# SPRİNG NOTLARI

# Properties Dosyaları ile Çalışmak

# 1.Sol kısımda resources klasörünün içine database.properties adında bir file oluşturuyoruz.Ve aşağıdaki koddaki gibi key-value ilişkisi ile dolduruyoruz.

**db\_url**=jdbc:mysql://localhost:3306/project?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC

**db\_password**=1234

# db\_username=root

# 2.Daha sonra bunu classlarda kullanabilmek için aşağıdaki gibi kodluyoruz.

Properties **pro** = **new** Properties();

**try** {

InputStream **inputStream** = **new** FileInputStream ("src/main/resources/database. properties");

pro.load(inputStream);

String *url* = pro.getProperty("**db\_url**");

String *password* = pro.getProperty("**db\_password**");

String *username* = pro.getProperty("**db\_username**");

pro.setProperty(“key”,”value”); // Şu şekilde Properties filesine kodlardan da ekleme yapabiliriz.

} **catch** (IOException **e**) {

*logger*.warn("Input Stream Çalışmadı" + e);

# }

# IoC (Inversion of Control) ve Dependency Injection

Dependency İnjection Spring’in en önemli parçalarından biridir.Aslında bir tasarım desenidir.Bu tasarım deseninde ise IoC’yi uygularız. **Inversion of control** ise bir yazılım tasarım prensibidir. **IoC**ile Uygulama içerisindeki obje instance(instanceOf-bir nesnenin bir türe ait olup olmadığını anlama(polymorphizm))’larının yönetimi sağlanarak, bağımlılıklarını en aza indirgemek amaçlanmaktadır.

ÖRNEK KOD:Bu kodda Oracle müşterileri ile Mysql Müşterilerini ICustomerDal classından implement ettik.Ve firmamıza artık postgresql kullanan bir müşteri gelirse kodları bozmadan sadece postgreCustomerDal classı oluşturup IcustomerDalı implement ederek yeni müşteriyi ekleyebiliriz.

**public** **interface** ICustomerDal {

**void** **add**();

**void** **remove**();

}

**public** **class** mySqlCustomerDal **implements** ICustomerDal{

**public** **void** **add**() {

System.***out***.println("Mysql Eklendi");

}

**public** **void** **remove**() {

System.***out***.println("Mysql Silindi");

}

}

**public** **class** oracleCustomerDal **implements** ICustomerDal{

**public** **void** **add**() {

System.***out***.println("Oracle Eklendi");

}

**public** **void** **remove**() {

System.***out***.println("Oracle Silindi");

}

}

**public** **class** CustomerManager {

ICustomerDal ICustomerDal;

**public** **CustomerManager** (ICustomerDal ICustomerDal) {

**this**.ICustomerDal = ICustomerDal;

}

**public** **void** **add**() {

ICustomerDal.add();

}

**public** **void** **remove**() {

ICustomerDal.remove();

}

}

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

CustomerManager **cM** = **new** CustomerManager(**new** oracleCustomerDal());

cM.add();

cM.remove();

}

**XML’den IoC Yapılandırması**

applicationContext.xml

<bean id ="mySqlDatabase" class="spring\_introo.mySqlCustomerDal">

</bean>

<bean id ="oracleDatabase" class="spring\_introo.oracleCustomerDal">

</bean>

main class

ClassPathXmlApplicationContext **context** = **new** ClassPathXmlApplicationContext ("applicationContext.xml"); //XML DOSYASINI BURADA VERİYORUZ.

CustomerManager **cM** = **new** CustomerManager(context.getBean ("mySqlDatabase", ICustomerDal.**class**));

cM.add();

cM.remove();

CustomerManager **cM1** = **new** CustomerManager(context.getBean ("oracleDatabase", ICustomerDal.**class**));

cM1.add();

cM1.remove();

**AppConfig ile**

AppConfig class

@Configuration

public class AppConfig {

@Bean

public IDbHelper MySqlDal() {

IDbHelper mySqlDal = new MySqlDal("mysqlURL","mysqlUsername","mysqlPassword");

return mySqlDal;

}

@Bean

public IDbHelper OracleDal() {

IDbHelper oracleDal =new OracleDal("oracleURL","oracleUsername","oraclePassword");

return oracleDal;

}

@Bean

public DbConnection DbConnection() {

DbConnection connection = new DbConnection(MySqlDal());

return connection; } }

main class

BeanFactory factory= new AnnotationConfigApplicationContext(AppConfig.class);

DbConnection connectionn = factory.getBean("dbConnection",DbConnection.class);

connection.getConnection();

connection.closeConnection();

**Constructor Arg İnjection(Parametreli bean oluşturma)**

Yukarıdaki kodlara ek ICustomerManager interfacesi oluşturduk.

**public** **interface** ICustomerManager {

**void** **add**();

**void** **remove**();

}

//Application Context//

Ref ile Constructor

<bean id ="mySqlDatabase" class="spring\_introo.mySqlCustomerDal">

</bean>

<bean id ="customerManager" class = "spring\_introo.CustomerManager">

<constructor-arg ref="mySqlDatabase"/>

Property ile Constructor

<bean id ="mySqlDatabase" class="spring\_introo.mySqlCustomerDal">

</bean>

<bean id ="customerManager" class = "spring\_introo.CustomerManager">

<constructor-arg name="database" value="mySqlDatabase"></constructor-arg>

</bean>

//Main Class//

//Constructor Arg kullandığımız nesne

ICustomerManager **cm2** = context.getBean("customerManager",ICustomerManager.**class**);

cm2.add();

cm2.remove();

}

**Class Config ile**

IoCConfig Classı

@Configuration

**public** **class** IoCConfig {

@Bean

**public** ICustomerDal **mySqlCustomerDal**() {

**return** **new** mySqlCustomerDal();

}

@Bean

**public** ICustomerManager **managerMySql**() {

**return** **new** CustomerManager(mySqlCustomerDal());

}

}

**Setter İnjection**

CustomerManager classının içine set metodu koyduk.

CustomerManager Class

**public** **void** **setCustomerDal**(ICustomerDal customerDal) {

**this**.customerDal = customerDal;

}

Application Context

<bean id ="mySqlDatabase" class="spring\_introo.mySqlCustomerDal">

</bean>

<bean id =" oracleDatabase " class="spring\_introo.mySqlCustomerDal">

</bean>

<bean id ="customerManager3" class = "spring\_introo.CustomerManager">

<property name="customerDal" ref="mySqlDatabase"></property>

</bean>

<bean id ="customerManager4" class = "spring\_introo.CustomerManager">

<property name="customerDal" ref="oracleDatabase"></property>

</bean>

Main Class

ICustomerManager **cm4** = context.getBean("customerManager4",ICustomerManager.**class**);

cm4.add();

cm4.remove();

ICustomerManager **cm5** =context.getBean("customerManager3",ICustomerManager.**class**);

cm5.add();

cm5.remove();

**Metinsel Değerler ile İnjection**

Diyelim ki applicationContext’te bir nesne oluşturucaz(oracleCustomerDal yada mysqlCustomerDal) ve bunun içinde ki String connetion’a yani metinsel değerede metini applicationContext.xml’den almak istiyoruz.Şu şekilde yapacağız: MySqlCustomerDal ve OracleCustomerDal classlarının içine String bir connection yani bir metinsel değer koyduk ve get set metodlarını ekledik.

String connection;

**public** String **getConnection**() {

**return** connection;

}

**public** **void** **setConnection**(String connection) {

**this**.connection = connection;

}

ApplicationContext.xml

<bean id = "metinseldeger" class = "spring\_introo.mySqlCustomerDal">

<property name="connection" value="mySql Bağlantısı Oluştu"></property>

</bean >

Yukarıda sadece nesneyi ve nesnenin içindeki connection metinsel değerini oluşturduk.Yani CustomerManager cm1 = new CustomerManager(); oluşturmadık henüz.

<bean id = "customerManager5" class = "spring\_introo.CustomerManager" >

<constructor-arg ref="metinseldeger" />

ICustomerManager **cm** =context.getBean("customerManager5",ICustomerManager.**class**);

cm.add();

cm.remove();

**İnner Bean ile İnjection**

<bean id = "DbConnection" class="com.project.db.DbConnection">

<property name="database">

<bean class= "com.project.db.Database">

<property name="url" value="mySqlDatabase"></property>

<property name="username" value="mySql"></property>

<property name="password" value = "11123"></property>

</bean>

</property>

</bean>

Üstteki Bean ile Alttaki bean arasında bir fark yok.

<bean id = "db" class= "com.project.db.Database">

<property name="url" value="mySqlDatabase"></property>

<property name="username" value="mySql"></property>

<property name="password" value = "11123"></property>

</bean>

<bean id = "DbConnection" class="com.project.db.DbConnection">

<property name="database" ref="db"></property>

</bean>

**List,Set ve Map ile İnjection**

**DBConnection Classı**

**public** **class** DBConnection {

**private** String username;

**private** String password;

**private** List<String> list;

**private** Map<String,String> maps;

**public** **void** **openConnection**() {

System.**out**.println("Username :" + **this**.username);

System.**out**.println("Password " + **this**.password);

System.**out**.println("Listedekiler:");

**for** (String **ozellik** : list) {

System.**out**.println(ozellik);

}

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

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

Set<String> **keys** = **this**.maps.keySet();

**for** (String **key** : keys) {

System.**out**.println(maps.get(key)); }

}

**public** **void** **closeConnection**() {

System.**out**.println("Sql Kapatıldı");

}

**public** String **getUsername**() {

**return** username;

}

**public** **void** **setUsername**(String username) {

**this**.username = username;

}

**public** String **getPassword**() {

**return** password;

}

**public** **void** **setPassword**(String password) {

**this**.password = password;

}

**public** List<String> **getList**() {

**return** list;

}

**public** **void** **setList**(List<String> list) {

**this**.list = list;

}

**public** Map<String, String> **getMaps**() {

**return** maps;

}

**public** **void** **setMaps**(Map<String, String> maps) {

**this**.maps = maps;

}

**ApplicationContext.xml’de List ve Maps Kullanımı**

<bean id = "dbConnection" class="com.project.database.DBConnection">

<property name="username" value = "SqlUSerName"></property>

<property name="password" value ="SqlPassword"></property>

<property name="list">

<list>

<value>Bağlantı1</value>

<value>Bağlantı2</value>

<value>Bağlantı3</value>

</list>

</property>

<property name="maps">

<map>

<entry key="bağlantilar1" value="5"></entry>

<entry key="bağlantilar2" value="10"></entry>

<entry key="bağlantilar3" value="100"></entry>

</map>

</property>

</bean>

**ÖNEMLİ:** Liste ref ile bean alırken şöyle kullanırız.

<bean id="BeanAuthor" class="com.project.model.Author">

<property name="name" value="JK Rowling"></property>

<property name="date" value="27.11.1980"></property>

<property name="books">

<list>

<ref bean="BeanBook1" />

<ref bean="BeanBook2" />

<ref bean="BeanBook3" />

</list>

</property>

</ bean >

**Main Class**

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

ConfigurableApplicationContext **configurableApplicationContext** = **new** ClassPathXmlApplicationContext("applicationContext.xml");

DBConnection **dbConnection** = (DBConnection) configurableApplicationContext.getBean("dbConnection");

dbConnection.openConnection();

dbConnection.closeConnection();

}

**AutoWiring konusu**

Bu konunun mantığı şöyledir.Diyelim ki DbConnection isminde bir classımız var.Ve bu classın içinde ise dbHelper türünden bir dbhelper adında property var;

public class DbConnection {

dbHelper dbhelper; }

İşte bu durumda applicationContext.xml dosyasının içinde dbConnection beanını oluştururken aşağıdaki gibi autowire=”byName” yazarsak eğer, applicationContext.xml içerisinde üstteki değişkenin adında(dbhelper) bir bean varsa bu ikisini bağlar.Ve artık dbConnection beanının altında property ref yada constructor ref vermemize gerek kalmaz.

<bean id="dbhelper" class="com.project.dalImpl.MySqlDal">

<property name="url" value="mysqlUrl"></property>

<property name="username" value="mysqlusername"></property>

<property name="password" value="mysqlpassword"></property>

</bean>

class="com.project.DbConnection.DbConnection" autowire="byName">

**Spring ile İnheritance(Kalıtım)**

Spring ile inheritance AppConfig class ile yapılmıyor.Sadece applicationContext.xml ile yapabiliriz.

<bean id ="db" class="com.project.db.Database" abstract="true">

//abstact true olmasının nedeni bunun beanının artık oluşturulamayacağını belirtiyor

<property name="username" value="UNornekXML"></property>

<property name="password" value="PasswordornekXML"></property>

</bean>

<bean id ="mySql" class="com.project.db.Database" parent="db">

<property name="url" value="Mysql"></property>

</bean>

<bean id ="oracle" class="com.project.db.Database" parent="db">

<property name="url" value="Oracle"></property>

</bean>

MAİN

Database **database** = (Database) context1.getBean("mySql");

DBConnection **connection2** = **new** DBConnection();

connection2.setDatabase(database);

connection2.openConnection();

connection2.closeConnection()

**İnheritance Merge Konusu(liste kalıtımı)**

Bu konunun mantığı şöyle diyelim ki abstract bir bean var ve bunu kalıtan başka bir class var.(üstteki örnek)Abstract beanda onu kalıtan beanlar için ortak olan bir list olabilir,hepsinin kullanabileceği.Ve kalıtan beanlarda bu liste kendi propertylerini ekleyebilir.Merge bu propertyleri eklemeye yarıyor.Alttaki örnekte daha iyi anlayacaksın.

<bean id ="db" class="com.project.db.Database" abstract ="true">

<property name="username" value="UNLornekXML"></property>

<property name="password" value="PasswordornekXML"></property>

<property name="liste">

<list> //(Abstract Class) Kalıtacak her classta ortak olan list.

<value>eleman1 </value>

<value>eleman2 </value>

<value>eleman3 </value>

</list>

</property>

</bean>

<bean id ="mySql" class="com.project.db.Database" parent="db"> //db kalıtıyor.

<property name="url" value="Mysql"></property>

<property name="liste">

<list merge="true"> // kalıttığı abstract classtaki ortak olan listide kalıtacağını onaylıyor.

<value>mySql elemanı</value>

</list>

</property>

</bean>

<bean id ="oracle" class="com.project.db.Database" parent="db"> //db kalıtıyor.

<property name="url" value="Oracle"></property>

<property name="liste">

<list merge="true"> // kalıttığı abstract classtaki ortak olan listide kalıtacağını onaylıyor.

<value>oracle elemanı</value>

</list>

</property>

</bean>

<bean id ="connection" class="com.project.db.DBConnection">

<property name="database" ref="oracle"></property>

</bean>

**Spring İnitialization and Destruction(Bean çalışmadan önce çalışacak metod ve spring close edilince çalışıcak metod)**

İnitialization mantığı şudur;Spring ile metodlar,beanlar çalışmadan önce başka bir metodu çalıştırırız.Örnekle şöyle Database e bağlanan bir connection classı var bu classı spring ile çağıracağız ama öncesinde initialization metodu çağırırız.Bu metodun içinde;url adresi doğru yazılmış mı username doğru mu yada bunlar null mu girilmiş gibi koşullar sağlanıyor mu gibi kontrol ederiz.Eğer sağlamıyorsa connection classı çağırmayız.

Destruction mantığı şudur;Bir beandaki bütün işlemler bittikten sonra çalışacak metodu çağırıyor.Örneğin ConfigureApp xxx = new Classpath diye springi çağırdığımız ve nesneyi kapattığımız işlem vardı. xxx.close(); diye işte bunu çağırdığımızda gelecek metodu belirliyor.Ve artık bunu çağırmamıza gerek yok.Diğer bir şekilde anlaman için biz openConnection dan sonra closeConnection yapıyorduk ya hani openConnection’a closeyi destruction olarak tanımlarsak openConnection metodunun içindeki bağlantılar çalıştıktan sonra closeConnection otomatik çalışacak ve bizim ekstra closeConnectionu çağırmamıza gerek kalmayacak.

**Database Classı**

**private** **void** **closeConnection**() { //Destruction yani spring close edilince çalışacak metod)

System.**out**.println("DATABASE KAPATILDI....!");

}

@SuppressWarnings("unused")

**private** **void** **validation**() { //Bean çalışmadan önce çalışacak metod

**if**(database.getUrl() == **null** || database.getUsername() == **null** || database.getPassword()==**null**) {

System.**out**.println("VeriTabanına Giriş Yapılamadı"); **throw** **new** NullPointerException("HATA!HATA!HATA!HATA!");

}

**else** {

System.**out**.println("Giriş Başarılıdır.Metod Çalıştı....!");

} }

* **ApplicationContext.xml ile**

<bean id="DbConnection"

class="com.project.DbConnection.DbConnection" init-method="**validation**" destroy-method="**closeConnection**">

</bean>

* **App Config ile**

@Configuration

**public** **class** AppConfig {

@Bean

**public** Database **database**() {

Database **database** = **new** Database("urlOrnek", "Samet", "1123");

**return** database;

}

@Bean (initMethod = "**validation**", destroyMethod = "**closeConnection**")

//init method yani initialization metod --- destroyMetod ise Destrution metod

**public** DBConnection **dbconnection**() {

DBConnection **dbconnection** = **new** DBConnection();

dbconnection.setDatabase(database());

**return** dbconnection;

} }

s

**Bean Factory Post Processor**

Bu interface ile programda kaç bean var,beanları isimleri gibi bilgileri alabiliriz.Yapmamız gereken bir BeanFactory adında class oluşturmak. BeanFactoryPostProcessor interfacesini implement ettirip aşağıdaki kodu yazmak ve bu olayı applicationContext.xml’de aktif etmek,yada appconfig classına aşağıdaki kodu yazmak.

**public** **class** BeanFactory **implements** BeanFactoryPostProcessor {

@Override

**public** **void** **postProcessBeanFactory**(ConfigurableListableBeanFactory beanFactory) **throws** BeansException {

System.**out**.println("Bean Sayısı = " + beanFactory.getBeanDefinitionCount());

} }

**ApplicationContext.xml**

<bean class = "com.project.bean.factory.BeanFactory"></bean>

**AppConfig classı ile yapım**

@Bean

**public** BeanFactoryPostProcessor **beanFactory**() {

**return** **new** BeanFactory();

}

**Properties file ile İnjection**

Bu konunun mantığı şudur;Diyelim ki applicationContext.xml’de aynı metni birden fazla kez yazacağız,bunun için kendimizi sürekli tekrar etmek yerine aynı metinleri values.properties dosyasından alacağız.src dosyamıza sağ tıklıyoruz,new diyip bir file açıyoruz.Genelde ismini values.properties koyarız.Oranın içine metinID = “Metinimiz budur.” Gibi sol kısmı id benzeri sağ kısmına ise tekrar edilen metni yazıyoruz.Daha sonra applicationContext.xml dosyasına geliyoruz ve aşağıdaki gibi kullanıyoruz.

* Eğer bir properties dosyası kullanacaksak

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="location" value="values.properties"></property>

</bean>

* Eğer birden fazla properties dosyası kullanacaksak.

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<list>

<value>values.properties</value>

<value>valuess.properties</value>

</list>

</property>

</bean>

**ÖRNEK KULLANIM**

**values.properties file**

mySqlUrl=mySqlUrl

mySqlUsername=mySqlUsername

mySqlPassword=mySqlPassword

oracleUrl=oracleUrl

oracleUsername=oracleUsername

oraclePassword=oraclePassword

**applicationContext.xml**

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="location" value="values.properties"></property>

</bean>

<bean id="mySql" class="com.project.dalImpl.MySqlDal">

<property name="url" value="${mySqlUrl}"></property>

<property name="username" value="${mySqlUsername}"></property>

<property name="password" value="${mySqlPassword}"></property>

</bean>

<bean id="oracle" class="com.project.dalImpl.OracleDal">

<property name="url" value="${oracleUrl}"></property>

<property name="username" value="${oracleUsername}"></property>

<property name="password" value="${oraclePassword}"></property>

</bean>

<bean id="dbConnection" class="com.project.DbConnection.DbConnection">

<property name="helper" ref="oracle"></property>

</bean>

**main class**

BeanFactory factory = new ClassPathXmlApplicationContext ("applicationContext.xml");

DbConnection connection = factory.getBean("dbConnection",DbConnection.class);

connection.getConnection();

connection.closeConnection();

**Main Classı**

ICustomerManager **cd** = context1.getBean("managerMySql",ICustomerManager.**class**);

cd.add();

cd.remove();

**Class Config ile Properties İnjection**

**values.properties file**

mySqlUrl=mySqlUrl

mySqlUsername=mySqlUsername

mySqlPassword=mySqlPassword

**valuess.properties file**

oracleUrl=oracleUrl

oracleUsername=oracleUsername

oraclePassword=oraclePassword

AppConfig Classıguration

@PropertySources({

@PropertySource("values.properties"),

@PropertySource("valuess.properties") //Eğer tek bir properties filemiz olsaydı sadece bu satırı yazsak kabul olurdu.

})

public class AppConfig {

@Value("${mySqlUrl}")

private String mySqlUrl;

@Value("${mySqlUsername}")

private String mySqlUsername;

@Value("${mySqlPassword}")

private String mySqlPassword;

@Value("${oracleUrl}")

private String oracleUrl;

@Value("${oracleUsername}")

private String oracleUsername;

@Value("${oraclePassword}")

private String oraclePassword;

@Bean

public IDbHelper MySqlDal() {

MySqlDal mySqlDal = new MySqlDal();

mySqlDal.setUrl(this.mySqlUrl);

mySqlDal.setUsername(this.mySqlUsername);

mySqlDal.setPassword(this.mySqlPassword);

return mySqlDal;

}

@Bean

public IDbHelper OracleDal() {

OracleDal oracleDal = new OracleDal();

oracleDal.setUrl(this.oracleUrl);

oracleDal.setUsername(this.oracleUsername);

oracleDal.setPassword(this.oraclePassword);

return oracleDal;

}

@Bean

public DbConnection dbConnection() {

DbConnection connection = new DbConnection(OracleDal());

return connection;

}

}

Main class

BeanFactory factory2 = new AnnotationConfigApplicationContext(AppConfig.class);

DbConnection connection2 = factory2.getBean("dbConnection",DbConnection.class);

connection2.getConnection();

connection2.closeConnection();

**Spring @Required (Gereklilik)**

Bu konunun mantığı şudur.Diyelim ki database bağlicaz ama url’si girilmeden olmaz.Kesinlikle url girmesini istiyoruz.O zaman databasenin olduğu classa gidip urlnin Set metodunun üstüne @Required yazıyoruz ve applicationContext.xml’e gelip bu requiredlerin aktifleşmesi için şu kodu yazıyoruz:

<bean class="org.springframework.beans.factory.annotation.Required AnnotationBean PostProcessor"> </bean>

Artık url siz kodu çalıştırınca hata alırız.

**Spring @Autowired**

Bu konunun mantığı şudur;Diyelim ki bir Database oluşturucaz.ve bunu dbConnectiona bağlicaz.DbConnection classının içinde Database türünden bir değişken belirliyoruz.Sonra bu değişkenin üstüne @Autowired yazıyoruz.Bu şu demek oluyor.Eğer applicationContext.xml’de

Bir database beanı oluşturursan bu otomatik dbConnetiona bağlanır.ve dbConnection beanında property vermene gerek kalmaz.

Bunu xml dosyasında aktif etmek için yazmamız gereken kod:

<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor">

</bean>

Örnekte daha iyi anlicaksın.

**DbConnectionClassı**

@Autowired

IDatabase IDatabase;

**ApplicationContext.xml**

<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"> //AutoWiredi aktif etmek için yazılan standart kod

</bean>

<bean id = "database" class ="com.project.db.MySqlDatabase"> </bean>

//Databasenin propertyleri database classında verildiği için property value tanımlamadık.

<bean id ="DbConnection" class="com.project.db.DbConnection"> </bean>

//Database autowired olduğu için dbConnection beanının içine property value tanımlamadık.

**Main Class**

ConfigurableApplicationContext **context** = **new** ClassPathXmlApplicationContext ("applicationContext.xml");

DbConnection **db** = (DbConnection) context.getBean("DbConnection");

db.openConnection();

context.close();

**Spring @Qualifier**

Bu konu autowired anotasyonuna yardımcı bir anotasyondur.Diyelim ki DbConnection adında bir beanımız var,ve bunun içindeki IDatabase türündeki database’nin üstüne @Autowired yazdık.Fakat birden çok Idatabase yi implement eden class var.Peki DbConnection adlı beanı mainde oluştururken bu bean hangi classı kullanacağını nereden bilecek?İşte O @Autowired’in hemen altına @Qualifier anotasyonunu yazıp içine tırnak içinde kullanmak istediğimiz classın ıd sini verince.Artık o class kullanılacak.Örnekle Daha iyi anlicaksın.

* public class OracleDal implements IDbHelper {
* public class MySqlDal implements IDbHelper {
* public class DbConnection {

@Autowired

@Qualifier("mySqlDal")

IDbHelper helper;

**Spring @Component**

Bu konunun mantığı şudur;Bir classın üstüne @Component yazıyoruz.Ve artık bean oluşturmamıza gerek kalmıyor.Otomatik bean oluşuyor.Ve classın ismi artık ID’si oluyor.Fakat mainde beanı adını yazdığımız getbean(“ ”) olan yere bu classın adını yazarken baş harfi küçük yazıyoruz kesinlikle.

\*Bunu applicationContext.xml de aktif etmek için yazmamız gereken kod;

<context:component-scan base-package="com.project"/> //paketin ismi

Ve bu

<context:annotation-config></context:annotation-config>

\*Bunu AppConfig classında aktif etmek için yazmamız gereken kod;

@ComponentScan(value = "com.project") //paketin ismi

**DbConnection Classı**

@Component("DbConnection") //maine yazarken dbConnection baş harfi küçük.

**public** **class** DbConnection

**Spring Message Source**

Bu konunun mantığı şudur:farklı dillerde mesajların olduğu properties filesinden mesajları çekiyoruz.Uygulamayı farklı dillerle destekleyebiliyoruz.

Bunun için istediğimiz dil kadar file açıyoruz ama isimleri şöyle olmak şartıyla;

-messages\_en\_US.properties //İngilizce

-messages\_fr\_FR.properties //Fransızca

-messages\_tr\_TR.properties //Türkçe

İçlerini şu şekilde dolduruyoruz.

failure.login.error = Username or Password is wrong. //Sol taraf ID,Sağ taraf mesaj

Diğer dillerin olduğu propertieslerin içinede sol ID aynı olacak fakat sağdaki mesajlar o dile ait olacak şekilde dolduruyoruz.

**ApplicationContext.xml**

<bean //Standart Bean id,basename,messages falan hep aynı

id="messageSource"class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basename" value="messages"></property>

<property name="defaultEncoding" value="UTF-8"></property> //Eclipse i/ş/ğ gibi türkçe harflere duyarlı olduğu için bunu ekledik.Properties dosyasına sağ tıklayıp en altta propertiese girip other kısmını UTF-8’de yapabiliriz.

</bean>

**AppConfig Classı**

@Configuration

**public** **class** AppConfig {

@Bean

**public** MessageSource **messageSource**() { //Burdaki isimler çok önemli değişmemeli

ResourceBundleMessageSource **message** = **new** ResourceBundleMessageSource();

message.setBasename("messages");

message.setDefaultEncoding("UTF-8");

**return** message; } }

**Main Class**

ConfigurableApplicationContext **context** = **new** ClassPathXmlApplicationContext ("applicationContext.xml");

Locale **TR** = **new** Locale("tr", "TR"); //Eclipse TR desteklemediği için TR localesini biz oluşturduk.

String **messageEnglish** = (String) context.getMessage("failure.login.error", args,Locale.**US**);

System.**out**.println(messageEnglish);

String **messageTurkish** = (String) context.getMessage("failure.login.error", args,TR);

System.**out**.println(messageTurkish);

String **messageFrench** = (String) context.getMessage("failure.login.error", args,Locale.**FRANCE**);

System.**out**.println(messageFrench);

context.close();

**Spring ile bir beanın içindeki metodu başka beanda kullanma**

Aşağıdaki örnekte book adında bir classımız ve bu classın içinde showinfo adında kitabın bilgilerini gösteren metod var.Bu metod String değer döndürüyor ve bilgileri sıralı şekilde yazdırıyor.(void değil String). Daha sonra author adında bir classımız var ve bunun içinde de String türünde bookInfo adında değişken var.Ve aşağıdaki yöntemle String değer döndüren metodun çıktısını bookInfoya veriyoruz.

<bean id="BeanBook" class="com.project.model.Book">

<property name="name" value="Yüzüklerin Efendisi"></property>

<property name="publishDate" value="2010"></property>

<property name="page" value="456"></property>

</bean>

<bean id="BeanAuthor" class="com.project.model.Author">

<property name="name" value="JK Rowling"></property>

<property name="date" value="27.11.1980"></property>

<property name="bookInfo" value="#{BeanBook.showInfo('name', 'publishDate','page')}"></property> //parametreleri biz burada dolduruyoruz.

</bean>

**Spring ile bir beanın içindeki değişkeni başka beanda kullanma**

<bean id="BeanBook" class="com.project.model.Author">

<property name="name" value="Yüzüklerin Efendisi"></property>

<property name="publisDate" value="2010"></property>

<property name="pageCount" value="#{BeanBook1.pageCount}"></property>

</bean>

**Context start,stop,refresh(Context Lifecycle)**

Bu aşağıdaki konunun mantığı şudur.Maine ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml"); yazdıktan sonra context.start(); context.stop(); context.refresh(); yazınca consolede yazılacakları belirliyor.Aşağıdaki gibi kullanıyoruz.

public class ContextStart implements ApplicationListener<ContextStartedEvent>{

@Override

public void onApplicationEvent(ContextStartedEvent event) {

System.out.println("-----Context Started-----");

}

}

public class ContextStop implements ApplicationListener<ContextStoppedEvent>{

@Override

public void onApplicationEvent(ContextStoppedEvent event) {

System.out.println("-----Context Stopped-----");

}

}

public class ContextRefresh implements ApplicationListener<ContextRefreshedEvent>{

@Override

public void onApplicationEvent(ContextRefreshedEvent event) {

System.out.println("-----Context Refreshed-----");

}

}

**AOP(Aspect Oriented Programming) Mantığı**

Bu konunun mantığı şudur.Herhangi bir metodu çalıştırmadan önce çalıştırdıktan sonra yada çalışma esnasında yapılacakları belirtiyoruz.Örnek verecek olursak diyelim ki hibernate jpa ile veri tabanına kaydetme silme yada güncelleme işlemi yapacağız.Hibernate sorgularında başlangıçta transaction başlat-sonrasında transaction kapat gibi kodlar vardı.İşte biz örnekle saveQuery diye bir metod yazıyoruz.Yada delete yada merge farketmez.AOP ile bu metoddan önce transaction başlatma metodu yine AOP ile bu metoddan sonra transaction kapatma gibi metodlar yazıyoruz.Ve artık saveQuery yada deleteQuery metodunun içinde sadece kaydetme silme güncelleme kodu yazıyoruz.

Bunu projeye dahil etmek için mvnrepository’den aspectjrt ve aspectjweaver ın dependency sini alıp projeye dahil etmemiz gerekiyor.Daha sonra com.project.aop adında bir paket açıp AopLog adında class açıyoruz ve aşağıdaki gibi kodları yazıyoruz.

* **@Before-@After**

Bu alttaki 2 anotasyon kesin olmalı.!

@Aspect //Aspect olduğunu belli ettiğimiz anotasyon

@Component // Component yazıp bunu otomatik bean olarak tanımlıyoruz.

public class AopLog {

//@Before metoddan önce çalıştırılacağını belirtiyor.

//Bu alttaki mavi alanda execution kısmı standart,parantez içine ise hangi metodda uygulayacağımızı belirtiyoruz.

@Before("execution (public \* findProductNames(..))")

public void beforeAdvice() {

System.out.println("İşlem Başlatılıyor");

}

//@After metoddan sonra çalıştırılacağını belirtiyor.

//Bu alttaki mavi alanda execution kısmı standart,parantez içine ise hangi metodda uygulayacağımızı belirtiyoruz.

@After("execution (public \* findProductNames(..))")

public void afterAdvice() {

System.out.println("İşlem Kapatılıyor.");

}

}

**AppConfig Classı ile**

@Configuration

@ComponentScan("com.project")

@EnableAspectJAutoProxy

public class AppConfig {

}

* **@PointCut**

Bunun amacı kod fazlalığını engellemek için before yada after’ın içine yazığımız yukarıdaki kodu yani yolu tek bir yerde tutuyor.Sonra heryerde kullanıyor.

Bu birnevi before afterı kullancağımız classı belirtip ona bir takma ad veriyoruz.Sonra before aftera gelip onlarda bu takma adı kullanıyoruz gibi düşün.

@Aspect

@Component

public class AopLog {

@Before("findProductNames()")

public void beforeAdvice() {

System.out.println("İşlem Başlatılıyor");

}

@After("findProductNames()")

public void afterAdvice() {

System.out.println("İşlem Kapatılıyor.");

}

//Hangi classta kullanacağımızı burada belirttik.

//Bu metodun ismi önemli çünkü yukarıda kullanıcaz.

\*\*\*\*\*\* @Pointcut("execution (public \* findProductNames())")

public void findProductNames() {

}

//-----------------------------------------------

//-----------------------------------------------

@Before("pointCutProductRepository()")

public void beforeAdvice1() {

System.out.println("İşlem Başlatılıyor");

}

@After("pointCutProductRepository()")

public void afterAdvice1() {

System.out.println("İşlem Kapatılıyor.");

}

//Bütün classlarda kullan demek.

\*\*\*\*\*\* @Pointcut("within (com.project.repositoryImp.UserRepository)")

public void pointCutProductRepository() {

}

}

* **@JoinPoint**

Bu konunun mantığı.Yukarıdaki kodlar console’da çalışınca şöyle bir çıktı oluşturuyordu.

İşlem Başlatılıyor.

(\*\*\*METODUN İÇİNDEKİ İŞLEMLER\*\*\*)

İşlem Kapatılıyor.

Fakat bu işlemlerin hangi classta olduğu vs bilgisi gelmiyordu.Sonra şu kodları ekledik.

@Before("pointCutProductRepository()")

public void beforeAdvice(JoinPoint joinPoint) {

System.out.println("İşlem Başlatılıyor " + joinPoint);

}

@After("pointCutProductRepository()")

public void afterAdvice(JoinPoint joinPoint) {

System.out.println("İşlem Kapatılıyor. " + joinPoint);

}

@Pointcut("within (com.project.repositoryImp.\*)")

public void pointCutProductRepository() {

}

Artık çıktı şu şekilde,bize o metodun konumunu da veriyor.

İşlem Başlatılıyor execution(List com.project.repository.IProductRepository. findProducts())

(\*\*\*METODUN İÇİNDEKİ İŞLEMLER\*\*\*)

İşlem Kapatılıyor. execution(List com.project.repository.IProductRepository. findProducts())

* **@AfterReturning**

Bu konunun mantığı şudur,Genellikle return değeri olan metodlar için kullanılır.Ve return değerinden hemen sonra ama yine metod içinde kullanılır.Mesela return değerini döndürdükten sonra istediğimiz logu tutubilir,ayrıca metodun return değerini de yazırabiliriz.Normal @After’da metoddan çıkıp yazdırdığı için return değerini yazdıramazdık.

Bu özelliği kullanabilmek için pointCut kullanmamız şart.Aşağıdaki gibi anotasyonun parantezine kodu yazıyoruz,Aşağıdaki metodda göründüğü gibi String türünde metin parametresi var,o string tipi loglamak istediğimiz metodun return değerinin tipiyle aynı olmak zorunda.Şayet integer tipinde bir metodda bu özelliği kullanmış olsaydık int xx yazacaktık ve bu xx aynı zamanda anotasyonun returning özelliğinde de yazmalıydı.

@AfterReturning(pointcut = "execution (public \* findProductNames())",returning = "metin")

**public** **void** **afterReturn**(JoinPoint joinPoint,String metin) {

System.***out***.println(metin);

System.***out***.println("Return edildikten hemen sonra" + joinPoint);

}

* **@AfterThrowing**

Bunun mantığı şudur.Hata patlattıktan sonra çağıralacak metoddur.Mesela diyelim ki findProductById diye ıd ile product bulan sql metodumuz var.Bu metodun içini şöyle doldurduk.

**public** Product **findProductById**(**int** productId) {

**if**(productId>0) {

TypedQuery<Product> **typedQuery** = **this**.entityManager.createQuery (ProductQueries.***findProductById***,Product.**class**);

typedQuery.setParameter("productId", productId);

Product **product** = typedQuery.getSingleResult();

**return** product;

}

**else** {

**throw** **new** RuntimeException(); //Bak burada eğer productId yi 0 dan küçük girerse runtimeexception fırlat dedik.

}

}

Ve bizim afterthrowing metodunun içine geldik ve aşağıdaki kodları yazarak dedik ki runtime exception diye bir hata patlarsa findProductById metodunda şu metodu çağır.

@AfterThrowing(pointcut = "execution (public \* com.project.repositoryImp. ProductRepository.findProductById(\*))",throwing = "exception")

public void afterThrowingAdvice(JoinPoint joinPoint,RuntimeException exception) {

System.out.println("Geçersiz index değeri HATA\*HATA\*HATA\*HATA\*HATA");

System.out.println("METOD= " + joinPoint);

System.out.println("HATA= " + exception);

}

//After throwing anotasyonunun içindeki throwing özelliğindeki isimle(exception) bu adviceyi kullanacağımız metodun parametresindeki hata ismiyle aynı olmalı

//poincut özelliği kesinlikle olmalı yoksa throwing özellğini kullanamazsın.

//Joinpointte kullanarak hatanın olduğu metodu belirtirsek daha güzel anlaşılır olur.

* **@Around**

Belirlenen bir metodun hem başlamasından önce hem bitmesinden sonra olacak işlemleri yapmamıza yardımcı oluyor.

@Around("findProductByIdAdvice()")

**public** Object **aroundAdvice**(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {

System.***out***.println("Around başladı" + proceedingJoinPoint);

Object object=proceedingJoinPoint.proceed();

System.***out***.println("Around bitti" + proceedingJoinPoint);

**return** object; //Object türünden sonuç döndürmesi şart.

}

@Pointcut("within (com.project.repositoryImp.\*)")

**public** **void** **findProductByIdAdvice**() {

}

//ProceedingJoinPoint yazmamız şart

//İçerideki kodları throws Throwable ile yada try catch ile yapmamız şart

**Spring İmport**

Bu konunun mantığı şudur.Büyük projelerde tek bir applicationContext.xml dosyası kullanılmaz.Yine resources altında bir folder içinde yada dışında beans.xml, database.xml, abc.xml gibi farklı farklı xlm configuration dosyaları olur.Ve bunları mainde çağırırken isimleriyle falan uğraşmamak için bir tane applicationContext.xml olur herzamanki gibi ve oradan alırız.

**applicationContext ile yapma**

<import resource="beans/beans.xml"/> //bean folderının altındaki bean.xml i kullan

<import resource="beans.xml"/> //bean.xml i kullan

<import resource="database/database.xml"/> //bean folderının altındaki bean.xml i kullan

* Bu yukarıdaki kodları yazınca artık hangi beanı kullanmak istersek kullanalım main classpathxlm’de applicationContext.xml i yazacağız.

**AppConfig ile yapma**

@Configuration

**public** **class** BeanConfig {

BEANLAR BEANLAR BEANLAR

}

@Configuration

**public** **class** DatabaseConfig {

BEANLAR BEANLAR BEANLAR

}

@Configuration

@Import({

DatabaseConfig.**class**,BeanConfig.**class**

})

**public** **class** AppConfig {

}

* Bu yukarıdaki kodları yazınca artık hangi beanı kullanmak istersek kullanalım main annotation’de AppConfig classını kullanacağız.