Open-Source Report

Proof of knowing your stuff in CSE312

Guidelines

Provided below is a template you must use to write your reports for 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 need 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.

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.

FastAPI

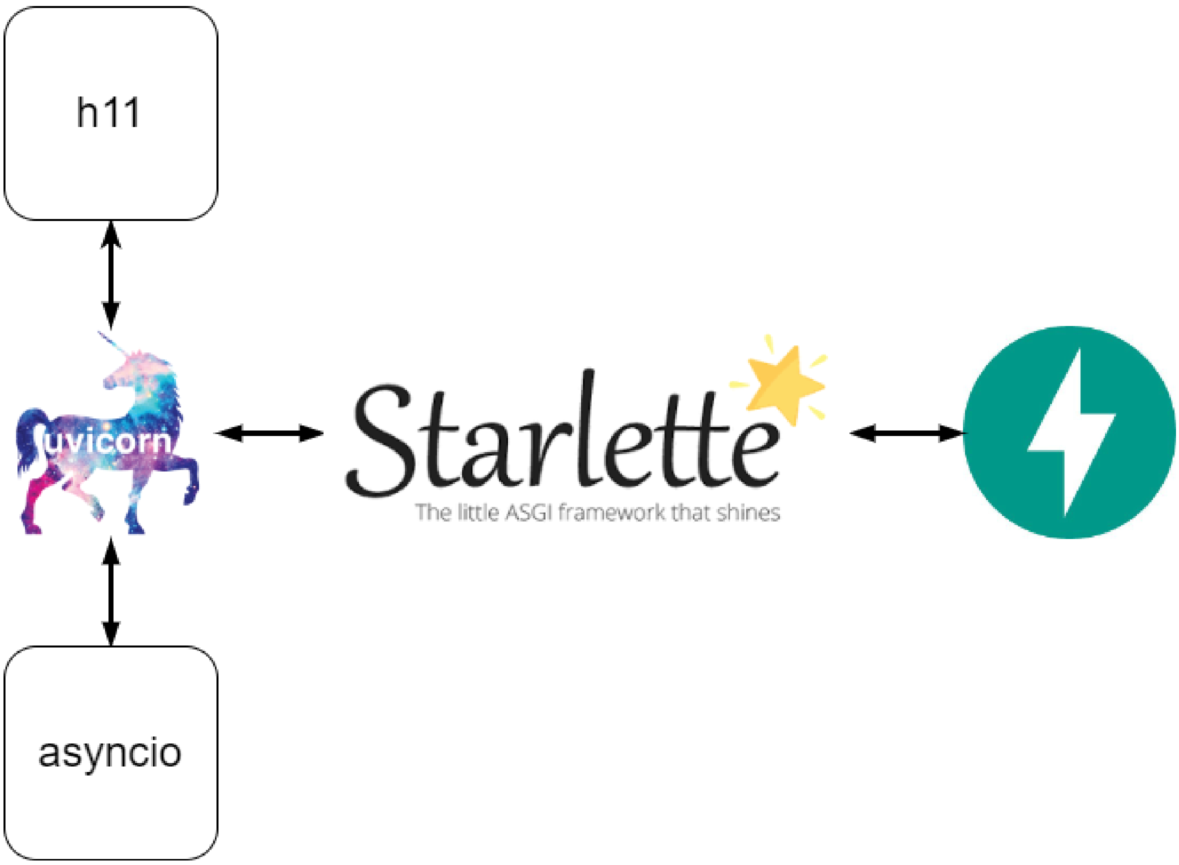
General Information & Licensing

Code Repository	https://github.com/tiangolo/fastapi
License Type	MIT License
License Description	• Permissive license • Allows unrestricted use without any limitation free of charge • Protects the copyright holders from liability for any consequences caused by anything utilizing the licensed code
License Restrictions	• Derivative code must also be licensed under the MIT License

Magic

FastAPI is a Python web framework that allows for rapid development of REST APIs. It is built on top of Starlette, which is a lightweight ASGI framework. FastAPI and Starlette do not ship with server code, but rather rely on an additional ASGI webserver to handle network connections and pass data to the Python application following the ASGI specification. As such, understanding the TCP connection requires understanding the ASGI webserver.

For our implementation, we used the ASGI webserver uvicorn. This is the program that actually opens sockets and handles network connections, so it will be the focus of the following discussion. At a high level, uvicorn opens a socket by invoking an API in the Python standard library asyncio. Asyncio then handles connection requests, reads data off of the TCP socket, and sends this data back to uvicorn via an asyncio protocol. Uvicorn then passes the raw bytes to the Python library h11 (which stands for HTTP 1.1). This library parses the bytes into a request object, which uvicorn then packs into an ASGI message and sends to FastAPI and Starlette. FastAPI and Starlette then generate a request object from the ASGI message, pass it through a series of middleware, and finally pass it to the application code. The application code generates the response, the response is returned to FastAPI and Starlette, and the process repeats in reverse. A diagram of the information flow is shown below.



Since we did our development on Windows, we will be describing the chain of calls that use the Windows APIs. Specifically, the library used to interact with the TCP socket, asyncio, uses different Python modules to manage sockets depending on the platform. Since we are using Windows with CPython, the modules of interest are _winapi and _overlapped. The functions in these modules perform the system calls that are necessary to perform the TCP socket IO for Windows devices. The full chain of calls is described below.

First, we call uvicorn.run() on line 144 in main.py to start our ASGI webserver
https://github.com/mattrrubino/cse312-group-project/blob/main/api/src/main.py#L144

```
142
143 if __name__ == "__main__":
144     uvicorn.run("main:app", host="0.0.0.0", reload=DEV, proxy_headers=True)
```

uvicorn.run() then initializes a config, instantiates a Server object, and then calls the Server class's run() function on line 578 in uvicorn's main.py
https://github.com/encode/uvicorn/blob/master/uvicorn/main.py#L578

```
576             Multiprocess(config, target=server.run, sockets=[sock]).run()
577         else:
578             server.run()
579         if config.uds and os.path.exists(config.uds):
580             os.remove(config.uds)  # pragma: py-win32
```

The Server's run() method then runs the Server's serve() method in a new asyncio event loop on line 61 of server.py. This starts the asynchronous processing, which will help in achieving high throughput
https://github.com/encode/uvicorn/blob/master/uvicorn/server.py#L61

```
59     def run(self, sockets: Optional[List[socket.socket]] = None) -> None:
60         self.config.setup_event_loop()
61         return asyncio.run(self.serve(sockets=sockets))
```

The Server's serve() method loads the config, logs some information, and then yields control to its startup() method on line 78 of server.py
https://github.com/encode/uvicorn/blob/master/uvicorn/server.py#L78

```
76         logger.info(message, process_id, extra={"color_messag
77
78         await self.startup(sockets=sockets)
79         if self.should_exit:
80             return
```

The startup method defines a factory function create_protocol() on line 96. This factory function takes in an asyncio event loop and returns an asyncio protocol object. Asyncio will use this factory function to generate a protocol object. This protocol object is the mechanism by which asyncio will eventually pass socket connection information and socket data back to uvicorn (more on this later). Finally, note that the default protocol specified in the configuration is the H11Protocol (standing for HTTP 1.1), which is defined in uvicorn/protocols/http/h11_impl.py
https://github.com/encode/uvicorn/blob/master/uvicorn/server.py#L96
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L78

```python
        config = self.config

        def create_protocol(
            _loop: Optional[asyncio.AbstractEventLoop] = None,
        ) -> asyncio.Protocol:
            return config.http_protocol_class(  # type: ignore[call-arg]
                config=config,
                server_state=self.server_state,
                app_state=self.lifespan.state,
                _loop=_loop,
            )
```

```python
class H11Protocol(asyncio.Protocol):
    def __init__(
        self,
        config: Config,
        server_state: ServerState,
        app_state: Dict[str, Any],
        _loop: Optional[asyncio.AbstractEventLoop] = None,
    ) -> None:
```

Back to the Server's startup() method, this routine invokes the asyncio event loop method create_server() on line 161, passing the protocol factory function described above. This is how uvicorn offloads the socket management to asyncio
https://github.com/encode/uvicorn/blob/master/uvicorn/server.py#L161

```python
160             try:
161                 server = await loop.create_server(
162                     create_protocol,
163                     host=config.host,
164                     port=config.port,
165                     ssl=config.ssl,
166                     backlog=config.backlog,
167                 )
```

We will now look at the source code of asyncio, since this is where the socket management occurs

The create_server() method is defined on line 1444 of asyncio/base_events.py. Critically, this method creates an instance of the Python socket object using the information specified in the configuration on line 1513
https://github.com/python/cpython/blob/main/Lib/asyncio/base_events.py#L1513

```
1509            try:
1510                for res in infos:
1511                    af, socktype, proto, canonname, sa = res
1512                    try:
1513                        sock = socket.socket(af, socktype, proto)
1514                    except socket.error:
```

This socket object is then bound to the IP and port based on the configuration information on line 1537. This is necessary so that the operating system can verify that the specified address and port are unused
https://github.com/python/cpython/blob/main/Lib/asyncio/base_events.py#L1537

```
1536                    try:
1537                        sock.bind(sa)
1538                    except OSError as err:
1539                        raise OSError(err.errno, 'error while attempting '
1540                                      'to bind on address %r: %s'
```

This socket object is then passed to an instance of the asyncio Server class, and the _start_serving() method is called on the server object on line 1561
https://github.com/python/cpython/blob/main/Lib/asyncio/base_events.py#L1561

```
1560        if start_serving:
1561            server._start_serving()
1562            # Skip one loop iteration so that all
```

The _start_serving() method starts the listening socket on line 315, which is necessary to start accepting connections. It then runs the _start_serving() method on the loop object on line 316
https://github.com/python/cpython/blob/main/Lib/asyncio/base_events.py#L316

```
314         for sock in self._sockets:
315             sock.listen(self._backlog)
316             self._loop._start_serving(
317                 self._protocol_factory, sock, self._ssl_context,
318                 self, self._backlog, self._ssl_handshake_timeout,
319                 self._ssl_shutdown_timeout)
```

The loop object is an instance of the BaseProactorEventLoop class. This class manages an asyncio event loop. The _start_serving() method of the BaseProactorEventLoop is defined on line 836 of proactor_events.py, which schedules the nested function loop() to run on the next event loop cycle on line 879 using the call_soon() routine
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L879

```
878
879         self.call_soon(loop)
880
```

In the next event loop cycle, the inner loop() function runs. In this function, the accept() method of the proactor object is called on line 861 and assigned to the variable f. This function returns a future which completes when a new TCP connection is accepted. When this occurs, the loop function will run again. This is because the function is scheduled as a done callback of the future on line 877, effectively turning the function into an infinite loop
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L861
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L877

```
859                 if self.is_closed():
860                     return
861                 f = self._proactor.accept(sock)
862             except OSError as exc:
863                 if sock.fileno() != -1:
864                     self.call_exception_handler({
```

```
875             else:
876                 self._accept_futures[sock.fileno()] = f
877                 f.add_done_callback(loop)
878
```

Inside the future, the function waits for connection requests. On Windows, the proactor object mentioned above is an instance of ProactorEventLoop (defined in windows_events.py). The accept() method on this class is defined on line 543. This method retrieves the information of the remote socket using the _get_accept_socket() method on line 545. It then accepts the TCP connection using the AcceptEx function from the _overlapped module (this comes directly from the Windows API, it is invoked using a Python C extension module) on line 547
https://github.com/python/cpython/blob/main/Lib/asyncio/windows_events.py#L547

```
543     def accept(self, listener):
544         self._register_with_iocp(listener)
545         conn = self._get_accept_socket(listener.family)
546         ov = _overlapped.Overlapped(NULL)
547         ov.AcceptEx(listener.fileno(), conn.fileno())
```

When the future completes, the loop() function is executed as a callback. This time, the variable f holds the future that just completed, so the connection and address can be retrieved on line 844
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L844

```
843             if f is not None:
844                 conn, addr = f.result()
845                 if self._debug:
846                     logger.debug("%r got a new connection from %r: %r",
```

The socket transport is then created on line 856, passing the connection information and the uvicorn-supplied protocol implementation
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L856

```
855                     else:
856                         self._make_socket_transport(
857                             conn, protocol,
858                             extra={'peername': addr}, server=server)
```

The _make_socket_transport() function creates a new instance of _ProactorSocketTransport and returns it on line 647
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L647

```
645     def _make_socket_transport(self, sock, protocol, waiter=None,
646                                extra=None, server=None):
647         return _ProactorSocketTransport(self, sock, protocol, waiter,
648                                         extra, server)
```

Inside the constructor of _ProactorSocketTransport, something critical happens. Specifically, _ProactorSocketTransport extends _ProactorReadPipeTransport, which extends _ProactorBasePipeTransport, and something important happens in the constructor of _ProactorBasePipeTransport. On line 67, the connection_made() function of the protocol is scheduled to run in the event loop, putting it inside the loop's queue _ready. Specifically, this will be the function H11Protocol.connection_made(). This function is the mechanism by which asyncio passes connection information back to uvicorn
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L67

```
65          if self._server is not None:
66              self._server._attach()
67          self._loop.call_soon(self._protocol.connection_made, self)
68          if waiter is not None:
69              # only wake up the waiter when connection_made() has been called
```

After connection_made() is scheduled, the function _loop_reading() is also scheduled with the event loop, similarly putting it inside the _ready queue. This occurs on line 192 in the constructor of _ProactorReadPipeTransport. Importantly, this routine contains the code for reading bytes off of the TCP connection
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L192

```
191         self._data = bytearray(buffer_size)
192         self._loop.call_soon(self._loop_reading)
193         self._paused = False
```

As the event loop runs, it will first pop the handle for the H11Protocol.connection_made function off of the _ready queue on line 1936
https://github.com/python/cpython/blob/main/Lib/asyncio/base_events.py#L1936

```
1934            ntodo = len(self._ready)
1935            for i in range(ntodo):
1936                handle = self._ready.popleft()
1937                if handle._cancelled:
1938                    continue
```

It then runs the handle on line 1951
https://github.com/python/cpython/blob/main/Lib/asyncio/base_events.py#L1951

```
1950                else:
1951                    handle._run()
1952            handle = None  # Needed to break cycles when an exception occurs.
```

This triggers the connection handling code on uvicorn on line 128 of uvicorn/protocols/http/h11_impl.py, which registers the connection with uvicorn
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L128

```
127     # Protocol interface
128     def connection_made(  # type: ignore[override]
129         self, transport: asyncio.Transport
130     ) -> None:
131         self.connections.add(self)
```

On the next tick of the event loop, it pops off the handle for _ProactorReadPipeTransport._loop_reading() off of the _ready queue on line 1936
https://github.com/python/cpython/blob/main/Lib/asyncio/base_events.py#L1936

```
1934            ntodo = len(self._ready)
1935            for i in range(ntodo):
1936                handle = self._ready.popleft()
1937                if handle._cancelled:
1938                    continue
```

It then runs the handle on line 1951
https://github.com/python/cpython/blob/main/Lib/asyncio/base_events.py#L1951

```
1950                else:
1951                    handle._run()
1952            handle = None  # Needed to break cycles when an exception occurs.
```

This triggers the reading of data from the TCP socket. Specifically, it triggers the _loop_reading() method on line 276 of asyncio/proactor_events.py. This calls the method recv_into on the _proactor object on line 306, which reads data from the supplied socket

into the supplied buffer _data
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L306

```
304                 if not self._paused:
305                     # reschedule a new read
306                     self._read_fut = self._loop._proactor.recv_into(self._sock, self._data)
307             except ConnectionAbortedError as exc:
308                 if not self._closing:
```

The _proactor object is an instance of IocpProactor, so this triggers the receive_into() method on line 482 of asyncio/window_events.py. This runs the WSARecvInto() method from the _overlapped module on line 487. This again invokes the [Windows API](), but this time for reading data from the socket
https://github.com/python/cpython/blob/main/Lib/asyncio/windows_events.py#L487

```
485         try:
486             if isinstance(conn, socket.socket):
487                 ov.WSARecvInto(conn.fileno(), buf, flags)
488             else:
489                 ov.ReadFileInto(conn.fileno(), buf)
```

After reading some data into the buffer, the _loop_reading() method reschedules itself on line 322. This is necessary in case the first read did not read all of the data. This implements the buffering of TCP data recursively
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L322

```
320         else:
321             if not self._paused:
322                 self._read_fut.add_done_callback(self._loop_reading)
323         finally:
```

The data in the buffer is then passed to the _data_received() method in the finally block on line 325
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L325

```
323         finally:
324             if length > -1:
325                 self._data_received(data, length)
```

The _data_receieved() method performs some checks of the supplied inputs and then passes them to the H11Protocol instance from uvicorn on line 274
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L274

```
272                 return
273         else:
274             self._protocol.data_received(data)
275
```

Inside the data_receive() method of uvicorn, the data is passed into the receive_data() method of the connection object on line 201. This attaches the received data buffer to the connection
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L201

```
198     def data_received(self, data: bytes) -> None:
199         self._unset_keepalive_if_required()
200
201         self.conn.receive_data(data)
202         self.handle_events()
```

On the next line of the data_receive() method, handle_events() is called, which will process the data attached to the connection in the previous step
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L202

```
198     def data_received(self, data: bytes) -> None:
199         self._unset_keepalive_if_required()
200
201         self.conn.receive_data(data)
202         self.handle_events()
```

Inside of handle_events(), the previous event data is retrieved using the method next_event() on line 207. The details of this method are described below
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L207

```
204     def handle_events(self) -> None:
205         while True:
206             try:
207                 event = self.conn.next_event()
208             except h11.RemoteProtocolError:
```

Inside next_event(), the method _extract_next_receive_event() is called in h11/_connection.py on line 469
https://github.com/python-hyper/h11/blob/master/h11/_connection.py#L469

```
468         try:
469             event = self._extract_next_receive_event()
470             if event not in [NEED_DATA, PAUSED]:
471                 self._process_event(self.their_role, cast(Event, event))
```

Inside _extract_next_receive_event(), the data in the receive buffer is passed to the _reader() function on line 411
https://github.com/python-hyper/h11/blob/master/h11/_connection.py#L411

```
410         assert self._reader is not None
411         event = self._reader(self._receive_buffer)
412         if event is None:
```

The _reader identifier is a reference to the function _maybe_read_from_IDLE_client() on line 75 of h11/_readers.py. This function then calls the method maybe_extract_lines() on the buffer on line 76
https://github.com/python-hyper/h11/blob/master/h11/_readers.py#L76

```
75 def maybe_read_from_IDLE_client(buf: ReceiveBuffer) -> Optional[Request]:
76     lines = buf.maybe_extract_lines()
77     if lines is None:
78         if buf.is_next_line_obviously_invalid_request_line():
```

The maybe_extract_lines() method parses the byte buffer into a list of bytearrays, where each bytearray represents a line of the HTTP request. It does this by performing a split on “\n” on line 126 in h11/_receivebuffer.py (and then removing trailing “\r”s)
https://github.com/python-hyper/h11/blob/master/h11/_receivebuffer.py#L126

```
123         # Truncate the buffer and return it.
124         idx = match.span(0)[-1]
125         out = self._extract(idx)
126         lines = out.split(b"\n")
127
128         for line in lines:
129             if line.endswith(b"\r"):
130                 del line[-1]
```

The HTTP request line is then validated on line 411 of h11/_readers.py because you can never trust your users!
https://github.com/python-hyper/h11/blob/master/h11/_readers.py#L83

```
82             raise LocalProtocolError("no request line received")
83     matches = validate(
84         request_line_re, lines[0], "illegal request line: {!r}", lines[0]
85     )
```

Finally, the reader parses the headers (described in detail in the header report) and returns a Request object that wraps the parsed data on line 86. This object is then returned all the way back up to handle_events()
https://github.com/python-hyper/h11/blob/master/h11/_readers.py#L86

```
86     return Request(
87         headers=list(_decode_header_lines(lines[1:])), _parsed=True, **matches
88     )
```

The event has type Request, so the conditional on line 226 of uvicorn/protocols/http/h11_impl.py evaluates to true. In this branch, an ASGI scope dictionary is created on line 229. This dictionary follows the ASGI specification, and is necessary for passing a message to an ASGI application like Starlette
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L229

```
228             raw_path, _, query_string = event.target.partition(b"?")
229             self.scope = {  # type: ignore[typeddict-item]
230                 "type": "http",
231                 "asgi": {
232                     "version": self.config.asgi_version,
233                     "spec_version": "2.3",
234                 },
235                 "http_version": event.http_version.decode("ascii"),
236                 "server": self.server,
```

This scope object is passed into an instance of RequestResponseCycle created on line 264, and the ASGI request is scheduled to run in the event loop on line 276
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L276

```
275             )
276             task = self.loop.create_task(self.cycle.run_asgi(app))
277             task.add_done_callback(self.tasks.discard)
278             self.tasks.add(task)
```

Inside the run_asgi() method, the ASGI app (Starlette/FastAPI) is invoked on line 428 according to the ASGI specification
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L428

```
426     async def run_asgi(self, app: "ASGI3Application") -> None:
427         try:
428             result = await app(  # type: ignore[func-returns-value]
429                 self.scope, self.receive, self.send
430             )
```

Uvicorn automatically wraps the user's application in middleware, so this middleware gets invoked first. Specifically, the invocation mentioned previously triggers the __call__ method of the ProxyHeadersMiddleware on line 49, which automatically managers header information for proxies
https://github.com/encode/uvicorn/blob/master/uvicorn/middleware/proxy_headers.py#L49

```
48
49      async def __call__(
50          self, scope: "Scope", receive: "ASGIReceiveCallable", send: "ASGISendCallable"
51      ) -> None:
52          if scope["type"] in ("http", "websocket"):
53              scope = cast(Union["HTTPScope", "WebSocketScope"], scope)
54              client_addr: Optional[Tuple[str, int]] = scope.get("client")
55              client_host = client_addr[0] if client_addr else None
56
```

This middleware then invokes the FastAPI application on line 78
https://github.com/encode/uvicorn/blob/master/uvicorn/middleware/proxy_headers.py#L78

```
76                  scope["client"] = (host, port)  # type: ignore[arg-type]
77
78          return await self.app(scope, receive, send)
```

The FastAPI application entrypoint is the __call__ method of the FastAPI type, defined on line 273 of fastapi/applications.py. This method is also a form of middleware, simply passing the ASGI message to the superclass Starlette on line 276
https://github.com/tiangolo/fastapi/blob/master/fastapi/applications.py#L276

```
273     async def __call__(self, scope: Scope, receive: Receive, send: Send) -> None:
274         if self.root_path:
275             scope["root_path"] = self.root_path
276         await super().__call__(scope, receive, send)
```

The Starlette application entrypoint is the __call__ method of the Starlette type, defined on line 118 of starlette/applications.py. This method assembles the middleware stack if it does not exist and then invokes it on line 122
https://github.com/encode/starlette/blob/master/starlette/applications.py#L122

```
118     async def __call__(self, scope: Scope, receive: Receive, send: Send) -> None:
119         scope["app"] = self
120         if self.middleware_stack is None:
121             self.middleware_stack = self.build_middleware_stack()
122         await self.middleware_stack(scope, receive, send)
```

The ASGI request data must then be passed through Starlette's entire middleware stack. Specifically, it starts by going through ServerErrorMiddleware in errors.py, then ExceptionMiddleware in exceptions.py, then AsyncExitStackMiddleware in asyncexitstack.py, and then it finally reaches Router in routing.py
https://github.com/encode/starlette/blob/master/starlette/middleware/errors.py#L147
https://github.com/encode/starlette/blob/master/starlette/middleware/exceptions.py#L53
https://github.com/tiangolo/fastapi/blob/master/fastapi/middleware/asyncexitstack.py#L12
https://github.com/encode/starlette/blob/master/starlette/routing.py#L697

```
147     async def __call__(self, scope: Scope, receive: Receive, send: Send) -> None:
148         if scope["type"] != "http":
149             await self.app(scope, receive, send)
150             return
```

```
53      async def __call__(self, scope: Scope, receive: Receive, send: Send) -> None:
54          if scope["type"] not in ("http", "websocket"):
55              await self.app(scope, receive, send)
56              return
```

```
12      async def __call__(self, scope: Scope, receive: Receive, send: Send) -> None:
13          if AsyncExitStack:
14              dependency_exception: Optional[Exception] = None
15              async with AsyncExitStack() as stack:
16                  scope[self.context_name] = stack
```

```
    async def __call__(self, scope: Scope, receive: Receive, send: Send) -> None:
        """
        The main entry point to the Router class.
        """
        assert scope["type"] in ("http", "websocket", "lifespan")
```

Inside the router middleware, a loop over all the routes is performed on line 712. When a route that matches the path is found, the corresponding handle method is called on line 718
https://github.com/encode/starlette/blob/master/starlette/routing.py#L718

```
712         for route in self.routes:
713             # Determine if any route matches the incoming scope,
714             # and hand over to the matching route if found.
715             match, child_scope = route.matches(scope)
716             if match == Match.FULL:
717                 scope.update(child_scope)
718                 await route.handle(scope, receive, send)
719                 return
```

The handle method then runs the supplied ASGI app on line 276
https://github.com/encode/starlette/blob/master/starlette/routing.py#L276

```
274             await response(scope, receive, send)
275         else:
276             await self.app(scope, receive, send)
```

This triggers the app closure defined within the request_response method on line 63 of starlette/routing.py. This app function passes the request object to the handler function on line 66.
https://github.com/encode/starlette/blob/master/starlette/routing.py#L66

```
63      async def app(scope: Scope, receive: Receive, send: Send) -> None:
64          request = Request(scope, receive=receive, send=send)
65          if is_coroutine:
66              response = await func(request)
```

This triggers the app closure defined within the get_request_handler method on line 168 of fastapi/routing.py. The app function does some additional parsing and then passes the request data to run_endpoint_function() on line 237
https://github.com/tiangolo/fastapi/blob/master/fastapi/routing.py#L237

```
236         else:
237             raw_response = await run_endpoint_function(
238                 dependant=dependant, values=values, is_coroutine=is_coroutine
239             )
240
```

In run_endpoint_function(), the request data is passed to the application through dependent.call() on line 163
https://github.com/tiangolo/fastapi/blob/master/fastapi/routing.py#L163

```
162         if is_coroutine:
163             return await dependant.call(**values)
```

At this point, the GET request for /user/Matt reaches the application code on line 77 of main.py
https://github.com/mattrrubino/cse312-group-project/blob/main/api/src/main.py#L77

```
76  @app.get("/user/{username}")
77  async def user(response: Response, username: str) -> dict:
78      # Cannot look up invalid username
79      if not validUsername(username):
80          response.status_code = 400
81          return "Bad Request"
```

The user() function from our code then returns a dictionary of user data on line 92
https://github.com/mattrrubino/cse312-group-project/blob/main/api/src/main.py#L92

```
90      del user["Password"]
91
92      return user
```

The raw response that is returned is then serialized in the call to serialize_response() on line 255 of fastapi/routing.py
https://github.com/tiangolo/fastapi/blob/master/fastapi/routing.py#L255

```
253             if sub_response.status_code:
254                 response_args["status_code"] = sub_response.status_code
255             content = await serialize_response(
256                 field=response_field,
257                 response_content=raw_response,
258                 include=response_model_include,
```

In serialize_response(), the dictionary is passed into jsonable_encoder() on line 142 and then returned
https://github.com/tiangolo/fastapi/blob/master/fastapi/routing.py#L142

```python
140         if errors:
141             raise ValidationError(errors, field.type_)
142         return jsonable_encoder(
143             value,
144             include=include,
145             exclude=exclude,
146             by_alias=by_alias,
147             exclude_unset=exclude_unset,
```

In the jsonable_encoder() function, the type of the response value is inspected, and the appropriate code to pack it into a dictionary object is executed. In this case, the response value is a dictionary, so the conditional on line 92 of fastapi/encoders.py returns true. Inside this function, each key and value is passed into the jsonable_encoder() recursively on lines 109 and 117 respectively. Once the original call returns, the value will be a JSON-serializable Python dictionary
https://github.com/tiangolo/fastapi/blob/master/fastapi/encoders.py#L109
https://github.com/tiangolo/fastapi/blob/master/fastapi/encoders.py#L117

```python
109                 encoded_key = jsonable_encoder(
110                     key,
111                     by_alias=by_alias,
112                     exclude_unset=exclude_unset,
```

```python
117                 encoded_value = jsonable_encoder(
118                     value,
119                     by_alias=by_alias,
120                     exclude_unset=exclude_unset,
```

This returned dictionary is then passed to the identifier actual_response_class on line 266, which points to the constructor for JSONResponse in starlette/responses.py on line 196
https://github.com/tiangolo/fastapi/blob/master/fastapi/routing.py#L266
https://github.com/encode/starlette/blob/master/starlette/responses.py#L196

```python
266             response = actual_response_class(content, **response_args)
267             if not is_body_allowed_for_status_code(response.status_code):
268                 response.body = b""
269             response.headers.raw.extend(sub_response.headers.raw)
```

```python
195     ) -> None:
196         super().__init__(content, status_code, headers, media_type, background)
```

The returned JSONResponse object is returned back up through the middleware to the app closure defined in request_response() mentioned previously. The response is then awaited on line 69 of starlette/routing.py, triggering the __call__ method of the Response class
https://github.com/encode/starlette/blob/master/starlette/routing.py#L69

```
67            else:
68                response = await run_in_threadpool(func, request)
69            await response(scope, receive, send)
70
71        return app
```

In the __call__ method of Response, two ASGI messages are sent. The first on line 164 signifies the start of an HTTP response with the status code and headers,
https://github.com/encode/starlette/blob/master/starlette/responses.py#L164

```
163    async def __call__(self, scope: Scope, receive: Receive, send: Send) -> None:
164        await send(
165            {
166                "type": "http.response.start",
167                "status": self.status_code,
168                "headers": self.raw_headers,
169            }
170        )
```

This message is passed up the middleware to the send() method defined in uvicorn/protocols/http/h11_impl.py on RequestResponseCycle. Inside this method, the response data is passed into an h11 Response object on line 509
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L509

```
508            reason = STATUS_PHRASES[status_code]
509            event = h11.Response(
510                status_code=status_code, headers=headers, reason=reason
511            )
```

This Response object is then passed to the send method on the connection object on line 512. This method from h11._connection.py converts the HTTP event into bytes that can be sent on the TCP connection
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L512

```
512            output = self.conn.send(event)
513            self.transport.write(output)
```

The output bytes are then passed to the write method on the transport object on line 513, which will send the raw bytes over the TCP socket
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L513

```
512            output = self.conn.send(event)
513            self.transport.write(output)
```

The transport object is an instance of _ProactorBaseWritePipeTransport. The write method of this class is defined on line 338 of asyncio/proactor_events.py. Importantly, on line 366, it invokes the method _loop_writing, passing a copy of the raw bytes
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L366

```
365                 # Pass a copy, except if it's already immutable.
366                 self._loop_writing(data=bytes(data))
367             elif not self._buffer:  # WRITING -> BACKED UP
```

The first time the _loop_writing method runs, the else block will run. Importantly, this initializes the _write_fut identifier to be the future returned by calling send on the proactor object on line 402
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L402

```
401             else:
402                 self._write_fut = self._loop._proactor.send(self._sock, data)
403                 if not self._write_fut.done():
404                     assert self._pending_write == 0
405                     self._pending_write = len(data)
406                     self._write_fut.add_done_callback(self._loop_writing)
407                     self._maybe_pause_protocol()
```

The proactor object is again an instance of IocpProactor defined in asyncio/windows_events.py. Inside the proactor send method, the function WSASend is invoked on line 537. This invokes the Windows API, writing the bytes in the buffer on the socket via a Python C extension module
https://github.com/python/cpython/blob/main/Lib/asyncio/windows_events.py#L537

```
535         ov = _overlapped.Overlapped(NULL)
536         if isinstance(conn, socket.socket):
537             ov.WSASend(conn.fileno(), buf, flags)
538         else:
```

When the send operation completes, the transport write operation returns. Since the response is not complete (the body is yet to be sent), the conditional on line 546 evaluates to false, and the outer send method returns as well

Back in starlette/responses.py on line 171, the send function is invoked a second time with the HTTP response body
https://github.com/encode/starlette/blob/master/starlette/responses.py#L171

```
169                 }
170             )
171         await send({"type": "http.response.body", "body": self.body})
172
```

This message is again passed up the middleware to the send() method defined in uvicorn/protocols/http/h11_impl.py. However, because the request has started, the conditional on line 481 evaluates to false this time, so the branch on line 515 is taken. This branch passes the bytes into an h11.Data object on line 529, which is used for abstracting

HTTP response bodies
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L529

```
528             else:
529                 event = h11.Data(data=body)
530             output = self.conn.send(event)
531             self.transport.write(output)
```

This Data object is then passed to the send method on the connection object on line 530. This method from h11._connection.py converts the HTTP event into bytes that can be sent on the TCP connection
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L530

```
528             else:
529                 event = h11.Data(data=body)
530             output = self.conn.send(event)
531             self.transport.write(output)
```

The output bytes are then passed to the write method on the transport object on line 531, which will send the raw bytes over the TCP socket
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L531

```
528             else:
529                 event = h11.Data(data=body)
530             output = self.conn.send(event)
531             self.transport.write(output)
```

The transport object is an instance of _ProactorBaseWritePipeTransport. The write method of this class is defined on line 338 of asyncio/proactor_events.py. This time, however, the transport is in the writing state. Thus, the conditional on line 363 evaluates to false, and it instead takes the branch on line 367. Critically, this updates the value of _buffer to be a mutable byte buffer containing the HTTP body bytes on line 369
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L369

```
367         elif not self._buffer:  # WRITING -> BACKED UP
368             # Make a mutable copy which we can extend.
369             self._buffer = bytearray(data)
370             self._maybe_pause_protocol()
```

When the first future completes, it reruns the _loop_writing method in asyncio.proactor_events.py. This time, data is none, so line 388 assigns data to be the bytes stored in _buffer in the previous step
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L388

```
387             if data is None:
388                 data = self._buffer
389                 self._buffer = None
```

Then, on line 402, this data is sent using the proactor send method
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L402

```
401             else:
402                 self._write_fut = self._loop._proactor.send(self._sock, data)
403                 if not self._write_fut.done():
404                     assert self._pending_write == 0
405                     self._pending_write = len(data)
406                     self._write_fut.add_done_callback(self._loop_writing)
407                     self._maybe_pause_protocol()
```

This method sends the body data using the Windows API. Specifically, it writes the supplied buffer to the socket using the WSAsend method
https://github.com/python/cpython/blob/main/Lib/asyncio/windows_events.py#L537

```
535         ov = _overlapped.Overlapped(NULL)
536         if isinstance(conn, socket.socket):
537             ov.WSASend(conn.fileno(), buf, flags)
538         else:
```

Note that the _write_fut again calls _loop_writing when this write completes. However, after writing the body data, data will be none and _buffer will be none. Thus, the conditional on line 390 will evaluate to true, and _call_connection_lost() will be scheduled in the event loop on line 392
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L392

```
390             if not data:
391                 if self._closing:
392                     self._loop.call_soon(self._call_connection_lost, None)
393                 if self._eof_written:
394                     self._sock.shutdown(socket.SHUT_WR)
```

In _call_connection_lost(), asyncio first notifies uvicorn that the TCP connection is being terminated by invoking the connection_lost() method of the protocol on line 158
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L158

```
157         try:
158             self._protocol.connection_lost(exc)
159         finally:
160             # XXX If there is a pending overlapped read on the other
```

Finally, on line 165, the socket is shut down. This is followed by the socket being closed on line 166, which terminates the TCP connection
https://github.com/python/cpython/blob/main/Lib/asyncio/proactor_events.py#L166

```
164             if hasattr(self._sock, 'shutdown') and self._sock.fileno() != -1:
165                 self._sock.shutdown(socket.SHUT_RDWR)
166             self._sock.close()
167             self._sock = None
```

At this point, all of the bytes for the HTTP status line, headers, and body for the response have been sent over the TCP socket to the client. Additionally, the TCP connection has been closed gracefully. Thus, the client has exactly what it needs to display the web content

These technologies are useful because they abstract many complexities of building a web application. Specifically, they completely abstract the process of managing an event loop as well as interfacing with the operating system to create a TCP listening socket, accept connections, read data, and send data. They also abstract the process of parsing bytes and managing middleware. All the application programmer needs to do is follow the framework, and they can build functional applications in very little time (hence the name FastAPI 😉)