

Servlets, Spring Boot

Last time:

Servlets, Chap. 2, embedded tomcat

- Chapter 2 introduces servlets, assuming use of a shared tomcat
- We're using embedded tomcat because it's so much better for code development
- What's the difference?
 - Mainly the URLs in use, a little shorter for embedded tomcat, and where to put the deployment
- Example: ch05email's servlet on a shared tomcat, using webapp named ch05email:
 - URL `http://localhost:8080/ch05email/emailList`
 - Deployment in `webapps/ch05email` under tomcat's document root (`/var/cs636/tomcat-8.5`)
- ch05emailS's servlet using embedded tomcat on port 9000:
 - URL `http://localhost:9000/emailList`
 - Deployment at this tomcat's doc root, in Maven-default `src/main/webapp`

Demo of email05S Project: explicit servlet code running under embedded tomcat

Available at `$cs636/email05S`, i.e. directory `/data/htdocs/cs636/email05S`

Look at README for basic instructions

`pe07$ run.sh` see Spring banner, other burps on way up, let it hang: it's now a server

Nearly last line: `Tomcat started on port(s): 9000 (http) with context path ''` ← you need your own port to avoid colliding with another student or me

In another ssh connection to pe07:

or, in a ssh connection to `users.cs.umb.edu`

`lynx localhost:9000/emailList`

`lynx pe07.cs.umb.edu:9000/emailList`

Need to authorize a cookie, then see the form

Fill it in, do return to submit it

See `Thanks for joining...` output from `thanks.jsp` in lynx output

Also see “`in doPost of emailList Servlet`” on server window: `System.out.println` output from servlet.

Demo 2 of email05S Project: explicit servlet code running under embedded tomcat

After unzipping provided zip at home (say a Windows system)

Look at README for basic instructions

F:\cs\cs636\ch05emailS> run see Spring banner, other burps on way up, including

APPLICATION FAILED TO START

Description:

The Tomcat connector configured to listen on port 9000 failed to start. The port may already be in use or the connector may be misconfigured.

...

Something is using this port...

Sure enough: browse to localhost:9000/emailList, see the form. Look at CMD windows, find an old run, control-C

Demo 3 of email05S Project

Unzip the zip on a Windows system

Look at README for basic instructions

`F:\cs\cs636\ch05emailS> run` see Spring banner, other burps on way up, including

Nearly last line: `Tomcat started on port(s): 9000 (http) with context path ''`

Browse to `localhost:9000/emailList`

See the form, fill it in, submit it

See `Thanks for joining..` output from `thanks.jsp`

Also see “`in doPost of emailList Servlet`” on CMD window: `System.out.println` output from servlet.

Demo 4 of ch05emails Project

- Set it up in eclipse using Open Project from File System...
- See 3 packages: business, like our domain, data, like our dao, email, like service+presentation
- In email package: “extra” file (over Murach’s sources) WebApplication.java

```
@ServletComponentScan(basePackages = "murach.email")
// This annotation is needed for getting embedded tomcat to run
// This app has no @Components (Murach uses static API methods
// in classes that don't need singletons)
@SpringBootApplication()
public class WebApplication {
// run the embedded tomcat
    public static void main(String[] args) throws Exception {
        SpringApplication.run(WebApplication.class, args);
    }
}
```

Demo 4 of ch05emailS Project, cont.

- Run it in Package Explorer by right-clicking WebApplication.java, selecting Run As Java Application
- Browse to localhost:9000/emailList to see form
- Submit it, see thanks response, little messages in eclipse's Console:
 `"in doPost of emailList Servlet", etc.`
- Look at servlet code, see `System.out.println("in doPost of emailList Servlet");`
- So it's that easy (with embedded tomcat) to see debug output from a servlet
 - Shared tomcat: this output shows up in the "tomcat log"
- Bring the program down by Console's red square, then double X's
- Similarly, set breakpoint, Debug as Java app, hit breakpoint, ... just like normal program, but expect "dead" browser window at breakpoint.

More on servlet programming: JSP

JSP's role in a MVC webapp (using Spring or not): generate the HTML returned to the user, based on information passed to it from the controller in variables (request vars so far).

So JSP, in this role, does not interpret user input, make decisions, access the DB, etc. It just generates HTML on direction from the controller.

Example: thanks.jsp of ch05email/ch05emailS: just displays user properties by accessing the user request variable with EL: `${user.email}` etc.

Chapter 5's ch05email has index.jsp for form instead of the index.html in Chapter 2, allowing it to display an error message if necessary, again with EL.

Look at Chapter 5 slides

JSP, EL and JSTL: book coverage

EL is covered in Murach, Ch. 8.

EL is in use in a jsp if you see `#{...}` in the JSP source.

But JSP can do more than just EL. It can loop through collections and do conditional display with JSTL (JSP Standard Tag Library).

JSTL is covered in Murach, Ch. 9.

JSTL is in use in a jsp if you see "`<c:>`" in the JSP source.

The page needs a "taglib directive" for JSTL, pg. 179

Examples of JSTL: `<c:if ...>` pg 179, `<c:forEach ...>` pg. 275 (in Chap. 9)

JSTL is used in pizza3 to display lists of toppings for example.

Note that Chap. 6 is on JSP including some JSTL, but mainly older forms we don't need.

Spring beans

Recall 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/loC 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.

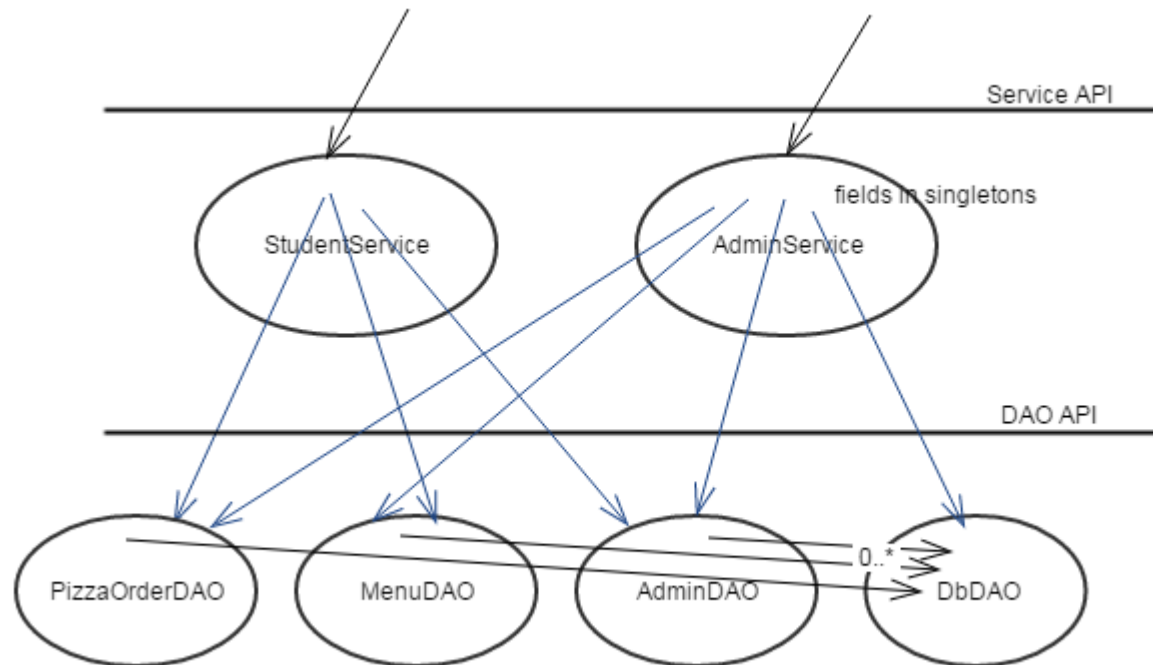
Crucial terminology for Spring

- **bean:** a POJO or Java Bean, in our use (and other similar 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 for cs636, it's FYI
- Pizza3: like pizza1, this just uses JDBC directly in the DAOs
 - This version uses Spring Boot's web support to do the needed web request controllers, and uses its embedded tomcat.
 - This one is the model for Project 2, so needs study.

Same big picture, now Spring beans...



Our setup of beans for pizza2S and pizza3: Note the each is uniquely identified by its class name, i.e., there's one of class StudentService, etc.

← 2 @Service beans

← 4 @Repository beans

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: `CommandRun`, 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?

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 xxxApplication {    // SBAApplication in pizza3,  
                                // ClientApplication in pizza2S  
  
    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`

```
@Service
public class StudentService {
    @Autowired
    private PizzaOrderDAO pizzaOrderDAO;
    @Autowired
    private MenuDAO menuDAO;
    @Autowired
    private AdminDAO adminDAO;
    @Autowired
    private DbDAO dbDAO;
```

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

Autowiring in action

The autowiring happens at program startup, in code called from main of xxxApplication, here simplified a bit (only does web case, not pizza3's SystemTest, etc.):

```
// This specifies where to look for @Components, ets.
@SpringBootApplication(scanBasePackages = { "cs636.pizza" })
public static void main(String[] args) {
    SpringApplication app = new SpringApplication(ClientApplication.class);
    System.out.println("starting ClientRun...");
    app.run(args);    ← calling into SpringApplication code, creating beans, autowiring
    System.out.println("... ending ClientRun");           -- and starting embedded tomcat
}    ← end of main, exit whole program
```

Mystery: how do we regain control here?

We just saw execution disappearing into Spring Boot 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.

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

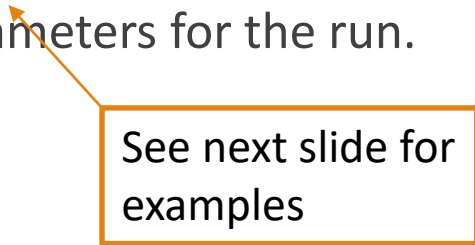
[Info on ... ellipsis notation](#): here for a variable-length array of Strings or “varargs”

Our CommandLineRunner

- We see that class ClientRun in pizza2S, or CommandRun 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 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 pizza2S, the args are parsed 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 overrides 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, and are read early in the startup sequence, specifying the jdbc parameters for the run.



See next slide for examples

Commands to run pizza3 (on pe07)

pe07\$ cd \$cs636 cd to class home directory

pe07\$ cd pizza3

pe07\$ cat runJarByProfile.sh look at shell script to see how it runs the all-dependencies jar

pe07\$ runJarByProfile.sh h2 web use the shell script to run pizza3 as web app (hangs)

(actually this failed in class demo because port 9002 was in use)

In another window:

pe07\$ lynx localhost:9002/welcome.html test the web app by displaying the welcome page

pe07\$ runJarByProfile.sh h2 SystemTest use the shell script to run pizza3's SystemTest (doesn't hang)

- We see that the args coming in to main say what the program should do.
- The first arg says what database to use, the second says “web” or the client/server app to run.

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

The default `app.run()` in `xxxApplication` 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:

```
// code in SBApplication for pizza3
if (!appCase.equals("web")) { // appCase is "SystemTest" or "web" or ...
    System.out.println("have arg " + appCase + " ,
                       assuming client execution");
    app.setBannerMode(Banner.Mode.OFF);
    app.setWebApplicationType(WebApplicationType.NONE); // don't start tomcat
}
app.run(args); // get tomcat running if appropriate, create and wire up beans
```

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"