Adavanced Java

Agenda

- Sping Boot - Hello World
- Dependency Injection
- Spring Bean Life Cycle
- Stereo-type annotations
- Spring Expression Language
- Auto-wiring

Spring Boot "Hello World" Explanation

Maven pom.xml (Example for Spring Boot 2.x and Java 11)

- parent POM: spring-boot-starter-parent
  - `<java.version>1.8</java.version>`
  - resources
    - `<include>**/application*.properties</include>`
    - `<include>**/application*.yaml</include>`
  - parent POM: spring-boot-dependencies
    - Manage all dependencies versions
      - `<jakarta-servlet.version>4.0.4</jakarta-servlet.version>`
      - `<thymeleaf.version>3.0.15.RELEASE</thymeleaf.version>`
      - `<mysql.version>8.0.29</mysql.version>`
      - `<hibernate.version>5.6.9.Final</hibernate.version>`
      - `<tomcat.version>9.0.63</tomcat.version>`
      - `<spring-framework.version>5.3.20</spring-framework.version>`
- dependencies:
  - spring-boot-starter

- spring-boot
- spring-boot-autoconfigure
- spring-core
- jakarta.annotation-api
- snakeyaml

Application bootstrapping

```
@SpringBootApplication
public class HelloSpringBootApplication {
    public static void main(String[] args) {
        SpringApplication.run(HelloSpringBootApplication.class, args);
    }
}
```

@SpringBootApplication annotation

- @SpringBootApplication = @ComponentScan + @Configuration + @EnableAutoConfiguration
- @ComponentScan
  - Auto detection of spring stereo-type annotated beans e.g. @Component, @Service, @Repository, @Configuration, @Controller, @RestController, ...
  - By default basePackage is to search into current package (and its sub-packages).
  - @ComponentScan("other.package") can be added explicitly to search of beans/config into given package.
- @Configuration
  - Spring annotation configuration to create beans.
  - Contains @Bean methods which create and return beans.
  - @Configuration classes are also auto-detected by @ComponentScan
- @EnableAutoConfiguration annotation
  - Intelligent and automatic configuration
  - Auto-configuration class are internally Spring @Configuration classes.
  - @Conditional beans

    - @ConditionalOnClass
    - @ConditionalOnMissingBean

SpringApplication class

- Entrypoint of Spring boot application
    - main() method is called by JVM.
- SpringApplication.run() does following
    - Create an ApplicationContext instance
    - Prepare Spring container for spring bean life-cycle management
    - Load all singleton beans
    - Execute CommandLineRunner if available -- Console application

Dependency Injection

- Spring container inject dependency object after bean object creation.
- There are two major varients of dependency injection.
    - Constructor based DI
    - Setter based DI

Spring Bean Dependency Injection (sb02_depinjection)

- step 1: New Spring Starter Project
    - Fill group id, artifact id, Spring Boot=3.x.y, Language=Java, Java version=17, packaging=Jar.
    - Modify pom.xml and update Maven project.
        - `<version>2.7.18</version>`
        - `<java.version>11</java.version>`
- step 2: Create java interface Person and implement into PersonImpl class (in same package).
    - fields: name, age
    - methods: Paramless constructor, Parameterized constructor, getters & setters
- step 3: Create java interface Account and implement into AccountImpl class (in same package).
    - fields: id, type, balance, accHolder

    - methods: Paramless constructor, Parameterized constructor, getters & setters, deposit(), and withdraw()
- step 4: Create beans.xml under src/resources to configure beans p1, a1, a2.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
    <!-- beans.xml -->
    <bean id="p1" class="com.sunbeam.dmc.PersonImpl">
      <property name="name" value="Nilesh"/>
      <property name="age" value="40"/>
    </bean>
    <bean id="a1" class="com.sunbeam.dmc.AccountImpl">
      <property name="id" value="101"/>
      <property name="type" value="Saving"/>
      <property name="balance" value="5000.0"/>
      <property name="accHolder" ref="p1"/>
    </bean>
    <bean id="a2" class="com.sunbeam.dmc.AccountImpl">
      <constructor-arg index="0" value="202"/>
      <constructor-arg index="1" value="Current"/>
      <constructor-arg index="2" value="200000.0"/>
      <constructor-arg index="3" ref="p1"/>
    </bean>
</beans>
```

- step 5: Import XML resource with @ImportResource on your main class (with @SpringBootApplication).

```
@ImportResource(locations = {"classpath:beans.xml"})
```

- step 6: Autowire ApplicationContext in the main class.

```
@Autowired
private ApplicationContext ctx;
```

- step 7: Inherit main class from CommandLineRunner interface and implement run() method to access beans.

```
Person p1 = (Person)ctx.getBean("p1");
Account a1 = (Account)ctx.getBean("a1");
Account a2 = (Account)ctx.getBean("a2");
// ...
```

- step 8: Run application and check if beans are created.
- step 9: Create a new Configuration class BankConfig in "SAME" package to create beans using JavaConfig.

```
@Configuration
public class BankConfig {
    @Bean
    public Person p2() {
        Person p = new PersonImpl();
        p.setName("Nitin");
        p.setAge(45);
        return p;
    }
    @Bean
    public Accont a3() {
        Account a = new AccontImpl();
        a.setId(101);
        a.setType("Saving");
        a.setBalance(5000.0);
        a.setAccHolder(p2());
        return a;
    }
```

```
    @Bean
    public Account a4() {
        Account a;
        a = new AccountImpl(202, "Current", 200000, p2());
        return a;
    }
}
```

- step 10: Access beans in main class run() method.

```
Person p2 = (Person)ctx.getBean("p2");
Account a3 = (Account)ctx.getBean("a3");
Account a4 = (Account)ctx.getBean("a4");
// ...
```

- step 11: Create a5 bean in BankConfig and inject accHolder via @Bean method param.

```
@Bean
public Account a5(Person p) {
    Account a = new AccountImpl(505, "Current", 500000, p);
    return a;
}
```

- step 12: Run application and observe error. Error: required a single Person bean, but 2 were found i.e. p1, p2.
- step 13: Add @Qualifier for a5() argument.

```
@Bean
public Account a5(@Qualifier("p1") Person p) {
    Account a = new AccountImpl(505, "Current", 500000, p);
```

```
        return a;
    }
```

- step 14: In main class run(), access the beans and print them. Run the application and test if it produce desired output.
- step 15: Move BankConfig into a new package (not the sub-package) like com.sunbeam.dac. Run the application and observe the error.
- step 16: Use @ComponentScan("com.sunbeam.dac") on main class to resolve this error.

Spring Bean Life Cycle

- Life cycle of an java object is governed by the container that creates the object.
- Container do call certain methods the object to initialize it or give some information.
- Java applets are executed by JRE plugin in the browser (referred as applet container). It used to call methods of applet.
    - init(), start(), paint(), stop(), destroy().
- Java servlets are executed by web container in the Java web server. It calls the certain methods of the servlet object.
    - init(), service(), destroy()
- The life cycle methods are callback methods i.e. they are implemented by the programmer and are invoked by the container.
- init() method is called by the container and all container services are accessible from there. This is not possible while construction of the object.

Bean life cycle demonstration (sb03_lifecycle)

- step 1: Spring Starter Project
    - Fill group id, artifact id, Spring Boot=3.x.y, Language=Java, Java version=17, packaging=Jar -- click Finish
    - Modify pom.xml and update Maven project.
        - `<version>2.7.18</version>`
        - `<java.version>11</java.version>`
- step 2: Create java interface Box and implement into BoxImpl class (in same package).
    - fields: length, breadth, height.
    - methods: Paramless constructor, Parameterized constructor, getters & setters, calcVolume()
    - Add System.out.println() in all methods.
- step 3: Create annotation config class - AppConfig (in same package).
    - Annotate with @Configuration
    - Create @Bean "b1" with setter based DI.

    - Create @Bean "b2" with constructor based DI.
- step 4. Inherit application main class from CommandLineRunner and autowire ApplicationContext in it. Use it to access beans in CommandLineRunner.run() method.

```
@SpringBootApplication
public class Main implements CommandLineRunner {
    public static void main(String[] args) {
        SpringApplication.run(Main.class, args);
    }

    @Autowired
    private ApplicationContext ctx;

    @Override
    public void run(String... args) throws Exception {
        BoxImpl b1 = (BoxImpl) ctx.getBean("b1");
        System.out.println(b1);

        BoxImpl b2 = (BoxImpl) ctx.getBean("b2");
        System.out.println(b2);
    }
}
```

- step 5: Run project and Check if beans are created (on console logs).
- step 6: Access ApplicationContext into main class using ApplicationContextAware interface (instead of @Autowired).

```
@SpringBootApplication
public class Main implements CommandLineRunner, ApplicationContextAware {
    private ApplicationContext ctx;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.ctx = applicationContext;
```

```
    }

    // ... main() and run() method
}
```

- step 7: In main class run() method, access spring beans and invoke business logic i.e. calcVolume(). Run application and check if you get correct output.
- step 8: Inherit BoxImpl class from bean life-cycle interfaces InitializingBean, DisposableBean, BeanNameAware, ApplicationContextAware and implement respective methods.
- step 9: Add @PostConstruct and @PreDestroy methods in BoxImpl class.
- step 10: Create a class CustomBeanPostProcessor implements BeanPostProcessor. Add log messgaes for "Box" type beans.

```
public class CustomBeanPostProcessor implements BeanPostProcessor {
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        if(bean instanceof Box)
            System.out.println("Before Initialization : " + beanName);
        return bean;
    }
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        if(bean instanceof Box)
            System.out.println("After Initialization : " + beanName);
        return bean;
    }
}
```

- step 11: Create CustomBeanPostProcessor @Bean in AppConfig.
- step 12: Run application and understand bean life cycle.

Explanation

- If @Configuration classes are in same package they are auto-detected due to @SpringBootApplication (@ComponentScan).

- All @Bean are by default singleton beans and hence created as soon as ApplicationContext is created (by SpringApplication.run()).
- ApplicationContext can be accessed in any Spring bean using ApplicationContextAware interface.
- ApplicationContext is used to access the beans using ctx.getBean() method.
- Spring bean provide various callback interfaces.
    - Reference: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-lifecycle
- Possible bean initialization callbacks
    - InitializingBean interface is legacy way of Spring bean initialization callback. Not recommended in modern applications.
    - XML configuration relies on `<bean ... init-method="initMethodName"/>`.
    - Annotation config usually follow JSR-250 annotation @PostConstruct.
- BeanPostProcessor
    - To perform certain actions on mutliple beans initialization, custom BeanPostProcessor is used.
    - There are multiple pre-defined BeanPostProcessor.
        - AutowiredAnnotationBeanPostProcessor
        - InitDestroyAnnotationBeanPostProcessor
        - BeanValidationPostProcessor

Bean creation

1. Constructor
2. Fields/setters initialized (DI)
3. BeanNameAware.setBeanName()
4. ApplicationContextAware.setApplicationContext()
5. CustomBeanPostProcessor.postProcessBeforeInitialization()
6. @PostConstruct method invoked
7. InitializingBean.afterPropertiesSet()
8. CustomBeanPostProcessor.postProcessAfterInitialization()
9. Bean is ready to use

Bean destruction

1. @PreDestroy method invoked
2. DisposableBean.destroy()

3. Object.finalize()
4. Bean will be garbage collected

Stereo-type annotations

- Spring container can auto-detect certain beans (without defining as @Bean method or XML config). It also manage their bean life cycle.
- Following annotations are auto-detected by Spring container using @ComponentScan.
    - @Component -- general-purpose (no special significance) spring bean.
    - @Service -- spring beans containing business logic
    - @Repository -- spring beans handling data/database connectivity
    - @Controller -- spring beans handling navigation, user-interaction in Spring web-mvc applications.
    - @RestController -- It is @Controller used for REST services.
    - @Configuration -- To be used for spring annotation config (not to be used as spring bean).
- Stereo-type annotations are always written at class level.
- Using @ComponentScan, all classes with above annotations in given package and its sub-packages will be instatiated as Spring beans.
- If no package is given, all classes in current package will be scanned for stereo-type annotations.
- This behaviour can be customized using includeFilters and/or excludeFilters.
- XML configuration equivalent of @ComponentScan `<context:component-scan>`.

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

Stereo-type Annoatations demonstration - sb04_stereoann

- step 1: Spring Starter Project
    - Fill group id, artifact id, Spring Boot=3.x.y, Language=Java, Java version=17, packaging=Jar -- click Finish
- step 2. Create package -- com.sunbeam.bank
- step 3. Create interface com.sunbeam.bank.Logger
    - methods: void log(String message);
- step 4. Create Logger implementation classes @Component com.sunbeam.bank.ConsoleLoggerImpl. Implement log() to print message on console.
- step 5. Create Logger implementation classes @Component com.sunbeam.bank.FileLoggerImpl. Implement log() to append message into log file.

    - Fields: logFilePath // hard-code in field initializer.
    - Methods: Constructor, Getters/Setters, log().
- step 6. Create @Configuration class com.sunbeam.bank.BankConfig class to @ComponentScan beans in this package.
- step 7. Make main class as CommandLineRunner and use ApplicationContext to check if logger beans are accessible. Execute application and test output.
    - consoleLogger = ctx.getBean(ConsoleLoggerImpl.class)
    - fileLogger = ctx.getBean(FileLoggerImpl.class)
- step 8. Detect config from non-root package using @ComponentScan or @Import.
- step 9. Create interface com.sunbeam.bank.Account and it's implementation class com.sunbeam.bank.AccountImpl.
    - Fields: id, type, balance, logger.
    - Methods: Constructor, Getters/Setters, deposit() and withdraw().
    - Log all deposit() and withdraw() transactions (if logger is not null).
    - Do not set logger in the constructor.
- step 10. In BankConfig class, create @Bean "acc" with hard-coded values (but no logger).
- step 11. In main class, assign ConsoleLoggerImpl into "acc" bean and call deposit() and withdraw(). Execute the application and test if logs are produced as expected.
- step 12. In main class, change "acc" logger to FileLoggerImpl and call deposit() and withdraw(). Execute the application and test if logs are produced as expected.
- step 13. Check if Logger bean is accessible. Execute application and observe the error.
    - logger = ctx.getBean(Logger.class)
- step 14. Make one of the logger implementation bean as @Primary to resolve the error (write @Primary on top of ConsoleLoggerImpl or FileLoggerImpl).

Assignments

1. Create an interface Sender with a method `void send(double value);`. Implement the method in three bean classes i.e. TcpSender, HttpSender, and UdpSender. In each implementation, write println() statement to print the given value on the console. Access these bean objects (using ctx.getBean()) into main class and call their send() methods. What error we get if we access ctx.getBean(Sender.class)? How to resolve the error?