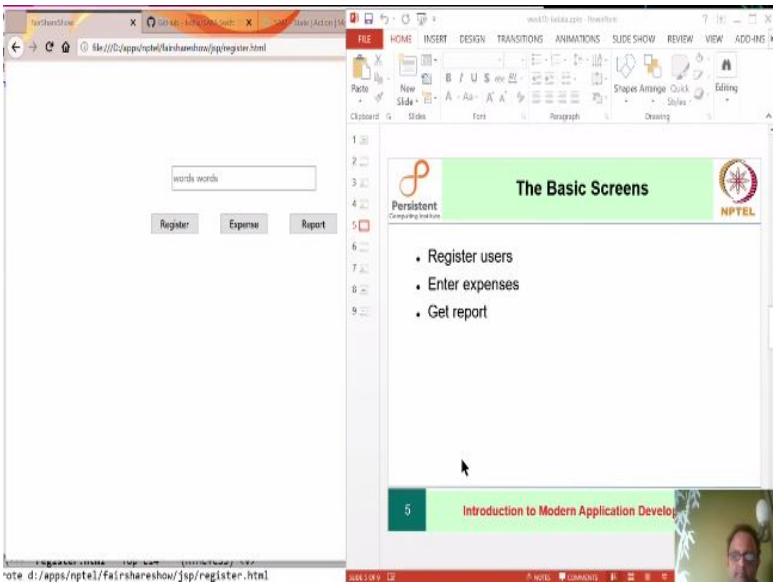
Introduction to Modern Application Development
Prof. Aamod Sane
FLAME University and Persistent Computing Institute
Indian Institute of Technology – Madras

Lecture – 30
Week 10 – Part 2

(Refer Slide Time: 00:11)



We are going to look at how it is that the screen which we created in the last part will be used for the actual application.

How does html sizing work?

To draw the screen, we created a file called register.html. It is just a layout so that you can experiment with what goes on before we actually make it a part of the application.

We saw that if you want to get a stable 20% distance from the top, then you have to use something called viewport dimensions which says padding-top=”20 vh” . It divides the vertical height into 100 parts and takes 20 of them to talk about how much space to use, but it is something interesting, if you look at the actual size of the body as you can see the browser shows you that the body is 909.6 pixels by 259 and the body in fact ends right after the buttons are done.

So as soon as this is done, the body is ending.

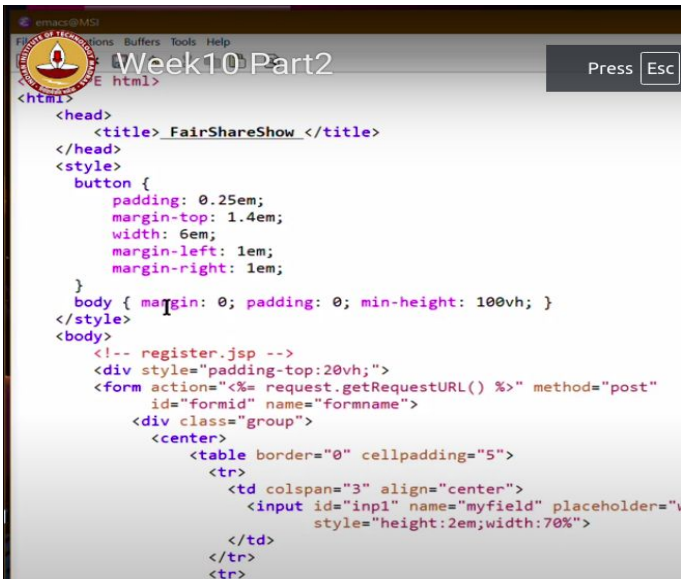
What about the rest of the space that was below this part of the page?

The answer is that the height of the body in normal html is determined by the content and that is not just the body, all tags in html are expressed in percentages of the size of their container. So the container for the body is actually the tag called html and it is the size of the html tag that decides how the body will be measured and so on as parts of the structure.

The *exception* to this is the direct use of viewport sizes.

The available content decides the size of the page and not the page itself, but when thinking about these kinds of applications, the ones that send information get it and so on which are not really necessarily document-like, it is useful to be able to think of the entire available viewport as your page and be able to extend the body to the bottom whenever the content is smaller.

This makes it a little bit easier to see what is going on and you can actually achieve that by doing the following:

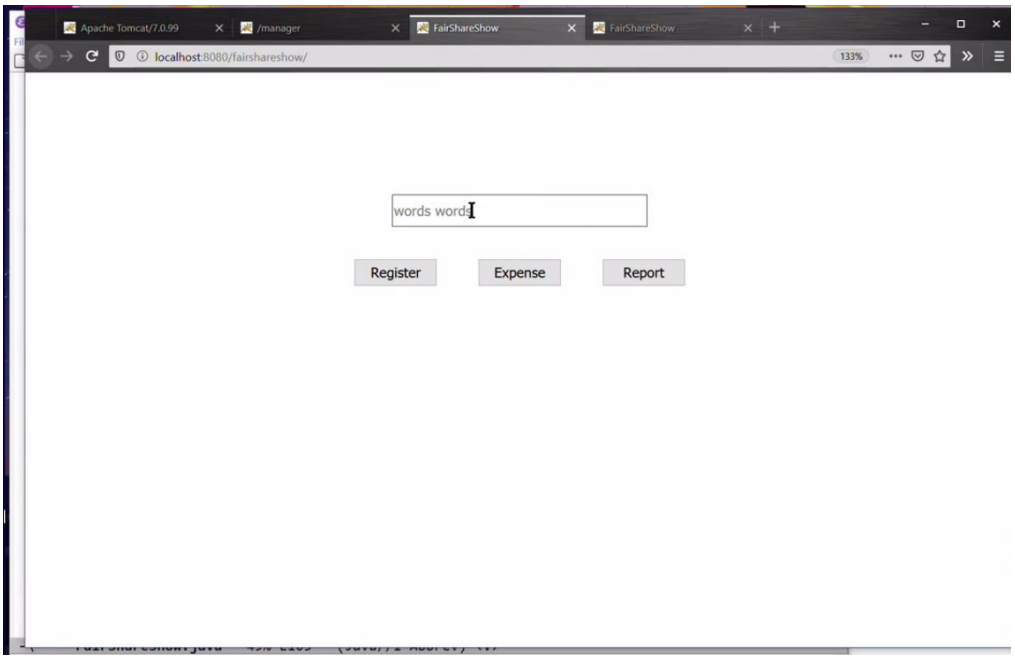


Add style for the body that says the following: margin of the body should be 0, the padding should be 0, and the minimum height of the body, so minimum means if there is not enough content, then the height should be 100% of the viewport height.

I am going to make it so small that a scrollbar appears and in that case the body is now extending below the fold because the scrollbar will take over the rest of the area. So this is how space gets used in the html.

As we saw last time, we have created this entire thing using a table, although in some more advanced ways to use CSS.

So first when we visit this app, the page looks like this:



In this development of the app, we are only going to look at using a part of the app which retrieves information that is already in the database. Then we will see what it takes to put the information into the database; first in the form of a simple java program and then we will include that in the actual app. While querying the database to get the data is relatively simple, updating the database is not. The reason is that the behaviors that we want from the app when it comes to updates are not at all simple to understand.

Why is database update from a web application harder than implementing querying?

Suppose you have three functionalities to implement: register users, enter expenses and get report.

Though it seems like straightforward tasks, some complications arise while implementing.

Example:

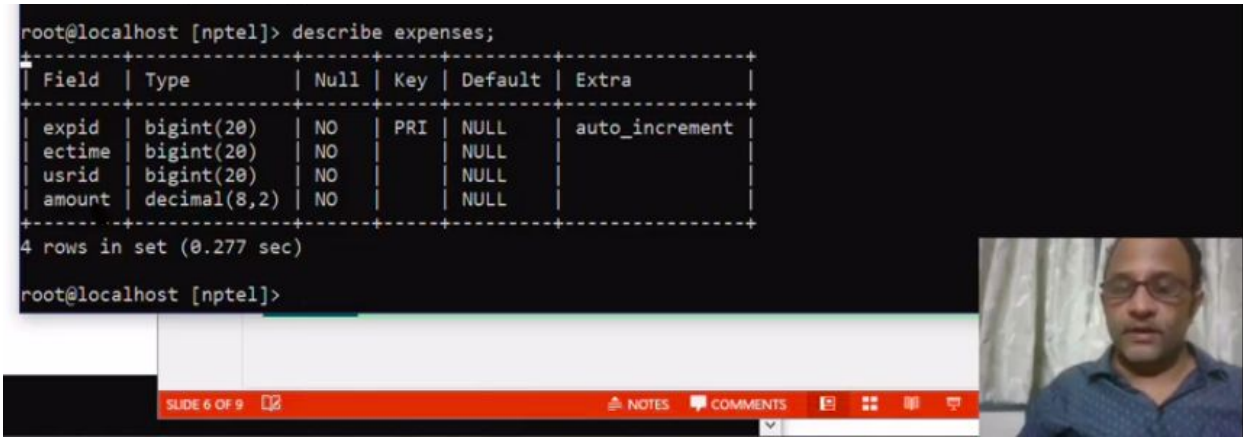
- Register user: What if the same person has already registered? What if 2 people have the same name? So even something as simple as registration is not at all actually simple and in part we solve this problem via some mechanisms of the database and in part by some mechanisms within the program itself.

- Enter expenses: What if there is an error in an earlier entered expense? So now all of a sudden we realise we need to support deletion when you did not actually plan any such thing. What if you accidentally enter duplicate expenses?

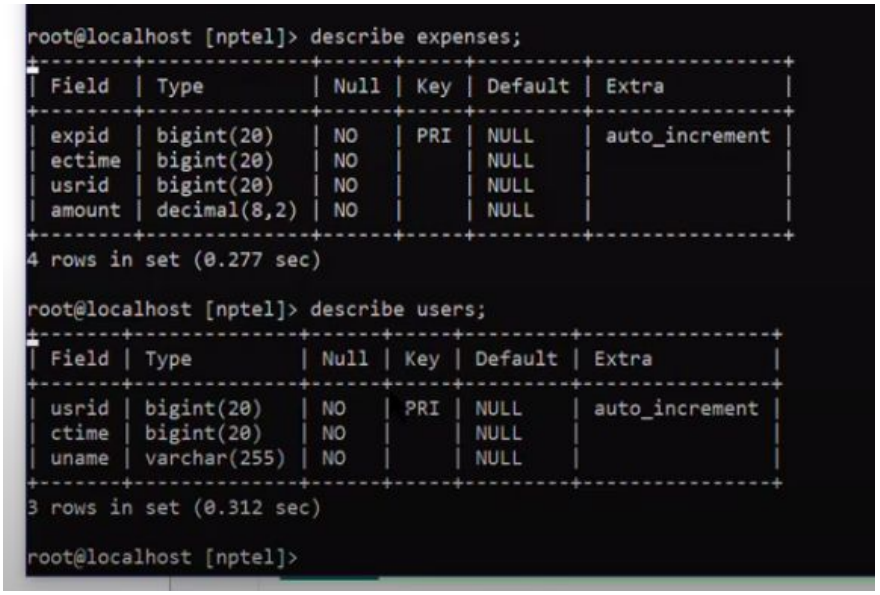
- Get report: Because it is a query, things are relatively simple.

<u>Implementing only queries in the web application(no updates)</u>

In the database nptel, there are these 3 tables(`expenses` and `users`, already seen in the last lecture) while `users2` is a new table.



Using the query: `describe expenses;` it shows the fields expense ID, expense creation time, user ID and the amount spent as shown in the above figure. All 3 of `expid`, `ectime`, `usrid` are 64 bit identifiers and the `expid` is the *primary key*.



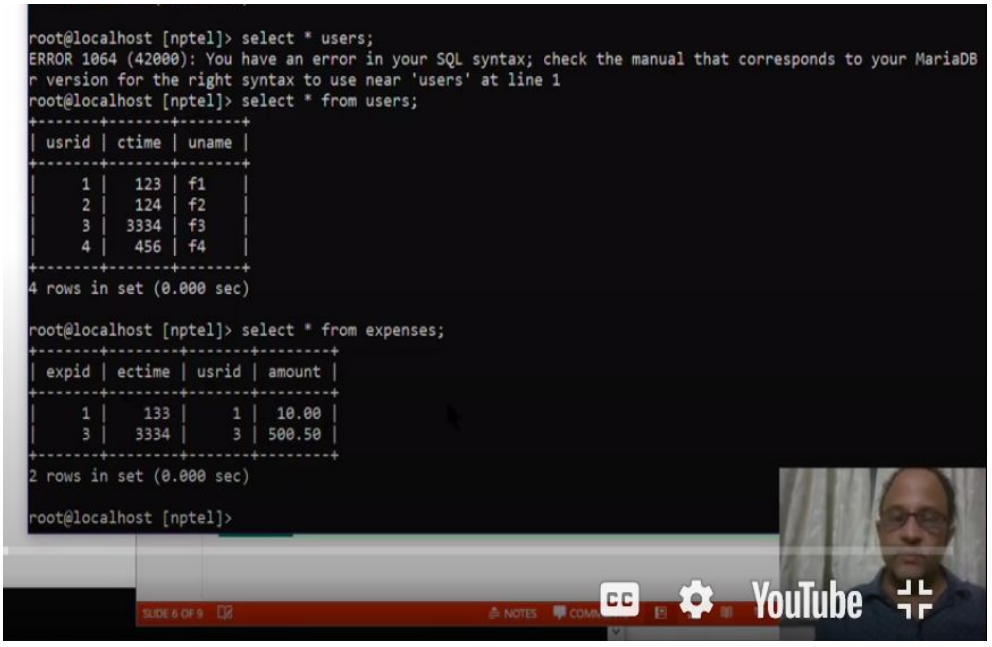
```
root@localhost [nptel]> describe expenses;
+--------+-------------+------+-----+---------+----------------+
| Field  | Type        | Null | Key | Default | Extra          |
+--------+-------------+------+-----+---------+----------------+
| expid  | bigint(20)  | NO   | PRI | NULL    | auto_increment |
| ectime | bigint(20)  | NO   |     | NULL    |                |
| usrid  | bigint(20)  | NO   |     | NULL    |                |
| amount | decimal(8,2)| NO   |     | NULL    |                |
+--------+-------------+------+-----+---------+----------------+
4 rows in set (0.277 sec)

root@localhost [nptel]> describe users;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| usrid | bigint(20)  | NO   | PRI | NULL    | auto_increment |
| ctime | bigint(20)  | NO   |     | NULL    |                |
| uname | varchar(255)| NO   |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
3 rows in set (0.312 sec)

root@localhost [nptel]>
```

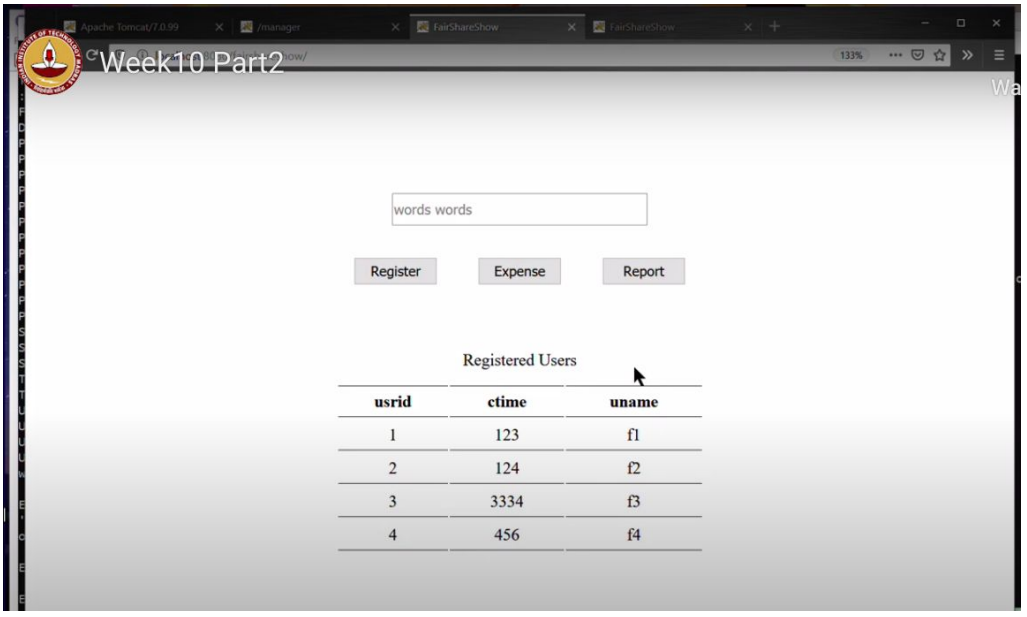
The users table has a similar structure with a user ID, creation time and name of the user. Here, the primary key is the `usrid`.

Standard unix time, which is milliseconds from a certain chosen instant in 1970 can be used for time stamps. There are different reasons to use this particular notation for time. Among other things, it is easy to share amongst different competitors and you can convert it to various forms on the output side.



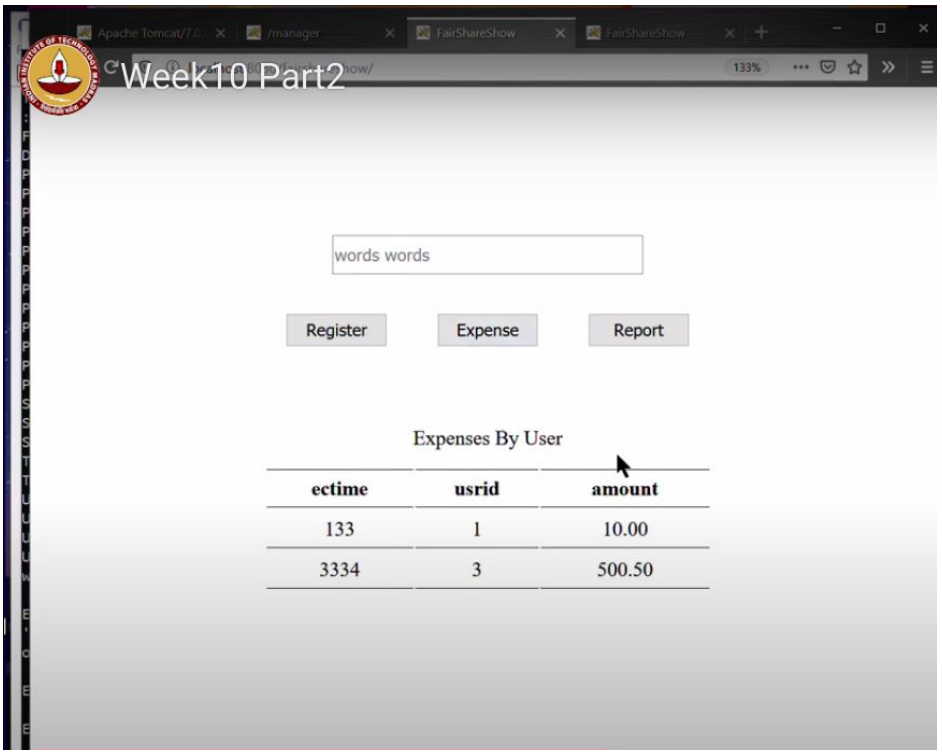
Register a new user using the web application, by entering data in the input field and clicking the register button in the web application. The system is going to just ignore the new entry as we have not yet implemented the registering functionality from the browser. Upon clicking the register button, the previously populated users table is displayed.

Upon clicking register button:

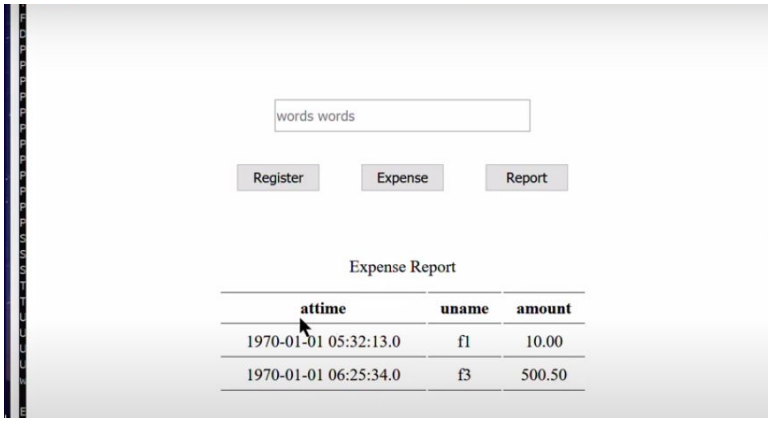


The interesting thing is compared to the java application we developed to display the sers table, we have a nicely structured output using HTML.

Upon clicking expense button:



It displays the expense table in nptel database, without the expense id as it might be irrelevant to the end user.



The report button updates the app and displays the Expense Report table, which displays the expenses incurred and the time converted from integer to a more readable format.

How did we get the above Expense Report table?

First, take a look at the structure of the html file

The expense report table combines data from the users table, which gives it a name and the expense table themselves which are written in terms of Ids and shows you the result

accordingly.

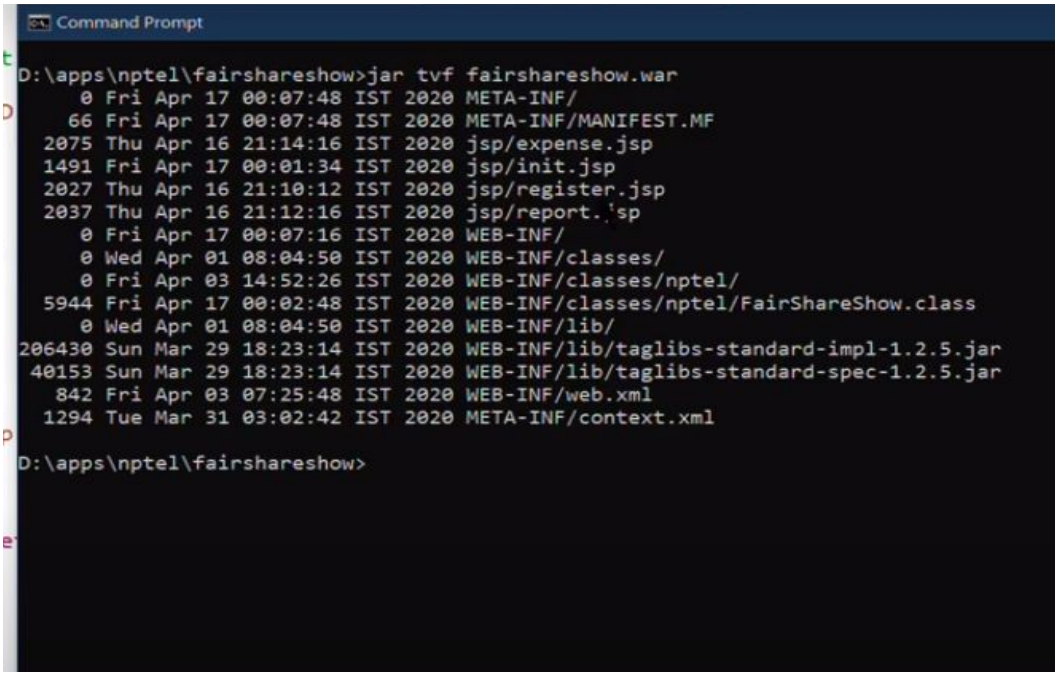
As seen in the above figures, we have 4 screens in this app. One thing that is happening is because we are doing this as a succession of post requests, as you will see; all of these visits will show up in the history of the browser.

This app simply retrieves whatever the database has, and displays on the browser.

What is in the war file of this application?

Firstly, `tvf` in the `jar` command means:

- v is for verbose
- t is for showing the contents
- f is for the war file itself



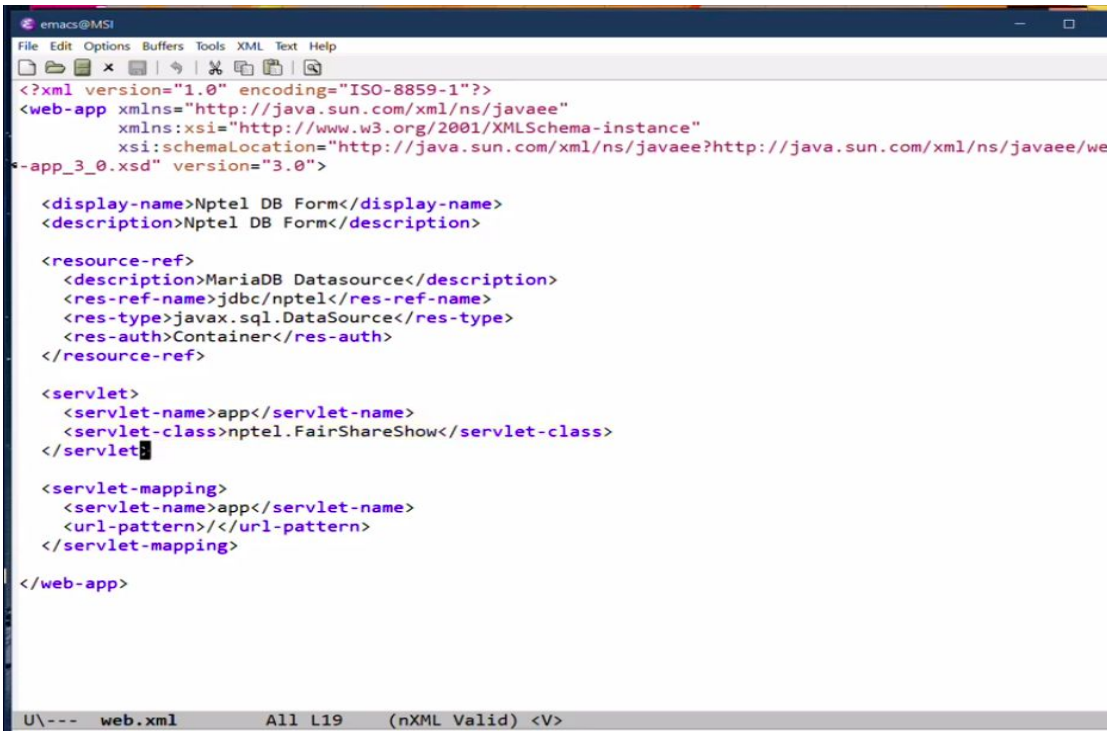
`jar` command creates

- META-INF and MANIFEST directories, as we have seen before.
- Inside the jsp directory, there are 4 jsps corresponding to the 4 pages.
- There is a class file in the classes directory: fairShareShow.class.
- In the lib directory, we have tag libraries. Tag libraries are for the jsp tags as we have

seen before.

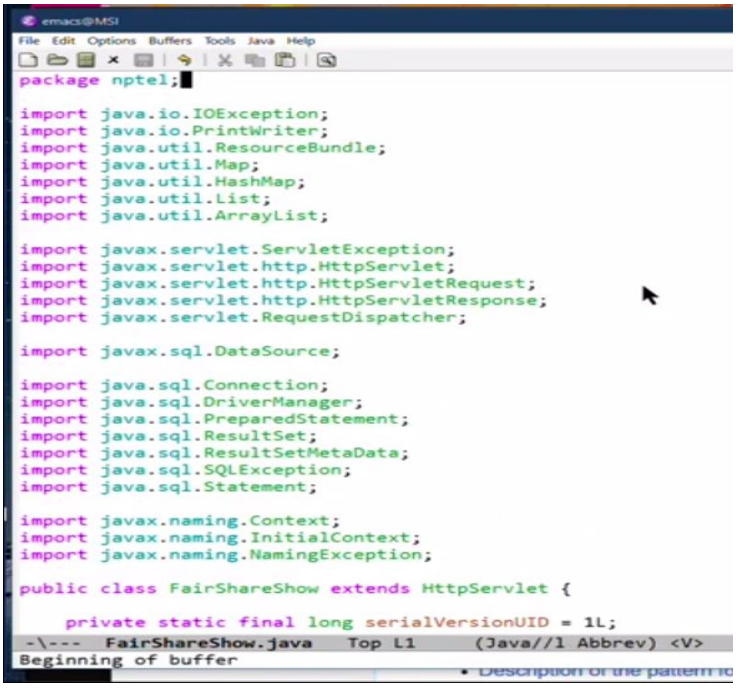
- web.xml shows how the web URLs are connected to the java servlet methods
- context.xml is intended for connecting the database via JNDI as we have also seen earlier.

Let us look at web.xml of this application:

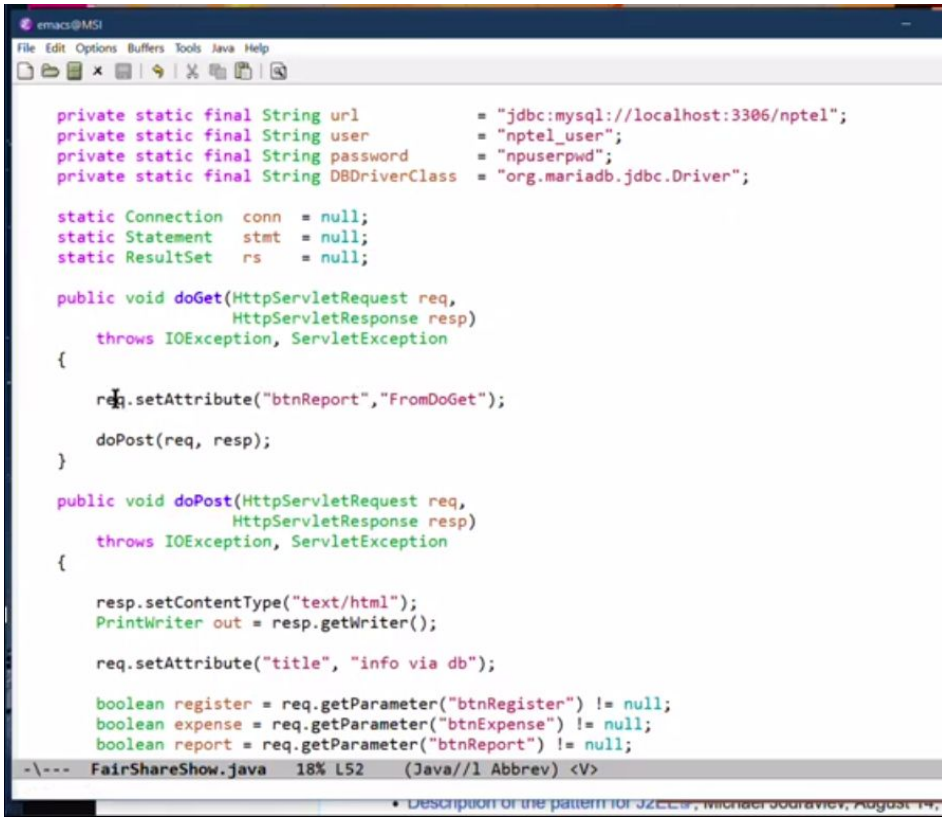


1. <resource-ref> tells where the database is.
2. Name of the servlet is app and the servlet class is nptel.FairShareShow
3. Mapping says that the URL /, which is the only URL in this app is mapped to the servlet.
4. There are no special URLs, that single URL is being reused for everything because there is no flow of pages. We are refreshing the same page again and again. This style is sometimes called a single page app and if you look at discussions on the web, they tend to be somewhat elaborate about what this app is for, does not have to be as complicated as they make out to be. At the same time, there is a technical reason why these things are called single page apps, whereas the kind of app which we are building, which rebuilds the entire page from the server every single time is not quite called that, but the effect is the same.
5. We are not changing the URL, we are keeping the URL the same and showing the

results of what it is we get. We could do a different design in which the report being shown for instance is not on the same page and each report has a distinct result, but that is kind of clumsy to use for an app like this.

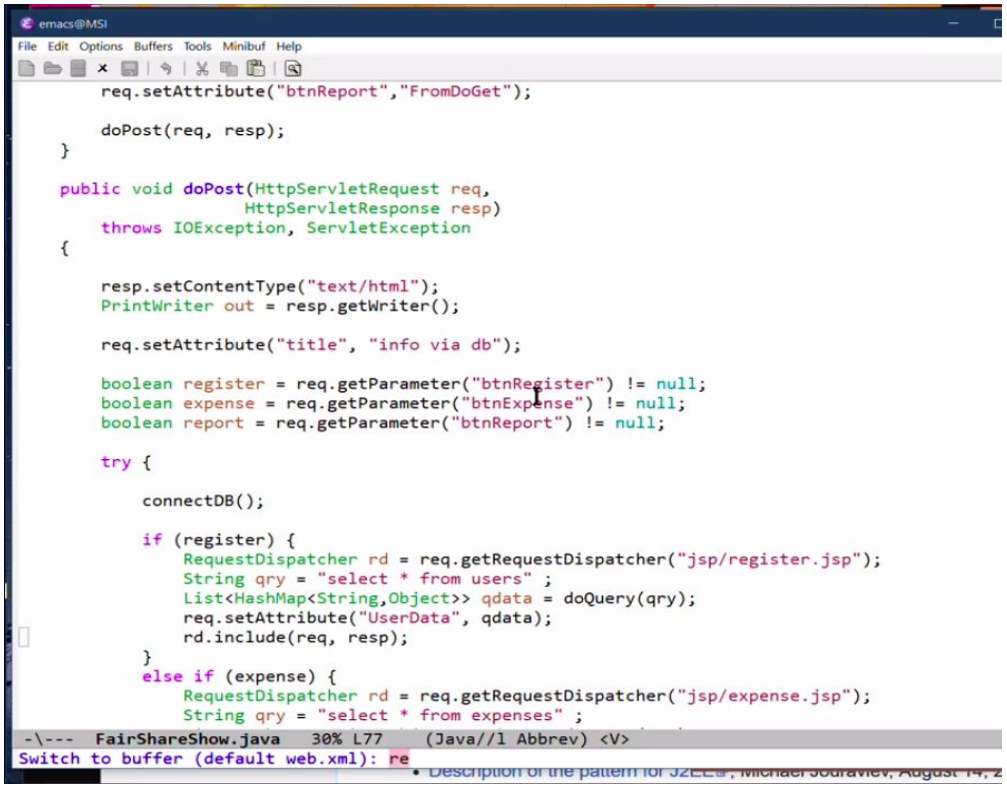


FairShareShow.java has a lot of imports

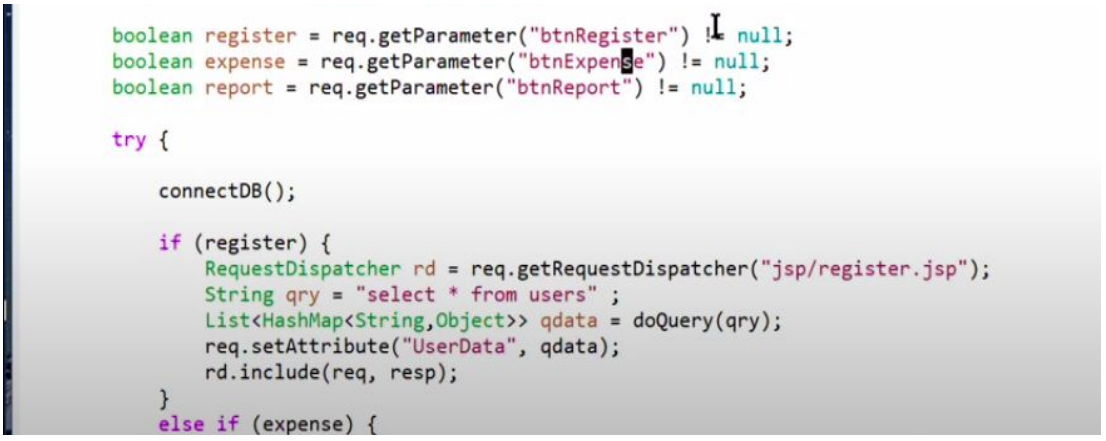


FairShareShow extends HttpServlet as usual.

The two key methods are doGet and doPost, as in a servlet. The doGet method actually just sets the attribute using class fields without explicitly entering the value and passes it down to post.



We set an attribute for the title using req.setAttribute as seen earlier. We look for request parameters using `req.getParameter()`, which return the button values in the form, the value is not null if the button is currently clicked.



init.jsp:

```
<tr>
  <td colspan="3" align="center">
    <input id="inp1" name="myfield" placeholder="words words"
           style="height:2em;width:70%">
  </td>
</tr>
<tr>
  <td>
    <button id="btn1"
            name="btnRegister" type="submit"
            value="Register">Register</button>
  </td>
  <td>
    <button id="btn2"
            name="btnExpense" type="submit"
            value="Expense">Expense</button>
  </td>
  <td>
    <button id="btn3"
            name="btnReport" type="submit"
            value="Report">Report</button>
  </td>
</tr>
</table>
</center>
```

Value of the name of the button shows up as a parameter(here, btnRegister).

So, you can have 3 types of buttons and if that parameter is not null, then that means that button has been pressed.

In a Boolean variable we do something fairly straightforward. You connect to the database. If it is a register command, run the query that is related to the register, convert the contents into a list of hashmaps as we have seen before, set that query data as user data and then include this thing.

If it is report, the query is somewhat complicated as we have seen in the last lectures, it combines two tables using join.

```
emacs@MSI
File  Edit  Options  Buffers  Tools  SQL  Help

select
   from_unixtime(ectime) as attime,
   uname,
   amount
from
   users, expenses
where
   users.usrid = expenses.usrid
```

So, we take this query and put it in a string in fairShareShow.java:

```
String reportQry =
    "select " +
      "from_unixtime(ectime) as attime, " +
      "uname, " +
      "amount " +
    "from " +
      "users, expenses " +
    "where " +
      "users.usrid = expenses.usrid ";
```

How to use the query strings?

```
emacs@MSI
File  Edit  Options  Buffers  Tools  Java  Help

        connectDB();

        if (register) {
            RequestDispatcher rd = req.getRequestDispatcher("jsp/register.jsp");
            String qry = "select * from users" ;
            List<HashMap<String,Object>> qdata = doQuery(qry);
            req.setAttribute("UserData", qdata);
            rd.include(req, resp);
        }
        else if (expense) {
            RequestDispatcher rd = req.getRequestDispatcher("jsp/expense.jsp");
            String qry = "select * from expenses" ;
            List<HashMap<String,Object>> qdata = doQuery(qry);
            req.setAttribute("ExpenseData", qdata);
            rd.include(req, resp);
        }
        else if (report) {
            RequestDispatcher rd = req.getRequestDispatcher("jsp/report.jsp");

            String reportQry =
                "select " +
                  "from_unixtime(ectime) as attime, " +
                  "uname, " +
                  "amount " +
                "from " +
                  "users, expenses " +
                "where " +
                  "users.usrid = expenses.usrid ";

            List<HashMap<String,Object>> qdata = doQuery(reportQry);

            req.setAttribute("ReportData", qdata);
-\---  FairShareShow.java    41% L91     (Java//l Abbrev) <V>
```

When you visit the URL for the first time, call init.jsp and the processing remains the same. If there is a failure, then you have a SQL exception which you can dump.

If there is any other failure, then we are currently just going to print a stack trace. Eventually, we will get these traces into the log and the web page, but for now, we will just leave it to be dump.

```
public void sqlExceptionDump(SQLException e, PrintWriter out) {
    e.printStackTrace(out);
    out.println ("Message: " + e.getMessage ());
    out.println ("State: " + e.getSQLState ());
    out.println ("MariaDB code: " + e.getErrorCode ());
    out.println ("cause: " + e.getCause ());
}

public void connectDB() throws NamingException, SQLException, ClassNotFoundException {

    String path = getClass().getProtectionDomain().getCodeSource().getLocation().getFile();

    boolean isEmbedded = !path.contains("webapps");

    if (isEmbedded) {
        Class.forName(DBDriverClass);
        conn = DriverManager.getConnection(url, user, password);
    }
    else {
        Context ctx = new InitialContext();
        DataSource ds = (DataSource)ctx.lookup("java:comp/env/jdbc/nptel");
        conn = ds.getConnection();
    }
```

The highlighted text is for testing with embedded tomcat. If we get time, we will look at the embedded Tomcat, at least mention what is happening here.

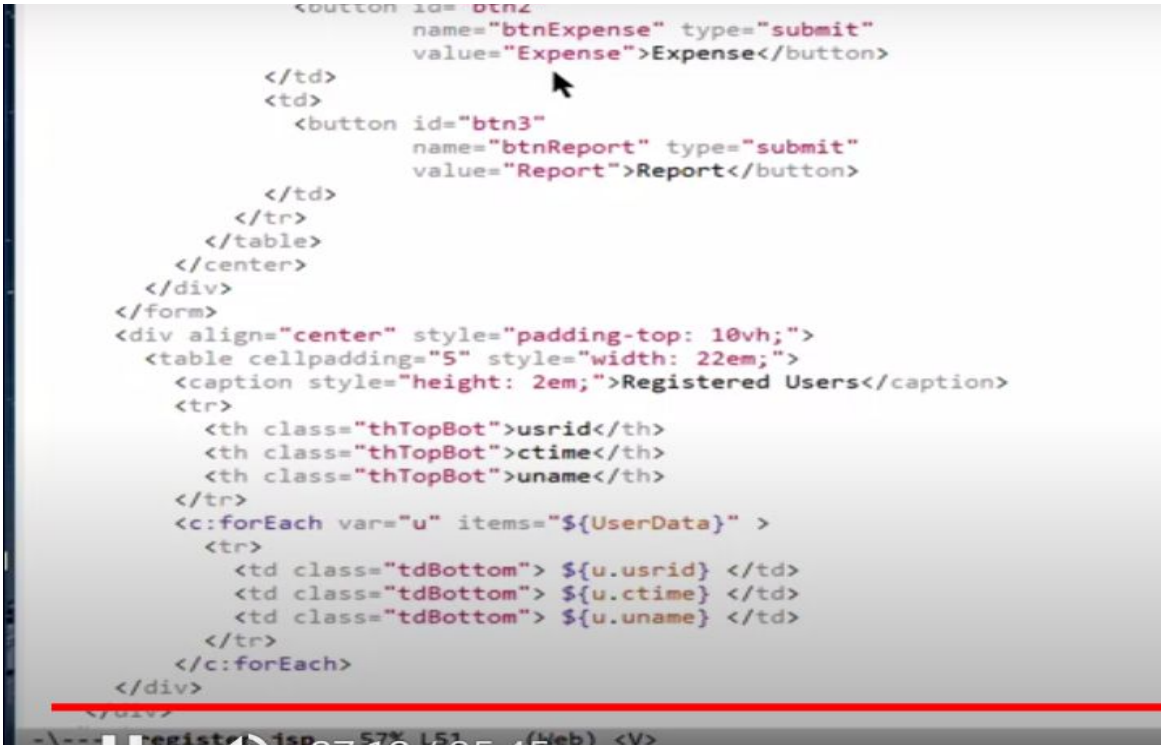
Finally, we close the database.

What does init.jsp contain?

init.jsp is just the layout page that we have discussed with height and width attributes.

```
<form action="<%= request.getRequestURL() %>" method="post"
      id="formid" name="formname">
  <div>
    <center>
      <table border="0" cellpadding="5">
        <tr>
          <td colspan="3" align="center">
            <input id="inp1" name="myfield" placeholder="words words"
                   style="height:2em;width:70%">
          </td>
        </tr>
        <tr>
          <td>
            <button id="btn1"
                    name="btnRegister" type="submit"
                    value="Register">Register</button>
          </td>
          <td>
            <button id="btn2"
                    name="btnExpense" type="submit"
                    value="Expense">Expense</button>
          </td>
          <td>
            <button id="btn3"
                    name="btnReport" type="submit"
                    value="Report">Report</button>
          </td>
        </tr>
      </table>
```

register.jsp.: It basically has the same code and adds a table called Registered users.



We have a table and the width of the table is 22em. Reason for the choice of width is to have a table which is slightly wider than the area spanned by the table which has an input field and 3 buttons.

By creating the height of the caption, I have created some space around it and the style of table borders, many user interface designers have learned over time that the simplest kind of table puts these horizontal lines and leaves the columns without any column separations, and it is not that bad looking, so we will stick with it.

In this table, what we have is table headers and part of the forEach loop produces the rows. It gets the user data and remembers user data is from FairShareShow.

We get a hash map and the hash map tells us what user ID, ctime, and uname are, which are the results returned by the query. All the cell descriptions, the td elements, have something called a class, td bottom and the headers have another class called th top and bottom. This kind of class thing is again a part of CSS.

For every node in html, you can add a class which says that this particular header is of the

type th top and bottom, so table header with the top and bottom, and this table cell here is of type td bottom, so it only has a single border is what we want to say.

What a class defines is a style, so that instead of using explicit style attributes, as we have seen before, we will be able to get some information from the style section.

We gather information into a class and say that it has a bottom border and it has a top border, and for the table itself, what we have done is we have not specified a border at all and so there is no implicit border. We actually get to say what this is, and for the table cells, we will say that the alignment has to be at the center.

The alignment for the header is already center that is given simply by using th and therefore with this CSS, we have the table appearance that we wanted. As usual, if you want to actually play with it and look at some of the details, you can always open up the web developer tools and start looking at what all is involved.

This is how we get a relatively clean layout of the information.