# Websockets

# FastAPI

## General Information & Licensing

| | |
|---|---|
| Code Repository | https://github.com/tiangolo/fastapi |
| License Type | MIT |
| License Description | • The license allows the software to be used and distributed in any way, including commercially |
| License Restrictions | • The only restriction when using the MIT license is that the software must include a notice containing copyright information, as well as that the notice that the software is free to use |

Magic

- The way FastAPI creates websockets is starting, as the TCP Server first receives and parses the headers in the HTTP request given. Assuming the request contains the necessary websocket upgrade headers Connection: Upgrade, Upgrade: websocket,  Sec-WebSocket-Key: *key*. Assuming these headers are used, FastAPI will begin the process of creating the websocket connection, verifying the headers and creating an asgi application to host the server. Once the server is established a request is sent back to the frontend verifying the websocket connection using the headers Connection: Upgrade, Upgrade: websocket, Sec-WebSocket-Accept: *key*. Once the connection is established a user can send chat messages or bids which gets parsed according to the websocket frame, along with terminating the connection.

- In our application, websockets are used for the auction. Once a user clicks the button "Set For Auction" seen here: 

an HTTP request is made to "GET

ws://localhost:8000/ws/auction/{auctionid}" with the type 101 Switching Protocols.
Once the handler in asyncio**events.py_run** (line 80) receives the proper http request containing the websocket upgrade and the headers finish parsing in https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/httptools_impl.py (line 252). A call is made to

/uvicorn/protocols/http/httptools_impl.py/**_get_upgrade** (line 151) which will check if connection and upgrade are in the headers and return True if so, which it will in the case of a websocket upgrade will do so.

The next method (same dir)/**_should_upgrade_to_ws**() takes result of previous method and if upgrade is = b"websocket" return True  if True upgrading on_headers_complete immediately returns nothing.

ibid/**on_message_complete**() runs should_upgrade which runs get_upgrade and should_upgrade_to_ws() again which returns the same result once these return the function **on_message_complete**() returns nothing

an exception is reached httptools.HttpParserUpgrade which runs self.**handle_websocket_upgrade**()  which sets some variables and importantly creates object of **WebSocketProtocol**
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/websockets/websockets_impl.py

```
57  class WebSocketProtocol(WebSocketServerProtocol):
58      extra_headers: List[Tuple[str, str]]
59
60      def __init__(
61          self,
62          config: Config,
63          server_state: ServerState,
64          _loop: Optional[asyncio.AbstractEventLoop] = None,
65      ):
```

Cont.

```
101             super().__init__(
102                 ws_handler=self.ws_handler,
103                 ws_server=self.ws_server,  # type: ignore[arg-type]
104                 max_size=self.config.ws_max_size,
105                 ping_interval=self.config.ws_ping_interval,
106                 ping_timeout=self.config.ws_ping_timeout,
107                 extensions=extensions,
108                 logger=logging.getLogger("uvicorn.error"),
109             )
```

This class importantly has a field self.**ws_server** which creates a **Server** Object and notable variables of **ws_server** and **ws_handler** along with running functions protocol.connection_made, data_receieved and set_protocol.

uvicorn\protocols\websockets\websockets_impl.py/**connection_made** sets variables notably server and client and also calls a different method much to my confusion also named **connection_made** located here:
https://github.com/aaugustin/websockets/blob/main/src/websockets/legacy/server.py

```
140     def connection_made(self, transport: asyncio.BaseTransport) -> None:
141         """
142         Register connection and initialize a task to handle it.
143 
144         """
145         super().connection_made(transport)
146         # Register the connection with the server before creating the handler
147         # task. Registering at the beginning of the handler coroutine would
148         # create a race condition between the creation of the task, which
149         # schedules its execution, and the moment the handler starts running.
150         self.ws_server.register(self)
151         self.handler_task = self.loop.create_task(self.handler())
```

**connection_made** "Registers the websocket connection with the server before creating the handler" this method does several important things
notably setting a write limit of **65536 bytes** (rather than the maximum size of 9,223,372,036,854,775,807 bytes) for the size of a websocket frame. Along with creating the handler. This method calls two methods:

self.**ws_server.register(self)**
self.handler_task = self.**loop.create_task(self.handler())**

The first one registers and creates a ws_server method while the second creates a task that handles websocket requests.

We are now back in **handle_websocket_upgrade**, with two methods left, data_received and set_protocol.

The **data_received** method is..... very straightforward as you can see in it's definition here:
asyncio\protocols.py

**set_protocol** on the other hand is.... also straightforward and sets the websocket protocol to our protocol object with its websocket server, handler, as well as other good methods such as the aforementioned frame limit.

Finally, we are finished! the functions all escape and we are back at the _run call and by finished I meant finished with getting everything ready for the websocket handshake!
A call will be made to the previously mentioned handler method which will first start with the websocket handshake found at line 165, with calls to another function at line 593. This

function processes the incoming HTTP request, reads the http_request at line 587 and processes a request at line 591. read_http_request() is called first which is similar to other **read_http_request**() methods, another call is made to process_request.

**process_request** prepares the websocket to start the ASGI application, it does this by verifying the authenticity of the necessary websocket headers: notably the websocket key.

Once that is done a task to run ASGI with the necessary headers is sent to the task queue and called by **events.py/_run** once it's ready.

Once it's ready a call to **run_asgi** on line 232 at:
uvicorn\protocols\websockets\websockets_impl.py is made to __call__
located at:
https://github.com/encode/uvicorn/blob/master/uvicorn/middleware/proxy_headers.py
at line 49, this method in this case basically determines whether asgi is handling an http request or a websocket request. It concludes by returning a call to self.app located here:
\Python310\site-packages\fastapi\applications.py

This method will then wait on a super method located on starlette\applications.py at line 123 and waits on a call to middleware_stack() located at: starlette\middleware\errors.py

This method will determine that the type is not http, in our case it will be 'websocket' cause as you've guessed we are trying to set up the websocket handshake.

This method waits on another method called app located here:
https://github.com/encode/starlette/blob/master/starlette/middleware/cors.py

This method does the exact same thing as the previous except it waits on another app method located here:
https://github.com/tiangolo/fastapi/blob/master/fastapi/middleware/asyncexitstack.py

This method does, if you couldn't guess, wait on another app method located here:
https://github.com/encode/starlette/blob/master/starlette/routing.py at line 685.

```
685     async def __call__(self, scope: Scope, receive: Receive, send: Send) -> None:
686         """
687         The main entry point to the Router class.
688         """
689         assert scope["type"] in ("http", "websocket", "lifespan")
690 
691         if "router" not in scope:
692             scope["router"] = self
693 
694         if scope["type"] == "lifespan":
695             await self.lifespan(scope, receive, send)
696             return
697 
698         partial = None
699 
700         for route in self.routes:
701             # Determine if any route matches the incoming scope,
702             # and hand over to the matching route if found.
703             match, child_scope = route.matches(scope)
704             if match == Match.FULL:
705                 scope.update(child_scope)
706                 await route.handle(scope, receive, send)
707                 return
708             elif match == Match.PARTIAL and partial is None:
709                 partial = route
710                 partial_scope = child_scope
711 
```

This method however, is finally the end of the long trail of waits and starts by trying to find a route that matches the incoming scope and handles it. It does this through checking all the routes and looking for one with a websocket type. Once a match is found it awaits on a handle method located at starlette\routing.py on line 340 which waits on..... another self.app method.

However this one is located in a method called websocket_session located here: starlette\routing.py this one creates a **WebSocket**() object.

The WebSocket() class is located at
https://github.com/encode/starlette/blob/master/starlette/websockets.py
on line 21 and verifies that the type is websocket along with assigning some variables, notably that the **WebScoketState** which is set to **CONNECTING**.

Once this method completes we are back at websocket_session which is now waiting on a method called func, func makes a call to **get_websocket_app**() in https://github.com/tiangolo/fastapi/blob/master/fastapi/routing.py on line 277.

This calls a method waits on a method called **solve_dependencies()** located at https://github.com/tiangolo/fastapi/blob/master/fastapi/dependencies/utils.py on line 473.

This method verifies the variables given and completes returning back to

**get_websocket_app**() this method then awaits on a .**call**() methods

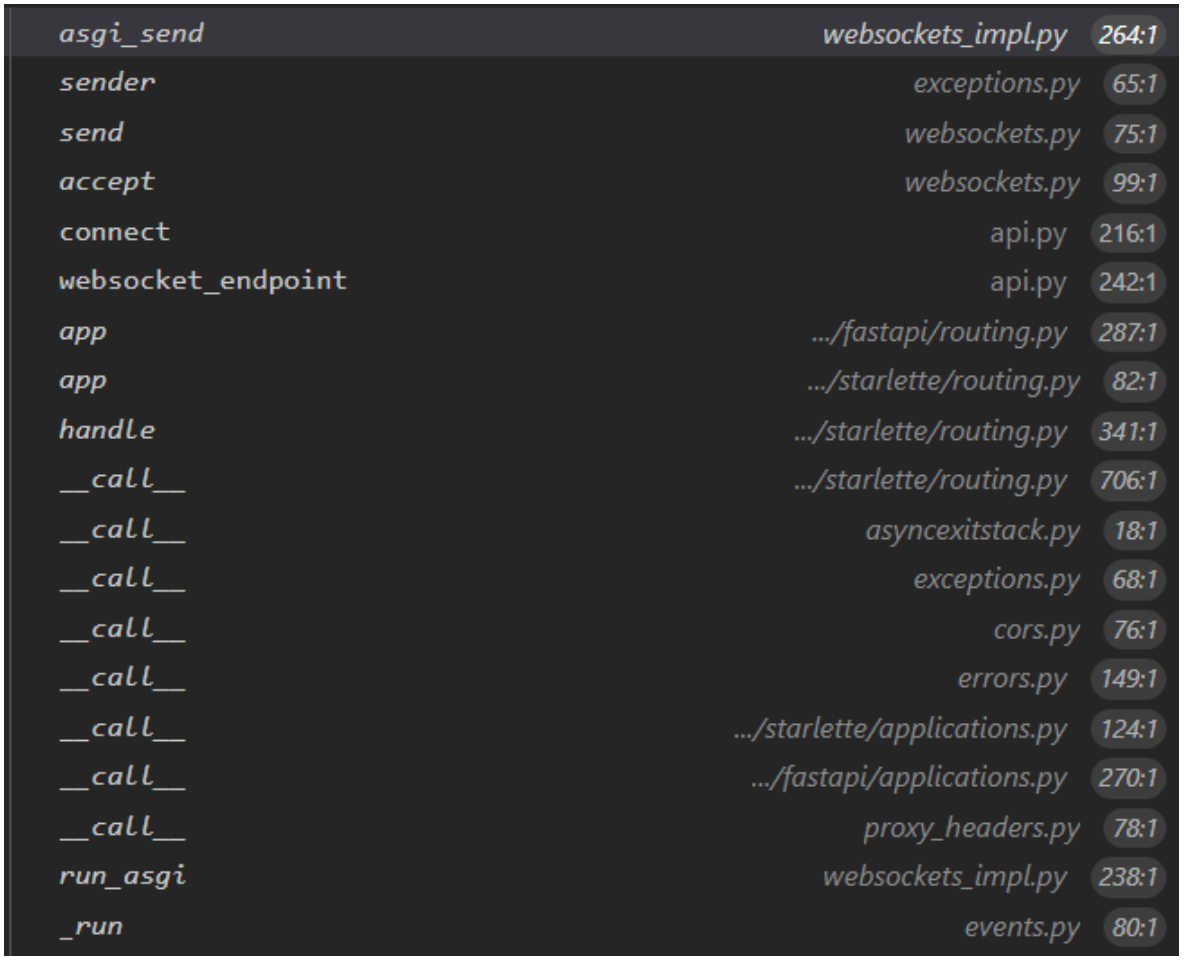
This .call() method is located inside of **our api.py file**! on line 241. This method will await on the **ConnectionManager**() to finish the **connect**() method located on line 216.
The connect method will call and wait on the websocket object's accept method.
This method takes us out of api.py and to starlette\websockets.py on line 94

as the **WebSocketState** is connecting it will await on a receive method() located at line 30 in the same file and assigns a variable message to the result of a receive() method that returns a dictionary with
the key-value pair: {'type' : 'websocket.connect'}

It will return this key-value pair, which is one after verifying and await on another method called send located at
https://github.com/encode/starlette/blob/master/starlette/websockets.py on line 60.

This method will set the state of the **WebSocketState** to **CONNECTED** and wait on a send method located at
https://github.com/encode/starlette/blob/master/starlette/exceptions.py
which will wait on another send after verifying the type is not "http.response.start'. this send is located at: uvicorn\protocols\websockets\websockets_impl.py on line 262 named asgi_send(). and will check the type to ensure it is "websocket.accept" and will cast a "WebSocketAcceptEvent", it will also print a message to the console saying the WebSocket was accepted.
It also gets any subprotocols which there shouldn't be along with calling the set() method in handshake_started_event.

The stack trace during the asgi_send method: Note that we started at events.py




Finally, the long list of wait calls comes to a close with api.py's connect method adding the page to the list of active_connections.

At this point the websocket handshake has completed after a very very long list of waits and we are finally ready to receive and parse data!

A try statement is created that disconnects the websocket on a **WebSocketDisconnect** message.

```
240  @app.websocket("/ws/auction/{auctionID}")
241  async def websocket_endpoint(websocket: WebSocket, auctionID: str):
242      await manager.connect(websocket, auctionID)
243      try:
244          while True:
245              data = await websocket.receive_text()
246              data = json.loads(data)
247              await manager.broadcast(data, auctionID)
248
249      except WebSocketDisconnect:
250          manager.disconnect(websocket)
251
```

Inside the try statement is a loop that never ends until the aforementioned try statement is reached. This calls and waits on the method websocket.receieve_text() which is located here: starlette\websockets.py on line 107.
This method will assign a variable message to the result of a receive call that it waits on located in the same file on line 30.
We have gone through this method before but as the WebSocketState is now Connected

the behavior is different. It will start by assigning a variable message to a receive call located at uvicorn\protocols\websockets\websockets_impl.py on line 331.
Similar to the last message the behavior is now different as we have established the websocket connection and will wait on another method handshake_completed_event.wait(). In the event that this method returns **lost_connection_before_handshake** the websocket will return a disconnect message.

If the websocket connection isn't lost the once data is received it will be turned into a msg object which is then decoded and parsed following the format in a **WebSocketReceiveEvent** for websocket frames. Finally, once the message is parsed it is returned which is assigned to data and broadcasted through the connection manager into a json format as seen on lines 246 and 247 of api.py.

The connection manager will finally send the json_data to all users currently connected to the websocket session as seen at line 234 of api.py which completes the process of the websocket handshake, receiving, parsing, and sending data using websockets!