# Intro to tomcat

# Last time: HTTP, Chap. 18 in Murach

HTTP request has verb: GET or POST (some others too)

Syntax of request first line: GET path HTTP/1.1 or POST path HTTP/1.1
After this: request headers, then a blank line to tie them off
After this, for POST only: the POST "body", the data being sent to the server
The data is typed by the Content-type request header

Response first line: status code and string
After this: response headers, then a blank line to tie them off
After this: response body, the data being sent from the server to answer the request (may be trivial for POST)
The data is typed by the Content-type response header

There is a special response for “redirects”, responses with status code 302, new URL in the Location header, asking the browser to do an immediate GET to that URL.

# HTTP is stateless

- All the request – response cycles are unrelated, to first approximation
- No “conversation” in HTTP, server vs, user
- Each request comes with all needed info for server to handle it: "self-contained"
- In particular, there is no “current directory” (known to the server) in browsing a website—each GET has full local path
- But by standard tricks, tomcat servers (and app servers in general) manage to track each certain user for us, a great help to web app design
- So our webapp can have a conversation with its user
- tomcat does this with “cookies”, or URL rewriting tricks (will cover details later)

# Basic idea of a servlet

A servlet is Java code executed inside a web server for a certain application. One app could have several servlets.

Tomcat as a web server gets an HTTP request, figures out which servlet it's for, calls into that servlet (our Java code) to handle it and produce HTML for tomcat to send back to the user.

URL accessing my pizza3-specific tomcat installation:
http://pe07.cs.umb.edu:9002/welcome.html

You used this in homework 3 to create a page flow diagram for pizza3.

Here the pizza3 app owns the whole website served by this tomcat, an "embedded tomcat". Tomcat can also run like apache, hosting many apps, but we'll use it this simpler (embedded) way when developing webapp projects, using Spring Boot 2.

# Using embedded tomcat as a simple web server serving test.html

I've put a simple test.html in pizza3's document root at /home/eoneil/636/pizza3/src/main/webapp

This is the standard setup for a Maven webapp using Spring Boot 2.

The sources are under pizza3/src/main/java as usual.

On pe07, anyone can use URL http://localhost:9002/test.html to access test.html with lynx.

This works even though that webapp area is non-public because this web server is running as user eoneil and thus has access to this area.

In fact, this tomcat can execute added JSPs as well: it's nearly as powerful as the shared tomcat at port 8080.

The difference is mainly in "hot deployment": changing the code running the webapp without restarting the web server, possible with a shared server. The normal use of Spring Boot: rebuild, get new jar, run it. Fine for development.

# Pizza3's document root

Look at the webapp directory (pizza3's document root):

```
$ pwd
/home/eoneil/636/pizza3/src/main/webapp
$ ls
admin  basicjsp images  includes  jsp  styles  test.dat  test.html  welcome.jsp
```

- We see the test.html that we can access at URL http://pe07.cs.umb.edu:9002/test.html, using tomcat as a plain web server.
- But we don't see welcome.html, even though the URL http://pe07.cs.umb.edu:9002/welcome.html returns a page.
- That's because pizza3's servlet *interprets* that particular incoming welcome.html URL and handles it using the Java code of the servlet.
- In fact it uses welcome.jsp to create the returned page.

# Tomcat in action on test.html

Consider the access to test.html, using URL http://pe07.cs.umb.edu:9002/test.html

Here GET command is `GET /test.html HTTP/1.1`, done after TCP connection to pe07.cs.umb.edu on port 9002.

Tomcat interprets the local path: `/test.html`, sees that / belongs to the servlet of pizza3, and checks if /test.html is registered as a path of interest ("endpoint") to that servlet. It isn't.

So tomcat works on it as an ordinary web server, looking in its document root for test.html and finding it.

Since it's HTML, tomcat just reads it and sends it back on the same TCP connection. (static web server behavior)

# Tomcat in action on welcome.html

Consider the access to welcome.html, using URL http://pe07.cs.umb.edu:9002/welcome.html

Here GET command is `GET /welcome.html HTTP/1.1`, done after TCP connection to pe07.cs.umb.edu on port 9002.

Tomcat interprets the local path: `/welcome.html`, sees that / belongs to the servlet of pizza3, and checks if /welcome.html is registered as a path of interest ("endpoint") to that servlet. It is.

So tomcat hands off the handling of this request to the servlet code.

In this case the servlet just "forwards" to welcome.jsp. More on this later.

# Our shared tomcat at port 8080

This is another tomcat web server running on port 8080 on pe07. It can handle multiple apps, much like Apache, but also supports Java.

Document root: /var/cs636/tomcat-8.5/webapps (note webapps here vs. webapp for embedded tomcat)

Root page: local path /, URL http://pe07.cs.umb.edu:8080: see picture of tomcat, etc.

HTML Test file: /eoneil/test.html local path

In file system:/var/cs636/tomcat-8.5/webapps/eoneil/test.html

URL: http://www.pe07.cs.umb.edu:8080/eoneil/test.html

(use tunnel from localhost:8888 to tomcat.cs.umb.edu:8080)

# Shared Tomcat in action on test.html

Consider the access to test.html, using URL http://pe07.cs.umb.edu:8080/eoneil/test.html

Here GET command is `GET /eoneil/test.html HTTP/1.1`, done after TCP connection to pe07.cs.umb.edu on port 8080.

Tomcat interprets the local path: `/eoneil/test.html`, considers this as possible servlet "eoneil", but finds no such servlet configured. Here is the difference from the embedded case. This tomcat expects to see multiple webapps running under it, each with a top-level directory.

Having found no servlet for test.html, tomcat works on it as an ordinary web server, looking under its document root (/var/cs636/tomcat8.5/webapps) for eoneil/test.html and finding it.

Since it's HTML, tomcat just reads it and sends it back on the same TCP connection. (static web server behavior)

# JSP: Java Server Pages

Spring Boot webapps use JSP or Thymeleaf to generate HTML. We'll use JSP.

Book Coverage

First example, pg. 147 form with HTML plus ${message}, ${user.email}, etc.

Those ${…} enclose EL, Expression Language, covered in Chap. 8. They can access Java variables (here message, user) made available to the JSP.

Then to implement conditional behavior, loops, etc., we need JSTL (JSP Standard Template Library), covered in Chap. 9

The book covers servlets first, then JSP, so those variables (message, user) were set up by a servlet, but we can do a few simple things with JSP without servlets first…

# Basic JSP Examples in webapps/basicjsp on our shared tomcat

Also in webapp/basicjsp on the pizza3 tomcat, but that is hidden away from you.

These use **EL: Expression Language**, Chap. 8

EL expressions are inside ${}, allowing dynamic HTML

**Example 1 add.jsp**: to show that EL can add

1 + 2 + 3 = ${1 + 2 + 3}

URL http://pe07.cs.umb.edu:8080/basicjsp/add.jsp

works with lynx on pe07 itself

with tunnel using port 8888: http:/localhost:8888/basicjsp/add.jsp

Response to GET: 1+2+3 = 6

# Example 2 date_el.jsp: Print current date

```
pe07$ cat date_el.jsp
<!-- create a bean-->
<jsp:useBean id="date" class="java.util.Date" />
<html>
  <body>
    <h1>  Today's date is ${date} </h1>
  </body>
</html>
```

URL http://pe07.cs.umb.edu:8080/basicjsp/date_el.jsp

This creates a Java object named "date", and evaluates it as a String (i.e., calls toString() on it)
Response:

**Today's date is Wed Mar 10 10:50:29 EST 2021**

# Example 3 form with action=getparam.jsp

Finally we can pick up user input from a form!

```
pe07$ cat testform.html
<form action="getparam.jsp" method="post">
<p> enter something: <input type="text" name="formitem"> </p>
<p> <input type="submit">
</form>
```

The above is just HTML, but it sends a POST request to relative getparam.jsp, and since this is /basicjsp/testform.html, the browser will POST to /basicjsp/getparam.jsp.

```
pe07$ cat getparam.jsp
<!-- for testform submission-->
Your input was: ${param.formitem}
```

# Testform-getparm execution

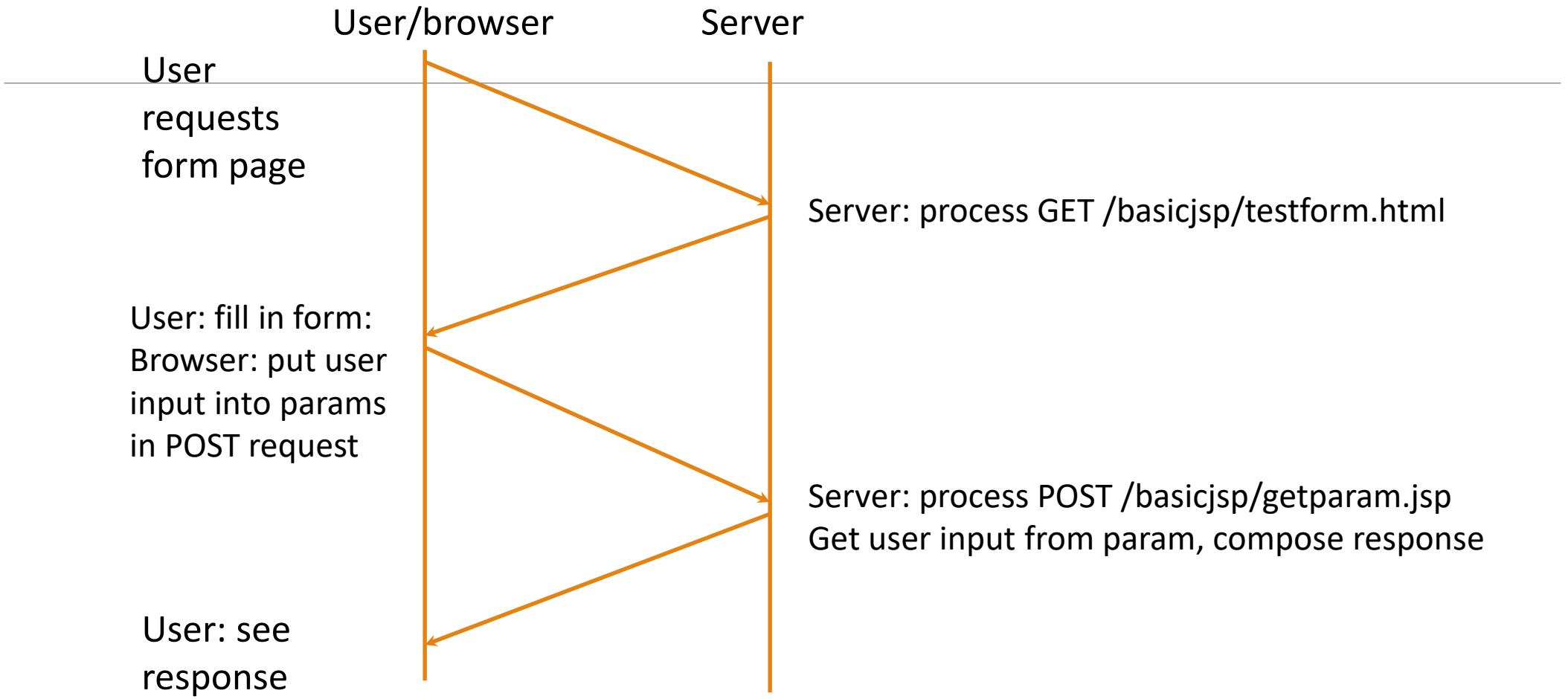
1. GET to form's URL http://www.topcat.cs.umb.edu:8080/basicjsp/testform.html

Response to GET: HTML of form

2. See form, fill in xyz, submit, causing POST to /basicjsp/getparam.jsp

Response to POST: Your input was xyz

# Communications Diagram: simple form handling



# Testform-getparm execution

Idea here: the form send a POST request with param formitem=xyz (whatever was entered). This info is in the POST request body.

The POST request is handled by tomcat, which creates an object named "param" and attaches to it all the formdata that came in, in this case just formitem=xyz.

We can access each param by its form name, like param.formitem, and get back the xxxx entered by the User.

The object named param is built into JSP, we don't need to create it (as we did with the date object earlier).

Param is actually a Java Map, and we're doing param.get("formitem") here at the Java level. See Murach, pg. 257.

# Model 1 vs. Model 2

These examples are "Model 1" examples, where requests are allowed to go directly to a JSP. Model 1 is good for small jobs, not full-blown webapps.

Murach mainly covers Model 2, which requires that *each request is handled by a servlet*, then *that servlet forwards the request to a JSP.*

See Chap. 2 of Murach on Model 1 vs Model 2. We will use Model 2 for our projects.

These examples can be run on the embedded tomcat set up for pizza3. I've copied the basicjsp directory to webapp there.

# Example 1 on embedded tomcat at port 9002

URL http://pe07.cs.umb.edu:9002/basicjsp/add.jsp

Response to GET: 1+2+3 = 6

URL http://pe07.cs.umb.edu:9002/basicjsp/testform.html

form: fill in xxyy, submit with POST

Response: Your input was xxyy

So you see that an embedded server can actually do multiple apps, but normally it just does one. It is a real tomcat, after all.

We'll cover Model 1 mainly on shared tomcat, then Model 2 on embedded tomcat, to get exposure both ways. Murach mainly covers Model 2 on shared tomcat.

Coming up:

1. You run some simple JSPs (Model 1) on the shared tomcat running on pe07 on port 8080.
2. You run embedded tomcat on pe07 for a project, pizza3 to start, get it working with your assigned TCP port (to come) on pe07. Way to test your project deliveries on pe07
3. You run embedded tomcat for a project on your development machine, using port 900x.

# Running pizza3 using embedded tomcat

Run pizza3 with the command:  runJarByProfile.sh oracle web  (using Oracle for db)

Where that .sh has

java -jar -Dspring.profiles.active=$1 target/pizza3-1.jar $2

This expands to a specific Linux command:

java -jar -Dspring.profiles.active=oracle target/pizza3-1.jar web

java –jar jarfile “runs the jar”, meaning the jar file itself knows the class to run: it’s built into the jar.

This just runs the appropriate class out of the jar file, much like pizza1, but this executable contains the needed tomcat code to run a tomcat server, and also the webapp code that handles the requests. All in Java.

In main of this class, the Spring boot code starts up that server on the given port (9002 in this case) with the app installed in it. The “web” at the end is an argument passed to main.

# Running pizza3 using embedded tomcat

Again, run pizza3 with the command:  runJarByProfile.sh oracle web

It’s that easy. We can replace oracle by mysql or h2, and web with SystemTest or other app.

When we do a HTTP request with a URL like http://pe07.cs.umb.edu:9002/welcome.html, it is served by the webapp of the executable.

The advantage of this over an external web server is that debugging involves only one executable, compared to two for a web app on an external web server like apache or shared tomcat.

In a way, it's really too magical! We have start with the shared tomcat. It works much like our departmental apache server, plus it can execute JSPs and Java.