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.

AIOHTTP

General Information & Licensing

Code Repository	https://github.com/aio-libs/aiohttp
License Type	Apache 2 licensed
License Description	https://www.apache.org/licenses/LICENSE-2.0
License Restrictions	• Can't name product in any way to hint that it is endorsed by Apache (can say powered by Apache) • Must list out all modifications done to the original software
Who worked with this?	• Ethan Richardson • Zayaan Rahman • Cindy Cheng

web.Response() line 15, 30, 64

Purpose

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

- This tech returns a server response. The web.Response() object is created and takes in certain arguments. The first argument is the response body (taken in bytes). The second argument is the response status (200 OK by default but could be other things such as 404, or 301). The fourth parameter is an optional named “headers”. This is for any additional headers we may want to send. For example, an authentication token can be sent using this header option. We would just set a cookie as a key and the value it’s authentication token. There are also other optional parameters. This tech essentially allows us to send responses in a nice and organized manner whereas in the HW, we had to build functions to do it ourselves.
- web.Response() is used in lines 15, 30, and 64 in file Bullboard/backend/server.py. The object is returned to indicate that the server has responded to a request. Any time we finish handling a request, we should have the appropriate headers to send back to whoever requested the data which is why it happens at the end of the function.

Magic ⋆★˚｡⋆ ☽ ˚｡⋆ ★

https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web_response.py ->
https://github.com/aio-libs/aiohttp/blob/master/aiohttp/http.py ->
https://github.com/aio-libs/aiohttp/blob/master/aiohttp/http_writer.py

- Essentially, in lines 498-757, what is happening is that a class for Response is initialized with None values at first. This is so that later on when a Response object is created, we can pass in anything we want to that specific instance. Once we do create an object Response and pass in certain arguments, then the Response object is initialized with those parameters. However, this alone does not accomplish the server response. The server response needs to actually be written with the data we passed through as a constructor. This is shown in the functions in the above link, “write_eof”, “_start”, and some compression functions. “Write_eof” essentially keeps writing the body into bytes to actually send back to each request. “_start” initializes the content length to be used in the server response. The actual response with this information is used in a function in the same file called “_write_headers”.
- The function “_write_headers” actually utilizes an underlying library known as http.server which has a function defined as “write_headers”. The functions provided in the 3rd link is the end of the trace regarding writing the headers. This entire file handles writing the actual headers to send (you can see the bytes getting written in code, lines 29-200).

web.WebSocketResponse() line 41

Purpose

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

- This tech handles incoming connections and attempts to upgrade them to the WebSocket protocol. WebSocketResponse is a class in the aiohttp library that is used as the WebSocket itself. So for example, if we wanted to maintain a list of all connected clients, that means we would append each instance of the WebSocketResponse that was instantiated. This class has various methods used for the WebSocket functionality. For example, the _handshake method is used to actually perform the handshake necessary to upgrade the connection. In this method, you can see how the handshake performed. For example, at the end of the function they check if the key was correctly computed and see if it matches the client key to check if the connection is good to be upgraded. The WebSocketResponse has much more functionality and functions to perform various tasks for us. The ones that are used specifically are mentioned in the “Magic” section.
- This tech is used in line 41 specifically Bullboard/backend/server.py. We obtain an instance of the WebSocketResponse so that we can append it to a list of clients. This is specifically used for the live interaction aspect of the project. In this project's case, the live interaction involved a live canvas where data in JavaScript was sent to the server where this WebSocketResponse received that same information and we sent it back as well.

Magic ⋆★˚｡⋆୨୧˚ ☽ ˚⁀➷｡˚★ ⋆✴︎˚｡⋆ ༄

https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web_ws.py
https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web_response.py
https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web_request.py
https://github.com/aio-libs/yarl
https://github.com/aio-libs/aiohttp/blob/master/aiohttp/payload.py

- When a WebSocketResponse is first instantiated, the handshake is the first thing that must be performed. In lines 177, the _handshake method begins. First, we obtain the request headers. It utilizes a BaseRequest to do this which is created in web_response.py. The BaseRequest class has everything we need to actually write the payload in a WebSocket, handle the headers, and much more. This is in lines the web_request.py, lines 116-837. For example, the functions in lines 402, 432, and 437 all work together to obtain the path of the URL. The host function simply gets the HOST which is provided as data in the “hdrs” variable of the object. The URL is built using a URL building library known as yarl. In lines 194-261 of yarl/yarl/_url.py, that is where that library builds the URL. In lines 629 of web_request.py, this is where the request body is read and returned in bytes for the BaseRequest class. In lines 359-426 of web_response.py, this is where the headers for the WebSocket response are written. In the _prepare_headers method, a payload writer is used. The actual payload is written using functions from payload.py In this file, there are different types of payload depending on what the payload contains. For example, in lines 384, JsonPayload is used if JSON data is necessary to be sent whereas in lines 248, StringPayload is defined that is used for string data.

aiohttp.WSMsgType.TEXT line 47 and aiohttp.WSMsgType.ERROR line 55

Purpose

- This tech simply allows us to determine what type of message is being sent over the WebSocket. It is important to know what type of data is being sent over the WebSocket so that it can handle and parse it appropriately. If the message being sent over the WebSocket is text, then we can safely assume it is valid data. However, if it is of type ERROR, then we should not proceed with handling the message and instead force close the socket.
- This tech is used in lines 47 and 55 in Bullboard/backend/server.py. We use it here so that we can check the WebSocket message and act accordingly depending on it. If the message is of type ERROR then the WebSocket connection automatically closes.

Magic

https://github.com/aio-libs/aiohttp/blob/master/aiohttp/http_websocket.py
https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web_ws.py

- In lines 52-81, the WSMsgType class is written. Defined as state variables are some message types. The types our project deals with are TEXT (0x1) and ERROR (0x102). To obtain the message type, we must parse the payload first which is done in this file as well. For example, in lines 409, the method parse_frame is defined to do exactly that. We can see that in lines 428 to 432, the FIN, RSV1, RSV2, RSV3 and opcode bits are parsed using bit shifting and masking (which is what was done in HW3). Lines 456 and 457 check for the mask (if one exists) and obtains the payload length. The parsing continues until lines 562 where the WebSocket frame is returned. Since we must perform a comparison to check if the data over the WebSocket is equivalent to any of the WSMsgType's defined, we must access the WebSocket. You can see this by looking into web_ws.py. Throughout the code, the message is shown as of type WSMessage. This is shown in lines 84 of http_websocket.py. The message is obtained in lines 423-482 of web_ws.py. In this method, WSMsgType is used to return a WSMessage with a certain WSMsgType associated with it. This is so that in our project code, we can access websocket's message and message type in order to determine what we want to do with it.

web.Application() line 72

Purpose

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

- This function simply returns an instance of the server so that we can run it later.
- This tech is used in bullboard/backend/server.py lines 72. It is here because we simply put it there. This line could have technically been anywhere as long as it put somewhere before the server runs since we need an instance of web.Application() first before we run it.

Magic ⋆★˚｡⋆୨୧˚ ☽ ˚⁀➴｡˚★彡⋆ ༄

https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web_app.py
https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web_routedef.py
https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web_urldispatcher.py

- This tech simply gives us the actual server application to use later on. For example, in lines 104-127 in web_app.py, it initializes various things that can be used in the server. This includes a URL router which uses UrlDispatcher(). The UrlDispatcher is shown in web_urldispatcher.py lines 978. The UrlDispatcher is responsible for URL routing. There are also various functions in the UrlDispatcher that deal with the different types of requests (GET, POST, PUT, DELETE, etc). Back in web_app.py, in lines 329, a _handle method is defined to handle requests. This method utilizes the router to “resolve” the request. Routers are defined in web_routedef.py. For example, in lines 58 of web_routedef.py, it adds routes to the router. In lines 102, a route is defined as taking in a method, path, and a handler. This function returns a RouteDef which also takens in a method, path, and handler. The RouteDef class (in lines 58) uses the register method which utilizes a UrlDispatcher in order to actually add the route. This is the primary tech that was used from the web.Application().

app.add_routes, web.get() and web.post() (lines 73-93)

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?
    - web.get and web.post helps create an endpoint to the redirected paths that we can handle using specified functions. They take in two parameters, the first parameter being the path we are handling from GET or POST and the second parameter being our function handlers on the paths. The app.add_routes simply adds these routes to the object web.Application (explained in the previous section). At the end of app.add_routes, the server has access to all available routes so that we can handle each of them accordingly.

- We use web.get and web.post in our server.py file line 87-107 to add routes for web.application() app. app.add_routes is used in lines 73.

Magic ★★.˚⋆☽˚⌒✈︎.˚★彡⋆༄

https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web_routedef.py
https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web_app.py

- Part of web_routedef.py was explained in the previous Magic section. We can see that in lines 114-139 in web_routedef.py, several methods are defined to simply route each request. As stated in the “Purpose” section, web.get() or web.post() takes in two main parameters. One is the path and the other is the handler. The handlers are in Bullboard/backend/server.py, lines 8, 24, 40, and 62. We have a handler for GET requests, a handler for POST requests, the websocket handler and an image handler. We simply pass in the handler function as the second parameter and the path that is associated with that handler/request type. The functions in 114-139 will then return a route according to whatever it is handling. As an example, suppose the login path is requested. This is then associated with GET requests and the get_handler we wrote in Bullboard/backend/server.py. The function in 114 returns a route associated with /login, and get_handler. Now, since we have a route. We can call app.add_routes. This is in lines 269 of web_app.py. This function simply adds routes to the router so that the server can handle each request properly.

web.run_app() lines 96

Purpose

- This function actually runs the server by using an underlying library called asyncio. It initializes the server by running “event loops” which essentially listen for incoming requests. This loop will continue forever until the loop/server is shutdown.
- This is used in lines 96. It must be used after an instance of web.Application() is created since web.run_app() takes in that instance as a parameter.

Magic

https://github.com/aio-libs/aiohttp/blob/master/aiohttp/web.py
https://github.com/python/asyncio/blob/master/asyncio/__init__.py (the original repo is closed for some reason)
https://github.com/python/asyncio/blob/master/asyncio/locks.py
https://github.com/python/asyncio/blob/master/asyncio/events.py
https://github.com/python/asyncio/blob/master/asyncio/futures.py
https://github.com/python/asyncio/blob/master/asyncio/base_events.py

- This tech uses an underlying library known as asyncio. __init__.py simply initialises everything regarding the OS. asyncio is multithreaded so in locks.py, it utilizes mutexes and semaphores to handle concurrent threads. Asyncio utilizes what is known as “event loops”. This is shown in events.py. In lines 215-242, several methods are defined to run these events. Run_forever runs an event loop until stop() is called (which is defined in lines 226. Run_until_complete runs an event loop until a Future is done. A Future represents the outcome of an asynchronous operation. Because Futures are not thread safe, mutexes and semaphores (utilized in locks.py) must be used to protect these threads. So for example, if the server is going through an event, that event will continue until the event is complete. This is how the server works. The server can be thought of as an event. This event runs forever until it forces shutdown (for example using Ctrl-C will shut down it, which also counts as a Future) and the loop will stop and be forced to shut down. The actual server is run using all of the above. This is in base_events.py. In lines 186, a Server class is written. This server class is written by taking in an event loop, sockets, among other arguments. The event loop will be used to run the server forever. Several functions in this file utilize the host (for example, localhost) and the port number (for example, 8080). The function _ipaddr_info utilizes this information in lines 101. It handles the socket information so that the host and port can properly be used as a connection.