solutions, as a project's or company's needs change. • Since the BSD license does not come with the legal complexity of the GPL or LGPL licenses, it allows developers and companies to spend their time creating and promoting good code rather than worrying if that code violates licensing. • The BSD license does not become a legal time bomb at any point in the development process.
License Restrictions	• Neither the name of Flask nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
Who worked with this?	• Shawn, Andy, Ryan, Kevin

Use as many of the sections below as needed, or create more, to explain every function, method, class, or object type you used from this library/framework.

class SecureCookieSession(CallbackDict, SessionMixin):

Purpose

- Session objects work like cookies and the object contains a dictionary object that stores session variables.
- When a user logs in to the server, it is given a session id, when the user logs out of the server the session is cleared.
- Throughout our project, we use sessions to store CSRF tokens and usernames.

Magic

- Flask cryptographically signs the session cookies and is set with a secret key. It means one can view the contents of the cookie but can't modify it unless they have the secret key.
- https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/sessions.py#L48

after_request(self, f: AfterRequestCallable) -> AfterRequestCallable:

Purpose

- It registers a function to run after each request.
- We use it to add security headers, such as X-Content-Type-Options = nosniff onto all responses.
- In our project: Line 43.

Magic

- It uses the @after_request decorator to register functions to run at the end of each request.
- def after_request()
- https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/scaffold.py#L558
- self.after_request_funcs
- Data structure that stores after request function
- https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/scaffold.py#L178

class Request(RequestBase)

Purpose

- We used the request object created by Flask to get the request type (GET, POST, etc) and create a response accordingly. We also use the request object to get data about a file upload and the socket ids of each client.
- For example: request.method, request.files, request.sid, request.form
- It is used everywhere in app.py because it has to check if the user is posting data or just getting the web page.

Magic ★★ ° ˙ ˚ ☽ ˚ ⌒ ˚ ★ ≡ ✧ ★ ⌯

- request calls object LocalProxy to get the top of the stack which is abstracted off of object Local which acts as basically a session and gets the latest request/session object.
    - https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/wrappers.py#L16
    - https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/globals.py#L55
    - https://github.com/pallets/werkzeug/blob/main/src/werkzeug/local.py#L329

- Flask Request objects are abstracted from the werkzeug.wrappers library.
    - https://github.com/pallets/werkzeug/blob/main/src/werkzeug/sansio/request.py#L131

url_for(endpoint: str, **values: t.Any) -> str:

Purpose

- The *url_for* function generates a URL to the given endpoint with the method provided.
- Throughout our project, we used this function to generate URLs for us used exclusively as a parameter for the Flask redirect function. Using the *url_for* function in conjunction with the redirect function allowed us to easily attain the URLs for our different pages used in the website by name like home, settings, register, etc. It is also used to render our static files in the HTML such as our CSS and JS file.
- The first time we used this function in our project can be found at the following link:
    - https://github.com/shawnz99/CSE312-GroupProject/blob/main/app/app.py#L58

Magic

- When the *url_for* function is called with the name of a page as the argument it will return the URL for that specific page. After the TCP socket connection is established every time a redirect function call is made in our app.py file, the URL generated from the *url_for* function serves as the argument passed into the redirect function.
- The *url_for* function is located on line 192 of the helpers.py file in the Flask framework. Generally, assuming an error hasn't occurred the function will create a variable, *appctx* which uses *_app_ctx_stack.top* this uses the LocalStack class located in the Werkzeug library. The *top* function is defined in the local.py file of the Werkzeug library and it keeps a stack of objects, the first object is returned from this function.
- Once *appctx* is updated with the top item in the stack it can use the *url_adapter* function to create an object in which the *url_quote* function can be called. This *url_quote* function is found in the Werkzeug library in the urls.py file on line 546. This function will URL encode a certain string as an argument. The function uses the variable *anchor* as the argument, this variable is taken from the *url_for* function argument (**values).

- The *url_for* function is defined at the following link:
    - https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/helpers.py#L192
- The *url_for* function uses multiple different libraries to accomplish the building of the URL to return. The first file used is the Flask/globals.py file.
    - https://github.com/pallets/flask/blob/main/src/flask/globals.py#L53
- In the Werkzeug library, *top* is defined in the local.py file.
    - https://github.com/pallets/werkzeug/blob/main/src/werkzeug/local.py#L142
- Lastly, the *url_quote* function is defined in the urls.py file of Werkzeug
    - https://github.com/pallets/werkzeug/blob/main/src/werkzeug/urls.py#L546

redirect(location: str, code: int = 302, Response: t.Optional[t.Type["Response"]] = None) -> "Response":

Purpose

- The *redirect* function will redirect the client from one page to another using the destination page URL as its argument.
- The *redirect* function is used throughout our project in conjunction with the *url_for* function. We used the *url_for* function to grab the URL for the destination page of our website as the argument, then the *redirect* function returns the response headers to the destination page which we can then render to the client or user using the *render_template* function. Examples of a redirect could be after a user creates a new account a redirect to the login page should fire off.
- The first time we used this function in our project can be found at the following link:
    - https://github.com/shawnz99/CSE312-GroupProject/blob/main/app/app.py#L56

Magic ★★ ° ˖ ° ☽ ° ⁀ ˖ ° ★ ≡ ★ ˖

- When the *redirect* function is called the first thing Flask does is import the function from werkzeug.utils. Once a client has established a TCP socket connection with the server, if the *redirect* function is called it will return a response object that contains a valid HTTP response that we can send back to the client.
- The *redirect* function first takes the URL (location is the argument name). The first thing the *redirect* function does is import the HTML library and use the *escape* function to escape any HTML characters for safety. Lines 256 - 265 of the utils.py file show the manual construction of the HTTP response headers.
- To obtain the valid URL the *redirect* function uses a function Werkzeug defined in the urls.py file called *iri_to_url*. This function parses the inputted location and creates a valid URL for the browser to redirect to.
- Once this string called “response” is generated it is returned from the function and can be used to send the client to the new page. The following line uses the location argument to set the header for the new location:
    - *response.headers["Location"] = location*

- Flask imports the *redirect* function directly from Werkzeug in the __init__.py file:
    - https://github.com/pallets/flask/blob/main/src/flask/__init__.py#L4
- The *redirect* function is defined in the Werkzeug library at the following link:
    - https://github.com/pallets/werkzeug/blob/main/src/werkzeug/utils.py#L221
- When the *redirect* function generates the valid URL it calls the *iri_to_url* function located in the urls.py file:
    - https://github.com/pallets/werkzeug/blob/347fdbb055c86efe1fd49546bd524cde4b98c103/src/werkzeug/urls.py#L752
- The *iri_to_url* function also uses the *url_parse* function which adds a scheme if the URL is lacking one. This function is in the same file and is found at the following link:
    - https://github.com/pallets/werkzeug/blob/347fdbb055c86efe1fd49546bd524cde4b98c103/src/werkzeug/urls.py#L456

route(self, rule: str, **options: t.Any) -> t.Callable

Purpose

- It routes a given URL path and associates it with a view function that provides the response for the request, defaults to GET, but can also use other methods: POST, PUT, DELETE.
- In our project, it routes HTTP requests such as “/” to a function home() that returns a response. (Lines 46, 68, 99, 131, 143)

Magic

- After the TCP socket connection, when a request is sent from the client, our server will look for a list of routes and the route function routes the URL request to our function that handles the request and returns a response
- The route function decorates a view function to register it with the given URL rule and options. It calls *add_url_rule*, which contains most functionality without the decorator notation. The URL rule is bound to the view function and the output of the view function will render in the browser. Flask doesn’t directly bind the URL with the view function, it uses an endpoint in between. An endpoint is an identifier that determines which function will handle the request.
- Ex. localhost:8000/hello ;
  @app.route(“/hello”)
  def hello_world():

The route function will bind the URL “/hello” with the endpoint “hello_world”.
Then the endpoint will use the view function “hello_world” to handle the request.

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

Line 413: def route

Line 445: def add_url_rule

Line 512: def endpoint

Class flask.Config(root_path, defaults=None)

Purpose

- Works exactly like a dict but provides ways to fill it from files or special dictionaries
- Used at the beginning of the app.py file on 23 27 and 28 to set up the secret key and uploads file
- The secret key is used to sign sesion cookies for protection against cookie tampering
- The upload folder is for the picture uploads

Magic ★★ ° ˙ ˚ ☽ ˚ ⌒ ˚ ★ ≡ ✧ ≫

- Flask repo https://github.com/pallets/flask
- After the tcp connection .config is called with ‘SECRET_KEY’ and ‘UPLOAD_FOLDER’ to add it to the immutable dict. After its called it goes flask/app.py for in the flask library and calles make_config(). This is used to create the config attribute by the flask constructor it is passed the ‘instance_relative’ parameter from the constructor of flask, https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/app.py#L420 What that parameter does is it indicates if the config should be relative to the instance path or the root path of the app. Then with the return statement of make_config() it calles config_class() with the root path of the application https://github.com/pallets/flask/blob/ea93a52d7d94ba093bbce4680c622cc4fc9771d8/src/flask/app.py#L612 . At this point it sets whatever was passed in, so in our case it sets the secret_key and the upload_folder to the config dict https://github.com/pallets/flask/blob/ea93a52d7d94ba093bbce4680c622cc4fc9771d8/src/flask/app.py#L232

render_template(template_name_or_list: t.Union[str, t.List[str]], **context: t.Any) -> str:

Purpose

- To render HTML templates with variables passed in to generate content.
- In our project, it allows us to dynamically render every online user and different DM forms for each user in the HTML template sent in the response.
- We use it to render all our web pages, it is also used to render flash() messages and session data. (Lines 52, 58, 96, 128, 164)

Magic

- Flask uses Jinja2, a python HTML templating engine.
- It renders a given HTML template from the template folder with the given context (variables available to the template). update_template_context() also adds additional commonly used variables such as request, session, config, and g in the template.
- After Flask calls Jinja's template.render in the function _render(), it will return the rendered template.

https://github.com/pallets/flask/blob/main/src/flask/templating.py
Line 124: def _render()
Line 133: def render_template()
https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/app.py#L731
Line 731: def update_template_context()

- Flask uses the Environment class to load templates as a template class.
- In Flask, Jinja2 is configured to automatically escape any data rendered in the HTML templates. This means it is safe to render user input. They import markup to escape HTML. Line 1074: https://github.com/alex-foundation/jinja2/blob/b7d13f278753d057bb3765b4d4a672c351d88bf3/jinja2/environment.py#L1074
- Jinja syntax is similar to Python. Anything between {{ and }} is an expression that will be outputted and anything with {% and %} indicates control flow statements like *if* and *for* loops.
- Template variables passed in the context can also use (.) dot and subscript syntax.

https://github.com/alex-foundation/jinja2/blob/master/jinja2/environment.py

Line 954: def render

flash(message: str, category: str = "message") -> None:

Purpose

- This function is used to show feedback messages to the user on the frontend. For example, we flash certain messages when users register or login and or when an error occurs.
- In the server code, we flash the message then use redirect and url_for to show the same page but with the feedback messages.
- It is used specifically in the registration, login, logout, settings flask app routes in app.py.

Magic ★★ ° ˙ ˚ ☽ ˚ ⌒ ˚ ★ ✧ ✧ ✧

- The function first starts by fetching all flash messages that may currently be in the session that hasn't been shown to the user (Line 387).
  - https://github.com/pallets/flask/blob/main/src/flask/helpers.py#L365
- The session in `session.get("_flashes", [])` calls the werkzeug.local LocalProxy library and looks up the session object with `partial(_lookup_req_object, "session")`
  - https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/globals.py#L56
- _lookup_req_object(name) references the werkzeug.local LocalStack library (L31) and fetches the top of the stack with top(self) (L142). After it returns the object at top of the stack, the built-in python function `getattr` is called to get the session object.
  - https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/globals.py#L30
  - https://github.com/pallets/werkzeug/blob/main/src/werkzeug/local.py#L142
- After the session object is fetched, the message that we passed in initially gets appended to the flash messages array. message_flashed.send is then called. message_flashed is a reference to the Namespace class which creates a _FakeSignal class.
  - https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/signals.py#L10
- On the front end, get_flashed_messages() is called which repeats the above steps regarding the top of the stack and session object and returns the flash messages array.
  - https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/src/flask/helpers.py#L397
- On the front end, the templating loops through the message array and HTML elements are created accordingly.

request.files['file'].save(self, dst, buffer_size=16384):

Purpose

- It saves the file uploaded by the user to a specified file path.
- It is used in line 159 of app.py.

Magic ⋆★˚｡⋆୨୧˚ ☽ ˚⁀➷｡˚★彡⋆ ༄

- We call the requests.file method on line 151. This is a method from the werkzeug.wrapper library.
  - https://github.com/pallets/werkzeug/blob/main/src/werkzeug/wrappers/request.py#L461
- The files method calls the _load_form_data() on itself which is a FileStorage object.
  - https://github.com/pallets/werkzeug/blob/347fdbb055c86efe1fd49546bd524cde4b98c103/src/werkzeug/datastructures.py#L2888
- _load_form_data() calls self._get_stream_for_parsing() to get the form data.
  - https://github.com/pallets/werkzeug/blob/347fdbb055c86efe1fd49546bd524cde4b98c103/src/werkzeug/wrappers/request.py#L267
- _get_stream_for_parsing() returns self.stream which is another method.
  - https://github.com/pallets/werkzeug/blob/347fdbb055c86efe1fd49546bd524cde4b98c103/src/werkzeug/wrappers/request.py#L294
- stream(self) calls get_input_stream() to finally initiate the stream.
  - https://github.com/pallets/werkzeug/blob/347fdbb055c86efe1fd49546bd524cde4b98c103/src/werkzeug/wrappers/request.py#L337
  - https://github.com/pallets/werkzeug/blob/347fdbb055c86efe1fd49546bd524cde4b98c103/src/werkzeug/wsgi.py#L141

Library: Flask-SocketIO

General Information & Licensing

Code Repository	https://github.com/miguelgrinberg/Flask-SocketIO
License Type	Flask-SocketIO is distributed under the MIT license.
License Description	• A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code. • Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so.
License Restrictions	• In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.
Who worked with this?	• Shawn, Andy, Ryan, Kevin

Use as many of the sections below as needed, or create more, to explain every function, method, class, or object type you used from this library/framework.

on(self, message, namespace=None):

Purpose

- The *on* function is used when an event occurs. It is basically an event handler that fires whenever the event happens. The event could be a predefined event like connect, disconnect, etc. or it could be a user-defined event like in our project, voting.
- We used the *on* function throughout the project to handle events like connection, voting, and sending messages. Because the *on* function can be used as an event handler a vote button or a message button can fire the *on* function depending on the argument passed in.
- One place in our project we use the *on* function from the Flask-SocketIO library is when the “vote” button is pressed on the home page. We used the event “vote” to call the function. *@socketio.on('send_msg')*
    - https://github.com/shawnz99/CSE312-GroupProject/blob/main/app/app.py#L172

Magic

- The *on* function is defined in the Flask-SocketIO library that is used as an event handler. This function can be used for the detection of a new TCP socket connection to the TCP server. After the TCP socket server is started this function is used in our project to update the current users of the website, since every time a user connects to the server the “loggedIn” flag needs to be updated.
- Another use of the *on* function is handling user-defined events. One event we created was a vote. Since every message can be upvoted, we needed to create an event that fired whenever the upvote HTML button is clicked by a client.
- The *on* function is a decorator of the SocketIO event handler. The *on* function calls the *_handle_event* function which is also defined in the Flask-SocketIO library in the __init__.py file. The *_handle_event* function uses the arguments passed in to return an object called ret instantiated by the *handler* function. The *handler* function takes the message passed into the *_handle_event* function. In our case, if the message is a user-defined message it passes that is the argument ex.) “vote”. If this object is returned the *on* function will execute.

- The wrapper/decorative function is defined in the __init__.py file:
    - https://github.com/miguelgrinberg/Flask-SocketIO/blob/main/src/flask_socketio/__init__.py#L258
- The *on* function calls the *_handle_event* function which is located in the same file.
    - https://github.com/miguelgrinberg/Flask-SocketIO/blob/main/src/flask_socketio/__init__.py#L734
- The *_request_ctx_stack.top.session* object used to create session_obj in the *_handle_event* function is found in the Flask library’s global.py file:
    - https://github.com/pallets/flask/blob/main/src/flask/globals.py#L52
- After the globals.py file includes uses the LocalStack() in the Werkzeug library:
    - https://github.com/pallets/flask/blob/main/src/flask/globals.py#L52
- Within the local.py file the *top* function is defined:
    - https://github.com/pallets/werkzeug/blob/main/src/werkzeug/local.py#L142

emit(self, event, *args, **kwargs):

Purpose

- The *emit* function is used to send out an event to one or more connected clients that are connected to the server. When called a user-defined event will be emitted that event handlers can listen to and handle the event.
- In our project, we use the *emit* function to send out or emit a “vote_update” event which fires whenever a clients’ user clicks on the upvote button of a message, this allows for the “vote” event handler to execute.
- This use of *emit* is found on lines 207-210 in our app.py file:
    - https://github.com/shawnz99/CSE312-GroupProject/blob/main/app/app.py#L207

Magic ★★ ° ˖ ° ☽ ° ⁀ ° ★ ≡ ✧ ⌯

- The *emit* function takes an event and a JSON string as arguments. For our project, we used “vote_update” as the event and a JSON object with “votes” and “div_id” elements. “Votes” to hold the messages votes, and “div_id” to hold the unique message ID. We call the *emit* function within our “vote” event handler. When the client votes on a message “Votes” is incremented by 1. And the *emit* function is called with this update “Votes” value.
- *emit* is declared in many different places in the Flask-SocketIO library. The definition that sends the server event out to clients is found in the test_client.py file. The definition found in app.py adds the necessary arguments and calls the *emit* function from test_client.py
- To emit the event to the server the *emit* function uses the *self.socketio.server._handle_eio_message* function to send out the event to the server and the clients connected to the TCP server. This function takes an “eio_sid” as an argument and that is set using the UUID library.
- The *emit* function is defined in the app.py file of Flask-SocketIO:
    - https://github.com/miguelgrinberg/Flask-SocketIO/blob/a10ea5cf65007061d7b3fd87b530c382007adebb/src/flask_socketio/__init__.py#L401
- The *emit* function that is used to actually send the event out to all of the clients or sockets connected to the TCP server is found in the test_client.py file:
    - https://github.com/miguelgrinberg/Flask-SocketIO/blob/a10ea5cf65007061d7b3fd87b530c382007adebb/src/flask_socketio/test_client.py#L137
- The UUID is set in the test_client.py file at the following link:
    - https://github.com/miguelgrinberg/Flask-SocketIO/blob/a10ea5cf65007061d7b3fd87b530c382007adebb/src/flask_socketio/test_client.py#L64
- The *uuid4* function is defined in the UUID library:
    - https://github.com/uuidjs/uuid/blob/main/src/v4.js