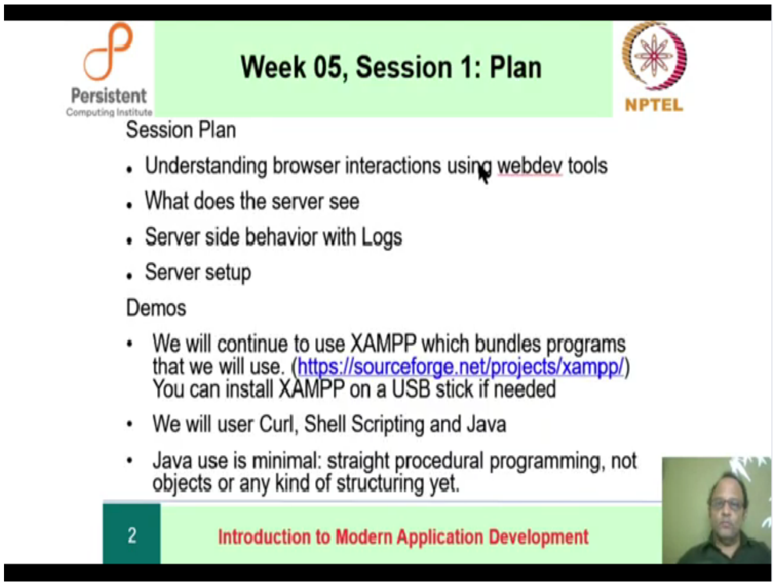
Introduction to Modern Application Development

Prof. Aamod Sane
FLAME University and Persistent Computing Institute
Abhijat Vichare
Persistent Computing Institute
Madhavan Mukund
Chennai Mathematical Institute

Lecture-18
Session 1-Part 2

Hello, welcome to modern application development.

(Refer Slide Time: 00:19)



Week 05, Session 1: Plan

Session Plan
- Understanding browser interactions using webdev tools
- What does the server see
- Server side behavior with Logs
- Server setup

Demos
- We will continue to use XAMPP which bundles programs that we will use. (https://sourceforge.net/projects/xampp/) You can install XAMPP on a USB stick if needed
- We will user Curl, Shell Scripting and Java
- Java use is minimal: straight procedural programming, not objects or any kind of structuring yet.

2 Introduction to Modern Application Development

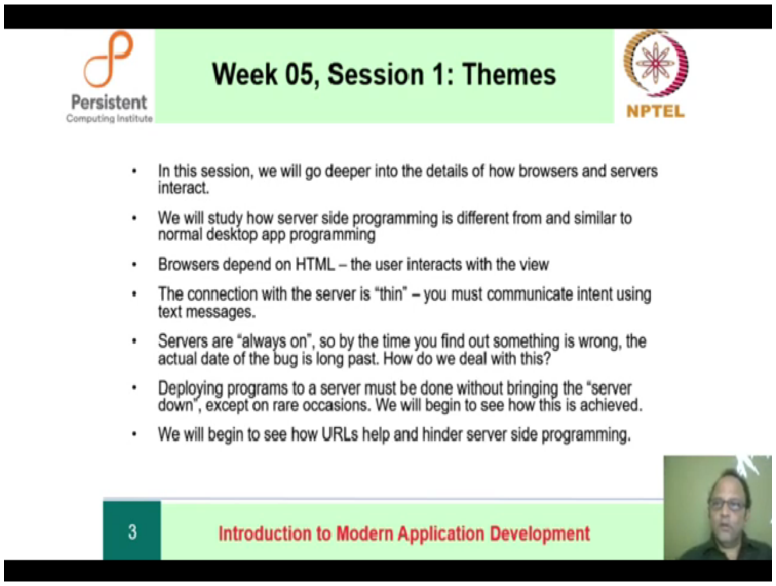
In this week, we are going to go beyond our basic understanding of HTTP that we looked at last time. We are going to study our applications by using the browser webdev tools, and also going to examine what it looked, like in detail, on the server side. Just as browser development tools allow us to understand how the browser views the requests and what it is doing with the current request. On the side of the server, we need tools to understand how it is that the server experiences our requests.

Our primary tool for this is logs and our server setup will have many features that go towards both understanding the server side behavior, as well as being able to deploy several services on the server at once. As we did last time, we will continue to use the XAMPP bundle and use apache and Tomcat and all the tools that it provides. As I noted last time, if you want to you can install xampp on a USB stick.

So, that you can carry the project around, as needed, which is very convenient. The other tools that we will use this time are Curl, Shell Scripting, and Java. For the shell, we will also use a few Unix utilities like awk, etc. to be able to write small scripts that show us what is happening on the server side when our requests get made. We will use our Java command line program wrapped as a CGI script as we saw last time, deliberately our user Java in this example is pretty minimal.

It is straight procedural programming with no object oriented or any other kind of structuring. Okay, now let us go ahead.

(Refer Slide Time: 02:16)



Week 05, Session 1: Themes

- In this session, we will go deeper into the details of how browsers and servers interact.
- We will study how server side programming is different from and similar to normal desktop app programming
- Browsers depend on HTML – the user interacts with the view
- The connection with the server is "thin" – you must communicate intent using text messages.
- Servers are "always on", so by the time you find out something is wrong, the actual date of the bug is long past. How do we deal with this?
- Deploying programs to a server must be done without bringing the "server down", except on rare occasions. We will begin to see how this is achieved.
- We will begin to see how URLs help and hinder server side programming.

3 Introduction to Modern Application Development

In this session, we are going to go deeper into the details of how it is that server programming is different from ordinary web programming, in particular, the details of how browsers and servers interact. Last time, we began with basic examples of interaction in the HTTP GET method, and this time, we are going to study our applications, which use both the GET method and the post method.

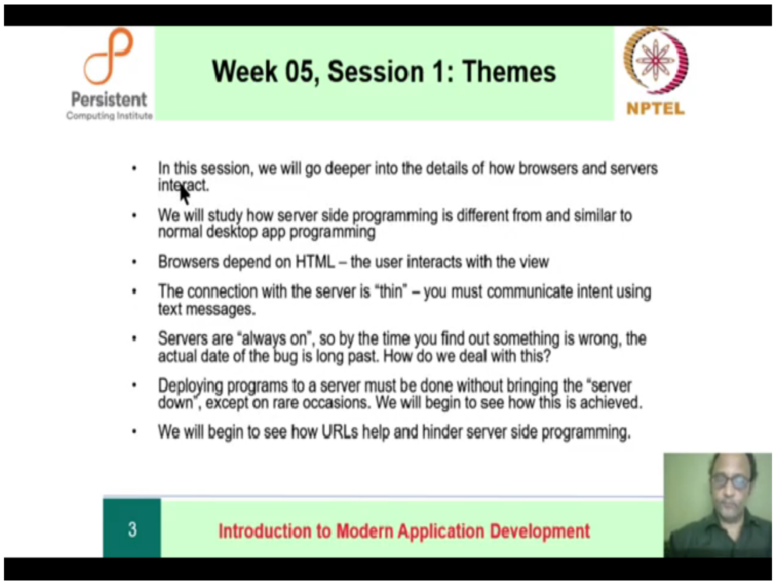
And we will see both of those with the browser web dev tools and take a look at what those tools have to tell us about our use of the facilities of the HTTP protocol of the browser and of the server. Since browsers depend on HTML, and they render that HTML into a view for the users,. We will also study some of the aspects of how HTML can be seen on the browser side, in the form of a model called a document object model, which is a set of objects inside the browser that are used to put together a tree of document elements, which is what we actually experienced as users.

One interesting thing about the split of the program into two pieces; the browser and the server, is that the connection between them tends to be rather thin. The browser has to convert our interaction into just enough data for the server to be able to compute with and return the results that we care about. The primary difference on the server side is there unlike a desktop program which gets started and stopped, there is no such thing as stopping a server side program.

If one user is not using it another one is, this means the program is perpetually in use. And so making changes to the programs has to be done in such a way that people do not experience any kind of disruption, or at least that the disruption they experience is absolutely minimal. Therefore, deploying programs to servers in such a way that people can use them is an art in its own right.

And we will see the beginnings of this as we look at how pieces of a program are assembled and deployed to servers. We will also begin to understand how the existence of URLs is both a help and a hindrance to the creation of server side programs.

(Refer Slide Time: 04:59)


Week 05, Session 1: Themes

- In this session, we will go deeper into the details of how browsers and servers interact.
- We will study how server side programming is different from and similar to normal desktop app programming
- Browsers depend on HTML – the user interacts with the view
- The connection with the server is "thin" – you must communicate intent using text messages.
- Servers are "always on", so by the time you find out something is wrong, the actual date of the bug is long past. How do we deal with this?
- Deploying programs to a server must be done without bringing the "server down", except on rare occasions. We will begin to see how this is achieved.
- We will begin to see how URLs help and hinder server side programming.

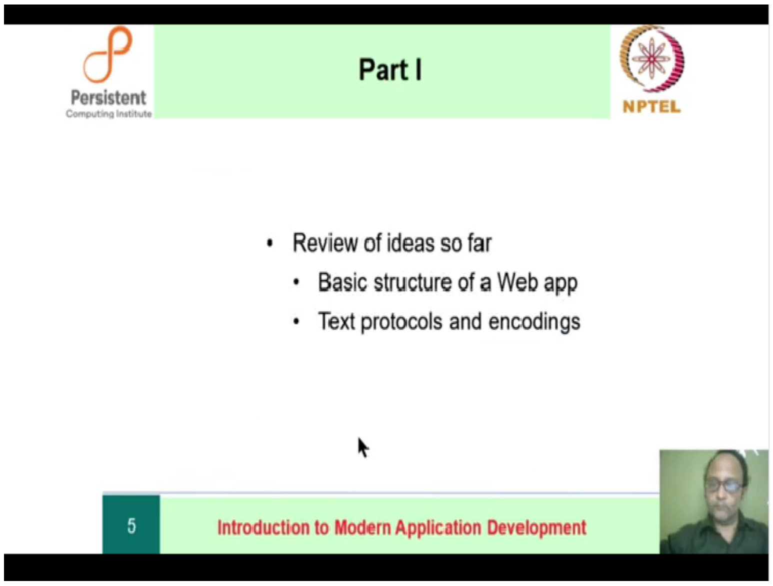
3 Introduction to Modern Application Development

Let us examine all the elements of the system that we are going to study this time. First, we have as we had the last time but unlike last time where the browser was simply something that sends HTML requests, HTTP requests, and expected HTML responses. This time we are going to spend some time on the DOM, which is the internal structure that the browser has. Next, when the browser sends an HTTP request to the server, we are going to see how it deals with the contents of a form.

We have seen one example of form contents when we looked at search boxes last time, but this time, we are going to look at it when we are submitting some data, which is supposed to change what is there on the server. Then, for the HTTP response, we will have HTML and CSS and we will take some time out to see what that structure looks like. On the server side, we are going to look at the deployment of programs.

And more specifically, we are going to look at the nature of log files, both for the servers by themselves, and log files which are specialized for our application.

(Refer Slide Time: 06:21)



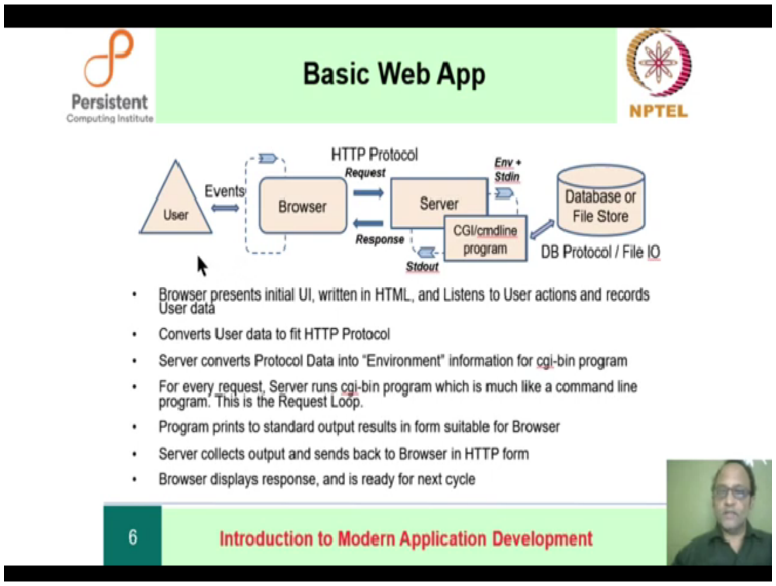
Part I

- Review of ideas so far
  - Basic structure of a Web app
  - Text protocols and encodings

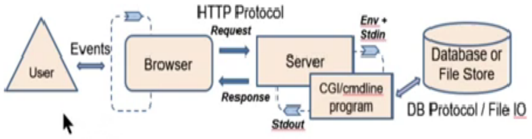
5 Introduction to Modern Application Development

Let us begin with a review of the things we have discussed last time. What we have here is the basic structure of a web app and the text protocols and encodings.

(Refer Slide Time: 06:33)



Basic Web App



- Browser presents initial UI, written in HTML, and Listens to User actions and records User data
- Converts User data to fit HTTP Protocol
- Server converts Protocol Data into "Environment" information for cgi-bin program
- For every request, Server runs cgi-bin program which is much like a command line program. This is the Request Loop.
- Program prints to standard output results in form suitable for Browser
- Server collects output and sends back to Browser in HTTP form
- Browser displays response, and is ready for next cycle

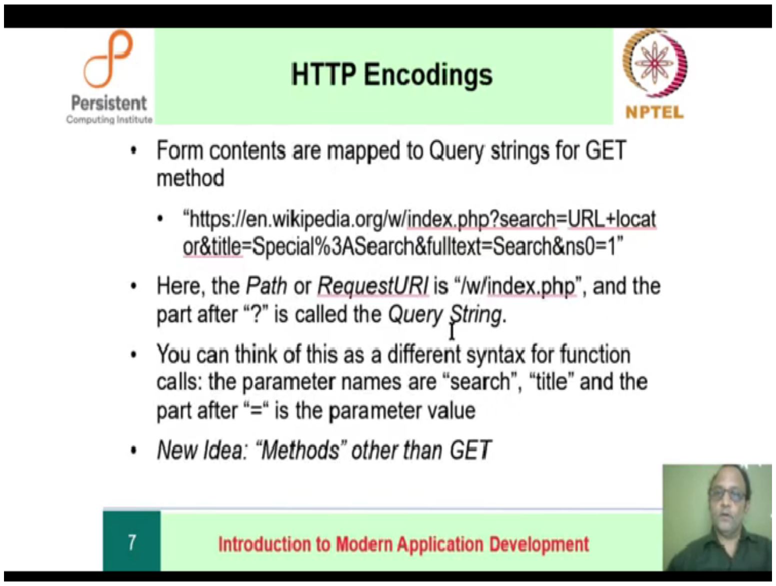
6 Introduction to Modern Application Development

From last time, there is the user whose events go to an event loop which is running within the browser. The browser presents the UI to the user. It is written in HTML and such things as filling in text and buttons are the primary means of user interaction. Then the browser converts the user requests into HTTP protocol requests and will eventually get a response back from the server.

Meanwhile, the server gets the data from the browser gives it to the CGI or the command line program in the form of environment variables and standard input and returns output; s standard out. And finally it may or may not use a database or a file store. Usually, of course, because we want to have long duration data, the programs on the server side will inevitably be storing something. So, the next time when the user comes back their data can be retrieved and given to the user.

All this we are very familiar with and this is the cycle that we have to keep in mind as we start analyzing the pieces of this cycle. Okay.

(Refer Slide Time: 07:50)



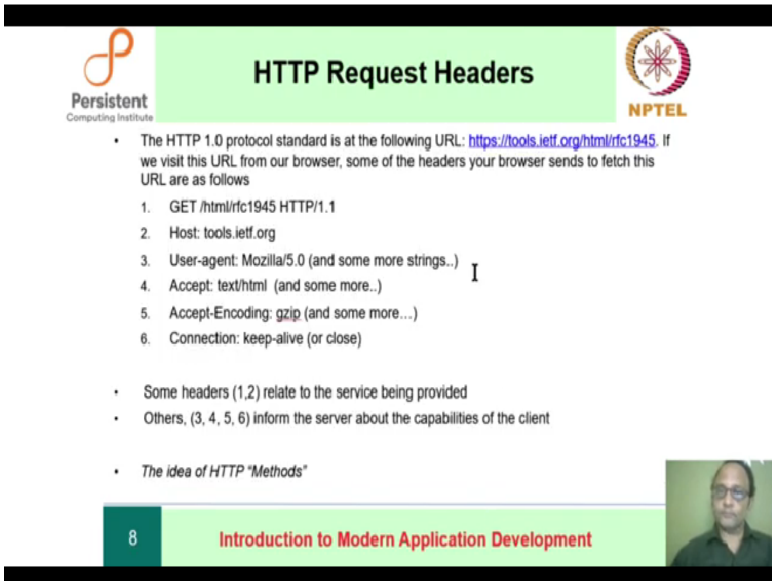
HTTP Encodings

- Form contents are mapped to Query strings for GET method
  - "https://en.wikipedia.org/w/index.php?search=URL+locator&title=Special%3ASearch&fulltext=Search&ns0=1"
- Here, the *Path* or *RequestURI* is "/w/index.php", and the part after "?" is called the *Query String.*
- You can think of this as a different syntax for function calls: the parameter names are "search", "title" and the part after "=" is the parameter value
- *New Idea: "Methods" other than GET*

7 Introduction to Modern Application Development

Next thing we understood last time was; form contents are mapped into query strings. But that is what happens for what is called the GET method. And what this exactly means is what we are going to discuss this time. As we saw last time, we have a query string. And also you can think of this as a kind of, you know different syntax for a function bar. But the new idea this time is that there are methods other than just the GET method that we normally saw last time. Okay.

(Refer Slide Time: 08:25)



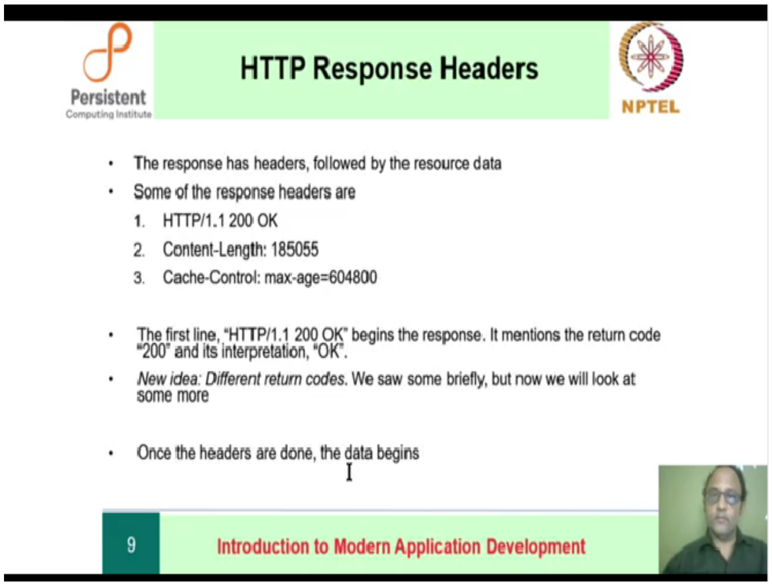
HTTP Request Headers

- The HTTP 1.0 protocol standard is at the following URL: https://tools.ietf.org/html/rfc1945. If we visit this URL from our browser, some of the headers your browser sends to fetch this URL are as follows
  1. GET /html/rfc1945 HTTP/1.1
  2. Host: tools.ietf.org
  3. User-agent: Mozilla/5.0 (and some more strings..)
  4. Accept: text/html (and some more..)
  5. Accept-Encoding: gzip (and some more…)
  6. Connection: keep-alive (or close)
- Some headers (1,2) relate to the service being provided
- Others, (3, 4, 5, 6) inform the server about the capabilities of the client
- *The idea of HTTP "Methods"*

8 Introduction to Modern Application Development

Finally, last time, we had this other idea, which is the notion that there are such things as headers, which help organize the protocol data, these headers in 1 and 2, in particular relate to the service that is being provided. While the rest of the header is about user agent and what kinds of data the user agent accepts, etc. relates to the capability of the client. The new idea here is to call out this particular part as what is called an HTTP method. okay.

(Refer Slide Time: 08:59)



HTTP Response Headers

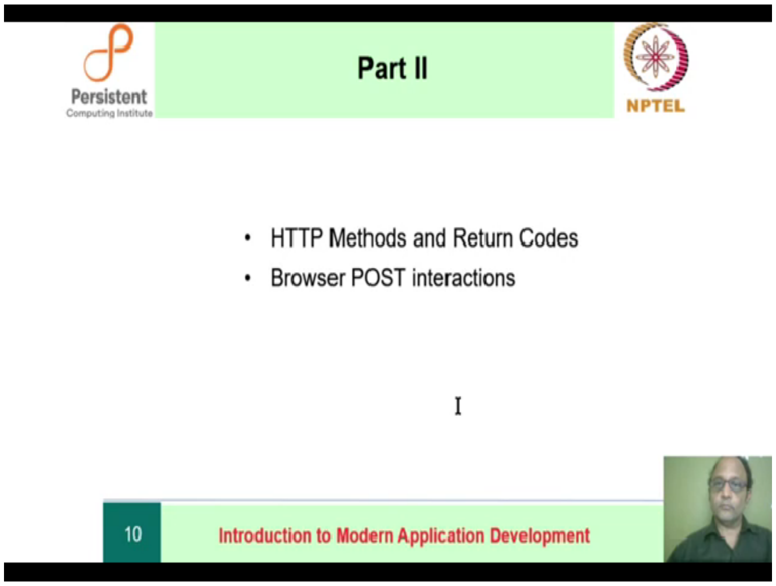
- The response has headers, followed by the resource data
- Some of the response headers are
  1. HTTP/1.1 200 OK
  2. Content-Length: 185055
  3. Cache-Control: max-age=604800
- The first line, "HTTP/1.1 200 OK" begins the response. It mentions the return code "200" and its interpretation, "OK".
- *New idea: Different return codes.* We saw some briefly, but now we will look at some more
- Once the headers are done, the data begins

9 Introduction to Modern Application Development

The next thing we studied last time was HTTP response headers, which contain the response followed by whatever data the server wants to send. Here, the new thing that we will dwell on this time is the existence of error codes and their interpretations. We have seen some of them briefly, but this time we will see some more. And once the headers are done, the data begins.

These headers are something you will become very familiar with once you become a web programmer.

(Refer Slide Time: 09:37)



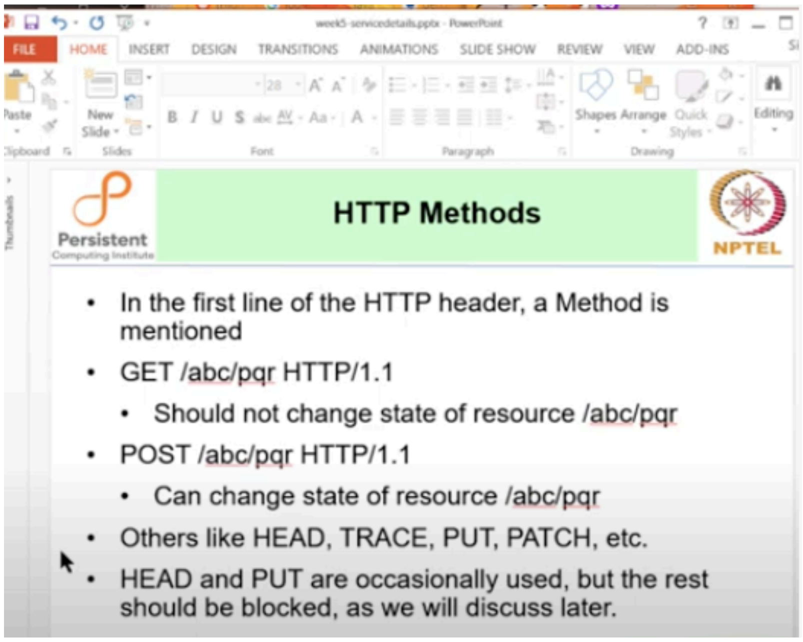
Part II

- HTTP Methods and Return Codes
- Browser POST interactions

Persistent
Computing Institute
NPTEL
10
Introduction to Modern Application Development

That is the review for what we have so far. Now let us start looking at the details for this session.

(Video Starts: 09:45)



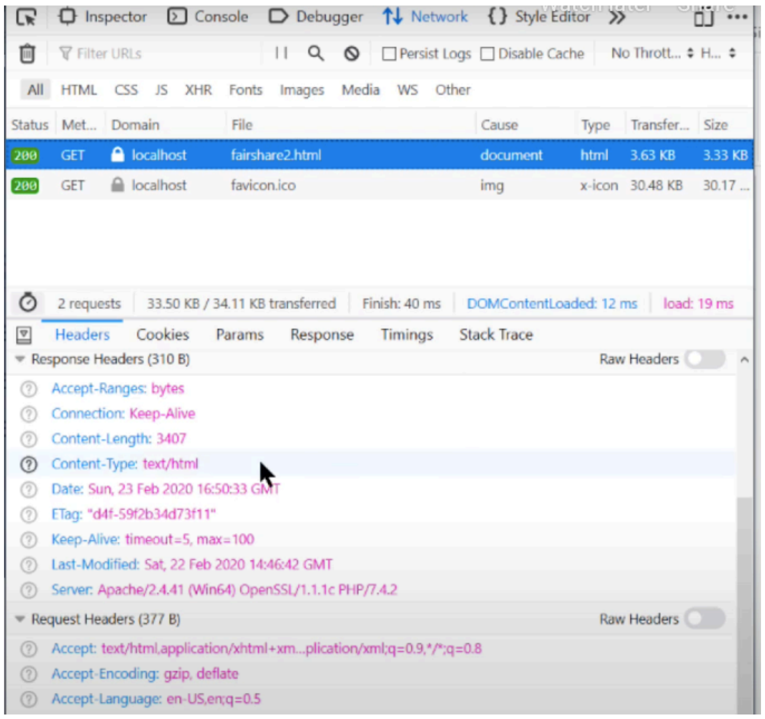
HTTP Methods

- In the first line of the HTTP header, a Method is mentioned
- GET /abc/pqr HTTP/1.1
  - Should not change state of resource /abc/pqr
- POST /abc/pqr HTTP/1.1
  - Can change state of resource /abc/pqr
- Others like HEAD, TRACE, PUT, PATCH, etc.
- HEAD and PUT are occasionally used, but the rest should be blocked, as we will discuss later.

Let us begin our study of the HTTP methods and return codes by looking at our command line app. As we saw in the first line of HTTP header, a method is mentioned, if the method name is GET, then a request is made. And the expectation with GET is that when we request the resource abcpqr, if we use the method get then we are not going to change the state of whatever resource is there.

However, if the POST method is used, then whatever resource we have requested, we can change its state. In fact, POST method is there in order to ask the resource to be changed. Besides GET and POST there are other methods like HEAD, TRACE, PUT, PATCH, etc. HEAD and PUT occasionally used, but the rest of the methods should actually be blocked. Although, they have been designed for convenience given that the internet turned into difficult place to keep secure you do not want to end expose any facilities to the outside world that somebody might use.

Because you inadvertently did not know that they existed. This keeping minimal facilities for the outside world is called the reduction of the attack surface, which is an idea that we will look at later when we understand what it takes to secure our systems. Now, let's begin by looking at demos for various elements of our application. Let us start by opening our web browser tools. Here we go. And what we are going to do is to begin by shift reloading that location, so that we see the natural form in which the application comes to us.



Status	Met...	Domain	File	Cause	Type	Transfer...	Size
200	GET	localhost	fairshare2.html	document	html	3.63 KB	3.33 KB
200	GET	localhost	favicon.ico	img	x-icon	30.48 KB	30.17 ...

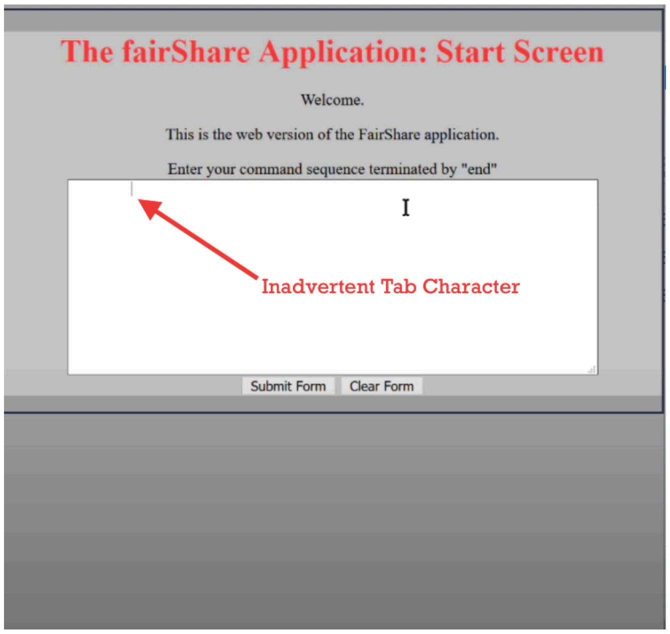
```
2 requests | 33.50 KB / 34.11 KB transferred | Finish: 40 ms | DOMContentLoaded: 12 ms | load: 19 ms
```

```
Response Headers (310 B)                                   Raw Headers
Accept-Ranges: bytes
Connection: Keep-Alive
Content-Length: 3407
Content-Type: text/html
Date: Sun, 23 Feb 2020 16:50:33 GMT
ETag: "d4f-59f2b34d73f11"
Keep-Alive: timeout=5, max=100
Last-Modified: Sat, 22 Feb 2020 14:46:42 GMT
Server: Apache/2.4.41 (Win64) OpenSSL/1.1.1c PHP/7.4.2
Request Headers (377 B)                                    Raw Headers
Accept: text/html,application/xhtml+xm...plication/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
```

And here we can see what the requests look blank. First of all, we have the GET method. The status is 200. So, we know everything okay, the domain, of course is localhost. And this is the URL we are looking at it is the index HTML document and the transfer and size and so on. As we looked at last time, fav icon is a standard request that is made for the first time pages visited.

Nothing unexpected here, we have the URL. We have these various headers. And you can take a look at like a said, any application there you happen to use to figure out what it is actually doing. So far so good, nothing unexpected. Now let us see what happens, when we first try to enter in this box. There's something interesting the cursor is actually blinking at a tab. So, somehow a tab seems to have gotten inserted.

This kind of thing, it does not matter much for simple applications. But careful designers take a look at the detail fit and finish as it were of the application. So, somewhere, we need to take some action to remove this extra tab character. This is shown in the picture given below.

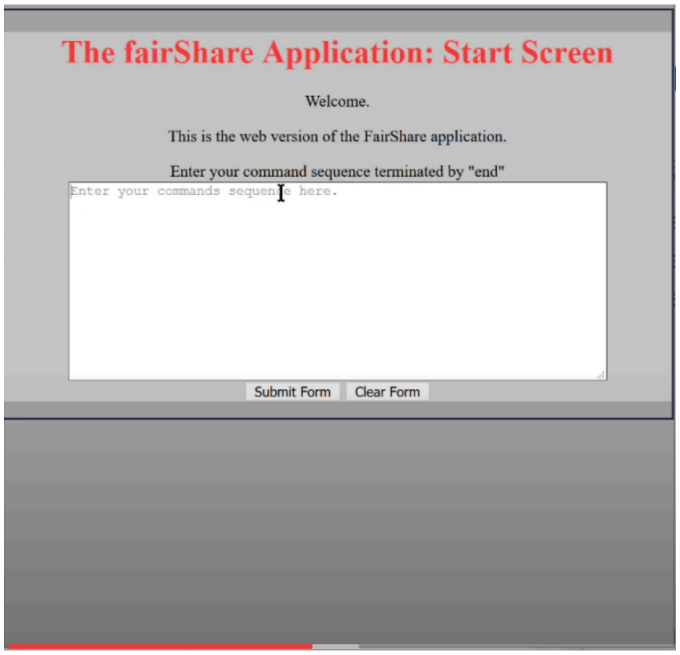

The fairShare Application: Start Screen

Welcome.

This is the web version of the FairShare application.

Enter your command sequence terminated by "end"

But for now, let us just remove the tab. Ahah! As we remove the tab, we see that there is actually a sentence here saying 'Enter your command sequence here', which is a sort of a help text that we have not heard.



The fairShare Application: Start Screen

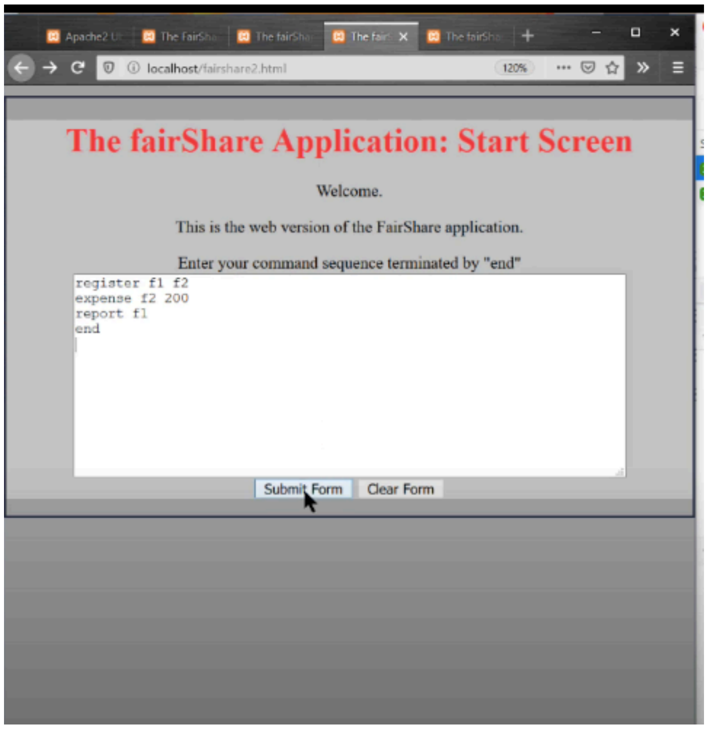
Welcome.

This is the web version of the FairShare application.

Enter your command sequence terminated by "end"

```
Enter your commands sequence here.
```

We can handle the problem with the *tab character* later. For now, let us enter the following commands:



The fairShare Application: Start Screen

Welcome.

This is the web version of the FairShare application.

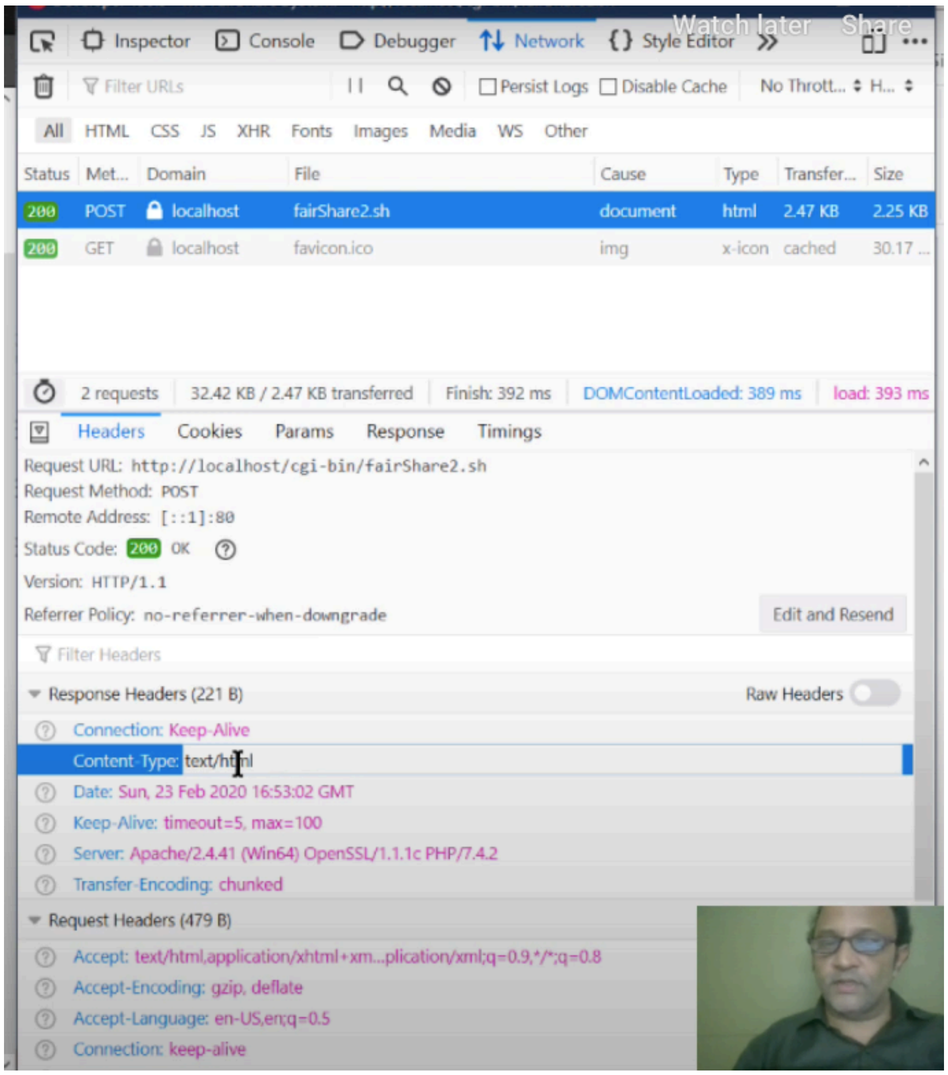
Enter your command sequence terminated by "end"

```
register f1 f2
expense f2 200
report f1
end
```

- registered f 1, f 2 and
- we have, let's say, expense f 2 200 and let's ask for report on f 1 behalf and then

- We have end, as it says “Enter your command sequence terminated by ‘end’ ”, we submit the form and we get the result. This is telling us that these are the register roommate.

These are the expenses. This is what f1 post. And the database details are also printed out. On the developer tool side, we have the POST method. And we have the URL that was called which is FairShare2.sh rather than FairShare2.html. But there is something a little odd in the sense that, although, the documents name is ‘sh’, the type is HTML. So, this is one of those interesting things about the web. In the beginning, we saw that a file that ends with dot HTML is also of type HTML.



Status	Met...	Domain	File	Cause	Type	Transfer...	Size
200	POST	localhost	fairShare2.sh	document	html	2.47 KB	2.25 KB
200	GET	localhost	favicon.ico	img	x-icon	cached	30.17 ...

2 requests | 32.42 KB / 2.47 KB transferred | Finish: 392 ms | DOMContentLoaded: 389 ms | load: 393 ms

```
Request URL: http://localhost/cgi-bin/fairShare2.sh
Request Method: POST
Remote Address: [::1]:80
Status Code: 200 OK
Version: HTTP/1.1
Referrer Policy: no-referrer-when-downgrade

Response Headers (221 B)
Connection: Keep-Alive
Content-Type: text/html
Date: Sun, 23 Feb 2020 16:53:02 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.41 (Win64) OpenSSL/1.1.1c PHP/7.4.2
Transfer-Encoding: chunked

Request Headers (479 B)
Accept: text/html,application/xhtml+xm...plication/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
```

But nothing in the system actually looks for any connection between the names of the file and the type of the file. The type of the file is given purely by a kind of response header such as **Content-Type: text/html.** So, just as the response header here says text/HTML, which is actually the file type. It has nothing to do, whatsoever, with any kind of extension or anything similar. We made a POST call and our request was like this.

In all this, if you take a look in the fairShare2.sh document file, which is given on the right side of the screen in the app, it says content length is 73, which means those sequence of commands that we typed were 73 bytes. It says content type is *application/x www form urlencoded*, which means the data was somehow encoded when it was sent to the server. But, where is the data? This data is normally not shown. And there is a sort of indirect step that you have to take, to look at the data.

If you press this 'Edit' and 'Resend' button here, you will find that now the tools allow you to actually make a POST request. And if you need to, you can edit this data. Of course, compared with our original data, let us take a look at the original data by going back the original data. So this is how the commands we type looked like.

But that is not what they seem to be in the inspector. So, let us submit all over again. And let's see this time what happens when we go to 'Edit' and 'Resend'. Okay. Now, compare this with your memory of, unfortunately, the requests that we had made earlier because if we go back then the system will change again. So, we have registered f 1 and f 2 and after that, so first of all the spaces have been changed to plus and carriage return line feed the sequence slash r slash n has been encoded as 0 D 0A, which is a hexadecimal encoding of those special characters.

And then we follow the same pattern for the rest. So, this is what it means to have content type x www form urlencoded, this kind of encoding is called URL encoding. This part which is commands equal to comes from the name of the form, as we will see shortly. Then testing it is very useful to be able to re-edit and send a certain request all over again. And because if there is a small typo somewhere you do not have to recreate everything you can just do some sort of local change.

What happens if we actually were to edit and resend, is something we will take a look at later. For now, we are just going to go ahead to the next step. Let us solve the small puzzle of the tab or extra space that we get when we first see the FairShare app. So, here there is an extra tab. Normally, HTML is not supposed to be sensitive to space. For example, if we interchange the places of *p* and *div* and put add some extra space, somewhere.

I save it and reload it and then do our usual thing, which is to register someone and just test it, find out what happens. And let's try and reload this thing to get back to the state that we want. So, where is this tab coming from? Unfortunately, it turns out that if you look at this carefully, you will see that there is an extra tab where the text area ends; that is right before </TEXTAREA>

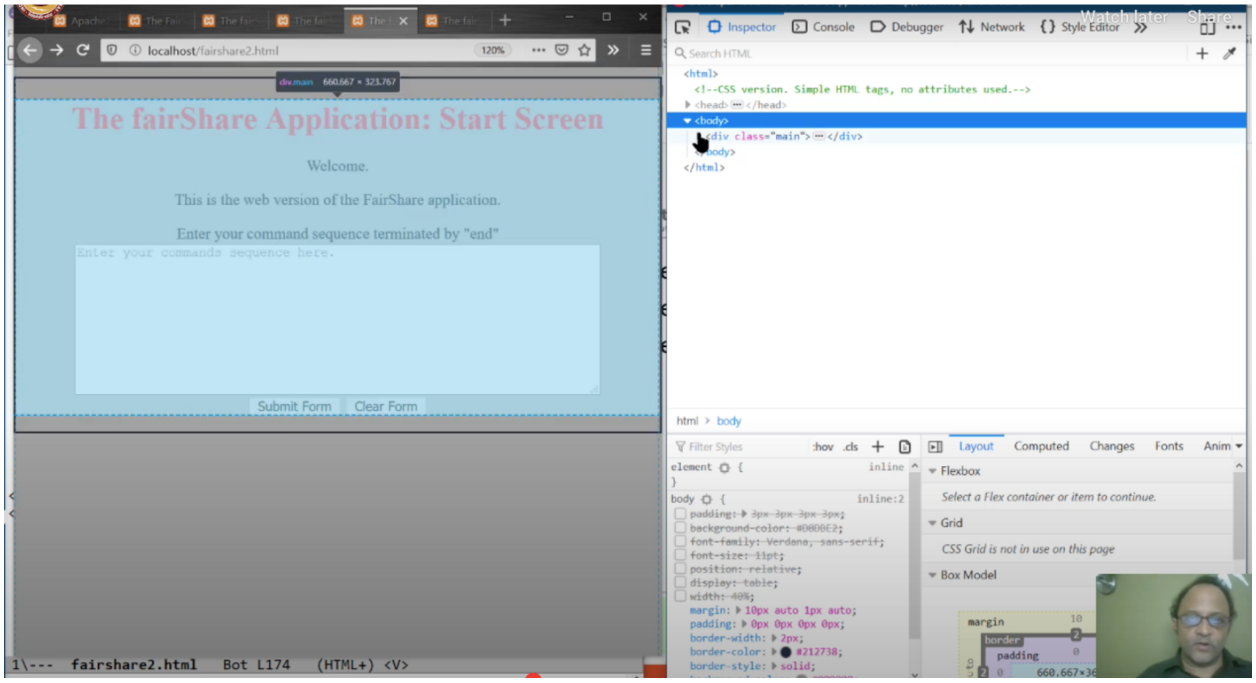
So, it is somewhat annoying that HTML is suddenly sensitive to these kinds of things in a few odd places, and otherwise, it is not anywhere else. But that is one of those things that we simply have to live with and extra spaces, especially in templates, and so forth, is one of those minor niggling details that can sometimes have surprising consequences. So, let us try and remove this space from here completely.

And then go back and see what happens, let us reload our program and now, our help sentence, which we put here as a placeholder is actually visible the first time you load it. So, you have to be quite sensitive to these kinds of things, especially when it comes to the browser. There are not too many of those, but I am giving you very simple examples to sensitize you to be careful in looking in detail at what you see when you are doing web programming.

The next example that we will have of this, unfortunately, is more severe. To see this example, let us do the following: Here in this form, we have ID tags. How the form is organized is like this there is a form tag, which ends with </FORM>. And inside the form tag, there is a label and there is a text area with a command sequence placeholder, some columns, some rows and so on and so forth. And you have two buttons: you have a button, submit and reset and whose status or submit form and clear form.

And this Id, *FairShareInput*, is useful on the server side to find out which button was pressed. And let me just make sure that you understand the nesting of the form, the label, the text area and the input on the actual display. So, on the display, what we have here is: ‘Enter your command sequence terminated by ‘end’’.

The label part is right above the text area. And the text area is over here and then the input buttons, right below the text area. Let us see what they look like first in the browser tools. So, this time we will go to the inspector and the inspector shows you the elements on the screen by blurring everything else on interface of the app. When you hover, it clears the part that you are inspecting and everything else still remains blurry.



The fairShare Application: Start Screen

Welcome.

This is the web version of the FairShare application.

Enter your command sequence terminated by "end"

Enter your commands sequence here.

```
<html>
  <!--CSS version. Simple HTML tags, no attributes used.-->
  <head>…</head>
  <body>
    <div class="main">…</div>
  </body>
</html>
```

```
element {                                   inline
}
body {                                      inline:2
  padding: 3px 3px 3px 3px;
  background-color: #D8DBE2;
  font-family: Verdana, sans-serif;
  font-size: 11pt;
  position: relative;
  display: table;
  width: 40%;
  margin: 10px auto 1px auto;
  padding: 0px 0px 0px 0px;
  border-width: 2px;
  border-color: #212738;
  border-style: solid;
```

Select a Flex container or item to continue.

CSS Grid is not in use on this page

```
1\---  fairshare2.html    Bot L174    (HTML+) <V>
```

And if you watch in this area, when I hover over the body, we observe that the sections like <body>, <div> or the primary application named as *<div class="main">* , the part covered by the code in the corresponding section is highlighted on the screen. This is the header, which it is highlighting. There will be a welcome tag. Here, we find different sections for paragraphs and form. So, let us take a look. This is the web version of the fairShare application.

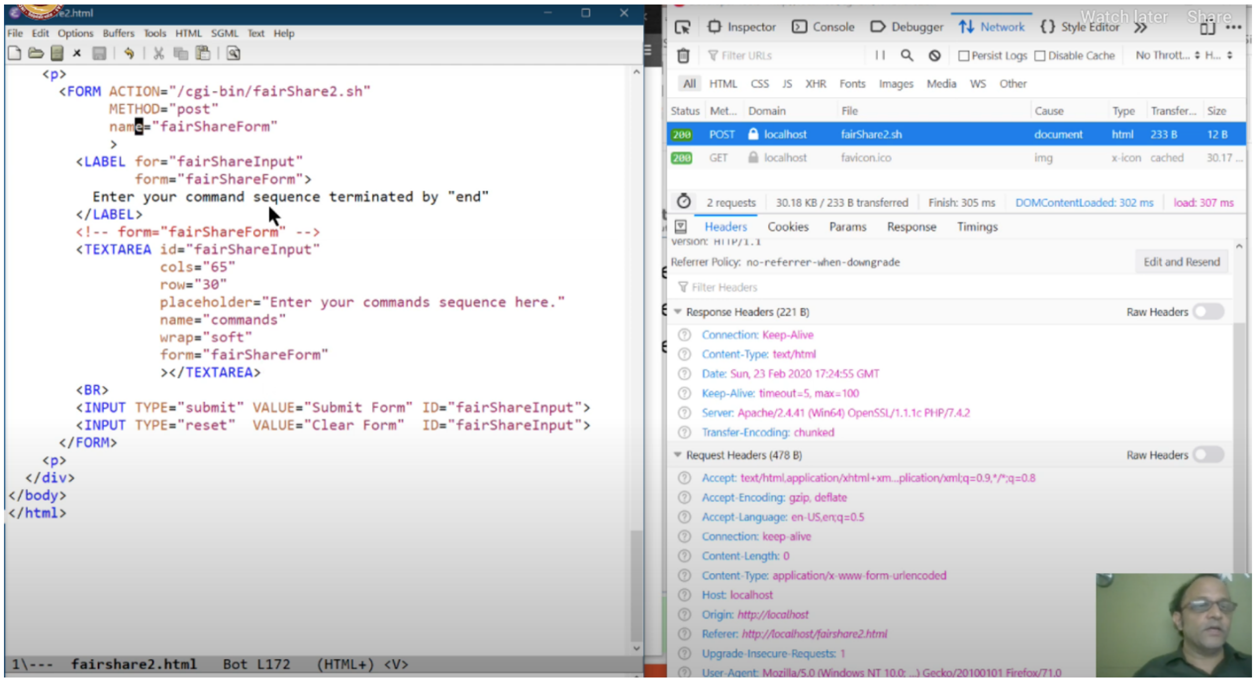
Then we have this separator ‘</p>’ which is not doing anything much, it is the separator is indicated by those dotted lines. Then you have the form, and the label of the form. Then you have the text area alone. And we have a small separator. We have one button.

It is highlighting all these elements on the other side. And we also have the other button. Finally, the form ends at this point, and there is a little space before the form ends.

Normally, these days you do not have to create these kinds of things. You can take care of it in a more sophisticated form. But for simple applications, like this, whatever does the job is typically good enough. Okay, now let us take a look at the problem that that sometimes can be created by things like this. And for that, we will go back to our network tab. Let us make a change and see what happens we know that this form works perfectly fine.

But let us do this. Something like the form has many different tags in it: one of these happens to be called name. So, instead of ID, let us change this thing to name and write this thing. Go back. Now, I am going to reload and I will put the commands we want f1,f2, f1 f3, whatever, expense f1 200. And let’s submit the form. What happened? We submitted the form but there is no result at all. As we will see for this example later, if you go and look at it on the server side, you can find out what has gone wrong.

But this is something interesting you can find out from the client side, as well. If we look at what happened here, it turns out that the request headers tell us that the content length is zero, which basically means that all the text that we entered was never sent by the browser. Well, it turns out that there are many different rules plus heuristics that browsers use to decide what to do when something unexpected happens.



```
    <p>
      <FORM ACTION="/cgi-bin/fairShare2.sh"
            METHOD="post"
            name="fairShareForm"
            >
        <LABEL for="fairShareInput"
               form="fairShareForm">
          Enter your command sequence terminated by "end"
        </LABEL>
        <!-- form="fairShareForm" -->
        <TEXTAREA id="fairShareInput"
                  cols="65"
                  row="30"
                  placeholder="Enter your commands sequence here."
                  name="commands"
                  wrap="soft"
                  form="fairShareForm"
                  ></TEXTAREA>
        <BR>
        <INPUT TYPE="submit" VALUE="Submit Form" ID="fairShareInput">
        <INPUT TYPE="reset"  VALUE="Clear Form"  ID="fairShareInput">
      </FORM>
    <p>
  </div>
</body>
</html>

1\---  fairshare2.html    Bot L172    (HTML+) <V>
```

Status	Met...	Domain	File	Cause	Type	Transfer...	Size
200	POST	localhost	fairShare2.sh	document	html	233 B	12 B
200	GET	localhost	favicon.ico	img	x-icon	cached	30.17 ...

```
2 requests   30.18 KB / 233 B transferred   Finish: 305 ms   DOMContentLoaded: 302 ms   load: 307 ms

Version: HTTP/1.1
Referrer Policy: no-referrer-when-downgrade

Response Headers (221 B)
Connection: Keep-Alive
Content-Type: text/html
Date: Sun, 23 Feb 2020 17:24:55 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.41 (Win64) OpenSSL/1.1.1c PHP/7.4.2
Transfer-Encoding: chunked

Request Headers (478 B)
Accept: text/html,application/xhtml+xm...plication/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Content-Length: 0
Content-Type: application/x-www-form-urlencoded
Host: localhost
Origin: http://localhost
Referer: http://localhost/fairshare2.html
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; ...) Gecko/20100101 Firefox/71.0
```

So, here, the browser says this is the name of the form. But then, how do we know that it is associated with this text area? That is the question it has to solve. It can use heuristics, since, the text area is inside. But it also has rules, which require that this thing should be Id. And if we put Id back, and reload. We have something simple f1 end, submit, we get some result. And of course, the content length is nonzero.

But let us try the same example with a slight change. We will change this back to name and instead, we will remove this form tab over here. Go back to our system. Reload, that even now it still works. What just happened? Well, the answer turns out to be that if the text areas indicator is missing, because the text area is nested, it is considered a part of this form. Why are there strange rules like this? Part of it is that when the web began, people began by doing strict rules the way we normally do it in a programming language.

But because HTML was written by people, without any programmer training, what happened was that if the browser tries to be too strict, it started irritating people and some browser vendors came up with the idea that you should try to be permissive. Of course, once people try to be permissive, they start putting in heuristic instead of strict rules. And over the years, those heuristics are ballooned into all sorts of strange things.

So, for example, when I first encountered this when I was building this program, I actually had no idea what exactly was going wrong because nobody can keep syntactic details, like this, just at the top of their mind. So, I did what everybody would do. Try to craft a nice query until somebody, somewhere has encountered this before and come up with a solution and get something going. Alternatively, try to find the original document which spells out what the rules are supposed to be.

And both of those methods shall we say, of approaching these kinds of problems is what people eventually have to resort to. Here, for example, are the pages that I found one page aid 'Text area value not getting posted with form' and I did not really notice, for example: this person has both name and ID. And then in further instructions he seems to have something, form, etc., etc. But then his text area was outside of the form.

And then I just started looking what are some things that look like they would work. So, it has something to do with text area inside the form maybe, or something about the form ID attribute. And somebody else finally said something that worked; that is using the form attribute etc. So, I figured it has something to do with form attributes and IDs and so on. And then finally, a look for what the actual specification is.

And it seemed to show what the text area form ID should look like. And I originally had name in there, but then I said, okay, let us try ID and see what happens. And then that is what worked. Of course, before I found that out, I had to go through other things, for example, 'Text area not posted by the browser's problem. And there was some discussion. There. But, this person had a clear mismatch of spellings that could lead to an error.

But, at least it is an indicator that these are the kinds of things that are possibly wrong. Now, let us see if we can go and fix that. Then we then we have our system and let us submit the form and see that everything is fine. Okay: The point of this example is that, Sometimes there are very subtle things, but the indication of failure or at least the reasonably clear, because you can actually check what it is that you are sending from the browser.

So, browser tools, in general, will help get you out of many sorts of irritatingly tricky situations, but it does require you to pay attention to a lot of details and as usual, do intelligent searching on the web to find out what exactly is going on. Some people find themselves fascinated by the kinds of details that we are talking about. And then if you want to find the definitive answer to many of these types of questions, a large number of explanations boils down to knowing the official specification as given by W3C.

So, when you search for W3C form specification, you find things like 'world wide web consortium', 'Forms in HTML documents'. And if you look for forms in HTML documents, between say for example, there is a text area and its attributes. And it says things this, 'the text area element creates a multi line text input control. User agent should use the contents of this element as the initial value and shoot render this text initially'.

This is why; the tabs and spaces between the beginning and the end of the tag are considered the initial value and are faithfully rendered by the browser. So, in a sense, the browser is not wrong and neither is the W3C. Because the browser and W3C arguing about whether or not a space and the text should be treated differently and these kinds of things, you do not actually want to do.

Because, what looks like space in one context, for example, if the background of the text area happens to be a colored thing, then space could be visible, although on a white background it is not. So, it is not that the rules are strange, it is that the idea of what looks good to people is complicated enough that specifying it in great detail is very difficult. So, this is another reason why rules like this exist.

If you look at the matter of the Id and the form name, then you can find these kinds of things which attempt to spell out, what it means for a form to have other elements associated with it. Unfortunately, unless you are a specialist and responsible for actually building web browsers, it is unlikely that you would find it easy to read these kinds of specifications, given that we are not really used to them, which is why the collective wisdom of the search engine and sites like stack over flow turns out to be the usual thing to do.

To understand what happens on the server side, instead of using this FairShare 2 file, which goes to the fairShare 2 CGI that we have written, instead, we will look at a simplified form of this thing. So, we will use a file called fairShare-post. And here, instead of sending it to our usual command, we will send it to a special command for post data, which simply sees what the server sees and prints it out for us.

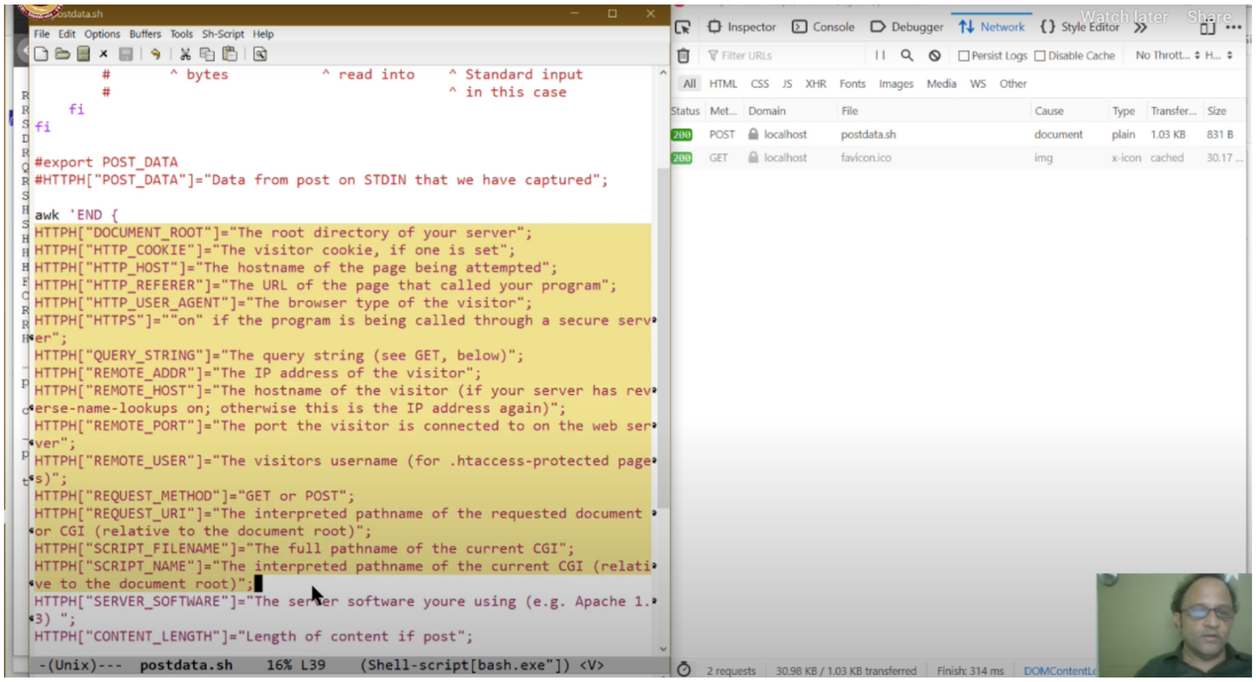
And then, we are in a much better shape to play with it and understand why is really going on. So, let us see first what happens. So, here we have this file fairShare post. And because we loaded it with shift reload, we also got this fav icon, which xampp has actually arranged to show an icon. So, for example, there is the a fav icon that gets returned automatically. Alright. Now, let us see what happens if we write something and submit this a form. Okay.

So, here, instead of our usual application, what we get is a file which is designed to print just the most interesting parts of the entire environment that gets sent to the system. If you want to see the complete environment by the way, there is a program called *show env CGI,* which does the same thing that we are as we did except that it prints a much more detailed and a somewhat prettier report that you have got before.

But, let us come back to our system here. So, what we have is:

- First of the entire request URI something about the remote address.
- What is the script that is executing will be given there; it is post data.sh.
- You will also see *xampp htdocs*, remote user, no query string, no remote host, script name: is it HTTPS.
- What is our server software; This is localhost etc., etc. Content length 20.
- Request method post and 'post data as the browser sends it', And,
- Post data after it is parsed out into original commands.

So, let us see, first of all how this gets done. So, we have post data.sh; what it does is: it has this header which is needed by HTTP. In this case, we add user bin to the path, because if you happen to be on *cygwin*. But, other than that on different operating systems, this may or may not be needed. It checks if the request method is post, it checks if there is a content and if there is, then it reads the content into this variable called post data.



```
        #          ^ bytes              ^ read into      ^ Standard input
        #                                                ^ in this case
    fi
fi

#export POST_DATA
#HTTPH["POST_DATA"]="Data from post on STDIN that we have captured";

awk 'END {
HTTPH["DOCUMENT_ROOT"]="The root directory of your server";
HTTPH["HTTP_COOKIE"]="The visitor cookie, if one is set";
HTTPH["HTTP_HOST"]="The hostname of the page being attempted";
HTTPH["HTTP_REFERER"]="The URL of the page that called your program";
HTTPH["HTTP_USER_AGENT"]="The browser type of the visitor";
HTTPH["HTTPS"]=""on" if the program is being called through a secure serv
er";
HTTPH["QUERY_STRING"]="The query string (see GET, below)";
HTTPH["REMOTE_ADDR"]="The IP address of the visitor";
HTTPH["REMOTE_HOST"]="The hostname of the visitor (if your server has rev
erse-name-lookups on; otherwise this is the IP address again)";
HTTPH["REMOTE_PORT"]="The port the visitor is connected to on the web ser
ver";
HTTPH["REMOTE_USER"]="The visitors username (for .htaccess-protected page
s)";
HTTPH["REQUEST_METHOD"]="GET or POST";
HTTPH["REQUEST_URI"]="The interpreted pathname of the requested document 
or CGI (relative to the document root)";
HTTPH["SCRIPT_FILENAME"]="The full pathname of the current CGI";
HTTPH["SCRIPT_NAME"]="The interpreted pathname of the current CGI (relati
ve to the document root)";
HTTPH["SERVER_SOFTWARE"]="The server software youre using (e.g. Apache 1.
3) ";
HTTPH["CONTENT_LENGTH"]="Length of content if post";
 -(Unix)---  postdata.sh    16% L39    (Shell-script[bash.exe"]) <V>
```

Status	Met...	Domain	File	Cause	Type	Transfer...	Size
200	POST	localhost	postdata.sh	document	plain	1.03 KB	831 B
200	GET	localhost	favicon.ico	img	x-icon	cached	30.17 ...

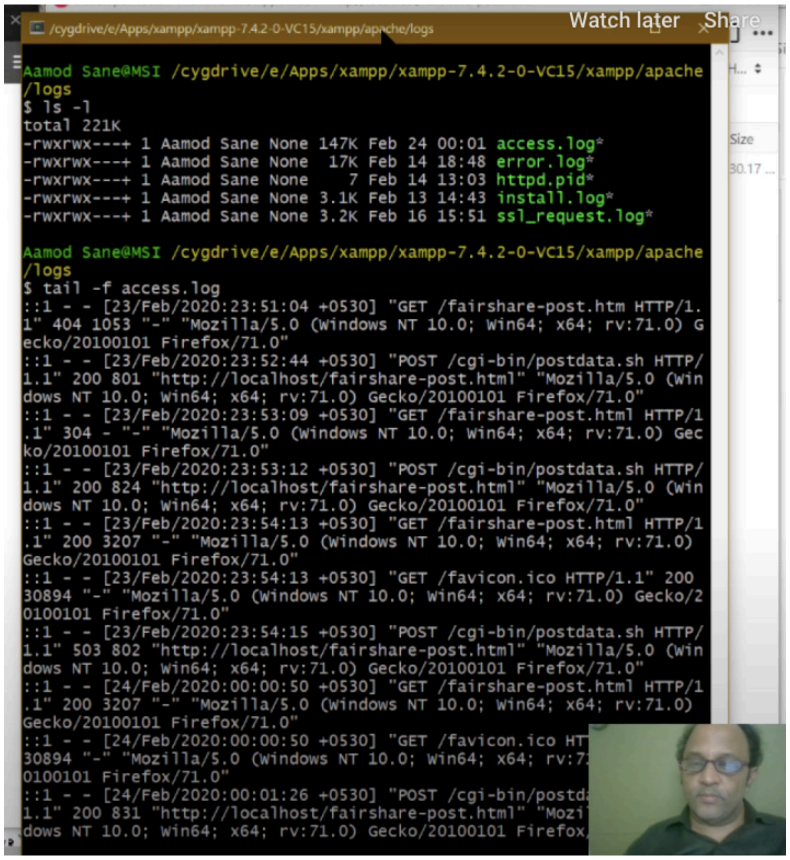
```
2 requests | 30.98 KB / 1.03 KB transferred | Finish: 314 ms | DOMContentL
```

Now, it does something interesting. This is an *awk* script. And I have just created an array of interesting variables that are relevant to HTTP. And you just look through them and print the result. And now you post the data as the browser sees it, you decode it; you want to see what will happen when those commands are passed out, and so on. So, here we are. here we get a basic file, which will allow us to understand what is happening on the server side.

Now, we will start taking a look at the log files and the overall setup of the server. Let us see what we have in our xampp distribution. Here are a bunch of things and some of which are noteworthy make bulltets there is apache, there is CGI bin here and then there is *htdocs*. The default service comes from *htdocs*. For example: if you just say localhost, it goes to *htdocs*.

And it will show you this a few thing where we will find fairShare post, fairShare HTML and fairShare2 HTML. Now, the standard structure of apache and many other servers is if you go to this directory, there is a con file which contains various configuration details. And there is '*logs*' which contains an interesting set of logs. So, first, let us see what those logs are.

Here, we have a very standard distinguishing distinction that is made, there is an access log which shows that a request arrived at all, there is an error log which is returned to if there is an error of some kind and then there is 'pid' which is used to start a limited number of apache processes. And there are a couple other things which record what happened during installation and what happens for some secure requests and so forth.



```
Aamod Sane@MSI /cygdrive/e/Apps/xampp/xampp-7.4.2-0-VC15/xampp/apache
/logs
$ ls -l
total 221K
-rwxrwx---+ 1 Aamod Sane None 147K Feb 24 00:01 access.log*
-rwxrwx---+ 1 Aamod Sane None  17K Feb 14 18:48 error.log*
-rwxrwx---+ 1 Aamod Sane None    7 Feb 14 13:03 httpd.pid*
-rwxrwx---+ 1 Aamod Sane None 3.1K Feb 13 14:43 install.log*
-rwxrwx---+ 1 Aamod Sane None 3.2K Feb 16 15:51 ssl_request.log*

Aamod Sane@MSI /cygdrive/e/Apps/xampp/xampp-7.4.2-0-VC15/xampp/apache
/logs
$ tail -f access.log
::1 - - [23/Feb/2020:23:51:04 +0530] "GET /fairshare-post.htm HTTP/1.
1" 404 1053 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0) G
ecko/20100101 Firefox/71.0"
::1 - - [23/Feb/2020:23:52:44 +0530] "POST /cgi-bin/postdata.sh HTTP/
1.1" 200 801 "http://localhost/fairshare-post.html" "Mozilla/5.0 (Win
dows NT 10.0; Win64; x64; rv:71.0) Gecko/20100101 Firefox/71.0"
::1 - - [23/Feb/2020:23:53:09 +0530] "GET /fairshare-post.html HTTP/1
.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0) Gec
ko/20100101 Firefox/71.0"
::1 - - [23/Feb/2020:23:53:12 +0530] "POST /cgi-bin/postdata.sh HTTP/
1.1" 200 824 "http://localhost/fairshare-post.html" "Mozilla/5.0 (Win
dows NT 10.0; Win64; x64; rv:71.0) Gecko/20100101 Firefox/71.0"
::1 - - [23/Feb/2020:23:54:13 +0530] "GET /fairshare-post.html HTTP/1
.1" 200 3207 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0)
Gecko/20100101 Firefox/71.0"
::1 - - [23/Feb/2020:23:54:13 +0530] "GET /favicon.ico HTTP/1.1" 200
30894 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0) Gecko/2
0100101 Firefox/71.0"
::1 - - [23/Feb/2020:23:54:15 +0530] "POST /cgi-bin/postdata.sh HTTP/
1.1" 503 802 "http://localhost/fairshare-post.html" "Mozilla/5.0 (Win
dows NT 10.0; Win64; x64; rv:71.0) Gecko/20100101 Firefox/71.0"
::1 - - [24/Feb/2020:00:00:50 +0530] "GET /fairshare-post.html HTTP/1
.1" 200 3207 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0)
Gecko/20100101 Firefox/71.0"
::1 - - [24/Feb/2020:00:00:50 +0530] "GET /favicon.ico HT
30894 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:7
0100101 Firefox/71.0"
::1 - - [24/Feb/2020:00:01:26 +0530] "POST /cgi-bin/postd
1.1" 200 831 "http://localhost/fairshare-post.html" "Mozi
dows NT 10.0; Win64; x64; rv:71.0) Gecko/20100101 Firefox
```

The usual way to look at this log is to do *tail - F*. Let us take a look at *access.log*. So, *tail-f acces.log* prints a number of last few requests. And let us create a little blank space so as to understand what happens when there are new requests. And when you submit this form, you will see a new request just arrived. It said that this request is from the localhost. It arrived at what time.

The method was *post* and the URL was *http://localhost/fairShare-post.html.* At this point, no matter what else happens, you know that the request has been received by the server. To verify this, let us go back and add that error to our fairShare script. What we will do is we will change this ‘id’ to name and form and now our browser is actually going to error out, we can reload. We will see first of all the fairShare post and fav icon both of which will be visible on the browser.

Then, submit the form. Okay, notice that because we added the error, our data is missing. Content length is 0. But a new request has arrived. So, this tells us that whatever happened, happened between the browser and the server. And that even on the server side no content was actually received. So, this is the other way we would know that one part of the interaction which is to send the request to the server.

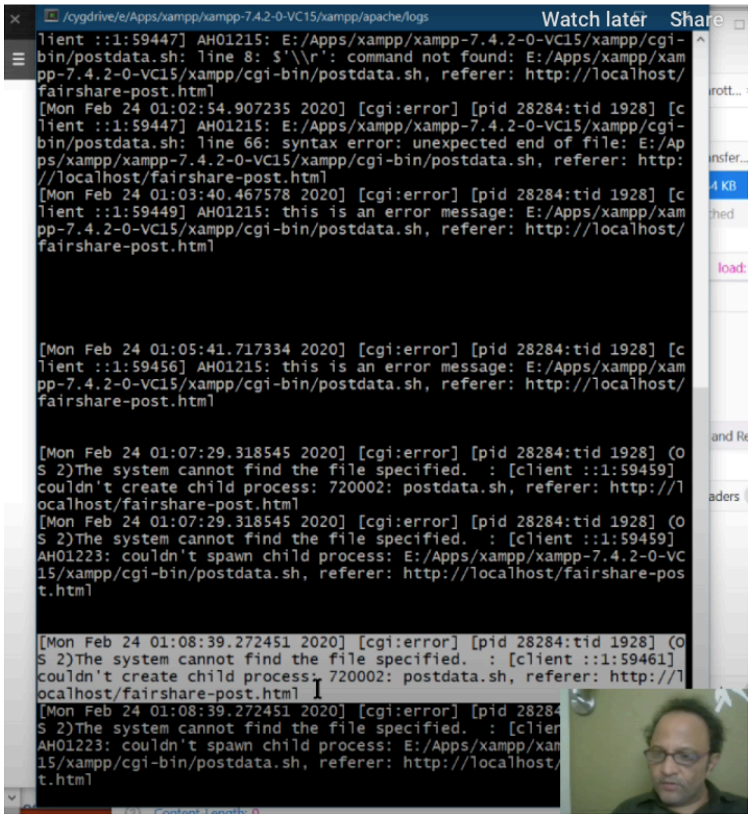
But completed, but when the server looked at what had been sent, it turned out that nothing had been sent. Now the question here, of course, is exactly where the problem happened. But since the rest of the setup all looks good and the content length is officially 0, which means the content must be missing and it should be missing on the browsers. This is the use of the access log. And if you have other things like remote host, etc. enabled in this place of these dashes, you can also see which hosts your request came from.

When you first set up a web server, you will be quite surprised and pleased to see that you are getting requests from all over the world when people come to see the content that you have posted. Okay, so, after this, we will take a look at all the other logs that the system has. To see what the error log does, we have to deliberately create some error messages. And here, the rule is that at least for CGI files, anything that the file writes to standard error becomes an error message.

So, we put an error message like: ‘this is an error massage’. And now we can go and run our program. And we should see an error message on the site. So, I do *tail-F error.log*, there were some earlier errors there. And now if I do this, here, I get a new message with some information: ‘This is an error message’ and the error message was generated by por app, which was referred to by this other the other URL.

This is how we use the error log. So, you could in principal, at least for CGI scripts, use the Apache logs also for error and application specific logging, and then filter it out. But normally, we do not rely so much on apache logs, and for most applications, they will generate their own log. So, that our next step will be to see how a Java program can manage its logs using Java libraries like *log4j*. It is not just that system which is explicitly written error messages are the only reason why an apache error log may have error messages.

Let us create another error situation. So, instead of giving a proper path to bash, suppose that we have made a mistake and have a typo in my code. In that case, you can see the same error as before thing. I have already generated this error once. As you can see, on one side, you get this information that there is error 500, which tells you that something went wrong and you can see the status at POST 500 column with red icon at this point.

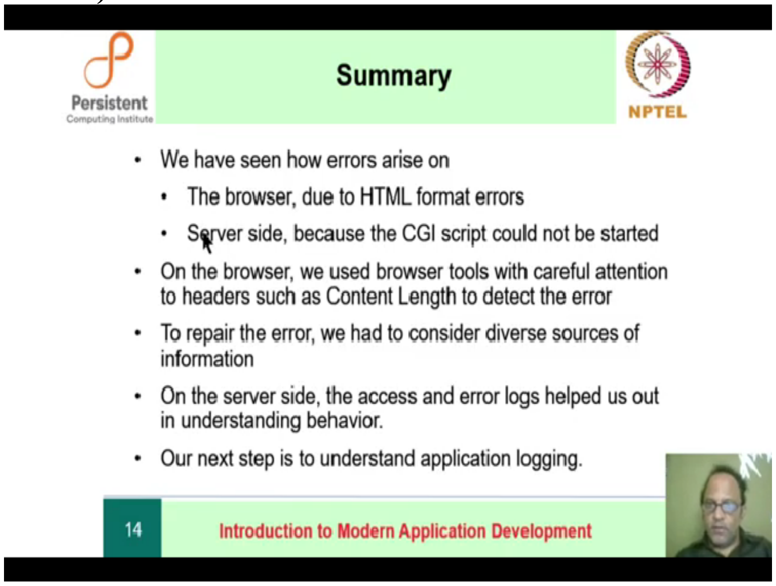


```
lient ::1:59447] AH01215: E:/Apps/xampp/xampp-7.4.2-0-VC15/xampp/cgi-
bin/postdata.sh: line 8: $'\\r': command not found: E:/Apps/xampp/xam
pp-7.4.2-0-VC15/xampp/cgi-bin/postdata.sh, referer: http://localhost/
fairshare-post.html
[Mon Feb 24 01:02:54.907235 2020] [cgi:error] [pid 28284:tid 1928] [c
lient ::1:59447] AH01215: E:/Apps/xampp/xampp-7.4.2-0-VC15/xampp/cgi-
bin/postdata.sh: line 66: syntax error: unexpected end of file: E:/Ap
ps/xampp/xampp-7.4.2-0-VC15/xampp/cgi-bin/postdata.sh, referer: http:
//localhost/fairshare-post.html
[Mon Feb 24 01:03:40.467578 2020] [cgi:error] [pid 28284:tid 1928] [c
lient ::1:59449] AH01215: this is an error message: E:/Apps/xampp/xam
pp-7.4.2-0-VC15/xampp/cgi-bin/postdata.sh, referer: http://localhost/
fairshare-post.html




[Mon Feb 24 01:05:41.717334 2020] [cgi:error] [pid 28284:tid 1928] [c
lient ::1:59456] AH01215: this is an error message: E:/Apps/xampp/xam
pp-7.4.2-0-VC15/xampp/cgi-bin/postdata.sh, referer: http://localhost/
fairshare-post.html


[Mon Feb 24 01:07:29.318545 2020] [cgi:error] [pid 28284:tid 1928] (O
S 2)The system cannot find the file specified.  : [client ::1:59459]
couldn't create child process: 720002: postdata.sh, referer: http://l
ocalhost/fairshare-post.html
[Mon Feb 24 01:07:29.318545 2020] [cgi:error] [pid 28284:tid 1928] (O
S 2)The system cannot find the file specified.  : [client ::1:59459]
AH01223: couldn't spawn child process: E:/Apps/xampp/xampp-7.4.2-0-VC
15/xampp/cgi-bin/postdata.sh, referer: http://localhost/fairshare-pos
t.html


[Mon Feb 24 01:08:39.272451 2020] [cgi:error] [pid 28284:tid 1928] (O
S 2)The system cannot find the file specified.  : [client ::1:59461]
couldn't create child process: 720002: postdata.sh, referer: http://l
ocalhost/fairshare-post.html
[Mon Feb 24 01:08:39.272451 2020] [cgi:error] [pid 2828
S 2)The system cannot find the file specified.  : [clie
AH01223: couldn't spawn child process: E:/Apps/xampp/xa
15/xampp/cgi-bin/postdata.sh, referer: http://localhost/
t.html
```

And here, we have response information plus, on the server side; we have a server error. And the server error looks like this. It says, ‘The system cannot find the file specified’, ‘Could not create a child process' and 'Could not spawn the child process'. So, these things depend on the details of the implementation. And now, let us restore things to work well so you can go in the code and fix it. Remove all this extra stuff and then submit the form again.

This time we go back to our application specific error message. And we can go back still further and fix the issue with the form in the first place; and fix the errors that we made in the code. Submit the form and reload it.

(Refer Slide Time: 49:27)



Summary

- We have seen how errors arise on
  - The browser, due to HTML format errors
  - Server side, because the CGI script could not be started
- On the browser, we used browser tools with careful attention to headers such as Content Length to detect the error
- To repair the error, we had to consider diverse sources of information
- On the server side, the access and error logs helped us out in understanding behavior.
- Our next step is to understand application logging.

14 Introduction to Modern Application Development

We have now seen how errors can arise on the browser and the server side. On the browser, the pause was HTML format errors and fairly subtle interactions between different elements of the tags. And on the server side, in the CGI script or any other method of writing server-side code. On the browser, we were able to use browser tools but you had to pay careful attention to the behavior of headers such as content length, etc. to detect what is happening in the first place.

And we had to know where the issue might lie. To repair the error on the browser side, it was pretty difficult because there is no easy method to understand all the different inconsistencies that can arise in HTML. For historical reasons, finding out these kinds of issues, I guess nobody has ever developed a good HTML compiler, which will do these kinds of compile checking or limiting.

Although, perhaps there are such things around or there should be by now it might make an interesting project for someone to do. In any case, because of the load of so many years of history and evolution there are many strange issues. But, we have enough sources to figure out what where the issue might lie. Plus, it is not just the browser. We also have tools on the server side, which can help us understand exactly where the problem might lie.

The access log tells us that the whether the access has happened at all, and with what parameters, and then the error log tells us what could have possibly gone wrong. Together with this split between the browser and the server and these tools, we have to put together what happens to the interaction between these systems. As you can see, this is quite different from doing straightforward application debugging using debuggers.

Even with GUI programs, such debugging is not simple because the behavior of events and callbacks is difficult. With these distributed systems, things are hard because we have some] different types of systems collaborating with each other, each with tremendous amount of history and reasons why they are the way they are. In our simple setup, both of these systems are on the same machine; easily accessible to us.

In a more complex real life system, there are still other sources of error; sometimes having to do with the DNS, with the intermediate routers and many of the elements that do not show up in our simple case. But, as time goes by, we will take a look at as many of these as is feasible within our limits. To successfully cope, you have to build in your own understanding, careful delineation of the kinds of issues that can arise on browsers, and the kinds of issues that can arise on servers.

As with all bug fixing, and debugging, the overall process is the same, you try to understand each part which is supposed to work and put together part by part, the path from where we started to where we are supposed to end and whether each link within the path is tested. In fact, in our setup, we did not just, you know, simply try to understand our application directly. We in fact, built a mock system, which had the UI of our actual system.

But the back end was completely different. So, this is another interesting thing that the web permits; relatively relative to things like GUI, mocking pieces; mocking means creating a piece of software that takes the place of another, but does not always complete all the original software's functionality is relatively easier in the web, because these are independent systems that we can test independently.

So, as you can see, there is complexity. But there are also saving graces. And the web has a remarkable elegance to it. We can use some of the same tools like shell scripts and all which are useful in simple things that we do on desktop and make them usable on the web, using tools such as CGI scripts. As we go on with more complex servers there are new situations arise. But then new tools are also developed to understand exactly what is going on.

So, all in all, we have to learn these programs together with the tools that are needed to understand them. So, this is our summary for this session. For the next session, or let us say the sub part of this overall session, we will look at application side logging using logging libraries. And we will look at some other tools like curl etc. which can be used for this kind of testing on a more automated basis. Okay, thanks, see you for the next part of this session.