# More on Spring Controllers

1

## Last time: Download app, SysTestServlet

- We looked at the book slides on the ch07download 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.
  - Shows how mp3's can be played by links in HTML (browser plays the mp3)
  - ch07downloadS: shows another way too: forward to .mp3 from a servlet (tomcat itself plays the mp3, using its DefaultServlet)
- We saw how to get SystemTest to run from a servlet.
  - 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.
  - This can be done first, before the main web UI is implemented.

2

## Executing the SysTestServlet

- We can execute SystemTest's run() from a simple servlet. It doesn't need any UI, but does need to read its input file. See next slide.
- Since we're using Spring to set up the infrastructure objects, we don't need the old code in PizzaSystemConfig.
- 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
- SystemTest itself no longer has a main: instead, its code is called via method runSystemTest(adminService, studentService). But it's the same old code once started.
- Where is the main now? It's in SBApplication in the presentation package. You can run the system from eclipse from there.

3

## SystemTest's access to its input file

- There is another technical detail: SystemTest reads a file, but a web app doesn't know where it is in the filesystem, or this is not obvious anyway.
- See pg. 141 of Murach for code accessing a file from a servlet.
- We can solve the input file problem by putting test.dat "on the classpath", simply by sticking it in src/main/resources, along with application.properties, etc., which also live there.
- All these files get copied to target/classes by the build, so they end up on the classpath at runtime.
- Here is the line from SystemTest that finds the file

```
Scanner in =
    new Scanner(this.getClass().getClassLoader().getResourceAsStream(inFile));
```

- This is a nice trick: it works for client-server as well as web executions.

4

## Spring Controllers and their handlers

We got started on this subject a few classes ago, Class 18.

Quick summary of most common case:

Request to endpoint URL →execute handler of Spring Controller → determine JSP to forward to

First we looked at hello-world type situations: an endpoint that just forwards to a JSP, no request vars or session vars. See slides 11-16 of Class18.

Then we looked at an endpoint that did set up request vars, specifically the Set<String> for the various size names and another for the topping names needed by the pizza order form.

Let's look at those slides again…

5

## In Spring controllers:
## What the Model is for: sending data to JSP

```
@RequestMapping("orderForm.html")      ← request to /orderForm.html
public String displayOrderForm(Model model) throws ServletException {
 List<String> allSizes = null;
 List<String> allToppings = null;
 try {
   allSizes = studentService.getSizeNames();   ←call service API
   allToppings = studentService.getToppingNames();
 } catch (Exception e) {…}
 model.addAttribute("allSizes", allSizes);   ←pack up data in model
 model.addAttribute("allToppings", allToppings);
 model.addAttribute("numRooms", 10);
 return "jsp/orderForm";    ←forward to orderForm.jsp in jsp dir
}
```

Spring code will attach the attributes from `model` to the `request`, making them request variables. The JSP will have access to request variables `allSizes`, `allToppings`, and `numRooms`.

6

## This forwards to JSP for pizza order page

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

…

<!--change to method="post" when development is done -->
<form method="get" action="orderPizza.html">


Pizza Size: <br>
<c:forEach items="${allSizes}" var="curSize">
<input type="radio" name="size" value="${curSize}">  ${curSize} <br>
</c:forEach>

…
```

*Needed for JSTL: taglib directive at top of page*

*Use JSTL to loop through allSizes*

## The Spring DispatcherServlet again…

Execution sequence:

1. GET /orderForm.html comes into tomcat (or POST), it locates right servlet, in this case Spring's DispatcherServlet.

2. That servlet has previously scanned for all the @RequestMappings in our code, and has a registry of them to consult. It finds that the endpoint "/orderForm.html" is handled by method "displayOrderForm" in StudentController.

3. The DispatcherServlet calls StudentContoller.displayOrderForm with an empty Model object, gets back a filled-in Model object and return value "jsp/orderForm". It uses the Model object to set request attributes, forming request variables allSizes, etc., and forwards the request to orderForm.jsp in the jsp directory of webapp, the document root.

## Compare to ch05emailS's EmailServlet

EmailListServlet (code on next slide, from Chapter5slides) handles GETs and POSTs to an endpoint "/emailList". It forwards to a JSP named thanks.jsp.

Execution:

1. Client sends a GET request to the endpoint, after opening a TCP stream connection to the tomcat server

GET /emailList HTTP/1.1

2. tomcat locates servlet by analyzing URL's path

3. tomcat looks at that servlet's registered endpoints, and on match, calls doGet in that servlet's code.

4. doGet interprets user input (not done by the Spring controller we looked at, but of course possible), then creates a User object and attaches it to the request object, making it into a request variable named "user" that can be used in the forwarded-to JSP, thanks.jsp.

Further note: "user" is more often a session variable, but it is just a request variable here.

### The EmailListServlet class

```
package murach.email;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

import murach.business.User;
import murach.data.UserDB;

@WebServlet(name = "EmailList", description = "Servlet to handle email list",
urlPatterns = {"/emailList"})

public class EmailListServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
                    throws ServletException, IOException {

        String url = "/index.html";

        // get current action
        String action = request.getParameter("action");
        if (action == null) {
            action = "join";  // default action
        }
```

*Line added to slide, for our setup (replacing web.xml): registers endpoint /emailList*

*doGet also comes here, because doGet calls doPost*

### The EmailListServlet class (continued)

```
        // perform action and set URL to appropriate page
        if (action.equals("join")) {
            url = "/index.html";    // the "join" page
        }
        else if (action.equals("add")) {
            // get parameters from the request
            String firstName = request.getParameter("firstName");
            String lastName = request.getParameter("lastName");
            String email = request.getParameter("email");

            // store data in User object and save User object in db
            User user = new User(firstName, lastName, email);
            UserDB.insert(user);

            // set User object in request object and set URL
            request.setAttribute("user", user);
            url = "/thanks.jsp";   // the "thanks" page
        }

        // forward request and response objects to specified URL
        getServletContext()
            .getRequestDispatcher(url)
            .forward(request, response);
    }
```

*How we provide a request variable named "user" to the JSP*

## Summary on Spring Controllers so far

So far, we have seen how to set up simple endpoint handlers in a Spring controller, a Java source file with class annotation @Controller.
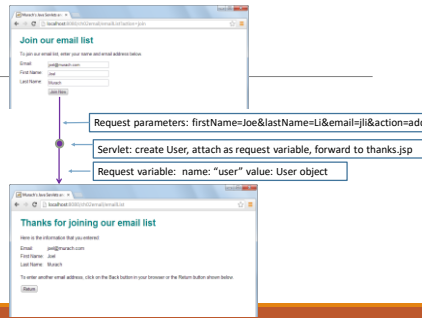
We have covered:

• How to specify the JSP to forward to: just return a String name, Spring will append ".jsp" to it (this is configurable).

• How to specify request variables that the forwarded-to JSP can access. Here the deal is: have a method argument "Model model", so that Spring can supply an empty Model object to the code, then, in the method, we add attributes to model that we want turned into request variables.
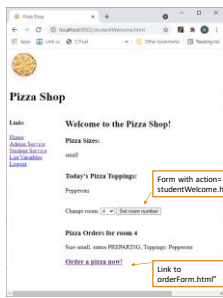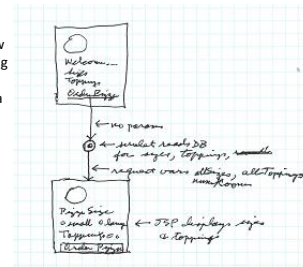
We need to cover in the future:

• How to get user input out of the incoming request object. (Coming up)

• How to use session variables to keep a conversation going with our user. (Now covered)

Another way to look at EmailList Servlet: as page flow with servlet execution on the way



Request parameters: firstName=Joe&lastName=Li&email=jli&action=add

Servlet: create User, attach as request variable, forward to thanks.jsp

Request variable: name: "user" value: User object

Rough page-flow picture for Spring controller that sets up the pizza order page in pizza3

## The Student Welcome Page



Form with action= studentWelcome.html"

Link to orderForm.html"

- This is the student welcome page.
- You can see /studentWelcome.html in the address bar, so you know this page resulted from a request to /studentWelcome.html
- That /studentWelcome.html is an endpoint of a Spring Controller handler that gets needed data and forwards to a JSP that generates this page.
- There are two side-by-side boxes here: the left-hand side comes from an included JSP that is included from many pages.
- The right-hand side is the main content of this page.
- You see it has a button to set the room number, i.e. UI, a form which submits back to this page.
- Also a link to the order form endpoint.

## What does that studentWelcome.html handler need to do?

- It needs to gather all the data needed by the student welcome page, including the left-hand part, and turn it into request variables.
- We already know how to do this: it's like the order-form setup.
- But it also needs to handle incoming submissions of the set-room-number button.
- We can see from the JSP that the submission request will have parameter room=6 if the user selects 6. The full path will be /studentWelcome.html?room=6
- In a simple servlet, we would use request.getParameter("room") to pick up the 6.
- How do we pick it up in a Spring controller handler?

## A Spring Controller Handler that accepts UI (incoming request parameter room=6 or …)

```
@RequestMapping("studentWelcome.html")
public String displayWelcome(Model model,
    @RequestParam(value = "room", required = false) String chosenRoom,
     HttpServletRequest request) throws ServletException {
    Integer roomNo = null;
    // take room parameter over session var in StudentBean--
    if (chosenRoom != null) {
        try {
            roomNo = Integer.parseInt(chosenRoom);
            System.out.println("Got roomNo from param = " + roomNo);
        } catch (NumberFormatException e) { … }
    }
    …
```

- The bold part says: look for parameter room=6 or whatever, put that 6 in variable chosenRoom, and don't cause an error if it's missing.

## More notes on @RequestParam

See https://www.baeldung.com/spring-request-param for a brief tutorial.

You can drop the value="xxx" if it's the same as the variable name: his example:

**public** String **getFoos**(@RequestParam String id) works the same as

**public** String **getFoos**(@RequestParam(value="id") String id)

In this case, since it doesn't say **required = false,** an error will be generated if there is no id=something in the incoming request, specifically an HTTP 400 Bad Request error

Another example from there: using defaultValue

**public** String **getFoos**(@RequestParam(defaultValue = "test") String id)

This is like required=false, in that the user no longer needs to supply the parameter. Here it will simply become id = "test" if not specified in the request.

## OK, we picked up the new room number, what next?

- The room number is the user identifier in the pizza project, much like a username in a bigger app.
- We need to put it in a session variable, so that it "sticks" to the user session.
- To keep things simple, we won't use Spring's session support*, so we want to just get the request object, ask it for the session object, and add info to it as we would in a servlet.
- But where is the request object??
- Just as the Model springs into action when we put it as a method parameter to the handler, all we need to do is put a method parameter of type HttpServletRequest in our handler and Spring will fill it in with the actual request object.

\* session support is complicated in larger web app deployments because the involved web server may change from request to request, so the database needs to be used. We'll assume a single server, knowing that Spring could help with the larger case if needed.

19

## Accessing the session variable via the request variable

```
@RequestMapping("studentWelcome.html")
public String displayWelcome(Model model,
    @RequestParam(value = "room", required = false) String chosenRoom,
    HttpServletRequest request) throws ServletException {
... (covered already)
    StudentBean student =
            (StudentBean) request.getSession().getAttribute("student");
    if (student == null) student = new StudentBean();
    if (roomNo != null)    student.setRoomNo(roomNo); // set newly obtained roomNo
    if (student.getRoomNo() > 0)
        roomNo = student.getRoomNo(); // just set or older setting
    request.getSession().setAttribute("student", student);
```

> Put HttpServletRequest param in handler, to provide access to the request, and the session

Alternatively, use a handler method param of type HttpServletSession and just get the session object

20
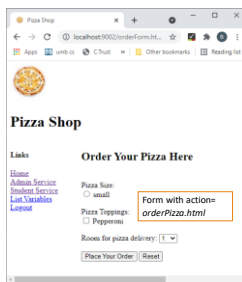
## The Order Form

- From the address bar, we can see this is the view for the endpoint /orderForm.html
- When the user fills out the form and clicks "Place Your Order", the request goes to /orderPizza.html, another endpoint of StudentController.
- That request will bear the user input in parameters like size=small&toppings=Pepperoni&room=2
- Or with multiple toppings available, toppings=Pepperoni&toppings=Onions

> Form with action= orderPizza.html

21

## Ordering pizza: handling a request with toppings=x&toppings=y

In a simple servlet, we would use request.getParameterValues in this case.

Here we see:
```
@RequestMapping("orderPizza.html")
    public String orderPizza(Model model,
        @RequestParam(value = "size", required = false) String chosenSize,
        @RequestParam(value = "room", required = false) String chosenRoom,
        @RequestParam(required = false) List<String> toppings,
        HttpServletRequest request) throws ServletException {
```

Here value=toppings, so the bold clause does match toppings=x of the query string. The fact that the parameter type is List<String> is the clue that Spring uses to expect toppings=x&toppings=y syntax, parse it, and load up a List<String> for us. Pretty neat!

22

## Redoing the form if necessary

- If the user fails to choose a size or any toppings, we need to show the order form over again, with an error message showing what they did wrong.
- Thus this orderPizza handler has two forward-to URLs:
  - Success: forward/redirect to studentWelcome.html to show the new order, etc.
  - Failure: forward to orderForm.html to reshow the order form, with error message
- This is easy to code: just arrange to return the right string from the handler.
- Note that the code actually uses redirect, but could use forward.
  - FYI: The difference is in how it affects the browser. The redirect makes the browser do another GET back and thus changes what's in the address bar. If the user does a refresh, that GET will be repeated, not the form-submission request, which can cause extra orders.
  - A complete (long-winded) discussion is in http://www.theserverside.com/news/1365146/Redirect-After-Post

23

## Endpoints and their handlers, and forward-to URLs

```
Request URL          Controller handler    forward-to-URL
                                            (for "view" or further servlet processing)
/welcome.html        welcome               /welcome.jsp
/studentWelcome.html displayWelcome        /jsp/studentWelcome.jsp
/orderForm.html      displayOrderForm      /jsp/orderForm.jsp
/orderPizza.html     orderPizza            /jsp/studentWelcome.html
                                       OR  /jsp/orderForm.html for form resubmission
/orderReceive.html   receivePizza          /jsp/studentWelcome.jsp
```

- Note that the URLs here are "servlet-context-relative", so /welcome.html means GET to /welcome.html for embedded tomcat, or /pizza3/welcome.html in non-embedded tomcat.
- It's good for the controller source file to define String constants for this urls, making clear their relationship. See AdminController.java and UserController.java of music3-setup for this setup.

24

## UI in general for a Collection of objects

- Here the UI has to manage various collections of objects: student orders for a room, toppings, sizes, days, all orders.
- The pattern we use is as follows:
  - **main page**: display the collection in rows. Provide a link to the add-form page to add a new object to the collection. If objects can be deleted, provide a button in the row for this action, with redisplay of the main page on submit.
  - **add-form page**: a form for additions to the collection, with redisplay of the main page to show the new collection.
- This two-page solution works fine for small collections, where the user can easily detect the change visually. For large collections, more pages may be used to assure the user that their change has actually been made.

25

## The AdminController

- Note that there is another controller class for the admin pages, AdminController. All its request URLs are of form /adminController/xxx, allowing easy recognition of admin actions as requests come in (but we are not taking advantage of this here, for simplicity.)
  - This URL syntax allows for security constraints based on this in the incoming URL. See pg. 507 for an example of a security constraint on URLs with /admin/ in the URL. We aren't covering this, however.
- See in sidebar.jsp, the included .jsp (this following line assumes use of embedded tomcat)*

  `<a href="/adminController/adminWelcome.html">Admin Service</a>`
- When the user clicks the "Admin Service" link, the browser's concept of current server directory changes to /adminController, so relative URLs in the view JSPs work to stay in this area.
- The user can click on another link in the sidebar to get to the student pages, again changing the current server directory in the browser's state.

\* For shared tomcat we could put "/pizza3/adminController/adminWelcome.html", but that's not great since it only works if the app is deployed as "pizza3". There is a JSTL tag c:url that allows this url to be coded `<c:url value="/adminController/adminWelcome.html" />`. See page 287.

26

## In a JSP for an admin page: uses a relative URL to access an endpoint

Example: in toppingView.jsp, a JSP for an admin page:

`<form action="toppings" method="post">`

Here "toppings" is a relative URL to the browser-known current path /adminController, so the full incoming path will be /adminController/toppings, and fit the @RequestMapping of AdminController's manageToppings controller.

`@RequestMapping(ADMIN_BASE_URL+"toppings")`

Also note that the endpoint names are not all of form xxx.html. Above, we have instead just "toppings", no ".html", and there are similar endpoints for sizes, orders, and days.

These are REST-like endpoint URLs. We are still returning HTML, not the usual JSON for REST endpoints. But it wouldn't be hard to switch over to JSON.

Note that in the music3 setup, there is no special prefix for admin URLs.

27

## Admin URLs

| Request URL | controller method | forward-to-URL |
|---|---|---|
| /adminController/adminWelcome.html | adminWelcome | /admin/adminWelcome.jsp |
| /adminController/toppings | manageTopping | /admin/toppingView.jsp |
| /adminController/sizes | manageSizes | /admin/sizeView.jsp |
| /adminController/orders | manageOrders | /admin/orderView.jsp |
| /adminController/days | manageDays | /admin/dayView.jsp |
| /adminController/initializeDB.html | adminInitDB | /admin/initializeDB.jsp |
| /adminController/SysTest.html … (some others) | sysTest | /admin/initialized.jsp (poor name) |

28

## Requests, Blow by blow (part 1 of 2)

1. When each request comes in, it gets a thread in tomcat's JVM, in which the whole request-response cycle will execute.
2. In this thread, tomcat calls into the Spring dispatcher servlet (doGet or doPost) with the newly created request and response objects and the associated Session object available.
3. The dispatcher servlet code interprets the incoming parameters and calls the right handler
4. The controller handler calls the service API
5. The service method runs a transaction, all in the one thread.
6. When the service method is finishing up, it calls the DAO to commit or rollback. The service layer returns to the presentation layer, here controller code.

29

## Requests, Blow by blow, continued

7. The controller returns to the dispatcher servlet with a string specifying the JSP or controller to forward to.
8. Back in dispatcher servlet code, the servlet sets up request variables, forwards to JSP/controller/static resource. This code is still running in the same thread.
9. The request and response objects, bearing request and/or session variables, become the request and response objects for the JSP (or other) servlet (this may be in another thread, but it doesn't matter, it happens in the server). The JSP code can access the request and session variables using EL and JSTL.
10. At the end of the response, the request and response objects are freed, but the session object lives on in tomcat, for the next request from the same user.

30

5

## Practical Details on JSP

Libraries Needed: Note that our Maven setup handles this for Maven projects. See the pom.xml entries with JSP and JSTL in comments.

Libraries needed for JSP with EL but not JSTL: none. Note how our shared tomcat's webapps/basicjsp has no WEB-INF/lib directory. The ch06emailS project also uses only EL in its jsp files. It has no JSTL dependency in its pom.xml. Its JSP-related dependency is for its embedded tomcat, which needs to be able to compile the JSPs.

JSTL, however, requires libraries See discussion of JSTL libraries on pg. 271. The shared tomcat's basicjstl has a WEB-INF/lib directory with two jars. Luckily, we are using Maven for our serious projects, so we just need the right dependencies in pom.xml. See the entries with JSP and JSTL in comments for all the projects *S since ch05emailS.

Needed at start of JSP file with JSTL: set up c: alias for core JSTL library:
```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
```
If you forget this in your file, any <c:...> gets sent out as part of the response, causing no end of trouble.

31