Java Spring & AWS
Sep 7, 2021 - Oct 8, 2021
Monday to Friday
9:30 AM ET - 4:30 PM ET

Agenda: Day -11

- The Spring Container
  - Overview of the Spring Container
  - Application Context Overview
  - ClassPathXmlApplicationContext, FileSystemXmlApplicationContext, AnnotationConfig ApplicationContext
  - API and Usage
- Dependencies and Dependency Injection (DI)
  - Examining Dependencies
  - Dependency Inversion
  - Configuration and Usage of Dependency Injection (DI) in Spring
- Aspect Oriented Programming
- Database access using spring
  - Spring Data JPA

Spring bean java based configuration using @Configuration and @Bean

- Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions. The @Bean annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

Difference between constructor and setter injection

- Partial dependency: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
- Overriding: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
- Changes: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

Dependency Injection with Factory Method in Spring

Spring framework provides facility to inject bean using factory method. To do so, we can use two attributes of bean element.

- **factory-method:** represents the factory method that will be invoked to inject the bean.
- **factory-bean:** represents the reference of the bean by which factory method will be invoked. It is used if factory method is non-static.

Factory method

- A method of a java class which is capable of constructing and returns its own class object.
- When spring bean class has private constructors we can not create its objects from outside . Then we can give instruction to spring container to create object of the bean class by using factory method.
- Factory method can be static or non static.
- Instruction to spring container to call factory method
- We should use factory-method attribute of <bean> tag to configure the factory method to be called to create the object

- A factory class contains factory method to instantiate class object that factory method can be static or instance method
- Incase of Static factory static method will return other class object

Ex:

- 1.Class c = Class.forName("ClassName");
- 2. Thread t = Thread.currentThread();
- 3. Calendar cl = Calendar.getInstance();

non-static factory

- A **non-static factory** method that returns instance of **another** class. It is used instance is not known and decided at runtime.

Difference between BeanFactory and the ApplicationContext

- The org.springframework.beans.factory.**BeanFactory** and the org.springframework.context.**ApplicationContext** interfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

Using BeanFactory

- The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

Resource resource=new ClassPathResource("applicationContext.xml");
BeanFactory factory=new XmlBeanFactory(resource);

- The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

Using ApplicationContext

- The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

ApplicationContext context =
new ClassPathXmlApplicationContext("applicationContext.xml");

- The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext

Drawbacks of Core Container or BeanFactory Container

ApplicationContext Container is advanced than Beanfactory Container...

1) BeanFactory Container is basic container, it can only create objects and inject Dependencies.But we can not attach other services like security, transaction ,messaging etc. To provided all the services we have to use ApplicationContext Container.
2) BeanFactory Container doesn't support the feature of AutoScanning , but ApplicationContext Container supports.
3) Beanfactory Container will not create a bean object upto the request time.It means Beanfactory Container loads beans laziely. While ApplicationContext Container creates objects of Singleton bean at the time of loading only.It means there is early loading.
4) Beanfactory Container support only two scopes (singleton & prototype) of the beans. But ApplicationContext Container supports all the beans scope.,like session, application

Singleton (vs) Prototype

- A singleton is a class that allows only a single instance of itself to be created and gives access to that created instance. It contains static variables that can accommodate unique and private instances of itself. It is used in scenarios when a user wants to restrict instantiation of a class to only one object. This is helpful usually when a single object is required to coordinate actions across a system.

Prototype:

- Prototype design pattern is used in scenarios where application needs to create number of instances of a class, which has almost same state or differs very little.

prototype

- For prototype: Spring.xml
- <beans>
- <bean id=”car” class=”Car”   scope=”prototype” />
- </beans>

- Class Test
- {
- p s v m(Str…arg)
- {
- ApplicationContext ap=new ClassPathXmlApplicationContext(“spring.xml”);
- Car car1=(Car)ap.getBean(“car”);
- Car car2=(Car)ap.getBean(“car”);
- System.out.println(car1==car2);//false
-
- }
- }

Auto wiring

The process of configuring bean properties and performing dependency injection on bean is called wiring operation.

These are two types of wiring operations

- Explicit wiring
- Auto wiring

If container is detecting dependent values to bean properties automatically without any configuration of properties, then its called auto wiring.

Auto wiring

- Auto wiring is A concept doing dependency injection by using auto searching and possible parameters injecting.
- It internally uses setter or constructor injection.
- We can do this auto wiring only for secondary types[It internally uses setter or constructor injection.]
- Adv: It requires the **less code** because we don't need to write the code to inject the dependency explicitly.
- Dis adv: No control of programmer and it can't be used for primitive and string values.

Auto wiring

Autowiring Mode	Description
No	No autowiring is performed. All references to other beans must be explicitly injected. This is the default mode.
byName	Based on the name of a property, a matching bean name in the IoC container will be injected into this property if it exists.
byType	Based on the type of class on a setter, if only one instance of the class exists in the IoC container it will be injected into this property. If there is more than one instance of the class a fatal exception is thrown.

Auto wiring

- autowire=”no” : in case of no ,user manually need to do the dependency injection

byName autowiring mode (xml file)

- In case of byName autowiring mode, bean id and reference name must be same.
- It internally uses setter injection.

byType autowiring mode

- In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type.
- It internally uses setter injection.

Reading data from multiple properties files

- We can read data from multiple files, if two files contains same property name, so the first property file value will be replaced to second property file value
- We may have any number of configuration files for the purpose of easy organization ,still programmatically we can consider them as a single file

I18n by application context container (Locale)

- Making our application projects working for multiple locales with flexibility in presentation logic to pass display labels of input fields or any information from outside the application is called applying I18n on the project.
- The project enables with I18n can give service to multiple client organization which may belong to multiple communities without changing the source code.

Locale = language + country

Examples of language code and country code

- En-us [English – united states]
- En –in [English – India]
- En au
- En ca
- En ie English Ireland
- En-gb English united kingdom

I18n in spring

- We write a separate properties file for each language and we furnish all the labels in it for display of labels based on internet options of browser.
- **Org.springframework.context.support.ResourceBundle MessageSource** class makes the underlying application context container recognizing multiple locale properties files on I18n that are give to the application

Locale class from java.util package (Continued)

- In java, we can use Locale class and ResouceBundle class of java.util package to apply the effect of I18n.
- While preparing multiple locale specific properties files the base file must be there because when no matching file is available then values will be taken from the base file.

getMessage() method

In order to make the container to get message from properties file we need to call getMessage() method on the container object.
For getMessage() method we have to pass 4 arguments they are

- Key in properties file
- Java.lang.object class array to pass arguments values to place holders available in the property values . If the place holders are not there then we can pass null as a value.
- Defualt message: when message from given key is not collectable default message will be displayed
- Java.util.Locale class object with language and country code

Log4j Introduction

- While developing Java/J2EE applications, for debugging an application that is to know the status of a java application at its execution time, in general we use system.out.println statements in the application right…
- But we have some disadvantages while using SOPL (system.out.println) statements.
- Generally SOPL statements are printed on console, so there are temporary messages and when ever the console is closed then automatically the messages are removed from the console.
- It is not possible to store the SOPL messages in a permanent place and these are single threaded model, means these will prints only one by one message on the console screen.

- In order to overcome the problems of SOPL statements Log4j came into picture, with Log4j we can store the flow details of our Java/J2EE in a file or databases.
- This is a Open Source tool given by Apache, for only java projects, to record or write the status of an application at various places.
- Working with log4j is nothing but working with classes & interfaces given in org.apache.log4j.*
- Log4j is a common tool, used for small to large scale Java/J2EE projects
- In Log4j we use log statements rather SOPL statements in the code to know the status of a project while it is executing.
- In real time, after a project is released and it is installed in a client location then we call the location as on-site right, when executing the program at on-site location, if we got any problems occurred then these problems must report to the off showered engineers, in this time we used to mail these Log files only so that they can check the problems easily.

Log4j

log4j is a reliable, fast and flexible logging framework (APIs) written in Java, which is distributed under the Apache Software License. log4j is a popular logging package written in Java.

log4j has three main components:

- **loggers**: Responsible for capturing logging information.
- **appenders**: Responsible for publishing logging information to various preferred destinations.
- **layouts**: Responsible for formatting logging information in different styles.

Types of objects

There are two types of objects available with log4j framework.

- **Core Objects:** These are mandatory objects of the framework. They are required to use the framework.
- **Support Objects:** These are optional objects of the framework. They support core objects to perform additional tasks.

Core objects include the following types of objects –

- **Logger Object**

The top-level layer is the Logger which provides the Logger object. The Logger object is responsible for capturing logging information and they are stored in a namespace hierarchy.

- **Layout Object**

The layout layer provides objects which are used to format logging information in different styles. It provides support to appender objects before publishing logging information.

Layout objects play an important role in publishing logging information in a way that is human-readable and reusable.

- **Appender Object**

This is a lower-level layer which provides Appender objects. The Appender object is responsible for publishing logging information to various preferred destinations such as a database, file, console, UNIX Syslog, etc.

log4j.properties

- The **log4j.properties** file is a log4j configuration file which keeps properties in key-value pairs. By default, the LogManager looks for a file named **log4j.properties** .

Configuring log4j involves assigning the Level, defining Appender, and specifying Layout objects in a configuration file.

log4j.properties Syntax:

```
# Define the root logger with appender X
log4j.rootLogger = DEBUG, X

# Set the appender named X to be a File appender
log4j.appender.X=org.apache.log4j.FileAppender

# Define the layout for X appender
log4j.appender.X.layout=org.apache.log4j.PatternLayout
log4j.appender.X.layout.conversionPattern=%m%n
```

Spring with log4j – client app (Spring)

```
public class TestClient {
   static Logger log = Logger.getLogger(TestClient.class.getName());

   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
      log.info("Going to create HelloWord Obj");
      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();

      log.info("Exiting the program");
   }
```


log4j.properties

```
# Define the root logger with appender file
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
# Set the name of the file
log4j.appender.FILE.File=C:\\log.out

# Set the immediate flush to true (default)
log4j.appender.FILE.ImmediateFlush=true
```

- `# Set the threshold to debug mode #Threshold is second filter for messages to be logged`

```
log4j.appender.FILE.Threshold=debug

# Set the append to false, overwrite
log4j.appender.FILE.Append=false

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```


Jakarta Commons Logging (JCL) API

- Alternatively you can use **Jakarta Commons Logging (JCL)** API to generate a log in your Spring application. JCL can be downloaded from the

https://jakarta.apache.org/commons/logging/ (https://jakarta.apache.org/commons/logging/)

- *Debugging* allows you to run a program interactively while watching the source code and the variables during the execution.
- A *breakpoint* in the source code specifies where the execution of the program should stop during debugging. Once the program is stopped you can investigate variables, change their content, etc.
- To stop the execution, if a field is read or modified, you can specify *watchpoints*.

Debugging support in Eclipse

- Eclipse allows you to start a Java program in *Debug mode*.
- Eclipse provides a *Debug perspective* which gives you a pre-configured set of *views*. Eclipse allows you to control the execution flow via debug commands.

- To define a breakpoint in your source code, right-click in the left margin in the Java editor and select *Toggle Breakpoint*. Alternatively you can double-click on this position.

Any queries?