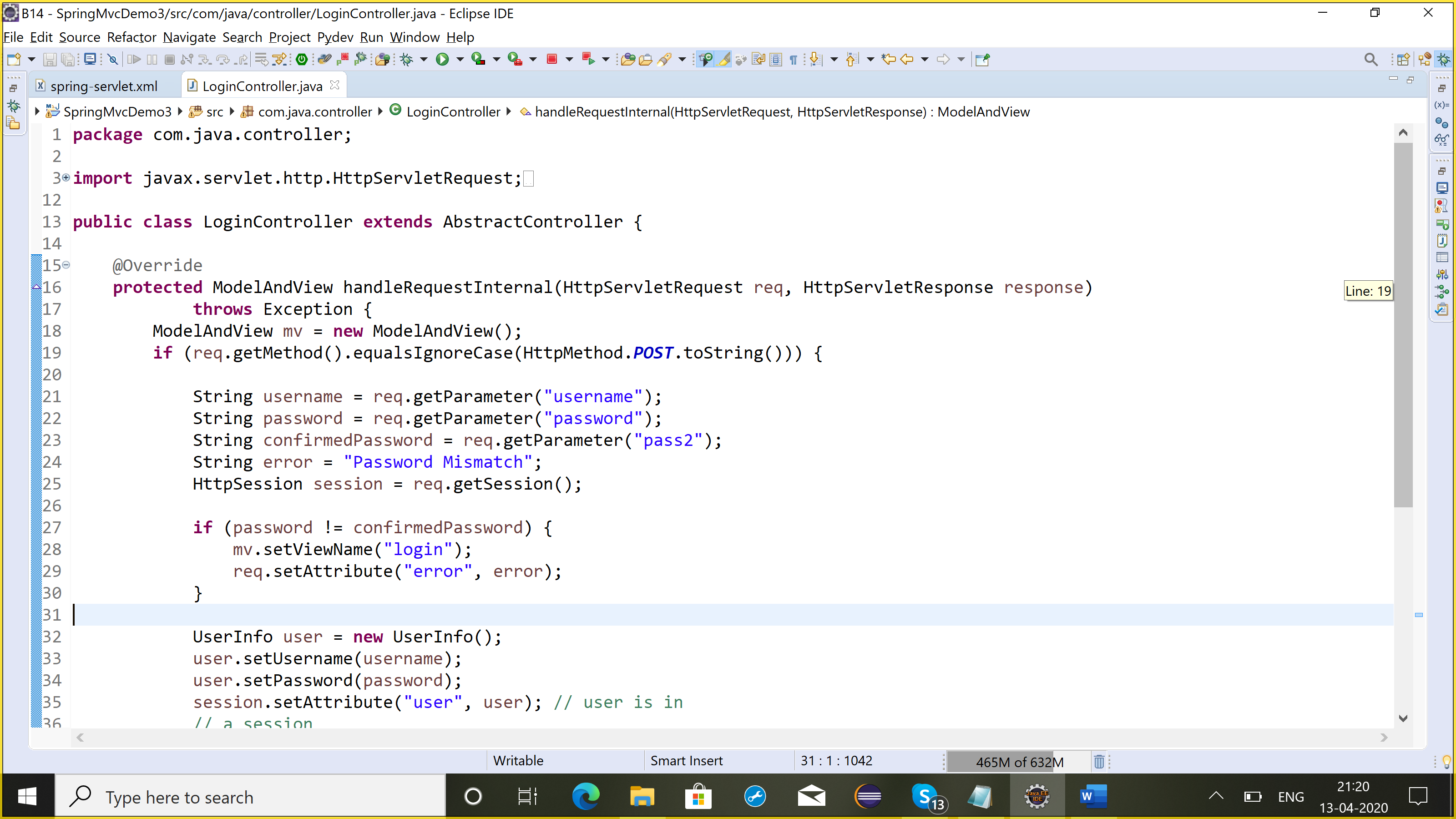
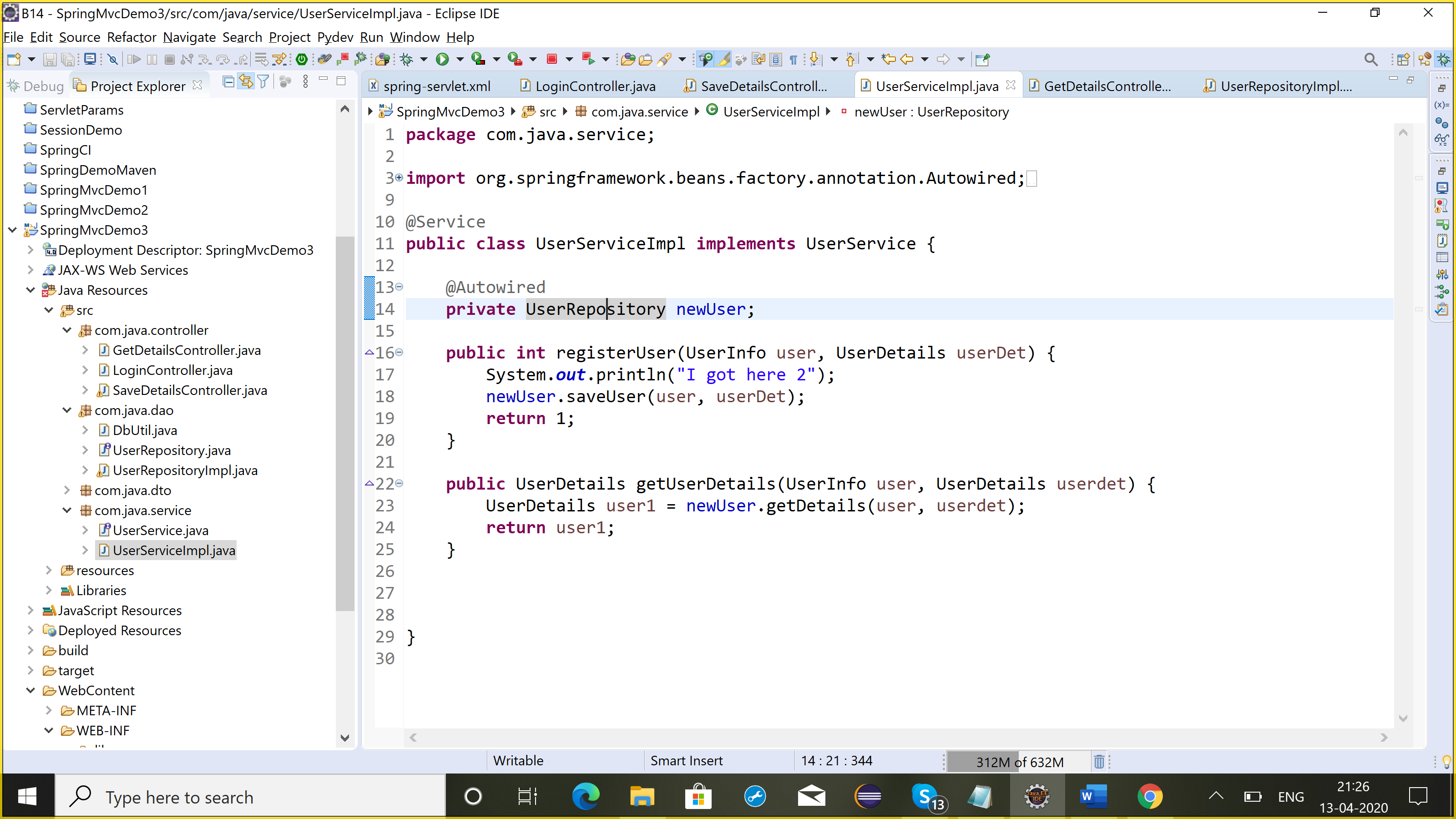
1. Instead of using **Controller** interface, you can also use **AbstractController**.



Autowired does not work with static variables



<context:component-scan base-package=*"com.java"*></context:component-scan>

This enables ur stereotype annotations

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

These annotation tell spring to create the objects of these classes

Session, SessionFactory configured as bean, means registered with spring. Lifecycle managed by spring container.

**public** **class** DbUtil {

**static** SessionFactory *sf*= **null**;

**static** {

Configuration cfg= **new** Configuration();

cfg.configure("hibernate-config.xml");

*sf*= cfg.buildSessionFactory();

}

**public** **static** Session getSession() {

**return** *sf*.openSession();

}

}

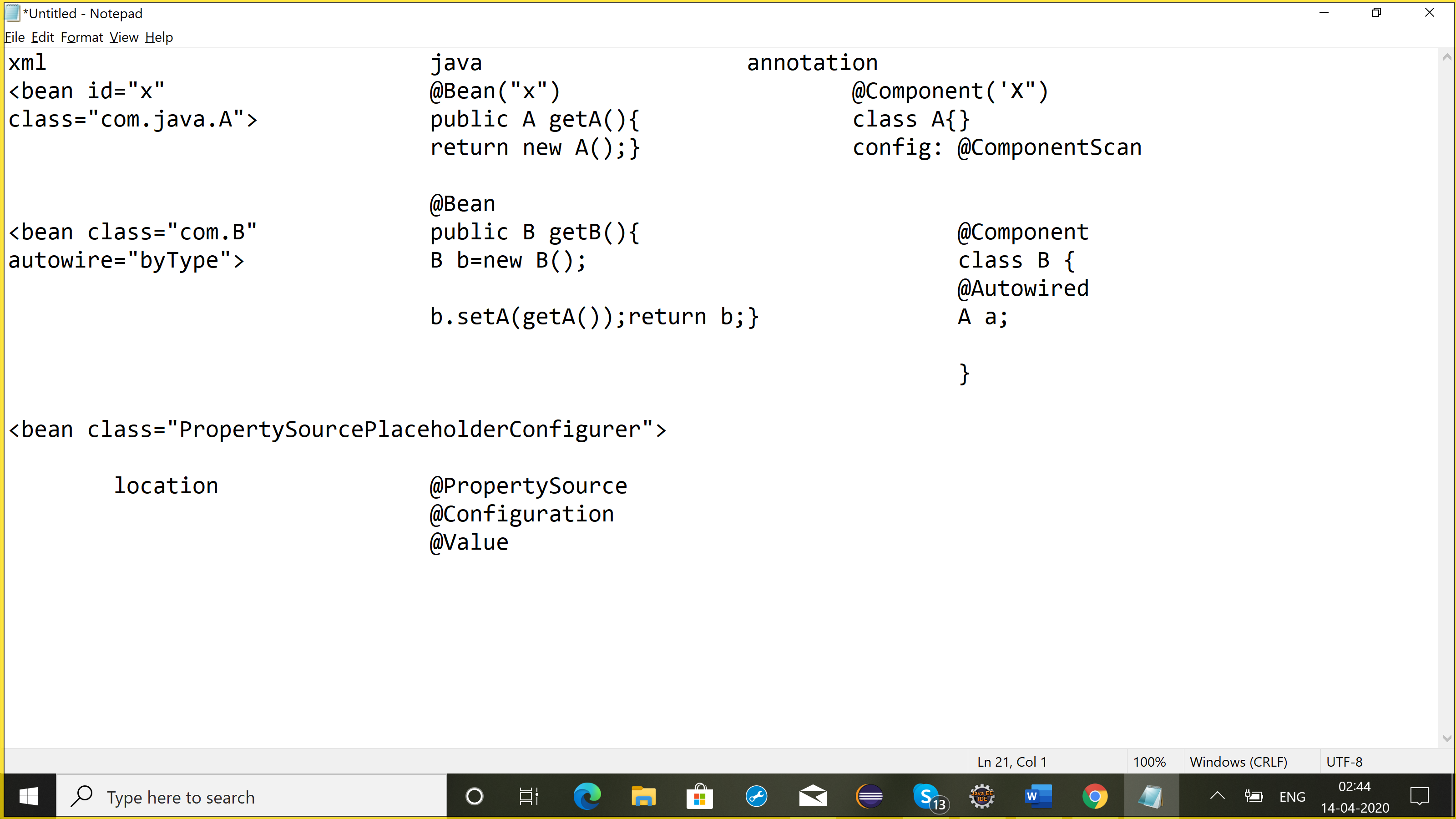
<bean class=*"com.java.dao.DbUtil"* factory-method=*"getSession"* scope=*"prototype"*/>

But sessionFactory is getting created by you.

**Integrate with hibernate:**

**Spring-orm**

Spring has provided you a class: LocalSessionFactoryBean, it can give you a sessionFactory object registered with spring context



Integrating with hibernate

1. Spring-orm : module which integrates spring +orm (jpa/ hibernate)
2. LocalSessionFactoryBean

Datasource: url, username, password: DriverManagerDataSource

hibernateProperties

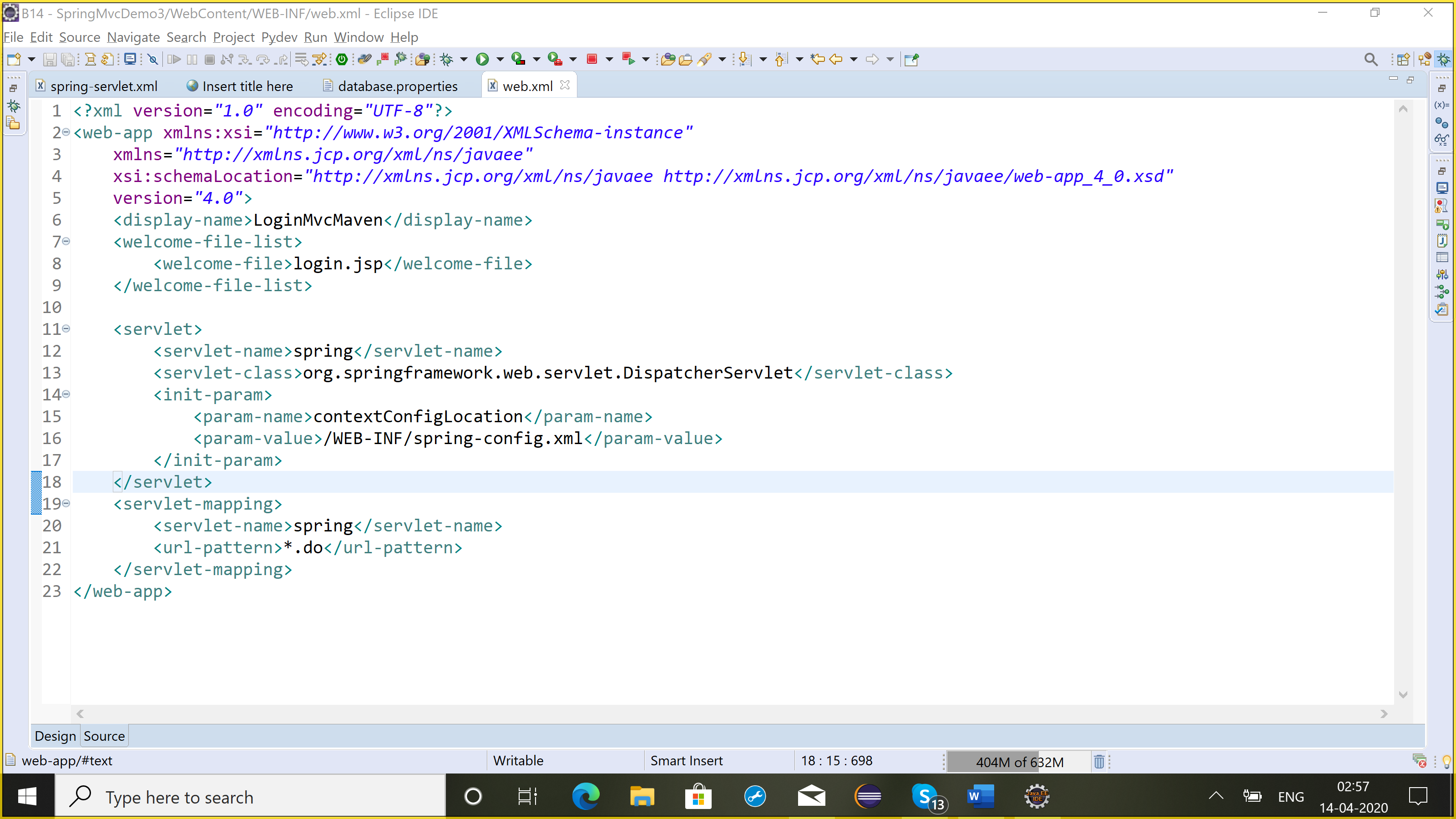
mapping class

bean.afterPropertiesSet(); //get connections to db, create schema

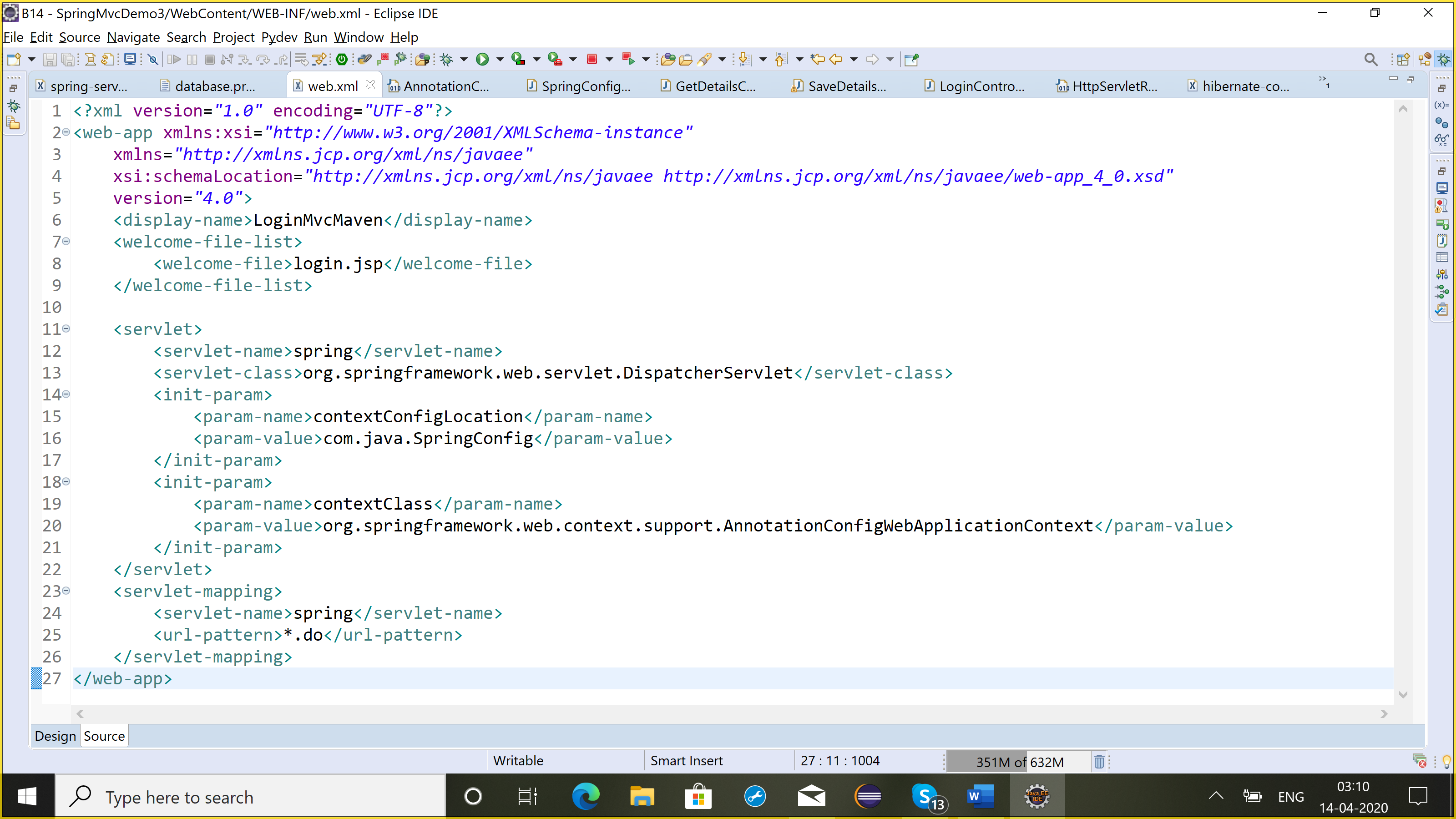
bean.getObject(): sessionFactory

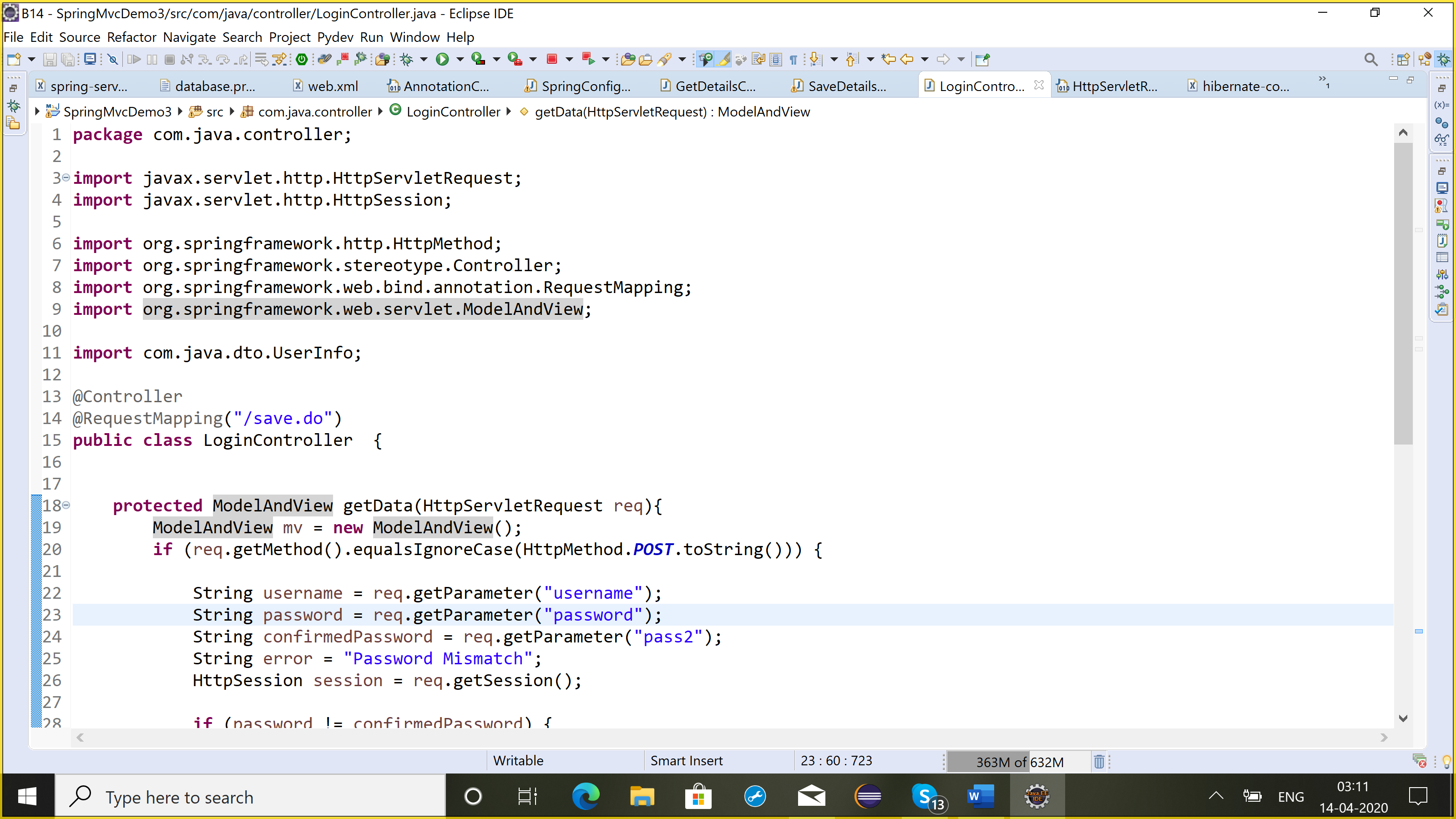
1. **If I want to give my config file the name as spring-config.xml?**

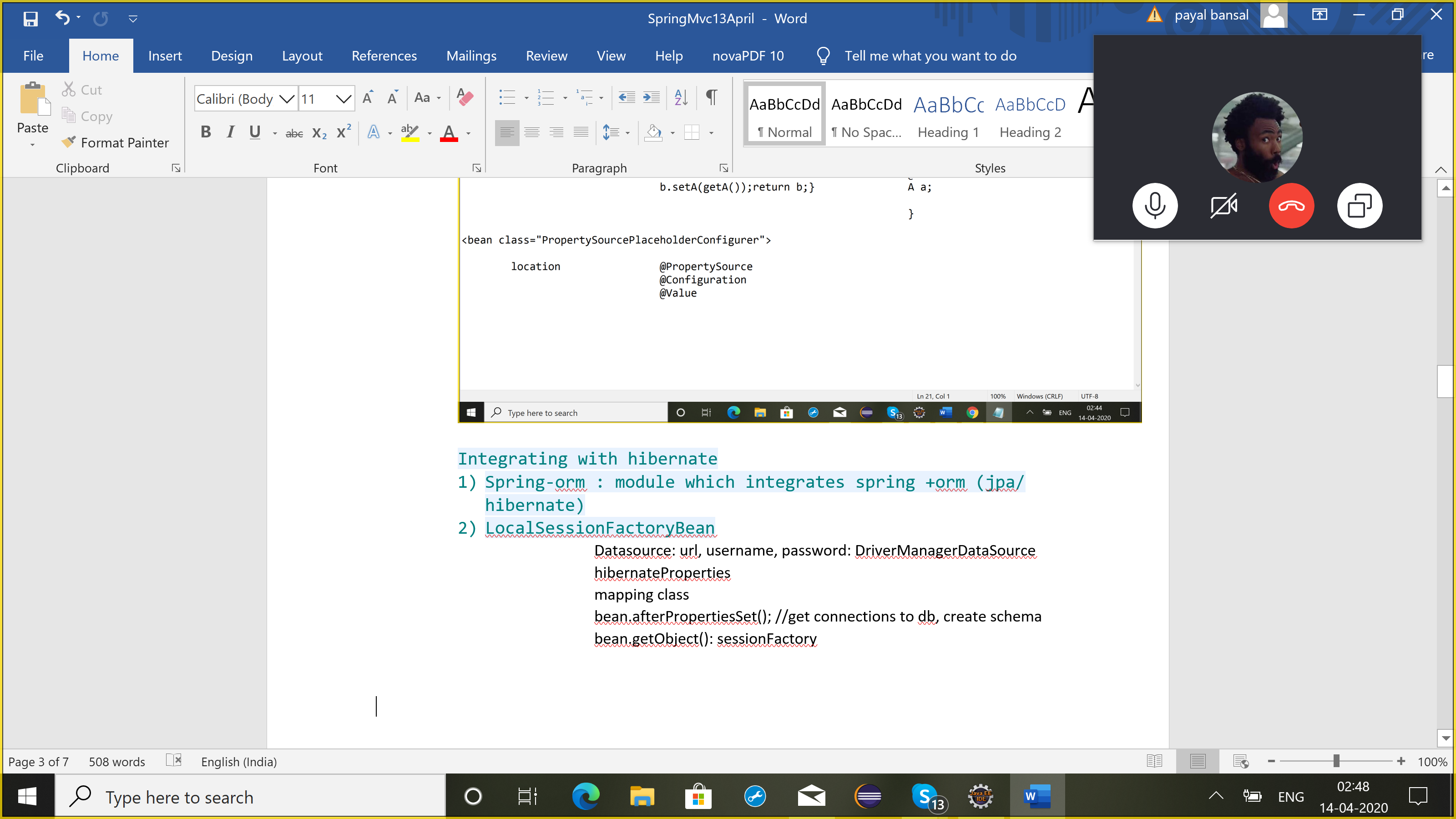
contextConfigLocation: filename



If I don’t have xml configuration but java configuration??







15 April //TODO

**public** **class** **MyWebApplicationInitializer** **implements** **WebApplicationInitializer** {

**@Override**

**public** **void** **onStartup**(ServletContext servletCxt) {

*// Load Spring web application configuration*

AnnotationConfigWebApplicationContext ac = **new** AnnotationConfigWebApplicationContext();

ac.register(AppConfig.class);

ac.refresh();

*// Create and register the DispatcherServlet*

DispatcherServlet servlet = **new** DispatcherServlet(ac);

ServletRegistration.Dynamic registration = servletCxt.addServlet("app", servlet);

registration.setLoadOnStartup(1);

registration.addMapping("/app/\*");

}

}

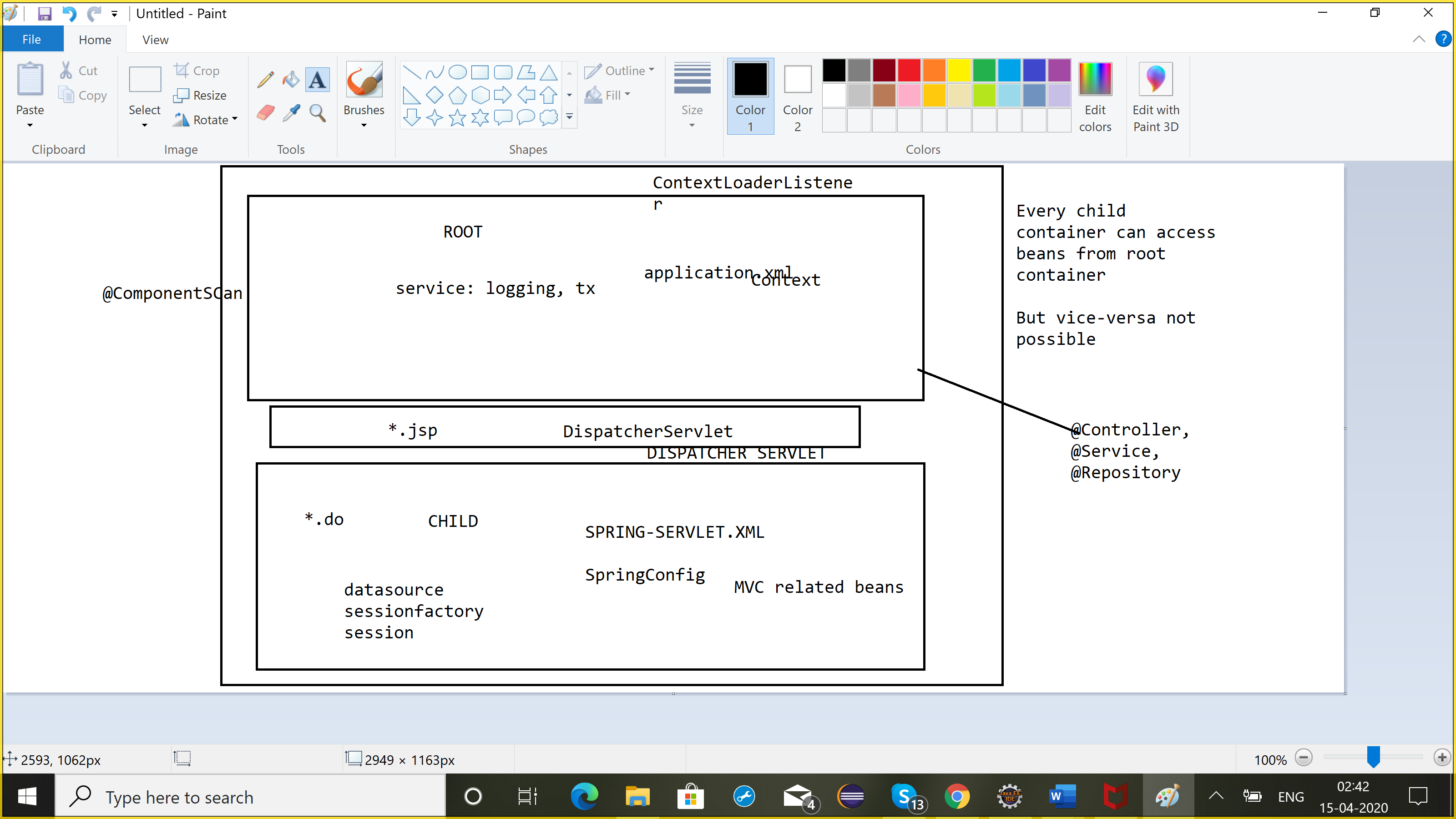
Spring web :

application.xml

Root container: ContextLoaderListener, child container (dispatcher)

Root container

Web.xml: dispatcherServlet: child container: SpringConfig



Bydefault ContextLoaderListener will try to load beans from applicationContext.xml

To change the config name:

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>springconfig.xml</param-value>

</context-param>

If u want root to read ur java config:

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.java.SpringConfig2</param-value>

</context-param>

<context-param>

<param-name>contextClass</param-name>

<param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>

</context-param>

<web-app>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/app-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>app</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value></param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>app</servlet-name>

<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

</web-app>

**public** **class** **MyWebAppInitializer** **extends** **AbstractAnnotationConfigDispatcherServletInitializer** {

**@Override**

**protected** Class<?>[] getRootConfigClasses() {

**return** **new** Class<?>[] { RootConfig.class };

}

**@Override**

**protected** Class<?>[] getServletConfigClasses() {

**return** **new** Class<?>[] { App1Config.class };

}

**@Override**

**protected** String[] getServletMappings() {

**return** **new** String[] { "/app1/\*" };

}

}

<web-app>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/root-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>app1</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/app1-context.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>app1</servlet-name>

<url-pattern>/app1/\*</url-pattern>

</servlet-mapping>

</web-app>

If an application context hierarchy is not required, applications may configure a “root” context only and leave the contextConfigLocation Servlet parameter empty.

**import** org.springframework.web.WebApplicationInitializer;

**public** **class** **MyWebApplicationInitializer** **implements** **WebApplicationInitializer** {

**@Override**

**public** **void** **onStartup**(ServletContext container) {

XmlWebApplicationContext appContext = **new** XmlWebApplicationContext();

appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

ServletRegistration.Dynamic registration = container.addServlet("dispatcher", **new** DispatcherServlet(appContext));

registration.setLoadOnStartup(1);

registration.addMapping("/");

}

}

 An abstract base class implementation of WebApplicationInitializer named AbstractDispatcherServletInitializer makes it even easier to register the DispatcherServlet by overriding methods to specify the servlet mapping and the location of the DispatcherServlet configuration.

**public** **class** **MyWebAppInitializer** **extends** **AbstractAnnotationConfigDispatcherServletInitializer** {

**@Override**

**protected** Class<?>[] getRootConfigClasses() {

**return** **null**;

}

**@Override**

**protected** Class<?>[] getServletConfigClasses() {

**return** **new** Class<?>[] { MyWebConfig.class };

}

**@Override**

**protected** String[] getServletMappings() {

**return** **new** String[] { "/" };

}

}

If you use XML-based Spring configuration, you should extend directly from AbstractDispatcherServletInitializer, as the following example shows:

Java

Kotlin

**public** **class** **MyWebAppInitializer** **extends** **AbstractDispatcherServletInitializer** {

**@Override**

**protected** WebApplicationContext **createRootApplicationContext**() {

**return** **null**;

}

**@Override**

**protected** WebApplicationContext **createServletApplicationContext**() {

XmlWebApplicationContext cxt = **new** XmlWebApplicationContext();

cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

**return** cxt;

}

**@Override**

**protected** String[] getServletMappings() {

**return** **new** String[] { "/" };

}

}

AbstractDispatcherServletInitializer also provides a convenient way to add Filter instances and have them be automatically mapped to the DispatcherServlet,

**public** **class** **MyWebAppInitializer** **extends** **AbstractDispatcherServletInitializer** {

*// ...*

**@Override**

**protected** Filter[] getServletFilters() {

**return** **new** Filter[] {

**new** HiddenHttpMethodFilter(), **new** CharacterEncodingFilter() };

}

}

Filter in spring mvc

Logging Filter

PerformanceFilter: time : request

Filter: servlet context (web.xml)

Filter cannot work on the lifecycle of spring beans