

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 – 34
Week 12 – Part 2

Goal: A last section on cookies, developing the final application.

Cookies: There is one get request to this URL cookie hello and as you can see there is no cookie that is being shown here neither are we sending any cookie in the request header. So, the cookie which we had set was one persistent cookie. You will find that localhost 8080 in fact has a cookie called bbbb which we set which is already expired. So, even though the cookie is known right it is not getting sent that is the meaning of expiry and our cookie quit manager will also show us the same thing it is highlighting this cookie as red. So, a cookie might be kept even though it has expired or it can be reclaimed after expiration that is up to the browser's policies of what to do with extra information like this.

But then there is a new cookie which is the J session ID cookie as we saw yesterday which was created which is created as soon as you contact Tomcat. And if you look at this J session ID cookie it will tell you that the path of this cookie is this highlighted thing cookie hello over here as does the manager which I need to refresh because I did this before I made the request. So, now if you take a look at localhost the J session ID cookie is a session cookie and the path is cookie hello.

Final completed app :

1. Introduce login with the help of a cookie.

Another primary use of the cookies is maintaining sessions. So, let us make a call to fair share login which is the name of our new command and now as soon as I did that you see I got a new J session ID cookie on the path called fair share login this is automatic functionality from any servlet container. This time though our familiar screen was not there. So, our usual screen for say fair share URL, the one where we created the URL has register expense and report.

The change we have is now we have *login* and *register* which are the two usual things you either get a new user to sign up or you get an existing user to login and then there is also a logout button that we introduced. So, **this login screen is a new screen**. So, in addition to our original code here we had a screen that we had simply called *init* this is a more sort of polished screen of the type of app it has. *Inside that we will have the same three screens as we had before* but with some changes.

So, we will see what it feels like first to use the app and then we will look at the details of how the app; the details of how the app is constructed.

Usage of the modified app:

Register and Login

Make a get call to the fair share login page and we have a session ID cookie which was set by Tomcat as usual and we have a new design page.

Upon adding a random non existent user name and trying to login it tells me that there is no such user but we could register such a user. All the expected headers have gone in and there are a few other pieces of information that are available here. Now the other alternative is we could either register a user but to begin with let us go with the user we already know exists.

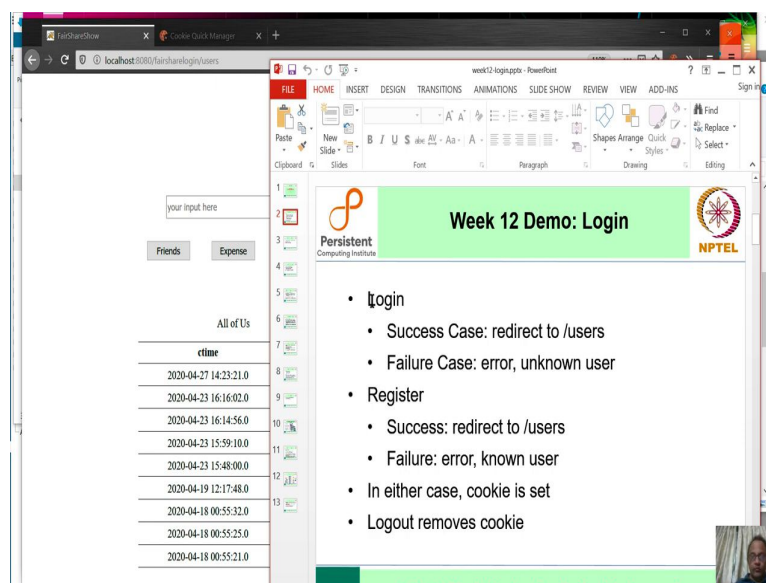
So now I am going to do a login and this time when I do a login I see a slightly different screen there is a logout button which will be present on every single screen and then **instead of register there is a friend's button** which just shows who the people are who are involved in this application. Notice that a few important features are missing like you *cannot remove users* and so on.

There ought to be for example a case where a new user joins but then they never participate in say the trip that you have gone to and so you delete them alternatively the software can just arrange to ignore because they neither paid because they did not pay anything. Now which decision you make is up to you. So, with that let us see what all happened. So, we had a post request.

And when we made a post request we got a new thing, a new set cookie call which means the browser is supposed to remember this and this is a cookie that we have generated. So, if I wanted to see that cookie let us see all the cookies again. So, I have localhost and this you named cookie its path is fair share login and then the session ID is also fair share login. Clearly we are just using the cookie to remember who is the logged in user right the most straightforward imaginable.

And this is exactly what it means to log into a website. There is some indication known to the browser and interpretable by the server that the user is currently viable.

(Refer Slide Time: 07:16)



We looked at the login success case which redirected us to users after the post and this was a **post redirect get** type thing. If you look at the history you will see that there was the original fair share login and then after the actual login page I made a second call to refresh when I showed you and after that we are still just in users which is get call alright.

Now this get call the redirection happened because we got this 302 form. So, the status code called 302 form has redirected us as we have seen before and after the get call we generated this page. So, I made all the changes of course to the JSP as you might expect. And then we have not seen the register yet which we will see in a while and so let us just first complete the entire sequence and then we will come back.

The main thing is in the failure case we just got an error which said unknown user. In the success case we got a redirect and in either success case I should say either success case the

cookie gets set right. And now let us try to see what happens when we do a logout. So, let us try a logout there we go so logout is another post call and it says as it comes back from the server by saying remove the cookie.

*The name the content of the cookie is null and the expiry date is the max age is set to zero as we saw yesterday. So, that constitutes login logout and **one consequence of logout is we are back at this page**. So, we had for example in our history we have the old history but users were never there because for users we made a post and then it got wiped out as a result of the redirect due to login ok.*

Nothing, for any other call like users etc which is meaningful only in the context of a logged in user all this although all the system does is it says you should let me just do this again. It says just go away from here so sending users it intercepts all of these calls and then just comes; we essentially get the contents of the login page ok.

(Actually there is a mistake. What should have been done is to actually redirect so as to remove this URL, why? Because trying to come in with that URL ought not to ever succeed even to the point of remembering it in history. But then the old URLs are still kept because the request that I made was a get request. So, the right thing to do would have been to come back and say that this is invalid and so you should do a 302 redirect. This is one of the considerations that one has to make in reducing the number of error cases.)

Upon trying to register an already existing user, say, abc1 it says 'already exists please login'. Whatever the URL was it is kept so this is why the redirect needs to be there because this URL is actually incorrect. If there is abc1 after correcting the URL then the URL remains unchanged but that is because the post just came back with this with this form right.

One choice is to do a redirect or the other choice is to make this into an **uncachable** form or just leave it as it is because submitting this form is not going to hurt anything. So, what constitutes smooth behavior of an application requires a lot of attention to small details.

We should pay attention not just to the application but also to things like what is the history showing what will happen when the user presses a back button and so on.

If the entry is valid then such a user gets registered. If we look at the cookie then we will have a surname kjll, so register not only succeeds it takes us to the to the user's URL.

When user kjll logs out, it removes the cookie. and in case register succeeded so the end game of both register and login is to get us to the user's page.

We now see the expenses screen and we are registered as the user kjll.

So, for example in fair share URL suppose that I wanted to add some expense for a user lmn2 with expense of 777 and if we hit expense even for a different user we can see it in the expenses table of kjll because we are sharing a single database even though these are independent applications. Because here in this screen even though you are logged in as kjll we can verify by looking at the cookie we are sending. We are sending this uname cookie here.

The post that we had called had entries to username and expense. Body of the post in the above case shows lmn2 + 777.

We can make a rule that says only those expenses will be registered which the current user is paying for. So, if we hit the expense button, then we have kjll and 888 as expected right.

Report Screen

The final screen is a report screen which is supposed to tell you who owes which user how much.

In the previous application our report was actually the list of all the expenses. In the current application we have changed the report as follows:

1. We show the total expenses and dues for everybody.
2. Example: ABC has spent this much money, etc.
3. We added all of it and have averaged it out.
4. So, everybody's share is 3634 whatever the average turns out to be.
5. And accordingly we calculated for those users who have fed more than average they receive some money and those who have paid less than the average they have to pay out some money.

And it is the first time in the app that we have actually done the logic right for which we wrote that diagram that is because in this course our goal was to learn the web infrastructure. In the real web applications frameworks and libraries and so on are available which package enough of this material that in an ideal world you can start focusing on your application over focusing on technology.

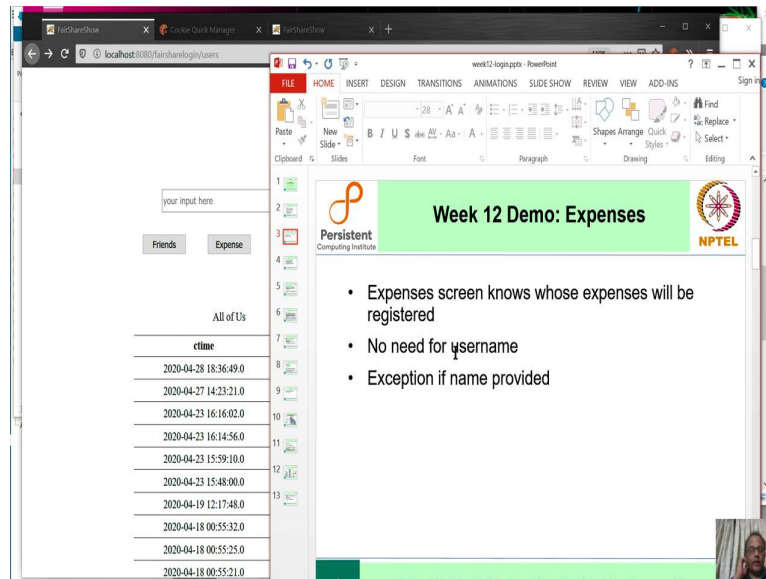
The ability to focus on the application gets better with experience because much of this technology by now is very standard. And unless you are doing something innovative in the application by and large existing tools will get you a long way all right. Now in the new application we actually say that you should not give the name of both the user and the amount right, we just want the amount.

There is a strange issue having to do with the behavior of the logout button: If I now try to give some unexpected input and then ask for expense then as you might expect we have; if you remember from last time we have deliberately left this particular error as an unhandled error in our application.

Not because of course that is what you should do but just to show you what happens when you do not handle these kinds of exception related errors. Mistake which many people make is to just catch every exception and dump it somewhere that is not what you should do. But if there is an unexpected exception then the application should try to recover its default state. At best it can either try to go back to what the user was doing right.

In some sort of for example if you are running a database transaction you could abandon the transaction and so on. That said, you may actually occasionally see applications out there where there exceptions are actually revealed to the users. It is important to never do that all right so that is another reason why showing this is helpful. And what else did I want to show so far so good so the exception expense page does what we want.

(Refer Slide Time: 22:19)



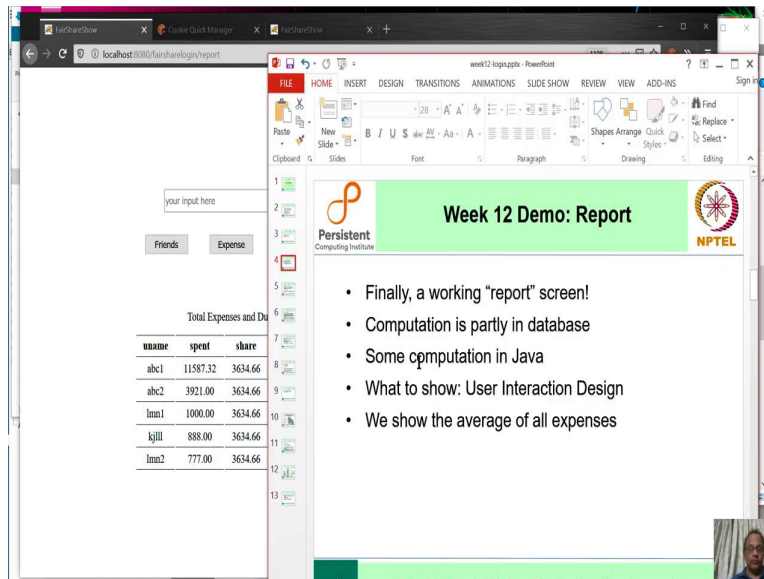
And including the expected error and because our input validation is limited by choice we got an exception. Now I showed you the working report screen which finally does the calculation that the app had actually set out to do. And what to think about in this page among other things is that technically speaking all you need to know is that the application knows how much money is spent and that the application has calculated my should pay amount correctly.

What is the role of a column like share?

The role here is to explain to users what the application is doing. Once they see that the share is calculated by averaging we can add another row here which actually shows the total for it for example. And then demonstrate that this number 3634 is arrived at by dividing the sum of all these things by 5. You can go as far as it makes sense for your users to be comfortable.

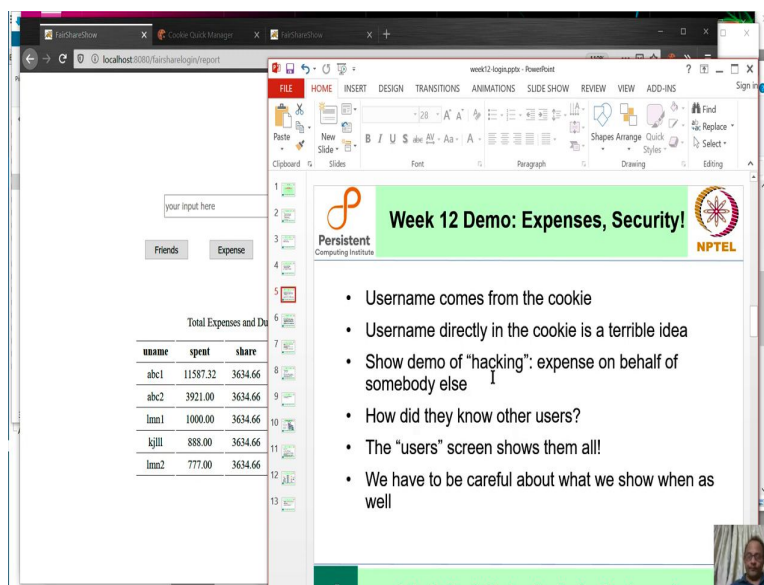
But technically speaking although the share column is unnecessary it is easier to understand what this calculation is once you have an adequate explanation.

(Refer Slide Time: 23:50)



The design of this page is an example of user interaction design to try to make users get comfortable all right.

(Refer Slide Time: 24:03)



We have actually created a cookie here right in the cookie manager in which the **user name is known to anybody**.

HACKER EXAMPLE: SECURITY HOLES

Now suppose there is a hacker who somehow finds out what are the usernames in your application and remember that this hacker may well be a person who logs into your application because after all they do not really care they may have they might they make

validate themselves with a stolen phone or a stolen credit card or god knows what but their goal is to get money from you any which way.

And if you make a mistake like this, such as having a cookie whose structure is understandable to the end-user then the end user can play games like this. I will show you the hack. What I am going to do is although I am logged in as the user kjll I am actually going to submit an expense in the name of a lmn1 here is how I will do this. First of all what I am going to do is to enter some expense right in terms of whichever user is logged in.

This is so that I can get a snapshot of what the interaction of the user is actually like all right. I have registered an expense here. Now what I do next is this, I go to the browser and I can do this in curl or you know any other system that that we might imagine because a browser is after all unknown to the server right we emphasized that the existence of the these two pieces of software communicating via a protocol means that they are independent.

The example I am going to show you is a situation where that independence can lead to problems and then we will defend this from the situation I am going to show you is kind of obvious and most apps already do it but you should see what happens all right. Now what I am going to do is I know that I want to add an expense for somebody else, so I am going to add an expense let us say 888, 999. I am going to add an expense for the user let us say lmn1.

Remember this is all just plain text which is getting sent to the browser so I can actually do this I can remove this session ID and then I will craft a new cookie which I know how to craft because your application was not very careful about it and I have added 999 remember the original input was 17. So, I have to make sure that I change the content length right. I made the content length of the body so I had 17 so there were 30 bytes and then I added one more number right 999.

And the next thing I did was I actually edited the cookie name to a valid user which I found out any which way. And now I actually could do this using curl a browser is strictly speaking not necessary it is simply a text protocol right. So, I am going to submit this request. Let us refresh the page on another screen so that there we go right. Even though I was locked in kjll

I am able to submit expenses in the name of a lmn1 this is what happens when you do things like plain texts cookies.

Therefore you know typically well-designed app the cookie will be some mysterious thing called xyz cookie and the **content of the cookie is encrypted** typically not just with a hidden key for encryption but also with a **time element attached** so that even the encryption is not necessarily stable and predictable in any fashion not that encryptions are particularly easy to predict but if you use encryption in a way that makes the end result predictable then you might open up a involuntary security hole.

KEY LESSONS:

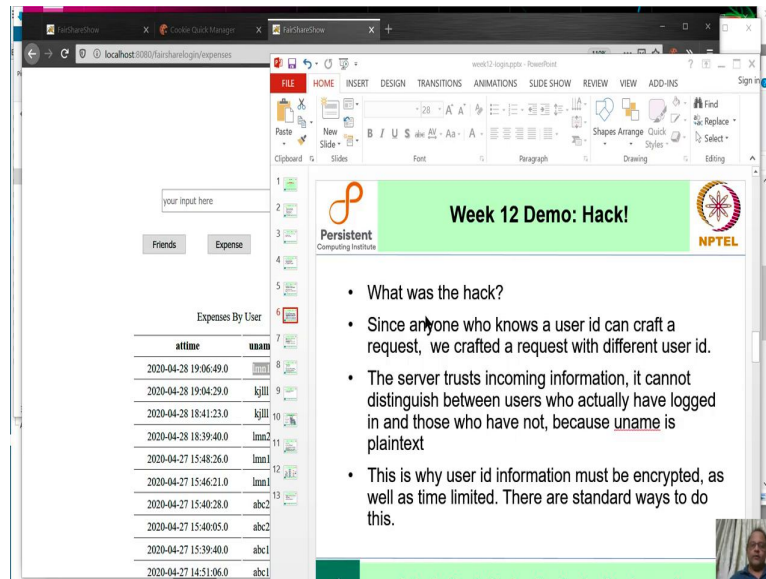
1. Use cookies for identity.
2. Not to make mistakes that allow a third party to actually predict what you are doing because the server cannot distinguish between a valid browser and invalid browser.

All it can say is somebody interacted with me. I gave them the permission to interact further because they validated their identity and from now on if I get an identity then I am just going to believe that the identity is actually true and going to take the action because after all I have no choice.

OWASP: Basic security practices:

I have a valid identity according to the rules that I set. Therefore you have to be careful and you know read up on some of the basic security matters on some website there is a standard best practices **owasp.org** or **owasp.net**. This website will teach you about basic security practices. How to do browser security well is an entire topic in its own right but owasp will get you started.

(Refer Slide Time: 31:34)

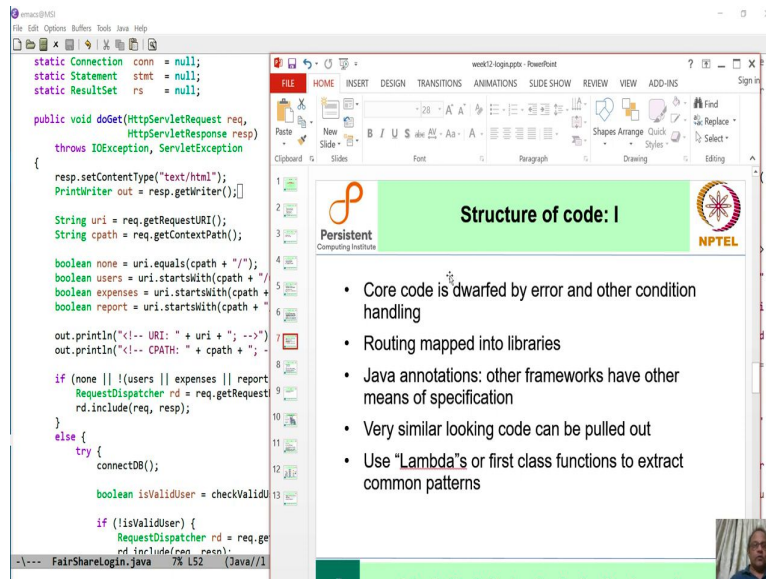


HACK DEMO

In the hack demo just to summarize this in this presentation. Since anybody who knows our user ID can craft a request because we designed our user IDs like that we crafted a request with a different user ID the server trusts incoming information because you are following the rules of the server and therefore it cannot distinguish between users actually logged in versus those who are not because you kept a username plaintext you as in the designer of the application.

As a lesson from this in general, whatever information not just user ID whatever information is sensitive to the server should always be plain text, should always be encrypted and the encryption should have a time element so that you cannot store the encrypted information for a long time and come back sometime later all right that is the primary lesson.

(Refer Slide Time: 32:42)



Let us now go and look at some of the structure of the code. What we will observe is that the core code is absolutely dwarfed by error and other sorts of condition handling and we will observe a few of these things and comment on it after we have sort of traverse through the code. The code begins with the do get and post statements as always. And because we have more buttons we have created these conditions sorry this is the get request so this is not the buttons one post is the buttons one.

For the get request we are getting these URLs and so we detected what the URL is and remembered that in a condition. Next what we did is if you know any other URL so either the URL has no extra material here and it is neither of all of these right because after all our URL is crafted by the user they may do absolutely anything.

And in this case we are going to the login page and as mentioned earlier, I allowed this strange URL to stay in place there is no need I could have redirected it via get to our canonical URL. If we compare the structure of our new app with the structure of earlier app like fair share URL what you will see is that in fair share URL after doing detecting the URL itself.

We were immediately connecting to the DB and so forth. In the more polished version fair share login we are actually ensuring that the URL at least got checked even though the right thing to do is not to just show the page but to actually do a redirect in this style to the starting page so that as we just said cluttered URLs do not remain in the in the browser. Now besides

that change what you will see is that after connecting with the database in the login app we are actually validating that the user is valid.

What this means is that you go to the database and check that the user is actually a pre registered user and if not then you redirect to the starting page again so that you start from a still clean slate. After that the structure still remains reasonably similar, that is we look at what the URLs are and then we show each page as the case may be. So, after checking validity each of those get calls can be made just as we were making them before.

So that is as far as the *get* is concerned, already you can see that adding error checking is increasing the complexity of the application. It is also doing one more thing, that is the *database is potentially being contacted for every single request.*

Post has a similar structure, we detect which button was pressed and then we are going to take actions accordingly.

The gist:

Instead of just register, expense and report we have logout and login as well. There is a register which is different and then there is a friends button which just shows who is part of this group. And register and login has no change at all because you either succeed and then you redirect to the users URL or not the only distinction is register succeeds when the user is new log in succeeds when the user is old that is it.

And then logout sets the cookie as we described. If none of those are involved then this is the regular work of the application and so again you check for validity and only if it is valid you take these actions as a valid user. Again the structure is basically similar if the expense is actually given to you. You add the expense and in case of success go to the expenses URL otherwise complain and show what the expenses are.

In this particular case the error we had before which had to do with you know the incorrect user and so on is actually it can be eliminated because the user is already known all right. And if neither of those cases are true then you redirect to the beginning that is the overall structure. It is true that as far as the input is concerned the user is always known and so the expense screen cannot fail with the wrong user.

This is a reasonable assumption to make as far as the interface is concerned but the database cannot do that really because a mistake could arise somewhere else and therefore you are still checking to make sure that insert only if username exists in the users table right. And moreover we are also going to indicate some sort of an error if nothing else if we do not if you need to report something to the user if there is a failure.

This time the report might be that there is in fact something internally wrong with the database thus by doing that we are able to protect our database whose integrity is quite important. To understand the importance of integrity checking right especially in the context of the web I will show you that there is actually a way where you can manage to inject invalid users at any time in the system especially because we know that just by changing the cookie we can actually change the users. Let us do that.

Here we have a username abc1 and we just you know registered we just logged in as such and so everything this supposed to be good right. Let us do some cleanup here and so we are at this screen let us do the expense screen and we are back at the expense screen again. Now I will show you yet another way to hack the system. Because the user name is visible I can actually select this cookie and change this name so for example I create something called jjjj and save the current cookie right.

No such user exists and now if I hit the report or any one of these URLs see what happens right we got thrown out because there was code here which checked for valid user check valid and then redirected us to the top of the copy. Similarly is there a way we could somehow get a bad user into add expense we do not know right it is better to be safe by doing as much protection of the system as you possibly can.

There could be any reasons including somebody made a mistake in the and detected a user as valid even though they were; in this way we safeguard the system properly.

(Refer Slide Time: 42:19)

The screenshot shows a code editor window on the left with the following Java code:

```

private boolean addExpense(HttpServletRequest req, HttpServletResponse res) throws IOException {
    out.println("<!-- addExpense Call; -->");
    boolean success = false;

    // insert only if uname exists in the use
    String qry =
        "insert into expenses (etime, usrid,
        "select unix_timestamp(), usrid, ?
        "from users where uname = ?";

    PreparedStatement upd = conn.prepareStatement(qry);
    String expense = req.getParameter("theexpense");
    try {
        if (expense != null && !expense.equals("")) {
            String uname = getUsernameFromCookie(req);
            upd.setString(1, expense);
            upd.setString(2, uname);

            int howmany = upd.executeUpdate();

            if (howmany == 0) {
                req.setAttribute("expensefail", "Expense failed");
                req.setAttribute("expense", expense);
                req.setAttribute("username", uname);
            } else {
                success = true;
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

The presentation slide on the right is titled "Structure of code: I" and contains the following bullet points:

- Core code is dwarfed by error and other condition handling
- Routing mapped into libraries
- Java annotations: other frameworks have other means of specification
- Very similar looking code can be pulled out
- Use "Lambda"s or first class functions to extract common patterns

There are certain structures here that have shown up, one is called **routing** which is how to map the URL to actions. There are mechanisms in various languages and frameworks for example Java uses something called annotations there are other techniques.

(Refer Slide Time: 43:36)

The screenshot shows the same code editor window as above. The presentation slide on the right is titled "Structure of code: II" and contains the following bullet points:

- Worth separating out
 - Database handling
 - Error handling
 - JSP
- Many "frameworks" package all these activities
- *They make sense once you understand all the pieces*
- This is why our course is structure bottom up
- All pieces first coming together at the end.

In the case of the report page let us quit with all the hacking and whatnot and let us just look at the report page here. What we did is we somehow got a summary of all the expenses and then did the calculation over here. What I want to, one way to do the calculation would simply be to pull out all the data into the expenses table and do the calculation ourselves but that is not how our code is structured.

We have done something different, abc1 has several expenses, abc2 etc. But all of these are combined in the table using databases.

There are advanced ideas called **stored procedures** which can do even better.

The query selects the username and it actually is capable of adding the amount column as usual and combines the users table and the expenses table on the user ID field and then it does a group by of the users. So, that all the data is compressed together and then all the elements of the group their amounts are summed up using this query.

Such users of the database are relatively or should be relatively standard but unfortunately many developers just get used to doing Java or whatever their languages and do not bother to learn databases properly. But this means that you might be actually not using one of the core facilities that is available to you. Therefore I am showing you a small example right in which we did some of the work in the database we also got this order by so that the heaviest spender is at the top.

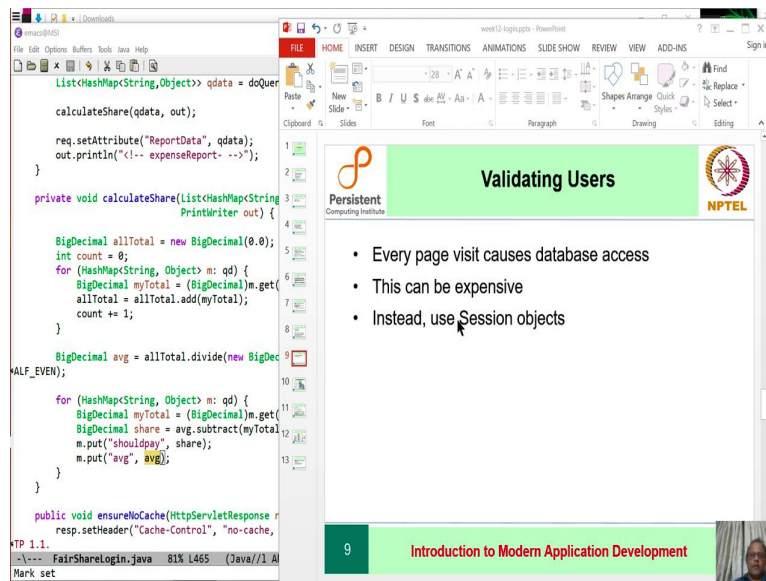
And the database did some of it but let us look at the kind of thing you might also do in application. So, in application what we did was we got all the data and the type we used for these calculations is called bigdecimal which is the type that is used for money calculations because you do not want to make mistakes in money calculations. And the usual floating-point numbers are kind of tricky; they are designed more for scientific purposes than business purposes.

The looks of the big decimal especially in Java is not all that great right but the important thing is it does the work correctly and what I am doing here is of course not at all complicated. You are just adding up the entirety of the thing you are counting how many users are there and then doing a division and creating this you know map which contains the share column with the same quantity put in everywhere so that JSP can go and render it ok.

So that is one of the other lessons which is to use databases and servers carefully. The second lesson of code here is to understand sessions. Remember we were saying that cookies are necessary for two kinds of things one of which is a session object and as we have seen there

is a session ID which is created by these let engines that we can use for storing data related to the sessions. So, let us go and do that next.

(Refer Slide Time: 49:14)



In the code we saw when we check for valid users every page visit can in principle cause database access because ultimately that is where the truth of the matter resides. But there is an alternative which has to solve the following problem. Let us say that I have a server-side session that is I begin interacting with your app at some point and then I logout at some subsequent point or perhaps I do not logout I just stop doing anything for half an hour.

Suppose then at least for that period of time it will be worthwhile if you could just visit the database once upon the first visit and store that information in the cookie anyway as we have been doing. But then the next time you come by instead of visiting the database if there was a way we could remember the correct user ID right inside the server inside the servlet itself. So that you do not have to go to the database.

So what we need here is to normally see what is happening is your get or post or whatever request comes in. All the data that comes in is bundled into the HTTP request these kinds of objects HTTP servlet requests etcetera and then the get method gets caught. So, this data is transient, it lives for the time of the request and is gone where we have space to remember something where a series of connected get and post requests connected by means of a cookie that is can actually be tied together right.

Tying them together is precisely the purpose of the session and we represent the idea of a session with a session object and as I have said before all the ideas that I am teaching you in this course although we are using Java as an example the idea itself necessarily has to exist in every other web application. Because those are those are fundamental things it does not matter what you call it whether you call it a servlet or a page or what have you is not very important.

(Refer Slide Time: 51:51)

The screenshot shows a presentation slide titled "Server Objects: Context, Session, Request" from NPTEL. The slide includes a diagram and a list of bullet points. The diagram illustrates the hierarchy of server objects: a large box for the "Server: localhost:8080" contains an "AppContext (fairsharedb)" box, which in turn contains an "AppContext (at startup) (fairsharelogin)" box. This second AppContext contains multiple "HttpSession" boxes, each representing a user's session. Each session contains "HttpRequest" and "HttpResponse" objects. The bullet points describe the lifecycle and characteristics of these objects: AppContext is created at startup and analyzes web.xml; HttpSession is created on demand and stores key-value pairs; sessions are associated with users and may have multiple requests/responses; and sessions expire by default after 30 minutes. The slide is part of a presentation titled "Introduction to Modern Application Development".

```
import java.math.BigDecimal;

public class FairShareLogin extends HttpServlet {

    private static final long serialVersionUID = 1L;

    private static final String url = "/";
    private static final String user = "admin";
    private static final String password = "admin";
    private static final String DBDriverClass = "com.mysql.jdbc.Driver";

    static Connection conn = null;
    static Statement stmt = null;
    static ResultSet rs = null;

    public void doGet(HttpServletRequest req,
        HttpServletResponse resp)
        throws IOException, ServletException {

        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();

        String uri = req.getRequestURI();
        String cpath = req.getContextPath();

        boolean none = uri.equals(cpath + "/");
        boolean users = uri.startsWith(cpath + "/us");
        boolean expenses = uri.startsWith(cpath + "/ex");
        boolean report = uri.startsWith(cpath + "/re");

        out.println("<!-- URI: " + uri + "; -->");
        out.println("<!-- CPATH: " + cpath + "; -->");
    }
}
```

The outermost box is the server itself that of course has a huge duration. Then inside the server we can start multiple applications at the same time we saw the example of fair share URL and fair share login.

So at runtime the server code itself of course exists then we generate an object called act context for the application which reads which does things like reading web.xml and so on and setting up database related connections. Now for every fresh user visit the only objects that are actually getting created are these HTTP requests and HTTP response objects.

Need for Session

One can have a server which just generates HTTP request and response objects and does not use the intermediates at all. But somehow that information which is shared across multiple requests has to be put somewhere. So, that is why such objects must exist in every web application. So, we have app context which represents the applications and the object representing the concept of session actually exists. It is called an HTTP session.

A session ID cookie is created by Java for every fresh user visit that comes in without any cookie.

The browser sends the J's a session ID cookie, so we also get a new Java session object corresponding to that particular session ID. You go to the request and you say get session, as a new request comes in, it has a session ID associated with it and you can use this session ID to look up in the server code, a hash table basically which says against this session ID there is a session object.

And then when you get a session object you can store, set and retrieve some attribute selling. After the registration is complete you create the new cookie and you also remember in the session object (similarly for login when the login succeeds). We create a new cookie and set an attribute in the session remembering that.

Then of course on failure we indicate the failure. And so check the valid user, get the username from the cookie and if there is no such cookie then naturally check valid user has failed immediately. Get the session object and check for the attribute.

If there is no such attribute go to the database and check and set the username. So, the first time right you check the cookie if it is not yet known to the session, this particular go to the database and get it now stored in the session. On the other hand of course if you do get it then just check that these 2 are equal and your job is done.

That way if the person sends you the same session but at a different username then it would not work. So, if you go back and look at the hack that I did right I removed the session ID as well therefore what happened then was that the people now I could change the user name to anything and depending on whether or not and where you were checking it can have different effects.

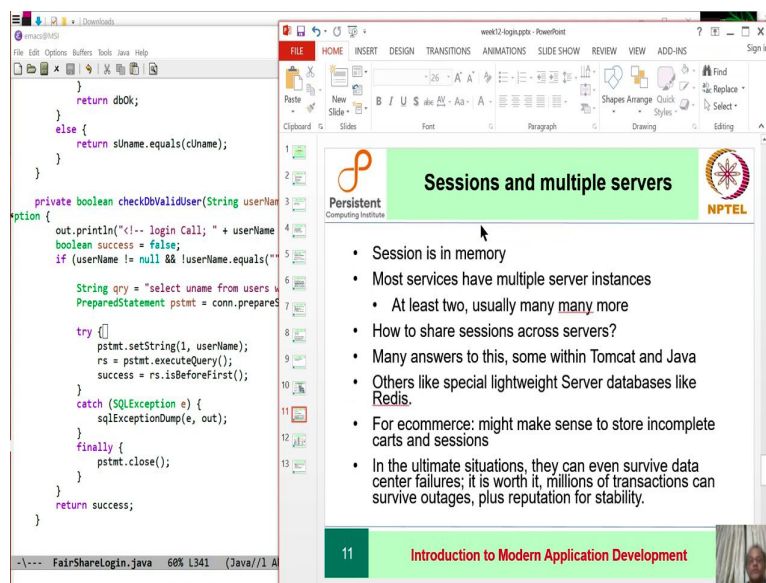
In our scenario, it got a username from the cookie. Because the session ID was different it just generated a fresh session because remember if the session ID cookie is missing you generate a new session.

Then you go to the DB and of course that particular user was valid so the validation check succeeded. If you I mean try similar hacks yourselves in your code you I had changed just the user name it would not have worked because the session would have remembered what the user name was all right. But sessions and all ultimately are clearly efficiency related things, they are not sources of truth, the right source of truth is always the database.

So let us look at it here you have the session object you can store key value pairs and within each session lifetime you can compare against the request response. By default session objects disappear in 30 minutes as to how you check further data for the valid user in the database that straightforward enough you do a select and then you check whether the result set that this function here RS is before first is a standard way of checking whether a result set has any content at all.

It gives you an exception if the check fails, ok sorry it returns a boolean if the check fails only if there was some issue with the sequel itself then you would get a data exception here all right. So, that is the point of the session objects there is a problem still session objects have their limitations.

(Refer Slide Time: 59:51)



The image shows a presentation slide titled "Sessions and multiple servers" from NPTEL. The slide content is as follows:

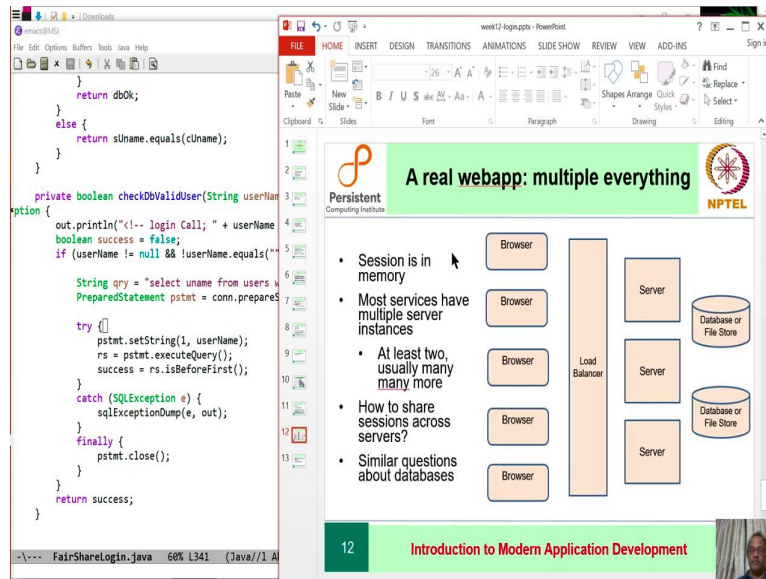
- Session is in memory
- Most services have multiple server instances
 - At least two, usually many many more
- How to share sessions across servers?
- Many answers to this, some within Tomcat and Java
- Others like special lightweight Server databases like Redis.
- For ecommerce: might make sense to store incomplete carts and sessions
- In the ultimate situations, they can even survive data center failures; it is worth it, millions of transactions can survive outages, plus reputation for stability.

The slide is part of a presentation titled "Introduction to Modern Application Development" (slide 11). In the background, a code editor shows the following Java code:

```
return dbOK;
}
else {
    return sUsername.equals(cUsername);
}

private boolean checkValidUser(String username)
{
    out.println("cli-- login Call; " + username);
    boolean success = false;
    if (username != null && !username.equals(""))
    {
        String qry = "select uname from users";
        PreparedStatement pstmt = conn.prepareStatement(qry);
        try {
            pstmt.setString(1, username);
            rs = pstmt.executeQuery();
            success = rs.isBeforeFirst();
        }
        catch (SQLException e) {
            sqlExceptionDump(e, out);
        }
        finally {
            pstmt.close();
        }
    }
    return success;
}
```

So, the issue is the session is in memory but most web services have multiple server instances at least two for redundancy if not many more. **(Refer Slide Time: 1:00:04)**



How to share data across sessions?

The diagram for every real web app looks something like browsers, a **load balancer** which takes incoming requests and spreads them across servers and then shares databases as well. Because you have multiple instances and a session object resides inside the server there is no guarantee I mean there are ways to guarantee it but normally there is no guarantee that one request from a user or a series of requests from a user will always go to the same server right.

Different user requests can go to different servers which is one of the consequences of statelessness because there is nothing in the protocol that says I must talk to a machine which knows something about. How could a load balancer get you to the same server it could do the job of looking at the cookie but that gets really complicated because load balancers typically are low level expensive entities that do not really have time to do fancy things like looking at cookies although of course you can get such load balancers if you really want to.

In any event the point is that because of statelessness and the existence of multiple servers one cannot assume that except in toy circumstances like the ones we have been building that sub requests in a single session will go to a single server of course this is problematic. And there are various aspects of load balancers that can help out but it is better not to rely on such things.

Interesting parts of sessions at least can be stored across multiple HTTP requests. There are features of **load balancers** like sticky load balancing and so on that can help mitigate this.

So that in the worst case you could always regenerate a new session object and go ahead from there. For something like an e-commerce system it might make sense to store the whole thing because after all it is quite important that your users succeed and then in the ultimate case these sessions themselves could even survive something as bad as a datacenter failure.

So sessions are used for many things but this is one example of where sessions can help even for small applications.

(Refer Slide Time: 1:03:20)

The screenshot shows a presentation slide titled "A real webapp: multiple everything" from NPTEL. The slide features a diagram of a distributed architecture with multiple browsers connected to a load balancer, which then routes traffic to multiple server instances. Each server instance is connected to a shared database or file store. The slide includes the following bullet points:

- Session is in memory
- Most services have multiple server instances
- At least two, usually many many more
- How to share sessions across servers?
- Similar questions about databases

The slide also includes the NPTEL logo and the text "Persistent Computing Institute". The slide number "12" and the title "Introduction to Modern Application Development" are visible at the bottom.

The app now is complete; it is distributed, it has multiple elements and so on. And now with the help of cookies we have tied it all together and made it possible to understand the structure of every single web app.

(Refer Slide Time: 1:03:49)

The screenshot shows a presentation slide titled "Next session" from NPTEL. The slide lists the following topics for the next session:

- Could extract login and register into separate parts of framework
- Actually; better than that – build an independent logging web service
- Implementing an independent login service
- Some other aspects of webservers

The slide also includes the NPTEL logo and the text "Persistent Computing Institute". The slide number "13" and the title "Introduction to Modern Application Development" are visible at the bottom.

There is one last problem before the app is complete.

The problem:

Add an input and hit a carriage return, it locks the user.

Reason:

We shut off the access of the user to the putting in input via carriage return. When we added the logout button we associated it with the only form that was on the page which is for my ID. Therefore that was counted as the first button. Remember when I showed you the hack involving invisible buttons the reason we could disable the carriage return was that the first button was made invisible.

But as soon as a button is added before that in the page that started counting as the first button and so it really is the carriage return thing. To show you there is a way to fix that I made a change in just one page which is the expense page.

A fix:

To fix it, we take the expense JSP and put a different form around this button. The form has no input elements or nothing shows up right. But it is now associated with this logout form and so it is no longer the first version for the input field.

Designing and testing requires a great deal of patience with your reward being that you create an interface which is smooth to use relatively error-free for users.