Slide 1

DataSource,
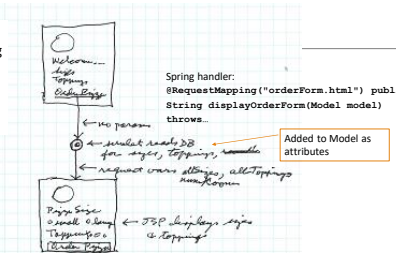# Multithreading Concerns

1

Slide 2

Last time: Spring Controllers

- We saw that Spring controller handlers help with collecting up incoming UI in the URL parameters
- And give us access to the request object too if we want it.
- Here are the examples in pictures…

2

Slide 3

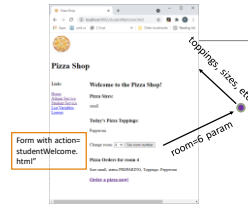Rough page-flow picture for Spring controller that sets up the pizza order page in pizza3

Spring handler:
@RequestMapping("orderForm.html") public String displayOrderForm(Model model) throws…

Added to Model as attributes

3

Slide 4

StudentWelcome page Form submission

Form with action= studentWelcome.html"

toppings, sizes, etc.

room=6 param

```
@RequestMapping("studentWelcome.html")
public String displayWelcome(Model model,
        @RequestParam(value = "room",
        required = false) String chosenRoom,
        HttpServletRequest request) throws
ServletException {

    <put chosenRoom in StudentBean, a session variable>

    <load toppings and sizes, etc. into Model>

    return "jsp/studentWelcome";
```

4

Slide 5

Order pizza

Form with action= *orderPizza.html*

params sizes=small& toppings=pepperoni&room=1

toppings, sizes, etc.

```
@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 …
```

Can pick up multiple toppings

5

Slide 6

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.

6

Session variables in pizza3

Last time we saw code putting the newly-set room number into place in a session variable in the displayWelcome handler (slide 20). That session variable was a StudentBean object.

A **StudentBean** is created for a user in displayWelcome and made into a session variable.

Other controller methods should check for the existence of the bean and forward to /studentWelcome.html and thus to displayWelcome if the session variable is not there: this is not in pizza3 but should be in music3. (it's already there in the admin pages)

Although StudentBean has "bean" in its name, it is not a Spring bean, just a Java bean. Note it lack of annotations.

It carries the crucial room number for the current user, the variable that needs to be remembered across several requests. See problem 4 of homework 5.

It could be replaced by a simple Integer object for the room number.

7

DataSource and Connection pooling

Connection pooling is discussed in Murach, pp 464-469.

It is important for web apps because they need Connections for such short periods of time that the Connection startup time with the database looks huge and would impact performance.

The Connection pool managed by the DataSource implementation has hot standby Connections that the web app can borrow and return.

We want one of these objects for the database in current use, so for h2 or mysql or Oracle.

Note that DataSource itself is just a Java interface. It's the implementation of it that does the real work, and needs the driver code to do this.

8

Using a DataSource

With a DataSource object ds, you can get a Connection by ds.getConnection().

At transaction start, a new Connection is pulled out of the pool by Connection connection = dataSource.getConnection(); Autocommit is turned off by connection.setAutoCommit(false)

A simple JDBC webapp can use this Connection normally, and must close it when done.

That "close" operation doesn't actually close the connection in the OS, but rather puts it back in the pool for another request to use.

The close (connection.close()) happens in DbDAO.commitTransaction, after connection.commit();

Also in rollbackTransaction, after connection.rollback();

9

Getting a DataSource bean via Spring: it's easy!

We are depending on Spring to set up an appropriate DataSource for us.

All we need to do is code an @Autowired DataSource reference (in one or more of our Spring beans), and Spring creates a DataSource bean (singleton, as usual) based on properties in the "active profile" (application-mysql.properties for example).

Almost magic! Here's a snippet from pizza3's DbDAO:

```
  @Repository
  public class DbDAO {
      @Autowired
      private DataSource dataSource;
```

When our app comes up, Spring reads application-mysql.properties (or whatever DB) and creates the needed DataSource objects using the JDBC database parameters listed there.

It needs access to the DB drivers to do this, so they are dependencies listed in the pom.xml.

10

Specifying the Spring profile, and thus the database to use in pizza3 and music3

- To make the web app read application-mysql.properties, we specify the "profile" called mysql by putting the right syntax in the command line.
- See runJarByProfile.cmd (or .sh) for details for the command line case.
- If running from eclipse, you can set up a "Run Configuration" with "VM argument" (not a "program argument") -Dspring.profiles.active=mysql.
- You can see this same string in runJarByProfile, before the jar file name, making it a VM arg.
- The argument after the jar (SystemTest or whatever) can be put in the program arguments section.
- See http://www.avajava.com/tutorials/lessons/whats-the-difference-between-program-arguments-and-vm-arguments.html.
- In that run configuration, put the main class to cs636.music.presentation.SBApplication (or pizza instead of music).

11

Spring Profiles, cont.

eclipse, default case: If you don't use a Run configuration, the system uses the default profile, application.properties in src/main/resources (pizza3's uses mysql, music3-setup's uses h2).

If you run SBApplication from Package Explorer by right-clicking and choosing Run As Java Application, you'll be using mysql for pizza3, h2 for music3.

Spring profiles work the same way with or without tomcat in use, i.e., client-server or web app. The Spring developers try very hard to keep the various features mix-or-match.

We could rework pizza2 a little bit to use profiles: it has a special-coded way of getting different Datasources for each of our databases.

Spring processes the profiles very early in its startup code (in app.run in SBApplication), before it creates all its beans. Thus the database url, etc. is set before it creates the DataSource bean.

12

2

Threads and JDBC Connections in a Web App as we have set it up

Each transaction is running in a Java thread, the same thread for the whole transaction and in fact the same thread for the whole request/response cycle, in tomcat's JVM.

In pizza3 and music3, we are using a DataSource object to provide JDBC Connections by conn = dataSource.getConnection(). The DataSource is necessarily a thread-safe class, i.e., multiple threads can be trying to get new Connections concurrently without trouble.

We save the DataSource object as a field in the DAO, specifically in DbDAO of pizza3 and music3.

At transaction start, a new Connection is pulled out of the pool by Connection connection = dataSource.getConnection(); Autocommit is turned off by connection.setAutoCommit(false), and the Connection is returned to service-layer caller, which is just starting one of its service calls that accesses the database.

`13`

Transaction methods now

```
public Connection startTransaction() throws SQLException {
    Connection connection = dataSource.getConnection();
    connection.setAutoCommit(false);
    return connection;   ← returned to service layer...
}
```
Passed down from service layer

```
public void commitTransaction(Connection connection) throws SQLException
{
    connection.commit();
    connection.close();
}
// also rollback, rollbackAfterException
```

`14`

The Connection variable: homed in a local variable of each service method

The service-layer method puts the Connection it gets from startTransaction() in a local variable, which is always thread-private, and uses it for various calls into the DAO.

It is a little impure to allow a JDBC class like Connection to be used in the service layer, but it's not domain data, so we are maintaining the important rule of a stateless service layer.

When the service-layer method is done with its action, it calls dbDAO.commitTransaction(conn), and in that call the Connection is closed, returning it to the pool

If we start another transaction in the same request cycle, we'll get another Connection from the pool, but usually we have only one transaction in a certain request-response cycle.

All transactions happen strictly inside service calls, so inside individual request cycles. They can be concurrent, however, so we need to argue that they don't interact badly.

`15`

Service method now: how it uses the Connection

```
public void removeTopping(String topping) throws ServiceException {
    Connection connection = null;
    try {
        connection = dbDAO.startTransaction();
        menuDAO.deleteMenuTopping(connection, topping);
        dbDAO.commitTransaction(connection);
    } catch (Exception e) {
        dbDAO.rollbackAfterException(connection);
        throw new ServiceException("Error while removing topping ", e);
    }
}
```
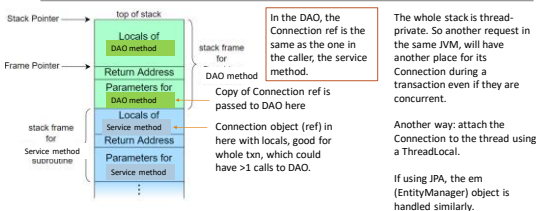Connection is a local variable

All DAO methods take a Connection argument now

• This change in storing the Connection is because we are now running **multithreaded** in the tomcat JVM.
• Local variables are per-thread variables, safe from other threads.

`16`

Call stack (edited from Wikipedia)

Stack Pointer → top of stack

Locals of DAO method

stack frame for DAO method

Frame Pointer → Return Address

Parameters for DAO method

Locals of Service method

stack frame for Service method subroutine

Return Address

Parameters for Service method

In the DAO, the Connection ref is the same as the one in the caller, the service method.

Copy of Connection ref is passed to DAO here

Connection object (ref) in here with locals, good for whole txn, which could have >1 calls to DAO.

The whole stack is thread-private. So another request in the same JVM, will have another place for its Connection during a transaction even if they are concurrent.

Another way: attach the Connection to the thread using a ThreadLocal.

If using JPA, the em (EntityManager) object is handled similarly.

`17`

Concurrency and "thread-safe" code

We have just argued that the Connection is thread-private, so safe from multithreading meddling.

But there is only one DataSource object for the web app, so it is subject to simultaneous or near-simultaneous calls from different threads.

Luckily this is a well-known problem, solved by the programmers of the DataSource code, so the DataSource is "thread-safe". It has internal mutexes to protect itself, forcing the calling threads to go through some critical code in single-file.

Another class that is shared across the web app and subject to access by many threads is the DispatcherServlet, and it is also thread-safe.

`18`

`3`

Tomcat Web app initialization

How does the tomcat web app start up?

We know that the Spring bean setup is independent of tomcat itself, since it works fine for the case we turn tomcat off entirely.

When Spring boot runs tomcat (after any CommandLineRunners), it (in effect) calls tomcat's main() and becomes the tomcat server.

We know that servlets (here the Spring dispatcher servlet) have an init() method that tomcat calls once when the servlet starts. That servlet code is Spring Boot code that processes the @RequestMapping annotations.

Running as tomcat, the server creates a group of threads in which to handle individual requests, to allow concurrent requests to execute. A particular thread will execute the whole request-response cycle.

19

Notable Spring Beans in pizza3 and their class annotations

The following are all Spring class annotations unless otherwise noted. For more info, see article at DZone on these.

The following annotations are @Component or specializations of it:
- CommandRun (@Component): top-level presentation code
- Service (@Service) and DAO (@Repository) objects, as in pizza2
- StudentController and AdminController (@Controller)

Note: We need to put @SpringBootApplication(scanBasePackages = { "cs636.pizza" }) on the main class to enable the scan for these annotations.

DataSource implementation, part of Spring infrastructure. Note that the DataSource interface is a Java interface Javadoc. It's not a @Component but it is a Spring bean, registered in the ApplicationContext

SysTestServlet (@WebServlet) WebServlet is a Java EE annotation Javadoc. We get Spring boot to pay attention to it and treat is as a component by annotating the main class with @ServletComponentScan(basePackages = "cs636.pizza")

20

The bigger picture...
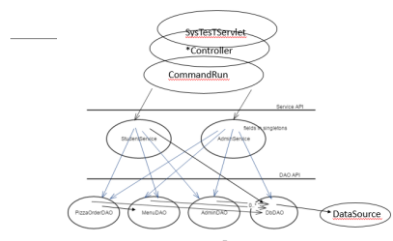
21

Getting the beans into action

All these singleton beans are created before tomcat code runs, so their creation is safely done before any incoming requests are honored.

As detailed on a previous slide, the SBApplication program morphs into a tomcat server after setting up basic beans and running the CommandLineRunners if any.

Then tomcat starts the DispatcherServlet, which has an init() with Spring Boot code that interprets the method annotations @RequestMapping, so it knows how to dispatch future incoming requests by calling the right handler for them.

These handlers are in already-created @Controller beans.

Finally, tomcat declares itself up and running, and allows in requests. Note that all the autowired fields in these beans (the bean-to-bean refs) will never change during this normal execution time.

22

Other objects might be changed by multiple threads: we have to be careful

Recall Murach's warning at the end of the Chapter 5 slides
```
public class EmailListServlet extends HttpServlet {
   // declare an instance variable for the page (or servlet)
   private int globalCount; // not thread-safe  ←This variable can be
                            changed by multiple threads, i.e., multiple requests coming in
```
Similarly, in Spring Boot Controllers, similar access is possible:
```
public class StudentController {
@Autowired
private StudentService studentService;   ←  OK, never changes after init.
private int globalCount;  // not thread-safe [added line]
```

- Remember, local variables are your friends in a multithreaded environment!
- Fields in a controller need to be thread-safe. We can in fact avoid using extra fields in our music3 controllers, as was done in pizza3.

23

Recall our rule about having a stateless service layer (and DAO layer)

```
@Service
public class StudentService {
@Autowired
private PizzaOrderDAO pizzaOrderDAO;  ← OK, constant after init.
```
```
private int globalCount;   // [added line] not thread-safe. Also counts as state in the
                           service layer, although not directly related to domain matters...
```
```
private int productOrderCount;   //[added line] not thread-safe and counts as
                                 domain state, so disallowed two ways
```

Can we run a thread-safe counter? Yes, but not with a simple int variable. Need a mutex involved. See AtomicInteger in the JDK.

What about that Connection variable we're using in the service layer now?

Answer: that was a local variable, not a field. Local variables are thread-private, and only last for the duration of their method call.

24

4

Generalize these examples to all beans…

All the Spring beans are singletons, so they are all shared between the many incoming concurrent requests of a busy web server.

We see that we need to be careful about their fields:

- They can be refs to other autowired beans without worry, because we have seen that these are finalized before the first request is let in.
- Otherwise, they need to be thread-safe or nonexistent (more likely for music3)

We are not covering how to write a thread-safe class. Instead, we want to show how we can write a concurrency-safe web app without writing our own thread-safe classes.

Of course we are depending on the thread-safe nature of DataSource and the Spring infrastructure code such as the DispatcherServlet.

25

When might we need a thread-safe field?

Here is a quick example

Suppose we are using web services to collect stock prices in real time.

So we aren't using a database for this data access.

We design a Map<String, StockInfo> to hold our data in the DAO layer, in a field, instead of (or in addition to) a DataSource. The String is the stock symbol, like "GOOGL" or "IBM".

But this is a mutable structure, so needs to be thread-safe.

This case is really easy to solve: just use ConcurrentHashMap<K,V> instead of HashMap<K,V>

So again we got away without writing our own thread-safe class!

26

Debugging music3

Debugging using System.out.println

With Spring Boot, we are just running our app from its main() as usual. Thus the output of println's shows up in the Console window as usual in eclipse, or the command-line window for command-line execution. (This is much simpler than with running on traditional non-embedded tomcat).

It is perfectly possible to debug music3 just using System.out.println output added in areas of problems. Also e.printStackTrace() if a stack trace doesn't happen.

Our layers help localize behavior in expected places. If you add a println to each method outputting the method name, you can localize errors to a particular method.

If the execution fails, it may be because you have another instance of the program running: the port can be used by only one process, and H2 can only serve one process for a particular file listed in its JDBC URL.

27

Debugging with the eclipse debugger.

With embedded tomcat, debugging is just the same as with ordinary Java programs.

Set breakpoints by clicking on the far-left-hand ribbon in the source editor window.

Run the debugger by right-clicking on SBApplication and using Debug As> Java Application.

For help on this, see Help>Help Contents>Java development user guide>Tasks>Running and Debugging>Local Debugging, and see various topics.

28