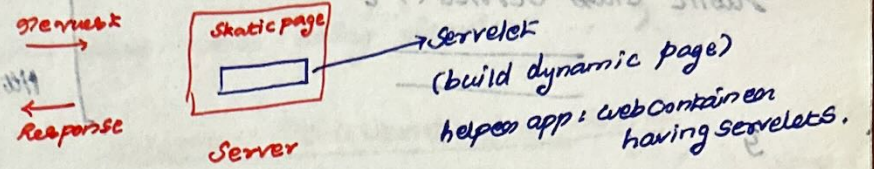
Tomcat (Apache)

* Servelet Container & web server from Apache.
* Standalone / used along with trad. web servers (Apache httpd)
* Request, response : serve webpages (both static & dynamic)

eclipse supports tomcat integration



* web container: Tomcat (goes to respective Servelet)
* deployment descriptor: which request calls which Servelet (mapping) — Configure (web.xml)

---

web.xml

1. Servelet tag (class name) : normal class extending HttpServelet
2. Servelet mapping tag (URL pattern)

↓ Take, process, Respond!

Note: Response (Server → client machine) : Response Object (format)

XML (avoid by using annotations) : map using annotation.

@WebServelet ("/abc.html")

Server side language: Java → Servelet

Ports

Sudo ss -lntu

Exercise 1

<form action="" → class name!

Enter num 1 : <input type="text">

Enter num 2 : "

</form>

Create a Servelet

Class :

★ Com.Company

Javax.servlet.http.HttpServlet
// Request
// Response

```
Public class AddServlet extends HttpServlet
{
    Public class Service() {
    }
}
```

Service()
↓
HttpServletRequest req,
HttpServlet_Response res

```
int i = req.getParameter("num1");
```
→ parseInt

name attribute in URL as parameter → HTML

Flow:

Submit → Tomcat → web.xml (configured) [→ deployment descriptor]

---

web.xml

```
<servlet>
    <servlet-name>  </servlet-name>
    <servlet-class>  <. . .>
</servlet>

<servelet-mapping>
    <servlet-name>  </. . .>
    <servl-pattern>  </url-pattern>
</. . >
```

Print in client page. (Response)

```
res.getWriter().println("Result : " + k);
```

GET and POST

HTTP → GET → fetch
HTTP → POST → submit
HTTP → PUT
HTTP → DELETE

<form action="add" method="get">

Servlet: Control methods (say: only post) → doPost (req, res)
→ doGet

→ doPost / doGet → runs using service(....)

2 Servlets – Call Servlet from Servlet

* Call Servlet → Request dispatcher
  → Redirect

* Send data b/w servlets → Session management

Call a Servlet:

1. Set request (req.setAttribute("k", k);
2. RequestDispatcher rd = req.getRequestDispatcher("sv");
3. rd.forward(req, res);

```
request.setAttribute("num", c);
RequestDispatcher rd = request.getRequest
                       Dispatcher("square");

rd.forward(request, response);
```

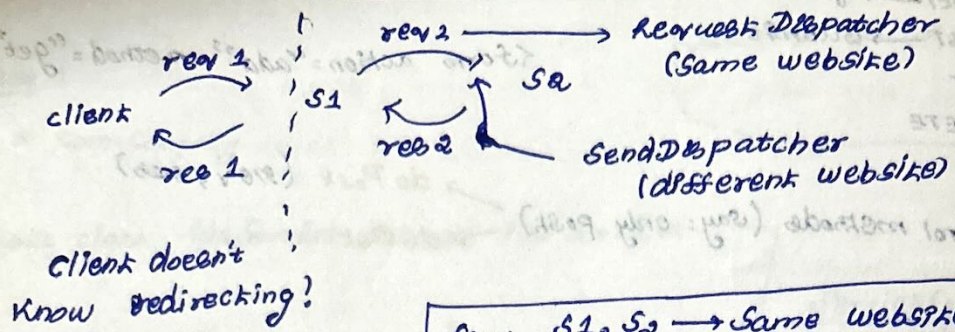
Response & Request object

* Request object → hold requesting values.
* Response object → send videos, Html etc..

web world: JSON

Provided by Tomcat!
(Interfaces!)

RequestDispatcher & SendDispatcher

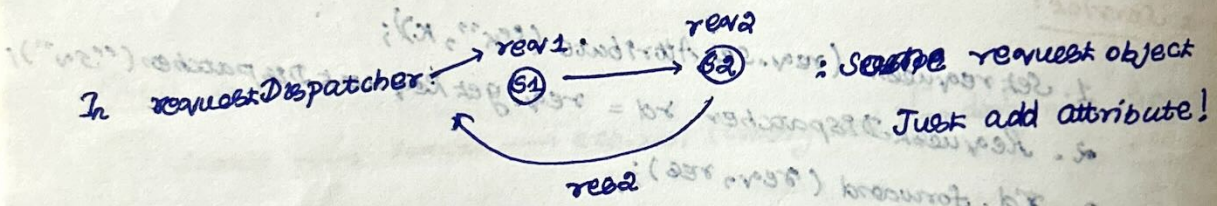


Say: S1, S2 → Same website

* when S1, S2 : different websites!

1. request to S2 : done from browser (redirected = visible)
(SendRedirect)

S1, S2 → Same website: RequestDispatcher
→ diff (say: website 1, paypal): SendRedirect.



: same request object
Just add attribute!

2) In SendRedirect : two different objects
(Session Management)!

SendRedirect

* we don't reuse objects!

res.sendRedirect("S2");

won't work why?

In Servlet2:

getParameter("k"); → null!

So need to set k / pass as URL!

AddServelet :

```
int k = i+j;
res.Redirect ("Sq");
```

SquareServlet:

```
int num = req.getParameter("k");
```

↓ null!

Solutions

↙ AddServelet : req.setAttribute("k",..) (Setting parameter) won't work! Since they can't share request objects!

↘ res.Redirect("Sq?k=" + k); URL rewriting!

→ SendRedirect → Just redirects (no object sharing)!

★ Say: we don't want URL writing! (Is there any other way?)

1. pass multiple values
2. To multiple Servlets (maintain data throughout Sessions)
3.

webapp (Tomcat): maintains session

Use Session in multiple servlets

→ Session object : given by HttpServlet

```
HttpSession session = req.getSession();
Session.setAttribute("num", value);
res.sendRedirect("Sq");
```

getHold of it

I want num to be added.

At another Servlet:

```
HttpSession session = req.getSession();   → get hold of it
int k = (int) session.getAttribute("k");  → I want "k"
```

→ Session is throughout available

→ Say I don't want a data : we can delete attribute!

→ session.removeAttribute("k");

eg: login (after) : remove the data!

Cookies

Cookie: each request, response (have cookie)

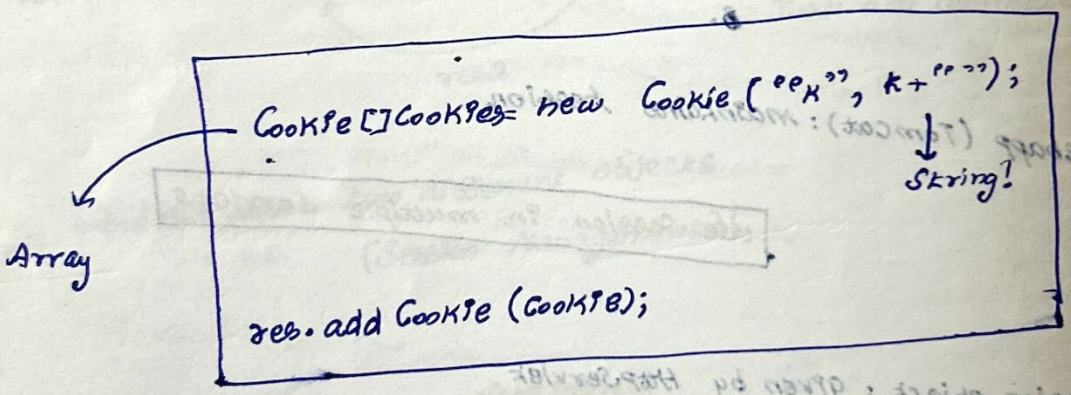
Server sends the same cookie (Server modifies) - Sends

eg: * Shop keeper need to give me 50 rs : he may forget it (token)

* go back, show token (cookie)! (meta data)

Note: Cookie will be in response

→ req.getCookies() [may be multiple cookies] in client side.



many cookies: Send by Client:

```
for (Cookie c : Cookies){
    if (c.getName().equals("k")) {

    }
}
```

Servlet Config & Servlet Context (Interfaces)

* get initial values for the servlet/application:

web.xml [mention initial parameters, use them in code]

How? <Context-param> tag

```
<context-param>
    <param-name> name </param-name>
    <param-value> Navin </param-value>
</context-param>
```

Key, value pair.

use Context-param in Code:

ServletConfig → for a particular servlet

ServletContext → for all servlets common.

Servlet Context

```
<context-param>
    <param-name> name </param-name>
    <param-value> Navin </param-value>
</context-param>
```

↓

Not inside any servlet!

Now in any servlet:

```
response.getWriter().print("hello");
ServletContext con = getServletContext();
con.getInitParameter("name"); → Navin
```

Servlet Config

```
<servlet>
    servlet-name
    servlet-class
    <init-param>
        <param-name> —
        <param-value> —
    </init-param>
</servlet>
```

↓

only for that servlet!

```
ServletConfig cg = getServletConfig();
cg.getInitParameter("name");
```

↳ Sukesh

↓

More powerful! (local)

Servlet annotations

→ XML (up to now)

→ Annotation (makes easy)!

On top of Servlet class:

@WebServlet ("/sa") → URL

@WebServlet ("add")

Why JSP

Customize (html) : client page!

```
out.println ("<html> <body bgcolor = 'cyan' > ");
out.println ("output : " + k);
out.print ("</body> </html>");
```

↓

Tedious - problem!

dev & designers → different!

JSP feature (Jakarta Server Pages)!

Java code inside HTML!

↓

add.JSP

<%] delimiters!

%>

<%@ language = "java" contentType = "text/html; charset = UTF-8" pageEncoding = "UTF-8" %>

<!DOCTYPE html ... PUBLIC "-//W3C//DTD ... >

html

head

<body bgcolor = "cyan" >

<% int i =

```
<%
    int i = Integer.parseInt(request.getParameter("num1"));
%>
```

JSP - Java inside html

JSP

→ Normal Servlet: Servlet (create), method (service), object of printWriter, fetch data from request!

→ JSP: All objects default! (Simpler)

only write JSP, can't run it : Servlet runs on tomcat (Servlet Container)

* In Tomcat: only run servlet!

JSP → Converted Servlet

use: easy to write!

Conversion →

JSP

demo.JSP

```
<html>
<body>
<%
    int i = ——
    int j = ——
    int k = i+j;
    out.pri..(k);
%>
</body>
</html>
```

Servlet

```
Public class Demo extends HttpServlet {
    Service (req, res){
        int i = ——
        int j = ——
        ——
    }
}
```

Provided by Tomcat

copy paste!

"easily convertible"

JSP gives implicit objects

→ Converted to explicit in servlets.

<% ... %>] Scriptlet (whatever here goes to Servlet service() method)!

Problem: Code outside service()

Solution: <%! ... %>] declaration section

Say: Variables, methods.

'Inside class, outside service()'

Import a package: (outside class)

<%@ page import = "java.util.Date" %>

- page: directive
- import: attribute
- "java.util.Date": package name (comma seperated)

<%= K %> → out.print(K) (expression)

Tags

1. Directive (import) <%@ page import = "" %>
2. Declaration (outside service inside class) <%! %> <% %>
3. Service() (inside): Scriptlet
4. Expression [print]: <%= %>

Create a page: JSP (better)

Just Processing: Servlet (better)

Practical (netbeans) : Java EE

→ Note: multiple scriptlets <% %> allowed! (all goes inside service())

→ <%! %> ⟶ outside service & inside class [declarative]

→ <%@ page import="java.util.Scanner" %> ⟶ import → directive!

→ <%-- --%> ⟶ Comments

→ <%= %> ⟶ Expression (just print)

My fav number : <% out.println(coef); %>

Jsp way: My fav number : <%= coef %> ⟶ print value

JSP directives

@page ⟶ import package, language (for entire page) : attributes: values

@Include (other jsp files ⟶ include to current)

@ taglib (external tags apart from html tags) : Spring framework!

attributes : values

1. language = "java" ⟶ Scripting lan
2. extends = "className" ⟶ extend some class (∴ using JSP)
3. import = "importList – with commas"
4. session = "true/false" ⟶ we are using sessions?
5. autoFlush = " [buffer : send before filled)
6. ContentType = "content type info"
7. errorPage = "error_url"
8. IsErrorPage = "true/false"
9. info = "info about page"
10. isELIgnored = "true/false"
11. isThreadSafe = "true/false"

<%@ . . . %>

→ <%@ include file="header.jsp" %> ⟶ write header & footer once use it everywhere

@ Taglib

<%@ taglib uri="uri" prefix="fx" %>

↳ use tags like fx:tag1, fx:tag2] tag1 part of fx!

(interpretation)

Implicit objects in JSP

* used in scriptlet & Expression

request (HttpServletRequest)
response (HttpServletResponse)
PageContext (PageContext)
Out (JSPWriter) ~ PrintWriter Object
Session (HttpSession)
Application (ServletContext)
Config (ServletConfig)

In Servlet we need to create lot of objects/ references! (response, request)

JSP makes Job easy

1)

```
<%
    request.getParameter("name");
%>
```

2) PageContext.setAttribute("name", "navin")

↓ refers to page attribute

Session Scope: multiple Servlet
PageContext: same page scope (set & get values)
request scope: current page & page requesting!

```
PageContext.setAttribute("name", "name", PageContext.SESSION_SCOPE);
```

↓

Session, Servlet Context, Current Scope...

→ config.getParameter("name") [Retrieve]

JSP Exceptions

```
<%
    try{
        int k = 9/0;
    }
    catch(Exception e){
        out.println("Error"+e.getMessage());
    }
%>
```

Good idea → Seperate error page!

divByZeroError.jsp

```
<body>
    Error!
</body>
<%@ page ... isErrorPage = "true" %>
<%= exception.getMessage() %>
```

handle exception!

home.jsp

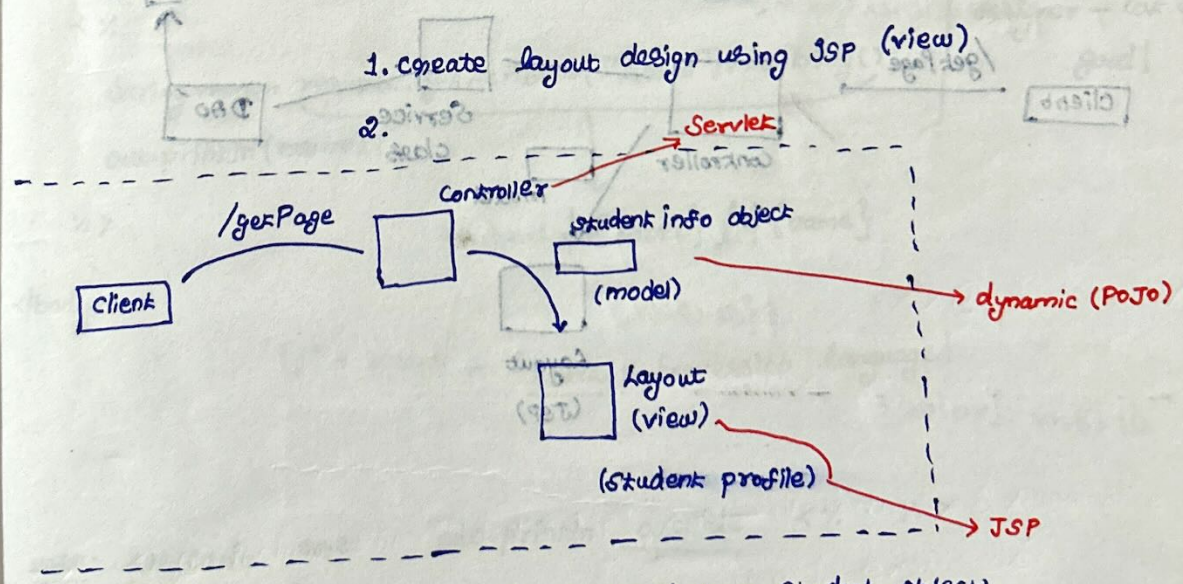
```
<%@ page language = "java" ...
    errorPage = "error.jsp" %>
<%
    int k = 9/0;
%>
```

MVC – Model View Control design

* Issue with JSP: first time compile time (JSP slower)
* Don't write business logic inside JSP (actual java code)
* go for servlet

Use JSP only for view

→ Using HTML: only static pages | JSP: generate dynamic page



POJO – plain old Java object (Student class → Student object)

Any processing: must be happen in Controller!

NTIER architecture: divide application into logical & physical layers.

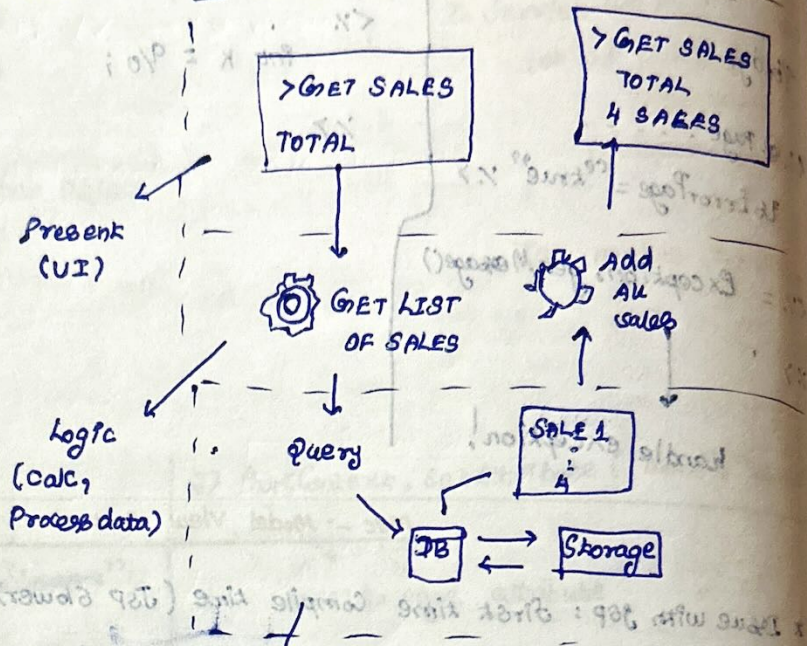
higher layers uses services of lower layer (not the other way around)

Software → Processing
→ data management
→ presentation

Physically & logically seperated (different machines..)

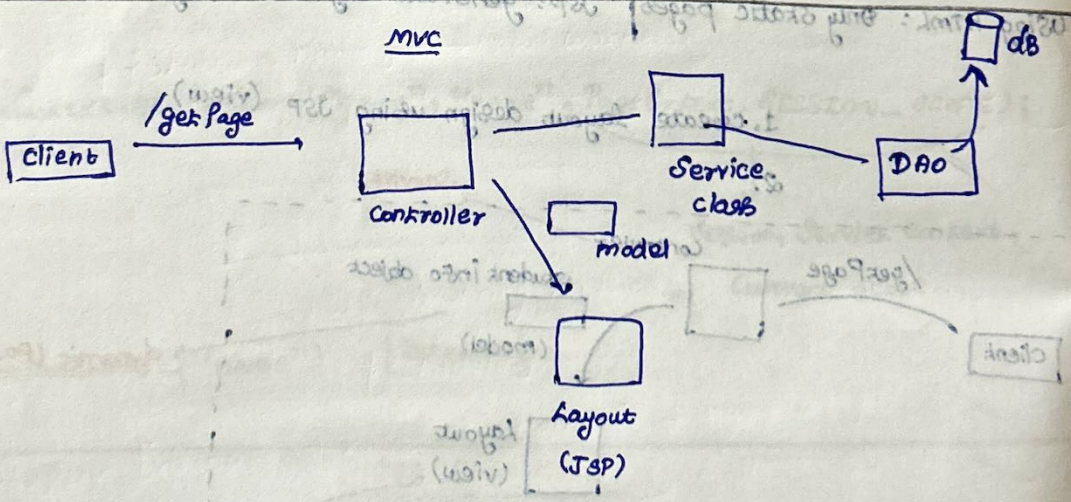
N → any number from 1 (N-TIER)

1. Logic tier
2. Presentation tier
3. Data tier



Benefits:

1. Secure, manage, Scale, flexible
2. Efficient development
3. Add new feature, reuse



★ Controller: don't write JDBC directly. [Controller: get response, Send response]
Don't write methods, operations!

★ use Service classes! (Future: change dB → change service (not controller))

DAO: Not compulsory in MVC

MVC → Controller, Layout, model

→ DAO: Data access object (Interact with database)

JSTL - JSP std. Tag library

*designer: Not comfortable with Java

*Convert Java to HTML alike code!

DemoServlet.java

```
@WebServlet("/DemoServlet")
public class DemoServlet extends HttpServlet{
    Protected void doGet(req, res) throws ServletException, IoException {
        String name = "Hari";
        req.setAttribute("name", name);
        RequestDispatcher rd = request.Dispatcher("display.jsp");
        rd.forward(request, response);
    }
}
```

display.jsp

```
<body>
<%
    String name = request.getAttribute("name").toString();
    out.println(name);
%>
</body>
```

designer – not feels good!

${label} / ${name}

JSTL: Expression language!

note: we can't use out.println outside <% %>

Solution: JSTL!

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

prefix → anything!

```
<c:out value="Hello world" />
```

Jstl.Jar | mySql Connector.Jar

${label} ⟶ 2nd version

<c:out value="${label}" /> ⟶ 1st version

<c: catch . . . >

<c: import url="http://teluskо. com" ></c: import>

<c: set ... > ⟶ Set variable, scope

<c: remove . . >

Student.class → name
→ roll no.

request.setAttribute ("student", S); → Student object

${student} ⟶ prints hashcode!

${student.name}

↳ error!

For objects: use getter & setters!

(automatically gets that : JSP)

${student.name} ⟶ now works!

Print entire object

toString method

${student}

"Student [rollno=" + rollno + ", name=" + name + "]";

Array

List<Student> students = Arrays.asList (new Student (), "A"), ...);

request.setAttribute ("students", students);

${students}

↳ prints all with comma!

∴ Student has toString()!

[Student [rollno=1, name="A"], . . .] → ${students} as list.

forEach

```
<jstl:forEach items="${students}"
              var="student">
    ${student}<br>
</jstl:forEach>
```

JSTL SQL tags

→ Connect with database!

```
<body>
<sql:setDataSource var="db"
    driver="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3308/mario"
    user="root"
    password="____" />
```

(var: variable we're going to refer)

```
<sql:query var="rs" dataSource="${db}">
    SELECT * from gadgets
</sql:query>

<c:forEach items="${rs.rows}" var="row">
    <c:out value="${row.gid}"></c:out>
    : <c:out value="${row.name}"></c:out>
</c:forEach>.
```

(store in rs (result set))

Note: for sql tags:

```
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>

<%@taglib prefix="c" uri=". . . /core" %>.
```

Powerful JSTL.

Step 1: mention taglib

Step 2: use tags with prefix!

Functions of Java:

<%@ taglib prefix="fn" uri="..../functions" %>

${fn:length(str)}

<c:forEach items="${fn:split(str,' ')}" var="s">

${fn:length(s)}

</c:forEach>

→ index: ${fn:indexOf(str,"is")}

${fn:contains(str,"Java")}

<c:if test="${fn:contains(str,"JSP")}">

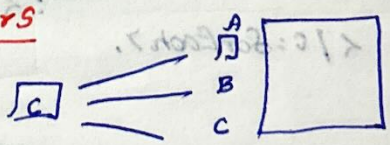
</c:if>

Single quotes

fn:endsWith(...)

${fn:toLowerCase(...)}

Filters

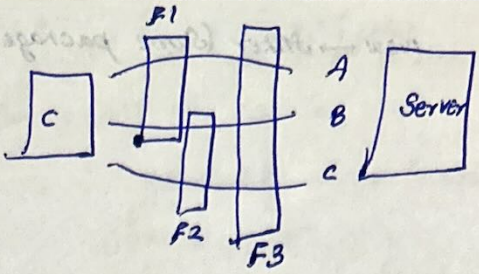


★ Why?

1. Requesting multiple servlets
   1. maintain log (A, B)
   2. maintain transaction (B, C)
   3. Security (A, B, C)

Make as module — Seperate!

(plug): Easy to Scale!



new D: easy to plugin features

F1 - log, F2 - transaction, F3: security!

Call A: Filter F1 → Filter F3 → go to A.

Any filters can return response! : don't allow further!

→ Note: F3 doesn't know where request is from (F1/F2). - Individual

→ Tomcat: chain multiple filters! (done by Tomcat: F1 doesn't know F3)

Configure in web.xml
which request which filter!

Create filter:

1. Extend ServletFilter

methods — init, doFilter, destroy — rare

(req, res)

Filter: like Servlet but not exactly (intercept request)

Configure individual filters (like ServletConfig)

↳ FilterConfig

validation: filters!

Id, Name — check values, cookies, logged in?

Filters: pluggable

1. Filters (independant): call next one (no need to mention which one)
2. Servlet (must mention next Servlet to call)!

∴ we are doing chaining!

new → filter (Same package as Servlet)

IdFilter : /addAlien

call!

```
Public Class IdFilter implements Filter{

    Public void destroy(){
    }
    public void init (FilterConfig fConfig) throws ServletException
    {
    }
    public void doFilter(Req, res, FilterChain chain) ——
    {
     chain.doFilter(request, response);
    }

}
```

Filter chaining!

webservers: Http (stateless protocol)

maintain session (management)

Login using Servlet & JSP

*Session management!

abc.com/login.JSP

abc.com/videos.JSP → enforce login (Session) / cookies (easy)

(ecommerce module)

Cookie: Client (alter, modify)

Save on server side!

(Login: paragon sessions)

HttpSession

Once login (verified): Call Servlet

→ Verify.java (verify credentials)

Redirect

Login → Welcome (Loggedin)

1) abc.com/login.jsp

2) abc.com/videos.jsp → Without login no access.

Redirect

Welcome (not loggedin) → login

Session.setAttribute("","");

put data into Session!

redirect/RequestDispatcher

Login → Verify.java → welcome.jsp / videos

Login → Incorrect!

verify.java

Check Session!

If (session.getAttribute("uname")==null)

Logout.java

session.removeAttribute("uname");

Loggedin

Login.jsp

videos

"do this for every Secure page"

Note: Caching: (browser) → Inspite of reloading → preloaded Content!

Back button → Can't be hacked (ASK to login)

Remove/Stop = Caching

Solution: meta tags

disable back:

disable cache : browser (don't cache)

response.setHeader("cache-control", "no-cache", "no-store", "must-revalidate");

Paste in all pages!

disable back button ⟶ gives bad experience!

Also (http 1.1 ⟶ ok)!

Oldder version ⟶ pragma (clear that too)

```
response.setHeader("pragma", "no-cache");
response.setHeader("Expires", "0"); //proxies
```

---

dynamic website:

* dB/Content management system ⟶ webpage
* client side, Seerver side scripting.

HTTP:

distributed, hypermedia Info systems (client, Seerver communication)

Img, queeries, HTML : TCP 80

1. Exchange data over web. (Request, response)
2. port 80
3. Stateless. (each request — new request)

Client: web browser, Search engine

Server: Servlet.

Request (to Serrer)

1. request
2. Source ip, proxy, port
3. dest ip, protocol, port, host
4. Requested URI
5. Req method & Content
6. Useer agent header
7. Connection control header
8. Cache control header

GET, POST, PUT, DELETE, OPTIONS, HEAD, TRACE

GET ⟶ header, efficient, second req ignored (until 1st responded)

POST ⟶ body, Secure

GET → Bookmarkable
→ Cacheable
→ length restriction
→ Not secure, only to retrieve info

POST → Not bookmarkable
→ not cacheable
→ No length rest.
→ Not retained in history

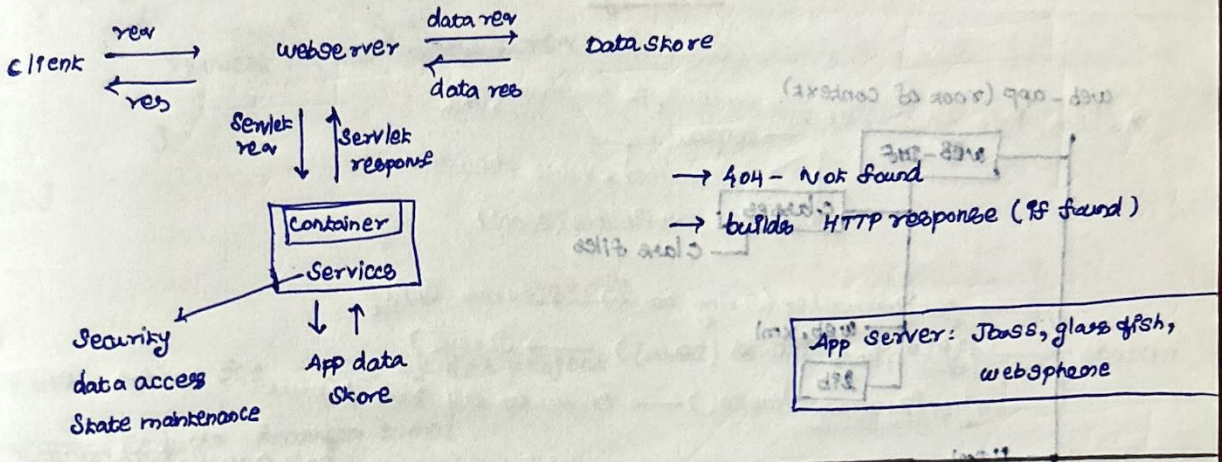
Servlet Container:

* Run time for JavaEE app. (dynamic serving)

Standalone, (Server, servlet → diff programs (plugin))

→ Server: (Container Container) → used for servlet, jsp, struts, jsf ... (Not for EJB)

→ host multiple sites; FTP, email, storage, gaming..



→ HTTP header: (MIME) - multipurpose Internet Mail Extension

→ Internet std. (allow extending sound, img, txt in a message)

Support Non ASCII

multiple attachments - including executables (audio, ..)

Unlimited message length.

Servlet API

Javax.servlet.http (only http classes & interfaces) — Servlet, ServletRequest, ServletResponse

Public void init (ServletConfig config)

Public void service (... req, ... res)

destroy ()

getServletConfig()

getServletInfo()

Life cycle methods (only once)!

invokable.

GenericServlet class: No service method - handle protocol independent request!

Lifecycle

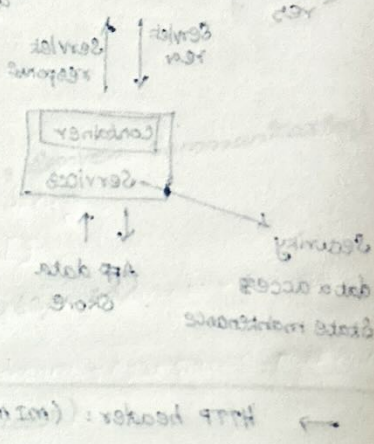
Invoke servlet class
↓
Create servlet instance
↓
Init
↓
Service (repeat)
↓
destroy

(Invoke servlet class, Create servlet instance, Init) — done during 1st request by web container

directory

web-app (root of context)

- WEB-INF
  - classes
    - class files
  - web.xml
  - lib
- HTML
- Static Resources (img, css etc)



Create servlet

- Implement servlet interface
- Inherit Generic servlet class
- Inherit HttpServlet class (widely used) (doGet, doPost..)

```
public class ____ extends HttpServlet{

public void doGet (req, res) throws ServletException, IOException {

        res.setContentType ("text/html");
        PrintWriter pw = res.getWriter();
        pw.println ("<html><body>");
        pw.println ("welcome");
        pw.println ("</body></html>");
        pw.close();
}

}
```

Servlet-api.jar (apache Tomcat)

JRE/lib/ext folder

deployment descriptor (web.xml)

```
<servlet>
   :
</servlet>
<servlet-mapping> .. </servlet-mapping>
```

→ Run TOMCAT server!

Working of servlet

1st time → Create & load Servlet class → Instantiate → init → Service()

next times → Service()

handle req:

map Servlet class → create req, res objects → call Service() → Intentionally

public Service() calls protected Service() → doGet() (default)

Casting:

```
public void service (ServletResponse res, ... req){
    try
    {
        request = (HttpRequest) req;
    }
}
```

War file

→ web archive file : files of web project (used to deploy project)

→ app: Reduces transfer time!

jar -cvf projectname.war *

- * → Include sub directory
- -cvf → archive file name
- -v → generate verbose o/p
- -c → create file

deploy → Server console

→ manual (apache tomcat) → does all things

```
jar -xvf projectname.war
```

web.xml

```
<web-app>
  <welcome-file-list>
    <welcome-file> home.html </welcome-file>
  </welcome-file-list>
</web-app>
```

load-on-startup

*load servlet : deployment (or) server start → +ve value

(Pre Initialization)

Avoid time : during working!

(-)ve value : 1st request load (Consume time)

```
<web-app>
  <servlet>
    <load-on-startup> 0 </loadon-startup>  ⟶ 1st loaded during servlet start
  </servlet>
  <servlet>
    <load-on ... > 1 ... ⟶ loaded second
  </servlet>
</web-app>
```

0, 1, 2, ...

default : start during 1st call of servlet.

ServletRequest Interface

* client info to servlet
  - content type
  - length → URL
  - parameter, value, header
  - info, attributes (JSON)

methods
- getParameter(String name) returns String
- getParameterValues(" ") ⟶ String[]
- getParameterNames() ⟶ enum of all req parameter names

Content length, type, encoding, getInputStream, host name, port number (Read binary data)

Request dispatcher

* dispatch request to other resource (html, jsp, servlet) / Servlet Collaboration

RequestDispatcher
- forward(req, res) ⟶ forward to other servlet/resource
- include(req, res) ⟶ include content (key, value)

```
RequestDispatcher rd = request.getRequestDispatcher("servlet1");
                                                      ↓
                                                  url pattern
rd.forward(req, res);                  (forward to this path resource)!
```

1. index.html (get user i/p)
2. Login.java (verify password) —Forward→
3. Welcome.java (display welcome) (may be jsp)

SendRedirect

⟶ SendRedirect() method from HttpServerResponse()

⟶ redirect to another resource!

forward()

1. Server side
2. Same req, response object
3. With in Server

sendRedirect()

1. client side
2. New req, res objects
3. Both within, outside

(Session management)

response.sendRedirect("http://www.google.com");

Servlet Config

* Servlet Config. Created for each web container's servlet.
* Web.xml : No need to edit servlet

Public String getInitParameter(String name);
Public Enumeration getInitParameterNames();
Public String getServletName()
Public ServletContext getServletContext();

Specific to each servlet

Local

```
<web-app>
  <servlet>
    <init-param>
      <param-name> — </param-name>
      <param-value> — </param-value>
    </init-param>
  </servlet>
</web-app>
```

Servlet Context

1. only one ServletContext object per webContainer — global

getInitParameter(String name)
getInitParameterNames()
setAttribute(obj, object)
name
getAttribute(String name)
removeAttribute(...)

```
<web-app>
  ...
  <Context-param>
    <param-name>
    <param-value>
  </Context-param>
</web-app>
```

global

---

1. request scope
2. Session scope
3. App scope

SetScope
GetScope
removeScope

SetAttribute
getAttribute
getInitParameterNames()

removeAttribute(Name)

→ Context.setAttribute("company", "IBM");

→ Context.getAttribute("company");

ServletContext → Servlet 1, Servlet 2

web.xml

ServletConfig – local

ServletContext – global

Session tracking

* Session: particular interval of time
* Maintain state (data) of user : session management.

Http : Stateless – use session management to maintain state.

each req – new request.

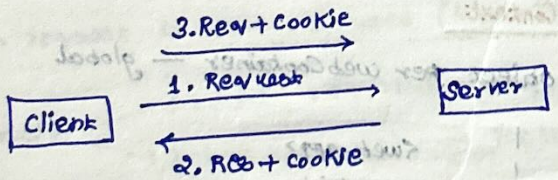
Session tracking techniques:

1. Cookies
2. Hidden form field
3. URL Rewriting
4. HttpSession

Cookie – easy to maintain state, client side

→ Small piece of information : multiple client request

name, single value, attributes, path, domain qualifiers, max age, version number.



Types → Persistance (won't removed: browser closed – only sign out / logout)

→ Non persistance (Removed: browser closed)

disadv: client may mess, only text info, Cookie may disabled.

gmail → uses cookie

Cookie class

Cookie()

Cookie(name, value)] constructor

setMaxAge(int expiry) → Seconds

getName() | setName()

getValue() | setValue()

other methods

getCookies() → Cookie[]

addCookie(Cookie ck) → void

Create cookie: `new Cookie ("user", "Vimal");`

`response.addCookie(ck);`

delete cookie: `ck.setMaxAge(0);`

`response.addCookie(ck);`

get Cookie: `Cookie ck[] = request.getCookies();`

```
for(....) {
    ck.getName();
}
```

eg: Store user info → Cookie. → display it in the page!

Login & Logout

Login, Logout, Profile.

index.html, link.html, Login.html, LoginServlet.java, LogoutServlet.java, ProfileSe...java.

web.xml

index.html	link.html	Login.html (post)
Login	Login	username
Logout	Logout] 2 classes!	Password.
Profile	Profile]	

Login (success) → addCookie → Profile (Logged in?) → check Cookie!

Loggedout → clean cookie.

---

2. Hidden form field

* Store in hidden field : invisible text field

use: submit form in all pages & we don't depend on browser

`<input type="hidden" name="uname" value="Vimal">`

App: Content form website (page id, name : each page uniquely identified)

* No need for cookies
* extra form submission
* maintained at server
* only textual

] disadv

---

3. URL rewriting

* Append token inside URL of next servlet (parameter):

url ? name = value & age = value & ??

Adv: no cookies, No form submission req!

disadv: only work with links, only textual info

out.print("visit");

↳ use that parameter in next one!

4. HttpSession Interface

1. Container creates a session id for each user.

HttpSession → bind objects
→ View, edit info of a session (identifier, time of creation, last accessed time)

Constructors

getSession()

getSession(true) → no current session, is true create one!

methods

getId()
getCreationTime()
getLastAccessedTime()
Invalidate()

Servlet 1

1. getAttribute/parameter : read object
   create session, setAttribute

Servlet 2

1. getSession
2. getAttribute → null (not there!)

↓

Login, Logout, Profile!

Event Listeners (Javatpoint)

*occured (trigger)! - Count total, current Logged in users, create tables at project deployment, create dB connection object.

classes

1. ServletRequestEvent
2. ServletContextEvent
3. ServletRequestAttributeEvent
4. ServletContextAttributeEvent
5. HttpSessionEvent
6. HttpSessionBindingEvent

Interfaces

1. ServletRequestListener
2. ServletRequestAttributeListener
3. ServletContextListener
4. ServletContextAttributeListener
5. HttpSessionListener
6. HttpSessionAttributeListener
7. HttpSessionBindingListener
8. HttpSessionActivationListener

ServletContextEvent

1. Notified when webapp deployed on server

Use: Trigger: Create dB, tables — at deployment

constructor: getServletContext() → method

ServletContextListener → contextInitialized(ServletContext e) : Invoked when deployed
→ contextDestroyed(") → undeployed.

Implement this Interface, define method
↓
Create mysql connection
add attribute (Session)
↓
use everywhere.

Webapp undeployed → destroy Connection!

HttpSessionEvent, HttpSessionListener

when session created: sessionCreated()
destroyed: sessionDestroyed()
] Set a Session attribute
↓
use: total logged in users!

Req: triggers

Servlet Filter

*pre/post processing — Conversion, log, Compression, encrypt, decrypt, I/p validation

Pluggable (remove web.xml entry - remove filter) — No dependency!

Interfaces

Filter
FilterChain
FilterConfig
] javax.servlet

Filter — init()
— doFilter(req, res, FilterChain chain)
— destroy()

FilterChain(req, res) : [pass control to next chain resource]

xml

<filter>
filtername
class
</filter>
<filter-mapping>
</filter-mapping>

eg: Authentication

```
<web-app>
    <servlet>
        name
        class : AdminServlet
    </servlet>
    <servlet-mapping>
        name
        url : /servlet1
    </servlet-mapping>
    <filter>
        name
        class : MyFilter>
    </filter>
    <filter-mapping>
        name
        url : /servlet1
    </...>
</web-app>.
```

FilterConfig (Local) — Init()
- getInitParameter
- getServletContext()
- getInitParameterNames()

* parameter: value

```
<filter>
    :
    <init-param>
        name
        value
    </init-param>
</filter>
```

App: Users for single page,
Authenticate
Total response time.
before — after

ServletInputStream

→ ServletInputStream sin = res.getInputStream(); → int readLine(byte[] b, int off, int len)

→

→

ServletOutputStream

ServletOutputStream out = res.getOutputStream();

print(boolean b)
int
"
"

println()

Annotation

```
javax.servlet.annotation.WebServlet;

@WebServlet("/url")
```

Single Thread Model interface

* one request at a time (no methods : marker interface)

```
public class myClass extends HttpServlet implements SingleThreadModel{
    doGet(){
    }
}
```

SSI (Server side Include): Included in webpage (webserver analyze when req is given) → in HTML

```
<HTML>
  :
  <SERVLET CODE=Cfg>
  </servlet>
```

App: get upload
Download (write into o/p stream)!
send mail.
Write data into a PDF file.

Maven

1. powerful project management tool (pom based: Project object model)

   * builds, dependency, doc.

Why maven?

1. Add jar in each project (right one)
2. Create right structure (Struct, Servlet)
3. Build & deploy project

easy to build, uniform build (all maven project), dep list, mail list, unit test report, log doc

Maven manages: Builds, doc, reports, SCMs, releases, distribution.

Build tool: generate source code, doc, Compile it, JAR or zip, Install in local repo/Server

Ant vs Maven

→ Maven has convention: place source code, compiled code (no need pom.xml)

→ Declarative (Just say what to do not how to do)

→ Life cycle (in Maven)

→ ANT (toolbox), Maven (Framework)

→ Project man. tool

→ Reusable plugins, popular one (maven)

Install

→ latest version (zip)

→ home ⟶ env variables

mvn -version

Maven repo: JAR file (pom.xml)

local ⟶ central ⟶ Remote repo

Maven search for dependencies.

1) Local: local system (C:\Users\SSS IT\.m2) **settings.xml** (we can change)

2) Central: web (Apache maven community): http://repo1.maven.org/maven2/

3) Remote: maven repo (JBoss, ...)

Pom.xml

project - root element

modelVersion - sub element (4.0.0)

groupId - Subelement (Id of project group)

artifactId - Subelement of project (artifact: produced/used by project
JAR, source, binary distribution, WAR)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.java.app1</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>

</project>
```

Additional elements

packaging - Packaging type (JAR, WAR, ..)

name - name of maven project

url - url of project

dependencies -

dependency

scope - compile/provided/runtime/test & system

```
:
<version>1.0</version>
<name>My project</name>
<url>http://maven.apache.org</url>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</Project>
```

Intellij → maven project

Compile

mvn clean compile [go to Myapp directory]

Java.com.javatpoint.App

mvn Completes build life :

1. validate
2. Compile
3. Test
4. package (mvn package) : creates package
5. Integrating test
6. verify
7. Install
8. Deploy

Exercise : Maven webapp

Maven plugins

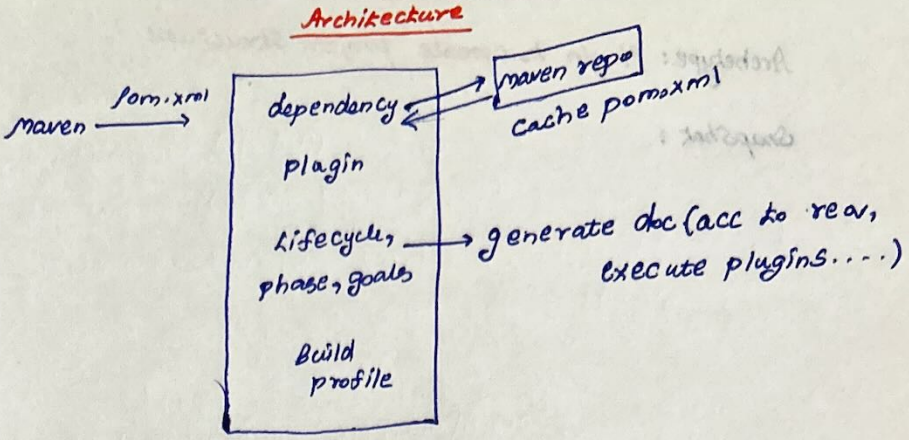
Build plugins (executed at time of build) <build>

Reporting plugins (time of reporting : site generation) <reporting>

clean — cleanup after build
Compiler — Compiles Java source code
deploy — deploys the artifact to remote repo
failsafe — runs JUnit tests in an isolated class folder
Install — Install built artifact in local repo
resources — Copies resources to o/p directory including (JAR)
site — generate site for current project
Surefire — run JUnit unit test in an isolated classloader
Verifier — Verify existence of certain condt (Integration test)

Plugins
- codehaus.org
- code.google.com
- http://repo.maven.apache.org/maven2/org/apache/maven/plu

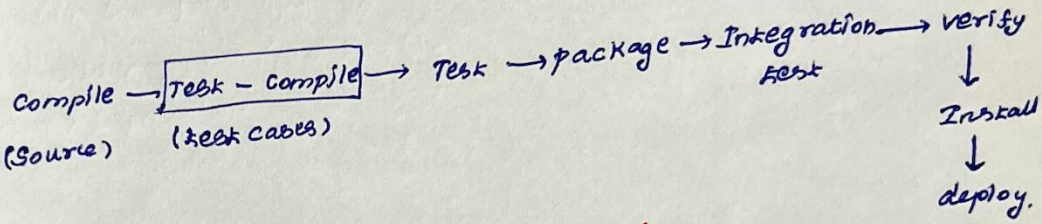
Architecture



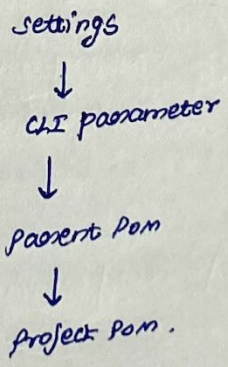
Maven build life cycle

★ Collection of steps followed to build a project

default : handle deployment
clean : cleaning
site : site documentation gener...

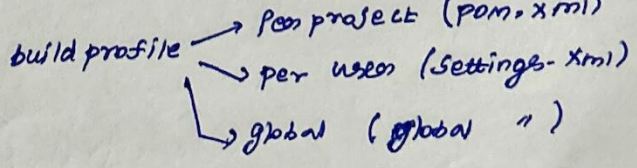


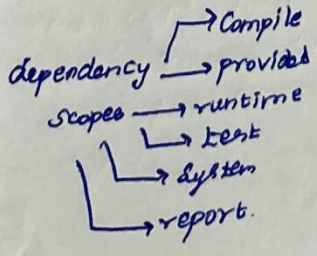
Maven order of inheritance



Plugin

clean: clean file, dir, (generated by MAVEN build process) : class files, jar..





Activate profiles (build)

★ Cmd
★ maven settings
★ env variable
★ os settings.