

Spring boot: `@Transactional` (Part1)

Critical Section

Code segment, where shared resources are being accessed and modified.



When multiple request try to access this critical section, Data Inconsistency can happen.

Its solution is usage of **TRANSACTION**

- It helps to achieve ACID property.

A (Atomicity):

Ensures all operations within a transaction are completed successfully. If any operation fails, the entire transaction will get rollback.

Report Abuse X

C (Consistency):

Ensures that DB state before and after the transactions should be Consistent only.

I (Isolation):

Ensures that, even if multiple transactions are running in parallel, they do not interfere with each other.

Durability:

Ensures that committed transaction will never lost despite system failure or crash.

BEGIN_TRANSACTION:

- Debit from A

- Credit to B

if all success:

 COMMIT;

Else

 ROLLBACK;

END_TRANSACTION;

In Spring boot , we can use `@Transactional` annotation.

And for that:

1. we need to add below Dependency in pom.xml
(based on DB we are using, suppose we are using RELATIONAL DB)

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

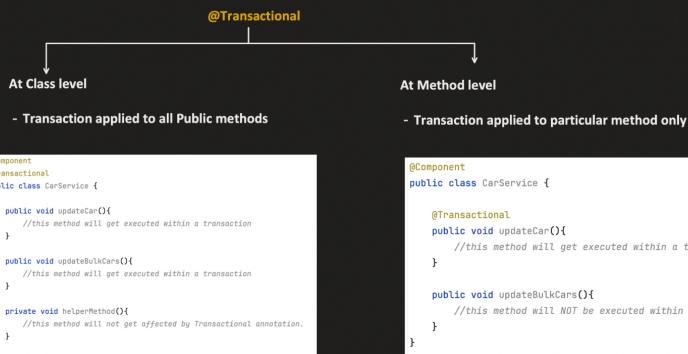
+

Database driver dependency is also required (that we will see in next topic)

2. Activate Transaction Management by using `@EnableTransactionManagement` in main class.
(spring boot generally Auto configure it, so we don't need to specially add it)

```
@SpringBootApplication
@EnableTransactionManagement
public class SpringbootApplication {

    public static void main(String args[]) { SpringApplication.run(SpringbootApplication.class, args); }
}
```



Transaction Management in Spring boot uses AOP.

- 1. Uses Point cut expression to search for method, which has `@Transactional` annotation like:
`@within(org.springframework.transaction.annotation.Transactional)`
- 2. Once Point cut expression matches, run an "Around" type Advice.
Advice is:
`invokeWithinTransaction` method present present in `TransactionalInterceptor` class.

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    User user;

    @GetMapping(path = "/updateuser")
    public String updateUser(){
        user.updateUser();
        return "user is updated successfully";
    }
}
```

```

}
}

@Component
public class User {

    @Transactional
    public void updateUser(){
        System.out.println("UPDATE QUERY TO update the user db values");
    }
}

```

The diagram illustrates the execution flow of the `updateUser()` method within a transactional context. It shows the code structure with various annotations and comments:

```

@Available
protected Object invokeWithinTransaction(Method method, @Nullable Class<?> targetClass,
                                         final InvocationCallback invocation) throws Throwable {
    ...
    * BEGIN_TRANSACTION
    Object retVal;
    try {
        // This is an around advisor. Invoke the next interceptor in the chain.
        // This will normally result in a target object being invoked.
        retVal = invocation.proceedWithInvocation();
    } catch (Throwable ex) {
        // Target exception
        completeTransactionAfterThrowing(txInfo, ex);
        throw ex;
    } finally {
        cleanupTransactionInfo(txInfo);
    }

    if (retVal != null && txAttr != null) {
        TransactionInfo status = txInfo.getTransactionStatus();
        if (status != null) {
            if (txAttr instanceof Future<?> future && future.isDone()) {
                try {
                    future.get();
                } catch (ExecutionException ex) {
                    if (txAttr.rollbackOn(ex.getCause())) {
                        status.setRollbackOnly();
                    }
                }
                catch (InterruptedException ex) {
                    Thread.currentThread().interrupt();
                }
            }
            else if (txAttr instanceof VavrDelegate.VavrTry<?> retVal) {
                // Set rollback-only in case of Vavr failure matching our rollback rules...
                retVal = VavrDelegate.evaluateTryFailure(retVal, txAttr, status);
            }
        }
    }
    ...
    * Any Failure, ROLLBACK
    will happen
    ...
    * All success,
    COMMIT the txn
    ...
    commitTransactionAfterReturning(txInfo);
    return retVal;
}
...

```

Annotations and comments in the code:

- Annotations:** `@Available`, `Method`, `Class`, `InvocationCallback`, `Throwable`.
- Comments:**
 - /* BEGIN_TRANSACTION */
 - /* Any Failure, ROLLBACK will happen */
 - /* All success, COMMIT the txn */
 - Some more code here too in this method, but skipping them, just to avoid getting confused.

Execution flow steps:

- BEGIN_TRANSACTION:** Starts the transaction.
- YOUR TASK:** The user's task is performed.
- Any Failure, ROLLBACK will happen:** If an error occurs, the transaction is rolled back.
- All success, COMMIT the txn:** If successful, the transaction is committed.
- BEGIN_TRANSACTION:** Ends the transaction.
- ROLLBACK;** Handles rollback logic.
- END_TRANSACTION;** Ends the transaction.
- Some more code here too in this method, but skipping them, just to avoid getting confused.**

NOW, we know, how **TRANSACTIONAL** works, we will now see below topics in depth:

- Transaction Context
- Transaction Manager
 - Programmatic
 - Declarative
- Propagation
 - REQUIRED
 - REQUIRED_NEW
 - SUPPORTS
 - NOT_SUPPORTED
 - MANDATORY
 - NEVER
 - NESTED
- Isolation level
 - READ_UNCOMMITTED
 - READ_COMMITTED
 - REPEATABLE_READ
 - SERIALIZABLE
- Configure Transaction Timeout
- What is Read only transaction
- etc..