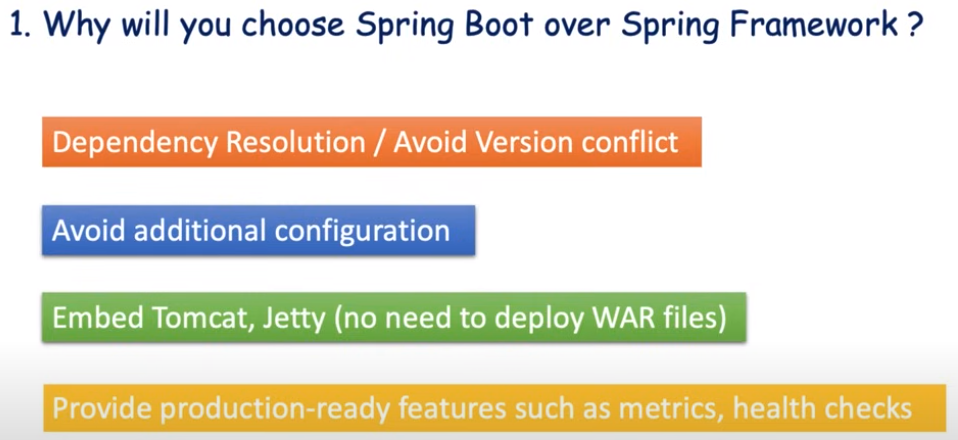
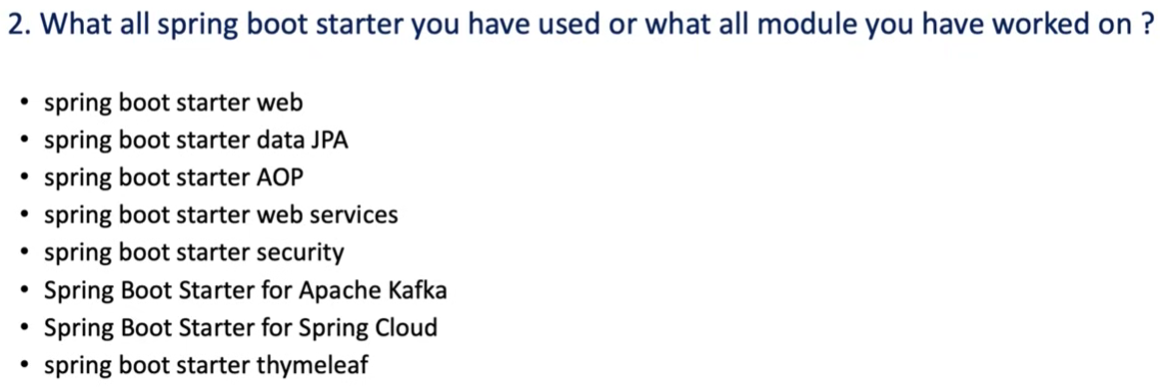
Spring Boot Interview Questions





<https://spring.io/projects/spring-data>



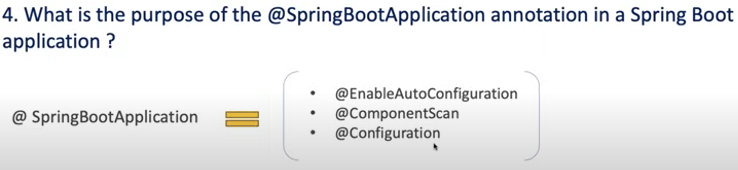
1 . we can run springboot application main class.

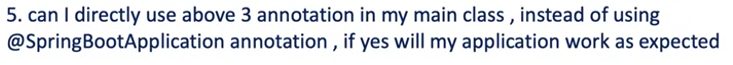
2 . mvn spring-boot::run ( maven plugin )

It will build any jar / war, application will run from target/classes folder , trigger the main class

3 . build jar by using mvn install

Java -jar <jar-path>





Yes



Keep “debug=true” in application.properties file

Positive matches & Negative matches



Springboot main class

Method 1

Example:

@Springbootapplication(exclude={DataSourceAutoConfiguration.class,AopAutoConfiguration.class})

Method 2

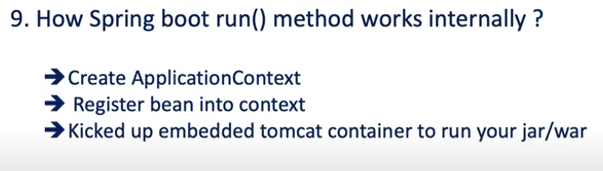
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.aop. AopAutoConfiguration,…



Byusing application.yml / application properties.

By default server port is 8080

If we want to override, specify in .yml / .properties







Main springboot application need to implement commandlinerunner funcational interface

Then implement run()

First execute springboot application run() method then command line runner run()





@Component ( interface ) is the parent stero type annotation.

Remaining are inherited intefaces from @Component

Inorder to understand purpose of the class we will use these stero type annotations.

@RestController – Endpoint expose / web related code will be there.

@Service – All business logic

@Repository – all data base connection & related operations logic will be there

@component – external resource to consume, client class, mapper class,

## import org.springframework.stereotype.Component;

## @Component

## public class MyComponent {

## public void doSomething() {

## System.out.println("Doing something in MyComponent");

## }

## }

In this example, **MyComponent** is a simple class annotated with **@Component**. This annotation tells Spring to automatically detect and register this class as a Spring bean during component scanning.

Now, let's create a Spring Boot application that uses this component:

## import org.springframework.boot.SpringApplication;

## import org.springframework.boot.autoconfigure.SpringBootApplication;

## import org.springframework.context.ApplicationContext;

## @SpringBootApplication

## public class MySpringBootApplication {

## public static void main(String[] args) {

## ApplicationContext context = SpringApplication.run(MySpringBootApplication.class, args);

## // Retrieve the bean and use it

## MyComponent myComponent = context.getBean(MyComponent.class);

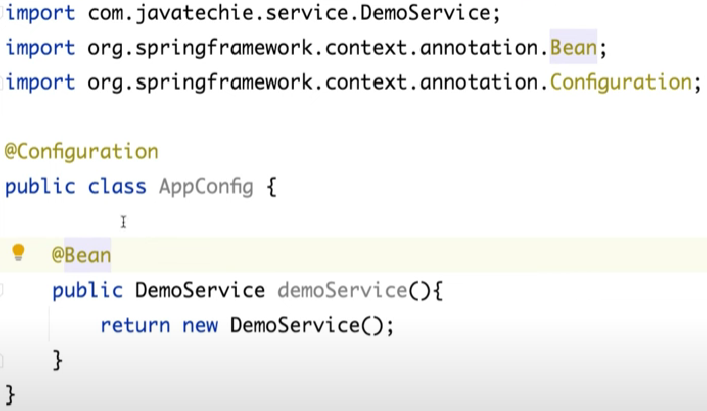
## myComponent.doSomething();

## }

## }

We can also use intechangablly, these annotation, means inplace of repository, we can use service but the purpose of these annotation is easily identify the purpose of the class by using these annotations. Spring will create bean for us.

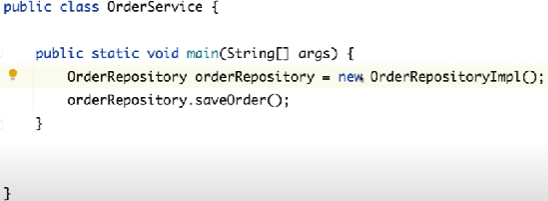


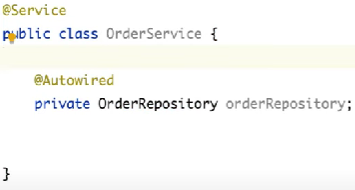


By using stero type annotations spring will create bean for us, apart from that we will configure additional configuration @Configuration will tell spring to load the configuration and create a bean of Demoservice class.



Dependency injection is a design pattern, and is used to achieve loose coupling between classes

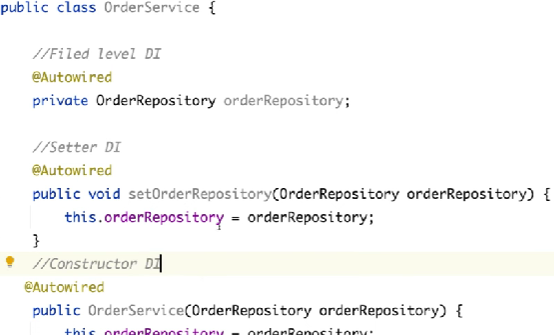


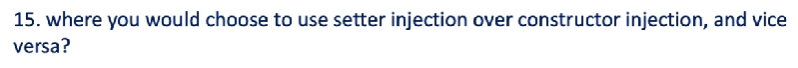


The dependency bean can be injected by spring framework , object creation to destruction , total life cycle of the bean will be managed by spring frame work.

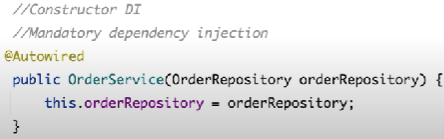
Dependency injection is the Backbone of spring framework

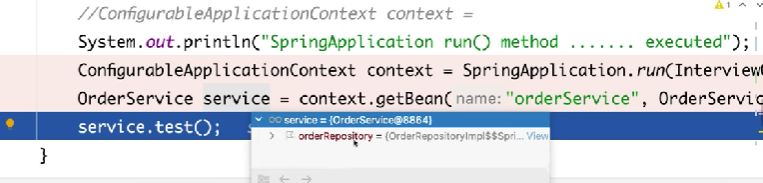


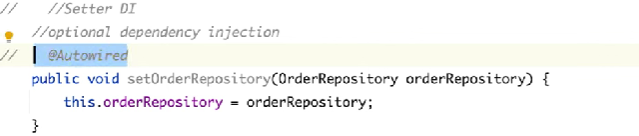




If dependency injection is Mandatory then constructor injection. Means in our example while creating order service , order repository bean creation is mandatorily then we will go for constructor DI



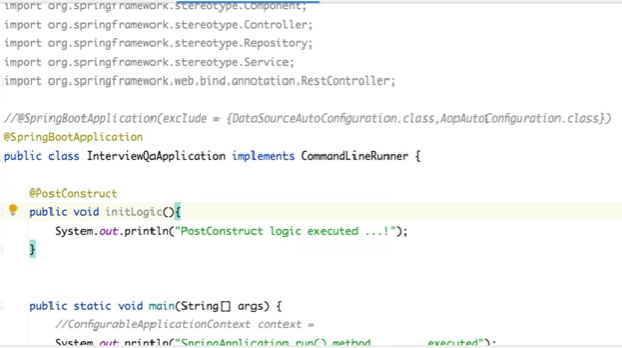




|  |  |
| --- | --- |
| **Constructor DI** | **Setter DI** |
| Mandatory DI | Optional DI |
| Immutable in nature | Not immutable in nature |
| Circular dependency can’t resolve | Circular dependency can be resolved |
|  |  |



Preprocessing login can be executed by application startup using command line runner and @post construct also



Order of execution

1 springboot run method

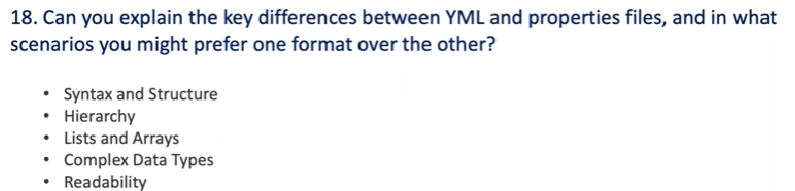
2 @postconstruct method

3 commandlinerunner run method









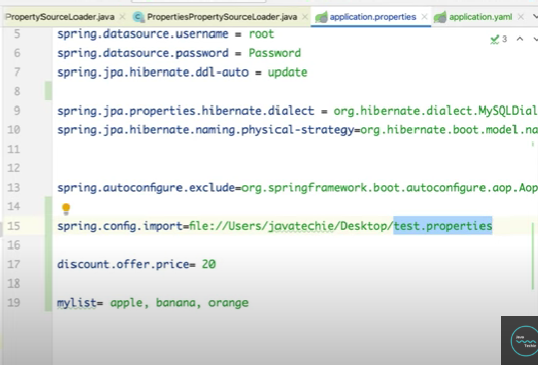


Same, some old windows os file extension will allow 3 chars as extension that’s yml, both are same structure and syntax.

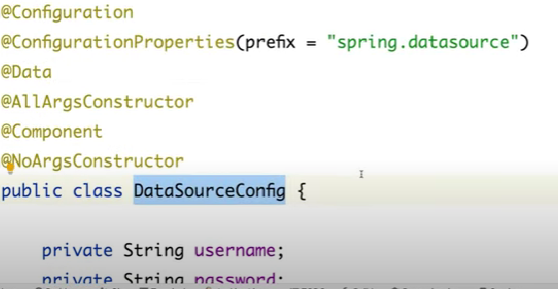


Property file first load then yml file, if same property is declared in both files then property file value will effect.

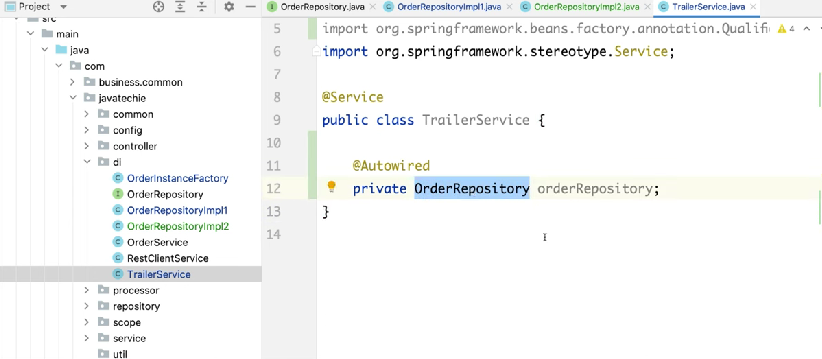


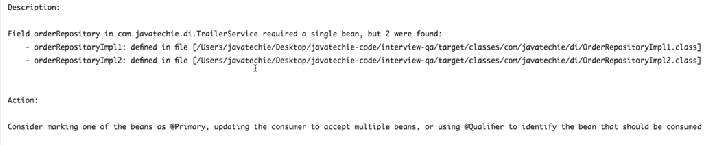




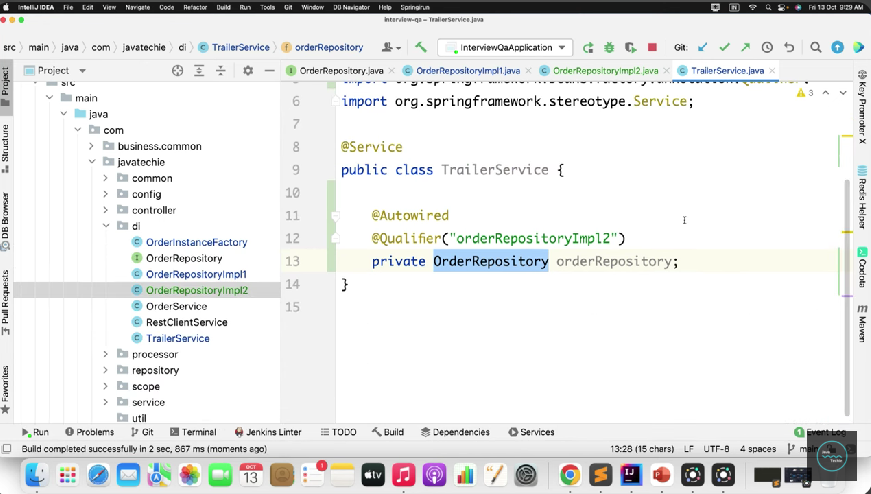




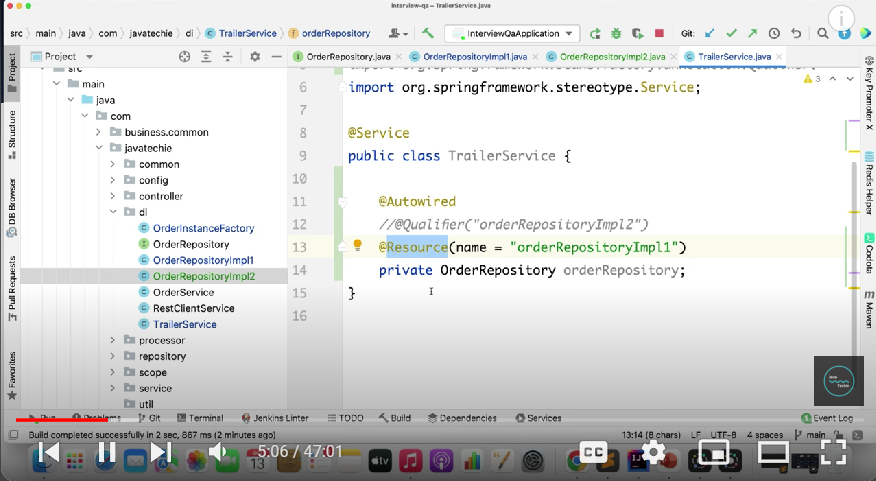


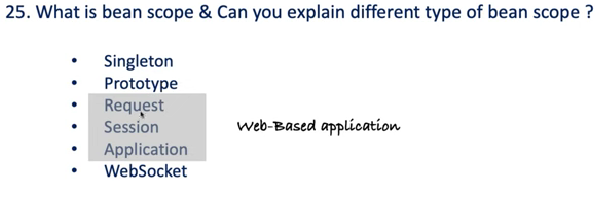


Solution @Qualifier(“<implementation class>”)



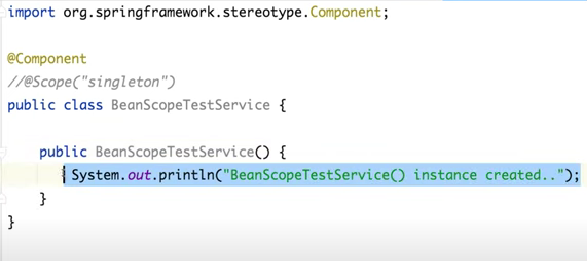


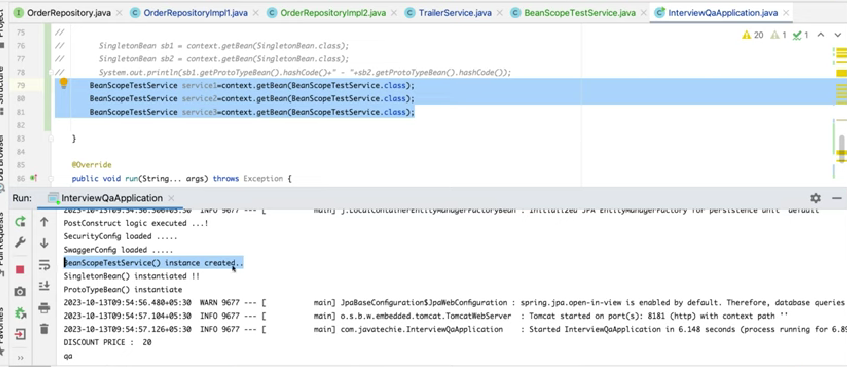




Single: ( default scope )





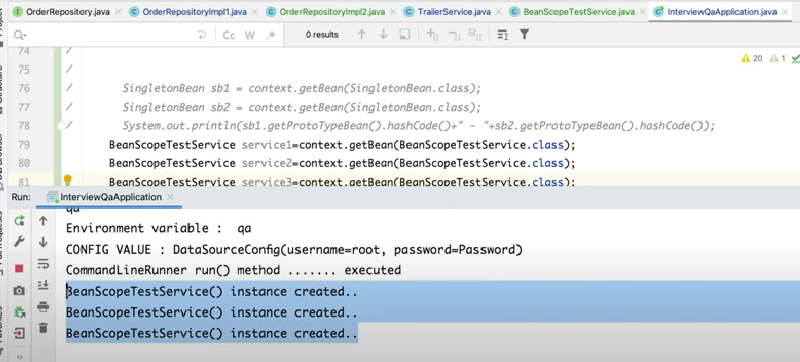


Only 1 time object created. Spring IOC container main only single instance.

**Prototype:**





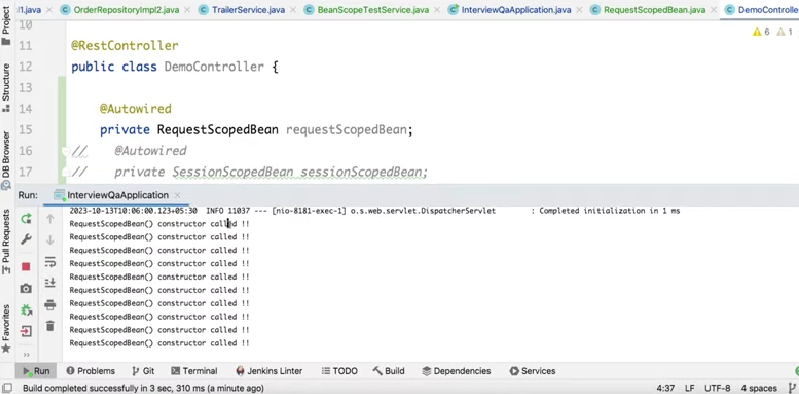


**Request: ( web application )**

****

****

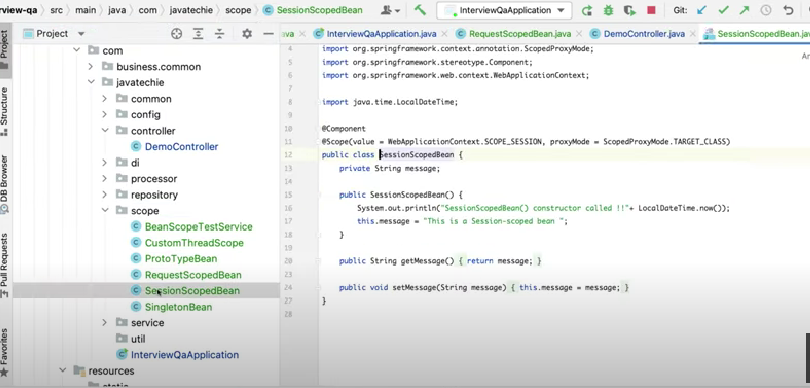
****

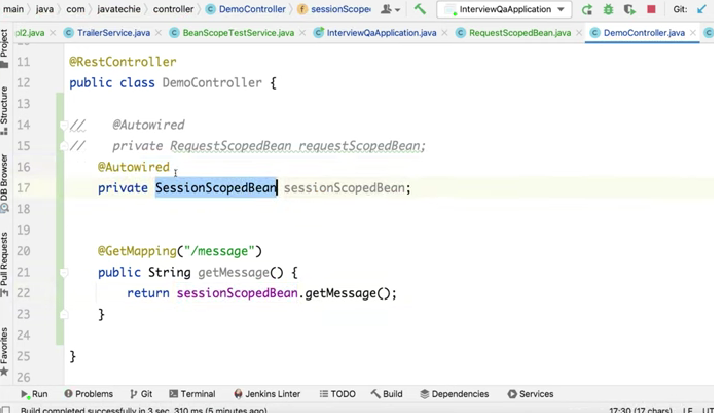
****

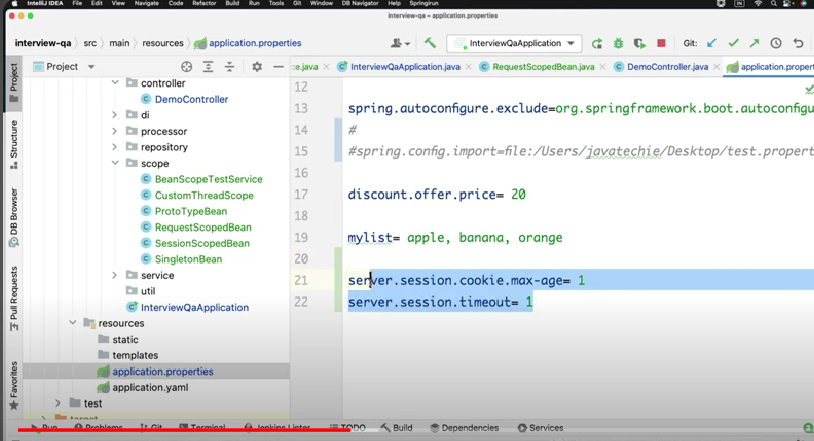
If we call the endpoint 10 times, bean was created 10times.

If we want to store the form data then we can use request scope.

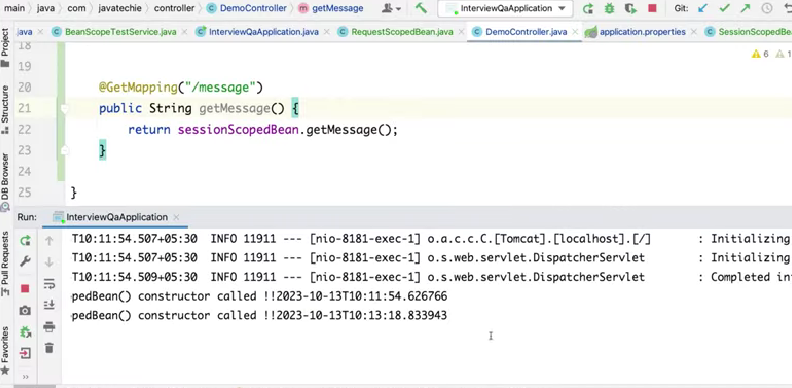
**Session : ( create per user session )**

****

****

****

**Inorder to invalidate session for 1 min**

****

User specific data need to store then we can use session store.

**Application: (web application)**

****

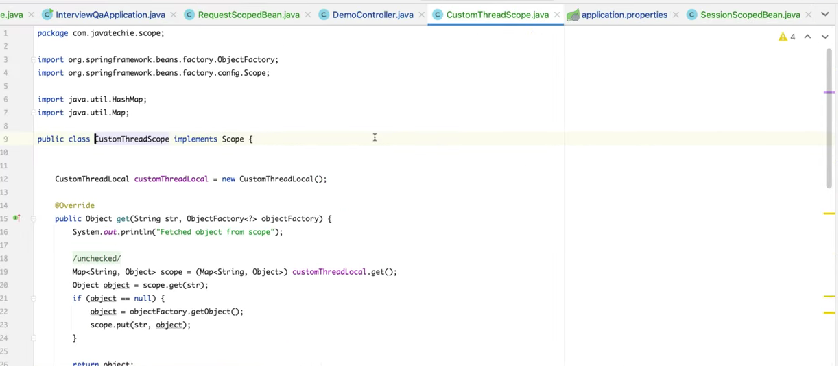
Application wise caching, shared resource the we can use application scope.

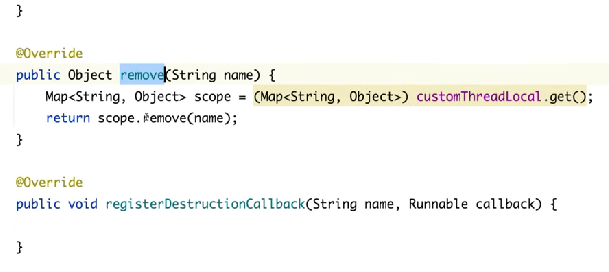
**WebSocket:**

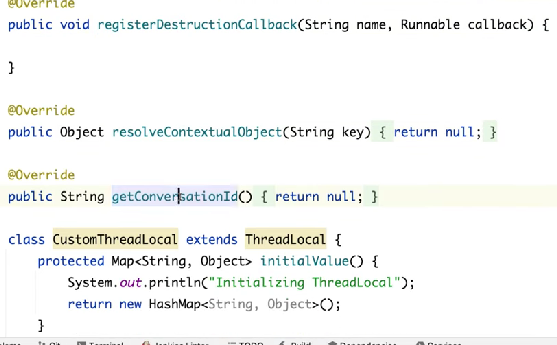
****

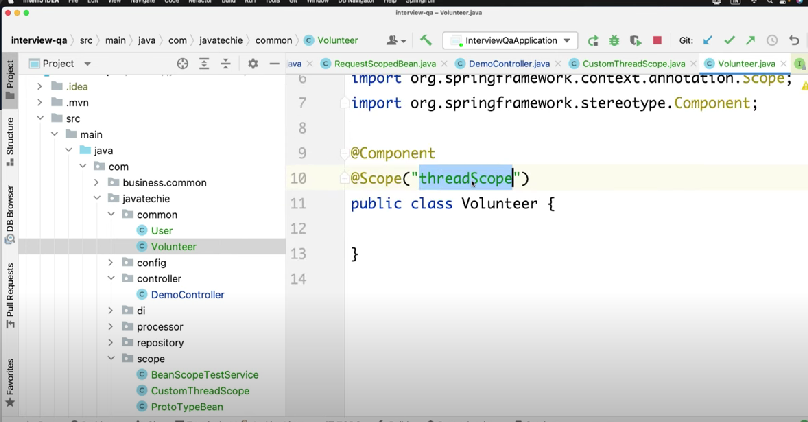
Specific to the WebSocket session.

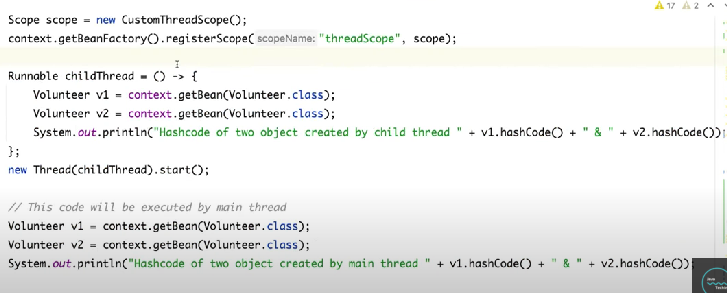
****

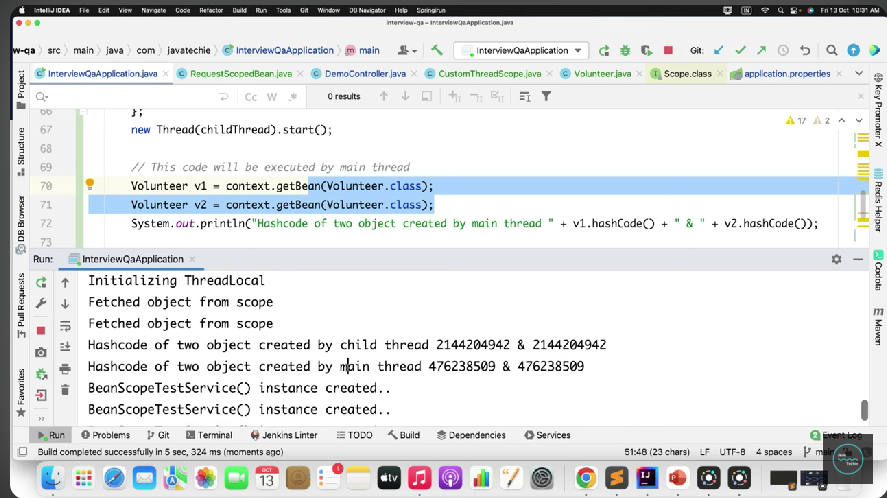
****

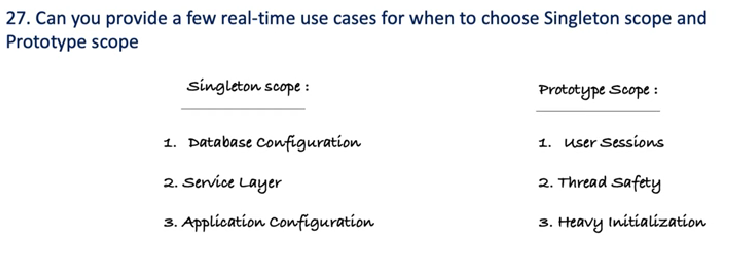
****

****

****

****

****

****

****

**Yes, it will lost the protype nature, it will give singleton nature, we will overcome by using following 3 methods.**

**In singleton bean , we will get the protype bean in 3 way with out losing it’s protype nature.**

**1 ) @Autowired**

**ApplicationContext context;**

**Public ProtoTypeBean getProtoTypeBean(){**

**return context.getBean(ProtoTypeBean.class)**

**}**

**2 ) @Autowired**

**Private ObjectFactory<ProtoTypeBean> protoTypeBeanObjectFactory;**

**Public ProtoTypeBean getProtoTypeBean(){**

**return protoTypeBeanObjectFactory.getObject();**

**}**

**3 )**

**@Lookup**

**Public ProtoTypeBean getInstance(){**

**Return null;**

**}**

**Public ProtoTypeBean getProtoTypeBean(){**

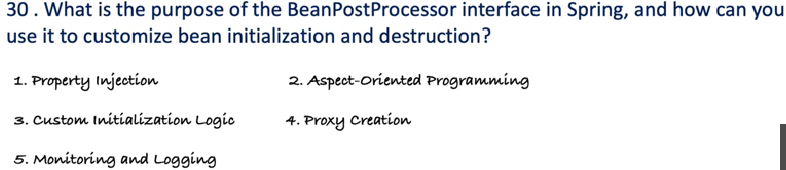
**return getInstance()**

**}**

****

**Spring Singleton will be the spring application context level single object will be maintained.**

**Plain singleton means, JVM level single object will be maintained.**

****