***OFFLINE DTD***

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

Pom.xml **EX 17**

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>5.3.8</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</ artifactId >

<version>8.0.26</ version >

</dependency>

DemoBean.Java

**public** **class** DemoBean {

JdbcTemplate jt;

**public** DemoBean(JdbcTemplate jt) {

**this**.jt = jt;

}

**public** **void** createTable(){

jt.execute("create table student(sid int(11),sname varchar(45))");

System.***out***.println("Table Created SuccessFully");

}

**public** **void** insertRow(){

**int** k = jt.update("insert into student(sid,sname) values(1001,'Shahnawaz')");

Spring.xml

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

**In Spring-5.2.2 no need to set *driverClassName***

<bean id=*"one"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/sample"*></property>

<property name=*"username"* value=*"root"*></property>

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

</bean>

<bean id=*"two"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<constructor-arg ref=*"one"*></constructor-arg>

</bean>

<bean id=*"three"* class=*"p3.DemoBean"*>

<constructor-arg ref=*"two"*></constructor-arg>

</bean>

</beans>

Client.java

**public** **class** Client {

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

ApplicationContext ap = **new** ClassPathXmlApplicationContext("Spring.xml");

DemoBean d = (DemoBean)ap.getBean("three");

d.createTable();

d.insertRow();

**MySql Connector jar file is also Required**

}

}

Employee.java **EX 18**

**public** **class** Employee {

**private** **int** empNo;

**private** String ename;

**private** **int** eSal;

**private** **int** deptNo; Getter and Setters

EmployeeDao.java(interface)

**public** **interface** EmployeeDao {

**public** **int** insertEmp(Employee e);

**public** **void** selectAllEmployee();

**public** **int** updateEmployee(**int** esal,**int** empNo);

EmployeeDaoIMPL implements EmployeeDao

**public** **class** EmployeeDaoIMPL **implements** EmployeeDao{

JdbcTemplate jt;

**public** **void** setJt(JdbcTemplate jt) {

**this**.jt = jt;

}

@Override

**public** **int** insertEmp(Employee e) {

**int** eNo = e.getEmpNo();

String eName = e.getEname();

**int** eSal = e.geteSal();

**int** deptNo = e.getDeptNo();

Object[] param = {eNo,eName,eSal,deptNo};

**int** k = jt.update("insert into employee values(?,?,?,?)",param);

**return** k;

}

@Override

**public** **void** selectAllEmployee() {

List l = jt.queryForList("select \* from Employee");

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

}

@Override

**public** **int** updateEmployee(**int** esal, **int** empNo) {

Object[] param = {esal,empNo};

**int** k = jt.update("update Employee set eSal = ? where eNo = ?",param);

**return** k;

//This totally called Persistency Logic

DemoBean.java

**public** **class** DemoBean {

**private** EmployeeDao ed;

**public** **void** setEd(EmployeeDao ed) {

**this**.ed = ed;

}

**public** **void** insert(Employee e){

**int** k = ed.insertEmp(e);

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

}

**public** **void** selectAll(){

ed.selectAllEmployee();

}

**public** **void** modify(**int** esal,**int** eNo){

**int** k = ed.updateEmployee(esal, eNo);

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

}

//This totally called Business Logic

}

Spring.xml

<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*

*http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"*>

<bean id=*"one"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/sample"*></property>

<property name=*"username"* value=*"root"*></property>

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

</bean>

<bean id=*"two"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<!-- <constructor-arg ref="one"></constructor-arg> -->

<property name=*"dataSource"* ref=*"one"*></property>

</bean>

<bean id=*"three"* class=*"p4.EmployeeDaoIMPL"*>

<property name=*"jt"* ref=*"two"*></property>

</bean>

<bean id=*"four"* class=*"p4.DemoBean"*>

<property name=*"ed"* ref=*"three"*></property>

</bean>

</beans>

Mainc.java

**public** **class** Mainc {

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

ApplicationContext ap = **new** ClassPathXmlApplicationContext("Spring.xml");

DemoBean d = (DemoBean)ap.getBean("four");

Employee e = **new** Employee();

e.setEmpNo(1);

e.setEname("Sam");

e.seteSal(2000);

e.setDeptNo(101);

d.insert(e);

d.selectAll();

***Just create one class*** EmployeeRowMapper ***in Ex:18***

***And make changes in*** EmployeeDaoIMPL ***inside***

***selectAllEmployee() method***

EmployeeRowMaper.java **EX 19**

**public** **class** EmployeeRowMapper **implements** RowMapper{

@Override

**public** Object mapRow(ResultSet rs, **int** a) **throws** SQLException {

Employee e = **new** Employee();

e.setEmpNo(rs.getInt(1));

e.setEname(rs.getString(2));

e.seteSal(rs.getInt(3));

e.setDeptNo(rs.getInt(4));

**return** e;

***Changes inside*** EmployeeDaoIMPL Class

**public** **void** selectAllEmployee() {

List l = jt.query("select \* from Employee",new EmployeeRowMapper());

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

**Get Single Employee class Object with where clause using query method**

List<User> listUser = **this**.jdbcTemplate.query("select \* from user where uname = ? ",**new** UserMapper(),uName);

**Spring *Data JPA***

**Important Interfaces**

The following are the 3 base interfaces defined in the Spring data commons project

1. **Repository**: It is a central interface in the spring data repository abstraction It is a **MARKER *interface*,** if we are extending this interface, we have to declare our own method and the **implementation provided by Spring at Run-Time**. Its required 2 Parameters **Respositry<Entity,Id>** here ID is primary key of our Entity, this is the super interface for **CrudRepository**
2. **CrudRepository:** Its provides method for Crud Operations Its extends Repository interface,Its required 2 Parameters **CrudRepository<Entity,Id>** here ID is primary key of our Entity
3. S save(S entity)
4. Iterable<S> save(Iterable<S> entities)
5. T findOne(Id primaryKey)
6. boolean exists(Id primaryKey)
7. Iterable findAll()
8. delete(Id primaryKey)
9. delete(entity)

**So,** if we are extending **CrudRepository** there is no need implementing your own methods, just extends this interface and leave it as blank, required implementation are provided at Runtime

1. **JPARepository:**
2. **PagingAndSortingRepository: It** is Specialized for pagination, it extends **CrudRepository**
3. Iterable<T> **findAll**(Sort sort)
4. Page<T> **findAll(**Pageable pageable**)**

**Example of CrudRepository (insert,select,delete,update) :: Useful for predefined methods**

Pom.xml

<!-- JPA Provider (Hibernate) -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>5.4.4.Final</version>

</dependency>

<!-- Spring Data JPA -->

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-jpa</artifactId>

<version>2.6.0</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>5.3.8</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.8</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.20</version>

</dependency>

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>42.2.9</version>

</dependency>

Employee.java

@Entity

@Table(name = "e")

**public** **class** Employee {

@Id

@SequenceGenerator(name = "seq\_gen",sequenceName = "MY\_SEQ\_GEN", initialValue=1, allocationSize=1)

@GeneratedValue(strategy = GenerationType.***SEQUENCE***,generator = "seq\_gen")

@Column(name = "eid")

**int** eid;

@Column(name = "eName")

String eName;

@Column(name = "sal")

**int** sal; ***//Getters and Setters***

**Need to create** a **SEQUENCE** in DB no matter if we **Identity**

***In SQL*** CREATE SEQUENCE *my\_seq\_gen* START 1;

EmployeeRepository.java ***in p2 package and remaing all in p1***

**package** **p2**;

**import** org.springframework.data.repository.CrudRepository;

**import** p1.Employee;

**public** **interface** EmployeeRepository **extends** CrudRepository<Employee, Integer> {

**No need to provide implementation for CrudRepository interface it will provided by Spring** }

EmployeeService.java

@Service

**public** **class** EmployeeService {

@Autowired

EmployeeRepository employeeRepository;

@Transactional

**public** Employee **addPerson**(Employee employee) {

**return** employeeRepository.save(employee);

/\* Now we are adding single employee into Db but we can save Collection of

\* Employee also like employeeRepository.save(Arrays.asList(e1,e2)) in one shot\*/

}

**public** Iterable<Employee> **selectAll**() {

**return** employeeRepository.findAllById(Arrays.*asList*(1,2));

}

**public** **void** **deleteEmployee**(Employee employee) {

employeeRepository.delete(employee);

}

**public** Employee **updateEmployee**(Employee employee) {

employee.setEid(1);

Employee employee2 = employeeRepository.findById(employee.getEid()).get();

employee2.seteName("New Name");

**return** employeeRepository.save(employee2);

AppConfig.java

**package** p1;

**import** java.util.Properties;

**import** javax.sql.DataSource;

**import** org.hibernate.jpa.HibernatePersistenceProvider;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

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

**import** org.springframework.jdbc.datasource.DriverManagerDataSource;

**import** org.springframework.orm.jpa.JpaTransactionManager;

**import** org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

**import** org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration

@EnableTransactionManagement

@EnableJpaRepositories("p2")

@ComponentScan(basePackages = "p1")

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

@Bean

**public** LocalContainerEntityManagerFactoryBean entityManagerFactory() {

LocalContainerEntityManagerFactoryBean lfb = **new** LocalContainerEntityManagerFactoryBean();

lfb.setDataSource(dataSource());

**/\* While you can't use JPA without the provider,Like in future if you want to**

**\* use TopLink or Ibatis so just change this provider Accordingly\*/**

lfb.setPersistenceProviderClass(HibernatePersistenceProvider.**class**);

**/\* Your LocalContainerEntityManagerFactoryBean needs to know the location of you @Entity classes\*/**

lfb.setPackagesToScan("p1");

lfb.setJpaProperties(hibernateProps());

**return** lfb;

}

@Bean

**public** DataSource dataSource() {

DriverManagerDataSource ds = **new** DriverManagerDataSource();

ds.setUrl("jdbc:postgresql://127.0.0.1:5433/Sample");

ds.setUsername("postgres");

ds.setPassword("");

ds.setDriverClassName("org.postgresql.Driver");

**return** ds;

}

**public** Properties hibernateProps() {

Properties properties = **new** Properties();

properties.setProperty("hibernate.dialect", "org.hibernate.dialect.PostgreSQL82Dialect");

properties.setProperty("hibernate.show\_sql", "true");

properties.setProperty("hibernate.hbm2ddl.auto", "update");

properties.put(AvailableSettings.***JPA\_TRANSACTION\_TYPE***, "RESOURCE\_LOCAL");

**return** properties;

}

@Bean

**public** JpaTransactionManager transactionManager() {

JpaTransactionManager transactionManager = **new** JpaTransactionManager();

transactionManager.setEntityManagerFactory(entityManagerFactory().getObject());

**return** transactionManager;

Client.java

**public** **class** Client {

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

ApplicationContext app = **new** AnnotationConfigApplicationContext(AppConfig.**class**);

EmployeeService emp = app.getBean(EmployeeService.**class**);

**//System.out.println(emp.addPerson(e));**

//emp.**selectAll**().forEach(System.out :: println);

Employee e = **new** Employee();

e.setEid(2); **//if we delete via Entity we need to set Eid in Object itself**

//emp.**deleteEmployee**(e);

emp.**updateEmployee**(**new** Employee());

NOW **update e set eName=?, sal=? where eid=?** as we are updating only **eName** but still its updating sal also, So to Resolve add **@DynamicUpdate** annotation on top of Entity class

NOW as you can see **EmployeeRepository** we are just using **predefined** methods now we want **CUSTOM** method

EmployeeRepository.java

**public** **interface** **EmployeeRepository** **extends** **CrudRepository**<Employee, Integer> {

**/\* Here findBy means select and eName is property of Entity \*/**

List<Employee> findByeName(String eName);

Employee findByeNameAndSal(String eName,**int** sal);

***For more details how we can create a Query by assigning into method names***

<https://docs.spring.io/spring-data/jpa/docs/1.3.4.RELEASE/reference/html/jpa.repositories.html>

|  |  |
| --- | --- |
| Keyword | Sample |
| And | findByLastnameAndFirstname |
| Or | findByLastnameOrFirstname |
| Between | findByStartDateBetween |
| LessThan | findByAgeLessThan |
| GreaterThan | findByAgeGreaterThan |
| After | findByStartDateAfter |
| Before | findByStartDateBefore |
| IsNull | findByAgeIsNull |
| IsNotNull,NotNull | findByAge(Is)NotNull |
| Like | findByFirstnameLike |
| NotLike | findByFirstnameNotLike |
| StartingWith | findByFirstnameStartingWith |
| EndingWith | findByFirstnameEndingWith |
| Containing | findByFirstnameContaining |
| OrderBy | findByAgeOrderByLastnameDesc |
| Not | findByLastnameNot |
| In | findByAgeIn(Collection<Age> ages) |
| NotIn | findByAgeNotIn(Collection<Age> age) |
| True | findByActiveTrue() |
| False | findByActiveFalse() |

NOW as you can see above, we are not using any **CrudRepository predefined** methods so we no need to extends **CrudRepository** we can extend **Repository** interface

**Hikari Connection Pool**

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-hikaricp</artifactId>

<version>5.2.10.Final</version>

</dependency>

<dependency>

<groupId>com.zaxxer</groupId>

<artifactId>HikariCP</artifactId>

<version>3.3.1</version>

</dependency>

**AppConfig.java** Replace Datasource method

@Bean

**public** DataSource dataSource() {

HikariConfig config = **new** HikariConfig();

HikariDataSource dataSource;

config.setDriverClassName(DRIVER);

config.setJdbcUrl(URL);

config.setUsername(USERNAME);

config.setPassword(PASSWORD);

config.setConnectionTimeout(300000);

config.setMaximumPoolSize(20);

config.setIdleTimeout(120000);

config.addDataSourceProperty("useLocalSessionState", "true");

config.addDataSourceProperty("cacheServerConfiguration", "true");

config.addDataSourceProperty("cachePrepStmts", "true");

config.addDataSourceProperty("prepStmtCacheSize", "250");

config.addDataSourceProperty("prepStmtCacheSqlLimit", "2048");

dataSource = **new** HikariDataSource(config);

**return** dataSource;

**@NamedQuery in JPA**

EmployeeRepository.java

**public** **interface** EmployeeRepository **extends** CrudRepository<Employee, Integer> {

List<Employee> **getByeNameInfoData**(String eName); **// Our Own Name**

Employee **findByeNameAndSal**(String eName,**int** sal); **// Spring Strategy Name**

}

Employee.java

@NamedQuery(name = "Employee.getByeNameInfoData",query = "from Employee where eName =?1")

@NamedQuery(name = "findByeNameAndSal",query = "from Employee where eName =?1 AND sal = ?2")

**@NamedNativeQuery in JPA**

Employee.java

@NamedNativeQuery(name = "Employee.getByeNameInfoData",query = "select \* from e where eName =?1",resultClass = Employee.**class**)

@NamedNativeQuery(name = "findByeNameAndSal",query = "select \* from e where eName =?1 AND sal = ?2",resultClass = Employee.**class**)

**@Query in JPA (for NON-Native)**

EmployeeRepository.java

@Query(value = "from Employee where eName =?1")

List<Employee> **getByeNameInfoData**(String eName);

@Query(value = "from Employee where eName =?1 AND sal = ?2")

Employee **findByeNameAndSal**(String eName,**int** sal);

**@Query in JPA (for Native)**

@Query(value = "select \* from e where eName =?1",**nativeQuery** = **true**)

List<Employee> **getByeNameInfoData**(String eName);

@Query(value = "select \* from e where eName =?1 AND sal = ?2",**nativeQuery** = **true**)

Employee **findByeNameAndSal**(String eName,**int** sal);

**@Query in JPA with @Param**

EmployeeRepository.java

@Query(value = "select \* from e where eName =:eName AND sal = :sal",nativeQuery = **true**)

Employee findByeNameAndSal(@Param("eName")String eName,@Param("sal")**int** sal);

**@Query in JPA for INSERT, DELETE and Update Operations**

EmployeeRepository.java

**@Transactional**

**@Modifying**

@Query(value = "update Employee set eName =:eName , sal = :sal where eid = :eid")

**void** updateEmployeeById(@Param("eName")String eName,@Param("sal")**int** sal,@Param("eid")**int** eid);

**@Async in JPA** Now if we want execute some query in separate thread

EmployeeRepository.java

**@Async**

CompletableFuture<Employee> findBySal(**int** sal);

EmployeeService.java

**public** CompletableFuture<Employee> findBySal(**int** sal){

**return** employeeRepository.findBySal(sal);

AppConfig.java

**@EnableAsync on Top of class**

Client.java

CompletableFuture<Employee> completableFuture = emp.findBySal(1000);

**Set<Thread> threadSet = Thread.*getAllStackTraces*().keySet();**

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

Employee x = completableFuture.get(7, TimeUnit.***SECONDS***);

System.***out***.println("X " + x);

Bold is use to check our currently all running *Threads*

**@Transactional**

### *Transactions and Proxies*

At a high level, **Spring creates proxies for all the classes annotated with***@Transactional*, either on the class or on any of the methods. The proxy allows the framework to inject transactional logic before and after the running method, mainly for starting and committing the transaction.

**What's important to keep in mind is that,** if the transactional bean is implementing an interface, by default the proxy will be a Java Dynamic Proxy. This means that only external method calls that come in through the proxy will be intercepted. Any self-invocation calls will not start any transaction, even if the method has the **@Transactional** annotation.

Another caveat of using proxies is that **only public methods should be annotated with @Transactional.** Methods of any other visibilities will simply ignore the annotation silently as these are not proxied.

we can now annotate a bean with ***@Transactional*** either at the ***class*** or ***method*** level:

**How @Transactional Work?**

Its like we have method and inside that we are calling **3** **save** method. If **2** **save** method run properly while running **3rd**, we got some exception in that method so all the **save** method will be **rollback.**

Note that by default, rollback happens for runtime, unchecked exceptions only. **The checked exception does not trigger a rollback** of the transaction. We can, of course, configure this behavior with the *rollbackFor* and *noRollbackFor* annotation parameters.

***@Transactional***

**public** FlightBookingAcknowledgement bookFlightTicket(FlightBookingRequest request) {

Student s = **new** Student();

s.setsName("Shahnawaz");

studentRepository.save(s);

PassengerInfo passengerInfo = request.getPassengerInfo();

passengerInfo = passengerInfoRepository.save(passengerInfo);

PaymentInfo paymentInfo = request.getPaymentInfo();

paymentInfo.setPassengerId(passengerInfo.getpId());

paymentInfo.setAmount(passengerInfo.getFare());

paymentInfoRepository.save(paymentInfo);

**return** **new** FlightBookingAcknowledgement("SUCCESS", passengerInfo.getFare(),

UUID.*randomUUID*().toString().split("-")[0], passengerInfo);

}

**Propagation Level**

***@Transactional***(propagation = Propagation.***REQUIRED***)

If an inner method causes a transaction to rollback the outer method will fail to commit and will also be rollback the transactions.

**public** FlightBookingAcknowledgement bookFlightTicket(FlightBookingRequest request) {

Student s = **new** Student();

s.setsName("Shahnawaz");

studentRepository.save(s);

innerBean.save(passengerInfo); ***If its failure outer will also rollback because both are using same transactions***

***@Transactional***(propagation = Propagation.***REQUIRES\_NEW***)

Its like vice verso of above if inner fail outer still commits

Propagation.***NESTED***: It’s related to savePoint and should be use in Spring JDBC manage transactions

Propagation.***MANDATORY***: Existing opened transaction must already exist. If not and exception will be thrown by the container.

Propagation.***NEVER***: Existing opened transaction must NOT already exist. If exist exception will be thrown by the container.

Propagation.***NOT\_SUPPORTED*** and ***SUPPORTS***

**Rollback for Checked or unchecked exceptions**

***@Transactional***(**rollbackFor** = IOException.**class**, **noRollbackFor** = ArithmeticException.**class**)

**Spring Boot Project for JPA**

Pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

application.properties

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

spring.datasource.url = jdbc:mysql://localhost:3306/sample

spring.datasource.username = root

spring.datasource.password =

spring.jpa.show-sql = true

spring.jpa.hibernate.ddl-auto = update

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect

server.port=8080

In a class in which our Spring boot ***main*** method is there on top of class.

**@SpringBootApplication**

***@EnableTransactionManagement***

**@SpringBootApplication**

The **@SpringBootApplication** is a combination of three annotations **@Configuration** (used for Java-based configuration), **@ComponentScan** (used for component scanning), and **@EnableAutoConfiguration** (used to enable auto-configuration in [Spring Boot](http://javarevisited.blogspot.sg/2018/01/how-to-learn-spring-core-spring-mvc-boot-security-framework.html)).