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/Uvicorn]

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	• when using the MIT license software must include a notice containing copyright information, as well as users notice that the software is free to use

Magic

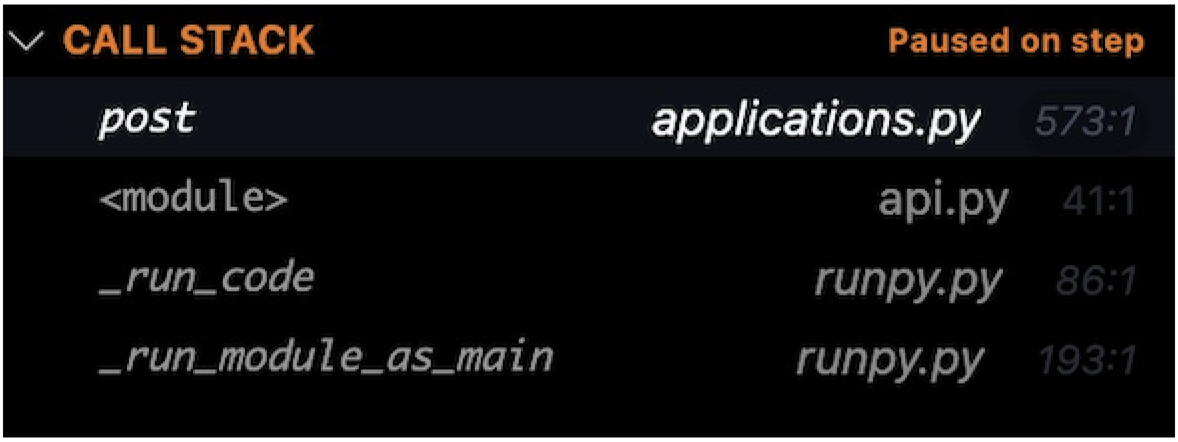
FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints. It is built on top of the Starlette web framework and the Uvicorn ASGI server.

Uvicorn is a lightning-fast ASGI server implementation, using uvloop and httptools. It is able to handle a high number of concurrent connections, making it well-suited for serving APIs that require high performance.

Together, FastAPI and Uvicorn make it easy to build high-performance APIs with Python. FastAPI provides a simple, intuitive interface for defining the routes and handling the request and response data, while Uvicorn handles the low-level details of serving the API over the network.
We can use them together to build and serve an API.To do this, you first need to define the API using FastAPI. This involves defining the routes and the request and response data for each route. Here is how it is being done in our project:
https://github.com/thatonenerd2000/312_hooligans/blob/main/backend/api.py
At line 17 , `app = FastAPI()`
At line 39, `@app.post("/createUser")`
Here we are defining a route, '/createUser, that returns a JSON object containing a simple message. Once the app is defined, you can use Uvicorn to run it and serve the API:

```
uvicorn app:app
```

This command runs the app using Uvicorn, and serves it on the default host and port. You can then access the API using a web browser or other HTTP client, and send requests to the defined routes.



The chain of calls starts at line 2 in api.py where we import fastAPI
https://github.com/thatonenerd2000/312_hooligans/blob/main/backend/api.py

```
rom fastapi import FastAPI, WebSocket, WebSocketDisconnect, Cookie
```

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints. It is built on top of the Starlette web framework and the Uvicorn ASGI server.

Uvicorn is a lightning-fast ASGI server implementation, using uvloop and httptools. It is able to handle a high number of concurrent connections, making it well-suited for serving APIs that require high performance.
When a client makes an HTTP request to a web application running on uvicorn, the server establishes a TCP (Transmission Control Protocol) connection to the client. This is the underlying transport mechanism that is used to transfer the HTTP request and response messages between the client and the server.

Once the TCP connection is established, uvicorn uses the asyncio event loop to

asynchronously handle the request and send the response. This allows the server to handle multiple requests concurrently and efficiently, without blocking on any single request.

Together, FastAPI and Uvicorn make it easy to build high-performance APIs with Python. FastAPI provides a simple, intuitive interface for defining the routes and handling the request and response data, while Uvicorn handles the low-level details of serving the API over the network.
We can use them together to build and serve an API.To do this, you first need to define the API using FastAPI. This involves defining the routes and the request and response data for each route. Here is how it is being done in our project:
https://github.com/thatonenerd2000/312_hooligans/blob/main/backend/api.py
At line 17 , `app = FastAPI()`
At line 39, `@app.post("/createUser")`
Here we are defining a route, '/createUser, that returns a JSON object containing a simple message. Once the app is defined, you can use Uvicorn to run it and serve the API:
From each @app.post() or @app.get() the next call in the chain is to applications.py
At line 573 https://github.com/tiangolo/fastapi/blob/master/fastapi/applications.py
`return self.router.post`

`uvicorn app:app`
This command runs the app using Uvicorn, and serves it on the default host and port. You can then access the API using a web browser or other HTTP client, and send requests to the defined routes.

From lines 21 to 27 we are using CORSMiddleware from the fastapi.middleware.cors library.

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"]
)
```

CORS Middleware is a type of software that is used to manage cross-origin requests in web applications. Cross-origin requests occur when a client, such as a web browser, makes a request to a server that is located at a different domain than the client.

CORS Middleware is used to handle these cross-origin requests by adding specific headers to the response that allow the client to access the resources on the server. These headers specify which domains are allowed to access the resources on the server, and what types of requests are allowed.

In our use case from lines 21 to 27 , the CORSMiddleware is applied to the API using the app.add_middleware decorator. The allow_origins, allow_credentials, allow_methods, and allow_headers parameters are used to define the CORS policies for the API, allowing all origins, methods, and headers. Once the middleware is applied, it will automatically add the appropriate headers to the response for any cross-origin requests made to the API. This allows clients to access the resources on the server, while protecting the server from unauthorized access.
In our project we are also using the WebSocket, WebSocketDisconnect, and Cookie class from the fastAPI library.WebSocket, WebSocketDisconnect, and Cookie are three components of the FastAPI web framework that are used to build real-time, two-way communication applications.
WebSocket is a protocol that allows for bidirectional, full-duplex communication between a

client and a server over a single, long-lived connection. It allows the client and server to send and receive messages asynchronously, allowing for real-time communication and data exchange.
Lines 299 - 309 on api.py

```
@app.websocket("/ws/auction/{auctionID}")
async def websocket_endpoint(websocket: WebSocket, auctionID: str):
    await manager.connect(websocket, auctionID)
    try:
        while True:
            data = await websocket.receive_text()
            data = json.loads(data)
            await manager.broadcast(data, auctionID)
    except WebSocketDisconnect:
        manager.disconnect(websocket)
```

here, the websocket_endpoint function is a FastAPI websocket endpoint that accepts incoming WebSocket connections on the /ws/auction/{auctionID}" path. When a new connection is made, the accept() method is called to accept the connection. WebSocketDisconnect is an exception that is raised when a WebSocket connection is closed, either by the client or the server. It can be used to handle the disconnection gracefully and perform any necessary clean-up or notification actions.To handle disconnections, the code uses a try/except block to catch any WebSocketDisconnect exceptions that may be raised. When a disconnection occurs, the WebSocketDisconnect exception is raised and caught.
Cookie is a class that is used to manage cookies in FastAPI applications. It allows you to set, get, and delete cookies, as well as to specify the cookie options, such as the expiration time and the domain.
Lines 93-94

```
@app.get("/verifyAuth")
async def verifyAuth(authToken: Union[str, None] = Cookie(default=None)):
```

here, the verifyAuth is a FastAPI route that accepts an optional user_id parameter. This parameter is decorated with the Cookie class, which tells FastAPI to automatically parse the "user_id" cookie from the request and bind its value to the user_id parameter.

In summary, FastAPI and Uvicorn work together to provide a simple, efficient way to build and serve high-performance APIs with Python. FastAPI provides an easy-to-use interface for defining the API, while Uvicorn handles the low-level details of serving the API over the network.