Spring, esp. Spring Boot

Web apps using tomcat

We have discussed servlets and tomcat, the servlet-capable web server, and saw how tomcat can handle incoming requests to a "service endpoint", a particular URL.

From slide 8 of class 13, about the pizza3 webapp:

- Consider the access to welcome.html, using URL http://pe07.cs.umb.edu:9002/welcome.html
- Here GET command is GET /welcome.html HTTP/1.1, done after TCP connection to pe07.cs.umb.edu on port 9002.
- Tomcat interprets the local path /welcome.html, sees that / belongs to the servlet of pizza3, and checks if welcome.html is registered as a path of interest ("endpoint") to that servlet. It is.
- So tomcat hands off the handling of this request to the servlet code.
- Obvious loose ends here: how are endpoints registered, how does the handoff work?
- Quick answers: via web.xml or equivalent annotation, by call to doGet() in the servlet.

First Spring Boot example: ch05emailS

- We can run embedded tomcat by just running the app that embeds it.
- The example app, ch05emailS (S for Spring) is available. See run.sh and run.cmd for running it. It uses a servlet explicitly, very much like the Murach's example in Chap. 5. ch05email.
- Port 9000 is what we'll use for ch05email on home systems. Find your pe07 port in CS636Ports.html
- Follow the README in ch05emailS's directory
- Its service endpoint (home case, or on pe07 case) is http://localhost:9000/emailList
- The URL doesn't contain "ch05emailS" because the app owns the root of the website in the embedded case, i.e., it has free use of top-level URLs like /emailList.
- This uses Spring Boot only for jar version control and embedded tomcat. It doesn't use "Spring beans" as we are expecting to do for the pizza and music projects. It doesn't use "Spring controllers" to simplify handling web-requests as we also want to do.

Spring Boot Roadmap

- Spring Boot is now the leading framework for Java web apps: see IBM page on it
- It's really Spring, made easier to use by having a tool called Spring Initializr to set up projects to be built using Maven or Gradle

Major Spring features we will cover, in this order:
1. Using its supported embedded tomcat for much easier service/webapp development
2. Using Spring beans for our singletons, with "auto-wiring" get Spring to set up the basic dependencies
3. Using Spring controllers for our service endpoints (URLs that access the web app), so we no longer need to write full servlets, only the code needed to handle the various requests

FYI coverage: Spring JPA Repositories with their no-code finders.

First Spring Boot example: ch05emailS

- The code for this example is very close to that of Murach's ch05email described in the book. It uses a servlet to receive the requests, and "forwards" to thanks.jsp. Thanks.jsp is an auto-generated file that is returned to the user’s browser.
- Embedded tomcat is a boon for web development, but it isn't expected to be a production vehicle like shared tomcat.
- The code only needs to handle these requests, so it's easy to add a feature.
- So once a web app is working based on development on embedded tomcat, it can be rebuilt for the "real" tomcat. We aren't going to bother with this last step, however.
- An example of an additional feature of a production tomcat is the ability to add a servlet to a running system. On embedded tomcat, we would have to restart the program for this.
Book coverage

Readings on Servlets: Murach Chap 2, then later tackle Murach Chap. 5.

Note that the email list examples in Chapter 2 and Chapter 5 are identical in Java code, and almost identical otherwise.

So start by reading Chapter 2 and then continue in Chap. 5.

Look at the slides for Chap 2.

Later, Chap. 6-9.

Servlet processing of formdata

> We need to understand how an incoming query string is processed, so the servlet code can get the user input items.

> When tomcat receives this GET or POST request, it parses the parameters and attaches them to the request object provided to the servlet in the incoming doGet call, doGet(request, response)

> In the servlet, we can pick up the parameter value from the request object by

```
String firstName = request.getParameter("firstName"); // makes firstName = "John" in this case
```

> See pg. 139.

How does the servlet pass on this User object to thanks.jsp (i.e. its servlet)?

> First, the servlet attaches the User object to the request as a request attribute. Specifically, it creates a new User object named user to represent the person joining, and does:

```
request.setAttribute("user", user); //create request attr
```

> Then the servlet does a "forward" to thanks.jsp. This invokes the servlet for thanks.jsp directly in the server, and passes the request and response objects to that other servlet. (Recall that each JSP compiles into a servlet).

> The thanks.jsp servlet can access the attributes of the request object. Once attached to the request object, the User object becomes a "request variable".

How does thanks.jsp get the User info out into its response HTML so the real user can see it?

> The short answer is "using EL", the JSP expression language, which is covered in Chap. 6 [intro] and Chap. 8. EL lets us access request variables easily, including dotting into them to get their properties.

> We saw some examples of EL in class 12: look for {$…}

```
${1+2+3}<h1>Today's date is ${date} </h1>
```

> In thanks.jsp, we see

```
${user.firstName},
```

> and this compile to user.getFirstName() in its servlet, where user is a request variable:

```
User user = request.getAttribute("user");  // extracting req. var.
```

Parts of ch05email (AKA ch02email)

- Pictures of app’s pages: pg. 35 (also good for ch05email)
- First page: index.html: pg. 37 (also good for ch05email)
- Servlet code: pg. 41 (one line (class annotation) different in ch05email)
- web.xml: pg. 43 (but we will use that class annotation for this in ch05email)
- User.java: pg. 45 (also good for ch05email)
- Picture of second page in browser, method="get" case, showing URL with query string: pg. 135 (also relevant for ch05email)

A servlet is at the heart of a Java web app

> We have seen how a servlet can receive requests, and use JSP to create a nice response in HTML. This is an important pattern to understand.

> That doesn’t mean we have to code servlets in our projects—they are somewhat low level.

> Spring allows us to abstract the ideas of service endpoints and the code that needs to execute on their behalf, into Spring controllers. Of course there has to be a servlet underneath the covers.

> But before we launch into that, let’s explore the most classic Spring constructs, the Spring beans that we have foreshadowed when we looked at the singletons (in the “big picture”) we were using.
Spring beans

Note that Spring Boot is just Spring with some help doing its configuration, otherwise a big chore. Plus the capability of running tomcat as an embedded server.

We have already been using the version control of Spring Boot in pizza2, with the help of maven’s ability to catalog the jar files by version.

See pizza2’s .pom.xml. In particular, the <parent>

Look at Intro to Spring at DZone (just the intro paragraphs).

We have been using the DI dependency injection/IoC ideas since pizza1: create DAOs, then hand them to service objects, so the service objects never do “new DAO()”.

Now all those singletons are "beans", Spring-managed beans in the Spring container. We let Spring create them for us, based on annotations we write.

Crucial terminology for Spring

• bean: a POJO or Java Bean, in our use (and other simple setups) a singleton object
• Spring container, or Bean Factory, or ApplicationContext: collection of beans for an app
• Component: a bean (singleton), whose class is marked with a class annotation @Component, or one of its specializations
  @Service (for service objects),
  @Repository (DAO objects)
• @Controller (web app bean, in presentation)

Note that Spring projects are expected to have a service layer, not just a DAO layer as we have seen with Murach’s code.

Our Spring Boot pizza projects

• Pizza2S: like pizza2, this is client-server with JPA
  • This version uses Spring Boot’s support of transactions and JPA, including JPA Spring Repositories with their amazing no-code DAOs.
    • You don’t need to study this one, it’s PFI
  • Pizza3: like pizza2, this just uses JDBC directly in the DAOs
    • This version uses Spring Boot’s web support to do the needed web request controllers, and we will use embedded tomcat.
    • This one is the model for Project 2, so needs study.

Creating the beans

Now each bean class is annotated with @Component or one of its specializations:

*Service.java has @Service
*DAO.java has @Repository

There is one more @Component-marked class: ClientRun, which we could put at the top of the diagram in presentation, with refs to StudentService and AdminService.

At app startup, Spring searches the classes and creates beans for the ones with these class annotations.

We used to create the singletons in PizzaSystemConfig: now that code is gone.

How does Spring know what classes to search?

The top-level annotation (on the class with the main()) in it tells it where to "scan" the annotations:

@SpringBootApplication (scanBasePackages = { "cs636.pizza" })

public class ClientApplication {
    public static void main(String[] args) {
        ...}

    Note that this class is the one you can select in eclipse, and right-click, Run as Java Application, to run with H2 and SystemTest.

    How do we get Spring to create the needed bean-to-bean references?
How do we get Spring to create the needed bean-to-bean references?

The references between classes are marked by @Autowired.

For example, in StudentService.java:

```java
public class StudentService {
    @Autowired
    private MenuDAO menuDAO;
    @Autowired
    private AdminDAO adminDAO;
    @Autowired
    private DbDAO dbDAO;
    @Autowired
    private BedDAO bedDAO;
}
```

Spring finds an @Autowired annotation on a field, finds the singleton of that class, and fills in the field with its ref. This all happens at program startup, in main of ClientApplication.

Mystery: how do we regain control here?

We just saw execution disappearing into Spring code, eventually returning, and then the program exits, since this is main.

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.

This interface is a "functional interface", i.e., has a single method traditionally called "run" but no longer required to (to accommodate lambda functions), with String array arguments (like main) and void return value.

```java
void run(String... args) { // required method, traditional form, how we'll do it }
```

How can we specify which database to use?

This has to be done really early, so that the right DAO objects are created in the bean-creation phase.

In pizza2, the args are passed at the start of main to get the database params, before Spring Boot gets control. Then those parameters are used to create a custom emf bean for JPA. When you create a bean of a certain type, it overwrites the default bean, so the system uses the custom emf, and thus the desired database.

In pizza3, we use the mechanism of "profiles" to handle the different params of the three databases. There are profiles named h2, mysql, and oracle, each with a file of properties in src/main/resources. These profiles can be specified on the command line executing the jar file, or in pizza2S, or set up things for the web case. This has to be done really early, so that the right DAO objects are created in the bean-creation phase.

Autowiring in action

The autowiring happens at program startup, in code called from main of ClientApplication, here simplified a bit:

```java
public static void main(String[] args) {
    SpringApplication app = new SpringApplication(ClientApplication.class);
    app.run(args); // calling into SpringApplication code, creating beans, autowiring system not visible... ending ClientRun();
}
```

Our CommandLineRunner: ClientRun

- We see that class ClientRun in pizza2, or CommandLineRunner in pizza3, which is a @Component and thus a bean in the container, implements CommandLineRunner, so it is the one called here, with args from the original main's args.
- The code of the CommandLineRunner has our app-specific code to start up the system. Armed with refs to the service objects, it can create SystemTest or TakeOrder or AdminApp, or set up things for the web case.
- We have skipped over one important initial config here: the database we're using. This has to be done really early, so that the right DAO objects are created in the bean-creation phase.

How can pizza3 run a webapp, or run SystemTest or ...?

The default app run() in ClientApplication 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:

```java
if (!SystemTest.class.equals(appCase)) { // do not start tomcat execution
    app.setWebApplicationType(WebApplicationType.NONE); // don't start Tomcat
}
```

```java
app.run(args); // get tomcat running if appropriate
```
Summary on annotations in pizza2S, pizza3

- Classes in domain have JPA class annotations @Entity, etc. get managed by the em, in pizza2S but not in pizza3, where they are the same as in pizza1 (no annotations)
- In both projects, Classes in *Service in service and *DAO in dao, have Spring annotations, @Service or @Repository, specializations of @Component, making them beans (singletons) in the Spring container.
- In both projects, there is a CommandLineRunner marked as a @Component (and thus a bean) that does the app-specific startup work, such as creating a SystemTest object.
- In both projects, @Autowired is used to mark fields of bean-to-bean refs.

Some classes have no class annotations

- PizzaSystemConfig: just has methods called from elsewhere, never new’d. No longer sets up singletons.
- ServiceException: created as needed, very lightweight
- SystemTest, ShopAdmin, TakeOrder: get created explicitly in ClientRun as needed. They could be made into @Components, but then would all be created when only one is needed.
- PizzaOrderData: created just for data transport, not "important"