**Spring Bean is an object that is instantiated, assembeled and managed by spring IOC.**

**IOC can be done in 2 ways:-**

* [**BeanFactory**](https://www.geeksforgeeks.org/spring-beanfactory/)
* [**ApplicationContext**](https://www.geeksforgeeks.org/spring-applicationcontext/)

**Spring used XML based configuration, annotation or java based configuration to create Spring bean**

1. **Using xml based configuration - BeanFactory**

bean-factory-demo.xml

<?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.xsd">

**<bean id="student" class="com.gfg.demo.domain.Student">**

**<constructor-arg name="name" value="Tina"/>**

**<constructor-arg name="age" value="21"/>**

</bean>

</beans>

Java main class – to access the bean

**BeanFactory factory = new ClassPathXmlApplicationContext**("bean-factory-demo.xml");

Student student = (Student) factory.getBean("student");

System.out.println(student);

1. **ApplicationContext**

ApplicationContext is the superset of BeanFactory, whatever features provided by BeanFactory are also provided by ApplicationContext.

**BeanFactory does not support Annotation-based configuration whereas ApplicationContext does.**

**Different Type of Dependency Injection in Spring**

1. Constructor \*\*\*\*\*(Most commonly used)
2. Field
3. Setter method

Annotation To create Srping Bean

1. @Component
2. @Service
3. @Repository
4. @Controller

Java Based Configuration

1. @Configuration
2. @Bean

Instantiating/Wiring Spring Bean

They are wired using **ApplicationContext**, in spring boot application, ApplicationCOntext is usually instantiated automatically when application starts ----SPringApplication.run()

public class Demo {  
 public Demo(){  
 System.*out*.println("inside demo constructor");  
 }  
 public void firstmethod(){  
 System.*out*.println("inside first method");  
 }  
}

To access this class we we need to do –

@SpringBootApplication  
public class MathServiceApplication {  
  
 public static void main(String[] args) {  
 **Demo d = new Demo();  
 d.firstmethod();**}  
}

But with Spring – we can use this – **Add @Component in this Demo class** to make it a bean and the run method returns the ApplicationContext which handles the beans. So

**@Component**

public class Demo {  
 public Demo(){  
 System.*out*.println("inside demo constructor");  
 }  
 public void firstmethod(){  
 System.*out*.println("inside first method");  
 }  
}

To access this class methods now, we don’t have to create a new object

**@SpringBootApplication**  
public class MathServiceApplication {  
  
 public static void main(String[] args) {  
 var **applicationContext** = SpringApplication.***run***(MathServiceApplication.class, args);  
 applicationContext.**getBean(Demo.class).firstmethod();**

Constructor Dependency Injection in SpringBoot

This approach is the recommend approach for doing the DI via spring boot

When spring sees a class with a constructor that takes parameters, it attempts to automatically inject the necessary dependencies into those parameters, it’s called constructor injection. If there are multipe parameters in the constructor, spring resolves each parameter by matching it to a corresponding bean from the spring application context.

Creating a class – which will call a another class with one single constructor

**@Component**public class SIngleConstructor {  
  
 private Demo demo;  
  
 public SIngleConstructor(Demo demo){  
 this.demo = demo;  
 }  
 public void getDemoMethod(){  
 demo.firstmethod();  
 }  
}

**With Spring how we can call this class method**

@SpringBootApplication  
public class MathServiceApplication {  
  
 public static void main(String[] args) {  
 var applicationContext = SpringApplication.*run*(MathServiceApplication.class, args);  
 *//without spring boot how to access the singleconstructor***SIngleConstructor s = new SIngleConstructor(new Demo());  
 s.getDemoMethod();**

*//to access the singleConstructor method using spring* **applicationContext.getBean(SIngleConstructor.class).getDemoMethod();**

**Behind the scene Spring is adding the @Autowired annotation to the constructor here**

**@Autowired**

public SIngleConstructor(Demo demo){  
 this.demo = demo;  
 }

Now what if we have 2 parameter constructor

Let’s say there are two classes – Demo and Demo2 and they are getting called into a another class constructor

public SIngleConstructor(Demo firstclass,SecondClass secondclass ){  
 this.firstclass = firstclass;  
 this.secondclass = secondclass;  
}

it will fail because when we have a constructor more then one parameter with object, **you need to add the @Autowired above it** and also in all of the class **@Component as well**

**@Autowired**

public SIngleConstructor(Demo firstclass,SecondClass secondclass ){  
 this.firstclass = firstclass;  
 this.secondclass = secondclass;  
}

**Field Based Dependency Injection**

**It’s less recommended approach, here @Autowired annotation will automatically injects the dependency of a class via field.**

@Component  
public class FieldLevelInjection {  
   
 *// this is field level injection* @Autowired  
 private Demo firstclass;  
  
 *// this is field level injection* @Autowired  
 private SecondClass secondclass;  
  
 *// we removed the constructor here, becuase they won't be required in field level injection* public void getDemoMethod(){  
 firstclass.firstmethod();  
 secondclass.SecondCLassMethod();  
 }  
}

**How does Value Injection Works:-**

1. Create a java class with **@configuration annotaition**
2. Create some private variable **with @Value annotation** and provide the path of the properites variable name
3. Provide this package name where this java config class you have created with **@ComponentScan(basepackage={“”})**

If you have a properties file – and you want to access the variable directly in the class – then mark that class @configuration and provide the value like this

Playwright.browser = chrome

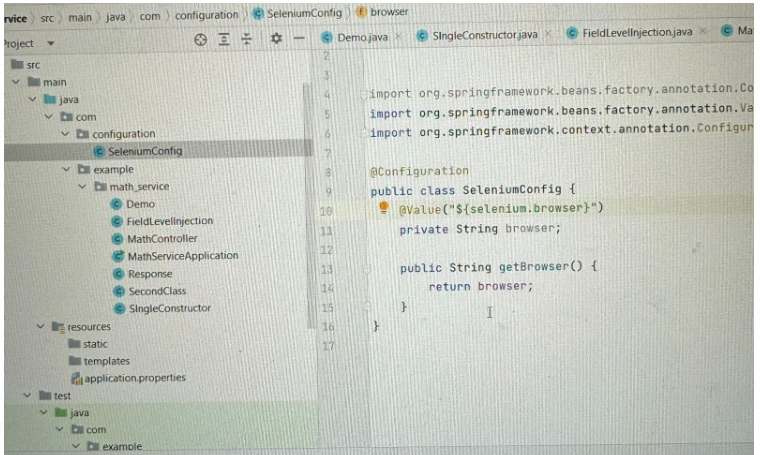
Playwrite.data = host.com

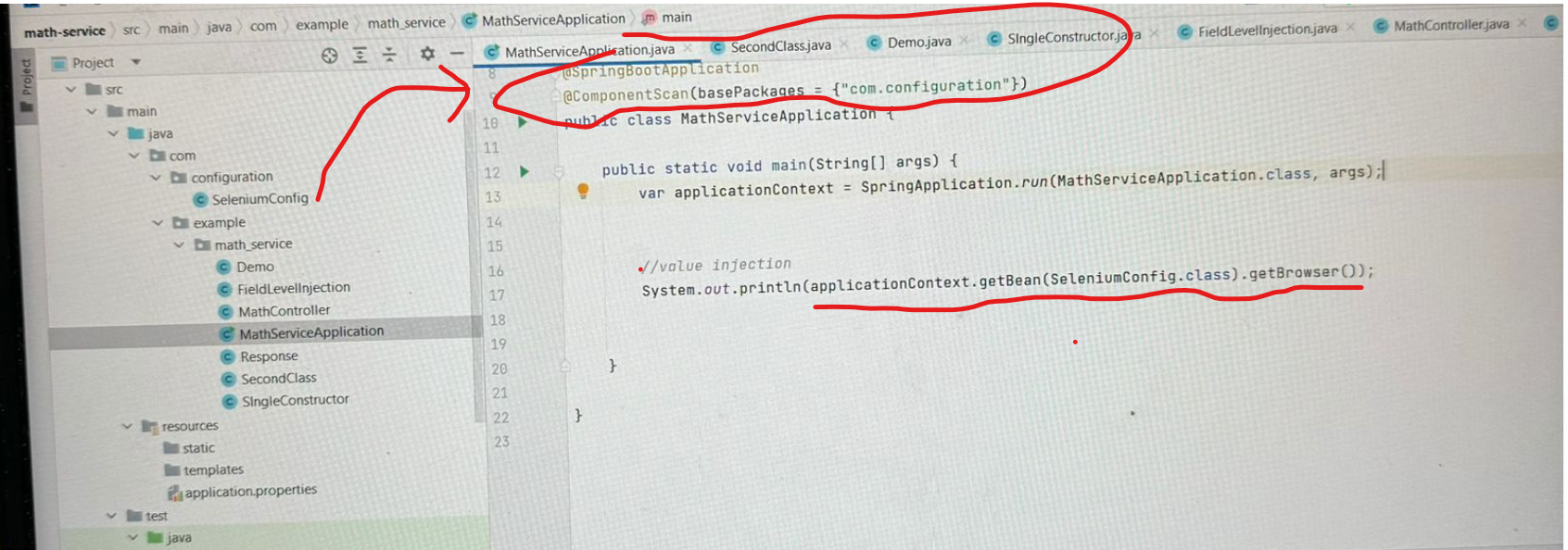
@Value(“${playwright.browser:chrome}”)

Private String browser;

**Component Scanning**

If your configuration file is outside of the runnable package then you need to give the path of the package where you have kept your configuration file – config file is nothing just a file from where we are fetching our values from properties file.





Or for componentScanning we can also use the

**@Import(SeleniumConfig.Java)**

@SpringBootApplication  
*//@ComponentScan(basePackages = {"com.configuration"})***@Import(SeleniumConfig.class)**  
public class MathServiceApplication {  
  
 public static void main(String[] args) {  
 var applicationContext = SpringApplication.*run*(MathServiceApplication.class, args);  
  
  
 *//value injection* System.*out*.println(applicationContext.getBean(SeleniumConfig.class).getBrowser());  
  
  
 }

**Different Profiles ---**

Let’s say you have the multiple env and you can have the multiple application.properties file

Create different profiles – i.e

1. application-dev.properties
2. application-qa.properties

when running the code, edit configuration and set **spring.profiles.active=qa**

we can also add the @Profile(“dev”) or @Profile(${env}) and pass this env at run time

Selenium – Spring boot setup ::

Add the selenium.browser and url in application.properties file and do there field level injection in below

**@Configuration**  
public class WebDriverConfig {  
  
  **@Value("${selenium.browser::chrome}")**  
 private String browser;  
  
 **@Value("${selenium.url}")** private String url;  
  
 public WebDriver getWebdriver(){  
 WebDriver driver = new ChromeDriver();  
 driver.get(url);  
 return driver;  
 }  
}

In the test folder if I want to get this bean wither I can do this

* 1. Use @Import

**@SpringBootTest**  
**@Import(WebDriverConfig.class)**class MathServiceApplicationTests {  
  
 **@Autowired**  
 private WebDriverConfig webdriverconfig;  
  
 @Test  
 void contextLoads() {  
 WebDriver driver = webdriverconfig.getWebdriver();  
  
 *//3. Navigate to a specific URL* driver.navigate().to("http://eaapp.somee.com");

}}

* 1. Use ComponentScan – so that It can scan for the bean

**@SpringBootTest**  
**@ComponentScan({packkagepath})**class MathServiceApplicationTests {  
  
 **@Autowired**  
 private WebDriverConfig webdriverconfig;  
  
 @Test  
 void contextLoads() {  
 WebDriver driver = webdriverconfig.getWebdriver();  
  
 *//3. Navigate to a specific URL* driver.navigate().to("http://eaapp.somee.com");

}}

Note – How to use @Bean annotation (Used on methods with @configuration at the class level, mainly used for third party libraries)

To directly use the Webdriver instance in the tests, add @Bean tag in the method where we are returning the webdriver instance and then in the class create a direct variable of driver instead of WebDriverConfig.

**@Bean**  
public WebDriver getWebdriver(){  
 WebDriver driver = new ChromeDriver();  
 driver.get(url);  
 return driver;  
}

Inside the Test -----------

**@SpringBootTest  
@Import(WebDriverConfig.class)**  
class MathServiceApplicationTests {  
  
 **@Autowired  
 private WebDriver driver;**  
  
 @Test  
 void contextLoads() {driver.navigate().to("http://eaapp.somee.com");

*//POM Code  
HomePage homePage = new HomePage(driver);  
var loginPage = homePage.clickLogin();  
homePage = loginPage.performLogin("admin", "password");  
var employeeListPage = homePage.clickEmployeeList();  
var createEmployeePage = employeeListPage.clickCreateNew();  
createEmployeePage.createNewEmployee("AutoUser2", "22222", "autotestuser2@gmail.com", "20000", "Middle");}*

Now let’s introduce the field level injection as well in the pom –

Add @Component in the all pages classes and create there page class variable in the Test class and add @Autowired in it.

@SpringBootTest  
@Import(WebDriverConfig.class)  
class MathServiceApplicationTests {  
  
 @Autowired  
 private WebDriver driver;  
  
 @Autowired  
 private HomePage homePage;  
  
 @Autowired  
 private CreateEmployeePage createEmployeePage;  
  
 @Autowired  
 private EmployeeListPage employeeListPage ;  
  
 @Autowired  
 private LoginPage loginPage;  
  
 @Test  
 void contextLoads() {  
 *//3. Navigate to a specific URL* driver.navigate().to("http://eaapp.somee.com");  
  
 *//POM Code* homePage.clickLogin();  
 loginPage.performLogin("admin", "password");  
 homePage.clickEmployeeList();  
 employeeListPage.clickCreateNew();  
 createEmployeePage.createNewEmployee("AutoUser2", "22222", "autotestuser2@gmail.com", "20000", "Middle");  
  
  
 }

**Using @PostConstruct to initialize the pageFactory for pageObectModel**

We have this code in each of the pom

public CreateEmployeePage(WebDriver driver){  
 this.driver = driver;  
 PageFactory.*initElements*(driver, this);  
}

To remove this create a new Base file

public class BasePage {  
 public BasePage(WebDriver driver){  
 PageFactory.*initElements*(driver, this);  
 }  
}

And then use this in the Pages

public class HomePage extends BasePage{  
 private WebDriver driver;  
 public HomePage(WebDriver driver) {  
 super(driver);  
 }

**There’s another way to above thing by using @PostConstructor :**

The PostConstruct annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization. It will be called only once.

public class BasePage {

@Autowired

Private Webdriver driver;

@PostConstruct  
 public void init(){  
 PageFactory.*initElements*(driver, this);  
 }  
}

And inside the pages of POM just extend this class and remove the constructor.

**Using @Profile – Running test in different environemts/browsers**

Create different profiles in resources

application-dev.properties

application-qa.properties

application-prod.properties

To set the environment either we can use the **@ActiveProfile(“dev”)** or we can provide the **spring.profiles.active=dev** when running the tests

To Run the tests in Testng –

* Add testing dependency in the pom.xml
* Extend the tests class with **AbstractTestNGSPringContextTests** – this is getting extended to support the spring to run testng tests