[FastAPI]

General Information & Licensing

Code Repository	https://github.com/tiangolo/fastapi
License Type	MIT
License Description	● The license grants the software free of charge ● The license allows the software to be used and distributed in any way, including commercial
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 they notice that the software is free to use

Magic ⋆★˚｡⋆°⋆.˚☽˚⁀➷｡˚★彡⋆ ༄

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

- How does this technology do what it does? Please explain this in detail, starting from after the TCP socket is created

  FastAPI handles HTTP requests by using the Startlette library. After the TCP socket is created, it is sent to the *datastructures.py* file from the Starlette directory (link: https://github.com/encode/starlette/blob/master/starlette/datastructures.py).

  In datastructures.py, there is a class named *Headers()*. The *Headers()* class creates a data structure (immutable, case-insensitive multidict) and initializes the data to None. This class also includes @property methods that get and set the keys and values in an HTTP request. This sets up the data structure for which can be populated and parsed through.

  The multidict created from the *Headers()* class is then sent to *formparsers.py*. This is also found from the Starlette directory (link: https://github.com/encode/starlette/blob/master/starlette/formparsers.py). From *formparsers.py*, the python libraries, *multipart* and *multipart.parse_options_header* are imported. Also in *formparsers.py* is the class, *MultiPartParser.* This class is useful because it includes the method *parse()*. The first lines of *parse()* get the multipart boundary:

```
    async def parse(self) -> FormData:
        # Parse the Content-Type header to get the multipart boundary.
        content_type, params =
parse_options_header(self.headers["Content-Type"])
```

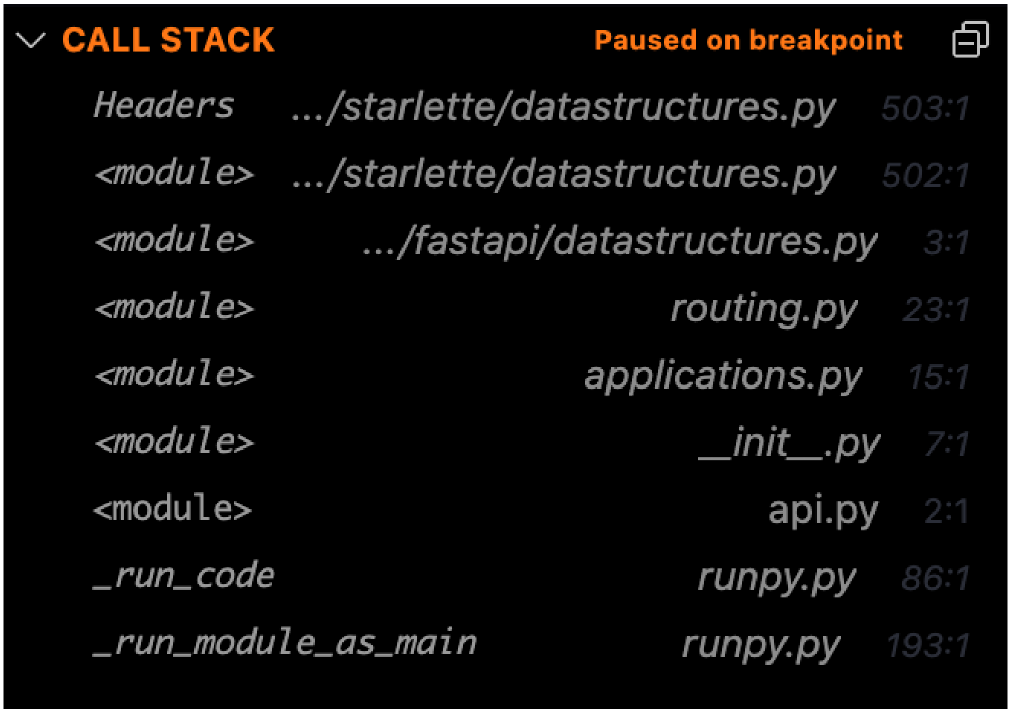
  After parsing the correct boundary, *parser()* creates the parser using:

```
        # Create the parser.
        parser = multipart.MultipartParser(boundary, callbacks)
```

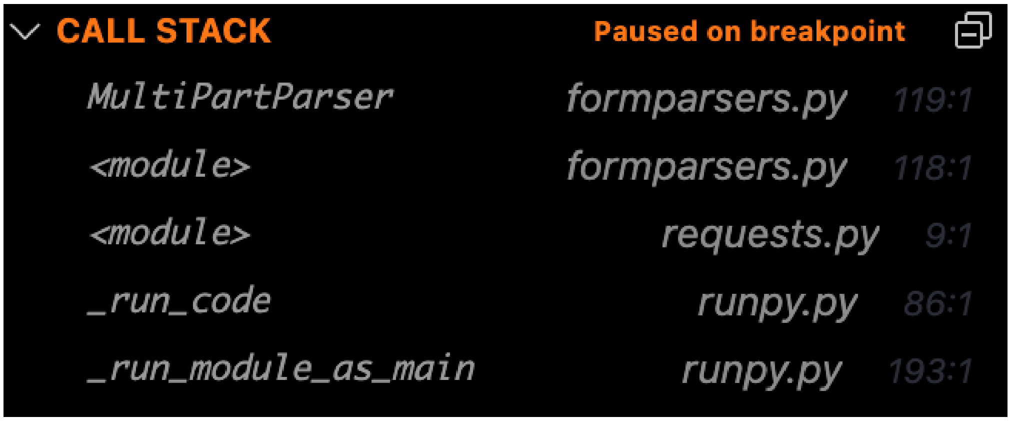
  This parser then goes through all the header fields and returns the data as type *FormData* which was imported from *datastructures.py*. Each header field in the *FormData* is in the format of a key-value pair, as they are received from the get/post request. In this step, the data is read in and parsed through such that the *FormData* includes all the HTTP headers and is then ready to be sent to the *requests.py* in Starlette (link: https://github.com/encode/starlette/blob/master/starlette/requests.py).

  From *requests.py*, the class *HTTPConnection()* is used. *HTTPConnection()* is responsible for handling HTTP connections. In this, it has a @property method named *headers()* which essentially set the headers from requests.py to the new class's headers that have more functionality built-in. *HTTPConnection()* is the last time that FastAPI parses are manipulates HTTP headers. At this point, the headers are then served back to the original @app.get() or @app.post() that the data was received on.

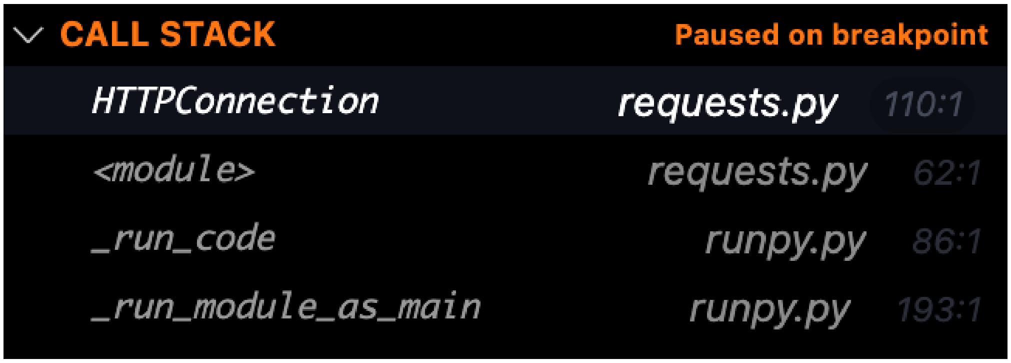
- Where is the specific code that does what you use the tech for? You ***must*** provide a link to the specific file in the repository for your tech with a line number or number range.
    - If there is more than one step in the chain of calls *(hint: there will be)*, you must provide links for the entire chain of calls from your code, to the library code that actually accomplishes the task for you.



```
Headers               .../starlette/datastructures.py   503:1
<module>              .../starlette/datastructures.py   502:1
<module>                 .../fastapi/datastructures.py    3:1
<module>                                    routing.py   23:1
<module>                               applications.py   15:1
<module>                                   __init__.py    7:1
<module>                                        api.py    2:1
_run_code                                     runpy.py   86:1
_run_module_as_main                           runpy.py  193:1
```



```
MultiPartParser                      formparsers.py  119:1
<module>                             formparsers.py  118:1
<module>                                requests.py    9:1
_run_code                                  runpy.py   86:1
_run_module_as_main                        runpy.py  193:1
```



```
CALL STACK                                Paused on breakpoint
    HTTPConnection                        requests.py   110:1
    <module>                              requests.py    62:1
    _run_code                             runpy.py       86:1
    _run_module_as_main                   runpy.py      193:1
```

Pictured above are the stack traces for all code involved in the chain of calls. The first image is for serving the parsing methods the prepared data structure to fill with data and then parse on. (Setting up to be parsed) The second image is for the multipartparser which is where the headers are actually parsed. The third image shows the call stack for where the data is send after fastapi/starlette parses the headers.

The chain of calls starts in *api.py*. (link: https://github.com/thatonenerd2000/312_hooligans/blob/main/backend/api.py)
From *api.py*, line 2, we import FastAPI is imported from the fastapi libaray:

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
```

On line 18 of *api.py,* we initalize a variable named *app* to FastAPI():

```
app = FastAPI()
```

FaskAPI() always be invoked whenever the code does an app.post() or app.get(). These HTTP method calls can be found from lines 30-194. An example of the first get request is for the path “/createUser” on line 30:

```
@app.post("/createUser")
```

From each @app.post() or @app.get() the next call in the chain is to *__init__.py* (link: https://github.com/tiangolo/fastapi/blob/master/fastapi/__init__.py)

The usage of this is from line 7 of *__init__.py:*

```
from .applications import FastAPI as FastAPI
```

This gets the FastAPI reference so it can be used by the following code calls. Next is the library import for *applications* in the *applications.py* file.

Line 15 from *applications.py* imports *routing* from *fastapi*:

```
from fastapi import routing
```

The *routing* import brings the call stack to *routing.py*.(link: https://github.com/tiangolo/fastapi/blob/master/fastapi/routing.py) On line 23 of *routing.py,* this is where the HTTP data structure which will include the headers is included:

```
from fastapi.datastructures import Default, DefaultPlaceholder
```

After the *datastructures* are imported, the code makes its way to *datastructures.py* (link: https://github.com/tiangolo/fastapi/blob/master/fastapi/datastructures.py).

The purpose of *datastructures.py* is to imports *URL*, *Headers,* and *FormData.* These libraries are responsible for handling and parsing incoming HTTP connections, including the headers. These can be found on lines 3-6:

```
from starlette.datastructures import URL as URL  # noqa: F401
from starlette.datastructures import Address as Address  # noqa: F401
from starlette.datastructures import FormData as FormData  # noqa: F401
from starlette.datastructures import Headers as Headers  # noqa: F401
```

Headers (line 6) is the most crucial, low-level, library necessary for parsing the HTTP headers. This then sends the code to Starlette's *datastructures.py (*This was previously from fastapi)

In *starlette/datastructures.py,* (link: https://github.com/encode/starlette/blob/master/starlette/datastructures.py) line 502 is where the class *Headers()* is found:

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

Lines 529-537 of *Headers():*

```
    @property
    def raw(self) -> typing.List[typing.Tuple[bytes, bytes]]:
        return list(self._list)

    def keys(self) -> typing.List[str]:  # type: ignore[override]
        return [key.decode("latin-1") for key, value in self._list]

    def values(self) -> typing.List[str]:  # type: ignore[override]
        return [value.decode("latin-1") for key, value in self._list]
```

These methods inside of *Headers*() initialize the data structure (An immutable, case-insensitive multidict) that is ready to be passed up to the parser methods. This data structure is referred to as a *FormData*. The parsing process starts in Starlette's *requests.py* (link: https://github.com/encode/starlette/blob/master/starlette/requests.py).

On line 9 of *requests.py,* the multpartparser is imported from starlette.formparsers:

```
from starlette.formparsers import FormParser, MultiPartException,
MultiPartParser
```

Also, on line 7, the *FormData* data structure in imported:

```
from starlette.datastructures import URL, Address, FormData, Headers,
QueryParams, State
```

Finally, the chain of calls is sent to Starlette's *formparsers.py* (link: https://github.com/encode/starlette/blob/master/starlette/formparsers.py).

formparsers.py imports *multipart* on line 8:

```
    import multipart
```

This is the base library for *parse_options_header* and *MultipartParser.* The *parse_options_header* allows the code to extract the media type and any other options that are included in the Content-Type header. The *MultipartParser* is the actual parser and includes method *finalize()* to cleanup and ensure formatting is correct after parsing is complete.

In *formparsers.py,* line 118 defines the class *MultiPartParser()*:

```
class MultiPartParser:
```

In *MultiPartParser(),* line 161 is the *parse()* method:

```
    async def parse(self) -> FormData:
```

Line 163 parses the Content-Type header to get the multipart boundary:

```
        content_type, params =
parse_options_header(self.headers["Content-Type"])
```

Line 185 creates the parser:

```
        parser = multipart.MultipartParser(boundary, callbacks)
```

From lines 198-249, *MultiPartParser()* feeds the parser with data from the HTTP request. For each header field that is in the data being parsed, it is added to the *FormData* from *Headers()*. When the parsing is finished, on line 251 *finalize()* is called on the parser:

```
        parser.finalize()
```

Line 252, the last line of *MultiPartParser(),* is the return with the *FormData* populated with parsed data:

```
        return FormData(items)
```

The header data has been completely parsed up to this point and is passed back to *starlette/requests.py* (link: https://github.com/encode/starlette/blob/master/starlette/requests.py).

In this *requests.py* file on line 66, there is a class named *HTTPConnection(),* that adds functionality to the HTTP connections it was just given as a *FormData.*

```
class HTTPConnection(typing.Mapping[str, typing.Any]):
    """
    A base class for incoming HTTP connections, that is used to provide
    any functionality that is common to both `Request` and `WebSocket`.
    """
```

This includes getting a mapping of key-value pairs from the headers and serving it all the way back up to api.py to be easily accessed and interpreted.