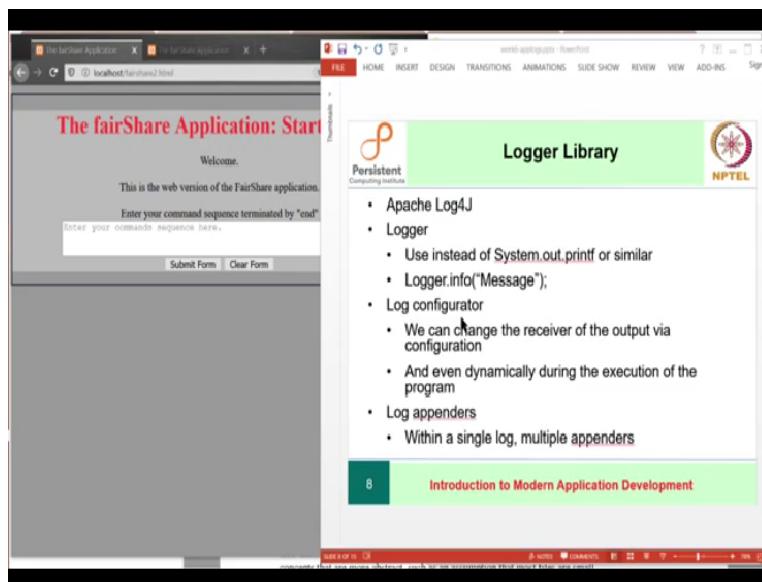


**Introduction to Modern Application Development**  
**Prof. Aamod Sane**  
**FLAME University and Persistent Computing Institute**  
**Abhijat Vichare**  
**Persistent Computing Institute**  
**Madhavan Mukund**  
**Chennai Mathematical Institute**

**Lecture-20**  
**Introduction to Input in HTML**

(Refer Slide Time: 00:12)

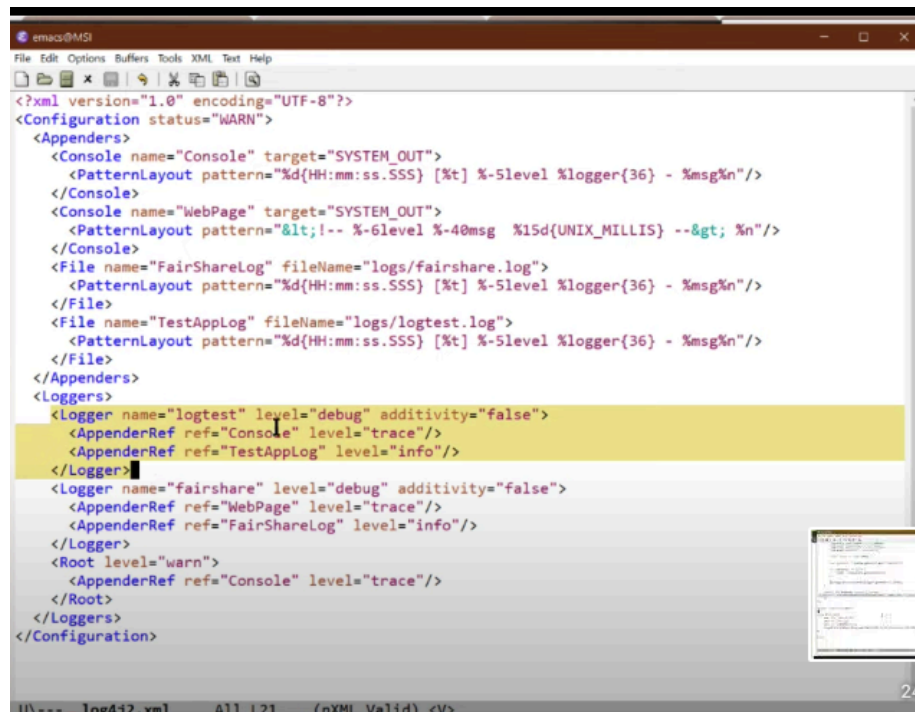


Welcome to Part 2 of session one of week 6. In the first part, we looked at the details of how loggers work with a small example for a log test. Now we are going to see logs in use in our application. What we have learned last time is that the log4j system provides a logger, log configurator and log appenders.

(Video Starts: 00:38)

We have seen some examples of this in the code for log test. So, let us just refresh our memory a little bit. There is a log manager here. Then you start making these calls *info*, *debug*, *trace*, etc. which is write these messages with a little tagline, which is then useful to separate out different

parts of the system and to duplicate it onto the console and the file and so on and so forth. We also saw that in the config file here, we had logged test.



```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
    <Console name="WebPage" target="SYSTEM_OUT">
      <PatternLayout pattern="%lt;!-- %-6level %-40msg %15d{UNIX_MILLIS} --> %n"/>
    </Console>
    <File name="FairShareLog" fileName="logs/fairshare.log">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </File>
    <File name="TestAppLog" fileName="logs/logtest.log">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </File>
  </Appenders>
  <Loggers>
    <Logger name="logtest" level="debug" additivity="false">
      <AppenderRef ref="Console" level="trace"/>
      <AppenderRef ref="TestAppLog" level="info"/>
    </Logger>
    <Logger name="FairShare" level="debug" additivity="false">
      <AppenderRef ref="WebPage" level="trace"/>
      <AppenderRef ref="FairShareLog" level="info"/>
    </Logger>
    <Root level="warn">
      <AppenderRef ref="Console" level="trace"/>
    </Root>
  </Loggers>
</Configuration>
```

And we have the logs going to the console and to the test app log. And the test app log that we see is in here we decided to put it in logs called it fair share dot log right now there is nothing. So, the place where the file name should be created was given here and accordingly the log file got created. Okay, now we also saw the overall arrangement of the classes in the logger and we saw that you get these types of lines with info warn, and error, fatal etc.

The screenshot shows a PowerPoint slide titled "Logger example: levels". The slide is part of a presentation by the Persistent Computing Institute and NPTEL. It lists the following log levels and their meanings:

- Every line in the log file has a level
- INFO Application working normally
- WARN ok for now, trouble in future
- ERROR Has problems: act now
- FATAL Giving up
- DEBUG Detailed information
- TRACE Much more detailed

Below the list, there is a log output snippet:

```
01:09:59.961 [main] INFO logtest - log something t
01:09:59.961 [main] INFO logtest - This is an info n
01:09:59.976 [main] WARN logtest - This is a warn
01:09:59.976 [main] ERROR logtest - This is an err
01:09:59.976 [main] FATAL logtest - This is a fatal n
```

On the right side of the slide, there is a code snippet for a Java class named `logtest` that demonstrates the use of the `log4j` logging framework:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.Level;
import org.apache.logging.log4j.core.config.Configurator;
import java.util.Properties;

public class logtest {
    public static void main(String[] args) {
        Logger logger = LogManager.getLogger("logtest");

        logger.info("log something that will be useful.");

        logger.debug("This is a debug message");
        logger.trace("This is a first trace message");
        logger.info("This is an info message");

        Configurator.setLevel(logger.getName(), Level.TRACE);

        logger.trace("This is a second trace message");
        logger.warn("This is a warn message");
        logger.error("This is an error message");
        logger.fatal("This is a fatal message");
    }
}
```

The slide number 10 is visible in the bottom left corner, and the title "Introduction to Modern Application Development" is at the bottom.

The exact shape of the message is decided by this factor layout. It has a somewhat cryptic structure but it is well explained in the `log4j` documentation. So, let us just briefly go over it – this is obvious this is the date part, this is the level and this is which logger are we talking about, and what the message is. Now there we see that we have this tag, here we have the date, the level, the name of the logger and the actual message, the time that we see here okay.

And this is the same file that we are seeing over there on the left. And now we are going to take a look at application logging. So, to understand how application functions, we have already seen how we use the browser tools and the logger. But beyond this, there are things we can do. For example, let us take a look at this thing. Below we show our familiar form, let us do our standard application, “*expense 200 report f2 end*”.

The screenshot shows the "The fairShare Application: Start Screen". It has a grey background with a red title. The text on the screen reads:

Welcome.

This is the web version of the FairShare application.

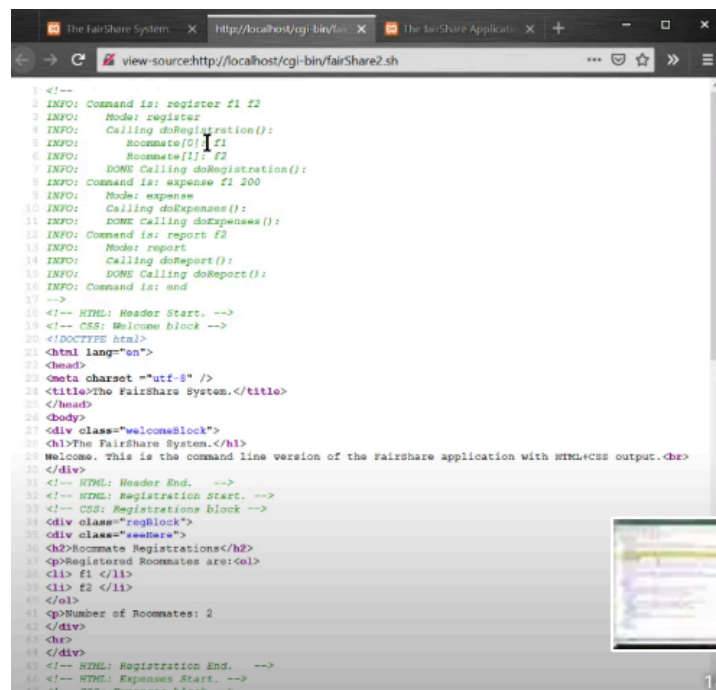
Enter your command sequence terminated by "end"

Below the text, there is a text input field containing the command sequence:

```
register f1 f2
expense |
```

At the bottom of the form, there are two buttons: "Submit Form" and "Clear Form".

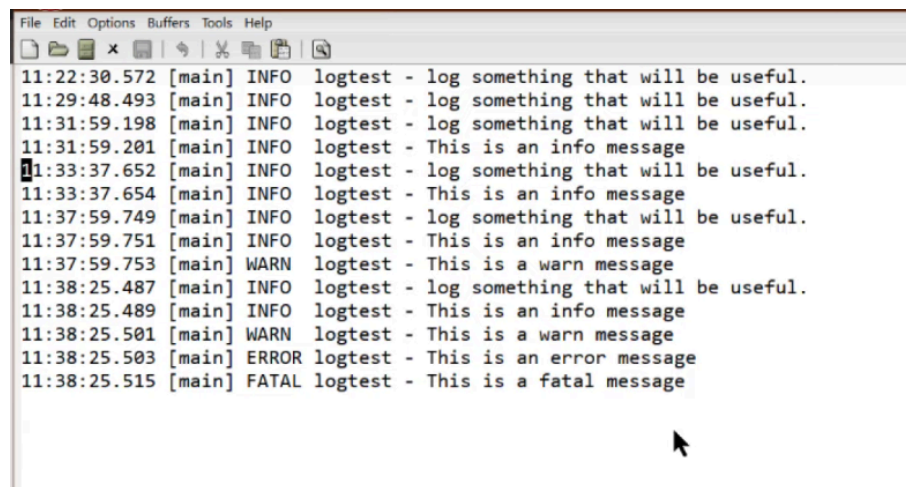
And now, when I submit the form, I get this. But if you look at the actual HTML for this by doing *Ctrl+U*, you will find that at the top hidden as an HTML comment is this kind of debug information. This tells us that the first command was analyzed, then it did registration. Then it looked at the second command, looked at the third command and finally the whole thing and how was this done. Let us take a look at the code for this thing.



So, we have fair share HTML interactive.java. And here for example, is where the code begins. So, in the function called do setup, we have this line, which prints out the comment part of the HTML, which is this tag here. This is HTML comment. And then throughout the body of the system, we have lines like this, for example system or print line, *infoPrefix*, etc. Now *infoPrefix* is defined as shown below.

```
static String errorPrefix = "Error";
static String infoPrefix = "INFO: ";
static String errorMessages;
static String outputMessages;
```

So, we have an error prefix and we have an info prefix. This serves the same purpose as the lines we have seen in the log file, namely these types of lines over here, which we also saw show up in the tests that we have done. Let us see, do we have log test logs here – yes, we do! So, here we have info warn, fatal etc. And these kinds of tags can also of course be hand-created by doing “infoPrefix” like this.

A screenshot of a text editor window with a menu bar (File, Edit, Options, Buffers, Tools, Help) and a toolbar. The main text area displays a series of log entries. Each entry consists of a timestamp, a thread identifier, a log level, and a message. The log levels include INFO, WARN, ERROR, and FATAL. The messages are variations of "log something that will be useful." and "This is a [level] message".

```
11:22:30.572 [main] INFO logtest - log something that will be useful.
11:29:48.493 [main] INFO logtest - log something that will be useful.
11:31:59.198 [main] INFO logtest - log something that will be useful.
11:31:59.201 [main] INFO logtest - This is an info message
11:33:37.652 [main] INFO logtest - log something that will be useful.
11:33:37.654 [main] INFO logtest - This is an info message
11:37:59.749 [main] INFO logtest - log something that will be useful.
11:37:59.751 [main] INFO logtest - This is an info message
11:37:59.753 [main] WARN logtest - This is a warn message
11:38:25.487 [main] INFO logtest - log something that will be useful.
11:38:25.489 [main] INFO logtest - This is an info message
11:38:25.501 [main] WARN logtest - This is a warn message
11:38:25.503 [main] ERROR logtest - This is an error message
11:38:25.515 [main] FATAL logtest - This is a fatal message
```

So, if you keep going down, if you search for print line, you will see that you have “Command is” you have this more and then there is some error analysis and within each of these things, so, this is the actual HTML output, but even before HTML output, if we look for said these kinds of things, “do registration” for instance, (do registration), we find this kind of material, which is doing the output. And from time to time, we also have the code that outputs this kind of thing.

So, let us say “Done calling to registration”. So, this is where we do registration, system out print line infoPrefix and errorPrefix, after catching the exception if something went wrong in the “do registration” and announced that you are doing the column. So, this can be done, of course, but we would like to do it using logs and we will see some of the advantages of doing these kinds of things using logs.

For this is a simple way of adding sort of directly useful type of information in the web page itself. And once we have begun printing these types of comments, we have to make sure that at the other end, we also close the comment so that when you look at the page, the actual original

page, the comments do not show up anywhere and here. Although the debug information is present, it does not interfere with what the user sees, which is good for us to understand what is going on.

At the same time as having the exact same output that the user sees. Okay, now let us take a look at the other version. We have fair share 3, and in this we are going to do a few different things. So, first of all, in fair share 3, we are going to have this information here log level, which is going to allow us to change the logging level that is visible. And we have also made some changes to the fair share HTML interactive 3, by using log lines.

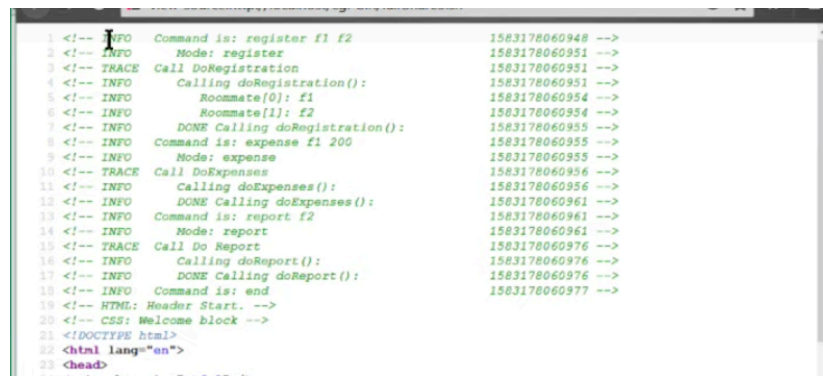
So, for example, instead of having system out print line here we could have something like logger trace. But this has a problem. The issue is that the logger is designed to work on a per message basis. And these loggers at least do not have a sense of beginning and end. And we would have to deal with this on a message by message basis. So, we are going to do something different. We will look at the results in a minute. But here is the change we made.

So, where the earliest system might have said something like, let us see. So here we have system out print line, followed by, in this case system out print line info prefix something, we have logger dot info "Command is". Now it is saying the same thing as this, but saying it in a way that gives us manifold set of advantages as we will see in a while. So here we have logger info, mode equals something here we have system out print line info prefix.

So, throughout this file we have gone and wherever the information intended for debugging was included, we have replaced this with logger. So, here we have this part, but the output itself, so, system out print line is still used for the output itself. So, this kind of thing, the actual HTML content is still collected in a string and printed out as output messages in this case.

This part we have left as it is and the rest of the what is sometimes called meta information, we are moved to the logger. Let us see what happens when we do this do this sort of logging. So, let

us submit the form. Let us start this thing here, submit the form and this time where the result is quite different, message by message – every single interesting message appears as a comment.



```
1 <!-- INFO Command is: register f1 f2 1583178060948 -->
2 <!-- INFO Mode: register 1583178060951 -->
3 <!-- TRACE Call doRegistration() 1583178060951 -->
4 <!-- INFO Calling doRegistration(): 1583178060951 -->
5 <!-- INFO Roommate[0]: f1 1583178060954 -->
6 <!-- INFO Roommate[1]: f2 1583178060954 -->
7 <!-- INFO DONE Calling doRegistration(): 1583178060955 -->
8 <!-- INFO Command is: expense f1 200 1583178060955 -->
9 <!-- INFO Mode: expense 1583178060955 -->
10 <!-- TRACE Call doExpenses 1583178060956 -->
11 <!-- INFO Calling doExpenses(): 1583178060956 -->
12 <!-- INFO DONE Calling doExpenses(): 1583178060961 -->
13 <!-- INFO Command is: report f2 1583178060961 -->
14 <!-- INFO Mode: report 1583178060961 -->
15 <!-- TRACE Call Do Report 1583178060976 -->
16 <!-- INFO Calling doReport(): 1583178060976 -->
17 <!-- INFO DONE Calling doReport(): 1583178060976 -->
18 <!-- INFO Command is: end 1583178060977 -->
19 <!-- HTML: Header Start. -->
20 <!-- CSS: Welcome block -->
21 <!DOCTYPE html>
22 <html lang="en">
23 <head>
```

And here we have the log level. Here we have the message as it is printed, and here is the time in milliseconds. So, this helps us understand how much time was spent in each of the stages. As you can see, a trace messages are visible here, as well as info messages. And this was achieved by these kinds of things. So, logger dot info, versus logger.trace, so, logger.trace. So, we have 2 kinds of messages. Trace, as we say, is detailed debugging.

And info is not a matter of debugging. So, for example, there is no reason to distinguish normally between beginning the call versus being inside the call and doing the actual work. But as an example, this is what we have done over here. One interesting thing is that we can control, as we said, fair shares logging without touching the java program. So, for example here, if I change the log level to info, save this file, go back here and submit the form again.

Now, by open up this new thing, as you can see, there are no trace calls in here, as there are over here, so without changing the actual Java program, we were able to change the output that we see in the place where we want debugging, but that is not all. We can also show that the log output has gone to the other place where we care about which is “tail -F logs fairshare.log”. Here, we have this output, just to confirm let us do this all over again.

We will go back. I will bring this over here. Let us submit. And as you can see this log also appeared over here. On this side, again, we have this info only. Let us go back and change this to

trace. So, we will change that to trace. Go back here and now watch for the difference between these 2. So, we submitted the form, but notice that there are no trace statements here. Whereas there are trace statements in this debugging:

```
1 <!-- INFO Command is: register f1 f2 1583178252736 -->
2 <!-- INFO Mode: register 1583178252738 -->
3 <!-- TRACE Call DoRegistration 1583178252738 -->
4 <!-- INFO Calling doRegistration(): 1583178252739 -->
5 <!-- INFO Roommate[0]: f1 1583178252743 -->
6 <!-- INFO Roommate[1]: f2 1583178252743 -->
7 <!-- INFO DONE Calling doRegistration(): 1583178252743 -->
8 <!-- INFO Command is: expense f1 200 1583178252744 -->
9 <!-- INFO Mode: expense 1583178252744 -->
10 <!-- TRACE Call DoExpenses 1583178252744 -->
11 <!-- INFO Calling doExpenses(): 1583178252744 -->
12 <!-- INFO DONE Calling doExpenses(): 1583178252748 -->
13 <!-- INFO Command is: report f2 1583178252749 -->
14 <!-- INFO Mode: report 1583178252749 -->
15 <!-- TRACE Call Do Report 1583178252749 -->
16 <!-- INFO Calling doReport(): 1583178252749 -->
17 <!-- INFO DONE Calling doReport(): 1583178252750 -->
18 <!-- INFO Command is: end 1583178252751 -->
19 <!-- HTML: Header Start. -->
20 <!-- CSS: Welcome block -->
21 <!DOCTYPE html>
22 <html lang="en">
23 <head>
24 <meta charset="utf-8" />
```

So, the logger infrastructure allowed us to do a bunch of things, it allowed us to send this information in 2 places. It allowed us to take this message and format it quite differently once for the web page. And once for the log file, the message remained the same, but the decoration around the message was changed. It also allowed us to have 2 different levels. So that in one place where we want detail, it is available, but in the other place, it is not.

And there is more that can happen. So, let us do one more change and then we will see something else. So, this time, I am going to inject an error by putting a command which the system does not know. Now, let us see what happens. So, here the system decided to ignore the command and continue to make sense of what it could, but to warn us it printed this message, instead error fair share, and error shows up here, and of course also here.

So, error is considered important enough that it is more important than trace and more important than info, and so it will show up in both places. Just to verify suppose we change the log level to “warn” which is a higher level than info then we go back we maintain our error over here. And let us clear up things and submit the form. This time the only message that we see is the error message. And same thing here, the only thing that shows up is the error message.



And the rest of the processing happens as normal. So, we were able to make all these changes without changing the code. Let us see logger dot error. So, this is the unknown mode message that we see. And there is I think one other place where we should make this change, so we can check this error prefix thing and ideally this should also be the last logger, “logger.error” and we don’t need the “*errorPrefix*” string now. But this time of course we will have to recompile the thing, which is fine, no big deal.

We can recompile it. This catches another error for which we do not have a model yet. But it is good that we can find this and change it to standard style. Let us go back and change warn to trace again.

After doing this, we will observe that we will be getting error over there. This is coming up. And so, we are back to the configuration we had started with. Now, let us see how this is done. So, all this is done entirely in the log4j config file. And this is the idea.

So, this is our logger for fair share. And it contains two things. It has an appender called web page, which works at the level of trace. And it has another appender called fair share log, which works at the level of info. The fair share log appender is this it is a file, which is log or fair share log and its pattern looks like this. Whereas the appender web page is in fact, a console appender. Because we know that for CGI, all we have to do is put out this information on the standard output and it automatically goes to the web page.

For this time, compare this pattern with the pattern for the log here, this thing and **lt** semicolon is an HTML escape sequence, which generates the less than character. Similarly, **gt** generates the greater than character, these 2 are responsible for the tag <--- and this tag over here --->, let us arrange it so that we can see both things at the same time. So, we have this **lt** tag and we have -->, which is we have here. So, this thing does one level of interpretation and therefore the less than and greater than have to be encoded in this time.

Then we have left justified so this is this is a level tag with 6 characters left justified. So, we have info and trace arranged like this. And then the message so the message is left justified in a 40-

character field, so that we get these things nicely aligned. And then we have milliseconds Unix style that we add here. And so, we keep the message the same, but get the result in 2 different places with 2 different levels and 2 different formats.

And so, both of these requirements of getting the log to the user directly to the developer directly into the web page, as well as information in the log is achieved. Now let us see how we were able to change the log level without changing the Java program and here the idea is pretty straightforward. We have an environment variable that we have set in this shell script, which drives the Java program.

And then the Java program, what it does is this thing, have a *setLogLevel*. So, I created a function called **setLogLevel()**, which does the following. It sets up a hash map which contains these strings and maps them to the corresponding level. This thing is needed because the apache logger acts in by using the enum called level rather than strings. So, we make it happy. And this is somewhat unfortunate because this unnecessary enum means that we need these kinds of conversions.

```
static void setLogLevel() {  
    var logLevels = new HashMap<String,Level>();  
  
    logLevels.put("ALL",Level.ALL);  
    logLevels.put("TRACE",Level.TRACE);  
    logLevels.put("DEBUG",Level.DEBUG);  
    logLevels.put("INFO",Level.INFO);  
    logLevels.put("WARN",Level.WARN);  
    logLevels.put("ERROR",Level.ERROR);  
    logLevels.put("FATAL",Level.FATAL);  
    logLevels.put("OFF",Level.OFF);  
  
    Level level = Level.INFO;  
  
    var asklevel = System.getenv().get("LOGLEVEL");  
  
    if (asklevel != null) {  
        level = logLevels.get(asklevel);  
    }  
  
    Configurator.setLevel(logger.getName(), level);  
}
```

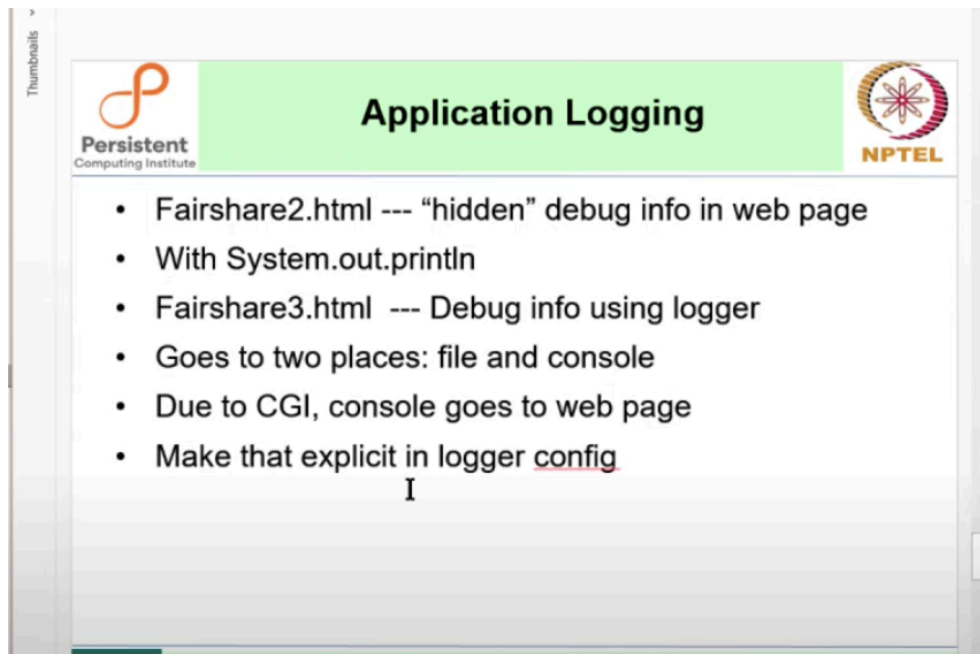
But in any case, we have this map. So, what we now do is we first set the level to be info by default, we check if the environment has a variable called log level. And if the variable exists and it is not null, then we actually get the level that it wants and set the configurator to this level, this is, by the way, the same as what we were doing for log test, where we set configurator log level directly by way of showing you how to make the change dynamically.

By the way, this is not the best way that you could use to set up the log level because we actually had to go and change this file to make the change. A much nicer way is to either provide a query term in the URL or use a cookie or you can also check which IP address the system is coming from. And if the IP address belongs to the development organization, then you can directly include the debug information every single time.

All these 3 things are possible and we will see some of them later on. But for now, for this simple program, I just use this same method, which not only creates a new variable and a new file called fair share 3 plus fair share 3 interactive that uses the log to show what can be done and the reason I do not use a query string or a cookie is that we have not done that yet. And we do not know how to do it.

Plus setting it using the query string means that some amount of parsing is needed, which for a shell script and the CGI setup that we are using, it is a little troublesome. So, when we go over to servlets, and how some of these kinds of parsers etc. directly available, then we will start using these more sophisticated methods. So, at this point, this is what we have seen, we have seen that there are loggers, there are 2 different loggers that are available.

And one of them can be directed to the webpage whereas the other to a file. We have also seen that in CGI console goes to web page so we can say that, you know writing to the web page is the same as writing to the console and we can change the meaning of these appenders to get the effect we want. Plus, we can get the message format we want as well. Alright, here is what we are done, we saw that fair share to hide hidden debug info, it was generated with system dot print line.



Thumbnails

Persistent Computing Institute

## Application Logging

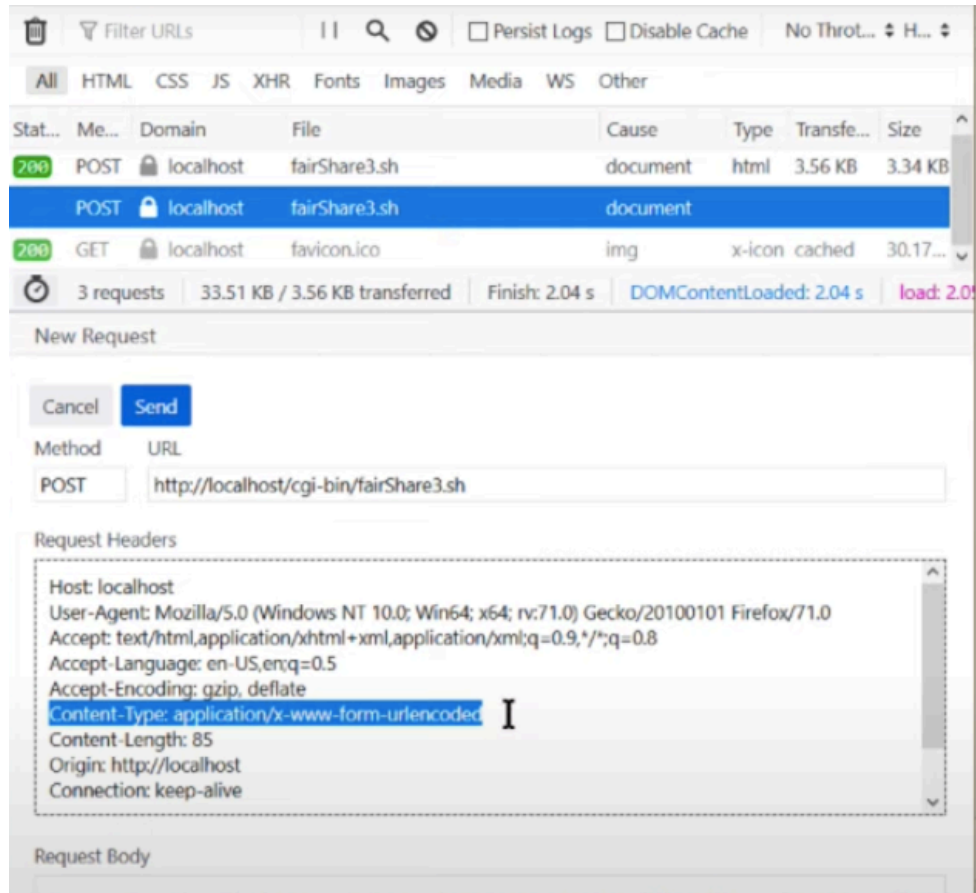
NPTEL

- Fairshare2.html --- “hidden” debug info in web page
- With System.out.println
- Fairshare3.html --- Debug info using logger
- Goes to two places: file and console
- Due to CGI, console goes to web page
- Make that explicit in logger config

I

We changed it over to using logger. And we made the connection between the web page and the logging explicit in the logger config. Alright, now let us take a look at one more use of this kind of logging level etc. Let us see how we can use curl to interact with the application. And what we will do this time is we will take these two command sets and okay. So, first of all, we are going to use curl, curl has an option called `-d`, where you can provide data like this.

And this time with `curl -d` does a post call, it automatically gives the content type application x-www-form encoded. It asks for data and determine the other headers yourself and how did we get the data that is of course, our standard approach which is go to this POST, use “Edit and Resend” and copy this request body. Everything else like this thing content type application **x-www-form** encoded is taking care of by curl. So, let us go in the dev-tools of chrome.



And let us see what happens. We get the result. And we did the log. The nice thing about this is because these are command line applications, which give you the results, it is very easy to write automated tests that do this kind of a job. And as time goes by, as you will see, we have accumulated quite a bit of the log here. Let us go here and do this and as you can see, already fair share log is actually that big.

And if you look at how many lines have accumulated, we have around 112 lines of content in the fair share log, most of which is essentially the same junks of info but naturally if we wanted to, you could also try – “*tail -f fairshare.log*”.

And here we have curl and we can add instead of report, we exchange this to something else “report x” there will be an error there and we can send this and now we have an error.

Plus, this information may or may not be correct at this point, because the report was not properly specified. But which is okay, we expected that we intentionally injected an error and the rest of the posting part curl to care of on its own. If you want to see what curl does, you can add the usual sort of verify. So, let us remove this error and add -v.

```
Aamod Sane@MSI /cygdrive/e/Apps/xampp/xampp-7.4.2-0-VC15/xampp/cgi-bin
$ curl -v -d 'commands=register+f1+f2%0D%0Aexpense+f1+200%0D%0Areport+f2%0D%0Aend%0D%0A' http://localhost/cgi-bin/fairShare3.sh
* STATE: INIT => CONNECT handle 0x8000bb478; line 1491 (connection #-5000)
* Added connection 0. The cache now contains 1 members
* STATE: CONNECT => WAITRESOLVE handle 0x8000bb478; line 1532 (connection #0)
* Trying ::1:80...
* TCP_NODELAY set
* STATE: WAITRESOLVE => WAITCONNECT handle 0x8000bb478; line 1611 (connection #0)
* Connected to localhost (::1) port 80 (#0)
* STATE: WAITCONNECT => SENDPROTOCONNECT handle 0x8000bb478; line 1667 (connection #0)
* Marked for [keep alive]: HTTP default
* STATE: SENDPROTOCONNECT => DO handle 0x8000bb478; line 1685 (connection #0)
> POST /cgi-bin/fairShare3.sh HTTP/1.1
> Host: localhost
> User-Agent: curl/7.66.0
> Accept: */*
> Content-Length: 73
> Content-Type: application/x-www-form-urlencoded
* upload completely sent off: 73 out of 73 bytes
* STATE: DO => DO_DONE handle 0x8000bb478; line 1756 (connection #0)
* STATE: DO_DONE => PERFORM handle 0x8000bb478; line 1877 (connection #0)
* Mark bundle as not supporting multiuse
* HTTP 1.1 or later with persistent connection
< HTTP/1.1 200 OK
< Date: Mon, 02 Mar 2020 20:07:07 GMT
< Server: Apache/2.4.41 (win64) OpenSSL/1.1.1c PHP/7.4.2
< Transfer-Encoding: chunked
< Content-Type: text/html
```

Okay, so here we see the headers. And here we have the interactions. As you can see curl made up all the correct headers it said its post, post user agent, etc. etc. Content length application x www form, URL encoded and these are the usual sort of received headers. Good. So, there we go. Besides curl, there are other libraries. There are Python libraries, there are Java libraries, which allow you to automate this kind of browser interaction. But for simple things, curls and logs does the job pretty well.

The image is a screenshot of a PowerPoint presentation. The title bar at the top shows 'week6-applogs.pptx - PowerPoint' and 'Watch later - Share'. The ribbon includes 'HOME', 'INSERT', 'DESIGN', 'TRANSITIONS', 'ANIMATIONS', 'SLIDE SHOW', 'REVIEW', 'VIEW', 'ADD-INS', and 'Sign in'. The slide content has a green header with the title 'Testing with CURL and Logs'. On the left is the 'Persistent Computing Institute' logo, and on the right is the 'NPTEL' logo. The main content area contains a bulleted list:

- curl -d <data> URL
- Get the data from the browser
- "-d" uses Content-Type:application/x-www-form-urlencoded
- Curl asks for data, and determines other headers itself
- You can put any header you want, however.
- "-F" uses a different type of Form encoding, called multipart encoding
- At this point, parsing form content starts getting more complicated

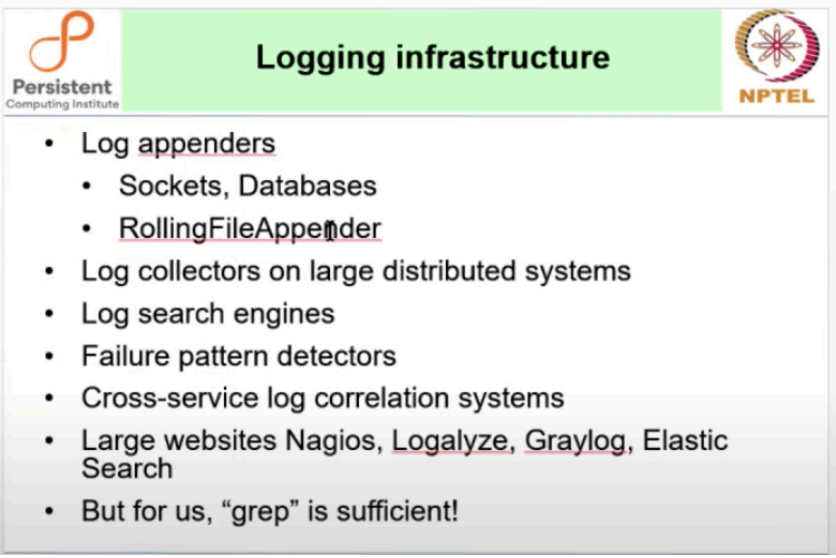
Incidentally, in addition to -d, there is a different sort of form that curl is capable of sending using the encoding called multi part encoding. And at that point, however, parsing starts becoming complicated. And working around with CGI scripts, we kind of come to the end of the things you can do with CGI scripts and now you are to go to more you know infrastructure which caters to all these kinds of complexities.

That is usually servlets. But technically speaking if any Java library which deals with the input properly could be used. So, we will when we go over to servlets, we will discuss exactly what is limiting about CGI bin. But CGI bin is one of the more remarkable successes of the web, because it let us people write simple programs and get useful websites with very little effort. In general, logging is a very important part of web infrastructure.

And besides the appenders that we have seen there are others that go to sockets and databases. There are still other interesting ones. For example, there is something called a rolling file appender. Rolling file appender which you could configure here in this config file log4j2.xml. So, instead of a file you could have a rolling file appender and rolling file appender will actually

once a day or some other specified criteria such as size or time, it will copy an existing file and start a fresh log file for the next day.

It is common to have months worth of log files sitting around, often zipped for to save space, but it is necessary to go back and look at the history of what was happening in application many times, in order to figure out what is going wrong now. On very large distributed systems, people have written log collectors, they have written logs search engines, there are failure pattern detectors, there are cross service log correlation by cross service, I mean across multiple applications.



The slide features a green header bar with the title "Logging infrastructure" in black text. On the left of the header is the "Persistent Computing Institute" logo, and on the right is the "NPTEL" logo. Below the header, a bulleted list is presented on a light gray background. The list includes: "Log appenders" (with sub-bullets "Sockets, Databases" and "RollingFileAppender"), "Log collectors on large distributed systems", "Log search engines", "Failure pattern detectors", "Cross-service log correlation systems", "Large websites Nagios, Logalyze, Graylog, Elastic Search", and "But for us, 'grep' is sufficient!".

- Log appenders
  - Sockets, Databases
  - RollingFileAppender
- Log collectors on large distributed systems
- Log search engines
- Failure pattern detectors
- Cross-service log correlation systems
- Large websites Nagios, Logalyze, Graylog, Elastic Search
- But for us, "grep" is sufficient!

And there are many packages available. For example, large websites use things like *Nagios*, *Logalyze*, *Graylog*, and I am sure more of these will be coming as people write them for one reason or another. Of course, we are just writing the basic web app. And for us grep is sufficient. But it is useful to know that logging is an important part of web application development.

**(Video Ends: 33:35)**

We got involved ourselves in logging very early on. So that you get into the habit of testing using logs, which will serve us very well when we get to the more complex parts of our application. Okay, this is the end of part 2, and see you next time.

Thanks.



