Interacting with multiuple DB s/ws from spring App using spring data jpa

creating

=====

(or)

multiple DataSources from spring App using Spring data Jpa / Spring boot data JPA

boot

boot

physical

=>To interact with multiple db s/ws or to interact with different Logical DBs of same DB s/w we need to use this concept.. Here we can not enjoy DataSource object that comes through AutoConfiuration. The AutoConfiguration based DataSource object always point Single DB s/w or single logical DB.. But we need pointing multiple Db s/ws or multiple Logical DBs of same Db s/w... So we need to go for manual Configuration of spring beans including DataSources using 100%code driven cfgs or Java Config cfgs in spring boot Application. (Indicates that we need to go for lots of manual cfgs in spring boot App)

usecases:: a) Transffering bank accounts details from one bank to another bank if one bank aquires another bank

Basics recap

==========

we

b) Transfer Moneny operation between two banks (IMPS,RTGS, NFTS Apps)

c) Save the product details in mutiple Db s/ws one for main use, another

for backup

is

d) One App/Project saving different products with different Db s/ws

like customers info oracle Db s/w and products or offers info in mysql Db s/w

e) Website dispalying the info /report by collecting from different Db s/ws. and etc...

configuration

=>if do not provide bean id for @Bean method based spring bean then method name it self will be taken as the default bean id.

=>Logical DB is a Logical Partition of the physical DB s/w which will be created on 1 per Project basis i.e every project related DB tables will be created in the project related Logical DB.

=> In oracle DB s/w,every Logical DB is identified with its SID (service ID) => In mysql DB s/w every Logical DB is identified with its db name

Physical DB s/w (oracle)

Logical DB1 sid: p1

Logical DB2 sid: p2

Logical DB3 sid: p3

DB Engine

Physical DB s/w (MySQL) Logical DB1 db name:: p1

Logical DB2 db name:: p2

Logical DB3 db name:: p3

Logical DB4 db name:: p4

DB engine

@Bean

public DataSource createDs(){

returns ds;

}

the default bean id is: createDs

=>if we want to use IOC container supplied object in the @Bean method it can be done in two ways

all

(a) Inject to @Configuration class and use in @Bean methods

(The Injected object is visible in all @Bean methods)

@Configuration

public class DBConfig{

@Autowired

private Environment env;

@Bean

public DataSource createDs1(){

}

// use "env" object

@Bean

public DataSource createDs2()}{ //use "env" object

This object holds properties file,

sys properties, env. variable, profile values.

=> option (a) is good if the

object

container managed is required

in multiple @Bean methods of

@Configuration class

note:: HEre container managed object is nothing but spring bean

} }

(b) Pass it as the parameter of @Bean method

@Configuration

public class DBConfig{

@Bean

(Here the Injected object can be made specific to one @Bean method) public DataSource createDs1 (Enviroment env){ ... //use env object

} }

=> option (b) is good if the

container managed object is required

only in one @Bean method of @Configuration class

when we get DataSource object through AutoConfiguration.. the following other object will be created internally while dealing spring data JPA

needs

with

TransancationManager obj→ EntityManagerFactory obj_needs

(TransactionManager(1)

Impl class obj) ↓

[For commit(), rollback() activities]

(EntityManagerFactory (1)

Impl class obj)

[It is like sessionFactory object) ↄ

(It is platform/base for

performing curd operatons

like persist(-),update(-),get(-,-)...)

to

DataSource obj (DataSource(1) Impl class obj)

interacts with

Db s/w

(maintains jdbc con pool with con objs)

Linked Spring Data Repositories

(like JpaRepository/PagingAndSortingRepository and etc..)

=>While interacting with multiple Db s/ws, stop using Autoconfiguration based DataSoruce, EntityManagerFactory, TransactionManager objects.. start creating them manually using Java Config approach /100% code driven cfg with the support of @Bean methods.

To interact with two DB s/ws or two logical dbs of a Db s/w

Spring Boot App

needs

TransancationManager obj→→→→ EntityManagerFactorypbj_needs DataSource obj1—

interacts with

Db s/w 1

Linked To

spring Data Repository 1 (@EnableJPaRepository)

needs

2

TransancationManager obj2→ EntityManagerFactory obj_needs DataSource obj2 -

Linked To

spring Data Repository2

(@EnableJPaRepository)

interacts with

Db s/w 2

note1:: SEssionFactory obj is Hibernate Object that represents multiple services required for completing the CURD operations note2:: TransactionManager is responsible for completing the CURD Operations by executing Do Every thing or nothing principle logic on the Persistence operations

Like this we need n sets of TransactionManager, EntityManagerFactory, dataSource objs to interact with "n" Db s/ws from our spring data jpa application

Another way of creating DataSource object using @Bean methods (with out using Autoconfiguration DataSource obj)

=============

=========================

====================

note:: To change from one DB s/w to another DB s/w (i.e at a time one DB s/w interaction) then we need to use Profiles in spring boot

In application.properites

oracle.datasource.driver-class-name-oracle.jdbc.driver.Oracle Driver

oracle.datasource.jdbc-url=jdbc:oracle:thin:@localhost:1521:xe

oracle.datasource.username=system

oracle.datasource.password=manager

mysql.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

prefix is our choice like

oracle.datasource and the last nodes in the keys are fixed like (driver-class-name, jdbc-url, username, password)

mysql:.datasource.jdbc-url= ......

mysql.datasource.username= root

mysql.datasource.password=r root

In contiguration class

======

@Configuration

public class DBConfig{

@Bean

bulk Injection

@Configuration Properties(prefix="oracle.datasource") public DataSoruce create oracleDs()

return DataSourceBuilder.create().build();

prefix is our choice like

mysql.datasource and the last nodes

in the keys are fixed like (driver-class-name, jdbc-url, username, password)

if we add spring-data-jpa-starter to the

the Project it intenrally uses hikari cp DataSource by default..

}

Factory class

(bulk Injection)

note:: To interact with more than one DB s/ws at a time, take the support of above process.. (manual configuration of multiple datasources, EntityManagerFactory objs, Transaction Manager objs)

DataSourceBuilder is predefined class having properties dirverClassName, url,username, password. By using the jdbc driver details injected to this DataSource Builder we can create DataSource pointing certain jdbc con pool

@Bean

@ConfigurationProperties(prefix="mysql.datasource")

public DataSoruce createMySQLDS()

return DataSource Builder .create().build();

}

Example App

static method

method chainig

to

(making spring boot data jpa app inserting Product info oracle Db table and Offers Info mysql DB table)

step1) keep both mysql and oracle Db s/w ready..

step2) create spring boot starter project adding the following starters (jars)

X Lombok

X Spring Data JPA

X MySQL Driver

X Oracle Driver

S

step3) write jdbc properties for both Db s/w having two different custom prefixes for keys in applicaiton.properties file (For the same keys suffixes are fixed accoringDataSourceBuilder class) (last words in the keys)

in application.properties

#jdbc properties for oracle

oracle.datasource.driver-class-name-oracle.jdbc.driver.Oracle Driver

oracle.datasource.jdbc-url=jdbc:oracle:thin:@localhost:1521:xe

oracle.datasource.username=system

```
oracle.datasource.password=manager
#jdbc properties for mysql
mysql.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
mysql.datasource.jdbc-url=jdbc:mysql:///ntspbms714db
mysql.datasource.username=root
mysql.datasource.password=root
```

step4) Create Two Configuration classes for two different Db s/ws having

Here the prefixes are not fixed but

suffixes are fixed

@Bean methods creating DataSource, EntityManagerFactory TransactionManager objects

=>LocalContainerEntityManagerFactoryBean is factoryBean(selfless bean) that gives EntityManagerFactory! class object as the Resultant..

=>To create LocalContainerEntityManagerFactoryBean class object

we need EntityManagerFactoryBuilder object.. which comes through

autoconfiguration when we add "spring-data-jpa-starter".

=> Use EntityManageFactoryBuilder object to create LocalContainerEntityManagerFactoryBean object.. which in turn is used

to create EntityManagerFactory object

=> LocalContainerEntityManagerFactoryBean is factory bean class .. So when

it is made dependent to other spring bean (target bean) then it injects the resultant EntityManagerFactory object

```
//OracleDBConfig.java
package com.nt.config;
import java.util.HashMap;
import java.util.Map;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
```

=>FactoryBean is selfless bean i.e

when we ask container to give object

of FactoryBean ..it does not give that object..

it gives the resultant object of FactoryBean

if

taken

FactoryBean as dependent to targetBean..

then the FactoryBean object will not be injected.. the Resultant object given by FactoryBean will be

Injected to taget bean..

Generally FactoryBean classes implement FactoryBean(1)

and contains FactoryBean word at the end of the class name..

note: A bean that implements FactoryBean(1) cannot be used as a normal bean.

A FactoryBean is defined in a bean style, but the object exposed for bean references (getObject()) is always the object that it creates.

import org.springframework.orm.jpa.JpaTransactionManager;

import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean; import org.springframework.transaction.PlatformTransactionManager; import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration

The pkg name of Custom Repository(1)

@EnableTransaction Management

@EnableJpaRepositories(basePackages = "com.nt.repo.prod",

entityManagerFactoryRef = "oracleEMF", transactionManagerRef = "oracleTxMgmr")

public class OracleDBConfig {

@Bean

@Primary

@Configuration Properties(prefix = "oracle.datasource")

public DataSource createOracleDs() {

return DataSourceBuilder.create().build();

(colleted from

@Bean methods)

Gives DataSource obj as spring bean

pointing to oracle jdbc con pool

//MYSQLDBConfig.java package com.nt.config;

import java.util.HashMap;

import java.util.Map;

import javax.persistence.EntityManagerFactory;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.boot.context.properties.Configuration Properties;

import org.springframework.boot.jdbc.DataSourceBuilder;

import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder; import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

import org.springframework.orm.jpa.JpaTransactionManager;

import org.springframework.orm.jpa.Local ContainerEntityManagerFactoryBean; import org.springframework.transaction.PlatformTransaction Manager; import org.springframework.transaction.annotation.EnableTransaction Management;

@Configuration

@EnableTransaction Management

@EnableJpaRepositories(basePackages = "com.nt.repo.promotions",

entityManagerFactoryRef = "mysqlEMF", transactionManagerRef = "mysqlTxMgmr")

public class MySQLDBConfig {

}

@Bean(name="oracleEMF")

@Primary

pre-defined class

public LocalContainerEntityManagerFactoryBean

}

@Primary

On One DataSource spring bean, EntityManagerFactoryBean, Transaction Manager Bean we need to place @Primary to support internally injections that takes as part of AutoConfiguratuin activity AutoConfiguration object

createOracleEntityManagerFactoryBean(EntityManagerFactoryBuilder builder) {

//create Map object having hibernate properties

Map<String, Object> props=new HashMap();

props.put("hibernate.dialect", "org.hibernate.dialect.Oracle 10gDialect");

props.put("hibernate.hbm2ddl.auto", "update");

Gives EntityManager

Factory obj as spring bean pointing to

//create and return LocalContainerEntityManagerFactoryBean class obj which makes oracle Db s/w

//EntityManagerFactory as the sprign bean

return builder.dataSource(createOracleDs()) // datasoruce

.packages("com.nt.model.prod") //model class pkg

.properties(props) //hibernate properties

.build();

@Bean(name="oracleTxMgmr")

public PlatformTransactionManager

To container

to use our TxMgmr

obj not the TxMgmr

obj given by spring boot's auto configuration

@Bean

@Configuration Properties(prefix = "mysql.datasource") public DataSource createMySQLDs() {

}

return DataSourceBuilder.create().build();

@Bean(name="mysqlEMF")

public LocalContainerEntityManagerFactoryBean

createMySQLEntityManagerFactoryBean (EntityManagerFactoryBuilder builder) { //create Map object having hibernate properties

Map<String, Object> props-new HashMap();

props.put("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");

props.put("hibernate.hbm2ddl.auto", "update");

//create and return LocalContainerEntityManagerFactoryBean class obj which makes

//EntityManagerFactory as the sprign bean

return builder.dataSource(createMySQLDS()) // datasoruce

.packages("com.nt.model.promotions") //model class pkg

.properties(props) //hibernate properties .build();

createOracleTxMgmr(@Qualifier("oracleEMF")

Gives JpaTransactionManager

}

EntityManagerFactory factory) {

obj as spring bean pointing

to oracle Db s/w

return new JpaTransactionManager(factory);

}

}

note:: Based on @Primary kept in OracleDBConfig.java class the spring-boot-data-jpa-starter related autoconfiguration takes the required datrasource, EntityManagerFactory and TranactionManager objects from Oracle DBConfig.java class.

step5) Develop two seperate model classes in two different packages..

//offers.java

package com.nt.model.promotions;

import java.time.LocalDateTime;

import javax.persistence.Column;

import javax.persistence.Entity; import javax.persistence.GeneratedValue;

```
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
@Entity
@Table(name="MDS_OFFERS")
@Data
@NoArgsConstructor
@RequiredArgsConstructor
public class Offers {
}
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Integer offerId;
@NonNull
@Column(length = 20)
private String offerName;
@Column(length = 20)
@NonNull
private String offerCode;
@NonNull
private Double discountPercentage;
@NonNull
private LocalDateTime expirtyDate;
```

step6) Develop two seperate Repository interfacaces in two different packages.. //IProductRepo.java

```
package com.nt.repo.prod;
import org.springframework.data.jpa.repository.JpaRepository;
import com.nt.model.prod.Product;
public interface IProductRepo extends JpaRepository<Product,Integer>{
}
```

step7) Develop Runner class to Inject the Repository objects and to test the application.

```
//MultiDataSourceRunner.java
package com.nt.runners;
import java.time.LocalDateTime;
import java.util.Arrays;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
import com.nt.model.prod.Product;
import com.nt.model.promotions.Offers;
import com.nt.repo.prod.IProductRepo;
import com.nt.repo.promotions.IOffersRepo;
@Component
public class MultiDataSourceRunner implements CommandLineRunner {
@Autowired
private IProductRepo prodRepo;
@Autowired
private Offers Repo offersRepo;
@Override
public void run(String... args) throws Exception {
//save objects
prodRepo.saveAll(Arrays.asList(new Product("table", 100.0,60000.0),
}
//Product.java
}
@Bean(name="mysqlTxMgmr")
public PlatformTransaction Manager createMysqlTxMgmr(@Qualifier("mysqlEMF")
```

To make Container not to use

```
EntityManagerFactory factory) {
```

the AutoConfiguration basaed EntityManagerFactory object

```
return new JpaTransactionManager(factory);
```

and to use our EntityManagerFactory obj

```
package com.nt.model.prod;
import javax.persistence.Column;
import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
```

```
@Entity
@Table(name="MDS_PRODUCT")
@Data
@NoArgsConstructor
@RequiredArgsConstructor
public class Product {
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Integer pid;
@NonNull
@Column(length = 20)
private String pname;
@NonNull
private Double qty;
@NonNull
private Double price;
}
```

//1OffersRepo.java

```
package com.nt.repo.promotions;
import org.springframework.data.jpa.repository.JpaRepository; import com.nt.model.promotions.Offers;
public interface IOffers Repo extends JpaRepository<Offers, Integer> {
}
```

MDS_PRODUCT

Result Grid

Filter Rows:

Edit: ✔ B & Export/Import:

Columns Data Model | Constraints | Grants Stati:

offerId

EX Sort.. Filter:

discountPercentage

6

100

PID PNAME PRICE QTY

7

200

```
1 149 table 60000 100
```

8

100

expirtyDate 2021-11-20 10:11:00.000000 B1G1 2021-11-20 10:11:00.000000 B1G2 2021-11-20 10:11:00.000000

offerCode offerName

Buy-1-Get-1

Buy-1-Get-2

B2G2

Buy-2-Get-2

NULL

NULL

NULL

NULL

2 150 chair

6000

10

3

151 sofa

62000

11

(For mysql DB)

(for oracle DB)

new Product("chair", 10.0,6000.0),

new Product("sofa", 11.0,62000.0)

));

System.out.println("Products are saved");

=");

System.out.println("=========

offers Repo.saveAll(Arrays.asList(new Offers("Buy-1-Get-1","B1G1",100.0,LocalDateTime.of(2021,11,20, 10, 11)),

new Offers("Buy-1-Get-2","B1G2",200.0,LocalDateTime.of(2021,11,20, 10, 11)),

new Offers("Buy-2-Get-2","B2G2",100.0,LocalDateTime.of(2021,11,20, 10, 11))

));

System.out.println("offers are saved");

prodRepo.findAll().forEach(System.out::println);

--"); System.out.println("===========display records(offers): offersRepo.findAll().forEach(System.out::println);

System.out.println(" ==="); System.out.println(": =======display records(product)

System.out.println("-

=");

watch these videos before comming for tommorrows class

Model1, MVC1 and MVC2 architecture videos

https://www.youtube.com/watch?v=HeA8AGNLjPw

https://www.youtube.com/watch?v=_kKEjRUqVqs&t=13s

https://www.youtube.com/watch?v=Zu5E8jGqoUU&t=19s

");

BootDataJPA14-MultipleDataSources [boot]

>Spring Elements

#src/main/java

>com.nt

com.nt.config

>MySQLDBConfig.java

> OracleDBConfig.java

com.nt.model.prod

> Product.java

com.nt.model.promotions

› ▸ Offers.java

com.nt.repo.prod

>IProductRepo.java

com.nt.repo.promotions >IOffersRepo.java

com.nt.runners

> MultiDataSourceRunner.java

>#src/main/resources

>

src/test/java

> JRE System Library [JavaSE-11]

> Maven Dependencies

> src

>

target

W HELP.md

mvnw

mvnw.cmd

M pom.xml