Spring Core and Spring Boot

DGP Solutions

Contents

Spring Core Primer for Spring Boot

DGP Solutions

Spring

- A lightweight framework that addresses each tier in a Web application
  - Presentation layer – An MVC framework that is most similar to Struts but is more powerful and easier to use
  - Business layer – Lightweight IoC container and AOP support
  - Persistence layer – DAO template support for popular ORMs and JDBC
- Uses the **Inversion of Control** (IoC) pattern is that programmers don't need to create your objects but instead, they need to describe how they should be created e.g. factories, **configuration files. The** Spring IoC container is responsible for using the configuration to assemble object graphs
- XML files were used to provide the configuration, however, the trend has been towards annotation and java configuration techniques

19920612 Copyright DGP Solutions S-4

Dependency injection is a pattern used to create instances of objects that other objects rely on without knowing at compile time which implementation class will be used to provide that functionality. With Spring, the implementation of dependency injection can be achieved via setter and/or constructor injection techniques

Key Interfaces and Implementations

- Spring provides the org.springframework.beans.factory.BeanFactory interface to create and manage our beans
- The BeanFactory implementation is the actual *container* which instantiates, configures, and manages a number of beans
- The ApplicationContext builds on top of the BeanFactory. Common Implementations: include ClassPathXmlApplicationContext, FileSystemApplicationContext, XmlWebApplicationContext

Xml Dependency Injection Configuration

- Below an XML configuration shows BeanDefinitions which, upon the loading of the ApplicationContext, will be parsed into a BeanDefinition objects. These BeanDefinitions are actually used by the IoC container to create Managed Bean instances

```
<bean id="myService" name="service" class="com. service.MyServiceImpl"  >
      <property name="dao" ref="myDao"/>
 </bean>

<bean id="myDao" class="com.service.MyDaoImpl"/>
```

Dependency Injection

```
<bean id="myService" name="service" class="com. service.MyServiceImpl"  >
      <property name="dao" ref="myDao"/>
  </bean>

<bean id="myDao" class="com.service.MyDaoImpl"/>
```

```
public class MyDaoImpl implements
MyDao  {

}
```

```
public class MyServiceImpl implements
MyService {

    private MyDao dao;
    public void setDao(){
        this.dao=dao;
    }
}
```

Interacting with Spring

- So, how does Spring do this? Well, our Spring client first of all instantiates an ApplicationContext instance to represent the IoC Container. **It needs the XML configuration file.**
- Once that is done, BeanDefinitions are created from the configuration. Then, Spring Managed Bean instances are created EAGERLY from the bean definitions

```
private ApplicationContext context;
private MyService service;

context = new ClassPathXmlApplicationContext("beans.xml");
service =  context.getBean("myService", MyService.class);
```

19920612 Copyright DGP Solutions S-8

Spring has been designed with the Singleton pattern in mind

However, it is not synonymous with the traditional Java singleton, *i.e.*, one instance of a class per JVM

It is based off a single instance per "id" attribute of the BeanDefinition

If we repeatedly ask the ApplicationContext for the bean via the same "id," we will get the same instance

We can change this to lazily instantiate a different bean upon every request by using the scope="prototype" property

Multi Xml files

- Although we could have one XML file import other files…

```
<import resource="moreBeans.xml"/>
 <bean id="myService" name="myServ"
          class="com.service.MyServiceImpl">
        <property name="dao" ref="dao"/>
  </bean>
```

- …this tightly couples our XML file to another file for good. However, we could keep them separate and use a ClassPathXmlApplicationContext constructor

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans1.xml","beans2.xml");
```


File beans1.xml

```
<bean id="myService" name="myServ"
   class="com.service.MyServiceImpl"  >
        <property name="dao" ref="dao"/>
  </bean>
```

File beans2.xml

```
<bean id="dao" class="com.service.MyDaoImpl"/>
```

Order

- The IoC container creates instances of beans and then controls subsequent dependency injection on those beans . We can trace the order in which this "inflation" of beans is carried out by registering an implementation of the core Interface BeanPostProcessor. This requires implementing the following two methods. These methods will be kicked off by the container before and after all initializers

```
public class MyPostProcessor implements BeanPostProcessor{
    public Object postProcessAfterInitialization(Object arg0, String arg1)
    throws BeansException {
        return arg0;
    }
    public Object postProcessBeforeInitialization(Object arg0, String arg1)
    throws BeansException {
        return arg0;
    }
}
```

Scan for Annotations

- We register BeanPostProcessors for specific processors that scan for particular Annotations through reflection and can take action. This would be quite tedious to individually register all of Spring's processors, as in the notes. However, we can register all of them at once by leveraging XML schema configurations using the context namespace

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd">
   <context:annotation-config/>
</beans>
```

@Component

- This will turn on a means to achieve dependency injection, but we also want to get the bean definitions OUT of the XML. Spring 3.5 added support for **auto-detecting** beans on the classpath (rather than using XML config files)

- The **@Component** indicates that a class fulfills the role or **stereotype** of a component (a Spring Managed Bean)
  - It is in the package org.springframework.stereotype

```
@Component("xyz")
public class MyServiceImpl implements MyService  {
```

- It is the equivalent of:

```
<bean id="xyz" class="com.service.MyServiceImpl" >
```

Component Scanning

- To auto detect these classes and register the corresponding beans, you need to include the following element in XML, where the base-package element is a common parent package identifier for candidate classes

```
<context:component-scan base-package="com.service" />
```

- We have registered the other annotation BeanPostProcessors as well `<context:annotation-config/>`
- Furthermore, the AutowiredAnnotationBeanPostProcessor and CommonAnnotationBeanPostProcessor are both included implicitly when you use the component-scan element

Filters

- By default, classes annotated with @Component, @Repository, @Service, or @Controller are the only detected candidate components
- However, you can modify and extend this behavior simply by applying custom filters. This approach means that you do not have to use the @Component annotation at ALL if your class complies with the filter expressions

```
<context:component-scan base-package="com.service" >
    <context:include-filter type="regex" expression="com.service.*"/>
    <context:exclude-filter type="annotation“
       expression="com.service.HealthAnnotation"/>
    <context:exclude-filter type="assignable" expression="com.service.Health"/>
</context:component-scan>
```

@Autowired

- The org.springframework.beans.factory.annotation.Autowired annotation provides a type-driven injection strategy
- It can be applied to:
  - Constructors, fields, and setter methods for normal injection
  - A method with an arbitrary name and arguments in which all arguments are autowired with a matching bean
  - Collections or an Array where it injects all beans of a type

```
@Component
public class MyServiceImpl implements MyService  {
@Autowired
public void setDao(MyDao dao) {
  this.dao = dao;
}
```

```
@Component
public class MyDaoImpl implements MyDao {
```


Autowired can be applied to constructors and even Lists

Autowiring by Type normally fails if there are multiple candidates to wire
However, for a Collection (or Array) it will instantiate and populate the Collection with ALL candidates for you
You can annotate the field itself instead of the setter, in which case, for Dependency Injection purposes, you can remove the setter for that property

@Value and @Scope

- The org.springframework.beans.factory.annotation.Value annotation is used to inject simple values that, just like in XML configurations, trigger PropertyEditors

```
@Component @Scope("prototype")
public class MyService implements MyService  {
    @Value("/bbc.com")
    private String url;
    @value("8080")
    private int port;
```


Using @Scope, we can define our bean to be a prototype

The **@Required** annotation can be applied to a setter method to indicate the property must be populated at configuration time, either by annotations or xml configuration. Otherwise, the container will throw an exception

```
@Required
public MyDao getDao() {
    return dao;
}
```


@Qualifier

- Because autowiring by type may lead to multiple candidates where only one is required or even allowed, the multiple candidates are secured and only one is required so the IoC container will throw an exception. Therefore, it is often necessary to implement more control over the selection process
- Spring's **org.springframework.beans.factory.annotation.Qualifier annotation** gives you some control over the injection
- You can associate the @Qualifier attribute "value" with specific arguments in order to narrow the set of potential type matches so that a specific bean is chosen

19920612 Copyright DGP Solutions S-17

In the following situation, is problematic;

```
@Component
public class MyDaoImpl implements MyDao  {

}


@Component
public class MyDaoNull implements MyDao  {

}

@Component(value="mylService")
public class MyServiceImpl implements MyService{
@Autowired
Private MyDao dao;
```

LabS.1 – XML

LabS.1 - XML Configuration

- Objective: Review an XML project to understand XML configured dependency Injection
- Open up the project in the directory S1_Student in your setup directory. In the file beans.xml, complete the configuration to “wire up” the classes below that adhere to the illustrated design;

- A test class has been provided for you, ensure that everything runs once you have completed the XML //TODO statements

@Qualifier Options

- Option 1: Name your property with the exact same name as the id of the DAO that you want

```
@Component(value="myService")
public class MyServiceImpl implements MyService{
@Autowired private MyDao myDaoImpl;
```

- Option 2: Introduce a @Qualifier annotation to link to the dependency bean's "id"

```
@Component(value="myService")
public class MyServiceImpl implements MyService{
@Autowired @Qualifier(myDaoImpl") private MyDao dao;
```

- Option 3: Introduce a @Qualifier annotation to link to the dependency bean's "qualifier"

```
@Component(value="myService")
public class MyServiceImpl implements MyService{
@Autowired @Qualifier(dallas") private MyDao dao;
```

19920612 Copyright DGP Solutions S-20

```
@Component @Qualifier("dallas")
public class MyDaoImpl implements MyDao  {

}
```

JSR250

- @Resource can also be used to resolve well known resolvable dependencies by id

```
@Resource(name="myDaoImpl")
private MyDao dao;
```

- The CommonAnnotationBeanPostProcessor not only recognizes the @Resource annotation but also the JSR-250 javax.annotation life cycle annotations

```
@PostConstruct
private void init() {System.out.println("Postconstruct");}
@PreDestroy
private void cleanUp() {System.out.println("PreDestroy");}
```

JSR330

- Using the javax @Inject annotation core Java's equaivalent of the Spring @Autowired annotation
- The @Inject annotation can be used to qualify a class variable or any method (including setter method) which takes the injected type as an argument

```
@Named(value="myService")
@Singleton
public class EmployeeServiceImpl implements MyService{
private MyDao dao;
@Inject
public MyServiceImpl(@Named("myDaoImpl")MyDao dao) {
       this.dao = dao;
}
```


If multiple bean types are available for injection, then Spring will be unable to make a decision on which bean to inject and will throw an Exception. In such cases, we can use the @Named(value="..") annotation and give the name of the bean that we want Spring to inject. This is equivalent to Qualifiers.

JSR330 does not have the equivalent of @Value for simple injection
However, JSR250 has the lifecycle methods. So, to complete our Service/Dao configuration, we can use a @PostConstruct annotated method

JUnit

- The Spring Framework provides the following set of Spring-specific annotations that you can use in your unit and integration tests in conjunction with the TestContext framework
- **@ContextConfiguration** defines class-level metadata that is used to determine how to load and configure an ApplicationContext for tests
- Specifically, @ContextConfiguration declares either the application context resource locations or the annotated classes that will be used to load the context

@ContextConfiguration

- JUnits (JUnit4.*) that use Spring have the ApplicationContext injected into them by a specific JUnit runner
  - Our @ContextConfiguration annotation injects our ApplicationContext
  - This approach allows you to directly inject, via @Inject, @Resource, or @Autowired, any dependencies into the JUnit class for testing

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:beans.xml"})
public class MyServiceImplTest {
       @Inject
        private MyService service;
        @Test
        public void testService() {

        }
```

LabS.2 – Annotations

LabS.2 – Annotation Configuration

- Objective: Use Annotations to wire up the prior lab
- Take a copy of the prior solution. In that copy remove the XML bean definitions from the beans.xml file and replace it with a component scan and component filters (See notes)
- Use Spring annotations to sire up your classes @Autowired is fine
- The JUnit should not change


```
<context:component-scan base-package="com.rail">
    <context:include-filter type="regex" expression="com.rail.*"/>
    <context:exclude-filter type="regex" expression="com.rail.core.*"/>
    <context:exclude-filter type="assignable" expression="com.rail.Application"/>
    <context:exclude-filter type="assignable"
expression="com.rail.service.BasicServiceTest"/>
  </context:component-scan>
```

@Configuration Classes

- As of Spring 4.*, there is a way to configure Spring beans in Java while still decoupling the configuration from the managed beans themselves
- The central artifact for this Java-configuration support is the @Configuration-annotated classes that take the place of XML configuration files and finally omit the need for an XML file at all
- Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions
- **@Bean** annotated methods that return a bean define the BeanDefinition itself

Java Configuration – Dependency Injection

- BeanDefinition declarations in a configurable class are shown below:

```
@Configuration
public class AppConfig {
    @Bean
    MyService getService(){
        MyServiceImpl service = new MyServiceImpl();
        service.setDao(getDao());
        return service;
     }
    @Bean(name="dao")
    MyDao getDao(){
       return new MyDaoImpl();
   }}//end of class brace here for slide spacing
```



```
<bean id="service" class="com.service.MServiceImpl">
     <property name="dao" ref="dao"/>
</bean>
<bean id="dao" class="com.serviceMy"/>
```

AnnotationConfigApplicationContext

- AnnotationConfigApplicationContext is used to recognize @Configuration classes
- In much the same way that Spring XML files are used as input when instantiating a ClassPathXmlApplicationContext, @Configuration classes may be used as input when instantiating an AnnotationConfigApplicationContext. Indeed so can @Component classes

```
ApplicationContext ctx = new
AnnotationConfigApplicationContext(AppConfig.class);
MyService service = ctx.getBean(MyService.class);
```

Using multiple Configuration Classes

- A @Configuration class may have dependencies on beans from another @Configuration class that comprise part of the context. You can inject them, like any defined bean, as shown below if they provide part of the ApplicationContext.

```
@Configuration
public class AppConfig {
    @Inject private MyDao dao;
    @Bean
    MyService getService(){
        MyServiceImpl service = new MyServiceImpl();
        service.setDao(dao);
        return service;
     }
```

```
@Configuration
Class AnotherConfig {
    @Bean(name="dao")
     MyDao getDao(){
          return new MyDaoImpl();
    }
}
```

JUnit

- All we need to do is amend the @ContextConfigLocations attribute to use config classes and not XML

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={AppConfig.class, AnotherConfig.class})
public class MyServiceImplTest {
       @Inject
        private MyService service;
        @Test
        public void testService() {

        }
```


If you do not want a lot of configuration classes exposed to your JUnit you can import one into another

```
@Configuration
@Import(AnotherConfig.class)
public class AppConfig {
    @Inject private MyDao dao;
    @Bean
    MyService getService(){
          MyServiceImpl service = new MyServiceImpl();
          service.setDao(dao);
          return service;
     }
```

Externalize Configuration

- Sometimes we may want to externalize some configurations, particularly those @Value annotations. We can do so via a properties file referenced by a Spring provided **PropertyPlaceholderConfigurer**. Using Expression Language (EL) expressions, the IoC container can eagerly resolve their values to the properties file. Conversions are implicit. This happens when the IoC container is inflated

```
@Bean
public PropertyPlaceholderConfigurer getProperties(){
    PropertyPlaceholderConfigurer ppc = new PropertyPlaceholderConfigurer();
    Resource[] resources = {new ClassPathResource("my.properties")};
    ppc.setLocations(resources);
    return ppc;
}
```

```
public class MyServiceImpl implements MyService{
 @Value("${active}")  private String studentName;
```

studentName=Fred

@PropertySource

- A specialization class that can resolve placeholders expressions ${} within bean definition property values in XML an/or @Value annotations against the current Spring Environment and its set of **PropertySources**

```
@Bean
public static PropertySourcesPlaceholderConfigurer placeHolderConfigurer() {
   return new PropertySourcesPlaceholderConfigurer();
}
```

- Note the "static" as this must be resolved **before any other beans as it is amending metadata**. Our sources can be set via the class annotation **@PropertySource**

```
@Configuration
@PropertySource(value="classpath:my.properties")
public class AppConfig {
```

19920612 Copyright DGP Solutions S-33

Frequently, we want to inject values from property files into our configuration classes in order to keep out domain classes annotation free. Below we inject values into a bean definition method.

```
@Bean
Service getService(@Value("${studentName}")
String x){

}
```

OR

```
@Value("${studentName}") private String student;
@Bean
Service getService(){
    System.out.println(student);
}
```

LabS.3 – Java Configuration

LabS.3 – Java Configuration

- Objective: Use Java Configuration Classes to wire up the prior lab
- Take a copy of the prior solution. In that copy remove your beans.xml file. Also remove ALL Spring annotations from all classes in your project
- Generate setters in the class BasicDaoImpl for the property stationList and dao in BasicServiceImpl

LabS.3 – Configuration Class

- Create a class com.rail.RailConfig and create two bean definitions for;
  - BasicDaoImpl – Simply use your setter for StationList with a new instance of it i.e. dao.setStations(new StationList());
  - BasicServiceImpl using your BasicDaoImpl bean definition in the appropriate setter
- Change your JUnit to use the class RailConfig and not beans.xml. It should run as before.

LabS.3 – PropertySources [Optional]

- In the class RailConfig, add a bean definition for a PropertySourcesPlaceholderConfigurer, make sure you put the @PropertySource("classpath:application.properties") above your class declaration
- In application.properties add an entry of
  - Service.name=“Railway Stations”
- In the class BasicServiceImpl add a member variable and annotate it with EL
  - @Value("${serv.name}")
  - private String serviceName;
  - Add a getter for this variable in the class
  - Write a JUnit for this new feature in the existing test class