Servlets, Spring Boot

Chapter 2 introduces servlets, assuming use of a shared 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
  - ch05email’s servlet using embedded tomcat on port 9000:
    - URL: http://localhost:9000/ch05email/emailList
    - Deployment in webapps/ch05email under tomcat’s document root (/var/cs636/tomcat-8.5)

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:

```bash
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 ''`

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 2 of email05S Project:

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

Look at README for basic instructions:

```bash
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.

A configuration file probably needs to be rebuilt, or a configuration file probably needs to be rebuilt, 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:

```bash
run
```

Browse to localhost:9000/emailList

See the form, fill it in, submit it

Also see "An example of a servlet" in server window: System.out.println output from servlet.

Demo 4 of email05S 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

```java
public static void main(String[] args) throws Exception {
    SpringApplication.run(WebApplication.class, args);
}
```

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

```java
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"
- "in doPost of emailList 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 ch05emailS just displays user properties by accessing the user request variable with EL: ${user.email} etc.
Chapter 5's ch05emailS 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.

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:if> in the JSP source.
The page needs a "taglib directive" for JSTL, pg. 179
Examples of JSTL: <c:if>, <c:forEach>...
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.

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 beans, 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.
- 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.
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?

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

```java
@SpringBootApplication( scanBasePackages = { "cs636.pizza" } )
public class xxxApplication { //SBApplication in pizza3, //ClientApplication in pizza2S
```

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?

The references between classes are marked by @Autowired

For example, in StudentService.java

```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.

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.

```java
void run(String… args)
```

Info on … ellipsis notation: here for a variable-length array of Strings or "varargs"
Our CommandLineRunner

- We saw that class ClientRun in pizza2S, or CommandLineRun 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 dao bean for JPA. When you create a bean of a certain type, it overrides the default bean, so the system uses the custom one, 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 jdbi parameters for the run.

Commands to run pizza3 (on pe07)

- pe07$ cd pizza3; cd to class home directory
- pe07$ ls | grep profile.sh look at shell script to see how it runs the all-dependencies.jar
- pe07$ run profile.sh h2 web use the shell script to run pizza3 as web app (hangs)
- pe07$ run profile.sh h2 ---server=emf actually this failed in class demo because port 9002 was in use
- pe07$ run profile.sh h2 ---server=emf -Dserver.port=9002 test the web app by displaying the welcome page

In another window:

- pe07$ lynx localhost:9002/welcome.html test the web app by displaying the welcome page
- pe07$ cd pizza3 cd to class home directory
- pe07$ cat runJarByProfile.sh look at shell script to see how it runs the all-dependencies.jar
- pe07$ cd pizza3
- pe07$ runJarByProfile.sh h2
- pe07$ runJarByProfile.sh web

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

The default app run in xwApplication 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
// code in xApplication
if (!appCase.equals("web")) { // appcase is "SystemTest" or "web" or ...
    out.println("Have any more args? ");
    args = args + " ";
    app.setBannerMode(Banner.Mode.OFF);
    app.setWebApplicationType(WebApplicationType.NONE); // don't start tomcat
    app.run(args); // get to main app
} else { // web app execution
    // code in Application
    app.run(args); // get to main app
}
```

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 @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"