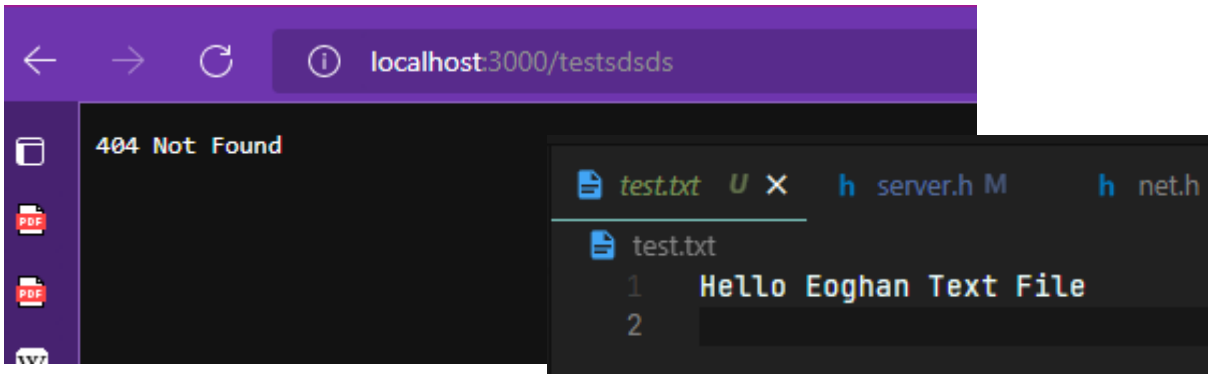
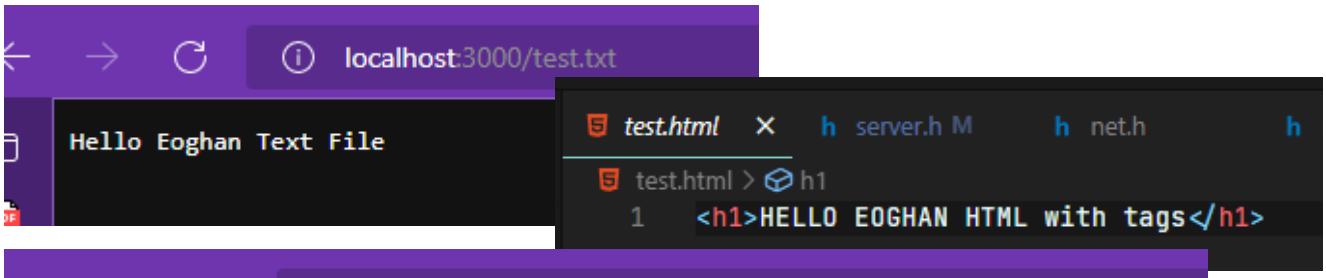
Question 1:

Also Implemented File not found and basic content type checking






```
t$ curl -v  localhost:3000/test.html
*   Trying 127.0.0.1:3000...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 3000 (#0)
> GET /test.html HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.68.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Length: 38
< Content-Type: text/html; charset=utf-8
<
<h1>HELLO EOGHAN HTML with tags</h1>

* Closing connection 0
```

```
$ curl -v  localhost:3000/test.txt
```

```
*   Trying 127.0.0.1:3000...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 3000 (#0)
> GET /test.txt HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.68.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Length: 25
< Content-Type: text/text; charset=utf-8
<
Hello Eoghan Text File


* Closing connection 0
```

Question 2:

Libuv is the system upon which internal calls are used in Node.js

It uses system calls and non blocking operations. This is different from a blocking operation where the program or OS has to check each socket individually for new information or packets or to see if the flag was changed. As more sockets are added to the queue the longer polling takes away from other system operations. This is normally solved by using a separate thread to monitor the socket which can quickly slow the entire system down as each thread adds to the load.

Particularly in xv6, when a read socket operation is happening a lock is put in place and if the required data isn't there it will sleep until the requested amount of bytes is found.

This is very slow compared to using a non blocking operation that is used in libuv. Libuv operates using an event loop, that listens for incoming data on the sockets its monitoring, if a socket has data a low level system interrupt is called to deal with this data at the relevant point.

The key thing with libuv is that it's event loop is consistently running, it does not wait, it has multiple queues within it containing flags regarding if write or read operations are scheduled to happen instead so that instead of happening immediately and locking the system until the operation is complete it can be flagged and will then be dealt with at the most appreciated time.

Whenever a system call is called, libuv is very easily interrupted as it is not blocking the operating system so there is little time wasted.

Question 3:

```
<-------->Packet Start<-------->
        <----> Mac <---->
        MAC Source:     82:85:10:0:2:2:
        MAC Destination:        82:84:0:18:52:86:
        MAC Destination:         8
        Ethernet Packet Type:   IP

        <----> IP <---->
        IP Source:      10:0:2:2:
        IP Destination: 10:0:2:15:
        IP Type: TCP

        <----> TCP <---->
<-------->Packet End<-------->
```

```
<-------->Packet Start<-------->
        <----> Mac <---->
        MAC Source:     82:85:10:0:2:2:
        MAC Destination:        255:255:255:255:255:255:
        MAC Destination:         1544
        Ethernet Packet Type:   ARP
        <----> ARP <---->
<-------->Packet End<-------->
```

```
<-------->Packet Start<-------->
        <----> Mac <---->
        MAC Source:     82:85:10:0:2:2:
        MAC Destination:        82:84:0:18:52:86:
        MAC Destination:         8
        Ethernet Packet Type:   IP

        <----> IP <---->
        IP Source:      8:8:8:8:
        IP Destination: 10:0:2:15:
        IP Type: UDP

        <----> UDP <---->
<-------->Packet End<-------->
```

Question 4:

When sockalloc() is called, a new sock{} struct is created. This sock structs sock->next is then set to the previous last socket on the queue. The list of sockets operates in a similar way to a linked list, where each new socket has a link to the “next” socket in the queue.

The sock{} data struct also contains a reference to an mbufq struct that contains two elements consisting of the head and tail of the elements within its queue comprising of mbuf structs. The mbuf struct contains the length of the buffer and what’s stored in the buffer.

The file struct can be created containing a file type designating that it’s a socket, in which case when the write() system call is called, when it is called on this file, it traverses from sys_write to filewrite which upon finding that the file is of type socket will instead call the dedicated sockwrite() function in sysnet.c.

When sockwrite() is called, the socket is locked, and a new mbuf is created to the mbufq struct of the socket using the net_tx_udp() call that will wrap the data with a UDP header and will start to create a packet. It will then call net_tx_ip, that will add the ip layer to the packet, which will then call e1000_transmit that will set up the mac layer and transmit the finalized packet to its destination socket.

Similarly when a file is read with sys_read() and its of type socket, the sockread() function will be be called that will put a lock in place and the mbuf is pulled from the top of the stack if it ready and not empty in which case it will sleep until it isn’t. Once an mbuf is found that isn’t empty the function will then check how many packets were requested as part of the read operation and wait until the number of bytes is met before releasing the lock and return the data collected from the data stored within the packet.