

# **IIT Madras**

## **ONLINE DEGREE**

**Modern Application Development – I**  
**Professor Nitin Chandrachoodan**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Madras**  
**What is Protocol?**

(Refer Slide Time: 00:16)

## Protocol

- What is a protocol
- Examples for HTTP



Hello, everyone, and welcome to this course on modern application development. So, now let us sort of try and understand. We have seen an example of what a web communication looks like. Let us try and understand a little bit more about what is a protocol and some examples in the context of HTTP.

(Refer Slide Time: 00:31)

## Protocol

- Both sides agree on how to talk
  - Walk into a room, greet each other, shake hands, sit down, chat about weather...
- Server expects "requests"
  - Nature of request
  - Nature of client
  - Types of results client can deal with
- Client expects "responses"
  - Ask server for something
  - Convey what you can accept
  - Read result and process



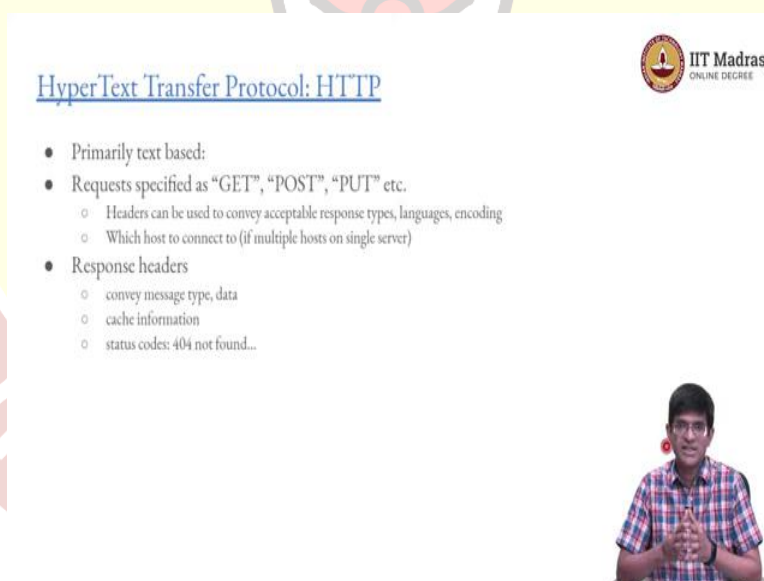
So, like I said, a protocol basically means that both sides have some communication agree on how to talk to each other. So, for example, when we walk into a room the usual protocol that

we are expected to follow is that we smile at whoever we see, we greet them, exchange names, maybe, possibly shake hands, those are all part of a protocol that we follow. That is between human beings.

In the same way you also have protocols for how computers are expected to communicate with each other. Now, if I connect to another computer on port 1500, I need to know what kind of information it is expecting and what it is likely to respond with. So, if I go there and just start typing text, the other side is most likely not going to understand what I am talking about, which is why we need to sort of have a protocol, which specifies the nature of my request, what kind of responses I am expecting to see, what I am as a client and what kind of responses I can actually deal with, so the server now has more information about me and can respond accordingly.

So, the client is also asking the server for something, the server is then responding based on what the client accepted. These two things, how do they communicate with each other is what constitutes the protocol.

(Refer Slide Time: 01:54)



The slide is titled "HyperText Transfer Protocol: HTTP" and features the IIT Madras logo in the top right corner. It contains a bulleted list of key HTTP features:

- Primarily text based:
- Requests specified as "GET", "POST", "PUT" etc.
  - Headers can be used to convey acceptable response types, languages, encoding
  - Which host to connect to (if multiple hosts on single server)
- Response headers
  - convey message type, data
  - cache information
  - status codes: 404 not found...

In the bottom right corner of the slide, there is a small video inset showing a man in a checkered shirt speaking.

Now, the hypertext transfer protocol is a very specific version of a type of protocol. It is primarily text-based. What I mean by that is, as you saw already, most of the actual information in the header information is encoded as text, so you can actually read the information, so you can see or GET and you can actually see it. It is not that, it is been interpreted as GET or something like that, it is not a number which got interpreted as GET, it actually has GET in the text, which means that if you happen to see the back and forth you

look at the bytes on the wire, as they are passing by, you can see that this is actually a GET request going to the server.

It also, the headers can be used to convey a lot of extra information. What kind of languages you are willing to accept, what kind of encodings what kind of response types you expect to see and so on. Similarly, the response headers will also be in text and convey information about the message itself. Also, about whether this information can be cached and used later, and most importantly, status codes.

We already saw the 200 OK. And like I said, the 404 not found, there are many other codes that are defined as part of the HTTP specification. These are obviously the most common ones. 200 OK, is what do you hope to see most of the time, 404 happens once in a while 500 is something you really do not want to see, because it means server error something crashed, the server crashed or something happened over there. There are similar ones. I mean, there are a bunch of like 300 errors, which are sort of harmless, they sort of are just warnings more than errors, and various different kinds of codes that are defined as part of the HTTP specification.

(Refer Slide Time: 03:37)

Use cases

- GET: simple requests, search queries etc.
- POST: more complex form data, large text blocks, file uploads
- PUT/DELETE/...
  - Rarely used in Web 1.0
  - Extensively used in Web 2.0
  - Basis of most APIs - REST, CRUD

IIT Madras  
ONLINE DEGREE

A small video inset shows a man with glasses and a red flower in his ear, wearing a plaid shirt, speaking.

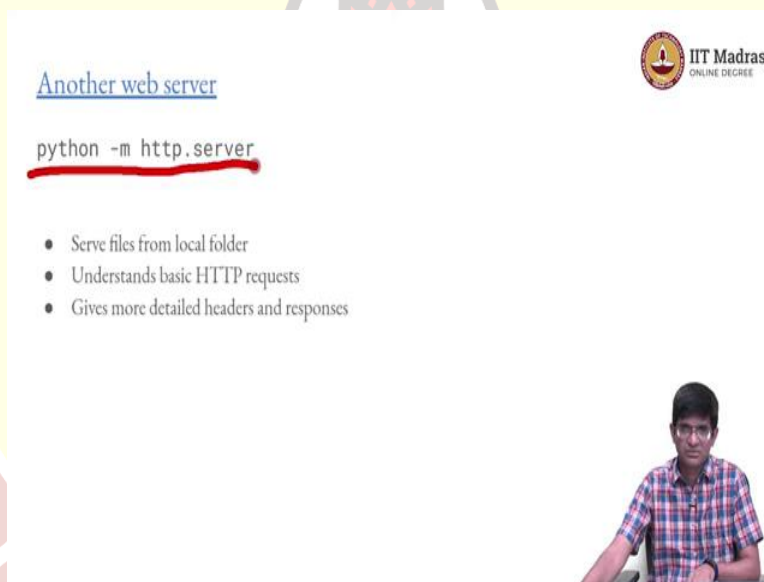
So, what are these use cases? As we saw already, with the simple web server GET is used for the most simple requests. It just request the information, gets back data. So, most of the time, things like search queries, and so on, when you are like not connecting to Google and doing a search, most of the time, what you would probably be doing would be a GET. POST can be used for more complex data. Let us say you have a form with multiple fields in it. There is a

name and address various other things, you cannot encode all of that into a GET at least you can do it easily.

POST is a much better way of doing this. It can accept large blocks of text. So, for example, even in Gmail, when you are typing in an email and hitting send, that entire block of information is finally being posted to the web server, which accepts it, puts it together, constructs an email out of it and sends it right.

There are many other requests as well. Some of them call put, some of them call delete, there are a few other variants, which are much less common, but are used quite extensively, especially in web 2.0 in the dynamic web and many sort of API's Application Programming Interfaces, including the so called REST interface and CRUD interfaces to databases that we will look at later are sort of built around these kinds of, these parts of the protocol.

(Refer Slide Time: 05:06)



So, just to give a few more examples, what I am going to do now is run something a slightly more complex web server. So, this is, again a one liner, I am just running the Python interpreter, and the dash m basically says, to execute something. And what is that something? It basically says take the module HTTP and run the function server from inside the HTTP module. So, in some sense, this is a one-line web server.

(Refer Slide Time: 05:45)

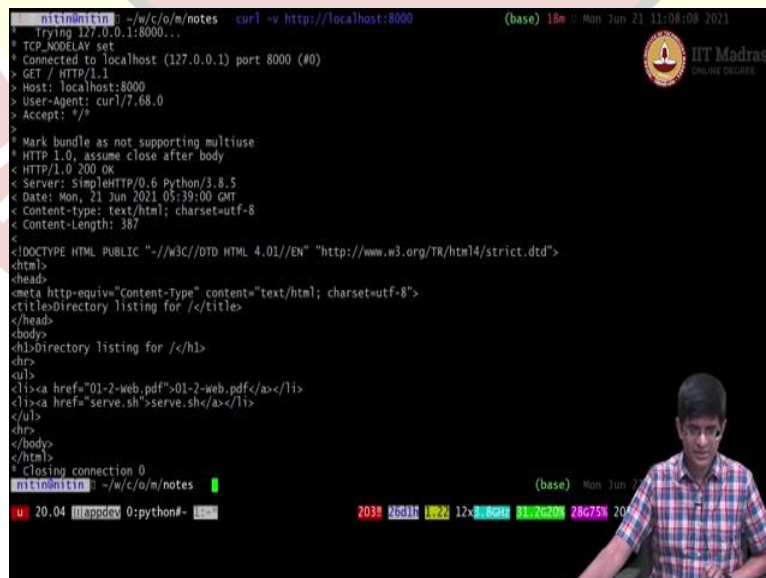


```
nitin@nitin: ~/n/c/o/n/notes$ python -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

The screenshot shows a terminal window with the command `python -m http.server` executed. The output is `Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...`. The terminal title bar indicates the user is 'nitin' and the current directory is `~/n/c/o/n/notes`. The system status bar at the bottom shows the time as 20:04 and the user as 'nitin'.

If I go back to my shell script, I can basically kill this, and clear this as well. And if I now run this command, Python minus m HTTP dot server, it now gives me a little bit more information than my shell script. It actually tells me serving HTTP on 0000. So, this is the local IP address. Well, more importantly, this actually says on any IP address that this machine is connected to, it also tells me that it is running on port 8000. I did not specify Port 8000, that just happens to be the default used by the Python HTTP server module.

(Refer Slide Time: 06:25)



```
nitin@nitin: ~/n/c/o/n/notes$ curl http://localhost:8000
Trying 127.0.0.1:8000...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8000 (#0)
> GET / HTTP/1.1
> Host: localhost:8000
> 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
< Server: SimpleHTTP/0.6 Python/3.8.5
< Date: Mon, 21 Jun 2021 05:39:00 GMT
< Content-type: text/html; charset=utf-8
< Content-Length: 387
<
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href="01-2-Web.pdf">01-2-Web.pdf</a></li>
<li><a href="serve.sh">serve.sh</a></li>
</ul>
</body>
</html>
Closing connection 0
nitin@nitin: ~/n/c/o/n/notes$
```


The screenshot shows a terminal window with the command `curl http://localhost:8000` executed. The output is a detailed HTTP response, including headers like `HTTP/1.0 200 OK`, `Server: SimpleHTTP/0.6 Python/3.8.5`, `Date: Mon, 21 Jun 2021 05:39:00 GMT`, `Content-type: text/html; charset=utf-8`, and `Content-Length: 387`. The body of the response is an HTML document with a title 'Directory listing for /' and a list of files: `01-2-Web.pdf` and `serve.sh`. The terminal title bar indicates the user is 'nitin' and the current directory is `~/n/c/o/n/notes`. The system status bar at the bottom shows the time as 20:04 and the user as 'nitin'.

And so now what happens if I once again run curl minus v http colon slash, slash localhost colon 8000. I can see that it actually gives me a lot more information out here. And this is



actually surprising. It is giving me information, which is much more than what I expected, because I do not really have anything over there. But if you look closely at it, you will realize that what it has done is it is automatically created a directory listing. It is showing me the list of files that I have.


(Refer Slide Time: 07:08)



### Another web server


```
python -m http.server
```

- Serve files from local folder
- Understands basic HTTP requests
- Gives more detailed headers and responses




So, rather than worrying about this, what I am going to do is take another couple of examples of what of requests that would go out when I run this Python web server. And if I use curl in order to connect to it.

(Refer Slide Time: 07:18)



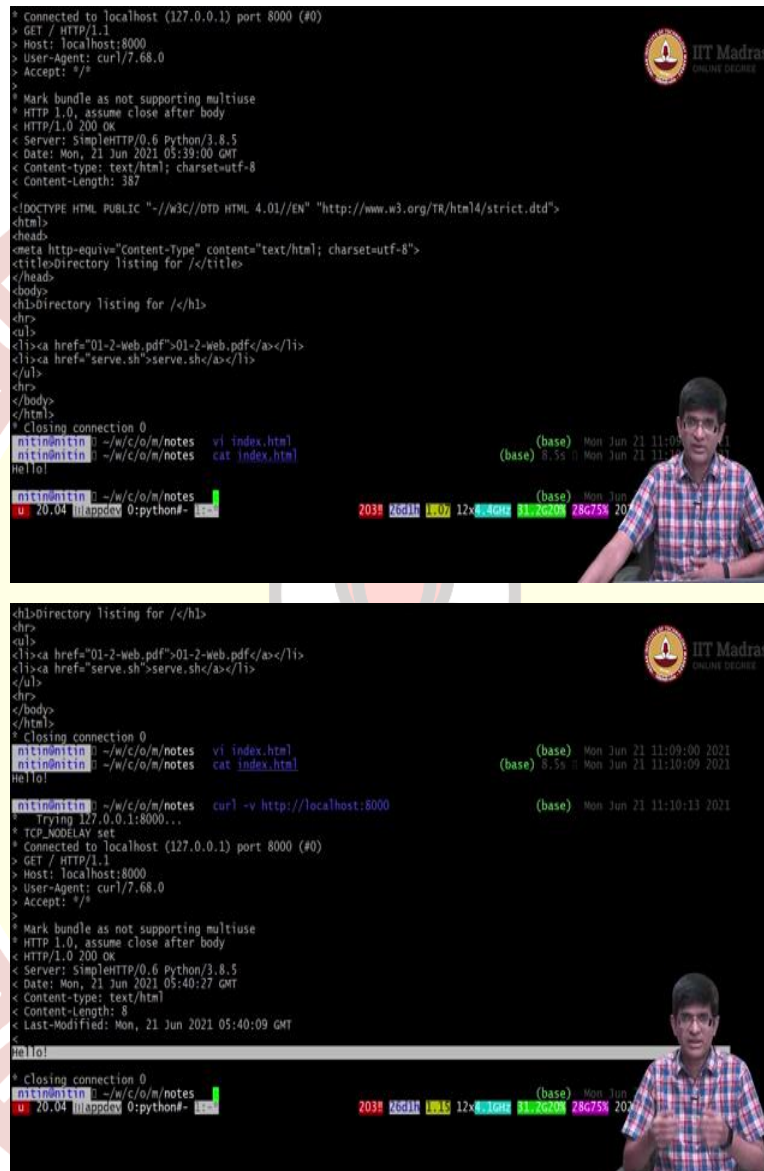
### GET /

```
* Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8000 (#0)
> GET / HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.64.1
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: SimpleHTTP/0.6 Python/3.9.2
< Date: Thu, 17 Jun 2021 03:11:15 GMT
< Content-type: text/html
< Content-Length: 31
< Last-Modified: Thu, 17 Jun 2021 03:10:18 GMT
<
<h1>Hello world</h1>
Hi there!
* Closing connection 0
```



If I do something like, get slash, and let us say, I did not have this particular thing. I did not have a directory listing. Normally, what it would do is take a file called text called index dot HTML.

(Refer Slide Time: 07:34)



```
* Connected to localhost (127.0.0.1) port 8000 (#0)
> GET / HTTP/1.1
> Host: localhost:8000
> 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
< Server: SimpleHTTP/0.6 Python/3.8.5
< Date: Mon, 21 Jun 2021 05:39:00 GMT
< Content-type: text/html; charset=utf-8
< Content-Length: 387
<
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href="01-2-Web.pdf">01-2-Web.pdf</a></li>
<li><a href="serve.sh">serve.sh</a></li>
</ul>
<hr>
</body>
</html>
* Closing connection 0
nitin@nitin: ~ - /w/c/o/n/notes vi index.html
nitin@nitin: ~ - /w/c/o/n/notes cat index.html
Hello!

nitin@nitin: ~ - /w/c/o/n/notes
u. 20.04 [!appdev 0:python#- 100%

(base) Mon Jun 21 11:09:00 2021
(base) 8.5% Mon Jun 21 11:09:09 2021

<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href="01-2-Web.pdf">01-2-Web.pdf</a></li>
<li><a href="serve.sh">serve.sh</a></li>
</ul>
<hr>
</body>
</html>
* Closing connection 0
nitin@nitin: ~ - /w/c/o/n/notes vi index.html
nitin@nitin: ~ - /w/c/o/n/notes cat index.html
Hello!

nitin@nitin: ~ - /w/c/o/n/notes curl -v http://localhost:8000
* Trying 127.0.0.1:8000...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8000 (#0)
> GET / HTTP/1.1
> Host: localhost:8000
> 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
< Server: SimpleHTTP/0.6 Python/3.8.5
< Date: Mon, 21 Jun 2021 05:40:27 GMT
< Content-type: text/html
< Content-length: 8
< Last-Modified: Mon, 21 Jun 2021 05:40:09 GMT
<
Hello!
* Closing connection 0
nitin@nitin: ~ - /w/c/o/n/notes
u. 20.04 [!appdev 0:python#- 100%

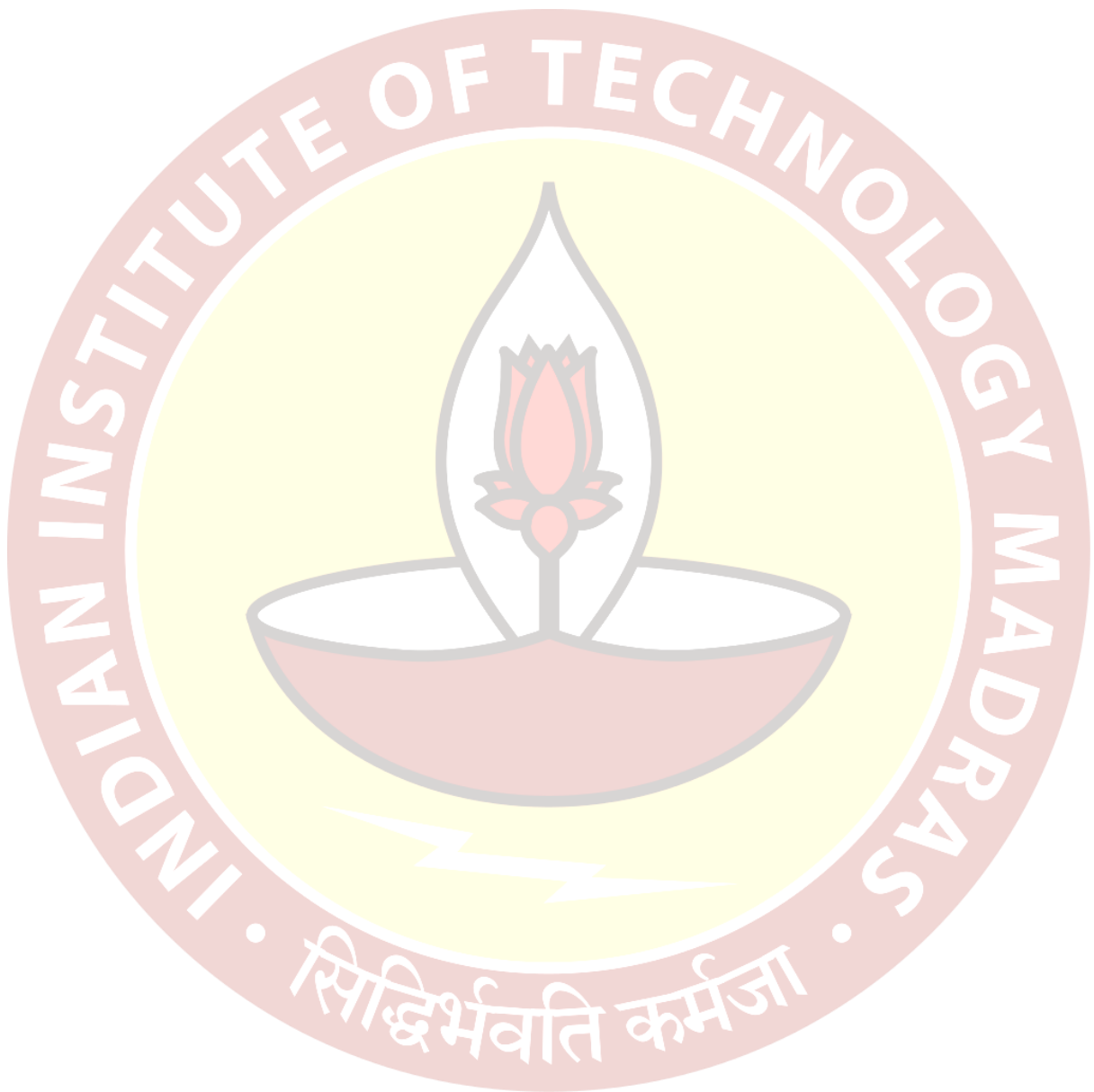
(base) Mon Jun 21 11:10:13 2021
```

Let us in fact, create a file like that. So, you can see that index dot HTML, I created it such that it just has one line saying Hello. Even though I call it a dot HTML file, it does not have any tags or anything that you would normally expect to see an HTML file.

Now, what would happen if I run the same thing again, I would find that this basically responds by giving me back whatever I had in index dot HTML. This is actually part of something called convention, it is not really part of the HTTP protocol. This is more of a convention, which says that in a given directory, if I find a file called index dot HTML and I



just get a request for the directory, GET slash or GET the directory name, go pick up whatever is in index dot HTML, and send that back to the requester.



(Refer Slide Time: 08:49)



```
GET /  
  
* Trying ::1...  
* TCP_NODELAY set  
* Connected to localhost (::1) port 8000 (#0)  
> GET / HTTP/1.1  
> Host: localhost:8000  
> User-Agent: curl/7.64.1  
> Accept: */*  
>  
* HTTP 1.0, assume close after body  
< HTTP/1.0 200 OK  
< Server: SimpleHTTP/0.6 Python/3.9.2  
< Date: Thu, 17 Jun 2021 03:11:15 GMT  
< Content-type: text/html  
< Content-length: 31  
< Last-Modified: Thu, 17 Jun 2021 03:10:18 GMT  
<  
<h1>Hello world</h1>  
Hi there!  
* Closing connection 0
```



So, what we can see? I am going back over here, let us look at it in a little bit more detail. Like I said, the first three lines essentially anything with the star is just debugging information. These five lines correspond to what was sent by curl, exactly, the same as previously, the only difference being that now it is connecting to Port 8000 because I told it that the server is running on port 8000.

This star again, basically is just you know, extra information. I mean, this debugging information from curl, but the response from the Python web server is this bunch of lines. Notice one thing, it is using HTTP 1.0 not 1.1. Unlike the previous one that we had. It makes life even simpler because 1.0 is actually a very simple version of the protocol 1.1 even though I had 1.1 over there actually requires a little bit more information for it to work. It also gives back information about itself. It says that it is a simple HTTP server with Python 3.9.2 that is the version of Python running over here.

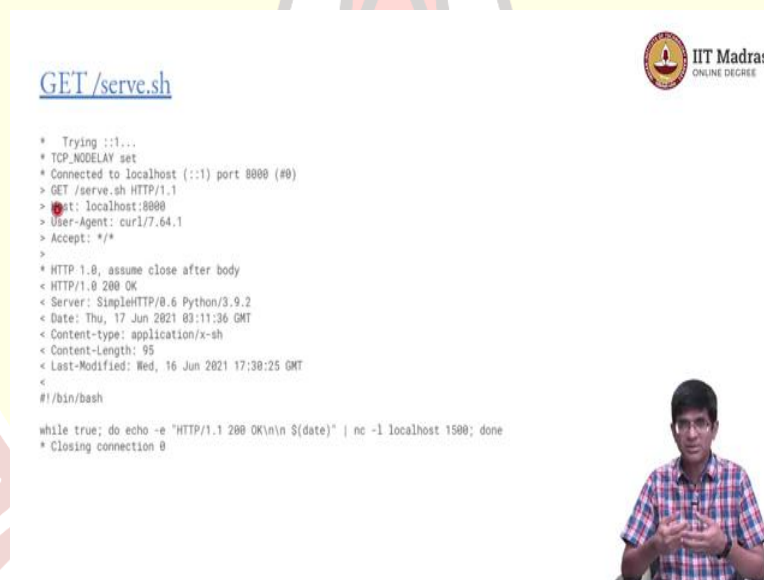
It gives the date, but this is part of the header. It is not something to be displayed and then comes the MIME type of the result. It basically gives the content type. And it says it is text slash HTML. The original one-line web server that I had did not have any of this information, but ideally, it should because this is what allows the browser to display the information correctly.

The content type and content length are actually important in pieces of information to have. In this case, this text slash HTML is useful. It basically says treat whatever you are getting

now as an HTML file, which means basically display it. If it said application slash PDF, then what would happen is it would download and probably open a preview or something to display it.

Content length is 31 bytes. Well, that is because I had some information out here, which was a total of around 31 bytes or so. It also gives a last modified basically telling, when was the file that it is sending to you? When was it last modified? Why is this information useful? Because from the clients' point of view, it might find that if you are trying to reload a page, but the server says it was last modified one week ago, the client might decide, okay, I do not need to do anything, I do not need to change what I am drawing on the page because nothing has changed. So finally, we have the actual text coming back from the index dot HTML file. So, this is sort of the transaction that goes back and forth when you do get slash. Just curl, HTTP colon, localhost colon 8000.

(Refer Slide Time: 11:41)



```
GET /serve.sh

* Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8000 (#0)
> GET /serve.sh HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.64.1
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.1 200 OK
< Server: SimpleHTTP/0.6 Python/3.9.2
< Date: Thu, 17 Jun 2021 03:11:36 GMT
< Content-type: application/x-sh
< Content-Length: 95
< Last-Modified: Wed, 16 Jun 2021 17:30:25 GMT
<
#!/bin/bash

while true; do echo -e "HTTP/1.1 200 OK\n\n $(date)" | nc -l localhost 1500; done
* Closing connection 0
```

Now, remember that I had this file server.sh sitting over there. And if I tried a request to that GET slash serve.sh, what do I expect will be the response? A similar set of headers from here, except that now the content type it now says is application slash x-sh, that is just a MIME type corresponding to a shell script and what is the actual data that comes back? The script itself? So, in other words, any file that you had, the server, if it is written appropriately, should be able to find out the MIME type, the content type, send that back to the client, and also send the data as required.