****Transactions :****

**The right way to configure Spring Transactions**, how to use the @Transactional annotation, and common pitfalls(problem).

Basically, there are two distinct ways to configure Transactions, annotations and AOP, each with their own advantages.

**Configure Transactions :**

**Spring 3.1 introduces the @EnableTransactionManagement annotation that we can use in a @Configuration class to enable transactional support:**

**@Configuration**

**@**EnableTransactionManagement****

**public **class PersistenceJPAConfig**{**

**@Bean**

**public LocalContainerEntityManagerFactoryBean**

**entityManagerFactoryBean(){**

**//...**

**}**

**@Bean**

**public PlatformTransactionManager transactionManager(){**

**JpaTransactionManager transactionManager**

**= new JpaTransactionManager();**

**transactionManager.setEntityManagerFactory(**

**entityManagerFactoryBean().getObject() );**

**return transactionManager;**

**}**

**}**

**However, if we're using a Spring Boot project and have a spring-data-\* or spring-tx dependencies on the classpath, then transaction management will be enabled by default.**

**Configure Transactions With XML**

**For versions before 3.1, or if Java is not an option, here is the XML configuration using annotation-driven and the namespace support:**

**<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">**

**<property name="entityManagerFactory" ref="myEmf" />**

**</bean>**

**<tx:annotation-driven transaction-manager="txManager" />**

**The @Transactional Annotation**

**With transactions configured, we can now annotate a bean with @Transactional either at the class or method level.**

**@Service**

****@Transactional****

**public class FooService {**

**//...**

**}**

**The annotation supports further configuration as well:**

* **the Propagation Type of the transaction**
* **the Isolation Level of the transaction**
* **a Timeout for the operation wrapped by the transaction**
* **a readOnly flag – a hint for the persistence provider that the transaction should be read only**
* **the Rollback rules for the transaction**

**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.**

**Potential Pitfalls**

**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.**

**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 (warning) 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.**

### ****Changing the Isolation Level :****

**@Transactional(isolation = Isolation.SERIALIZABLE)**

### ****Read-Only Transactions :**It's also important to understand that the readOnly flag is only relevant inside a transaction. If an operation occurs outside of a transactional context, the flag is simply ignored. A simple example of that would call a method annotated with:**

**@Transactional( propagation = Propagation.SUPPORTS,readOnly = true )**

**From a non-transactional context, a transaction will not be created and the readOnly flag will be ignored.**

### ****Transaction Logging :** A helpful method to understand transactional related issues is fine-tuning logging in the transactional packages. The relevant package in Spring is “org.springframework.transaction”, which should be configured with a logging level of TRACE.**

### ****Transaction Rollback :** We have two ways to rollback a transaction: declarative and programmatic.**

### **In the declarative approach, we annotate the methods with the @Transactional annotation. The @Transactional annotation makes use of the attributes rollbackFor or rollbackForClassName to rollback the transactions, and the attributes noRollbackFor or noRollbackForClassName to avoid rollback on listed exceptions.**

### **The default rollback behavior in the declarative approach will rollback on runtime exceptions.**

**Let's see a simple **example** using the declarative approach to rollback a transaction for runtime exceptions or errors:**

**@Transactional**

**public void createCourseDeclarativeWithRuntimeException(Course course) {**

**courseDao.create(course);**

**throw new DataIntegrityViolationException("Throwing exception for demoing Rollback!!!");**

**}**

**Next we'll use the declarative approach to rollback a transaction for the listed checked exceptions. The rollback in our example is on SQLException:**

**@Transactional(rollbackFor = { SQLException.class })**

**public void createCourseDeclarativeWithCheckedException(Course course) throws SQLException {**

**courseDao.create(course);**

**throw new SQLException("Throwing exception for demoing rollback");**

**}**

**Let's see a simple use of attribute noRollbackFor in the declarative approach to prevent rollback of the transaction for the listed exception:**

**@Transactional(noRollbackFor = { SQLException.class })**

**public void createCourseDeclarativeWithNoRollBack(Course course) throws SQLException {**

**courseDao.create(course);**

**throw new SQLException("Throwing exception for demoing rollback");**

**}**

**In the **programmatic** approach, we rollback the transactions using TransactionAspectSupport:**

**public void createCourseDefaultRatingProgramatic(Course course) {**

**try {**

**courseDao.create(course);**

**} catch (Exception e) {**

**TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();**

**}**

**}**

**The declarative rollback strategy should be favored over the programmatic rollback strategy.**

****How Does Spring @Transactional Really Work?****

* how to use features like propagation and isolation
* what are the main pitfalls and how to avoid them