Final Review

Look at syllabus again

We have covered

- Pre-web client-server programming using JDBC, Maven
- Basic web technologies : HTML, HTTP
- Using JPA2 object-relational mapping.
- MVC Web UI with JSP and servlets and Spring Boot.

The layered architecture, role of DB

- We have studied one layered architecture the whole term and showed that it can handle both client-server and web environments, and as a plus, also web services (not officially covered, i.e., not on the final exam).
- We push all the concurrency handling down to the database, and take full advantage of its ability to handle it.
- We have argued in detail that our code does not itself need explicit locks or semaphores, a great advantage.
- In practice, if you see a lot of locks or semaphores in use or proposed, think about putting the data in a database instead.

Scaling up

Recently we have covered its limits: really large sites, esp. in terms of multiple teams of developers, where microservices are now the preferred solution.

But this is much harder to do. Use hardware first to stretch this simple “monolithic” solution.

Recently, covered example of timings for a busy web site, showing that concurrency in the database is much smaller than concurrency among user sessions in a web app.

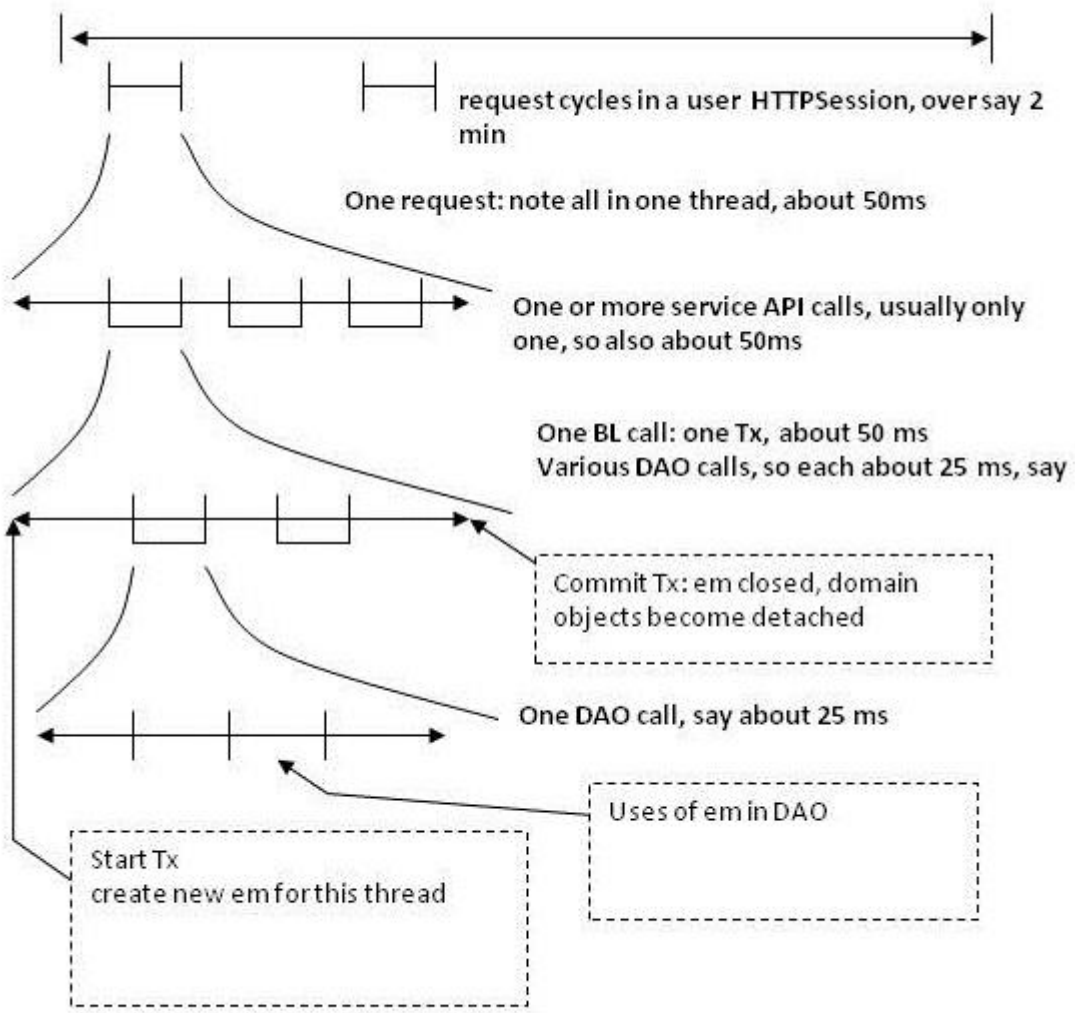
In 1000 concurrent active user sessions, we found only 5 concurrent transactions at a time.

This is good news for anyone worried about overloading the database.

However, it shows what a challenge it is to properly test a webapp for concurrency problems. Most tests end up running without any transaction concurrency.

Next slide: picture of execution at various levels

Here is a picture that tries to summarize the different time periods involved in web app execution: For non-JPA, replace "em" with Connection.



←User session across multiple requests

←request cycles

← service API calls by presentation

←one service method execution, with its transaction start and commit

em or connection is thread-private

← one DAO call uses em or connection for that thread/request

Short lifetime objects: in one request-response cycle

- Request, response objects, created by tomcat before calling doGet or doPost
- The Connection object, whose lifetime is one transaction in our setup (but in fact it goes back into the DataSource Connection pool)
- The domain objects fetched from the database for this request, and private to this request.
- Domain objects created for this request, and private to this request.

Objects in the tomcat JVM for pizza3

Since tomcat came up: **long-lived objects**

Tomcat's own classes

The Spring beans, including the API objects, the Controllers, SysTestServlet, the DataSource, its Connection pool, containing live Connections for later use.

Spring's DispatcherServlet, with its table of url paths to handler methods found by reading annotations

Intermediate lifetime app objects: session objects and things attached to them (session variables), last about 20 minutes after last request in that session.

For pizza3, the StudentBean is a session variable. It holds the room number, which represents the user in this app.

Short lifetime objects

Everything associated with one request-response cycle, less than about 50 ms we hope, and all private to this request.

- Request, response objects, created by tomcat before calling doGet or doPost
- The Connection object, whose lifetime is one transaction in our setup (but in fact it goes back into the DataSource Connection pool)
- The domain objects fetched from the database for this request, and private to this request.
- Domain objects created for this request, and private to this request.

Example of ordering pizza : two request cycles: first request cycle

- User clicks "Order Pizza" link, which has URL "orderForm.html", so HTTP request GET orderForm.html is generated to the tomcat server, where the Spring boot dispatcher servlet has set up appropriate servlet URL mappings, including one for orderForm.html.
- StudentController method displayOrderForm has annotation @RequestMapping("orderForm.html"), so gets called by Spring's dispatcher servlet.
- diplayOrderForm calls service layer twice (so 2 txns) to get topping names, size names, makes them request vars, forwards to JSP with order form
- The Order form JSP accesses the topping names and size names request variables, and the room number in the session

Example of ordering pizza : two request cycles: second request cycle

- User fills out form, submits it, generating POST (actually GET) orderPizza.html to tomcat
- StudentController method orderPizza has annotation @RequestMapping("orderPizza.html"), so gets called by Spring's dispatcher servlet.
- orderPizza code gets params for user choices (names of toppings, sizes, possibly new room no), then calls service layer: makeOrder (...), which creates new PizzaTopping and PizzaSize objects, then PizzaOrder object, and calls insertOrder in the DAO. Only one service API call here, so one txn. Finally, forwards (actually redirects) to studentWelcome.html
- studentWelcome.html is matched to method displayWelcome by the dispatcher servlet, so sets up variables for that page and forwards to studentWelcome.jsp, so that page is displayed to the user.

Compare Client-server and Web Apps

Client-server	Web App
One singleton graph per client, lifetime of app	One singleton graph for web app lifetime
Single-threaded execution in Java code	Multi-threaded execution in Java code, but argued that it's safe from race conditions.
One database holds all shared, changeable domain data—shared between clients	Same, though clients are now called web app users.
One JDBC Connection for app lifetime with plain JDBC, or use DataSource here too, though pool not needed.	One Connection for each txn, in fact coming from a connection pool maintained by the DataSource.

Client-server	Web app
Domain objects are POJOS and short lived, with exceptions for some immutable objects and/or user-private objects (e.g. cart)	Same. Short-lived means within a single request cycle.
Stateless service layer, also DAO layer. Call-down layers.	Same
Can have state in presentation layer, i.e., variables in the client app that save info from one action to another for a user.	Can have state in presentation layer, i.e., session variables that carry info from one request to another for a user.
Transactions start and end in service layer, since defining app actions.	Same
Service layer API defines what the system can do, most important API	Same
Used Maven to handle dependencies on various jar files.	Same.

Book coverage for Final Exam

For pre-midterm, see Midterm Review

Murach since midterm:

Chap. 5, servlets. We are using @WebServlet(...) instead of web.xml, so skip web.xml details and look at those annotations instead, as in ch05emailS, ch07downloadS, and ch07cartS projects. We are also using embedded tomcat, after a brief adventure with a shared tomcat on pe07.

Chap 6, JSP: can skip pp. 184-189 on topics only needed for Model 1 apps

Chap 7 Sessions to pg. 207, then can skip to pg. 224 and read to end of chapter (Download app, available as ch07downloadS in our form)

Chap 8 EL to pg. 263

Book coverage for Final Exam, cont.

Chap 9 JSTL to pg. 277, skip 278-279, read 280-285, skip 286-289, read 290-end of chapter (Cart app)

Chap 13 JPA See JPA2Notes.html for notes on this chapter. But not on final exam.

Chap 22 as basis of Music project

Pg. 639: Domain class diagram doesn't show id fields of domain classes, but they are in Murach's domain classes. We have a Track class too.

Pg. 652: The Controller layer. Murach is not using Spring, so doesn't have its dispatcher servlet to handle incoming requests and call the right controller method as we do. Instead, the controllers are themselves servlets, handling incoming requests according to their URL patterns specified in web.xml. For example, the CatalogController shown here is handling incoming requests to /catalog/product/* (servlet context relative), as shown by the servlet mapping on pg. 659, in web.xml.

Murach, cont: Pg 657: Directory Structure

Make sure to understand our deployable directories in music3:

```
──webapp
    ├──includes  (may be empty, just a placeholder)
    ├──sound
    │    ├──8601
    │    ├──jr01
    │    ├──pf01
    │    └──pf02
    ├──styles  (may be empty, just a placeholder)z
    └──WEB-INF
         ├──admin   jsps for admin pages
         └──jsp     jsps for user pages
```

Putting the jsps under WEB-INF hides them from direct web access. Here the sound files are directly available. Note that the classes directory doesn't show up here--it's in the target directory, and then gets built in to the fat jar, or accessed by mvn spring-boot:run and made available to the running executable

Murach coverage, last part

Pg. 660 web.xml has security setup we didn’t cover, and we're not using web.xml, but instead @WebServlet annotation in our one actual servlet, SysTestServlet, and letting Spring handle the relationship to tomcat for most requests.

Pg. 661 context.xml: way to get tomcat to create the needed DataSource. We're using Spring for this, an easier way.

Pg. 663: DB diagram: we have track table too.

Pg. 664: SQL script is mysql-specific. We have a portable version.

Pg. 666-669: Murach's ProductDB class has static methods. Our ProductDAO has object methods, and is used to create a singleton object, the Spring way.

Chap. 23 Apps of the Music website: Shows the UI. We are using a simplified UI for user actions.

Homework since the Midterm exam

HW4: pizza2 (not on final), requests to shared tomcat on pe07 at port 8080, simple JSPs, embedded tomcat at home running EMailList servlet (ch05emailS), running ch05emailS on pe07, trying out pizza3: web UI, SysTestServlet

HW5: ch07cartS and ch07downloadS projects for embedded tomcat, compare pizza1 and pizza3, analyze session variable in pizza3.Write a simple controller for pizza3 to get it to handle /index.html. Find the URL to execute SystemTest via the dispatcher servler by reading the annotations and methods in AdminController. Looking at the controller handlers to see how they forward to JSPs or other handlers (or do a redirect, but we're not covering that officially).

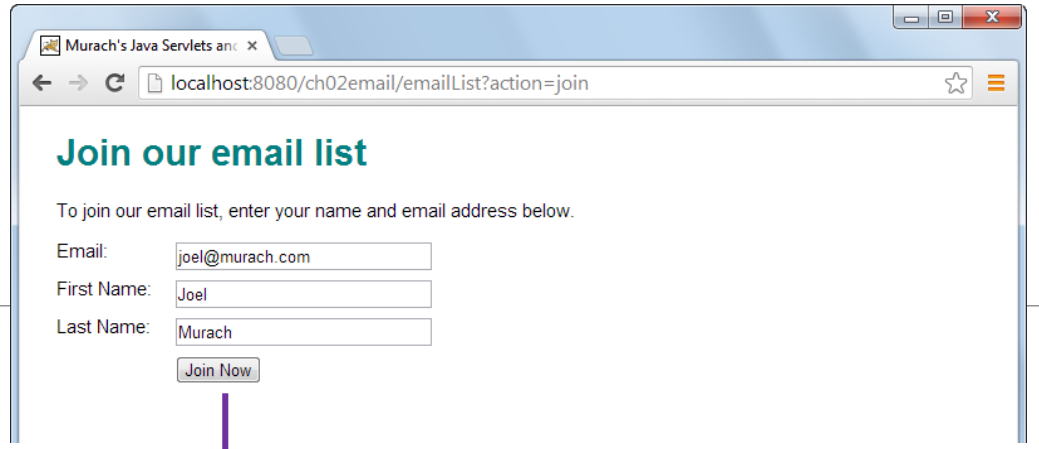
Important Servlet Examples: parts of project, really

ch05emailS @WebServlet says urlPattern is /emailList, application.properties says port=9000

1. Initial request to localhost:9000/emailList reaches the servlet, which forwards to index.jsp that generates a form. Could be index.html instead: no EL is needed for a blank form.
2. That form submission with action=*"emailList"* sends a request back to the servlet
3. The servlet interprets the form data coming in request parameters, creates a User object, attaches it to the request as a request variable and forwards to "thanks" JSP page showing the submitted data using EL.

Request parameters bring user input to the servlet

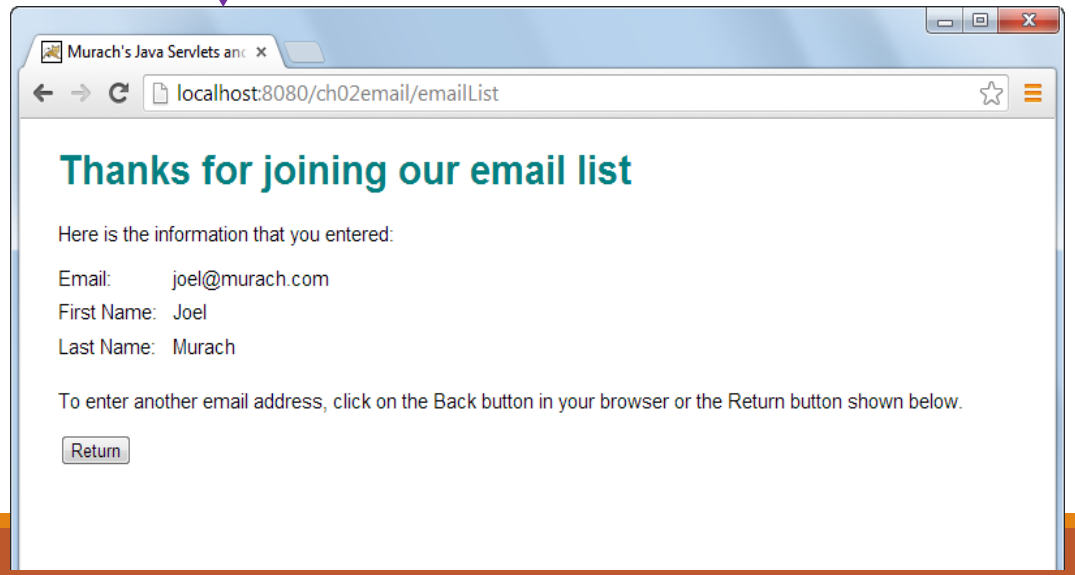
Request variables live and die in one request-response cycle



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

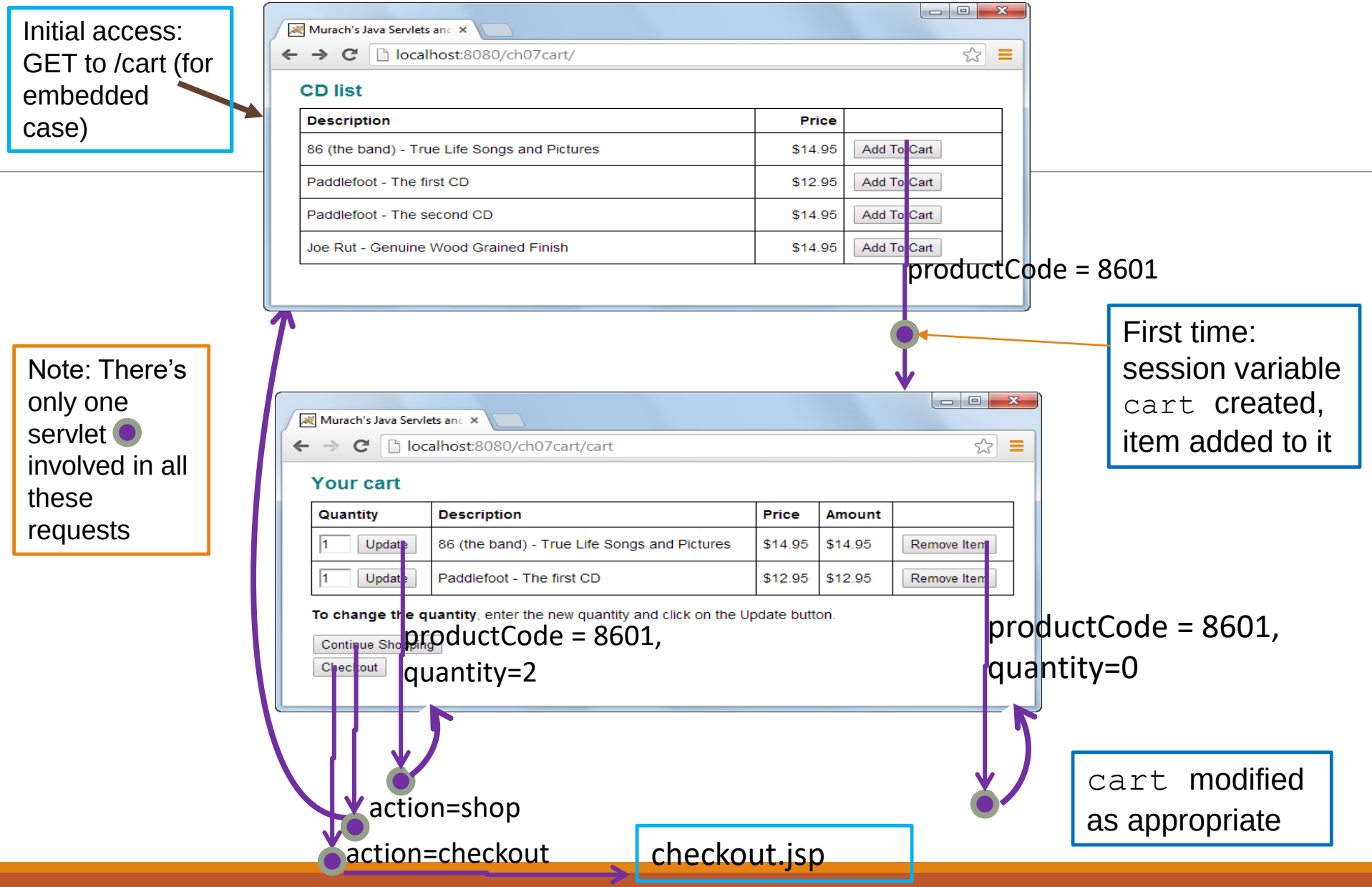


Important Servlet Examples: cart servlet

ch07cartS @WebServlet says urlPattern is /cart, application.properties says port=9004

1. Initial request to localhost:9004/cart reaches the servlet, which forwards to index.jsp that generates the page of links for each product
2. User clicks a link such as Add To Cart, generating a request to localhost:9004/cart?productCode=8601, reaching the servlet
3. The servlet gets the productCode out of the request param, adds that product to the Cart, creating it if necessary. Cart is a session variable. Then the servlet forwards to cart.jsp
4. cart.jsp displays the cart, and various controls on it….

Page flow with request parameters shown coming into servlet



Important Servlet Examples: Download Servlet

ch07downloadS @WebServlet says urlPattern is /download, application.properties says port=9001

Initial request to localhost:9001/download reaches the servlet. The servlet forwards to the first page, which shows list of albums (CDs), each with a link that carries a parameter for the product code.

When the user clicks a link, the request goes back to the servlet, and the servlet code checks if the user is already known (by cookie here), and is sent to the appropriate CD page if so, but if not, is sent to a registration page.

After registration, the user is sent on the previously-chosen CD's page. To remember what CD the user was interested in, across the registration process, the servlet uses a session variable for the productCode. We haven't made a page flow for this.

The CD-specific page has links to sample mp3's. Since there are several ways to handle playing those mp3s, we'll skip covering that.

Final Notes

pizza3: We have analyzed this to see that the room number is the one session variable here, in hw5. We made a page flow back in hw3.

Spring Boot: a big help for Java web apps, and gives us a way to use Spring without a huge learning curve.

Some of the credit belongs to Maven: it has relieved us of the job of hunting for the right combination of libraries for Spring and other software.

But don't forget the database: it relieves us of dealing with the various threads concurrent access to shared data.

Hopefully you'll be able to use some of this powerful software and tools in real projects, and of course as prep for the software engineering courses.