# Servlets, Spring Boot Controllers

## Last time: ch05emailS, start on pizza3

- Both use Spring boot’s embedded tomcat for easy development
- Both use Maven, which builds an all-dependencies jar file in the target directory
- Both use servlets, but we haven’t yet covered pizza3’s servlets (Spring controllers).
- Only pizza3 uses Spring beans for singletons. ch05emailS doesn’t use singletons, rather setting up APIs based on static methods, a competing pattern.
- Only pizza3 uses real database access, so we need to handle the database account params.
- Only pizza3 can run as a web app or as an ordinary executable, running SystemTest or whatever. This means app-specific code needs to run after Spring sets up the beans. This is a tricky point: turns out we need a @Component bean of a certain type to pick up the execution at that point.

## More on forwarding the request for ch05email

- The forwarding operation is a little mysterious because it utilizes a URL but does not involve handling an external request coming into the server.
- Instead it is used to “chain” work inside the server.  The code to do the forward in EmailListServlet is:

```
getServletContext().getRequestDispatcher(url).forward(request, response);
     where url = "/thanks.jsp"
```

- This shows the request and response objects being handed over to be reused with the target servlet.
- No other data objects are handed over, but we know lots of info can be attached to the request.
- The request attributes, added by, say, `request.setAttribute("user", user);` become **request variables**, accessible from the forwarded-to JSP

# Forwarding in tomcat

- The forwarding action occurs inside the tomcat server, so there is no round-trip time back to the browser involved.
- The browser doesn't know anything about it, and so the address bar in the browser doesn't change either, never displaying the name of the .jsp being forwarded to.
- The browser gets the HTML generated by the .jsp as the response to its request.
- Timeline: browser B uses URL for servlet S, which forwards to JSP J (really its servlet), which creates HTML response for browser B:

```
---*--------------------------**-------------------------------*---------

   B                          SJ                               B

   time--->
```

Forwarding in here, very fast

## The forwarding URL

- The url in the ch05emailS case is “/thanks.jsp”. This is neither a relative URL nor an absolute one.
- The / here represents the root directory of the webapp, webapp itself for embedded tomcat, webapps/ch05email for shared tomcat, so this is forwarding to thanks.jsp in that directory.
- We see that we don’t need to know the webapp name to code the webapp, or even whether it will run on embedded tomcat or shared tomcat, because this forwarding URL is relative to the root directory of the webapp, wherever that is.
- We must always use a URL like this for forward, that is, it must start with a /. Forwards can only go to somewhere in our current webapp, not out into the Internet or even to another webapp on this server.

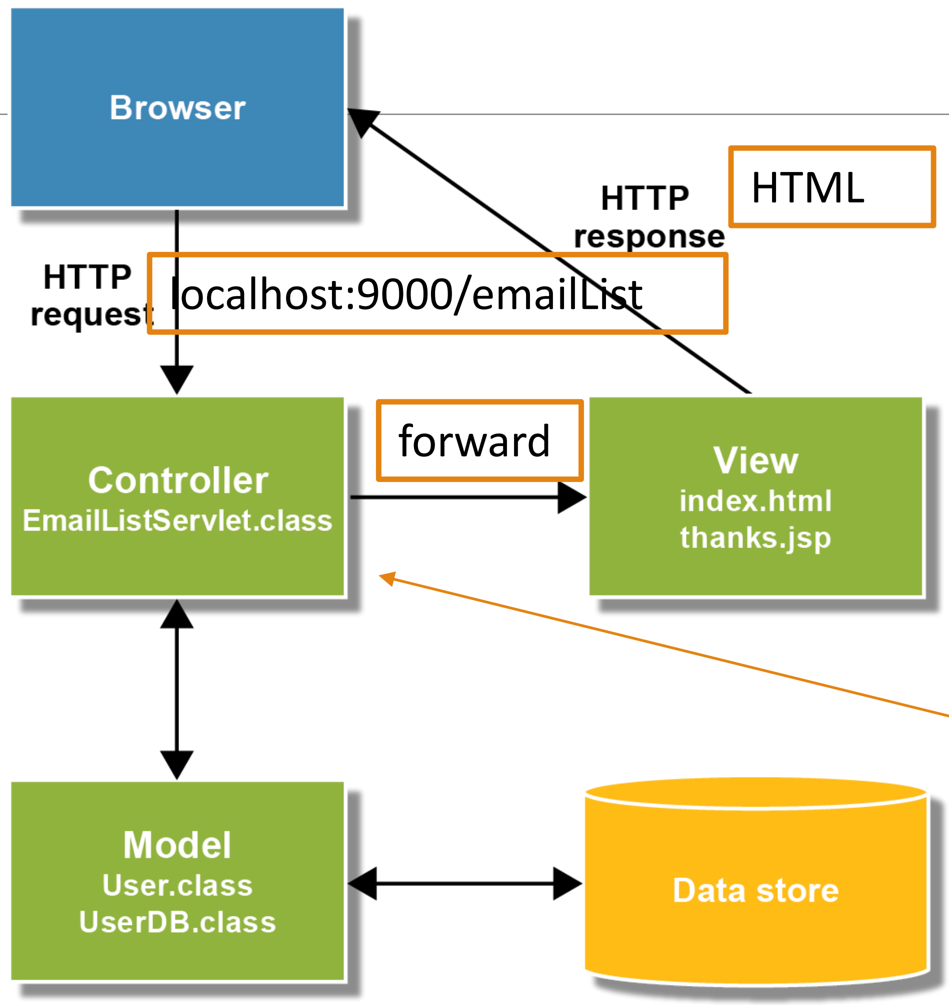
## Aside: Same-syntax URL, different meaning…

- Another URL starting with / is shown on pg. 101, in HTML: `<a href=/musicStore”>`…
- The / means something different outside of servlet forwarding: it means the document root of the webserver filesystem.
- It is a "document-root-relative" or "site-root-relative" or just "root relative" URL (apparently no standard terminology)
- For example, our class website, [https://www.cs.umb.edu/cs636](https://www.cs.umb.edu/cs636), could be linked to from any page on the whole departmental site by `<a href=“/cs636”> CS636 </a>`
- So there are three kinds of URLs in all: absolute, relative, and root-relative. Absolute includes a server hostname, root-relative starts with / and has no hostname, relative starts with non-/ and has no server hostname.
- This URL syntax for forward is special-purpose, not part of the general setup.

# Forwarding, further notes

- Only the request and response objects are handed over in a forward.
- All the other variables are not accessible to the forwarded-to servlet/JSP, unlike in PHP.
- The trick is to attach attributes to request to let them ride across to the target.
- In webapp ch05emailS, we saw that the servlet accessed the DB (just a stub actually), attached a User object as an attribute of the request object, and forwarded to a JSP, which used EL to access the request attributes and generate HTML
- A servlet can forward to a JSP, which compiles to a servlet, or an HTML file, which tomcat knows how to handle, or another servlet in its context.

# Ch05emailS MVS request-response cycle



View: thanks.jsp, composes HTML for response using data passed from the controller in a request variable

Controller: in charge of assembling all needed data for the view

## What about the layers?  Where do they fit in this MVC picture?

The view is pure presentation.

The Model is the DAO.

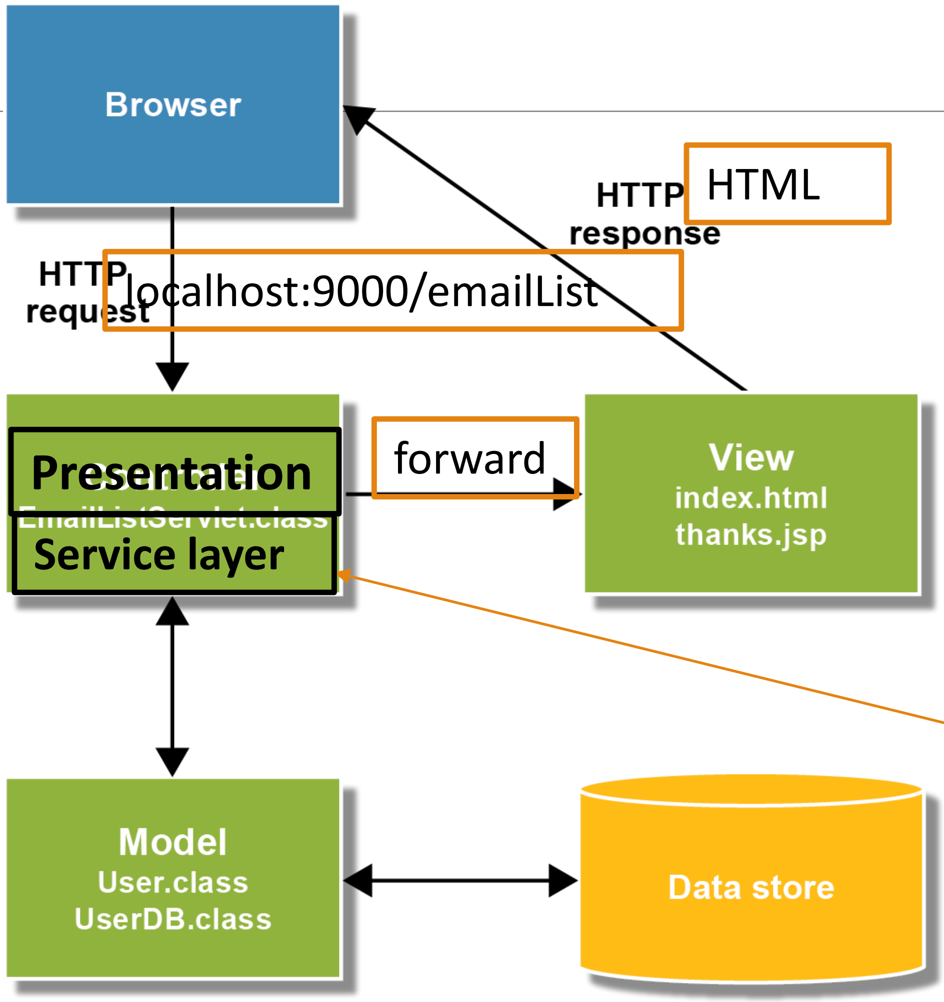
Thus by elimination, the service layer is in the Controller.

But not all of the controller code is service-layer.

The mechanics of accepting the user input is presentation code. Once the user's intention is known, calls are made to the service layer, then on to the DAO/Model.

With the help of the data from the DB, the service layer makes decisions, and finally returns to the presentation code of the controller, which attaches data to the request (the request variables) and forwards the request to the view JSP.

# Ch05emailS Layers



View: thanks.jsp, all presentation layer

Controller: partly presentation, partly service layer

## Now on to Spring Controllers…

- We have seen how to handle the endpoint with path "/emailList" with a servlet that passes a "user" request variable from the servlet to the JSP.
- Also the endpoint "/test" with a hello-world level servlet that just outputs HTML composed in Strings.
- Spring Controllers let us handle endpoints with less coding
- Let's look at the simplest case, the hello-world level, and then one that passes data from the controller to the JSP.

## Look at the simplest Spring controller in pizza3

```
@Controller                                  ← class annotation: this is a Controller bean
public class StudentController {
@Autowired
private StudentService studentService;  ←ref to service API

@RequestMapping("welcome.html")         ←path of incoming request (endpoint)
public String welcome(Model model) {    ←controller method with annotation
  return "welcome";                     ←says to forward to welcome.jsp
}
```

This controller method (a "handler") handles incoming requests to /welcome.html, the starting path of the web app. It doesn't do any computation, just returns "welcome" to tell tomcat to forward to welcome.jsp.

## The Spring DispatcherServlet at work

Execution sequence:

1. GET /welcome.html comes into tomcat (or POST), it locates right servlet, in this case Spring's DispatcherServlet, which has been in existence since the web app started.

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

3. The DispatcherServlet calls StudentContoller.welcome, gets back "welcome", and forwards the request to welcome.jsp.

## Compare to ch05emailS’s TestServlet

TestServlet (code on next slide, from Chapter5slides) handles GETs and POSTs to an endpoint “/test”. It outputs HTML directly for the response instead of forwarding to a JSP.

Execution:

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

GET /test HTTP/1.1 (vs. GET /ch05email/test HTTP/1.1 for shared tomcat’s ch05email)

2. tomcat locates servlet by analyzing URL’s path: / for embedded tomcat, /ch05email for shared

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

4. doGet outputs HTML to responses’s PrintWriter, which is then sent back to the client

## A servlet that returns HTML

```
package murach.email;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="MurachTestServlet", urlPatterns={"/test"})

public class TestServlet extends HttpServlet {

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

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        try {
            out.println("<h1>HTML from servlet</h1>");
        } finally {
            out.close();
        }
    }
```

Line added to original slide, for our setup. Registers endpoint /test for the servlet.

doGet also comes here, because doGet calls doPost

doGet is down here

## Summary on simplest endpoints

We have seen the simplest cases of endpoints using both explicit servlets and Spring controllers.

In both:

- The code has endpoint-registration via annotations:

Explicit servlet: **`@WebServlet(name="MurachTestServlet", urlPatterns={"/test"})`**

(older way, in textbook: use **`web.xml`**)

Spring controller: `@RequestMapping("welcome.html")`

- The code in the method specifies what HTML should be returned, but doesn't provide any variables to make properly *dynamic* HTML.
- So next we want to consider endpoint handlers that do provide variables for dynamic HTML.

# 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`.

## 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.

## 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.

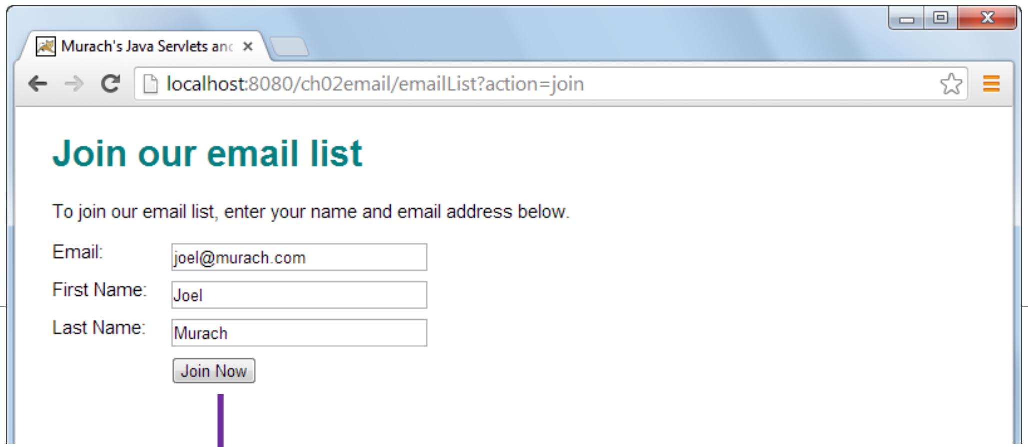
We have covered:

- 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.
- How to use session variables to keep a conversation going with our user.

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



Murach's Java Servlets and ×

localhost:8080/ch02email/emailList?action=join

### Join our email list

To join our email list, enter your name and email address below.

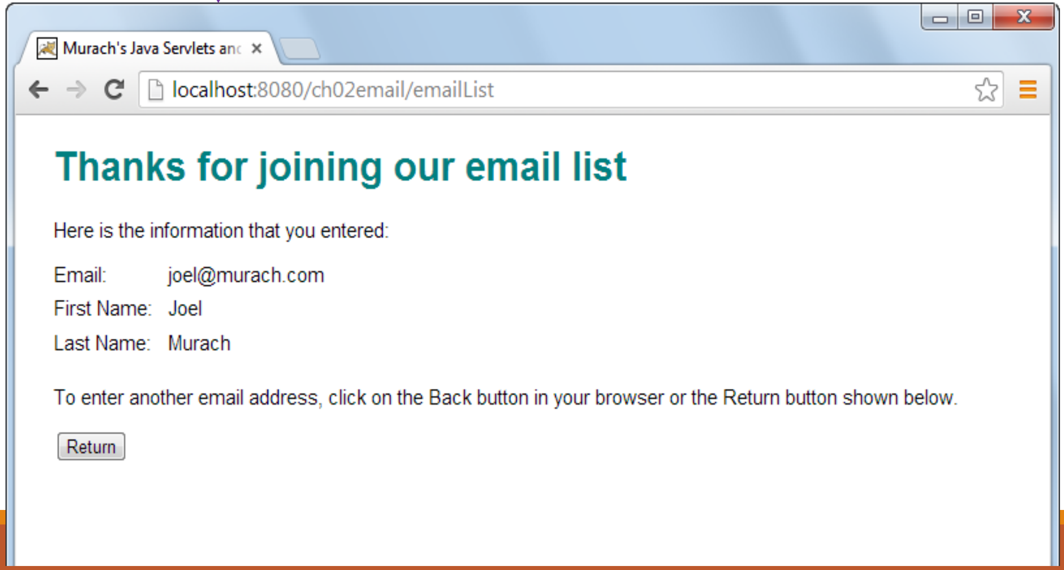
|  |  |
|---|---|
| Email: | joel@murach.com |
| First Name: | Joel |
| Last Name: | Murach |

Join Now

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

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

Request variable: name: “user” value: User object



Murach's Java Servlets and ×

localhost:8080/ch02email/emailList

### Thanks for joining our email list

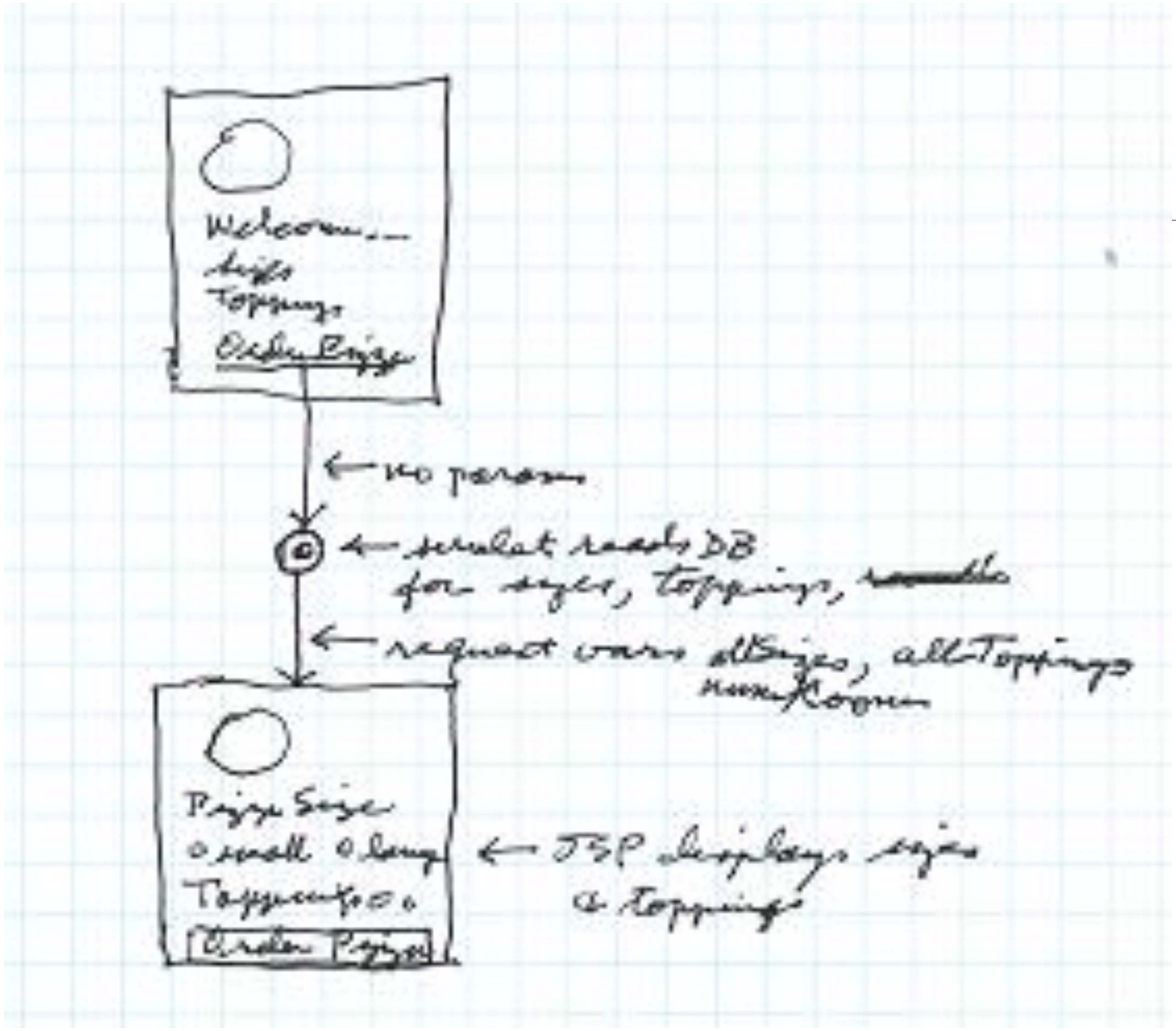
Here is the information that you entered:

|  |  |
|---|---|
| Email: | joel@murach.com |
| First Name: | Joel |
| Last Name: | Murach |

To enter another email address, click on the Back button in your browser or the Return button shown below.

Return

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



Welcome...
Sizes
Toppings
Order Pizza
← no params
← servlet reads DB for sizes, toppings, ~~[illegible]~~
← request vars allSizes, allToppings ~~[illegible]~~ Room
Pizza Size
○ small ○ large
Toppings ○ ○
Order Pizza
← JSP displays sizes & toppings

## Next: MVC webapps with session variables.

- Start reading Chap 7 on Session Variables.
- So far, we have been using "request variables" like the "user" attribute in ch05email and “allSizes” for pizza3’s show-order-form page.
- Request variables are great for communicating between the servlet controller and the JSP it forwards to, since they use the same request object.
- But as soon as the request-response cycle finishes, those request variables and values evaporate.
- In many cases, we need to remember information in the server for longer, from one request cycle to another. That's when we use **session variables**, or of course the database.
- The session variables hang off the "session" object just like the request variables hang off the request object. The session object hangs off the request object, so everything is accessible from the request object.

# Chapter 7: How to work with sessions

**Look at Chapter7 slides (6pp)**