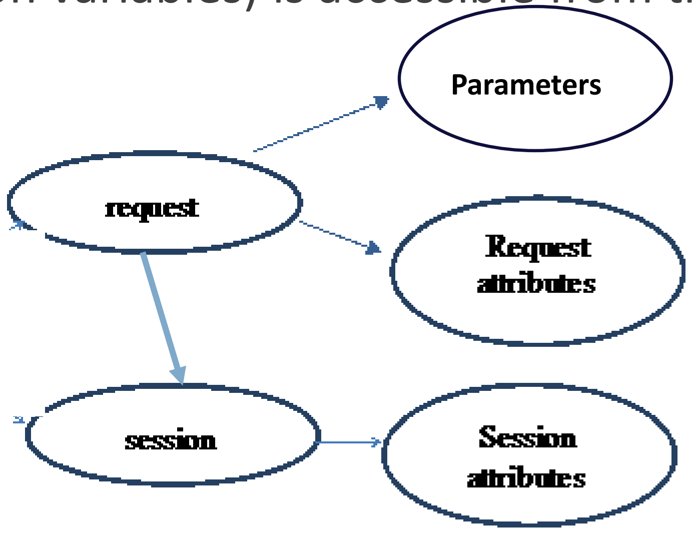
Session Variables, SystemTest in web case

Last time: MVC webapps with session variables.

- We looked at Chap 7 on Session Variables.
  - Cart app: `cart` object is a session variable, so as users add an item to cart, etc., it is remembered from request to request. No user login here.
  - Download app: Has user login, creating a `user` session variable. Presence of that user object means the user is "logged in", and can access the music samples. Also has a `product` session variable to remember which CD the user wanted to play samples from across user registration.
- Previously, we saw how to use "request variables" like the "user" variable in ch05email and "allSizes" for pizza3's show-order-form page.
  - Note that "user" is more often a session variable, tracking the logged-in user.
- The session variables belong to a certain user, like the longer-lived variables in presentation in client-server apps.

Accessing session variables from Java

- The session variables hang off the "session" object just like the request variables hang off the request object. The session object is obtainable from the request object, so everything (among request and session variables) is accessible from the request object.



Parameters: request.getParameter(...)

Request variables: request.getAttribute(…)

Session variables
request.getSession().getAttribute(…)

(also setAttribute in the lower two cases)

Accessing session variables from JSP

- EL can access session variables just as easily as request variables
  - Cart app: See in cart.jsp: using the `cart` sesson variable

```
<c:forEach var="item" items="${cart.items}">
```

  - Download app: `productCode` is a session variable for “pf01” or whatever:

```
<td><a href="/sound/${productCode}/filter.mp3">MP3</a></td>
```

- Be sure to use different names for session and request variables to avoid confusion

Session variables for music3

We need:

- UserData object for logged-in user (one who wants to listen to samples or check out)
- Cart object to maintain shopping cart (like ch07 cart example)
- Product id or code for remembering what product a user is interested in through the needed registration for sample downloads (like ch07download example)
- doingCheckout Boolean to guide checkout vs. play-sample after doing registration

These are all for a certain user session, so we can gather them together in an object, of suggested name UserBean. See UserBean.java in presentation.web.

Then we can make the UserBean a session variable and it will carry around all the above state.

But if you want, you can have separate session variables for the four objects.

How tomcat does session tracking

- On the first request by a user, a new session object is created, and a new session id is sent back to the browser in a cookie.
- The session object is held inside tomcat in a hashtable, for lookup by session id.
- Later requests have the cookie and thus the session id.
- Tomcat locates the right session object from its hashtable.
- Tomcat attaches the session object to the current request’s request object.

Steps in request handling by tomcat

Request comes into tomcat:

- Tomcat interprets request params from the query string of GET or body of POST (see Murach, pg. 152) and attaches them to a new request object
- Tomcat sets up new response object
- Tomcat uses cookie from request and its session id to look up and attach the right session object to the request object (or create a new one)
- Tomcat calls doGet or doPost (with the request and response objects as args) and enters our servlet code

Steps in request handling by tomcat

Our servlet code uses the request for params, attaches attributes (request vars), also uses session vars as needed

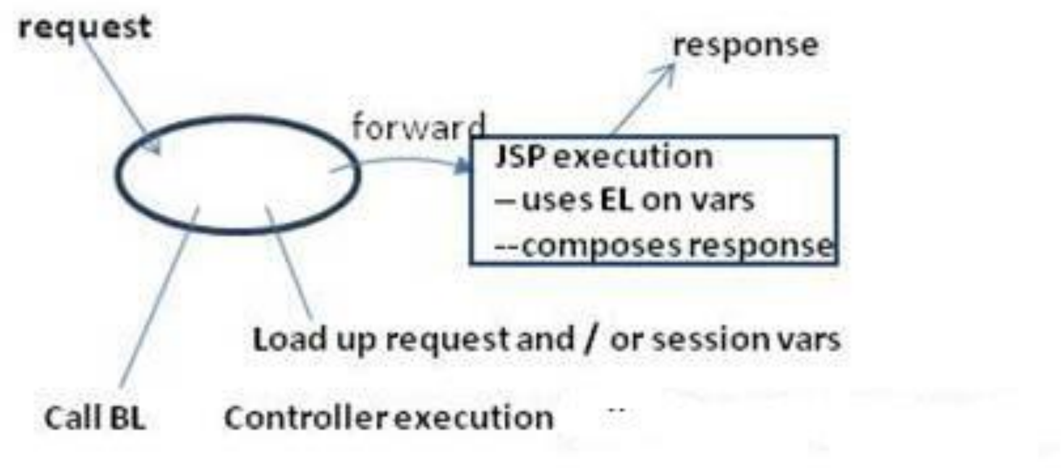
Tomcat uses the path in URL provided to forward method to locate jsp*, compile if necessary, load .class if necessary

Tomcat calls into the JSP servlet, reusing the same request and response objects, and starts generating content of the response.

The request and response objects are forwarded, so for example, the original servlet can attach attributes to the request and/or session and the JSP can use them.

*Or, if not forwarding to a JSP (which compiles into a servlet), it forwards to a non-JSP servlet, either a servlet we've coded up or tomcat's DefaultServlet that knows what to do with HTML or image files or audio or … (static resources).

Forwarding to a JSP



Here the oval is the servlet (being a Java class), and the JSP is in the box, being more restrictive in what it can do. It doesn't show how the request and session vars get transported across with the forward. Here BL = business layer, i.e. service layer.

Music3-setup is now available

And the project 2 assignment.

music3-setup.zip, meant for music3, later called pa2 for delivery

Music3-setup has

- Nearly the same DAO as pizza1: just JDBC.
- Nearly the same service layer as pizza2, with transactions
- Same shell scripts as pizza3
- The admin pages implemented with a Spring Controller called AdminController, and jsps.
- Start on UserController, a few handlers there, and starts on the various jsps needed.

Running pizza3: three ways: Jar, mvn, eclipse (music3 is similar)

```
runJarByProfile oracle|mysql|h2 web|SystemTest|TakeOrder|ShopAdmin
```

or

```
runMvnByProfile oracle|mysql|h2 web|SystemTest|TakeOrder|ShopAdmin
```

Note with either of the above, the web case hangs, so switch to browser to see stuff. The original window will be used for the tomcat log output.

For “runJar…”, need to build the jar with `mvn clean package`.

In eclipse, find class SBApplication, use Run As > Java Application or Debug As > Java Application. Uses “h2 web” case, for easy development.

To run tomcat in the background and output both stdout and stderr to a file on Linux/Mac (useful on topcat) (for example using Oracle):

runJarByProfile oracle web > tomcat.log 2>&1&

Next challenge: Moving pizza/music to the web container inside tomcat

We expect to get our BL+DAO code from pizza1 or pizza2 or music1 to execute inside the web container, i.e., in tomcat, but obviously the presentation layer has to be completely rewritten, a big job.

The BL+DAO code in pizza3 is close to the older code in pizza1, but there are some differences:

Differences between pizza1 and pizza3 in service and DAO layers:

- Use of Spring beans for service and DAO objects (like pizza2)
- Back to plain JDBC, no JPA in use in DAO.
- Use of transactions in each service method (like pizza2, but use JDBC transaction directly, not via JPA)
- Use of DataSource configured via Spring to provide JDBC Connections (unlike pizza2, need to discuss)

SysTestServlet: the minimal presentation layer

How can we start? We need an absolutely minimal presentation code. No user input.

Hey, we already have it, in SystemTest!

We will get SystemTest to run from a servlet…The System.out.println output will show up in tomcat's log (standard output for embedded tomcat)

The provided pizza3 project still has this SysTestServlet, so we can run it to see how this idea works.

You did this for homework 4, and saw the output on the console.

The fine print: Note that this doesn't use the whole presentation layer of pizza3. It shortcuts all that and just uses this servlet code plus the BL+DAO. And it is using Spring Boot's way of running tomcat as an embedded server. It would work with pizza1's BL+DAO too, at least with a single user.

Looking at SysTestServlet

Servlet config for SysTestServlet.java:

```
@WebServlet(
          name = "SysTestServlet",
          description = "Servlet to run SystemTest",
          urlPatterns = {"/servlet/SystemTestServlet"}
        )
```

This is running on pe07 at port 9002, so the URL for the servlet is

```
http://pe07.cs.umb.edu:9002/servlet/SystemTestServlet
```

We can use lynx on pe07 with this URL:

```
lynx localhost:9002/servlet/SystemTestServlet
SystemTest Result
   Success
   for more info, see console log
```

We are using Spring beans in music3-setup

- Since we're using Spring to set up the BL and DAO beans, we don't need the old code in PizzaSystemConfig.
- See class of Apr 7 for intro to the Spring beans. Recall the “big picture”.
- Since a @WebServlet class can be a Spring component (or bean), we can use @Autowired to pick up the service object references that SystemTest needs to do its job…

SysTestServlet code

```
// A WebServlet is a component, so we can use @Autowired
@WebServlet(
  name = "SysTestServlet",
  description = "Servlet to run SystemTest",
  urlPatterns = {"/servlet/SystemTestServlet"}
)
public class SysTestServlet extends HttpServlet {
@Autowired                                          ←autowired service beans
private AdminService adminService;
@Autowired
private StudentService studentService;
```

Running SystemTest in pizza3/music3

- SystemTest itself no longer has a main: instead, its code is called from the CommandLineRunner (class CommandRun) as `runSystemTest(adminService, studentService).` But it's the same old code once started.
- Where is the main now? It's in SBApplication (short for Spring Boot Application) in the presentation package, in both pizza3 and music3. You can run the system in eclipse from there.
- The corresponding main class in ch05emailS, ch07cartS, and ch07downloadS is called WebApplication, because it only handles the web app case.
- But pizza3 and music3 can run both as a web app (with embedded tomcat) and client-server (not running tomcat), i.e., as an ordinary program, so we use the name SBApplication to cover both cases.
- Since a Spring boot application by default runs tomcat, we have to explicitly turn it off in main.

From class 17: How can pizza3 run a webapp, or run SystemTest or …?

The default app.run() in xxxApplication starts up embedded tomcat, so when we just want to run SystemTest, we configure that `app` object like this, turning off tomcat execution this way:

// code in SBApplication for pizza3

```
if (!appCase.equals("web")) {  // appCase is “SystemTest” or “web” or …
   System.out.println("have arg " + appCase + " ,
                              assuming client execution");
   app.setBannerMode(Banner.Mode.OFF);
   app.setWebApplicationType(WebApplicationType.NONE); // don't start tomcat
}
app.run(args);  // get tomcat running if appropriate, create and wire up beans
```

Also from class 17: the mysterious CommandLineRunner

From two slides ago:

SystemTest itself no longer has a main: instead, its code is called from the CommandLineRunner (class CommandRun) as `runSystemTest(adminService, studentService).`

But we don't see CommandLineRunner called from main, so how does it get called at all?

From Class 17, slide 18:

How can we do app-specific startup actions?

The deal: if we set up a @Component (i.e. bean) of interface type **CommandLineRunner**, Spring will find it and run it (call its run method), using main's command line args as args to it. This will happen after all the beans are created and autowired.

Arrangements like this are called "hooks". From SO 467557 (what-is-meant-by-the-term-hook…)

Essentially it's a place in code that allows you to tap in to a module to either provide different behavior or to react when something happens.

Spring Boot startup summary

Start in main of xxxxApplication

Create xxxxApplication object, we called it “app”

Configure it if needed, such as turning off tomcat

Call its run method

When it’s finished creating beans and doing autowiring, it looks for a bean of interface type CommandLineRunner and calls its run method with main’s args.

- That’s where we create a SystemTest or a UserApp or whatever the args ask us to do
- We don’t do any config here for the web case but we could

For SystemTest, etc., returning from run exits the whole program, but in the web case tomcat is still running, serving requests.