Spring, a transaction represents a single unit of work that needs to be performed atomically — either all the changes to the database must be committed or none of them should be. Transactions ensure data consistency and integrity by providing the ability to undo a set of changes if an error occurs.

Spring uses a transaction manager to manage transactions. The transaction manager is responsible for starting and ending transactions. When a transaction encounters an error, it can be rolled back to its previous state. Rollback ensures that the database remains in a consistent state by undoing any changes that were made during the transaction.

In Spring, rollback can be performed automatically or manually. Automatic rollback occurs when a **runtime exception** is thrown from a **transactional** method. Manual rollback can be performed by calling the setRollbackOnly() method on the TransactionStatus object.

**Propagation** is another important concept in Spring transactions. Propagation determines how transactions should behave when they are nested within other transactions. There are several propagation options available:

* **REQUIRED**: This is the **default** propagation option. If a transaction already exists when a method annotated with REQUIRED is called, the method will execute within that transaction. If no transaction exists, a new transaction will be started.
* **REQUIRES\_NEW**: This option always starts a new transaction, even if a transaction already exists.
* **SUPPORTS**: This option allows a method to participate in a transaction if one already exists, but doesn’t start a new transaction if one doesn’t exist.
* **MANDATORY**: This option requires that a transaction already exists. If a transaction doesn’t exist, an exception is thrown.
* **NOT\_SUPPORTED**: This option specifies that a method should not participate in a transaction at all.
* **NEVER**: This option specifies that a method should never be called within a transaction.

**Propagation.REQUIRED**

This is the **default** propagation option, and it means that if a transaction is already active when the method is called, the method will run within that transaction. If a transaction is not active, a new transaction will be started. If an exception is thrown and caught within the transaction, the transaction will be rolled back, and any changes made to the database within the transaction will be undone.

Example: Suppose you have a placeOrder method that saves an order and its associated order items to the database. In the test method, the placeOrder method is called with an order item that has a quantity greater than the available stock, which causes an exception to be thrown. Both the order and its order items will be rolled back, and no data will be persisted in the database.

/\*\*  
\* OrderService.java  
\*/  
@Transactional(propagation = Propagation.REQUIRED)  
public void placeOrder(Order order) {  
 // Save the order  
 orderRepository.save(order);  
  
 for (OrderItem item : order.getItems()) {  
 // Find the product  
 Product product = productRepository.findById(item.getProduct().getId())  
 .orElseThrow(() -> new RuntimeException("Product not found"));  
  
 // Update the stock quantity  
 int newQuantity = product.getQuantity() - item.getQuantity();  
 if (newQuantity < 0) {  
 throw new RuntimeException("Insufficient stock");  
 }  
 product.setQuantity(newQuantity);  
 productRepository.save(product);  
  
 // Save the order item  
 item.setOrder(order);  
 orderItemRepository.save(item);  
 }  
  
 order.setStatus(OrderStatus.COMPLETED);  
 orderRepository.save(order);  
}  
  
@Test(expected = RuntimeException.class)  
public void testPlaceOrderWithRequiredPropagationAndException() {  
 Order order = new Order();  
 order.setCustomerName("testPlaceOrderWithRequiredPropagationAndException");  
 order.setStatus(OrderStatus.PENDING);  
  
 List<OrderItem> items = new ArrayList<>();  
 // Add order items  
 OrderItem item1 = new OrderItem();  
 Product product1 = new Product();  
 product1.setName("Product 1 created in testPlaceOrderWithRequiredPropagationAndException");  
 product1.setPrice(BigDecimal.valueOf(10.0));  
 product1.setQuantity(5);  
 productRepository.save(product1);  
  
 item1.setOrder(order);  
 item1.setProduct(product1);  
 item1.setQuantity(2);  
 items.add(item1);  
  
 OrderItem item2 = new OrderItem();  
 Product product2 = new Product();  
 product2.setName("Product 2 created in testPlaceOrderWithRequiredPropagationAndException");  
 product2.setPrice(BigDecimal.valueOf(20.0));  
 product2.setQuantity(10);  
 productRepository.save(product2);  
 item2.setOrder(order);  
 item2.setProduct(product2);  
 item2.setQuantity(15); // Here will cause exception because quantity greater than stock.  
 items.add(item2);  
  
 order.setItems(items);  
 orderService.placeOrder(order);  
  
 // Verify that the order was not saved  
 Order savedOrder = orderRepository.findById(order.getId()).orElse(null);  
 assertNull(savedOrder);  
  
 // Verify that the product 1 quantities were not updated  
 Product updatedProduct1 = productRepository.findById(product1.getId()).orElse(null);  
 assertNotNull(updatedProduct1);  
 assertEquals(5, updatedProduct1.getQuantity());  
  
 // Verify that the product 2 quantities were not updated  
 Product updatedProduct2 = productRepository.findById(product2.getId()).orElse(null);  
 assertNotNull(updatedProduct2);  
 assertEquals(10, updatedProduct2.getQuantity());  
}

Notes: Some of you may be curious about why the product records were committed. Because we didn’t declare the test method to be run in a transaction @Transactional. If we did, the two product records were rollback.

**Propagation.REQUIRES\_NEW**

This option always starts a new transaction, regardless of whether a transaction is already active when the method is called. Any existing transaction will be **suspended** until the new transaction is complete. If an exception is thrown and caught within the transaction, only the changes made within the current transaction will be rolled back, and the outer transaction will continue unaffected.

Example: Suppose you have a service method that will update the product quantity. Using REQUIRES\_NEW will run the product quantity update in another transaction. Any exception that occurred in this transaction will not affect the order creation transaction.

/\*\*  
\* ProductService.java  
\*/  
@Transactional(propagation = Propagation.REQUIRES\_NEW)  
public void updateProductQuantityWithRequiresNewPropagation(OrderItem item) {  
 // Find the product  
 Product product = productRepository.findById(item.getProduct().getId())  
 .orElseThrow(() -> new RuntimeException("Product not found"));  
  
 // Update the stock quantity  
 int newQuantity = product.getQuantity() - item.getQuantity();  
 if (newQuantity < 0) {  
 throw new RuntimeException("Insufficient stock");  
 }  
 product.setQuantity(newQuantity);  
 productRepository.save(product);  
}  
  
/\*\*  
\* SpringTransactionTest.java  
\* Description. : `updateProductQuantityWithRequiresNewPropagation` was using REQUIRES\_NEW propagation,   
\* if exception happened every action inside this transaction should rollback, but the   
\* outer transaction should continue without affected.  
\* Expected Result : Order record is created, product record is rollback.  
\*/  
@Test  
@Transactional  
public void testUpdateProductQuantityWithRequiresNewPropagationAndInnerException() {  
 // Create one order  
 Order order = new Order();  
 order.setCustomerName("testUpdateProductQuantityWithRequiresNewPropagationAndInnerException");  
 order.setStatus(OrderStatus.PENDING);  
 orderRepository.save(order);  
  
 try {  
 OrderItem item = new OrderItem();  
 item.setProduct(new Product()); // Here we passed empty Product, so exception occured.  
 productService.updateProductQuantityWithRequiresNewPropagation(item);  
 } catch (RuntimeException e) {  
  
 }  
  
 // Verify that the order was created.  
 Order savedOrder = orderRepository.findById(order.getId()).get();  
 assertEquals(order.getCustomerName(), savedOrder.getCustomerName());  
}

Notes: In the test method, If you never do a try-catch block, any exception thrown by the updateProductQuantityWithRequiresNewPropagation method will propagate to the caller. Any changes made to the database during the transaction will be rolled back.

**Propagation.MANDATORY**

This option requires that a transaction is already active when the method is called. If a transaction is not active, an exception will be thrown.

Example: Suppose you have a updateOrderStatus method that updates an order status in the database. If the method is called outside of a transaction, an exception will be thrown, since it requires a transaction to be active.

/\*\*  
\* OrderService.java  
\*/  
@Transactional(propagation = Propagation.MANDATORY)  
public void updateOrderStatus(Long orderId, OrderStatus status) {  
 Order order = orderRepository.findById(orderId)  
 .orElseThrow(() -> new RuntimeException("Order not found"));  
 order.setStatus(status);  
 orderRepository.save(order);  
}  
  
/\*\*  
\* SpringTransactionTest.java  
\* Description. : `updateOrderStatus` was using MANDATORY propagation,   
\* if the caller method doesn't initiate a transaction,   
\* an exception will throw.  
\* Expected Result : IllegalTransactionStateException occured.  
\*/  
@Test  
public void testMandatoryPropagationWithoutTransaction() {  
 Order order = new Order();  
 order.setCustomerName("testMandatoryPropagationWithoutTransaction");  
 order.setStatus(OrderStatus.PENDING);  
 orderRepository.save(order);  
 assertThrows(IllegalTransactionStateException.class, () -> {  
 orderService.updateOrderStatus(order.getId(), OrderStatus.COMPLETED);  
 });  
}

**Propagation.NEVER**

This option specifies that a transaction should not be active when the method is called. If a transaction is already active, an exception will be thrown.

Example: Suppose you have a service method that deletes an order and its details from the database. If the method is called within a transaction, an exception will be thrown, since it should not be run within a transaction.

@Transactional(propagation = Propagation.NEVER)  
public void deleteOrder(Order order) {  
 this.orderRepository.delete(order);  
 this.orderDetailRepository.delete(order.getDetail());  
}  
  
@Test  
@Transactional  
public void testNeverPropagationWithTransaction() {  
 assertThrows(IllegalTransactionStateException.class, () -> {  
 orderService.deleteOrder(new Order());  
 });  
}

**Propagation.NOT\_SUPPORTED**

This option specifies that the method should not run within a transaction, but if a transaction is already active, it will be suspended until the method is complete. This means that any changes made to the database within the method will be committed immediately, rather than at the end of the transaction.

/\*\*  
\* ProductService.java  
\*/  
@Transactional(propagation = Propagation.NOT\_SUPPORTED)  
public void createProductWithNotSupportedPropagationWithException() {  
 Product product = new Product();  
 product.setName("This product created with NOT\_SUPPORTED propagation and exception.");  
 product.setQuantity(10);  
 product.setPrice(BigDecimal.valueOf(10.0));  
 productRepository.save(product);  
 throw new RuntimeException("DummyException: Simulating an error");  
}  
  
/\*\*  
\* SpringTransactionTest.java  
\* Expected Result : product was created, order was rollback.  
\*/  
@Test(expected = RuntimeException.class)  
@Transactional  
public void testCreateProductWithNotSupportedPropagationWithTransactionAndException() {  
  
 Order order = new Order();  
 order.setCustomerName("testCreateProductWithNotSupportedPropagationWithTransactionAndException");  
 order.setStatus(OrderStatus.PENDING);  
 orderRepository.save(order);  
  
 productService.createProductWithNotSupportedPropagationWithException();  
}  
  
/\*\*  
\* SpringTransactionTest.java  
\* Expected Result : product was created, order was created.  
\*/  
@Test(expected = RuntimeException.class)  
public void testCreateProductWithNotSupportedPropagationWithoutTransactionAndException() {  
 Order order = new Order();  
 order.setCustomerName("testCreateProductWithNotSupportedPropagationWithoutTransactionAndException");  
 order.setStatus(OrderStatus.PENDING);  
 orderRepository.save(order);  
  
 productService.createProductWithNotSupportedPropagationWithException();  
}

In the first test case, we marked the test method to **@Transactional**, spring will establish a new transaction if there is not exists. createProductWithNotSupportedPropagationWithException() method is called within a transaction and since the method uses the **NOT\_SUPPORTED** transaction propagation setting, this method will run **non-transactionally** and the outer transaction will be **suspended**.

When we throw a **RuntimeException** deliberately, after calling the productRepository.save(). The product record was **no rollback**. Because no transaction is established in the **NOT\_SUPPORTED** method. Therefore, nothing to rollback. We expected the order record to rollback because the RuntimeException throws from the inner method were propagated to the outer method and it is not handled properly. And since the outer method was marked **@Transactional**, so order record was rollback.

Let’s see the second test case, because we never marked the test method to @Transactional, so even though the RuntimeException was propagated to the outer method, with nothing to rollback so both the product and order records were **committed**.

**Propagation.SUPPORTS:**

This option specifies that if a transaction is already active when the method is called, the method will run within that transaction. If a transaction is not active, the method will run outside of the transaction.

/\*\*  
\* ProductService.java  
\*/  
@Transactional(propagation = Propagation.SUPPORTS)  
public Product createProductWithSupportsPropagationAndException() {  
 Product product = new Product();  
 product.setName("This product created with SUPPORTS propagation and exception.");  
 product.setQuantity(10);  
 product.setPrice(BigDecimal.valueOf(10.0));  
 productRepository.save(product);  
 throw new RuntimeException("DummyException: Simulating an error");  
}  
  
/\*\*  
\* SpringTransactionTest.java  
\* Expected Result : product was rollback, order was rollback.  
\*/  
@Test(expected = RuntimeException.class)  
@Transactional  
public void testCreateProductWithSupportPropagationWithTransactionAndException() {  
 Order order = new Order();  
 order.setCustomerName("testCreateProductWithSupportPropagationWithTransactionAndException");  
 order.setStatus(OrderStatus.PENDING);  
 orderRepository.save(order);  
  
 productService.createProductWithSupportsPropagationAndException();  
}  
  
/\*\*  
\* SpringTransactionTest.java  
\* Expected Result : product was created, order was created.  
\*/  
@Test(expected = RuntimeException.class)  
public void testCreateProductWithSupportPropagationWithoutTransactionAndException() {  
 Order order = new Order();  
 order.setCustomerName("testCreateProductWithSupportPropagationWithoutTransactionAndException");  
 order.setStatus(OrderStatus.PENDING);  
 orderRepository.save(order);  
  
 productService.createProductWithSupportsPropagationAndException();  
}

In the first test case, we marked the test method to **@Transactional**, spring will establish a new transaction if there is not exists. createProductWithSupportsPropagationAndException() method is called within a transaction and since the method uses the **SUPPORTS** transaction propagation setting, this method will run within the existing transaction.

When we throw a **RuntimeException** deliberately, after calling the productRepository.save(). The product record was **rollback**. Subsequently, RuntimeException throws from the inner method were propagated to the outer method and it is not handled properly. And since the outer method was marked **@Transactional**, so order record was **rollback** too.

Let’s see the second test case, because the test method was not marked @Transactional, **SUPPORTS**method executes non-transactionally if none exists*.*so even though the **RuntimeException** was propagated to the outer method, nothing to rollback so both the product and order records were **committed**.

You may check out the source code from [GitHub](https://github.com/wynnteo/spring-transaction) to run the services and see the result in the database.