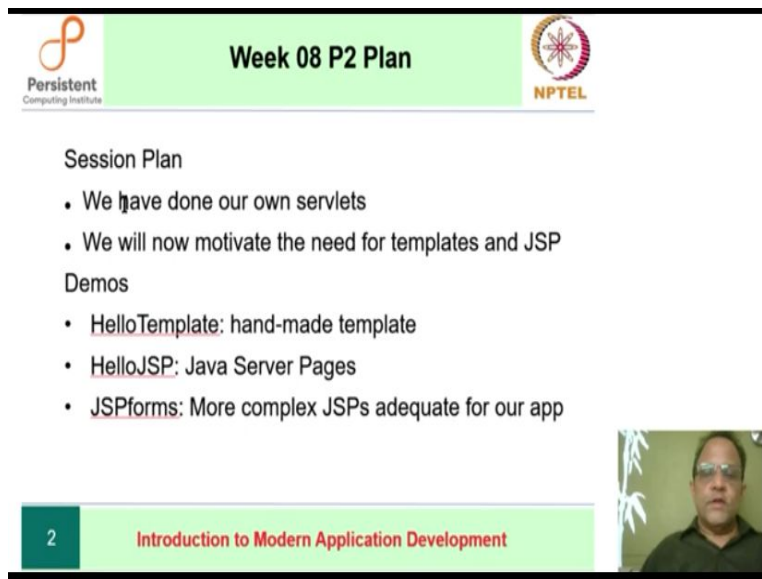


Modern Application Development
Prof. Aamod Sane
FLAME University and Persistent Computing Institute
Indian Institute of Technology – Madras

Lecture 25
Introduction to Modern Application Development

(Refer Slide Time: 00:28)



Week 08 P2 Plan

Session Plan

- We have done our own servlets
- We will now motivate the need for templates and JSP

Demos

- HelloTemplate: hand-made template
- HelloJSP: Java Server Pages
- JSPforms: More complex JSPs adequate for our app

2 Introduction to Modern Application Development

Hello, welcome to the second session of week 8. We have already done our own servlets.

- Remaining parts of JSP and servlets.
- Motivate the need for templates
- Java version of templates, which is Java Server Pages.

Demos:

- Hello world using templates
- Basic problem to be solved in templates, how Java Server Pages solve that.
- Use JSP to create a form application

(Refer Slide Time: 01:29)



Review



- Basic Servlet Layout
- Tomcat management


3

Introduction to Modern Application Development




Let us take a look briefly at what we learnt last time.

(Refer Slide Time: 01:37)




Servlet Layout



- Directory structure
 - ✦ src
 - WEB-INF
 - classes
 - lib
 - web.xml
 - Web.xml and URL mapping

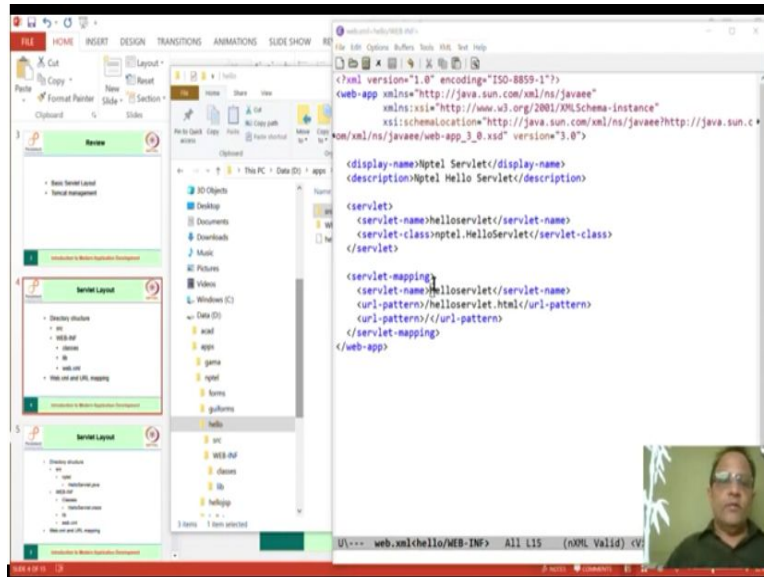
4

Introduction to Modern Application Development



- Basic server layout.
- Directory structure of the systems (as shown in Slide 1:37)

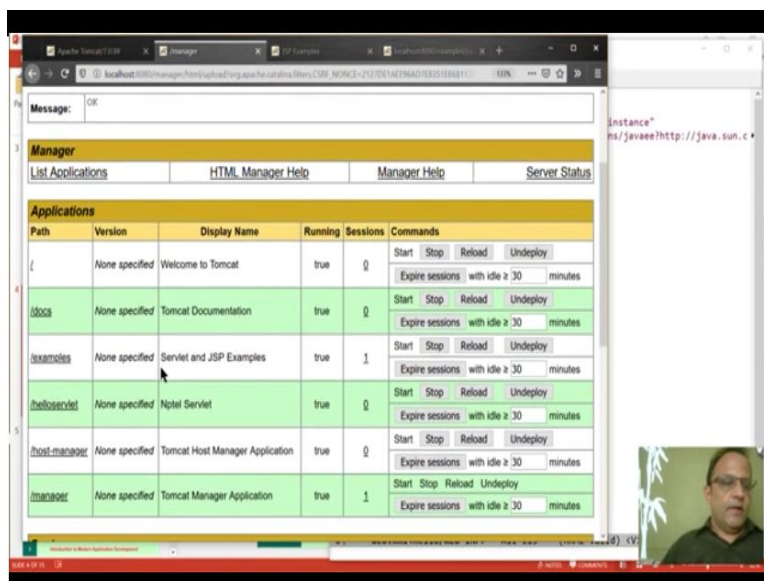
(Refer Slide Time: 02:35)



Web.xml, in the case of the Hello application:

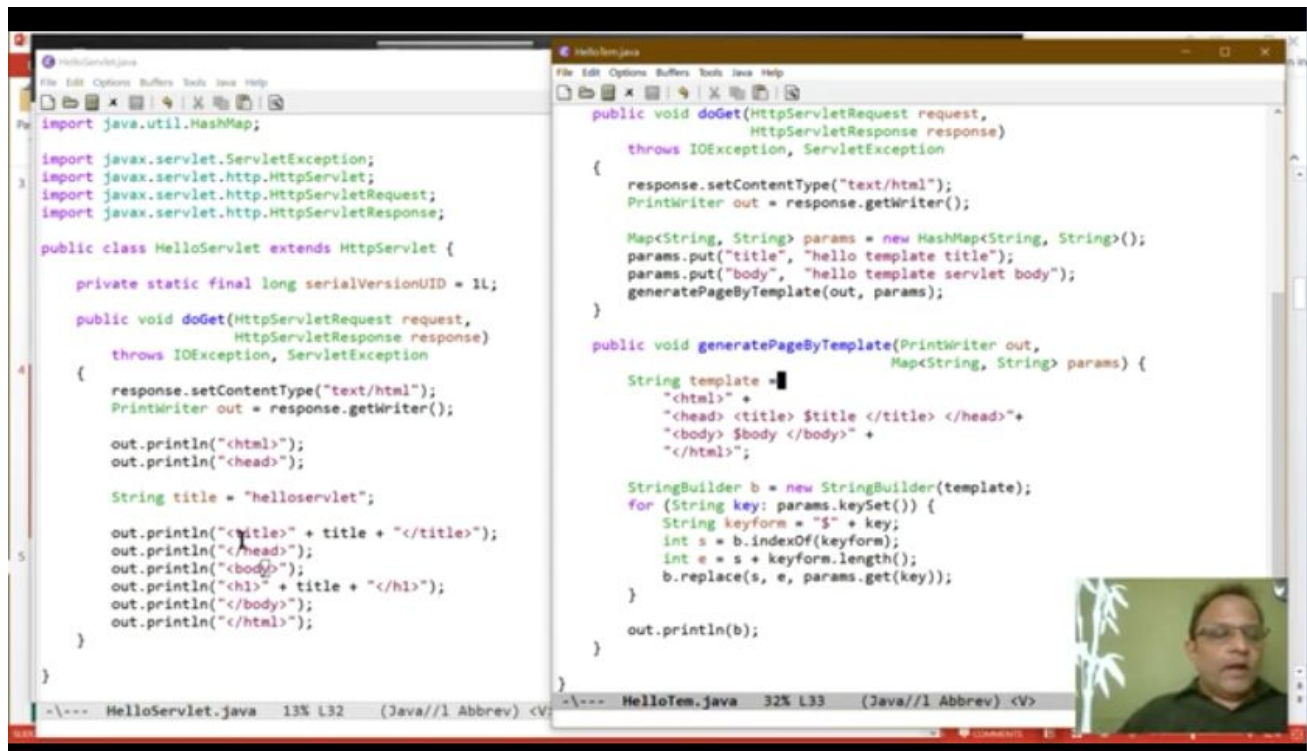
- Description to the Tomcat server what the arrangement of servlets is like
- URL maps: So what we have here is that we created a servlet, which is implemented in the class HelloServlet, whose name is HelloServlet and that name is available under helloservlet.html and “/”

(Refer Slide Time: 03:20)



HelloTem.java is the java template, vs HelloServlet.java

(Refer Slide Time: 04:29)



The difference:

In HelloServlet.java:

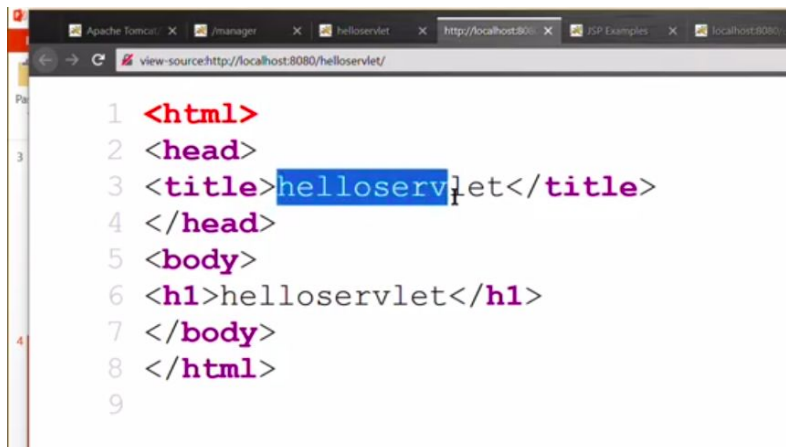
1. As soon as we get the information in HelloServlet.java, we start printing it out in the same style as in CGI.
2. The line by line output (out.println lines) clutters and dominates the contents of the file.

This is resolved by HelloTem.java. In HelloTem.java:

1. Templates are written as string(line 33 in HelloTem.java in figure above),where the string literals are written with "+" to concatenate them.

Note: We are writing String template in this style because Java does not have multi-line string literals. In Python and in some of the other languages and I think some of the newer versions of Java, you can write a single string that spans multiple lines and so we will be able to write all of this without having to add little string literals.

2. These string literals are markup like `<html>`, `<head>`, `<title>`, etc.



```
1 <html>
2 <head>
3 <title>helloworld</title>
4 </head>
5 <body>
6 <h1>helloworld</h1>
7 </body>
8 </html>
9
```

3. Figure above is the html printed in response, by HelloTem.java. The highlighted “helloworld” between `<title>` `</title>` and similarly between the markups `<h1>` and `</h1>` is the data part of the html file. These are not directly written in string “Template” to be printed.

```
generatePageByTemplate(out, params);
}

public void generatePageByTemplate(PrintWriter out,
                                   Map<String, String> params) {
    String template =
        "<html>" +
        "<head> <title> $title </title> </head>" +
        "<body> $body </body>" +
        "</html>";

    StringBuilder b = new StringBuilder(template);
```

4. We divide the problem of printing into two; one is HTML markup and second is the “data” part. The parts where we want to fill up the “data” within the HTML are identified by giving names in a particular way. Look at the above figure. Between the markup `<title>` and `</title>`, we just have `$title` and similarly for body, we are naming the data to be filled between `<body>` and `</body>` as `$body`. `$title` and `$body` are just placeholders for the data to be filled inside these tags.

```

        Map<String, String> params = new HashMap<String, String>();
        params.put("title", "hello template title");
        params.put("body", "hello template servlet body");
        generatePageByTemplate(out, params);
    }

    public void generatePageByTemplate(PrintWriter out,
                                       Map<String, String> params) {
        String template =
            "<html>" +

```

So how is the actual data printed?

5. First, a hash map is to be created as shown in figure above, which is this `Map<String, String> params = new HashMap<String, String>`, a hash map and in this map, we insert key value pairs, (here, `<title, "hello template title">` and `<body, "hello template servlet body">` are the two key-value pairs).
6. Call `generatePageByTemplate(out, params)`.
7. `generatePageByTemplate(out, params)` means *generate a page by using the template*. `StringBuilder b` in figure below, uses the template and using *for* to loop over *keys* of param, attaches value of keys into the template. Eg: wherever `$title` appears in template string, replace the value of key "title" in the hashmap param. Use `out` to print this "template substituted with values in place of \$key" in the last line of code.

```

        params.put("body", "hello template servlet body");
        generatePageByTemplate(out, params);
    }

    public void generatePageByTemplate(PrintWriter out,
                                       Map<String, String> params) {
        String template =
            "<html>" +
            "<head> <title> $title </title> </head>" +
            "<body> $body </body>" +
            "</html>";

        StringBuilder b = new StringBuilder(template);
        for (String key: params.keySet()) {
            String keyform = "$" + key;
            int s = b.indexOf(keyform);
            int e = s + keyform.length();
            b.replace(s, e, params.get(key));
        }

        out.println(b);
    }

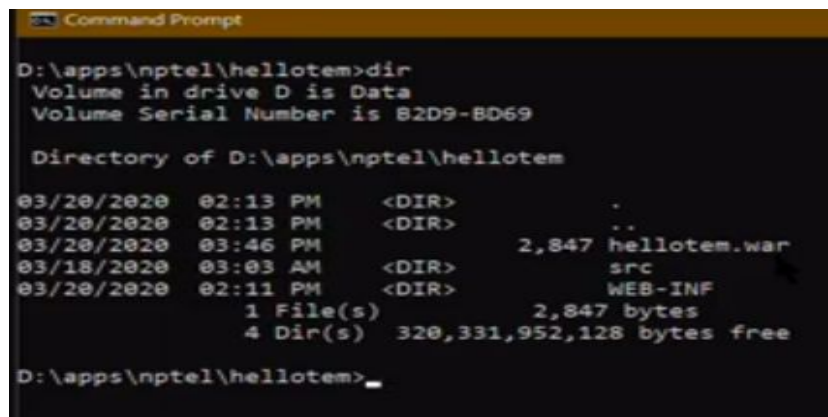
```


In this example, we calculated the length of the value of \$title. indexOf finds the first character where this dollar title appears and then the end character, of course is the start index plus the length of the key and so, from start to end, i.e. we replace \$title with the value, which means, we replace \$title with “hello template title”, whatever it is and similarly for body.

Note: Java string builder which is the standard way to manipulate strings efficiently

To summarize, we leave holes in the template and fill it up by sending the data to be filled in as parameters to generate. So this is straightforward enough and now we can execute this as usual.

Let's do a few things: Go to hellotem directory



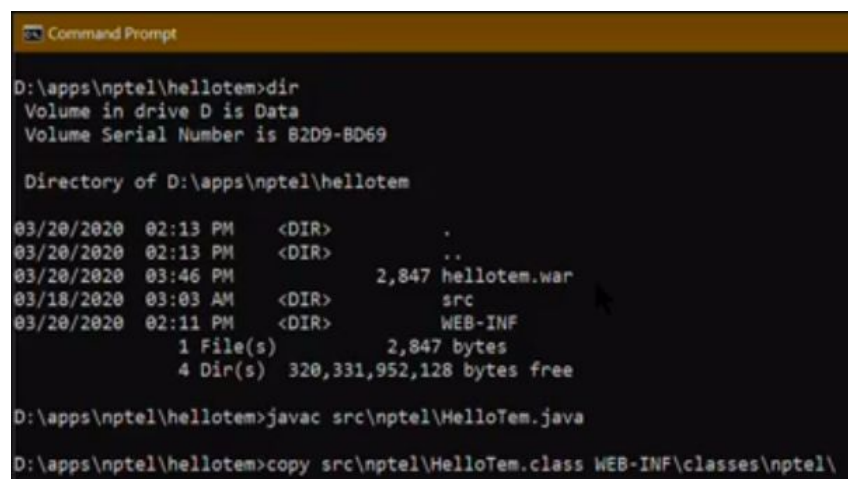
```
Command Prompt
D:\apps\nptel\hellotem>dir
Volume in drive D is Data
Volume Serial Number is 82D9-BD69

Directory of D:\apps\nptel\hellotem

03/20/2020  02:13 PM    <DIR>          .
03/20/2020  02:13 PM    <DIR>          ..
03/20/2020  03:46 PM             2,847 hellotem.war
03/18/2020  03:03 AM    <DIR>          src
03/20/2020  02:11 PM    <DIR>          WEB-INF
               1 File(s)              2,847 bytes
               4 Dir(s)  320,331,952,128 bytes free

D:\apps\nptel\hellotem>
```

Compile HelloTem.java;



```
Command Prompt
D:\apps\nptel\hellotem>dir
Volume in drive D is Data
Volume Serial Number is 82D9-BD69

Directory of D:\apps\nptel\hellotem

03/20/2020  02:13 PM    <DIR>          .
03/20/2020  02:13 PM    <DIR>          ..
03/20/2020  03:46 PM             2,847 hellotem.war
03/18/2020  03:03 AM    <DIR>          src
03/20/2020  02:11 PM    <DIR>          WEB-INF
               1 File(s)              2,847 bytes
               4 Dir(s)  320,331,952,128 bytes free

D:\apps\nptel\hellotem>javac src\nptel\HelloTem.java
D:\apps\nptel\hellotem>copy src\nptel\HelloTem.class WEB-INF\classes\nptel\
```

Assemble to a jar file using jar command:

```
D:\apps\nptel\hellotem>jar cvf hellotem.war WEB-INF
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/nptel/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/nptel/HelloTem.class(in = 2070) (out= 1084)(deflated 47%)
adding: WEB-INF/lib/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/web.xml(in = 681) (out= 298)(deflated 56%)
adding: WEB-INF/web.xml~(in = 695) (out= 295)(deflated 57%)

D:\apps\nptel\hellotem>
```

In real life, people include these in the scripts, but we want to be clear about the entire working and not use too many tools.

Deploy **hellotem.war**.

The screenshot shows the Apache Tomcat Host Manager web interface. At the top, there is a table listing the deployed applications:

URL	Context Path	WAR	Wrapper	Start	Stop	Reload	Undeploy
/helloservlet	None specified	Nptel Servlet	true	0			
/host-manager	None specified	Tomcat Host Manager Application	true	0			
/manager	None specified	Tomcat Manager Application	true	1			

Below the table, there is a "Deploy" section with a yellow header. It contains two sub-sections:

Deploy directory or WAR file located on server

Context Path (required):
XML Configuration file path:
WAR or Directory path:

WAR file to deploy

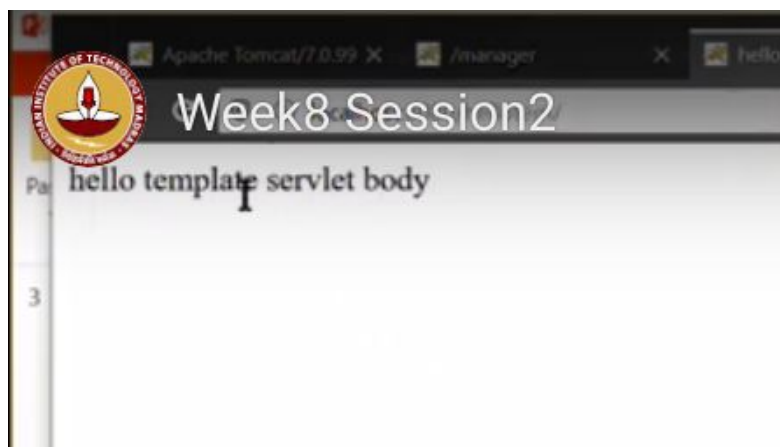
Select WAR file to upload hellotem.war

Open /hellotem in new tab.

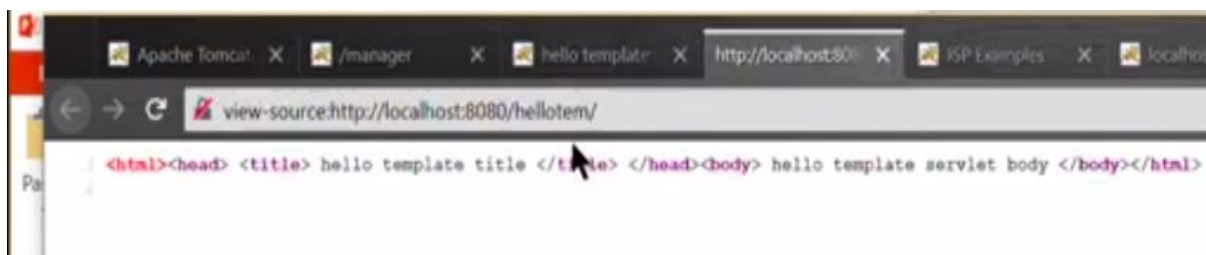


URL	Context Path	Application Name	Enabled	State	Start	Stop	Expire
/docs	None specified	Tomcat Documentation	true	0	Start	Stop	Expire
/examples	None specified	Servlet and JSP Examples	true	1	Start	Stop	Expire
/helloworld	None specified	Nptel Servlet	true	0	Start	Stop	Expire
/hellotem	None specified	Nptel Template	true	0	Start	Stop	Expire
/host-manager	None specified	Tomcat Host Manager Application	true	0	Start	Stop	Expire

Output :



View the source of this html file:



Note that since String template in HelloTem.java didn't have newline character, we see that the html file is printed in a single line in above figure.

JSP makes the above process of creating html file to be printed using templates much slicker than what we have done with this little handcrafted template.

```

public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws IOException, ServletException
{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    Map<String, String> params = new HashMap<String, String>();
    params.put("title", "hello template title");
    params.put("body", "hello template servlet body");
    generatePageByTemplate(out, params);
}

public void generatePageByTemplate(PrintWriter out,
                                   Map<String, String> params) {
    String template =
        "<html>" +
        "<head> <title> $title </title> </head>" +
        "<body> $body </body>" +
        "</html>";

    StringBuilder b = new StringBuilder(template);
    for (String key: params.keySet()) {
        String keyform = "$" + key;
        int s = b.indexOf(keyform);
        int e = s + keyform.length();
        b.replace(s, e, params.get(key));
    }

    out.println(b);
}

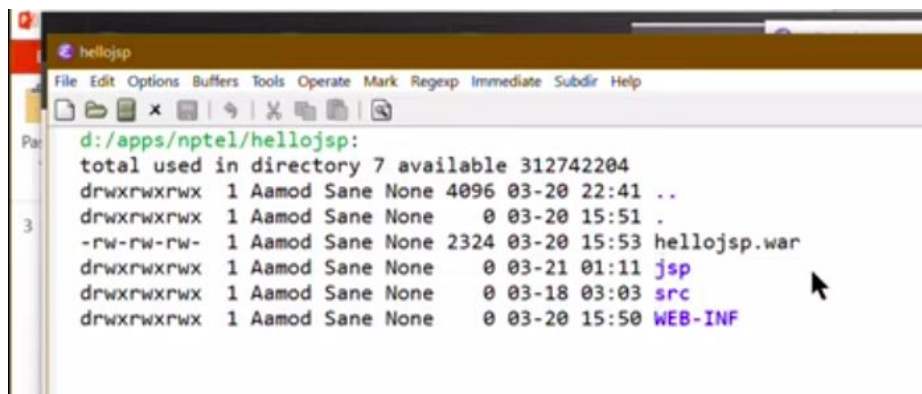
```

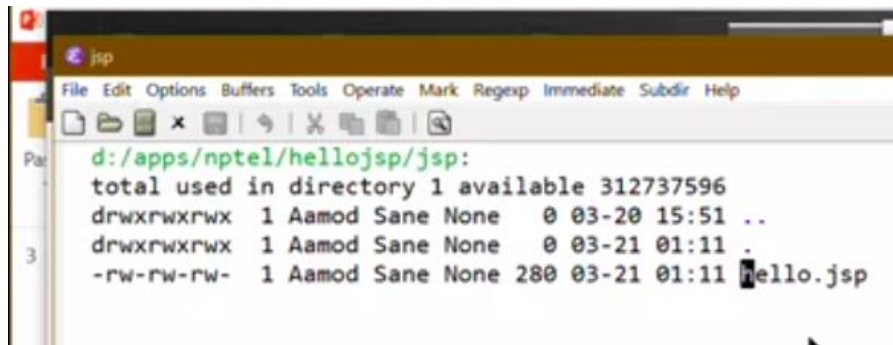
To summarize: In `HelloTem.java`, the `doGet (. .)` has a template and a hashmap and we fused the two in `b` and printed it.

Java Servlet Pages

Go to `nptel/hellojsp/src/nptel/HelloJSP.java`. The template is nowhere to be seen in this.

The template is in a new directory.





```
d:/apps/npTEL/hellojsp/jsp:
total used in directory 1 available 312737596
drwxrwxrwx  1 Aamod Sane None    0 03-20 15:51 ..
drwxrwxrwx  1 Aamod Sane None    0 03-21 01:11 .
-rw-rw-rw-  1 Aamod Sane None 280 03-21 01:11 hello.jsp
```

The above figure shows the new directory. The directory structure for our new system has hellojsp.war which has src and WEB-INF and here we have a **new directory called hello.jsp**, in which there is a **file called hello.jsp**.

(Refer Slide Time: 13:39)

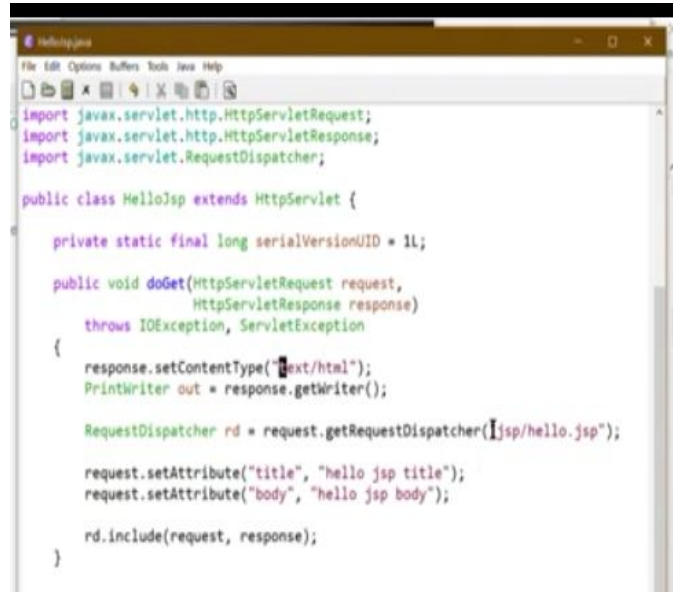


```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" %>
<!DOCTYPE html>
<html>
<head>
    <title> <% request.getAttribute("title") %> </title>
</head>
<body>
    <% request.getAttribute("body") %>
</body>
</html>
```

1. This markup looks much nicer than the markup created using a Java template inside the code.

2. The code here says: There is <html>, <head>, <title> etc., but this time, title and body are generated by snippets of the form : <%= request.getAttribute("markup_name") %>.

(Refer Slide Time: 14:30)



```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.RequestDispatcher;

public class HelloJsp extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        RequestDispatcher rd = request.getRequestDispatcher("jsp/hello.jsp");

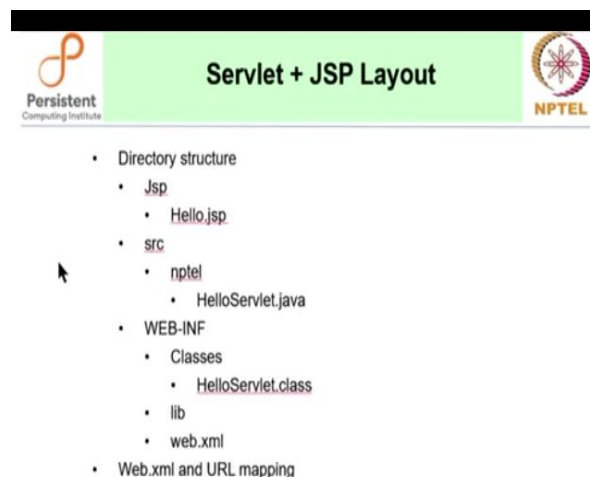
        request.setAttribute("title", "hello jsp title");
        request.setAttribute("body", "hello jsp body");

        rd.include(request, response);
    }
}
```

3. HelloJSP.java, as shown in figure above, takes the request object, using a standard API called `setAttribute()` sets key-value pairs.

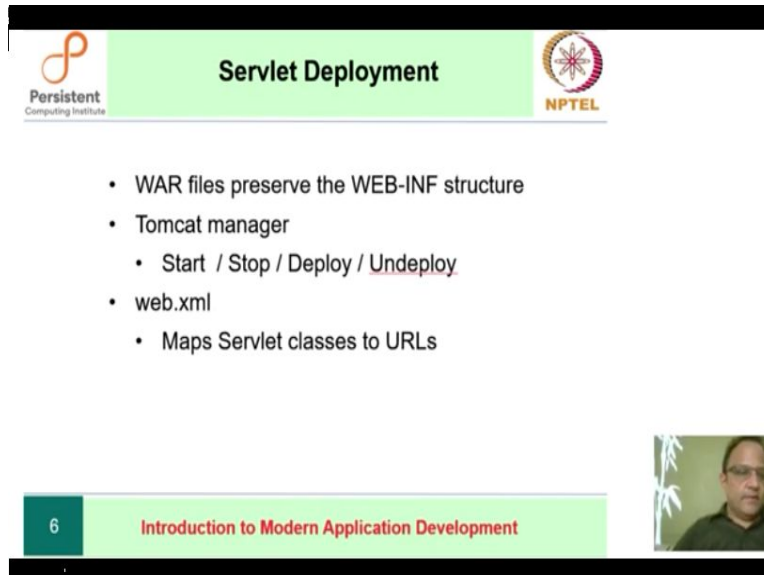
4. Associated with each request object, there is a `getRequestDispatcher` and a Request Dispatcher basically says what to do with the request object. Here, RequestDispatcher says: "do the next request of `jsp/hello.jsp`".

(Refer Slide Time: 18:00)



To summarize: We are going to maintain the above structure and add a JSP directory which has hello.jsp. This is a **servlet plus JSP layout**.

(Refer Slide Time: 18:36)



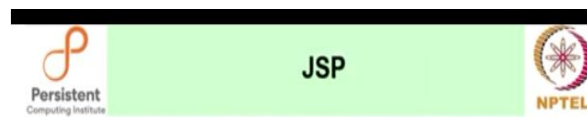
The slide is titled "Servlet Deployment" and features the Persistent Computing Institute logo on the left and the NPTEL logo on the right. The content is as follows:

- WAR files preserve the WEB-INF structure
- Tomcat manager
 - Start / Stop / Deploy / Undeploy
- web.xml
 - Maps Servlet classes to URLs

In the bottom right corner, there is a small video inset showing a man speaking.

6 Introduction to Modern Application Development



(Refer Slide Time: 18:38)



The slide is titled "JSP" and features the Persistent Computing Institute logo on the left and the NPTEL logo on the right.

- Basics of JSP
- How JSP enables data-driven web pages


(Refer Slide Time: 18:40)

 **JSP Key ideas** 

- Instead of Code generating Text
 - `HelloServlet`: explicit output
 - `HelloTem`: replacing placeholders
- Have Code exist within text
- The "View" part is made explicitly into a JSP file
- JSP is HTML interspersed with Java Code
- JSP's are compiled into Servlets
- JSP's can cooperate with Servlets

8

Introduction to Modern Application Development



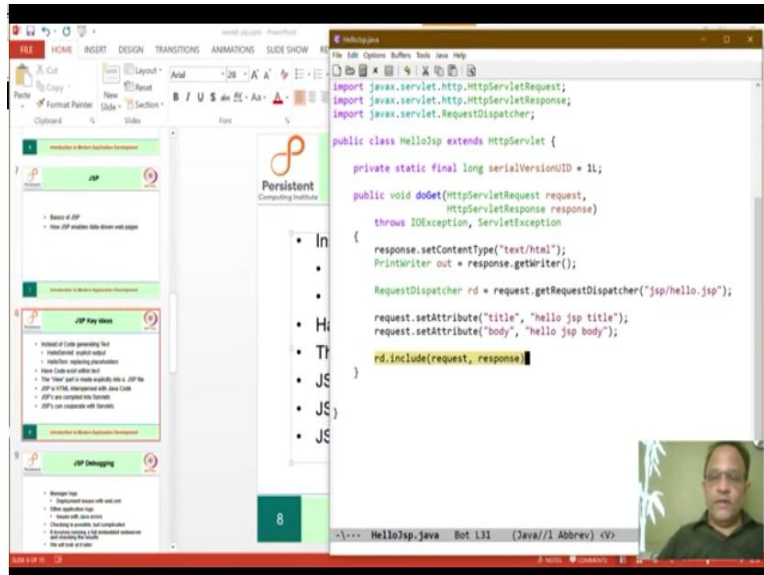
Transition in development style of Servlet Programs:

1. There are two ways of code generating text: one is via the style we used in `HelloServlet`, which is explicit output versus the second way, we did it which is with templates. So `HelloTem` which generates output by replacing placeholders, but it is still code.
2. Instead of having code which is generating text, we reverse the relationship; we have code existing within text and that is what this is doing.

It is taking the text as primary and putting in little bits and pieces of code(included within different kinds of signs: %, >, < and =). So code exists within text.

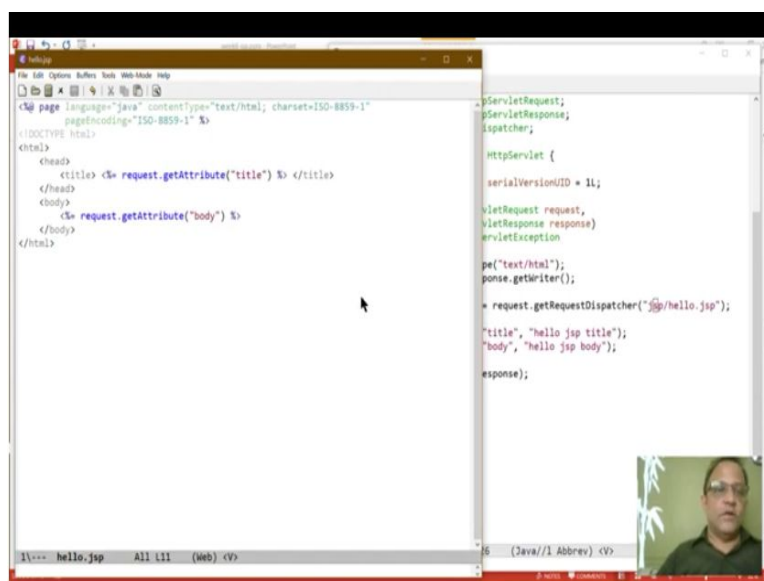
View part of the application i.e. what the user sees, is now explicitly made into a template JSP file.

(Refer Slide Time: 20:06)



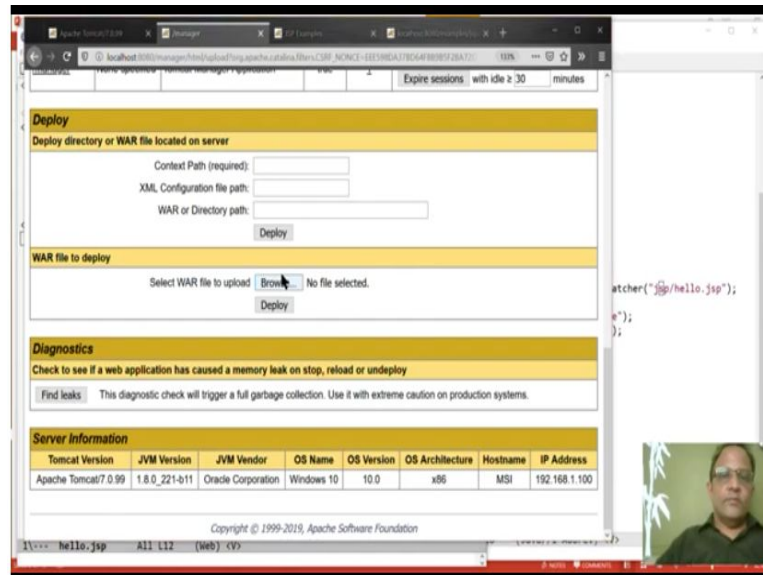
This is clearly just the view. Nothing to do with the content. Some content will be there; we do not know what it is and this is the interesting part: what happens is that, internally and we will see this in a minute JSP which is HTML interspersed with java code is actually compiled into a servlet and therefore, it can cooperate with a servlet. So, the reason that we can say include or in some sense, send forward or send the request to the JSP file and get its result and include that in the overall request. That is what is going on.

(Refer Slide Time: 20:54)

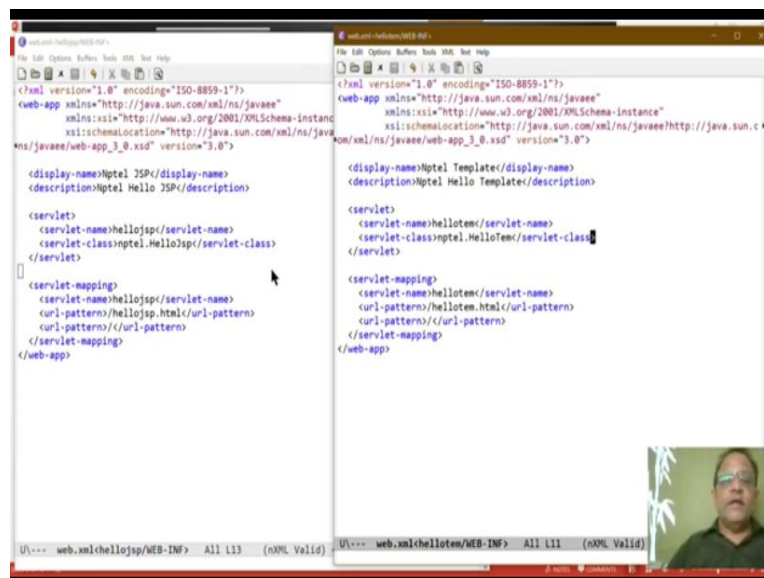


How this happens is like this; let us first make sure that. Let us first run it and then let us see what happens.

(Refer Slide Time: 21:11)



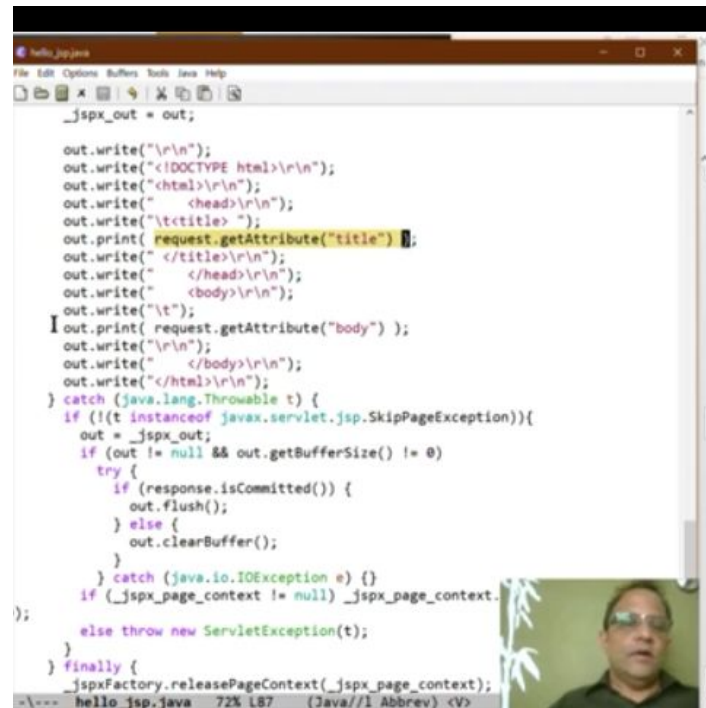
Compile the file: “javac HelloJsp.java”, we are now going to copy. (Refer Slide Time: 22:18)



Take a look at what the web.xml says: HelloJsp corresponds to the class HelloJsp as usual, but internally. So externally this looks not too different from what we have for helloworld.

But in reality we are going to make an internal call to the JSP file. This difference will start showing up in web.xml after deploying. (Refer Slide Time: 23:35)





```
hello_jsp.java
File Edit Options Buffers Tools Java Help

_jsp_out = out;

out.write("\n\n");
out.write("<!DOCTYPE html>\n\n");
out.write("<html>\n\n");
out.write("  <head>\n\n");
out.write("    <title> ");
out.print( request.getAttribute("title") );
out.write("  </title>\n\n");
out.write("  </head>\n\n");
out.write("  <body>\n\n");
out.write("    ");
out.print( request.getAttribute("body") );
out.write("\n\n");
out.write("  </body>\n\n");
out.write("</html>\n\n");
} catch (java.lang.Throwable t) {
    if (!(t instanceof javax.servlet.jsp.SkipPageException)){
        out = _jsp_out;
        if (out != null && out.getBufferSize() != 0)
            try {
                if (response.isCommitted()) {
                    out.flush();
                } else {
                    out.clearBuffer();
                }
            } catch (java.io.IOException e) {}
        if (_jspx_page_context != null) _jspx_page_context.
    }
    else throw new ServletException(t);
} finally {
    _jspxFactory.releasePageContext(_jspx_page_context);
}
-- hello_jsp.java 72% L87 (Java//l Abbrev) <V>
```

Visit `helloJsp`. For the first time, the server itself, which is Tomcat, creates **hello_jsp.java**. This is the translated version.

It converted our nice template in **hello.jsp** to **out.write(..)** statements.

All the unnecessary tedious work is done for us. It takes the strings as they are and interpolate it with a template except, in JSP, it is replaced on the fly and instead of compiling it, we actually had to call a replace method which does replacement at runtime.

We do not have nice separation between what you see in the view and what you see in the model, which is only this much.

So that is the core idea of JSPs.