**Spring framework – Master to beginner**

**Section1**

Spring is a dependency injection framework.

**Section 2**

@Component is used to tell spring to manage instances.

@Autowired is used to tell spring that there is a dependency among the instances of the components it manages.

**Spring terminology:**

**Beans**: the different instances that spring manages are called beans. The different objects managed by spring framework.

**Autowiring**: the process where spring identifies the dependencies, identifies the matches for dependencies, and populates them.

**Dependency injection**: dependency injection is the injection of a method into a complex class or bean.

**Inversion of control**: The control is taken from the class that needs the dependency to the framework to identify the dependency.

**IOC Container**: Generic term to represent anything that is implementing inversion of control.

**Application Context**: In spring frame work, the most important IOC container is the application context.

Application context is the place where all the beans are created and managed. It is the most important part of spring framework. It is where all the core logic of spring framework happens.

**Section 3**

Spring projects can be made using start.spring.io – generate a maven project with java and spring boot version (greater than 2). Dependencies can be added at this step.

Tight coupling: where a method is dependent on an algorithm inside it. Which required that the method be modified every time if there is a change in algorithm. This can be changed by created a separate method for that algorithm and invoking that method inside this method. Using interfaces helps it become more loosely coupled.

We can tell spring what the beans are by using @Component and tell it the dependencies by using @Autowired. We get the application context by running SpringApplciation.run(Main class, args of main class).

We can get beans from the application context by calling applicationContext.getBean().

Go to application.properties and put logging.level.org.springframework = debug to put the spring frame work in debug mode.

There are two types of injection for Autowiring – constructor injection and setter injection. If there are two components for injection, then @primary is used to define which component is the primary one.

If a setter is made instead of a constructor, then the injection is made by setter. Even if there is neither a constructor or a setter, setter injection is done by default. (Earlier recommendation was to use constructor injection for mandatory injections and setter injections for optional dependencies. However, Autowiring automatically takes care of dependencies.)

**Section 4**

In case of conflicts in Autowiring where the interface has multiple components. The easiest way to solve the conflict is to use the name of the required class when initializing the interface. @Primary has higher preference over the name of the initialized variable.

Another option is to use @Qualifier(“”) for specifying the name of the component. This qualifier has to be placed over the @Autowired and the @component class. Auto wiring by name is preferred as it is easier.

Bean Scope: singleton – one instance per spring context

Prototype – new bean whenever requested

Request – one bean per http request

Session – one bean per http session

When we request the application for the same bean multiple times, we get the same instance of the bean. The scope is singleton by default.

To get different instances every time, we can create @Scope (“prototype”) on top of the class. It is singleton by default.

Whenever we are trying to get a bean, but one of its dependencies is a prototype, then we should create a proxy (@Scope (value=”prototype”, proxyMode=ScopedProxyMode.TARGET\_Class)) for the dependency.

Whenever we create a spring boot application, a component scan automatically runs on the current package and the sub packages. Therefore, if there is a component in another package, we will have to add a component scan below @SpringBootApplication - @ComponentScan (“package name”)

Spring automatically manages the life cycle of a component. It is done by the IOC container.

@PostConstruct can be used to call a method which automatically runs after the bean is created.

@PreDestroy can be used to call a method which automatically runs before the bean is destroyed.

CDI tries to define common annotations for injections such as @Inject: for Autowiring

@Named: for component and qualifier

@Singleton: for singleton classes

To use CDI we need to define groupId – javax.inject, artifiactId – javax.inject and version 1 in pom.xml. It is basically an interface to use annotations.

Therefore for @Component I can use @Named and for @Autowiring I can use @Inject.

Spring-boot-starter is the spring boot dependency which enables sprint boot. If we aren’t using spring boot, we can change it to spring-core (in pom.xml) and then introduce spring-context for application context. Spring boot provides @SpringBootApplication to initialize an application context. We would need to use @Configuration to initialize an application Context (using annotationConfigApplicationContext). Spring boot also automatically defines a component scan where @SpringBootApplication is used. Therefore a @ComponentScan needs to be defined as well.

A spring project can be defined using an XML file instead of annotations. Before spring 2.5, everything had to be wired in XML. ApplicaiontContext.xml needs to be created and populated using XML metadata from the internet. We can define beans and their dependencies here.

Eg: <bean id=”” class=””> - for component

<property name=”” ref=”bean name”></property> - for Autowiring

For loading the application context, we can create a ClassPathXmlApplicationContext using ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext(“applicationContext.xml”);

We can get the beans which are loaded by called getBeanDefinitionNames.

In java context, we need to specify context in web.xml as follows :

<beans xmllns:context = “htt[://www.springframework.org/schema/context”

Xsi:schemalocation =”<http://www.springframework.org/schmea/context>

<http://www.springframework.org/schmea/context/spring-context.xsd>”>

Now, we can specify component scan in xml as follows <context:component-scan base=package=””/>

We can auto wire or use component scan while using xml to get bean data.

Inversion of control or IOC is a concept or a generic item which creates and manages dependencies. Spring has two methods of IOC: Application context and bean factory.

Bean factory only provides the wiring of dependencies, and basic management of beans ( all the concepts inside application core). Application context provides more features than bean factory. Can generally be assumed as bean ++.

Component Annotations: @Component - generic (can be used in purposes where we aren't sure whether the component is confined to a particular layer of the web app)

@Controller - used for MVC (UI layer)

@Service - for business layer

@Repository -for data layer

Using the appropriate annotation for the corresponding layer helps us use the features associated with the layer such as for @Repository, all the JDBC exceptions will be classified and handled automatically.

We can read values from external files by using the annotation @Value("${propertyName}"). The property file can be configured by defining @PropertySource("classPath:file.properties") in the main class

**Section 5**

Unit testing and JUnit

Unit testing is the process of testing individual methods by writing programs which run the test cases. JUnit is a framework which helps us do unit testing. These are automated tests. Therefore, we can keep running the code anytime there is a change of code and we can get notifications if it fails.

It is a good practice to create a separate source folder for unit tests. Create JUnit test case object using the wizard. There is a @Test annotation which indicates that it’s a JUnit method. JUnit sees the absence of failure as a success. If the test is a success, the JUnit green bar would appear.

JUnit has many methods like assertEquals(expected, result) (assert true, assert null etc) to know whether the expected value has been obtained as result. We can run a class as JUnit test and it will continuously run till it fails.

@Before can be used to run a method before running @Test. There is no guarantee of the order of when @Test methods run. @Before method will be invoked before every test. And similarly we have @After which runs after every test.

There are also class level annotations @BeforeClass and @AfterClass which are static methods. @BeforeClass runs before any test runs, and @AfterClass runs after every test method completes.

**Section 6**

Mockito

Mockito is a testing framework for java. Mockito is part of the spring boot dependencies by default.

A stub is a controllable replacement for an existing dependency in the system. We can use stubs to run tests without using the dependency directly. It is basically a mock object of sorts. However, using stubs has disadvantages such as writing a lot of code for every step, creating new stubs for different scenarios. Mockito is used to create mocks which dynamically generate data to help with testing.

A mock is created by using mock(). ClassA aMock = mock (ClassA.class)

We can use the method when to return data as required. When(aMock.getData()).thenReturn(new int[] {1, 2, 3});

Annotations can be used to create mocks easily.

@Mock – to create a mock object.

@InjectMocks - to inject mocks

@RunWith() - on top of test class to run as MockitoJUnitRunner.class

We can write thenReturn after a thenReturn for when methods to repeatedly return multiple values. Eg: When().thenReturn().thenReturn(). Mockito.anyInt() can be used to denote any integer. There are many ‘any’ methods for almost any data type. However, while using multiple scenarios, we cannot combine specific and generic test cases. Either all the scenarios in a test case must be generic or specific.

**Section 7**

Unit testing with spring framework

We need to add junit and Mockito as a dependency in pom.xml.

Group Id and artifact Id – junit, GroupId – org.mockito, artifact id – Mockito-core

While testing with spring framework, we need to load the context in the test class. Therefore, the test class will be defined with @ContextConfiguration(Classes=MainClass). MainClass is the class where the java context is loaded. We also need to add @RunWith(SpringRunnger.class) to specify the runner.

If we are using an XML configuration, we need to change the context configuration to the location of the XML. @ContextConfiguration(locations=”/applicationContext.xml”).

When we are testing with Mockito, we don’t need the spring context. So we are remove the context configuration.

**Section 8**

Spring boot

Spring boot is a framework which provides embedded servers, metrics, health checks and externalized configuration. However, it is not a code generation or an application server or a web server.

It has two auto configuration – Web, JPA

Spring boot supports starter projects. I.e., if we want to make a web application, if we use spring boot starter project and add a starter called spring boot starter web, we get spring mvc, spring core, a validation framework and a login framework. Similarly, we can use a spring boot JPA starter to get JPA, default implementation, hibernate and auto configuration for it.

It supports databases – Tomcat, Jetty or Undertow

With spring boot comes a concept called embedded server which helps us package a server along with the application jar. Therefore, the corresponding OS runs the corresponding server.

Spring boot provides a number of production features such as monitoring, externalized configuration etc.

When we set up a spring boot project using the initializer, we have the following folder structure:

Src/main/java - for java code

Src/main/resources - for content and application.properties

Now we’ll detail the steps to create a simple REST service. Tomcat runs on port 8080 by default. We can create the required beans. And then create a rest controller by giving the annotation @RestController.

In order to set a URL in the controller, we can use the annotation @GetMapping(“/URL”) above a method to hit that method every time the url hits port 8080.

@SpringBootApplication indicates it’s a spring context, enables automatic component scan and it automatically enables auto configuration. Spring boot auto configure automatically configures a lot of dependencies and servers and other things at the time of running the spring application. At startup, spring boot framework triggers the auto configuration jar and it looks through classes on the class path. If there is a specific class on the class path, then the dispatcher servlet is configured for that class and a view resolver etc. Spring boot looks at frameworks available on the class path and the existing beans configured for the application and based on this provides the configuration. The process concerned with auto configuration can be found out by using logging level or by using spring actuator.

|  |  |  |
| --- | --- | --- |
| **Spring** | **Spring MVC** | **Spring Boot** |
| Core problem it solves is testability | It is concerned with developing web applications and RESTful services. | Spring boot helps with automatically configuring all the dispatcher servlet, view resolver and other jars automatically. |
| Core of spring framework is IOC (inversion of control) or dependency injection. | It is great for separation of concerns – view, model, controller etc. | Provides different starter features such as – Spring boot starter actuator for monitoring |
| Most important feature is dependency injection. | Spring boot internally uses Spring MVC framework while developing web apps | Has starters for logging |
| It helps to build loosely coupled applications |  | Has starters for picking different embedded containers. |
| It also helps to reduce boiler plate or plumbing code |  |  |
| It also provided good integration with other frameworks |  |  |

Spring boot starter web is used by default for web applications. Spring boot starter web is the preferred starter for sprint boot to develop web applications and RESTful services. Spring boot starter web defines various dependencies such as spring boot starter tomcat, spring web, spring mvc framework, validation API – hibernate and starter JSON. Conversion to JSON automatically happens when returning a bean in spring boot because of JSON starter. One of the dependencies in spring boot web is the spring boot starter. Spring boot starter has spring bring auto configure, logging, monitoring etc. There is another starter called spring boot starter test which helps to write Junit tests and mockito.

Spring boot starter JPA is another starter used for ORM projects. It has various dependencies such as JDBC, AOP, Hibernate, transaction API etc.

Different starters offered by spring boot are:

* Spring boot starter web services: to develop SOAP Web services
* Spring boot starter web: for Web and RESTful applications
* Spring boot starter test: for unit tests and integration tests
* Spring boot starter JDBC: for applications using JDBC
* Spring boot starter hateoas: to add HATEOAS features in the services
* Spring boot starter security: has support for basic authentication, oauth authentication and oauth 2 authentication.
* Spring boot starter data jpa: to do JPA, uses hibernate as the ORM framework
* Spring boot starter cache: to implement caching. Has support for wide range of caching including distributed caching like hazelcast.
* Spring boot starter data rest: if using JPA, we can use this starter to expose spring boot jpa entities as Restful API services.

To use spring boot actuator, we need to go to pom.xml and add spring-boot-starter-actuator under the group id org.springframework.boot. Actuator brings in monitoring about the application and helps us read a lot of metadata about the application. Actuator exposes a lot of REST API’s which are according to the HAL standard. And to view this we can use the HAL browser by specifying spring-data-rest-hal-browser in pom.xml. The actuator URL has changed with every release. Now with spring boot 2.00, the actuator URL is localhost:8080/actuator. To enable the stuff actuator provides, we can go to application.properties and input “enable management.endpoints.web.exposure.include=\* “. We can use the HAL browser to browse through the actuator (localhost:8080/browser/index.html). The different features in actuator will have a performance impact.

Every time we make a change while developing a web app, we will have to restart the server to see the change. We can use spring-boot-devtools as a dependency to help with this. Any java changes we make will be automatically picked up.

**Section 9**

Spring AOP

While creating a spring boot project, add the dependency ‘AOP’. AOP is an approach to implement cross cutting concerns.

We need to implement CommandLineRunner for the main application. It has a method called run. We can write methods inside the run method and whenever the spring boot application loads up, run method is run.

@Aspect is used to define a class as an aspect of AOP.

( \* PACKAGE.\*.\*(..)) – first star refers to return type, PACKAGE refers to package, second \* refers to any class and the last one any method with any arguments. Therefore we can use @Before(“execution(\* package name.\*.\*(..)) above a method to intercept any calls from that class and run the method before that method call.

We can use a data type called JoinPoint to get details of the intercepted point by using JoinPoint as an argument in the method where @Before is defined.

* Point cut is the expression ( \* PACKAGE.\*.\*(..)). Here we can specify the methods we want to intercept.
* Advice is the login inside the intercept method.
* Aspect is a combination of the point cut and the advice.
* JoinPoint is the specific execution instance.
* Process where the whole thing is weaved around our code is called weaving.
* The framework which does this is called weaver.

@AfterReturning(\*) can be used to run some logic after an intercepted method finishes executing.

@AfterThrowing(\*) can be used to intercept any exception and run code after the exception is thrown.

@After can be used as a common interception method irrespective of whether it’s a return value or exception.

@Around(\*) is an annotation which helps us to run some logic before a method is executed and after a method is executed. ProceedingJoinPoint is another data type which helps us to proceed with the method after interception. And using ProceedingJoinPoint.proceed() lets the method proceed and we can write logic for steps after execution of the method.

It is generally better to have a separate file to store the point cuts. We can use the annotation @Pointcut(\*) to define a point cut. This point cut can be specified above a class and its qualified name can be set inside @Before.

@Pointcut(“bean(dao)”) – will get any bean which starts with dao

@Pointcut(“bean(\*dao\*)”) – will get any bean which contains the word dao

@Pointcut(“within()”) – will get everything within the package

We can create custom annotations for our needs. For eg: if we create an annotation @TrackTime above a method, we can right click and create an annocation and eclipse will create an @interface called TrackTime. We can specify the target which could be method or class and also the retention using annotaions @Target and @Retention.

Eg: @Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

We add annotation point cuts as fllows PointCut("@annotation(\*)").

**Section 10**

Spring JDBC

Import JDBC depdency while initiating springboot if we want to use JDBC. We can also use H2 to create an in memory data base for which we'll need to import H2 dependency.

In order to use H2 as an in memory database, we need to import it as a dependency while initiating spring boot. It will automaticall add the dependency in pom.xml. We can configure H2 for the application by writing spring.h2.console.enabled=true in application.properties. We can type localhost:8080/h2-console to open the H2 console. We can let the settings be embedded H2 and the driver class org.h2.Driver and the JDBC URL - jdbc:h2:mem:testdb. The h2 console is similar to almost every other database console. In memory database is only live as long as the application is running. It gets wiped when the application goes offline.

We can create a sql file in src/main/resources called data.sql and the data in this data.sql file will be automatically called to populate the H2 database everytime the program starts. This is another feature of spring boot. Data is populated by writing the queries one after another in the sql file.

We use @Repository to create a class which talks to the database (instead of @Component). An object of JdbcTemplate is created with @Autowired to make spring create a Jdbc connection. Then we can execute queries by writing jdbcTemplate.query("quer", new BeanPropertyRowMapper(Person.Class)). Person.Class is a class with the parameter names which match the column names of the table returned by the query.

**Spring quick guide**

**@SpringBootApplication** – for the main class (provided by spring boot)

**@Configuration** – for the main class(without spring boot)

**@Component** – for searching class (for telling spring that the class is a bean)

**@Primary –** for setting a component as primary

**@Autowired** – for Autowiring (to tell the class that the method or parameter is a dependency)

**@ComponentScan(“com.in28minutes.”)** – for scanning a package (base package by default), Spring boot automatically scans the current package and sub packages)

**@Scope(Value=”prototype”, proxyMode = ScopedProxyMode.target\_class) –** scope for the classes(singleton or prototype) – singleton by default

**@Qualifier(“bubble”) –** for specifying the names of components

**@PostConstruct –** this code runs after a bean is created and the dependencies are defined (must be void)

**@PreDestroy** – Runs just before the bean is destroyed

**CDI – Context dependency injection :** Defines an interface to do things such as

@Inject(@Autowired)

@Named(@Component&@Qualifier)

@Singleton(Defines a scope of singleton)

@Qualifier(Same)

@Scope(setting the scope – singleton or prototype)

@Singleton (Easily define singleton)

Add the following to pom.xml

{<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

}

The functionality does not change whether CDI or normal annotations are used. Spring can be used as the implementation with CDI annotations since CDI is a java EE standard.

**To remove Spring Boot**

Replace the spring boot dependency with these two:

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

@SpringBootApplication is provided by spring boot. We can run a spring application without spring boot by replacing this with @Configuration.

SpringApplication is a spring boot class too. So we will have to get an application context using ACAC (AnnotationConfigApplciationContext).

We also need to define a @ComponentScan which was done automatically by spring boot. We define this below @Configuration.

**Component Annotations -** @Component - generic

@Repository - For data layer

@Service - For business layer

@Controller - For UI layer

**Reading value from external file - @Value("${propertyName}") -** the property file can be configured by defining @PropertySource("classPath:file.properties") in the main calss

**JUnit** @Test – to define a test method

@Before – runs before every test

@After – runs after every test

@BeforeClass - class level methods (static methods)

@AfterClass - class level methods (static methods)

**Mockito** Create mocks using “mock()” and return data using “when()”

@Mock – to create mocks

@InjectMocks – to inject Mock

@RunWith() – to run as MockitoJUnitRunner.class

**REST Service**

@RestController – for rest controller

@GetMapping – for get url

**Spring APO**

@Aspect – define a class for AOP

@Before – intercept method calls

Use joinPoint to get details of the intercepted point.

@AfterReturning – used to run after intercepted method finishes

@AfterThrowing – used to run after an exception is thrown

@After – after annotation is used generically

@Around – is used to do something before and after the execution of a method

ProceedingJoinPoint allows us to continue with the method execution after interception.

@Pointcut – to define a point cut