gmCSE312 Group project Library report

Shijie Liu, Roshan Pradeep, Ryan Older, Eliza Koster

[Flask Library]

General Information & Licensing

Code Repository	https://github.com/pallets/flask
License Type	BSD-3-Clause Source License
License Description	● The BSD-3-Clause license is very similar to the MIT license and you can: ○ Use the code commercially ○ Modify the code ○ Distribute reworked versions or copies of the code ○ Are allowed to place a warranty on the licensed software
License Restrictions	● 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.
Who worked with this?	Armin Ronacher

[Flask app.py (Flask class)]

Purpose

What it does for us:

- The bulk of our web application is based on the help of Flask and especially the app.py file. This is where most of Flask's functionality comes from and we use this file for a multitude of things. We handle HTTP requests/response parsing and generation here notably.

Specifically:

- In our main.py and auth.py files we utilize many methods from the Flask class in the app.py file from the Flask library. Notable examples include the utilization of route(), redirect(), template configuration methods, URL rules, and jinja related methods. The Flask class also does a majority of what we did with the route() method documented below where we set up specific handlers for certain GET and POST requests that the user may send to our server.

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

  Usually, we create a flask instance in our main module and the object acts as the central object. We pass the name of the module or package of the application. This name will be used to resolve resources from inside the package or the folder the module is contained in.

  By establishing the main module of the application, this allows us to access the core purpose of the Flask framework and that is to access as the central object to access most of the methods that we need to create our web application.

- flask/src/flask/app.py, starting at lines 98 to line 2091

  After initializing the app by doing “app = Flask(__name__),” the Flask class starts initializing itself with several different classes from the Flask library to set up the app. Notably it access the following classes:

  Class: flask.Request : flask/src/flask/wrappers.py line 16

  Class :flask.Response: flask/src/flask/wrappers.py line 134

  And Class: werkzeug.routing.Rule: werkzeug/src/werkzeug/routing.py from

  https://github.com/pallets/werkzeug/blob/main/src/werkzeug/routing.py

  When initializing the Flask class for our app it also establishes a TCP connection for us to use in our web application and also accesses the **Ctx.py** file for the requests and handling of the connecting tcp connections.

  Inside the Flask class notable features we can access include:

  register_blueprint() method which helps us initialize blueprints for our web app

  add_url_rule() method which allows us to establish routing rules for our web app (explained in the Flask.route() method documentation below)

  A variety of template methods including **template_filter(), add_template_filter(),template_test(), and template_global()**

  Several http exception methods to handle HTTP exceptions such as **handle_http_exception(),trap_http_exception(), handle_user_exception(), and handle_exeption().**

And notable response/request methods that handle HTTP requests and responses such as **make_response() and process_response().** Whenever Flask handles any http requests and responses they usually go through these methods which stems from the Response and Request class in flask. Essentially this handles all the GET and POST requests/responses for the web app.

Make_Response:
It is used to add additional http headers to a response within a view's code.This function can be called instead of using a return and you will get a response object which you can use to attach headers.
This function accepts the same arguments you can return from a view function. Another use case for this function is to force the return value of a view function into a response with a 404 Error Code.

Process_Response:
Utilizes Ctx.py in the flask library to process specified responses. (documentation for Ctx.py below)

Flask:
The flask object implements a WSGI application and acts as a central object. The class starts at 97. It immediately creates a request class, a response class, and a jinja environment (described in templating). It has a default config which sets basic headers that are needed. This is located at lines 317-349. These headers include session_cookie_httponly, and other important headers. This is an immutable dict. Next it creates a url rule class, a url map class, a test client class, and a test_cli_runner_class, and a session_interface. The session interface handles the cookies, although it seems we did not end up utilizing this. The init function takes the name, the static url path, folder, host, template folder, instance path, and templates. It checks if the instance path is none, and if it is none, then it automatically finds one. Next it maps urls using url_map_class, described below. First it checks if it has a request, and then doesn't take any more requests once it has one, sort of like a semaphore. I've described a lot of this below, as most of it happens due to an imported library werkzeug, by the same developer.

Def Run is the most important file, it handles the connection of the application on a local deployment server. Here, it takes the hostname to listen on, a parem port, a parem debug, a parem options. ConfigAttribute handles this. Here, it creates the server name to config it. It checks if it is a host, and if it is not, then it sets it to host = 127.0.0.1. It creates a port of 5000 if no port is specified. It sets a default port to threaded.

TCP CONNECTION

I will describe the run() function. The run function takes the host name to listen on, which defaults to 127.0.0.1. It also has a port parameter, which is the port defined in the server name and defaults to 5000. It also takes load_dotenv, which loads the nearest file environment and uses that to set it's environment. This function occurs at line 804. Here, given the parameters, it configures the server name, port, and host. It also sets it's options to default threading. Finally, it calls run_simple from werkzeug. This function occurs on line 788 in it's serving directory. It takes parameters such as the request handler, static files, and other important information, when Flask calls it, it imports all of the options for the flask server object, it's port, itself, and it's host as a string. Here, it has a definition for log_startup, which handles the display message that we see when we run the server. It just turns the hostname, port, and display hostname, and configures that into the accessible url that it is running on. This happns on lines 982-920 in the github serving.py file. It also has an inner function, which is where the actual server happens. It calls make_server with all of the parameters such as hostname, port, threaded, processes, that are needed to start a tcp server. The make server creates a new server instance that is either threaded, or forks to a new request. This just returns the wsgi server. Finally, it serves forever at line 940, and the tcp connection has been made.

Here, I will only reference the flask class in app.py with respect to configuring the server and handling tcp responses and requests. Other information regarding the server handling is in jinja documentation.

I will describe the run() function. The run function takes the host name to listen on, which defaults to 127.0.0.1. It also has a port parameter, which is the port defined in the server name and defaults to 5000. It also takes load_dotenv, which loads the nearest file environment and uses that to set it's environment. This function occurs at line 804. Here, given the parameters, it configures the server name, port, and host. It also sets it's options to default threading. Finally, it calls run_simple from werkzeug. This function occurs on line 788 in it's serving directory. It takes parameters such as the request handler, static files, and other important information, when Flask calls it, it imports all of the options for the flask server object, it's port, itself, and it's host as a string. Here, it has a definition for log_startup, which handles the display message that we see when we run the server. It just turns the hostname, port, and display hostname, and configures that into the accessible url that it is running on. This happns on lines 982-920 in the github serving.py file. It also has an inner function, which is where the actual server happens. It calls make_server with all of the parameters such as hostname, port, threaded, processes, that are needed to start a tcp server. The make server creates a new server instance that is either threaded, or forks to a new request. This just returns the wsgi server. Finally, it serves forever at line 940, and the tcp connection has been made.

The function make_shell_context, which happs at line 759. This function returns the shell context processors, and makes it possible to call flask run in the terminal. The function takes the app, as well as it's globals, and then for all of the processors in it's context processors, it adds them to the rv dictionary, then returns them.

Parsing headers:
Flask uses werzeugo, so I'll describe that first.

wsgi.py : https://github.com/pallets/werkzeug/blob/main/src/werkzeug/wsgi.py

This is where the actual parsing of headers happens.

Buffering:

Class: Filewrapper: 492

Handles the reading of files. It defines some methods called in range wrapper and _. It's init includes a buffer size and a file. It has a close function which closes the buffer size, a definition called seekable, a function seek, a function tell, a function iter, and a function next. The seek just finds an attribute, seekable returns if the attribute exists, the tell returns the attribute, the next continues on.

Range_Wrapper: This class converts an iterable object into an iterable. (line 544). This will allow for buffering. It initializes as itself, an iteriable, a start byte, and a range.

It's init class checks if it has a seekable, and it also sends end_reached false. It states that nothing has been read. The next chunk just calls next and increases the read_length. If it fails then it has reached the end of the file. The first iteration sets chunk to none. It checks if it is seekable. If it isn't, then it reads it while the read length is less than it's start byte. Its the next chunk, verifies that it isn't none, then pulls the readlength from it. Then it returns the chunk and the contextual readlength, which is just the start length. This happens at line 593. The next function checks if read length is 0, and if it is, then sets itself to first iteration, and if it isn't, then it sets itself to next chunk. It checks if self.end_byte is not none and if the readlength is larger than the end byte. If that's the case, it's reached the end of the file, and stops. This happens at line 607.

MakeChunkIter on line 633 takes a stream, a limit, and a buffer size. It just cycles through the stream, and reads the file.

Make_line_iter goes through the input stream line by line.This happens at line 698. It just cycles through splitting by new lines, and then creates a list of the new lines. It does this until the buffer has ended. It also considers the edge case of ending at \r, and \n. It just checks the next character and compares it with the current one, and then returns a previous value if it needed to do that.

Werkzeug routing: https://github.com/pallets/werkzeug/blob/main/src/werkzeug/routing.py

Werzeugo Rule(RuleFactory) Line 557:

A rule is one URL pattern. It takes a string, which is just a URL path, an endpoint which just states where the rule ends, defaults, which creates default options for the same url, for instance a redirect default might be passed through inorder to also create a rule for redirect to the same url when creating a rule for get. The methods function is a sequence of HTTP methods that this rule applies to. If it isn't specified, then it takes all of the rules. It also has a websocket function that just matches for websockets by editing the ws:// or wss://.

The first thing that the init does is check if the methods exists. If it doesn't exist or is none, then it returns an error. If ti does exist, then it converts all of the method strings to upper. Now, it checks if there are certain errors or illegal rules that are being attempted to be created, for instance, it checks if there is some rule that is being created that doesn't work with websockets. It also checks if the default method has been added or not, and if it hasn't, it adds it. Now, it initializes it's methods, endpoints, and redirects to the defined variables above. The init function ends at line 771.

As described above, the headers are defaulted and listed. The function add_url_rule takes a rule, an endpoint, a vier function. It defaults to the GET method. It is located at line 1037. Here, it tries to find the method using the getattr function, and checks if the method exists. If it

doesn't, it returns a method saying that the allowed methods must be a list of strings. If it does exist, then it creates a set of methods named methods for all of the imputed methods. Next, it makes a list called required methods (of required methods) which takes the attributes for all of the methods, puts them them into a set, and saves them. Next it creates the actual rule by calling url_rule_class. This was instantiated at line 355, and it defaults to werzeug.routing.rule.

Converts to help with the compilation of a rule: unicodeconverter just uses regex to validate a a string with only one path segment, it creates regex for itself inorder to return the string with the length of it's string.

The number converter takes a number, and then has important functions for it. To_python returns the number, to_url returns the number as a string.

Map Class:

The mapp class stores all of the URL rules and some configuration params. It takes all of the url rules, a default subdomain, a charset, if slashes matter, merging slashes, if there should be redirect defaults, converters (2 of which are described above), sort parameters to determine the sorting of url paramateers, and sort key for url_encode. It first creates a dictionary for all of the converters. It's init function then also updates it's converters dictionary if there are more, and it also adds rules to the existing rule dictionary if there is any.

Bind_to_environ. This function exists in the map class at line 1611. It is used to take information from the wsgi environment. This allows it to read important headers such as HTTP_CONNECTIOn, and HTTP_UPGRADE to tell if it's a websocket. Next, it defines wsgi_string which just turns decodes the wsgi into a string. It encodes it to latin then decodes it into utf-8. This is located in internal.py. Internal.py also has an easteregg at line 563 that I don't know what it is. Eh. Next, it checks if the servername is none, and if it is then it turns it to the default server name, and if it isn't then it turns it to lowercase. We are selfhosting, so some of this stuff isn't relevant to us. Finally, it returns a the string of the scriptname, pathinfo, and querystring, and then binds it. Bind happens in line 1545. It returns a class map_adapter. It was described earlier.

MapAdapterClass:

This does the dispatching process. It looks up a view function, calls it, then returns a response object or WSGI application. It takes the viewfunction, the path information, the method that is being called and a boolean if we want to catch exceptions, which by default is set to false. I haven't talked about http exception catching because we didn't use it.

The match function mathes the method to the current path. It updates the map, then gets the path information and query argos and turns them to strings. It sets websocket to false if it wasn't past through. Now it just creates the path. Now it goes through all of the rules, then attempts to match the path to the method. If it doesn't exist, it will create a redirect function. It continues on if there is no more matching. Next it checks if the redirect_to exists, and if it does, it creates the rules based on redirect_to_exist. Line 1843 is where it starts.

Next it has a build function, which instead of matching urls, it builds the function. So earlier on it was parsing requests urls, and now it is building them.

https://github.com/pallets/werkzeug/blob/347fdbb055c86efe1fd49546bd524cde4b98c103/src/werkzeug/_internal.py#L149

I will start with the function make_shell_context, which happs at line 759. This function returns the shell context processors, and makes it possible to call flask run in the terminal. The function takes the app, as well as it's globals, and then for all of the processors in it's context processors, it adds them to the rv dictionary, then returns them.

Parsing headers in Flask:

https://github.com/pallets/werkzeug/blob/main/src/werkzeug/serving.py

As described above, the headers are defaulted and listed. The function add_url_rule takes a rule, an endpoint, a vier function. It defaults to the GET method. It is located at line 1037. Here, it tries to find the method using the getattr function, and checks if the method exists. If it doesn't, it returns a method saying that the allowed methods must be a list of strings. If it does exist, then it creates a set of methods named methods for all of the imputed methods. Next, it makes a list called required methods (of required methods) which takes the attributes for all of the methods, puts them them into a set, and saves them. Next it creates the actual rule by calling url_rule_class. This was instantiated at line 355, and it defaults to werzeug.routing.rule which is described above. SO essentially, as described, it handles the creating of the context for the responses, but depends on werzeug for the parsing of requests and mapping of the rules.

Ctx.py

Some of the handling of requests and handling happens in ctx.py and app.py. Ctx.py partly handles the request_ctx_stack and the app_ctx_stack.

The class AppCtxGlobals (24) is used as a namespace for storing data during an application context. It has the functions get, setdefault, contains, an iterator, and repr. Get takes an attribute by name, pop getes and removes an attribute by name and returns default if a value is not present. Setdefault gets the value of an attribute if it's present, otherwise returns a default value. Contains just takes a string (attribute) and returns it if it exists.

The class AppContext takes the application context and binds an application object implicity to the current thread or greenlet. A greenlet allows for synchronous programming.

Init takes the flask app, and then sets a url_adaptor by creating a url adopter by calling app.create_url_adapter, explained in the app section. It also sets g (globals) to app_ctx_globals_class() explained in the app.py file. App contexts can be pushed multiple times, but they need to be tracked. Refcnt is initialized as 0. (228-235)

It then contains a push function which binds the app context onto the app_ctx_stack, then signals that it's been pushed to itself, and also increases it's refcount, because an appcontext has been pushed. (237-241)

It also contains a pop function which subtracts 1 from the refcount and returns an error if the refcount is below or = to 0. Then it pops it and returns rv. (243-253)

It then has an enter function which returns appcontext, and it just returns itself.

It also has an exit function which pops an exception value from itself.

The requestContext contains relevant request information. It is created at the start of the request and pushed to the request_ctx_stack, then removed at the end of it. It also creates a URL adaptor and request object for the WSGI (web server gateway interface) environmnet that is provided. It will tear down a request context when it is popped, and also automatically pop it at the end of the request.

The init function takes itself, an app of type Flask object, an dictionary environ, a request and a session, and then it returns none.

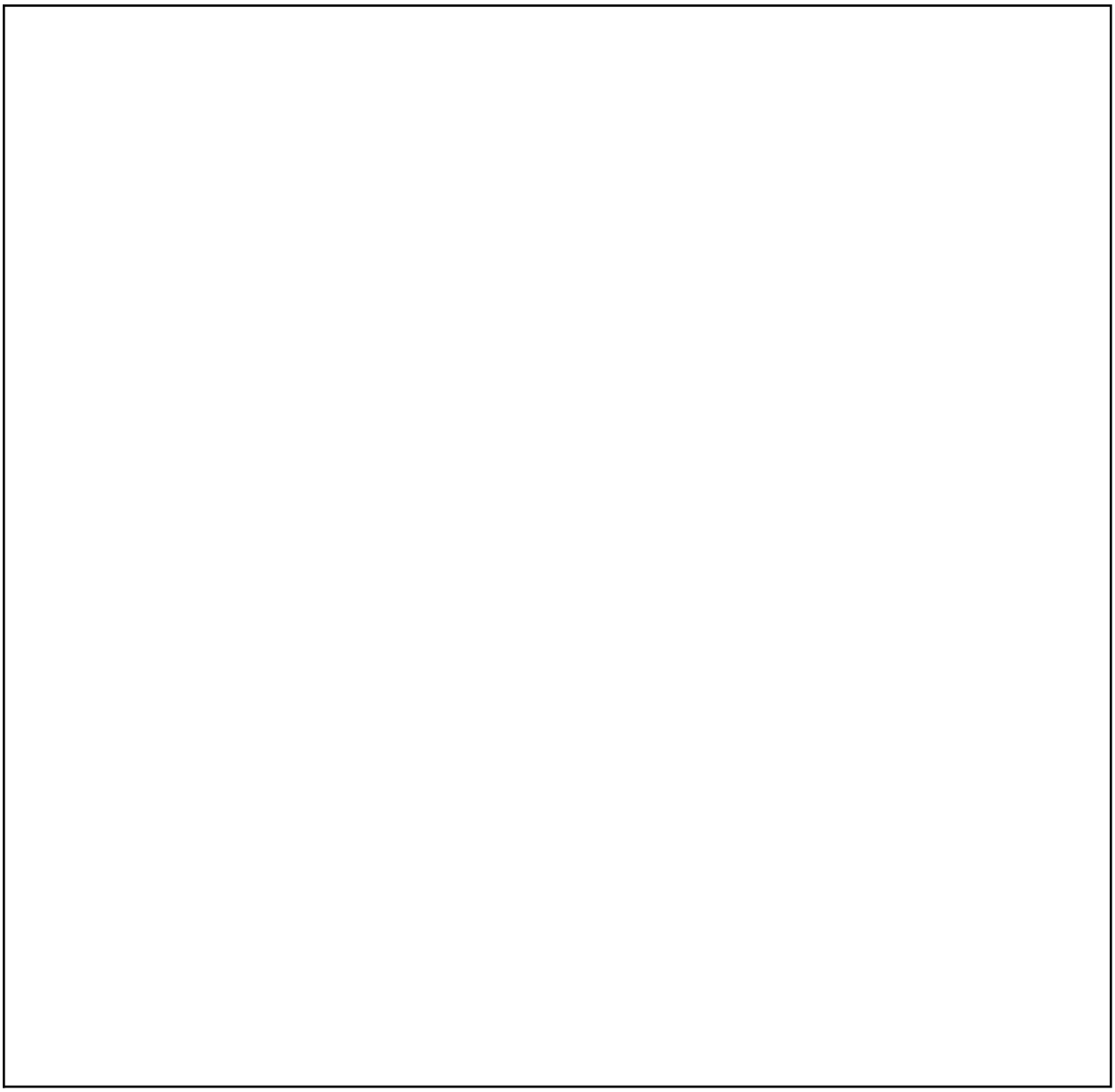
It checks if the request is none, if it is none, it will request a new class using app.request_class, explained in the app.py file.

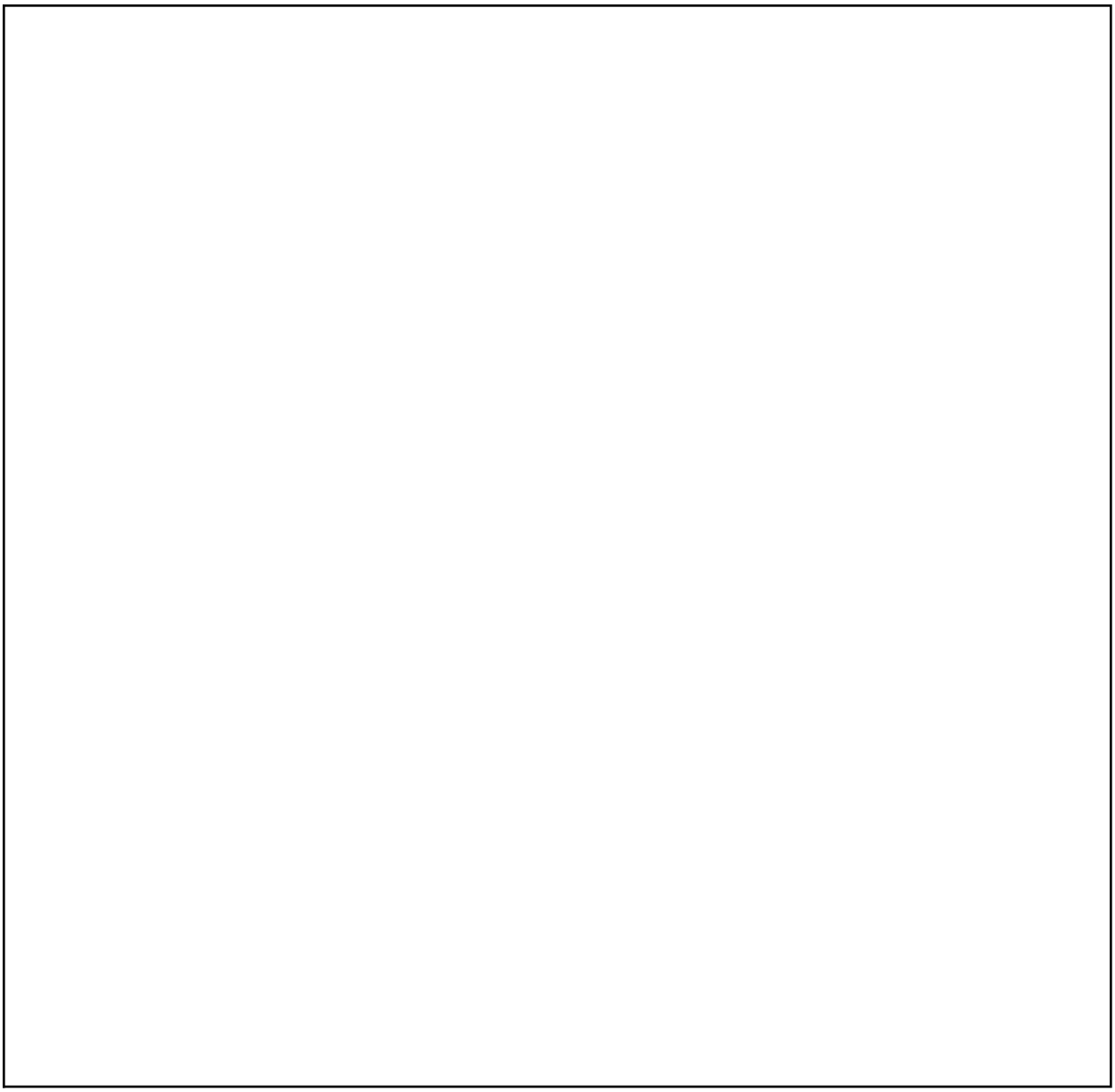
It additionally creates a url adapter for the request. The url adaptor is done in app.py

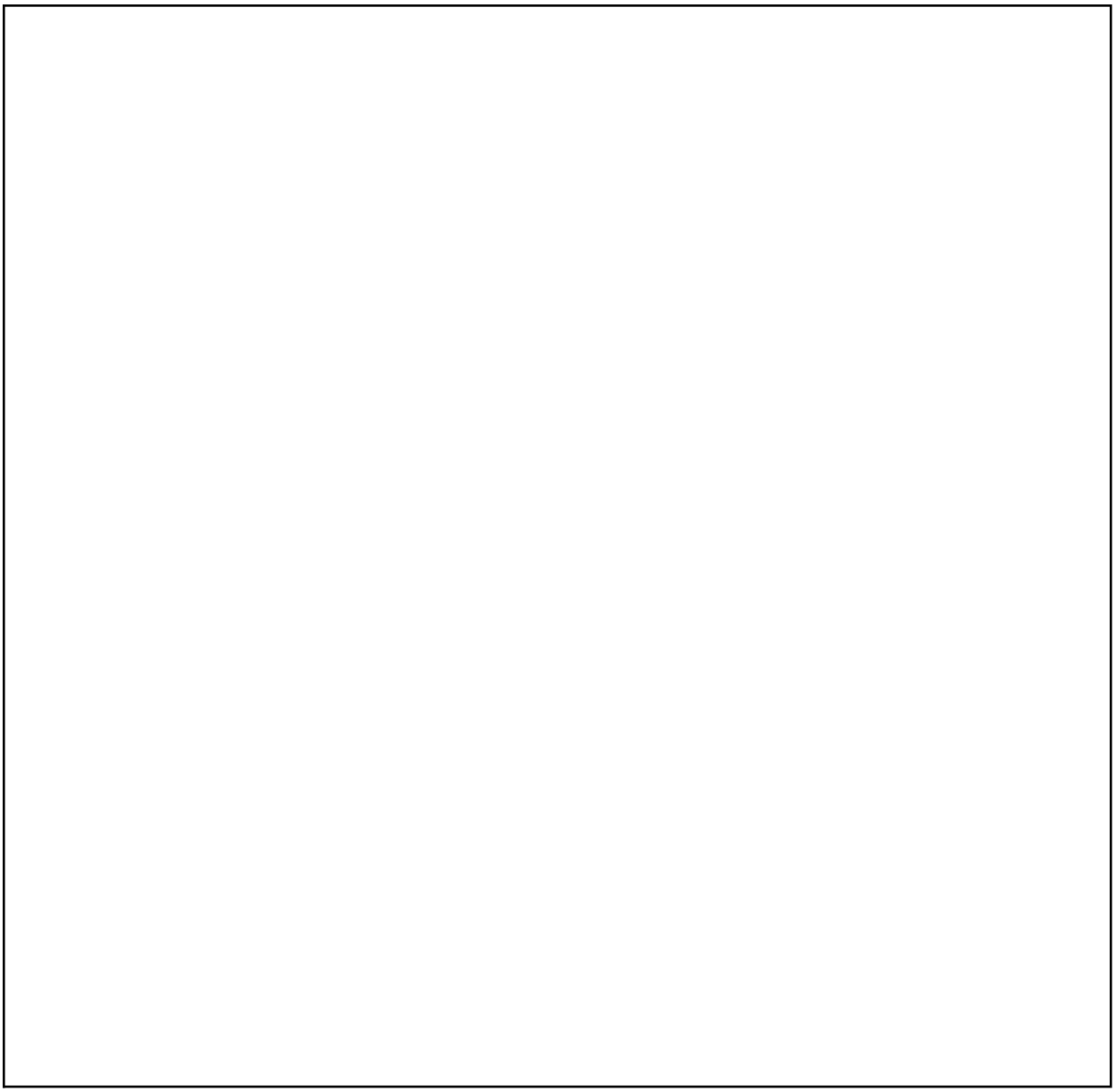
The push function binds the request to the current context, and is located at lines 390 -430. The push function first checks if there is an app context, and then if there isn't, it just appends nothing. If there is app context, it pushes the app context and pushes the request with the context to the stack. Next, it opens the session at the moment that the request context is available. It checks if the session exists, and if it doesn't, then it opens a new session with it's app and request. If it doesn't exist, then it just makes a null session. Finally, it matches the request url after loading the session, so that it's available to all url converters.

The pop function which takes an exception just pops the request context and unbinds it, triggering the execution of functions registered by teardown request.

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







[Flask Blueprint Class]

Purpose

What it does for us:

- Blueprints allow us to factor our application into sets of these blueprints. They allow us to record operations to execute when used in our web app. It is very similar to a flask application object.

Specifically::

-

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

- The blueprint class acts exactly the same as what you think a blueprint would do. It is the design plan for the code we would like to use and run in the flask application that contains a collection of routes and other flask code that can be added to the application. It acts as

[Flask @.route()]

Purpose

What it does for us:

- The route() function in flask allows us to set a url rule that allows us to call certain functions when the user visits the route that was specified in that route.
  (ie: @app.route(“/home”))

Specifically:

- This function is specifically used in our code in our server files (auth.py and main.py) We created multiple url rules using route() to have the server handle stuff when the user tries to sign up, login, and access the home page.
  (ie: main.py has a route(“/home”) rule that handles what the user will see and what their information will change into when they visit the home page.)

Magic ★★ ° ˙ ° ☽ ° ⌒ ° ★ ★

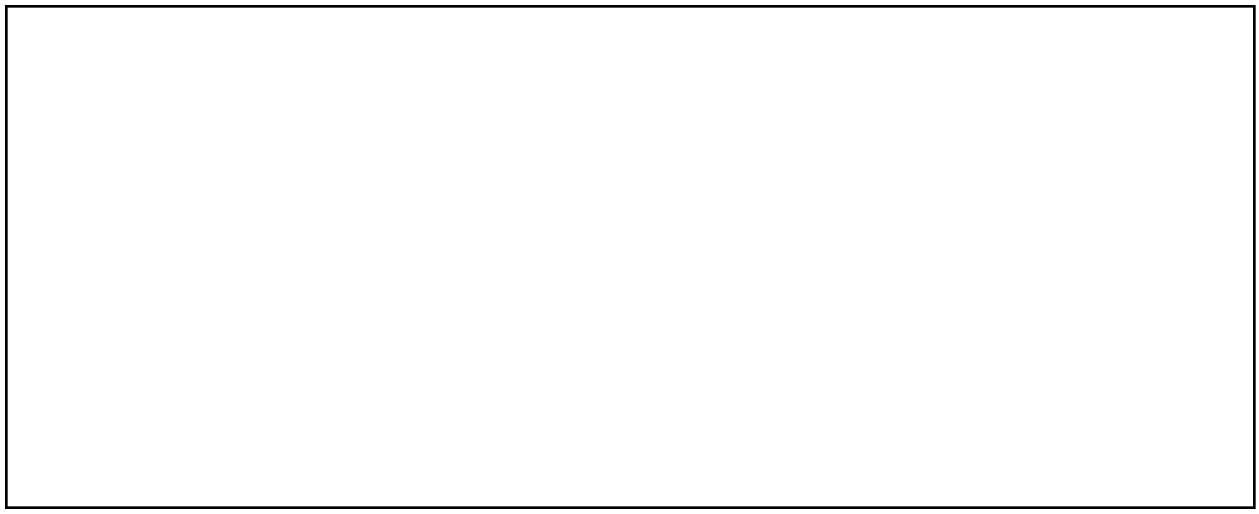
Dispel the magic of this technology. Replace this text with some that answers the following questions for the above tech:

- How does this technology do what it does for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

  -The user will have the ability to visit specific routes on our web application. We use route() to specify what our server and web application will do once the user visits a specific route that matches the route() that has that specific route as its parameter (@app.route("/home")). When declaring this route function, it calls the add_url_rule() function inside the flask library which handles most of the url rule adding stuff. The add_url_rule() function registers a rule for routing incoming requests and building URLs that the user requests. Along with helpful parameters to allow us to specify what to do if the request was a "POST" or a "GET," route() essentially utilizes add_url_rule to allow us to dictate what our web application does when the user accesses a specific route.

- flask/src/flask/app.py, starting at lines 1037 to line 1093
  https://github.com/pallets/flask/blob/main/src/flask/app.py

  When calling route(), it redirects to add_url_rule() where it will then check for the given parameters. First is the rule which is the specified route in which the rule is assigned to, the endpoint name to associate with the rule and view function. (Used when routing and building URLs.), the view function to associate with the endpoint name, and any extra options passed to the Rule object that the function references. This Rule stems from the werkzeug class and it essentially represents one URL pattern. There are some options for Rule that change the way it behaves and are passed to the Rule constructor. With the rule setup, we can properly use the route() function as a way to set rules whenever a user accesses a specific route and allows us to have the server do its magic when the specific route is accessed. The routing and handling of HTTP requests in the actual route() function is handled in the ctx.py and the rest of the app.py files in the flask library.



[Flask ctx.py]

Purpose

Replace this text with some that answers the following questions for the above tech:

- What does this tech do for you in your project?
- Where specifically is this tech used in your project? Give us some details like file location and line number, if applicable. If too cumbersome, a general description of where it's used for a given purpose is fine as well.

What it does for us:

- Requests are used to access incoming request data. Flask parses incoming request data for you and gives you access to it through global object request. Internally, flask also makes sure that you always get the correct data.

Specifically:

- In auth.py line 26, here request is used to access its method, i.e, the method the request was made with which in this case is 'POST'
- In auth.py lines 27 and 28, request.form['username'] and request.form['password'] are used to extract multipart form values from templates to access input names of username and password respectively.
- In main.py line 34, request.files['upload'] is used in this case to extract the file as a FileStorage object from a multipart form user input of type file.

The handling of requests and handling of connecting to the tcp server happens in ctx.py and app.py. Ctx.py handles the request_ctx_stack and the app_ctx_stack.

The class AppCtxGlobals (24) is used as a namespace for storing data during an application context. It has the functions get, setdefault, contains, an iterator, and repr. Get takes an attribute by name, pop getes and removes an attribute by name and returns default if a value is not present. Setdefault gets the value of an attribute if it's present, otherwise returns a default value. Contains just takes a string (attribute) and returns it if it exists.

The class AppContext takes the application context and binds an application object implicity to the current thread or greenlet. A greenlet allows for synchronous programming.

Init takes the flask app, and then sets a url_adaptor by creating a url adopter by calling app.create_url_adapter, explained in the app section. It also sets g (globals) to app_ctx_globals_class() explained in the app.py file. App contexts can be pushed multiple times, but they need to be tracked. Refcnt is initialized as 0. (228-235)

It then contains a push function which binds the app context onto the app_ctx_stack, then signals that it's been pushed to itself, and also increases it's refcount, because an appcontext has been pushed. (237-241)

It also contains a pop function which subracts 1 from the refcount and returns an error if the refcount is below or = to 0. Then it pops it and returns rv. (243-253)

It then has an enter function which returns appcontext, and it just returns itself.

It also has an exit function which pops an exception value from itself.

The requestContext contains relevant request information. It is created at the start of the request and pushed to the request_ctx_stack, then removed at the end of it. It also creates a URL adaptor and request object for the WSGI (web server gateway interface) environmnet that is provided. It will tear down a request context when it is popped, and also automatically pop it at the end of the request.

The init function takes itself, an app of type Flask object, an dictionary environ, a request and a session, and then it returns none.

It checks if the request is none, if it is none, it will request a new class using app.request_class, explained in the app.py file.

It additionally creates a url adapter for the request. The url adaptor is done in app.py

The push function binds the request to the current context, and is located at lines 390 -430. The push function first checks if there is an app context, and then if there isn't, it just appends nothing. If there is app context, it pushes the app context and pushes the request with the context it to the stack. Next, it opens the session at the moment that the request context is available. It checks if the session exists, and if it doesn't, then it opens a new session with it's app and request. If it doesn't exist, then it just makes a null session. Finally, it matches the request url after loading the session, so that it's available to all url converters.

The pop function which takes an exception just pops the request context and unbinds it, triggering the execution of functions registered by teardown request.

Magic

Dispel the magic of this technology. Replace this text with some that answers the following questions for the above tech:

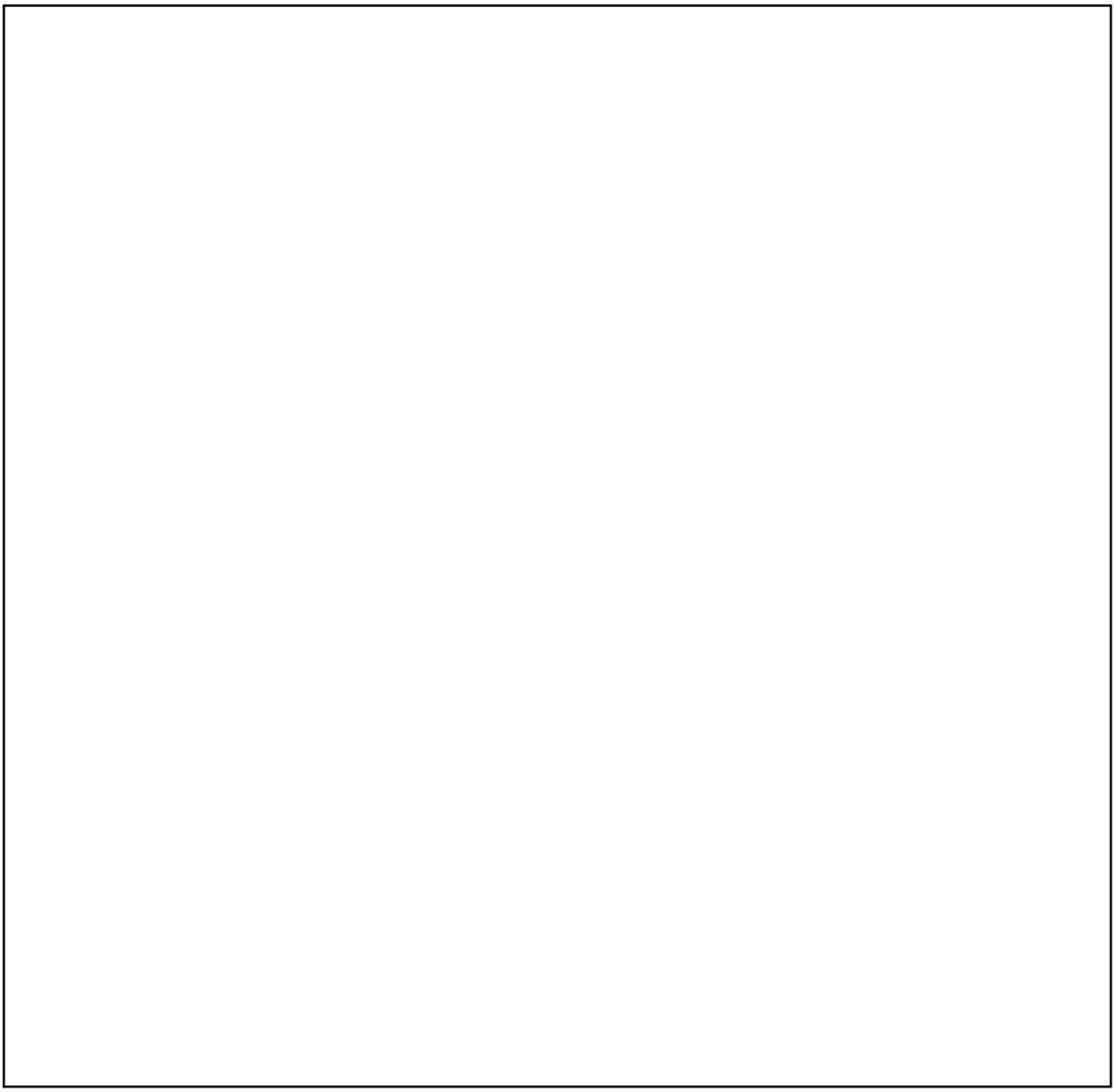
- How does this technology do what it does for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

  Usually, we create a flask instance in our main module and the object acts as the central object. We pass the name of the module or package of the application. This name will be used to resolve resources from inside the package or the folder the module is contained in.

  The first parameter is used to give Flask an idea about what belongs in our application. This name is used to find resources on the file system and can be used by extensions to improve debugging information and more.

  This creates and configures the app at the name provided in the first parameter.

- flask/src/flask/app.py, starting at lines 98 to line 2091



[Flask render_template]

Purpose

Replace this text with some that answers the following questions for the above tech:

- What does this tech do for you in your project?
- Where specifically is this tech used in your project? Give us some details like file location and line number, if applicable. If too cumbersome, a general description of where it's used for a given purpose is fine as well.

What it does for us:

- This will allow us to manage templates so that it is easier to produce dynamic pages for our users. The template for Flask consists of variables and expressions that will get replaced during the rendering of the template. It also consists of tags that will handle the logic behind the rendering of the template. This will also make it easier to produce pages that inherit from other pages so that the organization of our code will be improved and more maintainable.

Specifically:

- Templates will use handlebar syntax inspired by django and python. For most of our html pages, we will need to use templates in order to handle the rendering of unique pages for our clients. It will be used in all of our static html pages except for our login page, as after a user logs in, the pages will all be dynamic. For example, we may have a line in excersiseSchedule.html that displays the username, and in order to display the username, we might write <h1> {{username}} </h1> where username is the name we would like to access. In our news feed, we would need to use the inheritance functioning of templates, in-order to dynamically generate the posts that other users have made that may be of relevance to the logged-in user. We also use **flash** to help us with the template rendering that is under the Flask library.

Magic ★★ ° ˙ ˚ ☽ ˚ ⌒ ˚ ★ ≡ ★ ≡

More info

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

After the TCP socket is created in our server, we use Flask, as described in the above section, to parse GET requests. Mainly, when the request starts, a RequestContext object is created and pushed onto the request stack, which then checks to see if it it's respective AppContext is pushed onto the app context stack. If there isn't, it will push the respective AppContext onto the stack. After the request is generated and the response is sent, the request context is popped, and the functions associated with the request are executed, even if an exception was thrown during the handling of the response, and if the response was not adequate sent.

As a response to a request for a template file, we will use the Render_Template function, which takes a template file and it's context, in order to respond to the server with the correct html file.

We will break this section up into 2 parts. We will first describe the actual compilation that occurs in flask using Jinja for the render_template function, then we will describe how Jinja's render function works to render templates.

How the Render Templates works in Flask SRC:

Render Templates happens in the templating.py (1). Render_template takes the context, which is the variables that are available in the context of the template, and a template_name_or_list, which is the name of the template to be rendered or an iterable with template names, where only the first one will be loaded. Flask keeps a global localStack() which it uses to store all of the context's. It pops the top context off and then updates it using update_template_context(2), which is located in the app.py file in Flask, and just updates the context dictionary by adding all of the new keys and values from the context variable passed through it, and updating existing keys with new values if there are any. Finally, it calls the _render function, and returns rv, or the template to be rendered using Jinja with the render(context) method. It also passes into the render function the environment associated with the context using jinja_env, which is necessary for parsing the templates, as described in the jinja documentation below.

Note that Flask also adds additional information to the environment, so it creates a custom environment for Jinja, that considers flask-specific blueprint information. It also

creates a loader for Jinja called DispatchingJinjaLoader, which looks for templates in the application and all the templates in the blueprint folder in flask src. (47-48). Both of these are just customized Jinja environments and templates.

Render function in Jinja called by Flask:

The render function just renders the template by cycling through the file, and for everything from the file, it calls render_async on itself. Render_async occurs at lines 1291, and it creates a new context given it's arguments and dictionary. It calls root_render_func on the context, which is a namespace (1144), who's functionality is described in the loader section.

The render function happens in the environment src inside of github. The render function takes the context as a dictionary and returns a rendered template as a string. We will explain the important parts of the environment (3), lexer (4), and Loader (5) files, as well as additional files to explain how templating works in Jinja inorder to better explain the context behind the render function, which is called on an environment, which is created inside of flask, so that the render function will make more sense and the way that it's created will be more logical. Please note that if I do not go perfectly into detail regarding a function in one section, it's likely because I will go into detail into that function when discussing the file that it's located in. I will focus on creating connections / describe the callings in the file sections, and describe the respective information.

Lexer in Jinja:

The lexer init class takes the environment. It also stores some static regular expressions inorder to help with the creation of rules. The first thing that it does with the rules is compile the rules from the environment into a list of rules, which is regular expressions that scan for tokens. Essentially, it takes the environment tokens that have been stored, such as Token_Variable_Begin and stores it's respective length, then token itself, and the escaped tokens, and then stores it into an array. Next, it escapes using regular expressions, inorder to generate a string. The tokenize function generates a tokenstream. lexer (604-613). This first calls the wrap function on the tokeniter. The tokeniter returns all of the text as tokens in a generator, which is located in compiler.py (114-117), which is a code generator class from environment, which just takes an environment, name after joined path, filename, no stream during init, and a default defer_init = false, and comes with multiple helpful functions to generate code. It defaults to the first token as pos = 0, lineno = 1, stack = ["root"] in the lexer file lines 688 - 691. It then goes through all of the rules, and for each rule, goes through all of the tokens, and parses the rule (such as braces and parenthesis) if the stack is balanced. Then if they have extracted a block or variable, it creates a group, which accounts for blocks/strings. It enumerators through those tokens and matches the first named token to the group. It checks if the the group is an operator, then appends it to the balancing stack if it is not a },), or]. (earlier on, as was explained, the syntax that the user chooses compiles automatically to },),]) (792-803 in the tokenizer file). If it is a },),], it checks if the corresponding

{,[,(is on the stack, and if it is not, it returns an error that the token is unexpected, and if it is not, then it pops it off the stack. If the token is not in the data, it yields the linenumber, tokens, and data, and then increases the linno by the size of the data. Next, it fetches the position and a new group, inorder to check for an internal parsing error that could result in an infinite loop. It checks if it will, and if it does, it returns a runtime error saying that no group was matched.

The important thing to know about the lexer file (lexer) is that it has the tokenizer functions described above, as well as regex rules for the tokens, that are called in the template class inorder to help with the parsing of the template in the template_render function. The most important function is the tokeniter function which is called in the tokenize function, which returns a tokenstream, and is used to actually parse through the template file during the rendering of the template. The tokens are also wrapped so that the value is parsed as tokens that Jinja recognized, regardless of the tokens that are described in the environment. Flask does use the default characters described in environment.py, however, although template still converts the characters to one character type when creating tokens to different jinja tokens.

Environments in Jinja:

The init function for an environment in jinja creates base options for the environment, such as the block_start_string, which is “{%” . (6) The user has the ability to edit or customize any of these variables in the init function, although Flask just builds upon this with additional blueprinting information, rather than editing any of these values manually. The environment also has important methods that are called on the class Template that are used to actually render the files. For instance, lex(self, source, name, filename), returns an iterator that shows the a current token in the form (lineno, token_type,value). It uses tokeniter, from the lexer file, which is described below, to do this. It also has a _tokenize function which is needed inorder to preprocess and filter all of the extensions for their templating. This calls the pre_procoess function that preprocessess the source with all of the extensions (blueprints in the case of Flask) and then creates a stream from the tokenize method in lexer described below. It returns the lexer.Tokenstream. Compile functions are also included in this class, which takes the source code and the filename after it has been conjoined using join_path, and then returns a parsing tree of it in the form of either a string or a bytecode, depending on if raw is true or not.

The most important thing to note about environments in Jinja is that they store important variables that are required for the actual parsing of the template. These variables, such as the block_start_string, are just tokens such as ‘{‘ that can help determine the syntax for the parser. It also stores important information such as the loader, cache-size, and booleans like autoescape that the template class needs to determine how to parse a file. The parser takes the source code, and the context which includes these tokens that are used to determine the templating syntax, and parses it into a tree of tokens (characters), which the compiler uses to finally produce the compiled template. In addition to the environment, the lexer is used to tokenize the variables for the environment, and creates a tokenstream. Information relating to loaders and lexers are explained in more detail below.

The template class is a compiled template that can then be rendered given the context (a set of instructions + syntactical tokens) from the environment. The template object is immutable. All templates share the same environment. The template class starts at line 1118 in the src. When creating a new template, the template takes all of the tokens created in the environment, as well as any spontaneous environmental information, such as trimblocks and newline_sequences. Then it calls from_string from the template, which is in the environment class. It takes the self, the source, the globals, and an instance of the template class. It then calls make_globals which just makes a chainMap of all of the template globals which will only consider template-specific globals, so that changes won't effect global's changes to the environment in other templates. Then it makes cls the template class, then compiles the source as described in the compiler section, and , into the from_code definition at line 1199, which creates a template object. This from_code function creates a namespace with the environment, and the file name, then executes/ returns the code with respect to the name space. It then creates rv, which returns a template object, based on the given cls, environment, namespace, and globals. It creates a new template object and sets it's information based on the namespace dictionary, setting globals, name, filename, blocks, root into t. Then it stores t itself, and t's environment into the namespace. This occurs at line 1230.

Loader in Jinja:

Jinja loads templates using the packageLoader in Flask. Lines 234-392 in Loader. The package loader loads the templates from a python package. They take the name of the package that contains the template directory, the directory with the path that contains the templates, and an encoding of the file. In it's init function, it creates a package path using the package name and package path, and then it checks if the loader already exists before setting the template root and locating the pkgdirectory. If it doesn't already exist, it creates a list of strings called roots which has multiple elements for namespace packages, one element for one package, and none for only a single module file. If there is no template root, then it will raise a value error. It then creates the template root in it's init to the the list of strings. It also comes with the functions get_source, which is used in the environment file on multiple occasions, which just gets the template path, joins it, reads it, and then returns it if it is a file. If it isn't a file, it raises a templatenotfound error. Line 326-357.

An example of calling render templates is on line (96) of our auth.py code, and is written as render_template("Signup.html",xsrf=xsrfToken), where xsrf=xsrfToken is some of the context, and "Signup.html" is the template_name_or_list. We do this in multiple instances, in the same exact format described above.

Flash function in Flask:
The flash function in flask is used to flash a message to the next request from the client for the html template. It first gets the session and appends the flash category to flashes which is then sent to the server. When the html document loads get_flashed_messages pulls all flashed messages from the session and returns them when the template is rendered. They are then put onto the html document where the user can see them.

- https://tedboy.github.io/flask/_modules/flask/helpers.html#:~:text=def%20flash(message,message%2C%20category%3Dcategory)
- https://tedboy.github.io/flask/_modules/flask/helpers.html#:~:text=def%20get_flashed_messages(with_categories,category%2C%20message)%60%60%20instead.

1: https://github.com/pallets/flask/blob/main/src/flask/templating.py [133 - 151]
2: https://github.com/pallets/flask/blob/main/src/flask/app.py [731- 757]
3: https://github.com/pallets/jinja/blob/main/src/jinja2/environment.py [1257-1282]
0: https://flask.palletsprojects.com/en/2.0.x/reqcontext/
Lexer: https://github.com/pallets/jinja/blob/7d72eb7fefb7dce065193967f31f805180508448/src/jinja2/lexer.py#L325

Loader: https://github.com/pallets/jinja/blob/main/src/jinja2/loaders.py
Environment: https://github.com/pallets/jinja/blob/main/src/jinja2/environment.py

This creates and configures the app at the name provided in the first parameter.

1. src/flask/templating.py starting at lines 98 to line 2091
2. https://flask.palletsprojects.com/en/2.0.x/tutorial/templates/
3. https://github.com/pallets/jinja
4. https://github.com/pallets/jinja/blob/main/src/jinja2/compiler.py
5. https://github.com/pallets/flask/blob/9486b6cf57bd6a8a261f67091aca8ca78eeec1e3/s

rc/flask/templating.py#L124

*This section may grow beyond the page for many features.

[Flask-Login Library]

General Information & Licensing

Code Repository	https://github.com/maxcountryman/flask-login
License Type	MIT License
License Description	● The MIT License allows for the following: ○ Commercial Use, Modification, Distribution, Sublicensing, and Private use..
License Restrictions	● Author cannot be held Liable since the work is provided “as is” ● You must include the copyright notice in all copies or substantial uses of the work. ● You must include the license notice in all copies or substantial uses of the work.
Who worked with this?	Matthew Frazier, Alan Hamlett

[flask_login.login_user()]

Purpose

- This method allows our app to log a user into our server to access the features of our application.
- This method will be used when our server receives a log-in request from the login page requesting to be logged in.

What it does for us:

- It helps us correctly log-in a user of our app instead of us handling the entirety of the log-in system for our server/application.

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

  After a TCP socket is created the user would log into our application to access the features of our app. Our server will handle this log-in request by utilizing the flask_login.login_user() method logging in the user into our application. This login method checks for the information attached to the user object the login_user method takes as a parameter to decide if it should log in the user or not. This method also allows for specific durations of a logged in user until their remember cookie expires which is helpful for us. The method will return true if the user has successfully logged in and false otherwise.

- flask-login/flask_login/utils.py/ from lines 144 to 191
  - Utilizes the _update_request_context_with_user in flask-login/flask_login/login_manager.py in order to store the user that logged as ctx.user.
  -

[flask_login.login_required]

Purpose

Replace this text with some that answers the following questions for the above tech:

- What does this tech do for you in your project?
- Where specifically is this tech used in your project? Give us some details like file location and line number, if applicable. If too cumbersome, a general description of where it's used for a given purpose is fine as well.

What it does for us:

- This tech makes sure a user is logged in and authenticated before the user can view the page that is being requested.

Specifically:

- This tech is specifically used in our project to view pages that require logging in, for example, home page, direct messaging page, to log out of the account logged in.

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

  After creating the TCP Socket, when a user tries to access a page that requires them to be logged in, we use the login_required function whenever we wish a page to be accessed only after login/registration. If a user logs in successfully, they will be able to access all pages that require login_required. If we decorate a view like this, it will ensure that the current user is logged in and authenticated before calling the actual view.

  If the user is not logged in, it calls the LoginManager.unauthorized callback and doesn't display the page that the user is requesting.

- flask-login/flask-login/utils.py lines 233-278

[flask_login.logout_user()]

Purpose

- This method allows us to logout a user from our server.
- This method is used whenever a logged in user requests to logout from our website.

What it does for us:

- It helps us correctly log-out a user of our app instead of us handling the entirety of the log-out system for our server/application.

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

  Our server uses the logout_user function in flask_login to logout a user after they had requested it through our webpage. Logout_user first gets the user that is requesting to log out and removes them from the web socket session and also removes their remember me cookie.

- flask-login/flask_login/utils.py, from line 194 to line 220

[Flask-SocketIO]

General Information & Licensing

Code Repository	https://github.com/miguelgrinberg/Flask-SocketIO
License Type	MIT License
License Description	● The MIT License allows for the following: ○ Commercial Use, Modification, Distribution, Sublicensing, and Private use..
License Restrictions	● Author cannot be held Liable since the work is provided “as is” ● You must include the copyright notice in all copies or substantial uses of the work. ● You must include the license notice in all copies or substantial uses of the work.
Who worked with this?	Miguel Grinberg

[flask_socketio.SocketIO]

Purpose

What it does for us: - flask_socketio.SocketIO allows us to make a two way handshake with the client and be able to send messages to and from our server to the user in real time. - We are able to do this through sending events to and from our server with the client using socket.emit() - We use it in our web server to instantly change the user's username, send direct messages with other users, and post images to our site in realtime.

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

After a TCP socket is created and the user logs into our application and is authenticated, they are taken to our home page “index.html” and establish a web socket connection with our server through io.connect which connects the user to our socket.io server using its address and makes a new connection using the event handler socket.on(‘connect’, function(){});. The ‘connect’ event is defined in main.py with the @socketio.on(‘connection’) and when the function is run, the user gets put into their own room named after their username through the socketio function Join_room() and only allows either direct messages to that room using the to= parameter in emit or setting broadcast to true. With the connection established, we can further communicate with the client through event handlers and are able to allow direct messaging through them and communication with other clients.

- In main.py we have all our events that a user can request.
  - ‘Connection’ (lines 27-35)
    - Establishes a connection with the client on page load and has them join a room that is labeled as their username. Also sends a ‘response’ event back to the client to be run.
  - ‘Submission’ (lines 37-41)
    - This method takes the user's submitted message for all the users on the server. It sets the broadcast parameter to true so that it gets sent to all users on the server. Also sends a ‘my_response’ event back to the client to be run.
  - ‘Direct_message’ (lines 43-50)
    - This method takes the message to be sent to a specific user and the username of the user that it’s supposed to be sent to. It sends the message to their unique room and the user is able to receive the message. Also sends a ‘Direct’ event back to the client to be run.
  - ‘usernamechange’ (lines 52-63)
    - This method takes the new username that the user wants to change their name to and changes it in the database. It also changes our global list of online users and the name of the current user logged in so that the html template shows the new information on page load.
- In index.html the client has specific events that are called when a user does a specific action on the site or the server responds with a specific event that needs to be run. Lines (27-57)
  - ‘Connect’
    - Gets run on page load and emits an ‘connection’ event to the server to get put into a room.

- ‘Response’
  - Gets run after the connection event is run on the server. It tells the user that they have connected to the web socket.
- ‘My_response’
  - Gets run after the submission event is run on the server. It adds any messages that were sent from the server in the chat box.
- ‘Direct’
  - Gets run after the Direct_message event is run on the server. This event deals with a direct message from another user and creates a popup that shows the user who sent the message and what they sent.
- #sendbutton and #userchangebutton
  - These are both jquerys that emit a socket event to the server when:
    - A user submits a username change (userchangebutton)
    - A user sends a direct message to another user (sendbutton)

[flask_socketio.emit(event,*args, **kwargs)]

Purpose

- This method allows us to send an event to one or more connected clients in our socket server and communicate with them.
- It also allows us to send a JSON blob as payload with the event that we can use to display on the html.

What it does for us:

- This allows us to either send a message to all clients that are connected to our server passing in the element “broadcast = True” or only to one client if they are in a separate room passing in “to= (name of user’s room)”. We are also able to send events to our clients that are to be run on the site. The events are defined in the html as socket.on(“event name”,function(){“event goes here”}); and are defined as the first parameter in emit().

Specifically:

- We use emit for all of our websocket connections and it helps us to run events on our server and index.html and send information to and from the client without reloading the browser. On page load the client connects to our socket server at http:localhost:8080 and is then able to communicate with our server from emit. We explain these events in

the magic of SocketIO.

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

When a socket connection is created, the socket runs the “connection” event to connect the user to the server. Whenever an action is done that does something with the socket or an event is run, emit gets run to send an event to the server or client. Emit sends a custom event to the server or one or more connected clients. It has parameters such as event (event name as string), data (type of str, bytes, list, or dict), to: (parameter that holds the recipient socket name), broadcast (if True, sends a message to all clients), etc. When run, it gets called by the flask manager where it checks the kwargs and callback parameters to see if there are any changes. It then publishes the values inputted to the emit function and sends the message back to the client with the inputted fields that are serialized using the python pickle module.

- https://github.com/miguelgrinberg/python-socketio/blob/main/src/socketio/server.py#:~:text=def%20emit(self,callback%3Dcallback%2C%20**kwargs)

[flask_socketio.join_room()]

Purpose

- This function puts the user in a room under a namespace which is the string parameter that is imputed in this function.

What it does for us:

- This function allows us to separate the users into different rooms. Separating the different users allows us to be able to have a direct messaging system in our server with a world chat. Setting broadcast = True in emit sends a message to all users even if they are in different rooms so there is no disadvantage to putting them in different rooms. Also, because we put them in different rooms we are able to allow users to send direct messages to each other because their room space is named after their username and adding that to the to= parameter in emit will send the message directly.
- As explained before, we use join_room in the ‘connect’ event on our server.

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

- When a socket connection is created, the socket in the index.html runs the “connection” event to connect the user to the server. In this connection event we run join_room which takes a namespace string and calls the socketio.server.enter_room function in server.py in the flask documentation. Enter_room adds the client to a room based off of the namespace and creates a new one if the namespace does not exist. It also saves the name of the namespace and session ID of the client so that it can be used later.

- https://github.com/miguelgrinberg/Flask-SocketIO/blob/a10ea5cf65007061d7b3fd87b530c382007adebb/src/flask_socketio/__init__.py#:~:text=def%20join_room(room,room%2C%20namespace%3Dnamespace)

[flask_socketio.socketio.on()]

Purpose

- This allows us to register a socket event in our flask app

What it does for us:

- Being able to store the socket event allows us to manage the websocket handshake between the clients and call the events over the connection using the emit function.
- We use this to store all of our events and without it, we wouldn't be sending anything from server to user or vice versa.

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

- When a socket connection is created, the socket in the index.html runs the socket_io.on(“connect”) event to connect the user to the server. ‘Connect’, ‘message,’ and ‘disconnect’ are already event parameters that have been registered on runtime. At every instance of socketio.on a event is stored in a handler dictionary in self that is made when the server is initialized.

- https://github.com/miguelgrinberg/python-socketio/blob/3bd13578c82e8a94d6b0328180606c2aefd496f1/src/socketio/server.py#:~:text=def%20set_handler(handler,set_handler(handler)

[flask_socketio.socketio.connect()]

Purpose

- connect() allows us to connect the client to the server

What it does for us:

- It is used in our index.html file and without it, we could not make the web socket handshake between the client and the server.
- It establishes a connection to our server.

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 for you in the **Purpose** section of this report? Please explain this in detail, starting from after the TCP socket is created. Remember, to be allowed to use a technology in your project, you must be able to know how it works.

- After a TCP socket is created and the user logs into our application and is authenticated, they are taken to our home page “index.html” and establish a web socket connection with our server through io.connect and saves it as a socket variable which is used to keep the connection to the server and allows the client to handle events in index.html. The connect function in the Flask documentation takes the address to the server to connect to and creates an environment in the application and sends a packet to the server to establish and manage the connection.

- https://github.com/miguelgrinberg/Flask-SocketIO/blob/a10ea5cf65007061d7b3fd87b530c382007adebb/src/flask_socketio/test_client.py#:~:text=def%20connect(self,connected%5Bnamespace%5D%20%3D%20True

Some notes: We manually did xsrf, we used regular pymongo, not pymongo from db.