Spring Framework Master Class – Java Spring the Modern Way

Spring is a dependency injection framework

What is dependency?

Web Layer -> Business Layer -> Data Layer

What if I a complex business service is directly instantiating using new

so remove instantiation and pass argument in service class constructor

Loosely coupled algorithm is preferred

Use annotations:

1 @Component for businessservice

2 @Autowired the dependency for interface

Terminology in Spring

1 Beans

2 Autowiring

3 Dependency Injection

4 Inversion of control

5 IOC Container

6 Application Context

1 Tight coupling

2 Loose coupling

3 Spring to manage beans

4 Dynamic Autowiring

5 Types of autowiring

6 Spring Modules

7 Spring Projects

8 Why Spring is Popular?

No of ways to setup Spring Boot

1. Using start.spring.io (spring initializr)
2. Using Spring Starter Project
3. Using Command Line
4. Using Maven and convert to Spring Boot

What are the Beans? @Component

What are the dependencies for beans? @Autowired

Where should I search for beans? com.in28minutes (@SpringBootApplication uses @ComponentScan)

Steps

1. @ComponentScan
2. @Autowired Create instance of autowired class
3. Do autowiring by constructor injection

@Primary This component will get priority over other interface implementation

Add both @ComponentScan and @Primary

Types of Injection

1. Constructor (If many arguments then problem)

2. Setter No need to provide a setter just write @Autowired

Spring Modules

1. Spring Core
   1. Spring Beans
   2. Spring Core
   3. Spring Context
   4. spEL
2. Data Access / Integration (connect to web service using queue IBMMQ, Kafka, RabbitMQ)
   1. Spring JDBC
   2. Spring ORM
   3. Spring JMS (put me
   4. Spring OXM (object to xml)
   5. Spring Transaction (commit and rollback)
3. Spring Web
   1. Spring Servlet
   2. Spring MVC
   3. Spring Portlet (outdated)
   4. Spring WebSocket

4. Spring AOP (logging, transaction, AspectJ)

5. Spring Aspects

6. Spring Messaging

7. Spring Instrumentation

8. Spring Test (Unit Test, Integration Test)

Things which are applicable to more than one layer 🡪 Cross cutting concerns

Spring Projects

1. Spring Boot

2. Spring Cloud (cloud native apps, eureka, api gateway, hystrix)

3. Spring Data (consistent data access)

4. Spring Batch (track batch apps)

5. Spring Security (web services should be secure, OAuth, OAuth2)

6. Spring HATEOAS (hypertext as the engine, href containing data about data)

7. Spring Integration (connect applications)

Why Spring is Popular?

1. Enables writing testable code (good to write junit and mockito)

2. Reduce amount of code (spring uses unchecked exceptions)

3. Flexibility architecture (modularity)

4. Stay with trend (spring cloud, spring boot)

Spring in Depth

1. Autowiring and Qualifiers

2. Bean Scopes and Life Cycle

3. IOC Container and ApplicationContext

4. XML and Java ApplicationContext

5. Component Scan

6. External Properties

7. Container & Dependency Injection (CDI)

Container -> Service

Service -> Data

Data -> JDBCTemplate

Autowiring

1. Autowired by Name

2. by Type

3.

4.

Order of preference:

1. Using @Primary (only 1 class will be named as @Primary)
2. Using bubbleSortAlgorithm in impl
3. @Qualifier in ServiceImpl

Bean Scopes

1. Singleton (by default) (one instance per Spring Context) (everytime you request same beans will be returned)

2. Prototype (new bean whenever requested)(two different instances are returned everytime)

3. Request (one bean per HTTP Request)

(one bean per flow of request )

1. Session (one bean per HTTP session)

@Scope(“prototype”)

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

@Scope(ConfigurableBeanFactory.SCOPE\_SINGLETON)

PersonDAO uses JdbcConnection

PersonDAO is Singleton and JdbcConnection is Prototype

If we run it we get both as Singleton but we should have got Singleton and Prototype for PersonDAO and JdbcConnection respectively so to resolve that we need to use a proxy

@Scope(value = ConfigurableBeanFactory.***SCOPE\_PROTOTYPE***, proxyMode = ScopedProxyMode.***TARGET\_CLASS***)

Difference between Singleton GOF and Spring Singleton

One Instance per JVM = GOF Singleton

One instance per Application Context = Spring Singleton

ComponentScan

@ComponentScan (other package name)

Spring Bean LifeCycle maintained by IOC Container

@PostConstruct

@PreDestroy

@PostContruct

First dependencies are populated then do some action

@PreDestroy

Things to do before beans are destroyed

JPA and Hibernate

CDI and

Spring Without Spring Boot

@Configuration

@ComponentScan

Embedded Tomcat

Java 7 Auto Closeable

Read values from external properties file

@Value(“${external.service.url}”)

@PropertySource(“classpath:app.properties”)

XML Jdbc Connection

IOC Container vs ApplicationContext vs BeanFactory

Controller -> Service (using new)

Controller -> Service (using @Autowired) IOC Framework creates controller and service

IOC Container implementations:

1. BeanFactory (wiring)
2. ApplicationContext (i18n, AOP features, webapplicationcontext)

Stereotype difference

@Controller vs @Service @Repository vs @Component

Controller Layer: @Controller

Business Layer: @Service

Data Layer: @Repository

Other: @Component

JUnit

What is Unit Testing?

What is JUnit?

Before deploying testing a specific method or class

Writing individual test for each method

We can combine some tests as well

JUnit tests are automated and run everytime

JUnit used for continuous integration

JUnit : absence of failure is success

**package** com.in28minutes.junit;

**import** **static** org.junit.Assert.\*;

**import** org.junit.Test;

**public** **class** AssertTest {

@Test

**public** **void** test() {

*assertEquals*(1, 1);

*assertTrue*(**true**);

*assertFalse*(**false**);

*assertNull*(**null**);

*assertNotNull*(1);

*assertArrayEquals*(**new** **int**[] { 1, 2 }, **new** **int**[] { 1, 2 });

}

}

**package** com.in28minutes.junit;

**import** **static** org.junit.Assert.*assertEquals*;

**import** org.junit.After;

**import** org.junit.AfterClass;

**import** org.junit.Before;

**import** org.junit.BeforeClass;

**import** org.junit.Test;

**public** **class** MyMathTest {

// 1.

MyMath myMath = **new** MyMath();

@BeforeClass

**public** **static** **void** beforeClass() {

System.***out***.println("Before Class");

}

@Before

**public** **void** before() {

System.***out***.println("Before");

}

@Test

**public** **void** testSum\_with\_3numbers() {

System.***out***.println("Test 1");

*assertEquals*(6, myMath.sum(**new** **int**[] { 1, 2, 3 }));

}

@Test

**public** **void** testSum\_with\_1numbers() {

System.***out***.println("Test 2");

*assertEquals*(3, myMath.sum(**new** **int**[] { 3 }));

}

@After

**public** **void** after() {

System.***out***.println("After");

}

@AfterClass

**public** **static** **void** afterClass() {

System.***out***.println("After Class");

}

}

Mockito

OLD

Using Stubs (disadvantage for every scenario update the stub)

**package** com.in28minutes.mockito.mockitodemo;

**public** **interface** DataService {

**int**[] retrieveAllData();

}

**package** com.in28minutes.mockito.mockitodemo;

**public** **class** SomeBusinessImpl {

**private** DataService dataService;

// get largest element

**int** findGreatestFromAllData() {

**int**[] data = dataService.retrieveAllData();

**int** greatest = Integer.***MIN\_VALUE***;

**for** (**int** value : data) {

**if** (value > greatest) {

greatest = value;

}

}

**return** greatest;

}

}

**package** com.in28minutes.mockito.mockitodemo;

**import** **static** org.junit.jupiter.api.Assertions.*assertEquals*;

**import** org.junit.jupiter.api.Test;

**class** SomeBusinessStubTest {

SomeBusinessImpl businessImpl = **new** SomeBusinessImpl();

**public** SomeBusinessTest(SomeBusinessImpl businessImpl) {

**super**();

**this**.businessImpl = businessImpl;

}

@Test

**void** testFindGreatestFromAllData() {

**int** result = businessImpl.findGreatestFromAllData();

*assertEquals*(24, result);

}

}

**class** DataServiceStub **implements** DataService {

@Override

**public** **int**[] retrieveAllData() {

**return** **new** **int**[] { 24, 6, 15 };

}

}

New Way: Using Mockito

**package** com.in28minutes.mockito.mockitodemo;

**public** **interface** DataService {

**int**[] retrieveAllData();

}

**package** com.in28minutes.mockito.mockitodemo;

**public** **class** SomeBusinessImpl {

**private** DataService dataService;

**public** SomeBusinessImpl(DataService dataService) {

**super**();

**this**.dataService = dataService;

}

// get largest element

**int** findGreatestFromAllData() {

**int**[] data = dataService.retrieveAllData();

**int** greatest = Integer.***MIN\_VALUE***;

**for** (**int** value : data) {

**if** (value > greatest) {

greatest = value;

}

}

**return** greatest;

}

}

**package** com.in28minutes.mockito.mockitodemo;

**import** **static** org.mockito.Mockito.\*;

**import** **static** org.junit.jupiter.api.Assertions.*assertEquals*;

**import** org.junit.jupiter.api.Test;

**class** SomeBusinessMockTest {

@Test

**void** testFindGreatestFromAllData() {

DataService dataServiceMock = *mock*(DataService.**class**);

*when*(dataServiceMock.retrieveAllData()).thenReturn(**new** **int**[] { 24, 15, 3 });

SomeBusinessImpl businessImpl = **new** SomeBusinessImpl(dataServiceMock);

**int** result = businessImpl.findGreatestFromAllData();

*assertEquals*(24, result);

}

@Test

**void** testFindGreatestFromAllData\_ForOneValue() {

DataService dataServiceMock = *mock*(DataService.**class**);

*when*(dataServiceMock.retrieveAllData()).thenReturn(**new** **int**[] { 15 });

SomeBusinessImpl businessImpl = **new** SomeBusinessImpl(dataServiceMock);

**int** result = businessImpl.findGreatestFromAllData();

*assertEquals*(15, result);

}

}

Mockito Annotations

@Mock

DataService dataServiceMock;

@InjectMock

SomeBusinessImpl businessServiceImpl;

@RunWith(MockitoJunitRunner.class)