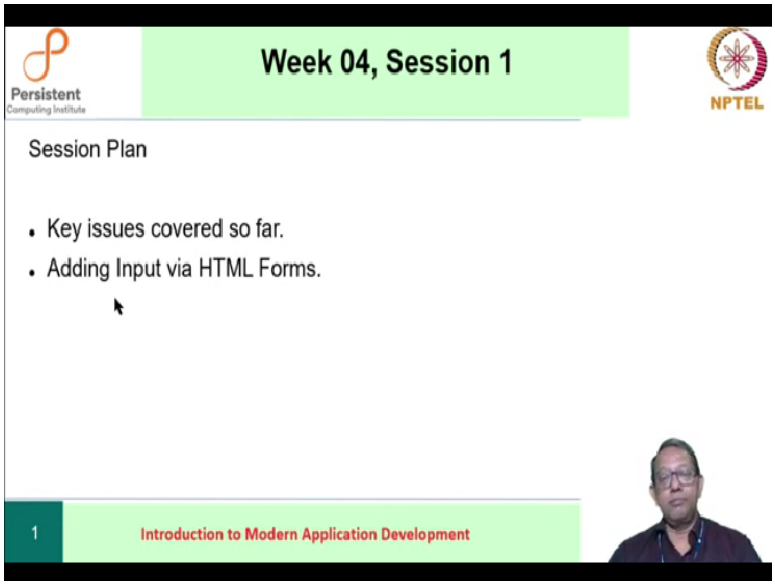
Introduction to Modern Application Development

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

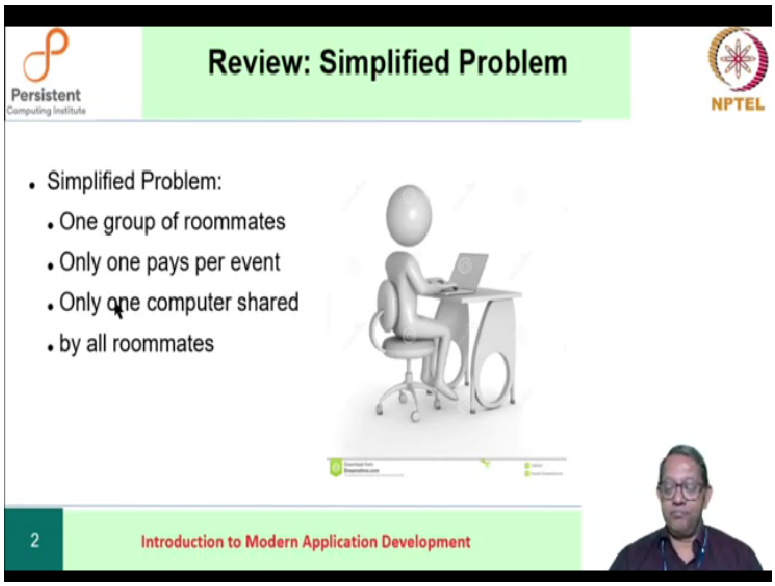
Lecture-13
Introduction to Input in HTML

(Refer Slide Time: 00:13)



Hello everyone, welcome to the 4th week of your course on modern applications development. We will start this week by reviewing some of the things that we have done before and continue with the main theme of this week. That is how do we accept input using HTML.

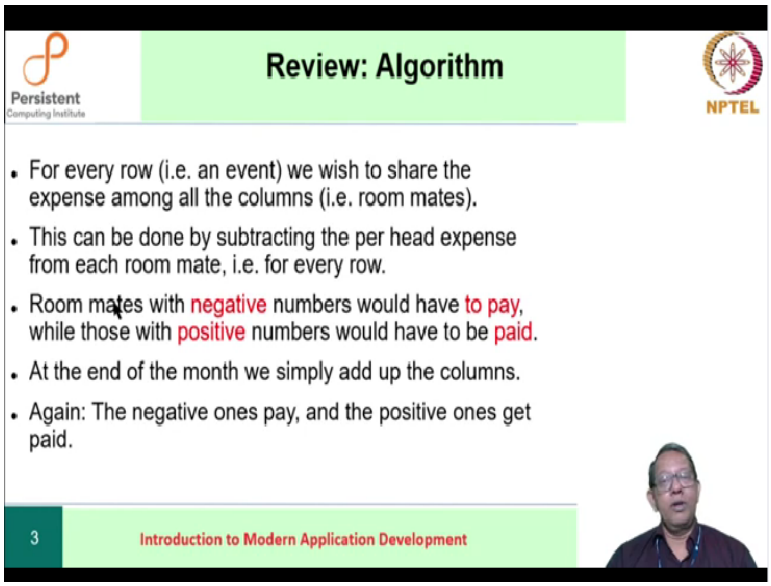
(Refer Slide Time: 00:36)



Recall that we started with a simplified problem. Our problem was a set of roommates who actually shared a number of events like watching movies or having dinners together. For each event, one and only one roommate paid for the entire expenses. However, over time, the group of roommates needed to reconcile their debts or credits. Our problem was to create a system, an application, which could be used by the roommates.

And they could determine the amount of money that they either owed by the group or they had to receive from the group. Eventually, our application is supposed to be on a computer. But right now, to simplify the problem, we are going to assume that there is only one computer that is shared by the roommates.

(Refer Slide Time: 01:48)

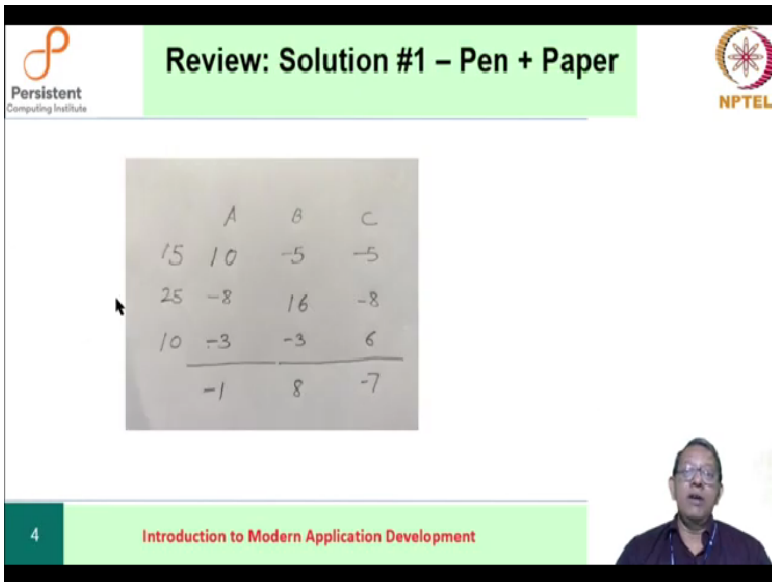


Of course, the most important thing that we need to do is to really work out the algorithm that will be used by all the roommates. The algorithm is independent of the computing system. It could be used anywhere. The idea of the algorithm is to represent events by rows and individual expenses of the roommates by columns. So, any expense by a given roommate for an event would be entered corresponding to the row for the event and the column of the roommate.

Given this data, we would compute the per head share for that event. Per head share is simply the actual expense divided by the number of roommates. This per head share would be subtracted from the money spent by each roommate. If the result of the subtraction is negative, then we interpret that as the money that is owed by that particular roommate to the rest of the group. If the number is positive, we interpret that as the money that is to be received by that particular roommate from the others in the group.

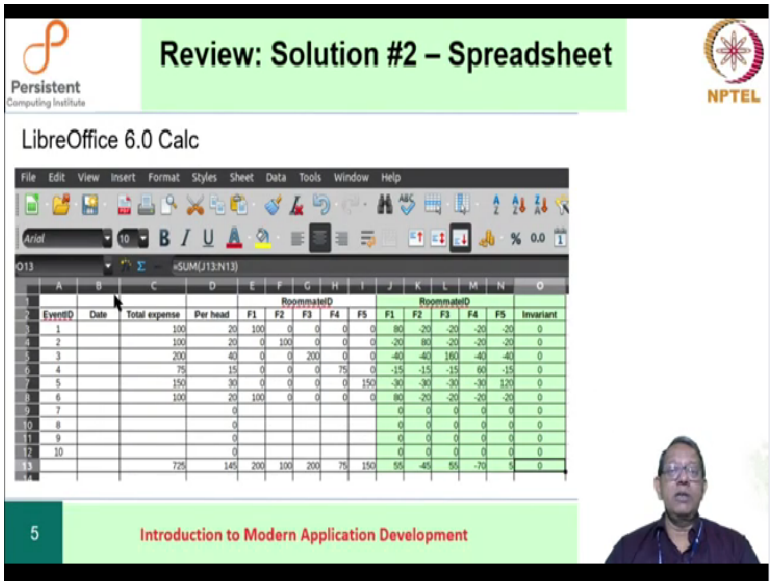
Notice that the sum of the entire per head shares has to be zero. In fact, this is the invariant of the problem. In other words, the sum will always be zero irrespective of the number of roommates or the number of events. We use this idea to argue the correctness of our algorithm. We need to argue about the correctness before we even implement it on a machine because machines do not have intelligence.

(Refer Slide Time: 04:02)



Having created an algorithm, our next step was to actually try it out. The simplest approach was demonstrated by Professor Sane using simply pen and paper. There are of course advantages and limitations of this particular technique of solving the problem. But we will continue with our story. One of the problems with this particular thing is that it is not computer based.

(Refer Slide Time: 04:34)

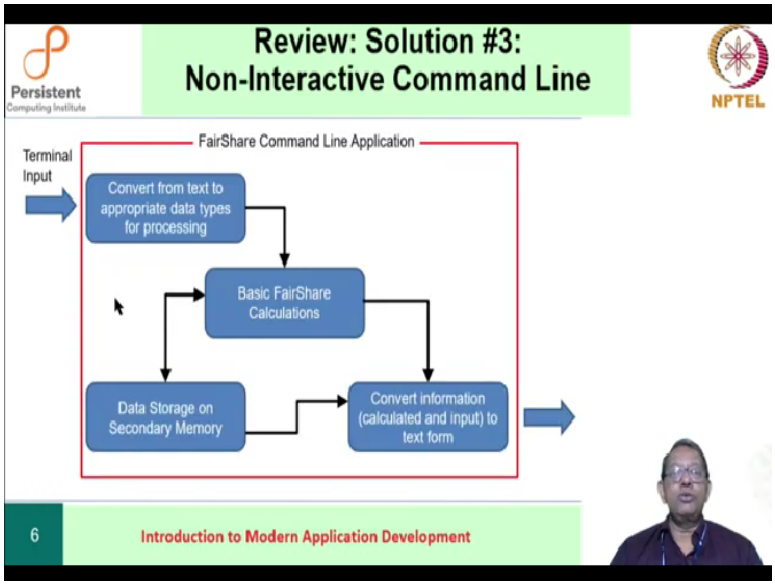


To make it computer based we experimented with a spreadsheet-based solution. The spreadsheet-based solution allowed us to focus on the actual algorithmic aspect, without worrying about the details of how the input is to be given to the computer which was taken care

of by the spreadsheet itself. We did not even have to bother about the output that was also taken care of by the spreadsheet.

This allowed us to focus on the details of the algorithm. And in fact, we explicitly solve how the invariant helped us to ensure that our calculations were correct.

(Refer Slide Time: 05:16)



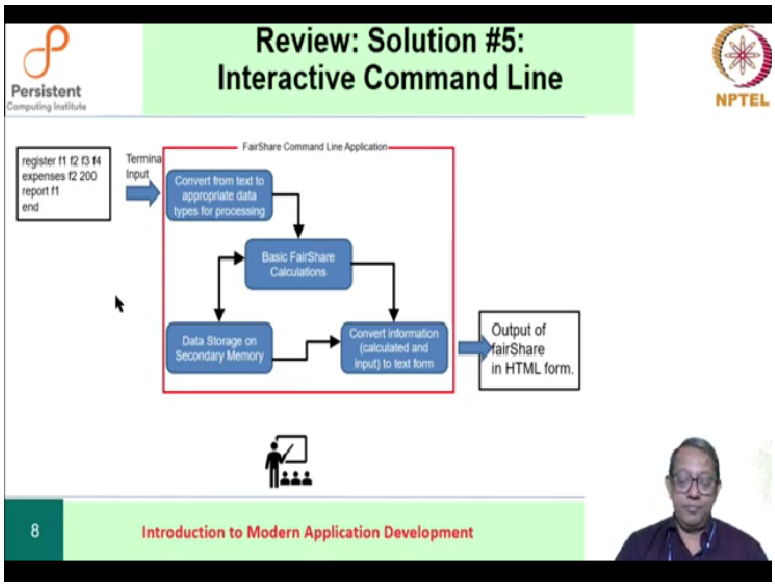
We then proceeded to the next part where we explicitly dealt with terminal input and terminal output. The terminal is perhaps a simplest input-output system, where input is accepted as text and the output displayed as simple text. There is no graphical interface. Nevertheless, the idea of a command line program that works on a terminal allows us to examine some simple concepts like interactivity.

Quick question: Whether the device on which you are watching this lecture is interactive?

We saw that non interactivity is when all information required for operation of the computer of the program key is given before the program starts running. And interactivity is needed when the information that is required for operation of the program is available after the program starts. We demonstrated the non-interactive version of the command line and had you to do the interactive

version of the command line via an assignment. What you see on the screen is a block diagram of the command line application.

(Refer Slide Time: 06:33)



The interactive command line version looks something like this. It was still the command line version except that the entire sequence of commands had to be given after the program started. What you see is a sample input, a sequence of commands where a few friends are registering themselves, there are a few expenses. There are a few reports and a single command called ‘end’ that tells our system that there are no more commands that follow.

Given this, the command line interactive program displayed output as before; it accepted the input, performed the computations and displayed the output as before. We then change that interactive command line version, so that it displayed the output using HTML. The basic markup language of the hypertext system (that is the web). Mainly, it allowed us to do was to become independent of the output device.

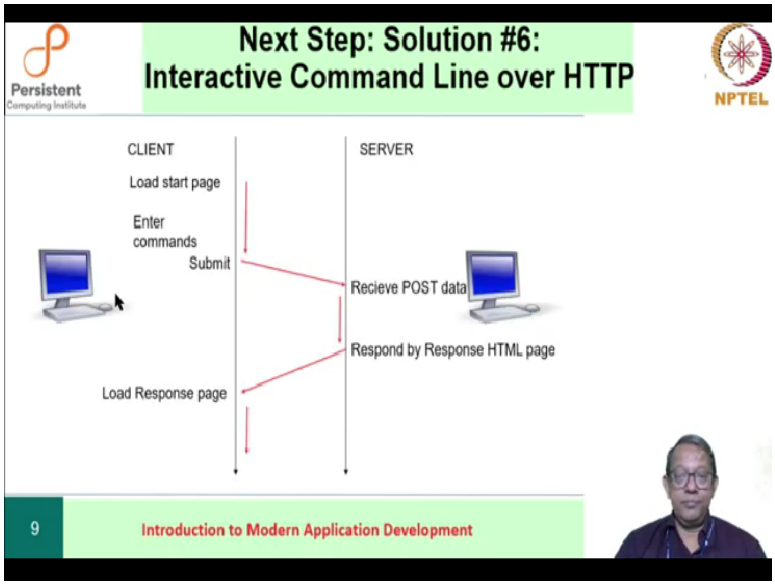
In the sense that, the output produced was simply marking up the details of the display, it did not worry about how those details were implemented on the output device. In fact, the entire markup system could be carried over the network to even a remote machine and could be displayed on a

remote machine. This was not possible on a text display. HTML allowed us to be independent of the output device.

A good appreciation of this problem comes in when you actually change the interactive command line version to a graphical interface, like the one that Professor Sane demonstrated. It brought out the *concept of a model and a view.* A model is when you actually have a computation occurring in the program and view is what is presented out. Separating these two allows you to display the output across the network, something that was not so easily possible before.

This is a good point to pause and review the material that has been covered so far. We will now continue with the problem of accepting input via HTML.

(Refer Slide Time: 09:25)



Note that, the way output has been separated from the computation and the output device, including the possibility that the output devices are actually across the network. We would like to imagine something similar for input, too. We would conceptually think of input as coming in from some kind of a client and our fair share application would be there which we will call **'a server'** from now.

They could be on the same machine, but because they are conceptually separated, now they could be across the network, too. That is why we think of a separate client system and a separate server system. The client is the one that will be used by our roommates. And the fair share application will be visible to client as an application on the client side. Whatever the application does is actually going to happen on the server.

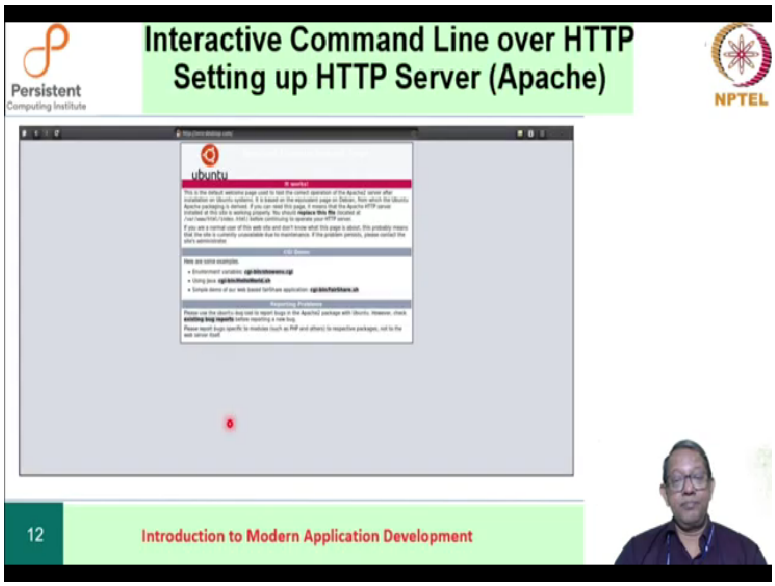
Between the client and the server, there will be some communication, some information that will be transferred. This is similar to the way output was displayed over a network. In the slide that you see, we have black lines which indicate the time on the client side and time on the server side. Conceptually clients, that is our group of roommates, we load some kind of HTML page, which is going to collect input.

Recall that the input is going to be a sequence of commands. Once the commands are entered, our clients are going to submit it to transfer the information from the client side to the server side. The server will then receive the information and process it. It will perform the calculations or computations required by the commands and create a result page. This is identical to our previous exercise where our command application generated output using HTML.

This is what the server will do and that HTML page will now be transferred back to the client. The client system will now load that page and show it to the users. From the point of view of the roommates they will see that they had an application which accepted their input. They submitted the details of that input. They really did not worry about where the application was. But the results were displayed on their own system.

Pictorially, this is what is happening. The information that is submitted by the client to the server is called as **POST** data. Post is a mechanism that HTTP defines for communicating information from the client to the server. The server response is whatever that is produced by the server in response to the information that it has received, but represented in HTML, that response is sent to the client.

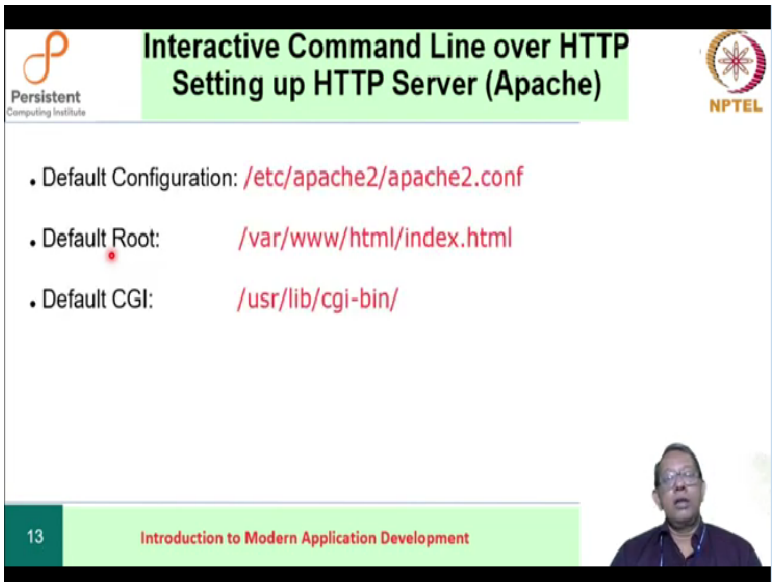
(Refer Slide Time: 13:19)



This means that we need to understand how web servers are set up. We do not need to go into details at this point, we just need to understand it conceptually. A web server is basically another program. On your screen, you see an Ubuntu system which has installed a HTTP server. Fortunately, Ubuntu by default offers us a working configuration and the fact that it has successful working configuration is shown by the page that ubuntu loads when we set up the server.

The screenshot is an entire web browser, and therefore the actual page appears smaller. In subsequent slides we will magnify this part of the page, so that it is clearly visible.

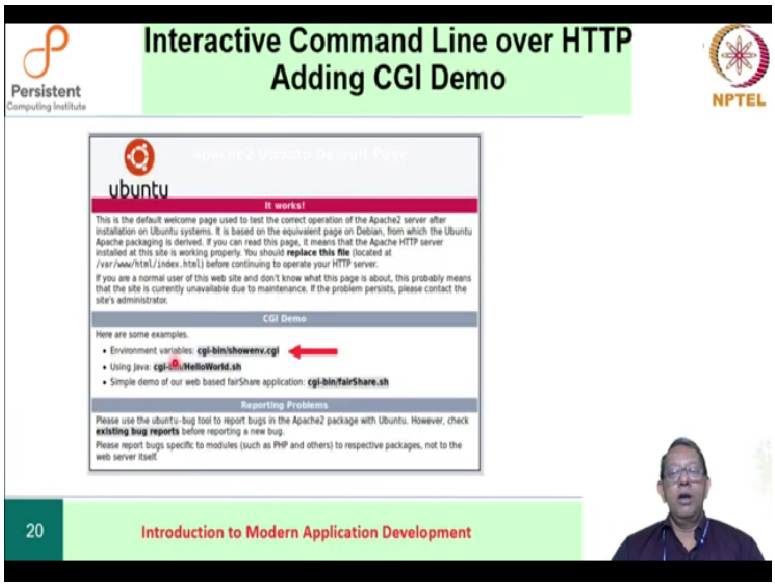
(Refer Slide Time: 14:26)



The default configuration of HTTP on ubuntu is in the directory `/etc/apache2` and the file is called `apache2.conf`. The web server will require a certain directory within which its HTML files will exist. This is called 'document root' of the web server and by default on ubuntu it is `/var/www/html` and the default html file is `index.html`. We will shortly examine the idea of `cgi-bin`.

But for now, we will note that the default directory for CGI common gateway interface is `/usr/lib/cgi-bin` on ubuntu systems. It is possible that default configurations will vary on different systems. The idea of which particular directory will change on different systems, however, the idea that there is some configuration file, there is some directory where there is a default document root and there is some directory where there is a default cgi-bin These ideas will always be there.

(Refer Slide Time: 16:05)



Here is the default ubuntu page. This part is typically standard ubuntu and this part CGI demo is something that we have modified in order to demonstrate how CGI bin works. The idea of CGI bin is to execute programs. A web server typically interacts in the following way:

- It receives some requests from some client,
- It responds to that request, and
- The request could return the `index.html` file to the client.

Well, in response to that, the server will pick up the index dot html file and transfer it across the network to the requesting client. So, a server interaction is a request and a response. That is one interaction of the server. The HTTP system allows a server to respond to a number of request response pairs of interactions. Each of these interactions are actually independent of each other.

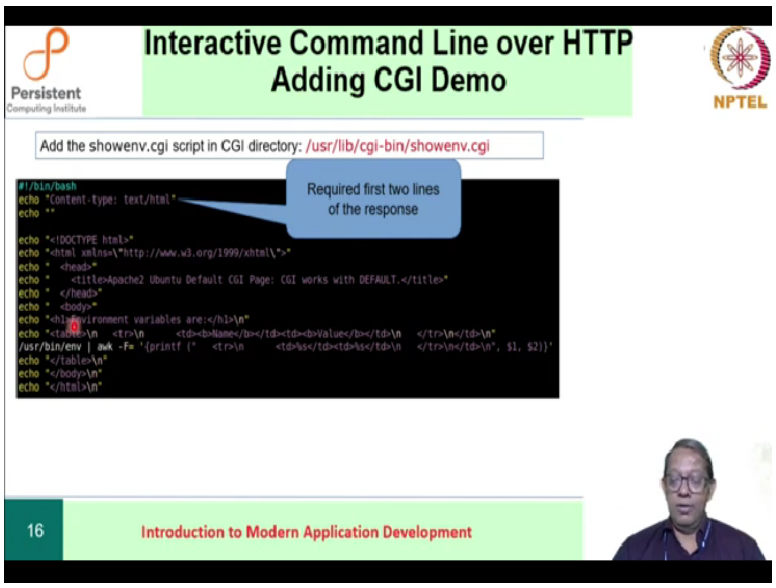
In other words, as far as the server is concerned, the server does not remember what happened in the previous interaction that is a request to response when it is actually dealing with the current interaction of request and response. We say that the server is stateless. If the web server is only responding by picking up files and sending them across the network to the client, then that would be a very uninteresting case. It would be more interesting if the server code, process information and then respond.

In order to process information, we will need to run programs that will process the information that has been received. This ability to run programs and process the received information is what the CGI bin system allows you to do. In the directory of the CGI bin, whatever it is in this particular case of ubuntu, it is `/usr/lib/cgi-bin`, you can place programs that you want the server to use in response to the information that it gets.

We are going to illustrate two programs the show environment program and the Helloworld program. The show environment program is a simple shell script which will display the environment variables of the server to the client. Incidentally, these environment variables are always available to any CGI program. The next program is a small shell script, which demonstrates how Java programs may be actually run by the server.

This would be a good point because we really need that ability to run Java programs all our programs have been written in Java. At some point later, we will also see how our fair share application will look like using a CGI system.

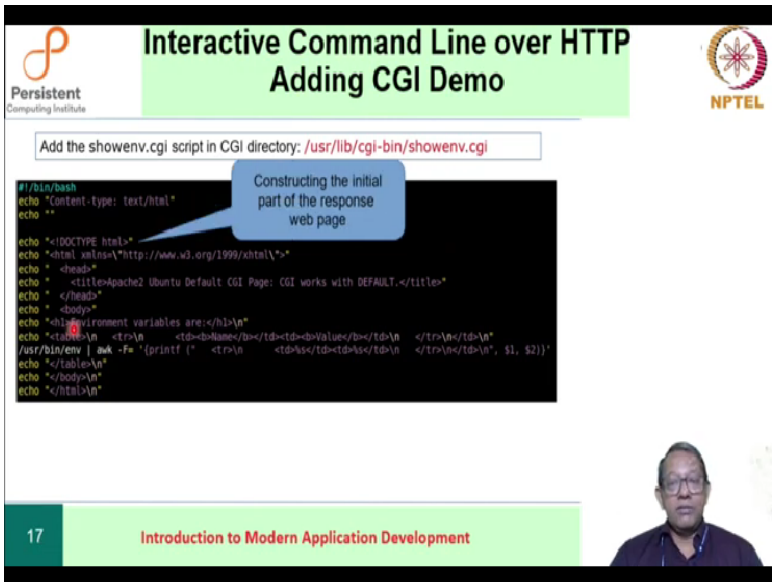
(Refer Slide Time: 19:58)



Here is a shell script, which shows the environment, it is a bash script. We hope that these commands are fairly self evident. The first two lines are required lines out of a server response. Recall that this particular script is going to be run when you click this particular link. So, the

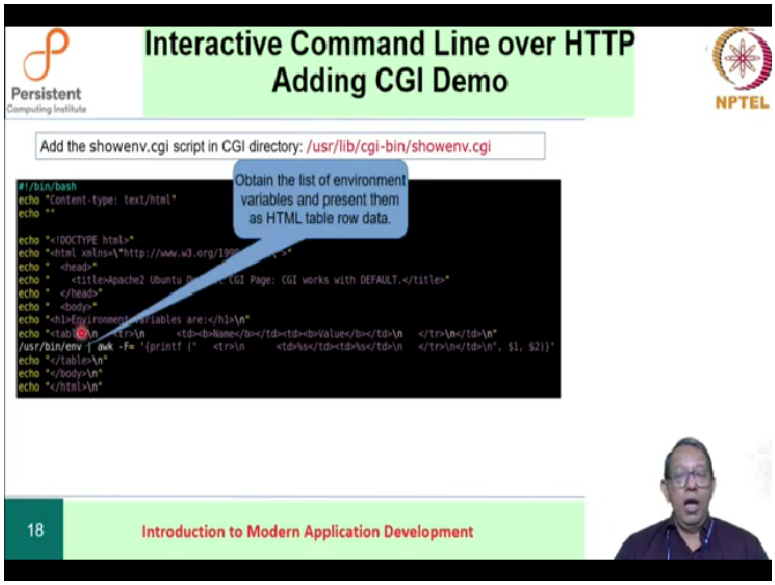
server is going to first print out these two lines that is required by the HTTP protocol and then it will execute these lines. Observe that what this executes is simply creating an HTML page.

(Refer Slide Time: 20:55)



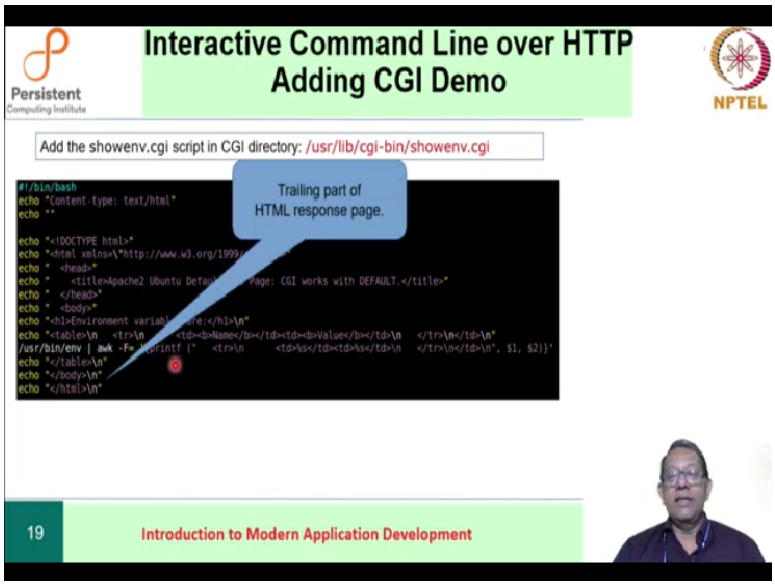
Start of the HTML page is over here, initial part of the response page.

(Refer Slide Time: 21:05)



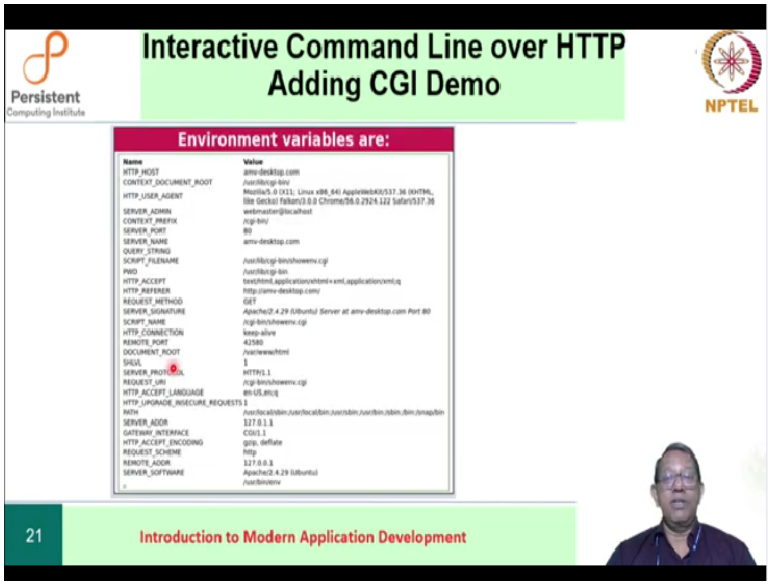
This command actually obtains the environment data and processes them using the **awk** command to create an HTML table of environment variable and it is value.

(Refer Slide Time: 21:21)



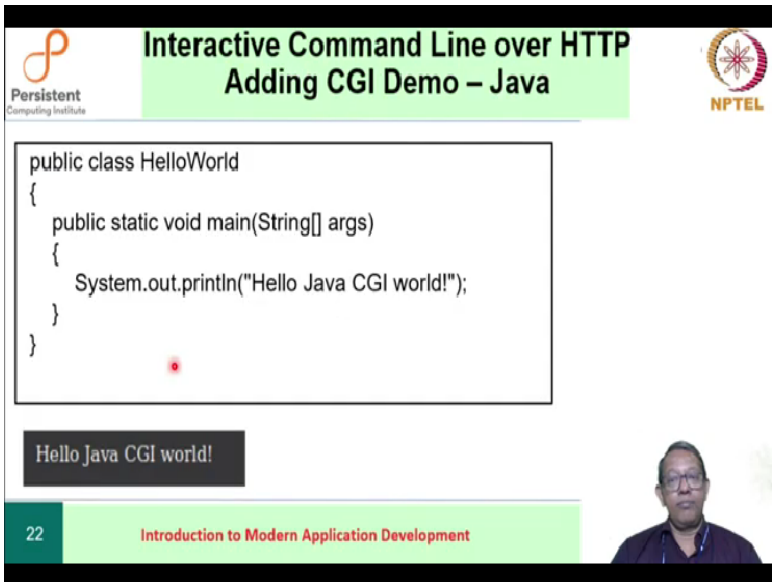
And finally, there is the trailing part of the HTML response. This simple CGI script constructs the actual web page of the response. The webpage when you click on that show environment CGI link, the webpage would look something like this.

(Refer Slide Time: 21:41)



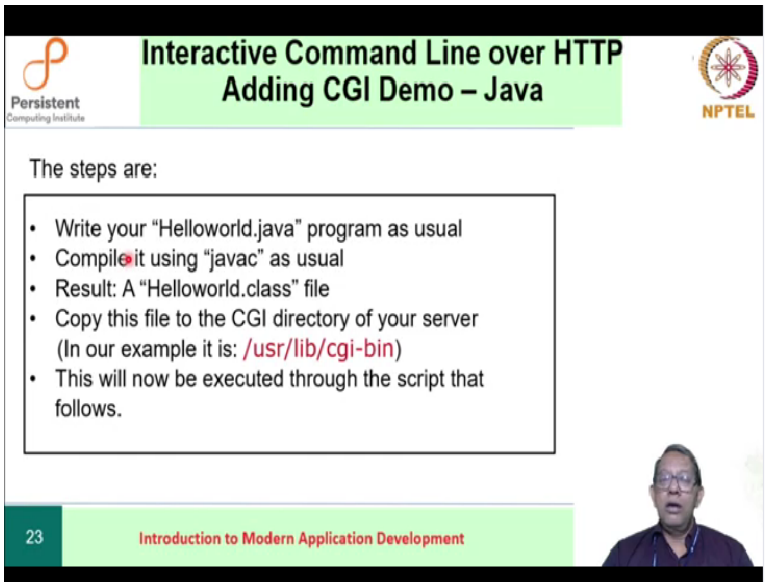
Actually, this web page has been modified using style sheets. That is why it looks nice. There is a nice title bar the names of the environment variables are on the left, the values are on the right and they are neatly presented out. If you remove the style sheets it would simply be a set of outputs; A bland HTML page.

(Refer Slide Time: 22:15)



Let us go ahead and look at how to run Java programs via the CGI bin method, here is a simple Java program or HelloWorld that simply prints Hello Java CGI world. On the browser, this will look like a simple text. Being a Java program, we simply need to compile it to obtain the corresponding class file to be executed. This class file will then be run via a shell script.

(Refer Slide Time: 23:01)



The steps in detail would be something like this:

- You would write your HelloWorld Java program as usual.

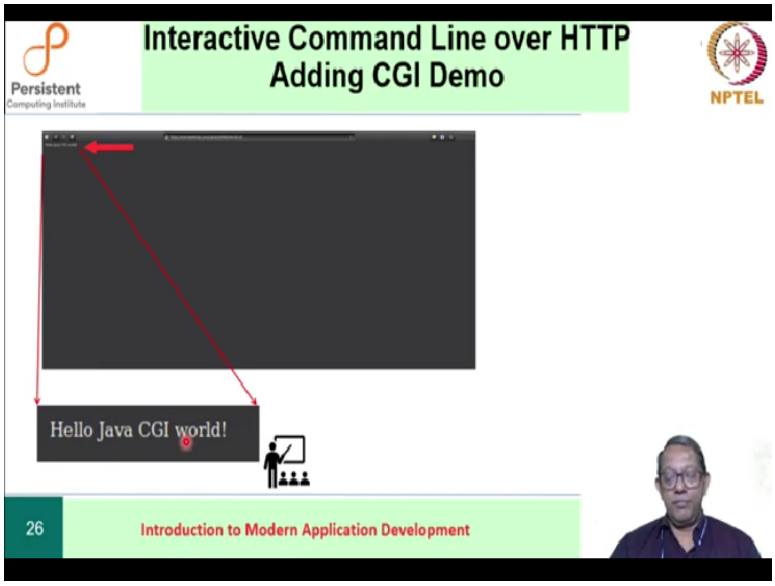
- You will compile it using Javac as usual.
- This will result in a HelloWorld dot class file again as usual.

But, now you will copy this class file into the CGI directory of your server. In our example, our CGI directory is slash users slash lib slash `cgi-bin`. Now this class file is ready for execution. But how do we execute it? We will need a shell script that follows.

Again, the shell script will first because it is the response script. It will first emit these two lines. Java will require class part to be set up so that the directory, where your class files are, is available in the path. You can then execute the HelloWorld class file using your usual method Java HelloWorld. But this class file was already copied into the CGI bin directory as before. The final thing that you now need to do is in your index dot HTML, create this link.

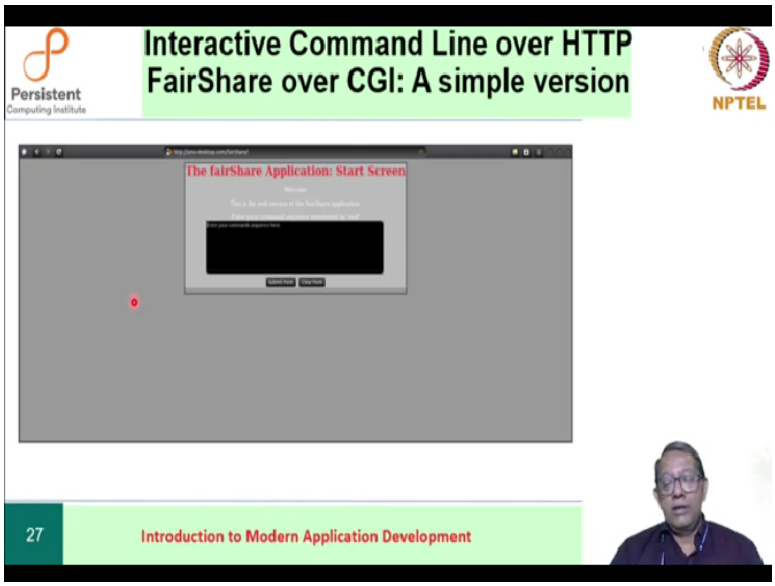
Recall that when you press this particular link, the HelloWorld.sh script that is this script is the one that will be executed. And this script again in turn will execute the actual HelloWorld program.

(Refer Slide Time: 24:52)



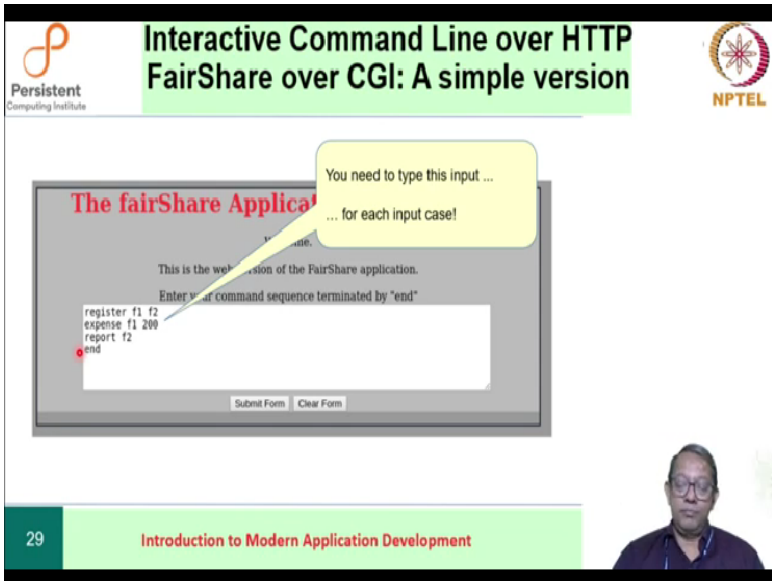
And when you do that, your browser will show you the output HelloWorld Hello Java CGI world like this. But because it is a browser, we have it magnified and show it over here. This is a good point to pause and review the material that you have learned so far.

(Refer Slide Time: 25:16)



Let us now go ahead and try to look at how our fair share application would look over HTTP from the client side that is for the roommates. We should first load an HTML page, which will ask for input. It would probably look something like this, when I look at the entire web browser.

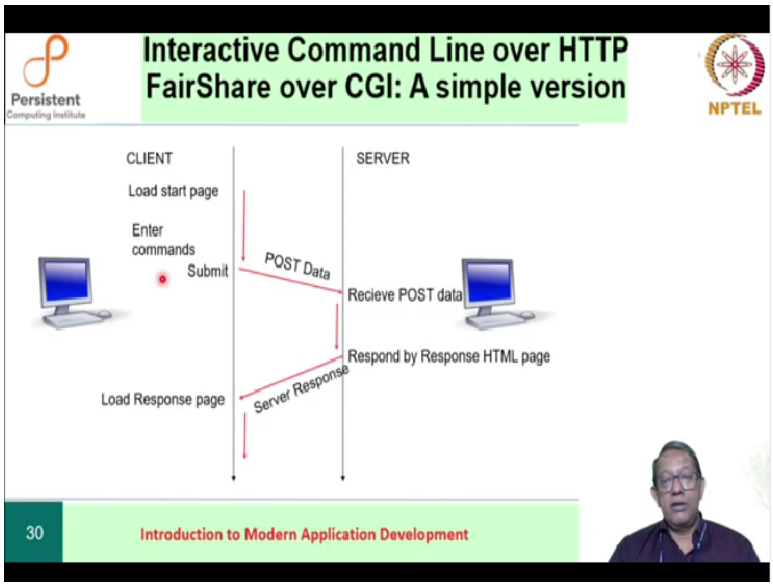
(Refer Slide Time: 25:50)



Let us magnify that and focus only on the part where our application is. It would look something like this. It would say welcome. This is a web version of the fair share application. And it will prompt the roommates to enter the command, sequence terminated by an end. Here are some

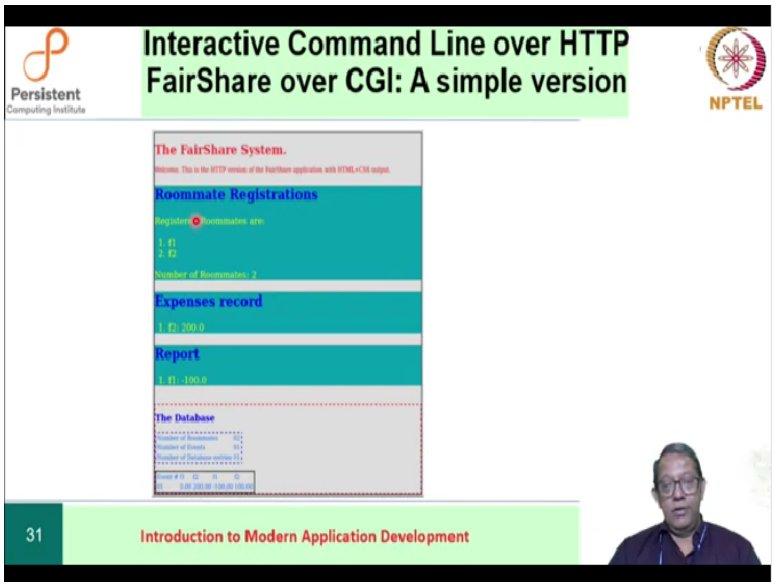
command sequences that our roommates have entered. They would then press the submit button in order to transfer the information that they have put in over to the server.

(Refer Slide Time: 26:35)



This part is what occurs when the user that is our roommate press submit. The information that they have entered is organized as post data and sent over to the server.

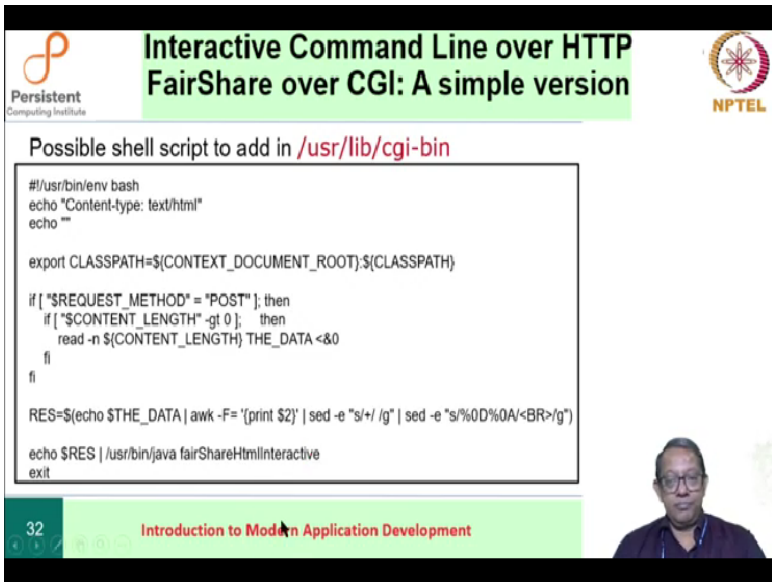
(Refer Slide Time: 26:56)



The server will perform the computation and present out the output in HTML form. Which, presumably, would look something like this. But we already know how to do this because this is

what we did to generate the output using HTML. That was our way to start learning HTML as well as the cascading style sheets idea. This is what we did last week.

(Refer Slide Time: 27:26)

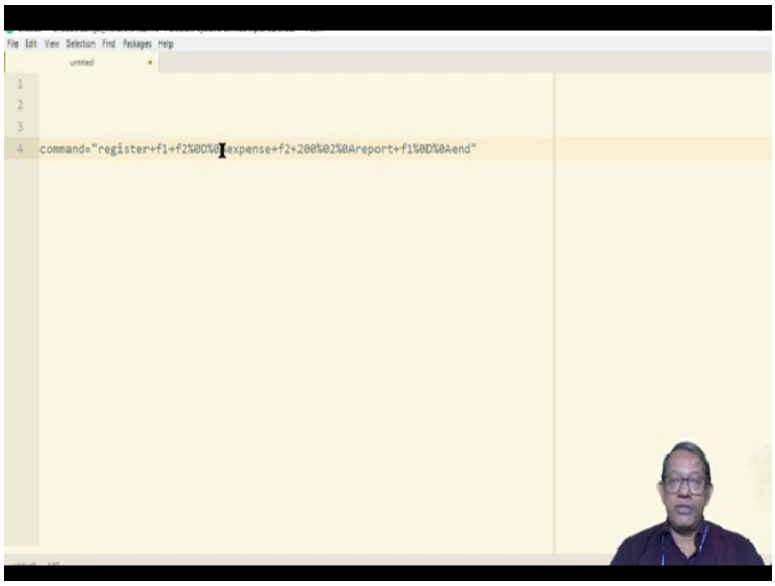


Here is a shell script that you might want to add to CGI bin. What it does is, as usual, printing out the two required lines by the HTTP protocol, setting up the class path to the context document root which is user lib CGI bin and then it processes the input that it has received. If the input method is post, which is what we are going to use and the number of input bytes is more than zero, then it will read those number of bytes into this variable from the standard input.

Next the variable will be printed out and processed using the awk command. Then, it will be processed using the sed command to remove to replace the plus characters by a space character. And then the next sed command will replace '%0D%0A' by the HTML code BR a break. This process output is then given as standard input to our Java fair share HTML interactive version. So, the sequence of events is something like this.

Our roommates press 'submit', and post and send the data. The HTTP system defines the post format. The format looks something like this.

(Refer Slide Time: 29:22)

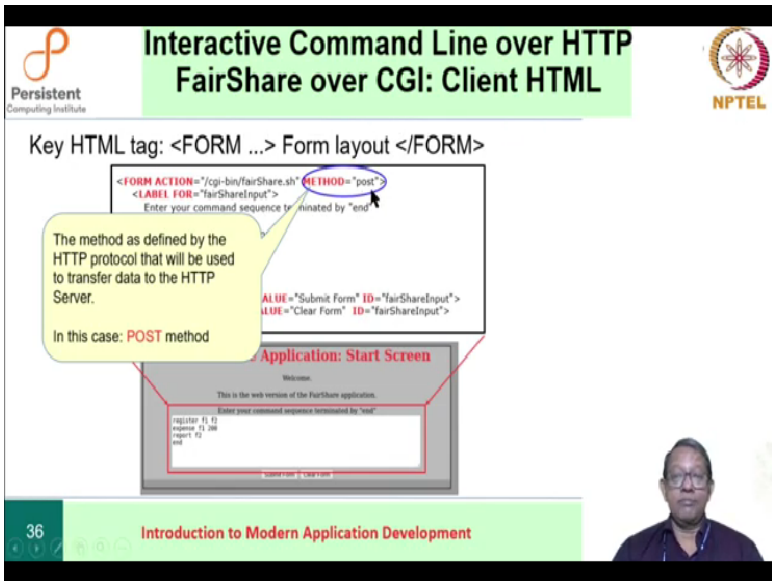


Our roommates had typed registers space f 1 space f 2 new line expense space f 2 200, new line and so on so forth. What the browser does is every space is replaced by a plus character. And every new line is replaced by percent zero D percent zero A which incidentally are ASCII values for carriage return and new line. This allows the browser to send a single stream of text as data as input data to the server.

That is why, we need the script which will take in the particular post data that has been sent by the client and process it such that the original lines of commands are recovered. Those commands that are recovered are then given to our usual program. Recall that our program the interactive HTML version of the program took input a set of lines of commands and generated an HTML output.

So, the way we have set up the system is when the users press submits, the post data comes to the server where our actual fair share application exists, which application? The one that we had written last week; an HTML producing application. We introduce a shell script in-between, which will convert the post data into the input that our Java application requires. That is all.

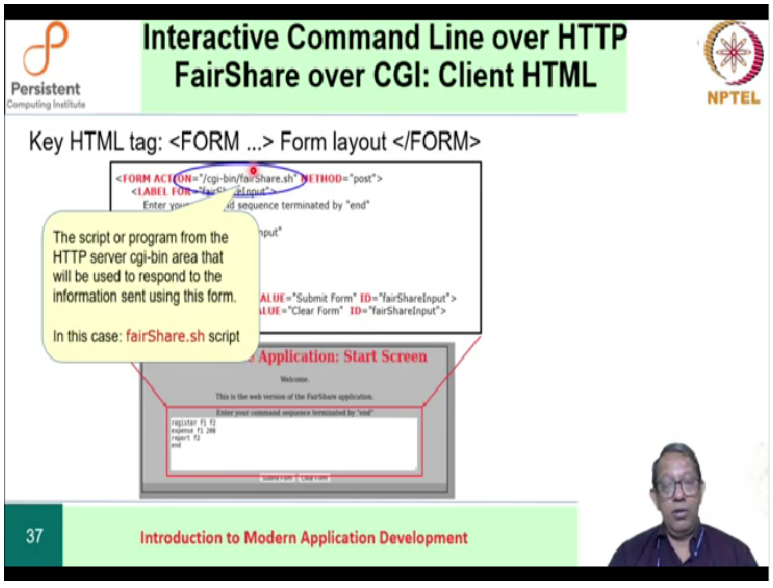
(Refer Slide Time: 31:16)



Let us go back to the way our client-side application looks like. The HTML page that has been presented to our roommates looks like this. And we saw that it contains a text box, a text area where the roommates type in the commands. This part of the code actually looks like this. We use the HTML element form to actually accept input and send it over to the server.

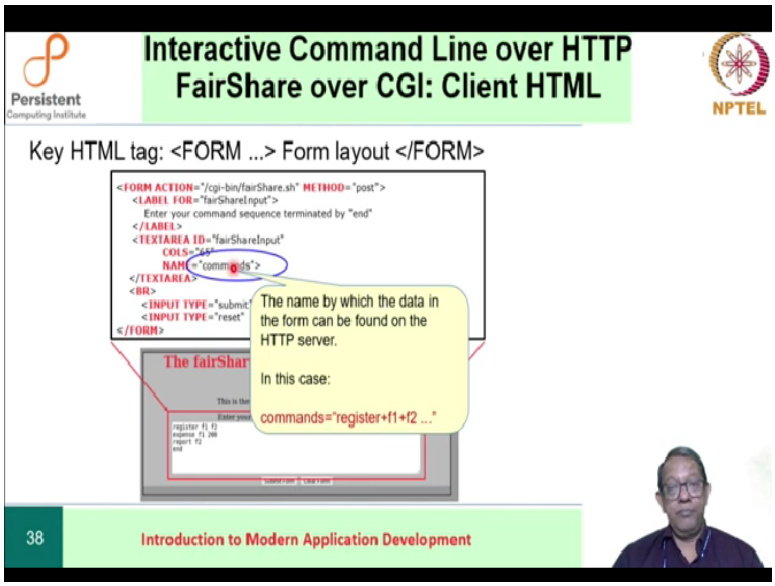
How do we do that? The form contains a number of different components. In the first place, it defines the method by which it should communicate the information to the server. In our case, we are going to use the post method that is why we specify post over here.

(Refer Slide Time: 32:27)



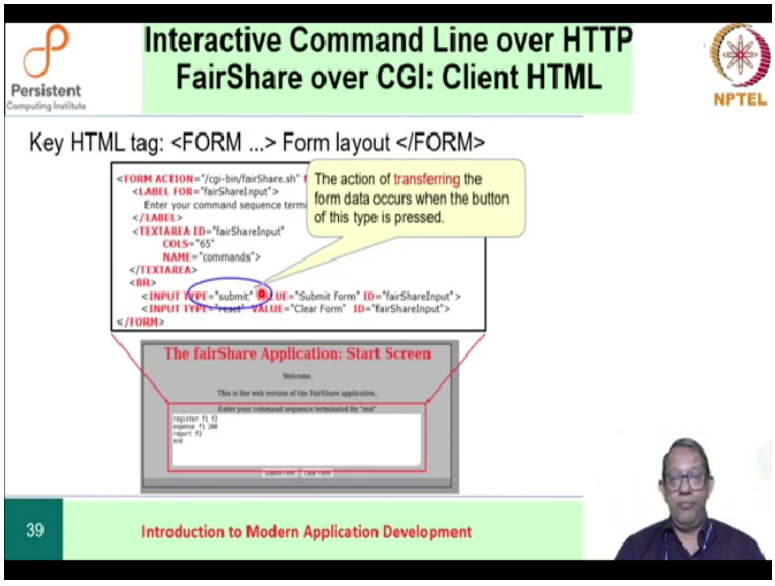
Then it tells the server what is the particular, then we need to tell the server which script on the server must be executed when data is received using post. Note that this information is available if this information is specified on the client web page.

(Refer Slide Time: 33:08)



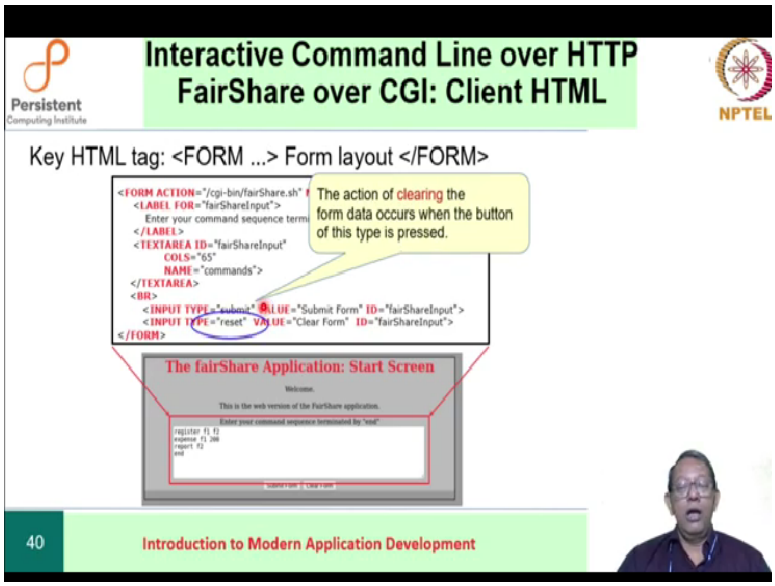
Whatever the information that has been collected by the client HTML page will be given a name 'commands', the name that you need is written out in the form over here. This is the name by which our server will recognize the valid command.

(Refer Slide Time: 33:33)



The action of transferring the form data, this is the form data to the server will be performed by this input type. This input type creates a button with the label 'Submit' on it.

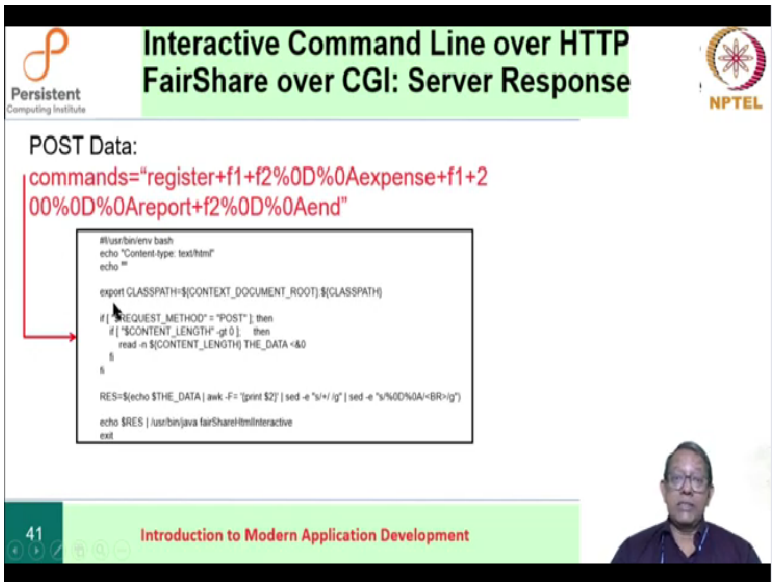
(Refer Slide Time: 33:52)



This input type will do the action of clearing of the form data. There is no sending, whatever (the form data) is entered is erased and you get a blank area again. Notice this part, this name 'Commands' is exactly the one that we had used over here. This entire line is what the server will receive. If the server receives this that our shell script must first use the **awk** command to split this entire string into two parts before the equal to sign and after the equal to sign.

The second part, after the equal to sign, is the one that we need to process. Whatever there is after equal sign is then processed by the sed command which will first look at the plus characters and replace them by spaces. And then the next sed command which will take up these characters "%0D%0A" and replace them by new lines. This is how we recover the lines of commands that we need.

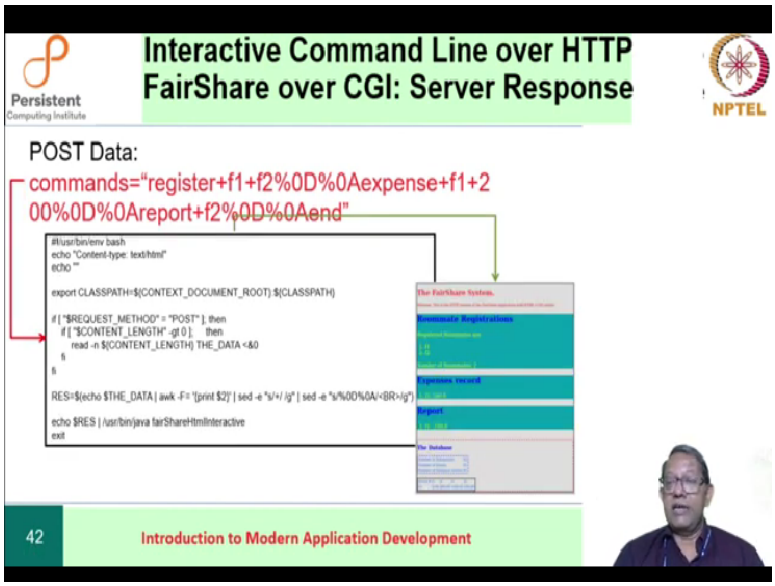
(Refer Slide Time: 35:21)



This is what is illustrated in this part of the slides. The post data arrives as commands equal to this entire string. This script that we saw before processes that the **awk** will use the equal to or the separator character and print the second part that is after the equal to the sed command will replace the plus signs by spaces. Then it will replace the "%0D%0A" by the HTML code break. This will result in individual commands on individual lines, something that our fair share application expects.

How would we know it expects? because we wrote it last week. The set of commands is then given as input to our interactive HTML producing application.

(Refer Slide Time: 36:24)



And that produces the HTML page that we see. Let us recollect back the picture very clearly. Our roommates load the start page, enter the commands and press submit button. Then the browser takes over whatever has been written by the commands that have been given by the roommates is converted to post data sent to the server. The server receives the post data processes that to recover the command sequence that is understood by our HTML Java program.

Having received the commands, the Java program produces an HTML that is sent out by the server back to the client as the result of this submit. Friends, we now have our first web application. We have separated the input output part from the processing part. The processing part, the model can potentially be remote. And the views that are presented on the client side, the initial view and the response view; they are completely on the client side.

It is this separation that has allowed us to create web applications. Why is it a web application? Because the actual data is transferred across the web and there is a server that is processing that information and responding back. Note that although from the client side, that is; our roommates, see a very graphical application on the server side, the processing is almost you always using command lines. We will stop at this point. Thank you and see you in the next session.