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

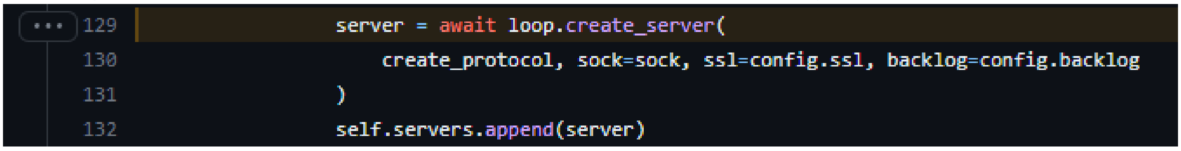
Code Repository	https://github.com/tiangolo/fastapi
License Type	MIT License
License Description	● Permissive license ● Allows unrestricted use without any limitation free of charge ● Protects the copyright holders from liability for any consequences caused by anything utilizing the licensed code
License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

Magic

As is described in the TCP Connection report, FastAPI is a web API framework which is built on top of Starlette, an ASGI framework. These libraries rely on an external ASGI server to handle network connections. We chose to use Uvicorn for this role, which itself utilizes the asyncio and h11 libraries.

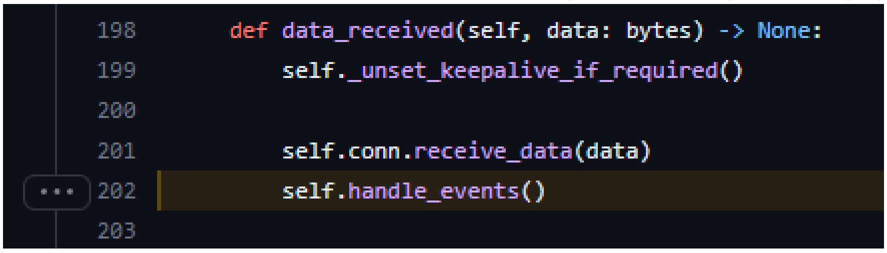
When a client's browser makes an HTTP request to our server, the browser adds a number of headers which are sent with the request. This includes information such as the requested host, the user's browser, and the user's cookies. These requests are received through a socket which Uvicorn opens through asyncio. Asyncio then handles this request, returning the raw bytes to Uvicorn. Uvicorn then passes these raw bytes to h11, which parses these bytes into an HTTP request object (this includes the parsing of the headers). Uvicorn then transforms this request object into an ASGI message to be sent to FastAPI and Starlette. The parsing of h11 is the primary focus of this report and is where the HTTP headers are parsed, though FastAPI and Starlette's handling of the request and its headers is also covered.

On the side of the ASGI server, Uvicorn uses the H11Protocol class to handle TCP connections. The H11Protocol class is passed to asyncio's AbstractEventLoop.create_server() function through the create_protocol variable on line 129 of server.py. For more details on how to get to this function call from the creation of the Uvicorn ASGI server, see the TCP Connection report
https://github.com/encode/uvicorn/blob/master/uvicorn/server.py#L129



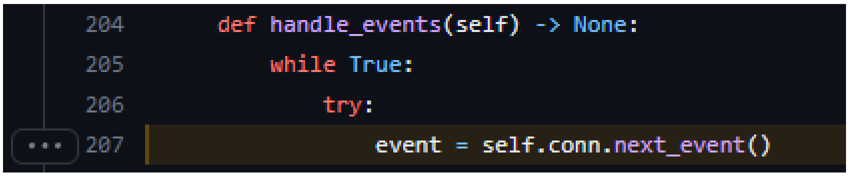
```
129             server = await loop.create_server(
130                 create_protocol, sock=sock, ssl=config.ssl, backlog=config.backlog
131             )
132             self.servers.append(server)
```

When new raw data is recieved by asyncio, it will pass the bytes to Uvicorn by invoking the data_received method of the H11Protocol class. H11Request.handle_events() is then called on line 202 of h11_impl.py to handle this new data
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L202



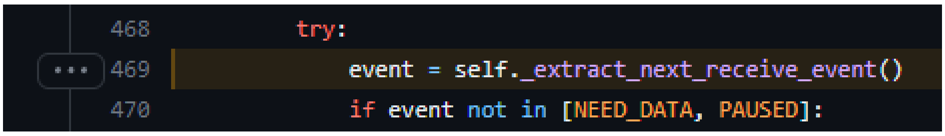
```
198     def data_received(self, data: bytes) -> None:
199         self._unset_keepalive_if_required()
200
201         self.conn.receive_data(data)
202         self.handle_events()
203
```

H11's Connection.next_event() function is called at the beginning of handle_events() to parse the request on line 207 of h11_impl.py
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L207



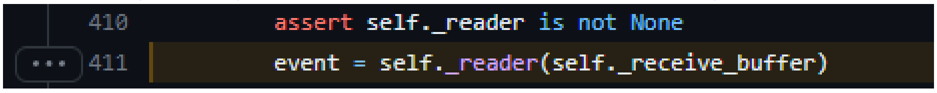
```
204     def handle_events(self) -> None:
205         while True:
206             try:
207                 event = self.conn.next_event()
```

next_event() then proceeds to call _extract_next_receive_event() on line 489 of _connection.py
https://github.com/python-hyper/h11/blob/master/h11/_connection.py#L469

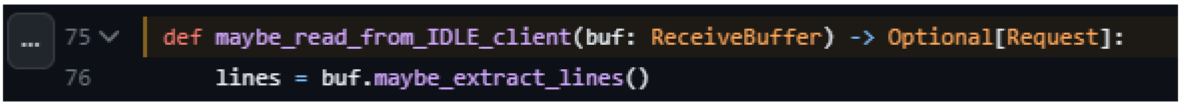


```
468         try:
469             event = self._extract_next_receive_event()
470             if event not in [NEED_DATA, PAUSED]:
```

In _extract_next_receive_event(), the raw data from the TCP byte buffer is passed to self._reader on line 411 of _connections.py. For details on how this raw data was received, see the TCP Connection report. _reader then passes the data to the function maybe_read_from_IDLE_client() which is defined on line 75 of _readers.py
https://github.com/python-hyper/h11/blob/master/h11/_connection.py#L411
https://github.com/python-hyper/h11/blob/master/h11/_readers.py#L75

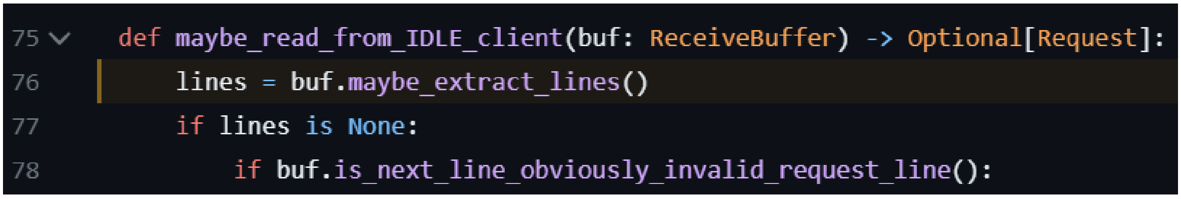


```
410         assert self._reader is not None
411         event = self._reader(self._receive_buffer)
```



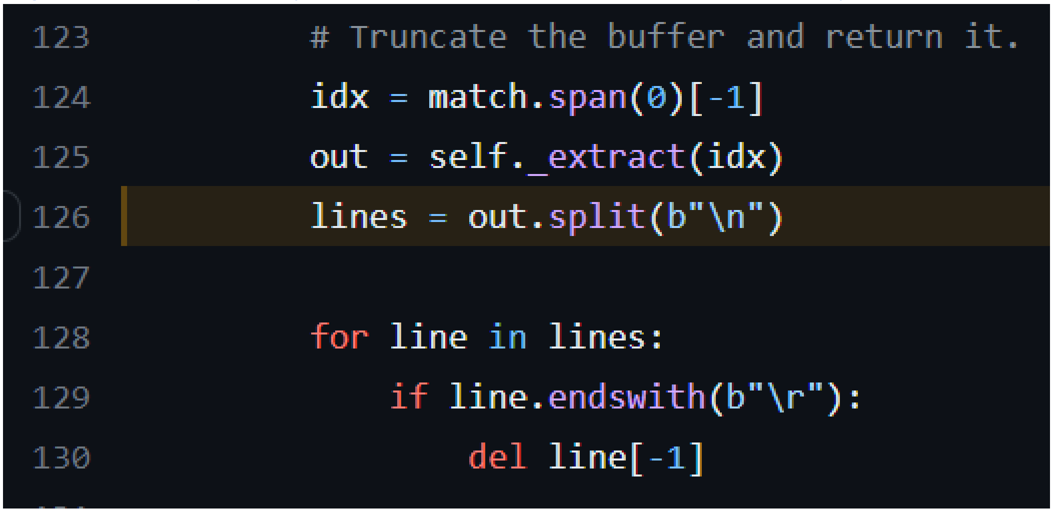
```
75  def maybe_read_from_IDLE_client(buf: ReceiveBuffer) -> Optional[Request]:
76      lines = buf.maybe_extract_lines()
```

maybe_read_from_IDLE_client() then calls maybe_extract_lines() on line 76
https://github.com/python-hyper/h11/blob/master/h11/_readers.py#L76



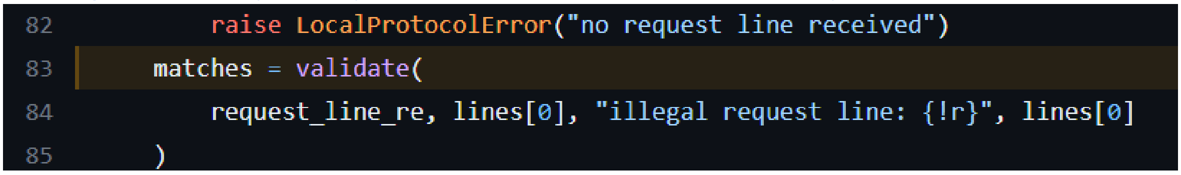
```
75  def maybe_read_from_IDLE_client(buf: ReceiveBuffer) -> Optional[Request]:
76      lines = buf.maybe_extract_lines()
77      if lines is None:
78          if buf.is_next_line_obviously_invalid_request_line():
```

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



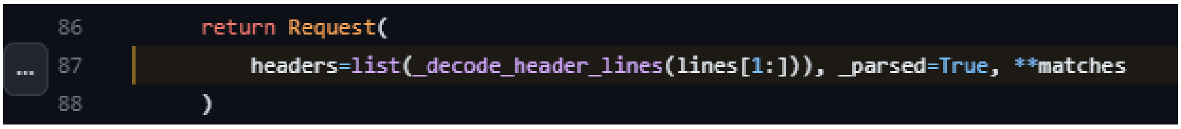
```
123         # Truncate the buffer and return it.
124         idx = match.span(0)[-1]
125         out = self._extract(idx)
126         lines = out.split(b"\n")
127
128         for line in lines:
129             if line.endswith(b"\r"):
130                 del line[-1]
```

The HTTP request line is then validated on line 411 of h11/_readers.py
https://github.com/python-hyper/h11/blob/master/h11/_readers.py#L83



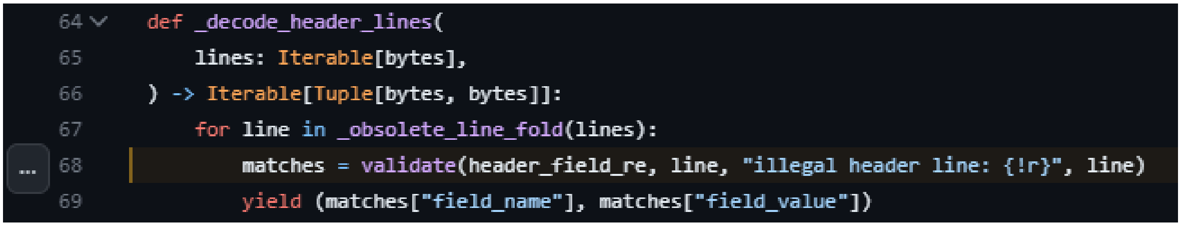
```
82          raise LocalProtocolError("no request line received")
83      matches = validate(
84          request_line_re, lines[0], "illegal request line: {!r}", lines[0]
85      )
```

All of the lines except the first are then passed to the_decode_header_lines() function on line 87 of _readers.py, and the request object is returned after the parsing completes
https://github.com/python-hyper/h11/blob/master/h11/_readers.py#L87



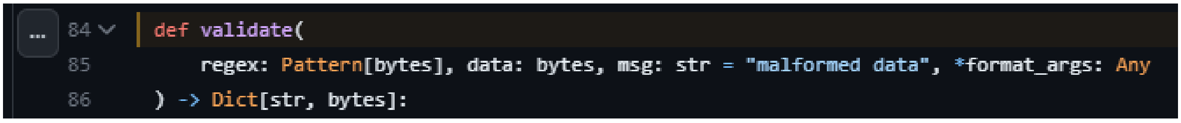
```
86         return Request(
87             headers=list(_decode_header_lines(lines[1:])), _parsed=True, **matches
88         )
```

_decode_header_lines() then runs a for loop to iterate through these lines, calling validate() on line 68 of _readers.py to parse the raw header data into key-value pairs. Each key-value pair is then returned through yield
https://github.com/python-hyper/h11/blob/master/h11/_readers.py#L68

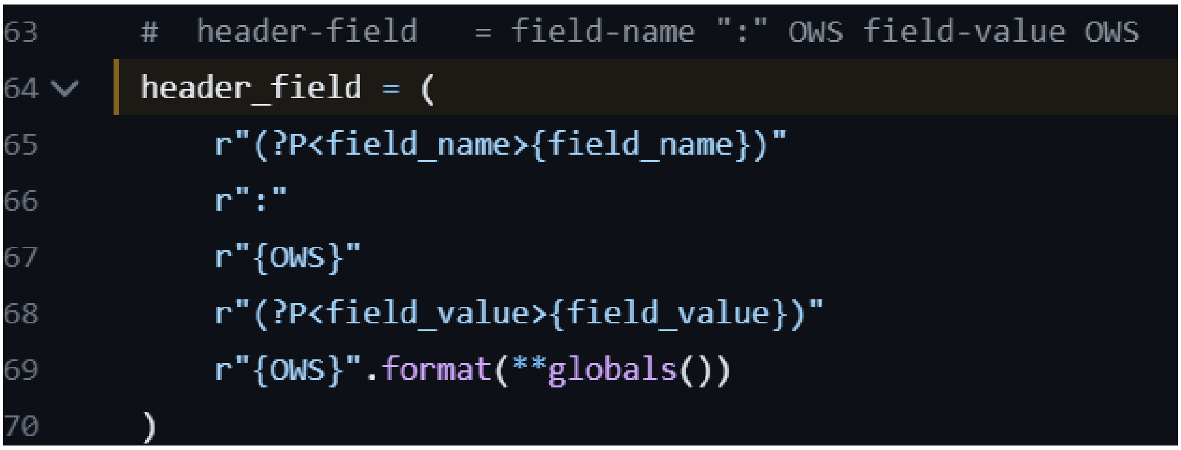


```
64  def _decode_header_lines(
65      lines: Iterable[bytes],
66  ) -> Iterable[Tuple[bytes, bytes]]:
67      for line in _obsolete_line_fold(lines):
68          matches = validate(header_field_re, line, "illegal header line: {!r}", line)
69          yield (matches["field_name"], matches["field_value"])
```

Finally, validate() is defined on line 84 of _util.py. It uses fullmatch() to perform a regex check on the raw header lines to isolate the header name and values. Specifically, it uses the regex defined on line 64 of _abnf.py to match the header key and header value. It then uses groupdict() to convert these matches into a dictionary to be returned. This is the equivalent to the header parsing performed in our homework code, where we used split instead of regex
https://github.com/python-hyper/h11/blob/master/h11/_util.py#L84
https://github.com/python-hyper/h11/blob/master/h11/_abnf.py#L64



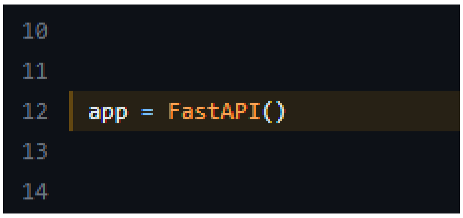
```
84  def validate(
85      regex: Pattern[bytes], data: bytes, msg: str = "malformed data", *format_args: Any
86  ) -> Dict[str, bytes]:
```



```
63  #   header-field    = field-name ":" OWS field-value OWS
64  header_field = (
65      r"(?P<field_name>{field_name})"
66      r":"
67      r"{OWS}"
68      r"(?P<field_value>{field_value})"
69      r"{OWS}".format(**globals())
70  )
```

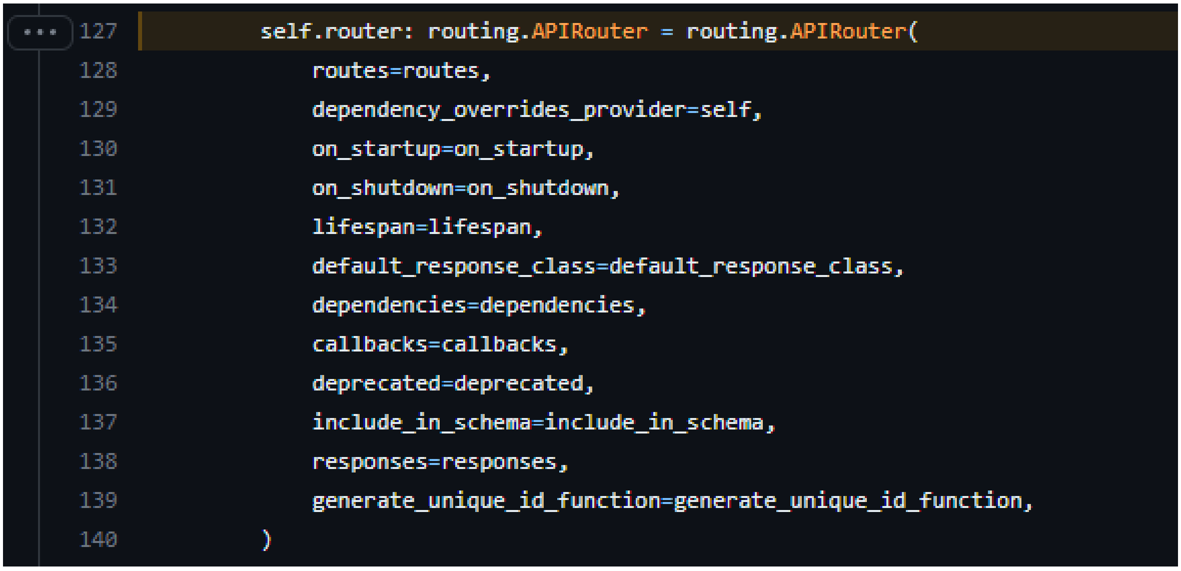
The Request object is then transferred to the FastAPI application via an ASGI message to be handled by custom application code. Details of this process are described in the TCP Connection report. Now, we will describe how FastAPI pulls the headers out of the ASGI message sent from Uvicorn

In our project, we create a FastAPI object on line 12 of main.py. This FastAPI object is used to define routes and handles communication between the Uvicorn ASGI server and our application.
https://github.com/mattrrubino/cse312-group-project/blob/main/api/src/main.py#L12



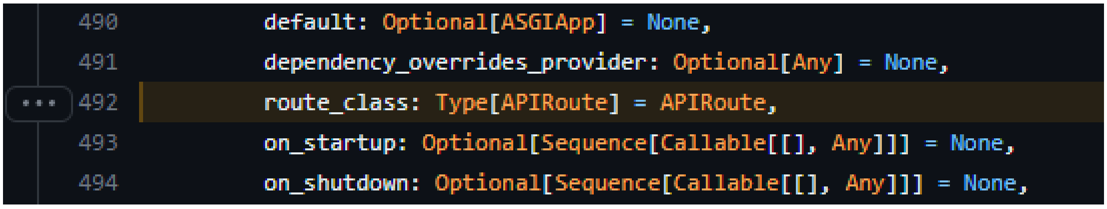
```
10
11
12  app = FastAPI()
13
14
```

The FastAPI class's router is defined in the FastAPI constructor as an APIRouter object on line 127 of applications.py
https://github.com/tiangolo/fastapi/blob/master/fastapi/applications.py#L127



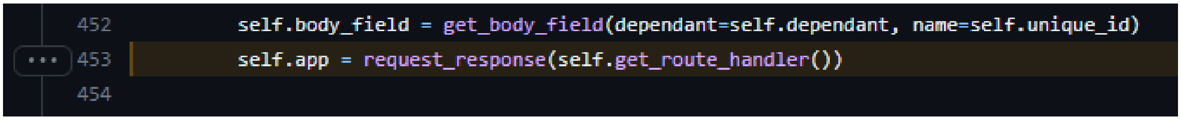
```
127         self.router: routing.APIRouter = routing.APIRouter(
128             routes=routes,
129             dependency_overrides_provider=self,
130             on_startup=on_startup,
131             on_shutdown=on_shutdown,
132             lifespan=lifespan,
133             default_response_class=default_response_class,
134             dependencies=dependencies,
135             callbacks=callbacks,
136             deprecated=deprecated,
137             include_in_schema=include_in_schema,
138             responses=responses,
139             generate_unique_id_function=generate_unique_id_function,
140         )
```

Then, the APIRouter's route_class variable is set to the APIRoute class type on line 492 of routing.py. This variable is then used when creating new route variables via decorators, instantiating them as APIRoute objects
https://github.com/tiangolo/fastapi/blob/master/fastapi/routing.py#L492



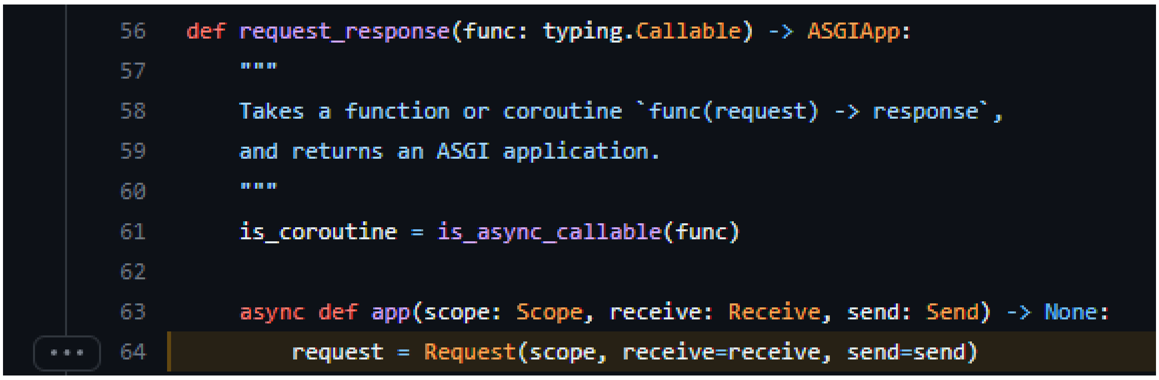
```
490         default: Optional[ASGIApp] = None,
491         dependency_overrides_provider: Optional[Any] = None,
492         route_class: Type[APIRoute] = APIRoute,
493         on_startup: Optional[Sequence[Callable[[], Any]]] = None,
494         on_shutdown: Optional[Sequence[Callable[[], Any]]] = None,
```

In the constructor for the APIRoute class, Starlette's request_response() function is called on line 453 of routing.py. This function wraps a route handler with middleware for handling the HTTP request-response flow
https://github.com/tiangolo/fastapi/blob/master/fastapi/routing.py#L453



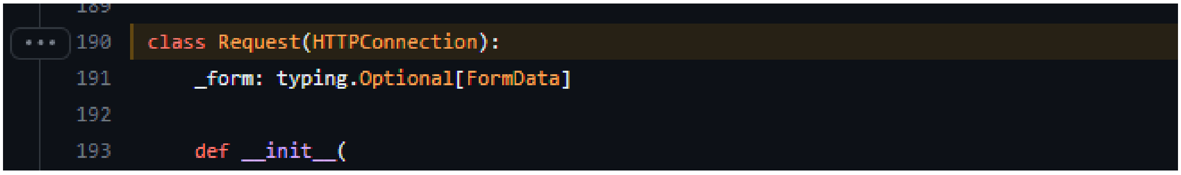
```
452         self.body_field = get_body_field(dependant=self.dependant, name=self.unique_id)
453         self.app = request_response(self.get_route_handler())
454
```

When this middleware is invoked, request_response() creates a Request object on line 64 of Starlette's routing.py, passing the scope from the ASGI message
https://github.com/encode/starlette/blob/master/starlette/routing.py#L64



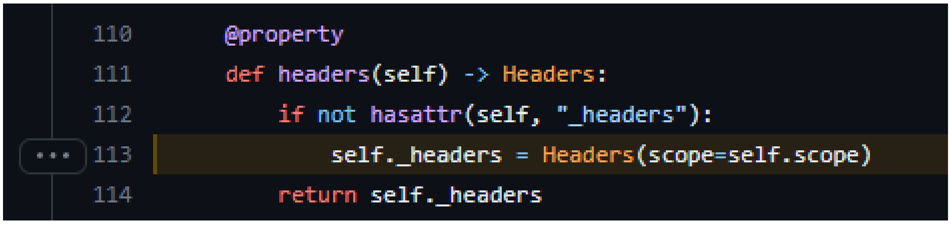
```
56  def request_response(func: typing.Callable) -> ASGIApp:
57      """
58      Takes a function or coroutine `func(request) -> response`,
59      and returns an ASGI application.
60      """
61      is_coroutine = is_async_callable(func)
62
63      async def app(scope: Scope, receive: Receive, send: Send) -> None:
64          request = Request(scope, receive=receive, send=send)
```

The Request class is a child of the HTTPConnection class, as seen in its definition on line 190 of requests.py
https://github.com/encode/starlette/blob/master/starlette/requests.py#L190



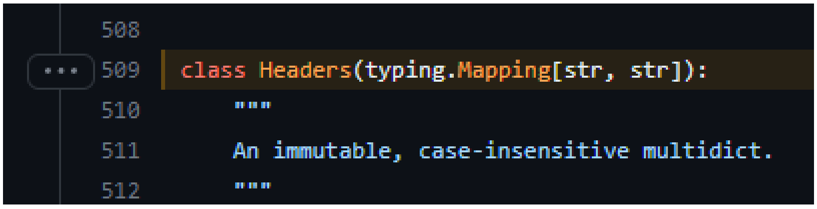
```
189
190 class Request(HTTPConnection):
191     _form: typing.Optional[FormData]
192
193     def __init__(
```

This HTTPConnection class exposes a property called headers. The headers property instantiates a new Headers object on line 111 of request.py if it does not exist, again passing the scope object from the ASGI message
https://github.com/encode/starlette/blob/master/starlette/requests.py#L113

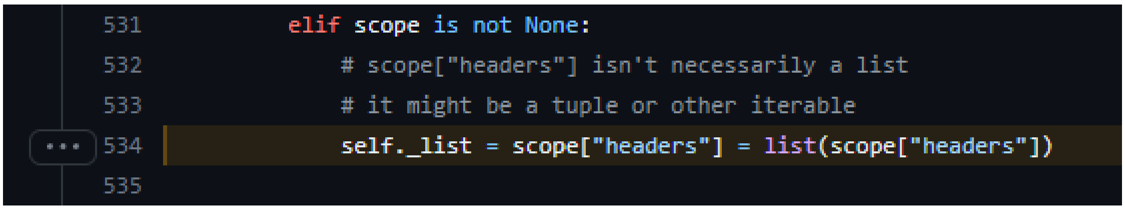


```
110     @property
111     def headers(self) -> Headers:
112         if not hasattr(self, "_headers"):
113             self._headers = Headers(scope=self.scope)
114         return self._headers
```

The Headers class then reads the parsed header info from the ASGI message and stores it in a private variable _list, which contains a key value pair for each header. These headers can then be accessed by the application code through the Headers object. The Header class is defined on line 509 of datastructures.py and the scope["headers"] variable is read on line 534 of datastructures.py.
https://github.com/encode/starlette/blob/master/starlette/datastructures.py#L509
https://github.com/encode/starlette/blob/master/starlette/datastructures.py#L534



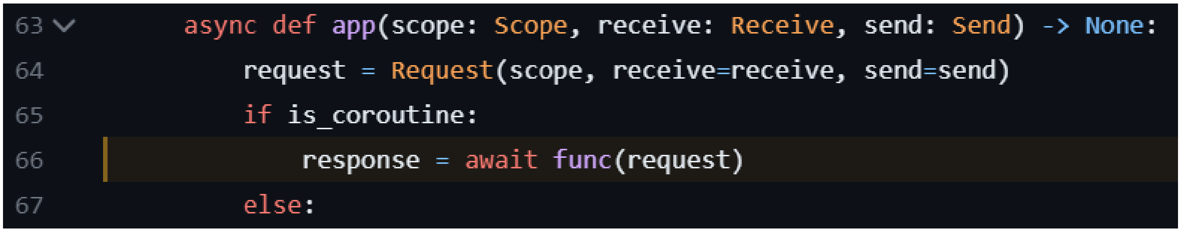
```
508
509 class Headers(typing.Mapping[str, str]):
510     """
511     An immutable, case-insensitive multidict.
512     """
```



```
531         elif scope is not None:
532             # scope["headers"] isn't necessarily a list
533             # it might be a tuple or other iterable
534             self._list = scope["headers"] = list(scope["headers"])
535
```

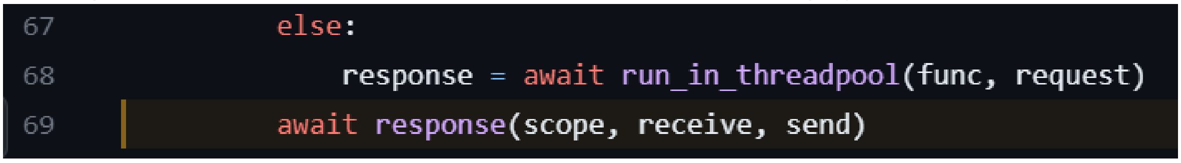
Now, we will describe how the headers in the returned response are converted into bytes.

As mentioned previously, the request_response() function wraps the ASGI app with middleware for handling the HTTP request-response flow. Thus, the response returned by the application (including the headers) is available to this middleware on line 66 of starlette/routing.py (see TCP Connection report for details of how this happens)
https://github.com/encode/starlette/blob/master/starlette/routing.py#L66



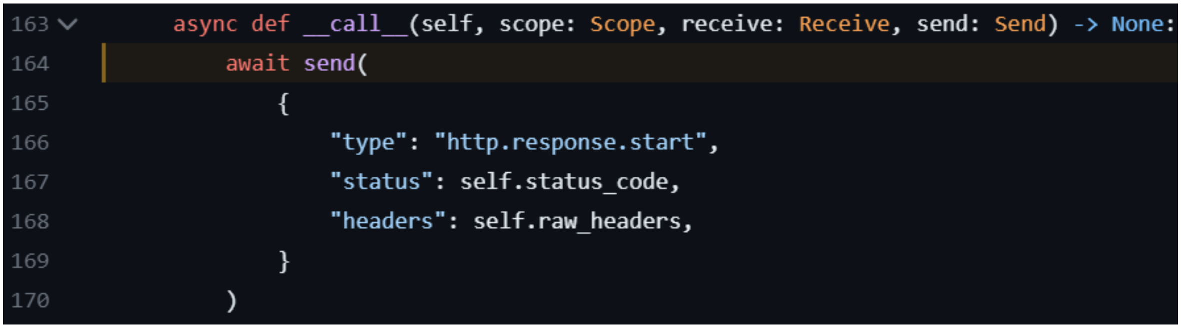
```
63      async def app(scope: Scope, receive: Receive, send: Send) -> None:
64          request = Request(scope, receive=receive, send=send)
65          if is_coroutine:
66              response = await func(request)
67          else:
```

This response object is then invoked as an ASGI app on line 69, triggering the __call__ method of the starlette Response class
https://github.com/encode/starlette/blob/master/starlette/routing.py#L69



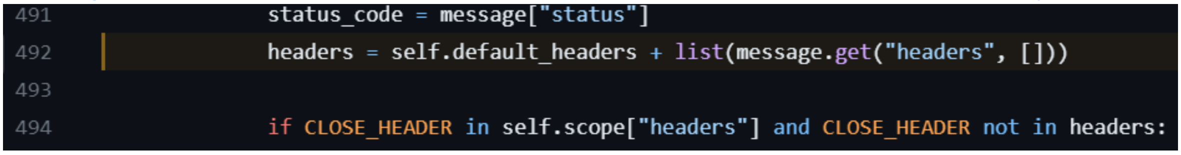
```
67          else:
68              response = await run_in_threadpool(func, request)
69          await response(scope, receive, send)
```

In the __call__ method of the starlette Response class, the raw headers from the response are then passed to the ASGI send method on line 164
https://github.com/encode/starlette/blob/master/starlette/responses.py#L164



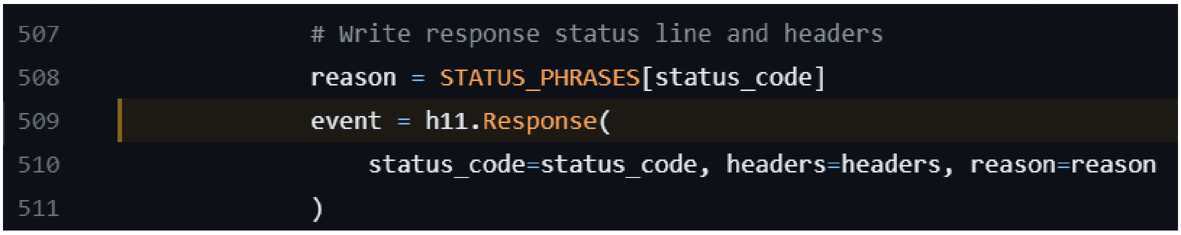
```
163     async def __call__(self, scope: Scope, receive: Receive, send: Send) -> None:
164         await send(
165             {
166                 "type": "http.response.start",
167                 "status": self.status_code,
168                 "headers": self.raw_headers,
169             }
170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

This send message then traverses up the middleware stack until it reaches the send method of RequestResponseCycle in uvicorn/protocols/http/h11_impl.py. At this point, the headers are retrieved from the ASGI message on line 492
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L492



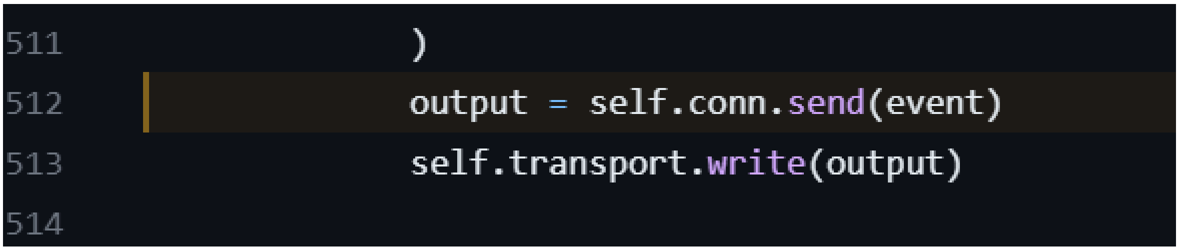
```
491         status_code = message["status"]
492         headers = self.default_headers + list(message.get("headers", []))
493
494         if CLOSE_HEADER in self.scope["headers"] and CLOSE_HEADER not in headers:
```

The headers are then passed to an h11 Response object on line 509
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L509



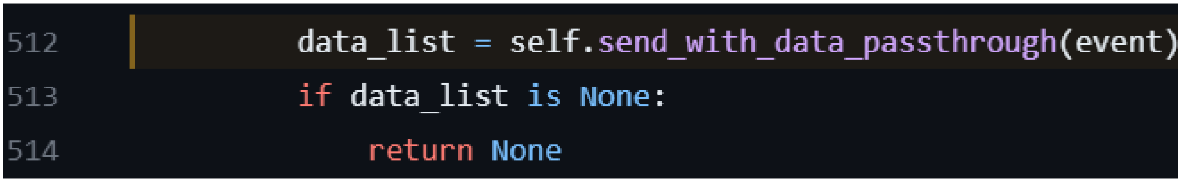
```
507         # Write response status line and headers
508         reason = STATUS_PHRASES[status_code]
509         event = h11.Response(
510             status_code=status_code, headers=headers, reason=reason
511         )
```

This Response object is then converted to bytes on line 512 by invoking the send() method on the h11 Connection object
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L512



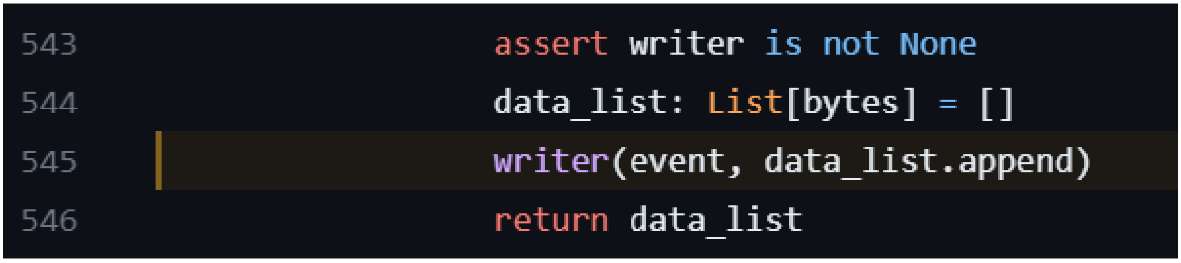
```
511                 )
512                 output = self.conn.send(event)
513                 self.transport.write(output)
514
```

In the send() method of the Connection object, the method send_with_data_passthrough() is invoked on line 512, which take the response event and converts it to a list of bytearrays (where each bytearray corresponds to one line in the HTTP response)
https://github.com/python-hyper/h11/blob/master/h11/_connection.py#L512



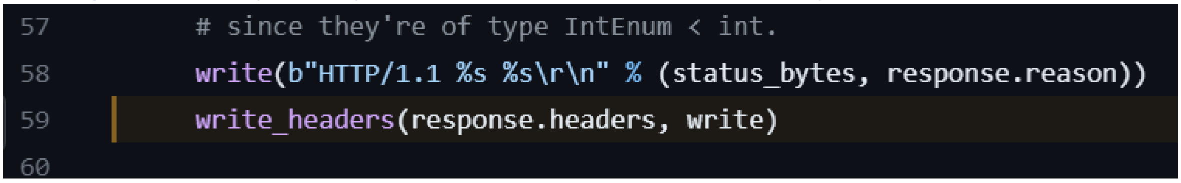
```
512         data_list = self.send_with_data_passthrough(event)
513         if data_list is None:
514             return None
```

In send_with_data_passthrough(), a writer in invoke on line 545, which takes in the response event and appends each line of the response to data_list using the callback function
https://github.com/python-hyper/h11/blob/master/h11/_connection.py#L545



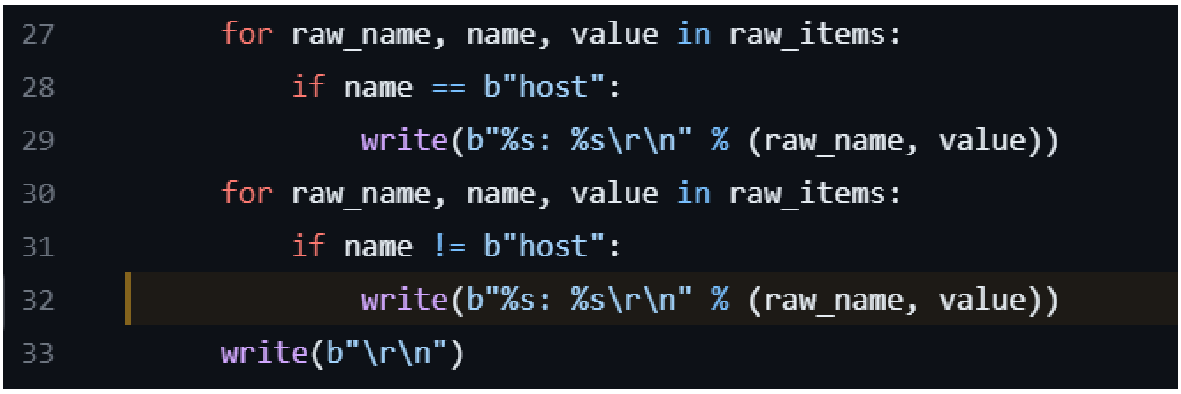
```
543                 assert writer is not None
544                 data_list: List[bytes] = []
545                 writer(event, data_list.append)
546                 return data_list
```

This triggers the write_any_response() function in h11/_writers.py. This appends the response status line on line 58, and invokes write_headers() on line 59 to append the headers
https://github.com/python-hyper/h11/blob/master/h11/_writers.py#L59



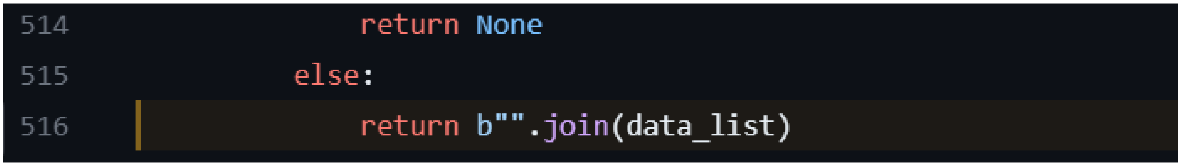
```
57      # since they're of type IntEnum < int.
58      write(b"HTTP/1.1 %s %s\r\n" % (status_bytes, response.reason))
59      write_headers(response.headers, write)
60
```

In write_headers(), the “Host” header is appended to data_list first on line 29, followed by the remaining headers on line 32
https://github.com/python-hyper/h11/blob/master/h11/_writers.py#L32



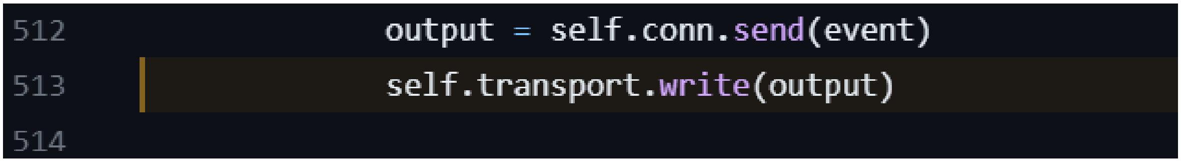
```python
    for raw_name, name, value in raw_items:
        if name == b"host":
            write(b"%s: %s\r\n" % (raw_name, value))
    for raw_name, name, value in raw_items:
        if name != b"host":
            write(b"%s: %s\r\n" % (raw_name, value))
    write(b"\r\n")
```

The variable data_list is then joined into one long bytearray and returned on line 516 in the original send() method
https://github.com/python-hyper/h11/blob/master/h11/_connection.py#L516



```python
514                 return None
515             else:
516                 return b"".join(data_list)
```

These bytes are then sent over the socket by invoking the write() method of the transport object on line 512 of uvicorn/protocols/http/h11_impl.py. Details of this process are described in the TCP Connection report
https://github.com/encode/uvicorn/blob/master/uvicorn/protocols/http/h11_impl.py#L513



```python
512         output = self.conn.send(event)
513         self.transport.write(output)
514
```

This completes the description of HTTP header parsing with FastAPI, including the process of parsing raw bytes into HTTP headers, passing these headers to the application, and converting the returned headers back into bytes to be sent on the TCP connection. This whole process is abstracted from the application, which is presented with a simple header interface (headers are simply a key-value store/dictionary). This makes working with HTTP headers trivial.