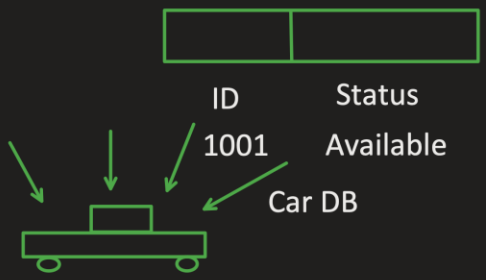
Spring boot: @Transactional (Part1)

Critical Section

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

```
{
    Read Car Row with id: 1001
    If Status is Available:
        Update it to Booked
}
```



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.

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)

Spring boot Data JPA (Java persistence API): helps to interact with Relational databases without writing much code.

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

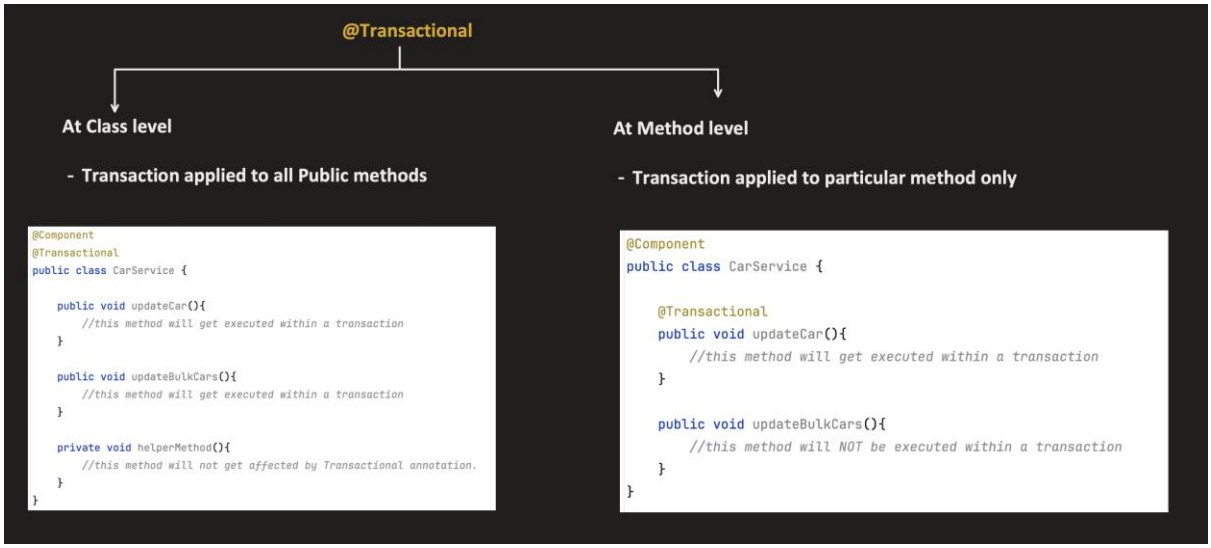
+

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

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

```java
@SpringBootApplication
@EnableTransactionManagement
public class SpringbootApplication {

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




```java
@Component
@Transactional
public class CarService {

    public void updateCar(){
        //this method will get executed within a transaction
    }

    public void updateBulkCars(){
        //this method will get executed within a transaction
    }

    private void helperMethod(){
        //this method will not get affected by Transactional annotation.
    }
}
```

```java
@Component
public class CarService {

    @Transactional
    public void updateCar(){
        //this method will get executed within a transaction
    }

    public void updateBulkCars(){
        //this method will NOT be executed within a transaction
    }
}
```

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.

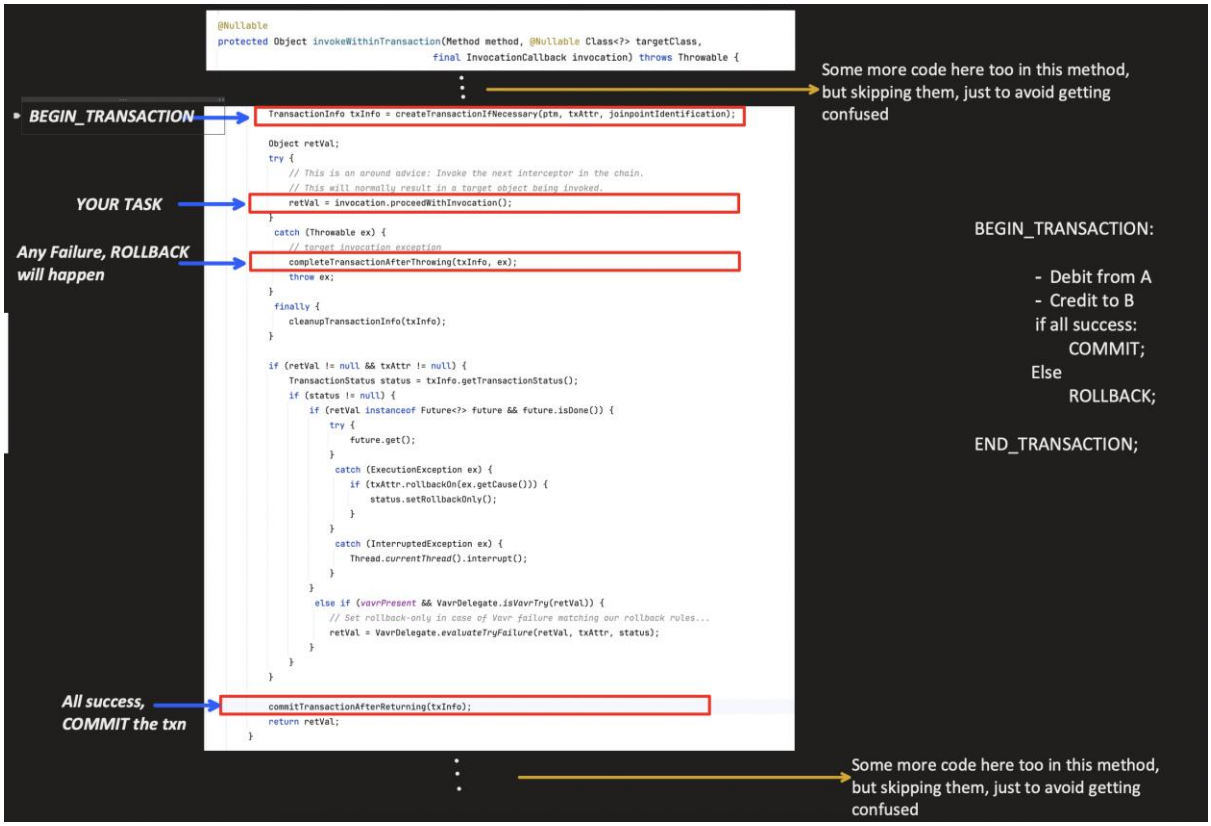
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    User user;

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

```java
@Component
public class User {

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


@Nullable
protected Object invokeWithinTransaction(Method method, @Nullable Class<?> targetClass,
final InvocationCallback invocation) throws Throwable {
Some more code here too in this method, but skipping them, just to avoid getting confused
BEGIN_TRANSACTION
TransactionInfo txInfo = createTransactionIfNecessary(ptm, txAttr, joinpointIdentification);
Object retVal;
try {
// This is an around advice: Invoke the next interceptor in the chain.
// This will normally result in a target object being invoked.
YOUR TASK
retVal = invocation.proceedWithInvocation();
}
catch (Throwable ex) {
// target invocation exception
Any Failure, ROLLBACK will happen
completeTransactionAfterThrowing(txInfo, ex);
throw ex;
}
finally {
cleanupTransactionInfo(txInfo);
}
if (retVal != null && txAttr != null) {
TransactionStatus status = txInfo.getTransactionStatus();
if (status != null) {
if (retVal 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 (vavrPresent && VavrDelegate.isVavrTry(retVal)) {
// Set rollback-only in case of Vavr failure matching our rollback rules...
retVal = VavrDelegate.evaluateTryFailure(retVal, txAttr, status);
}
}
}
All success, COMMIT the txn
commitTransactionAfterReturning(txInfo);
return retVal;
}
BEGIN_TRANSACTION:
- Debit from A
- Credit to B
if all success:
COMMIT;
Else
ROLLBACK;
END_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..