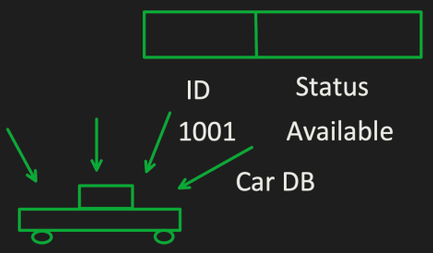
Critical Section

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.

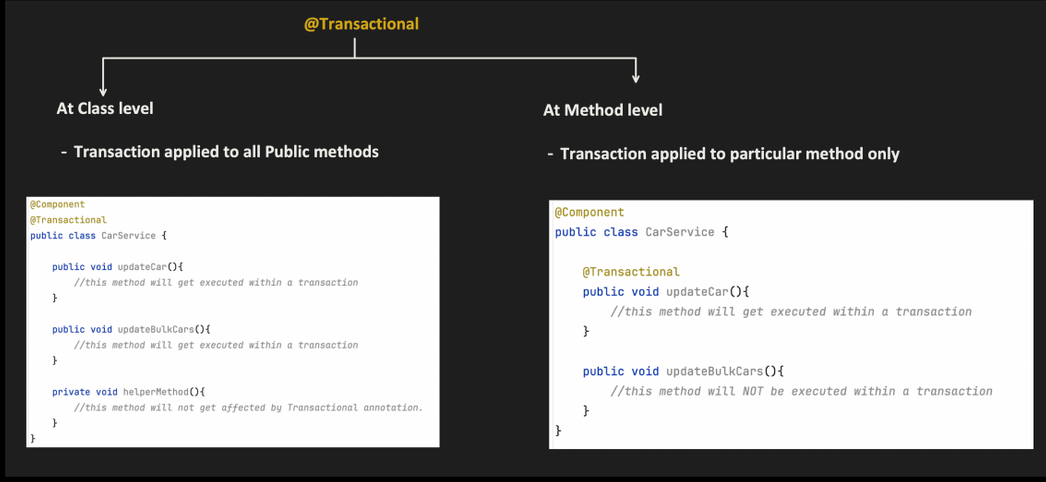
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

+

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)

```
@SpringBootApplication
@EnableTransactionManagement
public class SpringbootApplication {

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




```
@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.
    }
}
```

```
@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
    }
}
```

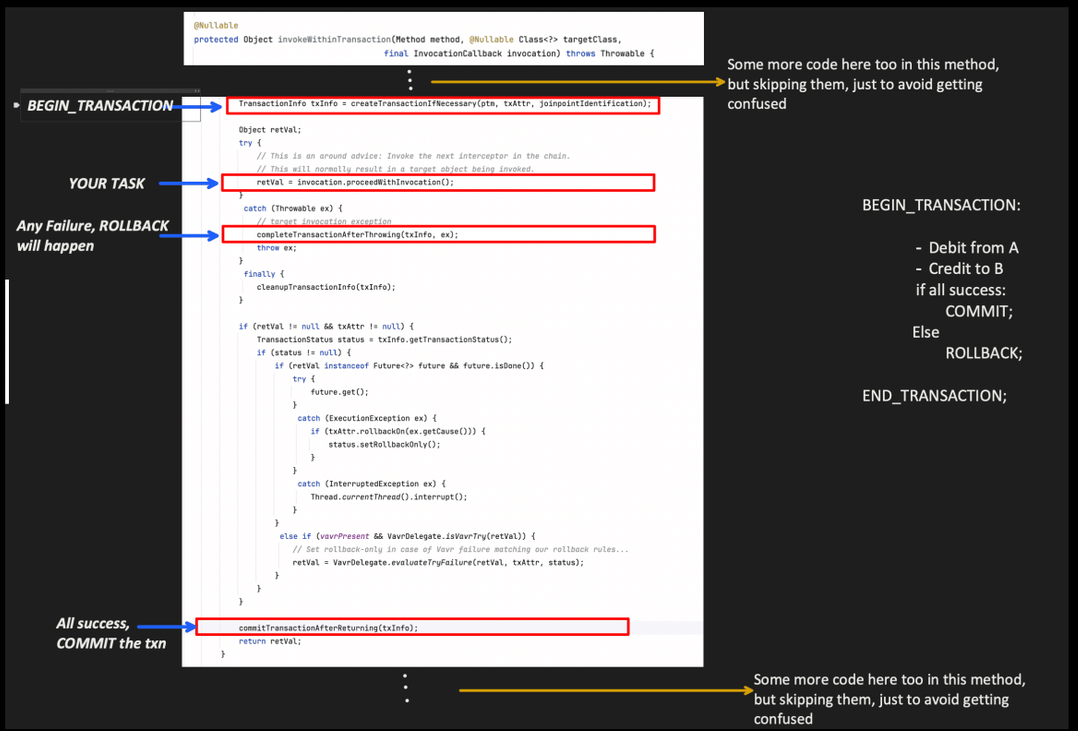
Transaction Management in Spring boot uses AOP.

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");
    }
}
```

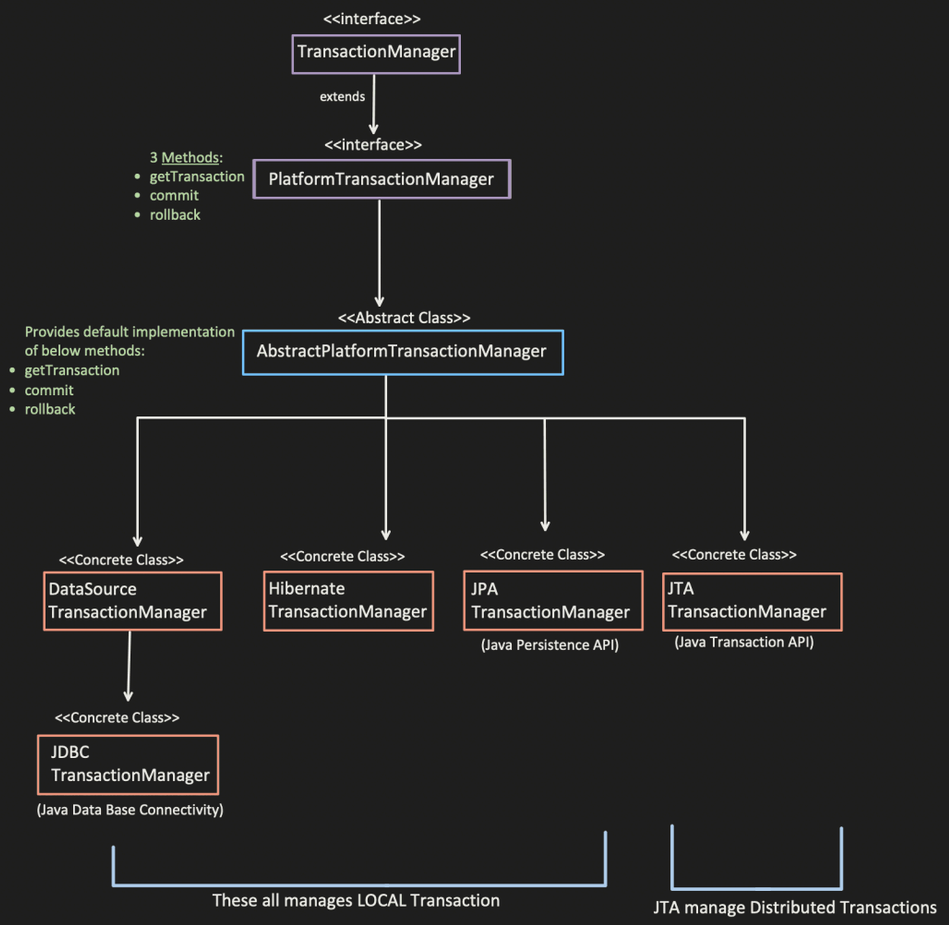


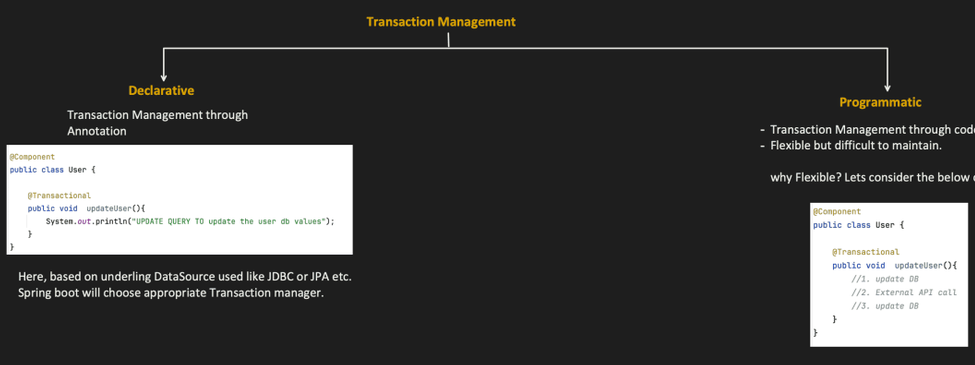
NOW, we know, how TRANSACTIONAL works, we will now see below topics in depth:

- **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..

Hierarchy of Transaction Managers





Declarative

Transaction Management through Annotation

```java
@Component
public class User {

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

Here, based on underling DataSource used like JDBC or JPA etc. Spring boot will choose appropriate Transaction manager.

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:testdb");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager userTransactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

```java
@Component
public class UserDeclarative {

    @Transactional(transactionManager = "userTransactionManager")
    public void updateUserProgrammatic() {
        //SOME DB OPERATIONS
        System.out.println("Insert Query ran");
        System.out.println("Update Query ran");
    }
}
```

2 ways to implement by Programmatic

Approach 1

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:testdb");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager userTransactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

```java
@Component
public class UserProgrammaticApproach1 {

    PlatformTransactionManager userTransactionManager;

    UserProgrammaticApproach1(PlatformTransactionManager userTransactionManager) {
        this.userTransactionManager = userTransactionManager;
    }

    public void updateUserProgrammatic(){
        TransactionStatus status = userTransactionManager.getTransaction( definition: null);
        try {
            //SOME INITIAL SET OF DB OPERATIONS
            System.out.println("Insert Query run1");
            System.out.println("Update Query run1");
            userTransactionManager.commit(status);
        }
        catch (Exception e) {
            userTransactionManager.rollback(status);
        }
    }
}
```

Approach 2

Usage of TransactionTemplate

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:testdb");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager userTransactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean
    public TransactionTemplate transactionTemplate(PlatformTransactionManager userTransactionManager) {
        return new TransactionTemplate(userTransactionManager);
    }
}
```

```java
@Component
public class UserProgrammaticApproach2 {

    TransactionTemplate transactionTemplate;

    public UserProgrammaticApproach2(TransactionTemplate transactionTemplate){
        this.transactionTemplate = transactionTemplate;
    }

    public void updateUserProgrammatic(){
        TransactionCallback<TransactionStatus> dbOperationsTask = (TransactionStatus status) -> {
            System.out.println("Insert Query ran");
            System.out.println("Update Query ran");
            return status;
        };
        TransactionStatus status = transactionTemplate.execute(dbOperationsTask);
    }

}
```

Now, lets see Propagation

When we try to create a new Transaction, it first check the PROPAGATION value set, and this tell whether we have to create new transaction or not.

- REQUIRED (default propagation):

  @Transactional(propagation=Propagation.*REQUIRED*)
  if(parent txn present)
  Use it;
  else
  Create new transaction;

- REQUIRED_NEW:

  @Transactional(propagation=Propagation.*REQUIRED_NEW*)
  if(parent txn present)
  Suspend the parent txn;
  Create a new Txn and once finished;
  Resume the parent txn;
  else
  Create new transaction and execute the method;

- SUPPORTS:

  @Transactional(propagation=Propagation.*SUPPORTS*)
  if(parent txn present)
  Use it;
  Else
  Execute the method without any transaction;

- NOT_SUPPORTED:

```
@Transactional(propagation=Propagation.NOT_SUPPORTED)
    if(parent txn present)
        Suspend the parent txn;
        Execute the method without any transaction;
        Resume the parent txn;
    else
        Execute the method without any transaction;
```

- MANDATORY:

```
@Transactional(propagation=Propagation.MANDATORY)
    if(parent txn present)
        Use it;
    Else
        Throw exception;
```

- NEVER:

```
@Transactional(propagation=Propagation.MANDATORY)
    if(parent txn present)
        Throw exception;
    Else
        Execute the method without any transaction;
```

Declarative way of usage:

```java
@Component
public class UserDeclarative {

    @Autowired
    UserDAO userDAObj;

    @Transactional
    public void updateUser() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());

        System.out.println("Some initial DB operation");
        userDAObj.dbOperationWithRequiredPropagation();
        System.out.println("Some final DB operation");
    }

    public void updateUserFromNonTransactionalMethod() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
        System.out.println("Some initial DB operation");
        userDAObj.dbOperationWithRequiredPropagation();
        System.out.println("Some final DB operation");
    }
}
```

```java
@Component
public class UserDAO {

    @Transactional(propagation = Propagation.REQUIRED)
    /**
     * if(parent txn present)
     *       use it
     * else
     *      create new
     *
     */
    public void dbOperationWithRequiredPropagation() {

        //EXECUTE DB QUERIES
        boolean isTransactionActive = TransactionSynchronizationManager.isActualTransactionActive();
        String currentTransactionName = TransactionSynchronizationManager.getCurrentTransactionName();
        System.out.println("*******************************");
        System.out.println("Propagation.REQUIRED: Is transaction active: " + isTransactionActive);
        System.out.println("Propagation.REQUIRED: Current transaction name: " + currentTransactionName);
        System.out.println("*******************************");

    }
}
```

Output:

```
Is transaction active: true
Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
Some initial DB operation
*******************************
Propagation.REQUIRED: Is parent transaction active: true
Propagation.REQUIRED: Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
*******************************
Some final DB operation
```

```
Is transaction active: false
Current transaction name: null
Some initial DB operation
*****************************
Propagation.REQUIRED: Is parent transaction active: true
Propagation.REQUIRED: Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDAO.dbOperationWithRequiredPropagation
*****************************
Some final DB operation
```

Programmatic way of usage:

(Approach 1)

```java
@Component
public class UserDeclarative {

    @Autowired
    UserDAO userDAOobj;

    @Transactional
    public void updateUser() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());

        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagationUsingProgrammaticApproach1();
        System.out.println("Some final DB operation");
    }

    public void updateUserFromNonTransactionalMethod() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagationUsingProgrammaticApproach1();
        System.out.println("Some final DB operation");
    }
}
```

```java
@Component
public class UserDAO {

    PlatformTransactionManager userTransactionManager;

    UserDAO(PlatformTransactionManager userTransactionManager) {
        this.userTransactionManager = userTransactionManager;
    }

    /**
     * if(parent txn present)
     *        use it
     * else
     *      create new
     */
    public void dbOperationWithRequiredPropagationUsingProgrammaticApproach1(){
        DefaultTransactionDefinition transactionDefinition = new DefaultTransactionDefinition();
        transactionDefinition.setName("Testing REQUIRED propagation");
        transactionDefinition.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
        TransactionStatus status = userTransactionManager.getTransaction(transactionDefinition);
        try {
            //EXECUTE operation
            System.out.println("******************************");
            System.out.println("Propagation.REQUIRED: Is  transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
            System.out.println("Propagation.REQUIRED: Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
            System.out.println("******************************");
            userTransactionManager.commit(status);
        }
        catch (Exception e) {
            userTransactionManager.rollback(status);
        }
    }
```

(Approach 2) : using TransactionTemplate

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:testdb");
        dataSource.setUsername("sa2");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager userTransactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean
    public TransactionTemplate transactionTemplate(PlatformTransactionManager userTransactionManager) {
        TransactionTemplate transactionTemplate = new TransactionTemplate(userTransactionManager);
        transactionTemplate.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
        transactionTemplate.setName("TRANSACTION TEMPLATE REQUIRED PROPAGATION");
        return transactionTemplate;
    }
}
```

```java
@Component
public class UserService {

    @Autowired
    UserDAO userDAOobj;

    @Transactional
    public void updateUser() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());

        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagationUsingProgrammaticApproach2();
        System.out.println("Some final DB operation");
    }

    public void updateUserFromNonTransactionalMethod() {
        System.out.println("Is transaction active: " +  TransactionSynchronizationManager.isActualTransactionActive());
        System.out.println("Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
        System.out.println("Some initial DB operation");
        userDAOobj.dbOperationWithRequiredPropagationUsingProgrammaticApproach2();
        System.out.println("Some final DB operation");
    }
}
```

```java
@Component
public class UserDAO {

    TransactionTemplate transactionTemplate;

    UserDAO(TransactionTemplate transactionTemplate) {
        this.transactionTemplate = transactionTemplate;
    }

    /**
     * if(parent txn present)
     * use it
     * else
     * create new
     */
    public void dbOperationWithRequiredPropagationUsingProgrammaticApproach2() {

        TransactionCallback<TransactionStatus> operations = (TransactionStatus status) -> {
            //some operations for this method
            System.out.println("*******************************");
            System.out.println("Propagation.REQUIRED: Is  transaction active: " + TransactionSynchronizationManager.isActualTransactionActive());
            System.out.println("Propagation.REQUIRED: Current transaction name: " + TransactionSynchronizationManager.getCurrentTransactionName());
            System.out.println("*******************************");
            return status;
        };

        TransactionStatus status = transactionTemplate.execute(operations);
    }
}
```

Output:

```
Is transaction active: true
Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
Some initial DB operation
*******************************
Propagation.REQUIRED: Is  transaction active: true
Propagation.REQUIRED: Current transaction name: com.conceptandcoding.learningspringboot.TransactionManagement.UserDeclarative.updateUser
*******************************
Some final DB operation
```

```
Is transaction active: false
Current transaction name: null
Some initial DB operation
*******************************
Propagation.REQUIRED: Is  transaction active: true
Propagation.REQUIRED: Current transaction name: TRANSACTION TEMPLATE REQUIRED PROPAGATION
*******************************
Some final DB operation
```

Isolation Level

Isolation level:
It tells, how the changes made by one transaction are visible to other transactions running in parallel.

```
@Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.READ_COMMITTED)
public void updateUser() {

    //some operations here
}
```

Isolation Level	Dirty Read Possible	Non-Repeatable Read Possible	Phantom Read Possible
READ_UNCOMMITTED	Yes	Yes	Yes
READ_COMMITTED	No	Yes	Yes
REPEATABLE_COMMITTED	No	No	Yes
SERIALIZABLE	No	No	No

↑ Concurrency High

↓ Concurrency Low

Default isolation level, depends upon the DB which we are using.
Like Most relational Databases uses READ_COMMITTED as default isolation, but again it depends upon DB to DB.

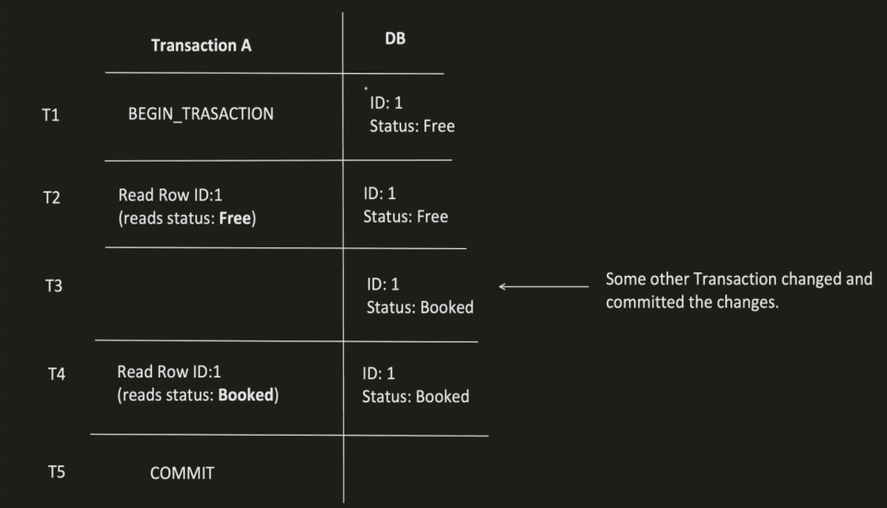
Dirty Read Problem

Transaction A reads the un-committed data of other transaction.
and
If other transaction get ROLLED BACK, the un-committed data which is read by Transaction A is known as Dirty Read.

Time	Transaction A	Transaction B	DB Status
T1	BEGIN_TRANSACTION	BEGIN_TRANSACTION	Id: 123 Status: free
T2		Update Row id:123 Status = booked	Id: 123 Status: booked (Not Committed by Transaction B yet)
T3	Read Row id:123 (Got status = booked)		Id: 123 Status: booked (Not Committed by Transaction B yet)
T4		Rollback	Id: 123 Status: Free (Un-committed changes of Txn B got Rolled Back)

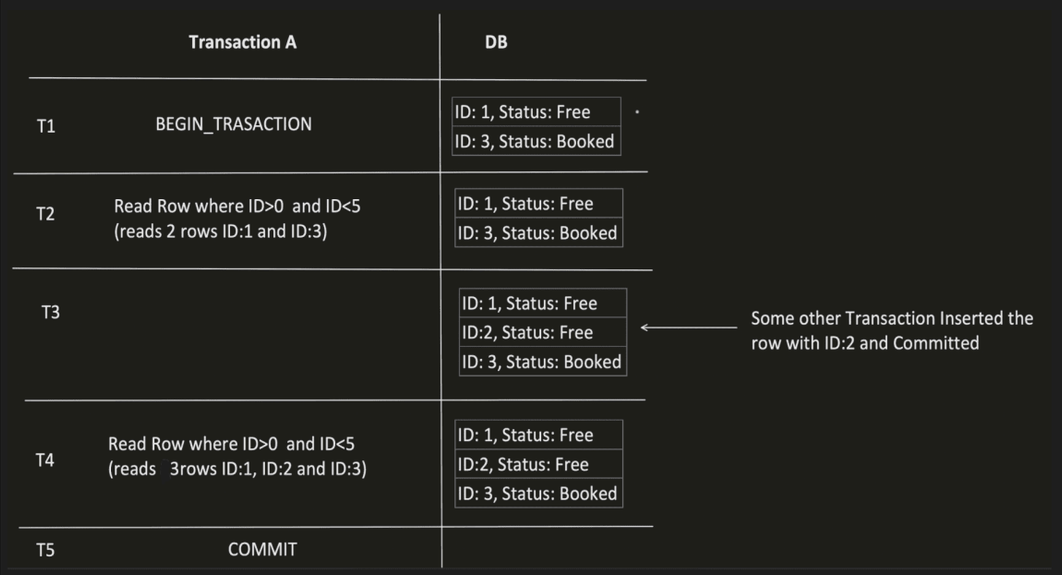
Non-Repeatable Read Problem

If suppose Transaction A, reads the same row several times and there is a chance that it get different value, then its known as Non-Repeatable Read problem.



Phantom Read Problem

If suppose Transaction A, executes same query several times but there is a chance that rows returned are different. Than its known as Phantom Read problem.



DB Locking Types

Locking make sure that, no other transaction update the locked rows.

Lock Type	Another Shared Lock	Another Exclusive Lock
Have Shared Lock	Yes	NO
Have Exclusive Lock	NO	NO

- Shared Lock (S) also knows as READ LOCK.
- Exclusive Lock(X) also know as WRITE LOCK.

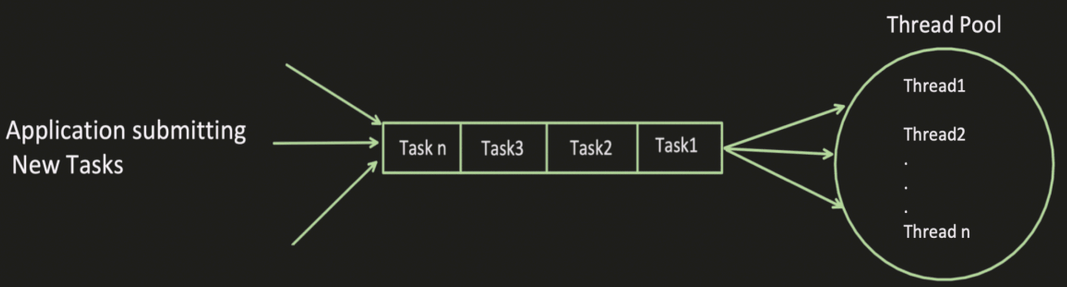
Let's see this table again:

Isolation Level	Dirty Read Possible	Non-Repeatable Read Possible	Phantom Read Possible
READ_UNCOMMITTED	Yes	Yes	Yes
READ_COMMITTED	No	Yes	Yes
REPEATABLE_COMMITTED	No	No	Yes
SERIALIZABLE	No	No	No

ISOLATION LEVEL	Locking Strategy
Read Uncommitted	**Read** : No Lock acquired **Write**: No Lock acquired
Read Committed	**Read**: Shared Lock acquired and Released as soon as Read is done **Write**: Exclusive Lock acquired and keep till the end of the transaction
Repeatable Read	**Read**: Shared Lock acquired and Released only at the end of the Transaction **Write**: Exclusive Lock acquired and Released only at the end of the Transaction
Serializable	Same as Repeatable Read Locking Strategy + apply Range Lock and lock is release only at the end of the Transaction.

What is ThreadPool:

- It's a collection of threads (aka workers), which are available to perform the submitted tasks.
- Once task completed, worker thread get back to Thread Pool and wait for new task to assigned.
- Means threads can be reused.



In Java, thread pool is created using ***ThreadPoolExecutor*** Object

```
int minPoolSize = 2;
int maxPoolSize = 4;
int queueSize = 3;

ThreadPoolExecutor poolTaskExecutor = new ThreadPoolExecutor(minPoolSize, maxPoolSize, keepAliveTime: 1,
        TimeUnit.HOURS, new ArrayBlockingQueue<>(queueSize));
```

Now lets start understanding, how @Async Annotation works

Async Annotation

- Used to mark method that should run asynchronously.
- Runs in a new thread, without blocking the main thread.

Example:

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

    @Autowired
    UserService userServiceObj;

    @GetMapping(path = "/getuser")
    public String getUserMethod(){
        System.out.println("inside getUserMethod: " + Thread.currentThread().getName() );
        userServiceObj.asyncMethodTest();
        return null;
    }
}
```

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

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

```java
@Component
public class UserService {

    @Async
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

Output:

```
2024-08-11T15:27:21.985+05:30  INFO 44004 --- [nio-8080-exec
2024-08-11T15:27:21.986+05:30  INFO 44004 --- [nio-8080-exec
inside getUserMethod: http-nio-8080-exec-1
inside asyncMethodTest: task-1
inside getUserMethod: http-nio-8080-exec-2
inside asyncMethodTest: task-2
inside getUserMethod: http-nio-8080-exec-3
inside asyncMethodTest: task-3
inside getUserMethod: http-nio-8080-exec-4
inside asyncMethodTest: task-4
inside getUserMethod: http-nio-8080-exec-5
inside asyncMethodTest: task-5
inside getUserMethod: http-nio-8080-exec-6
inside asyncMethodTest: task-6
inside getUserMethod: http-nio-8080-exec-7
inside asyncMethodTest: task-7
inside getUserMethod: http-nio-8080-exec-8
inside asyncMethodTest: task-8
```

Creating new thread, each time we run

So, how does this "Async" Annotation, creates a new thread?

Many places you will find, which says

Spring boot uses by default "***SimpleAsyncTaskExecutor***", which creates new thread every time.

I will say, this is not fully correct answer.

So, what's the right answer, What's the default Executor Spring boot uses?

If we see below Spring boot framework code, it first looks for *defaultExecutor*, if no *defaultExecutor* found, only then *SimpleAsyncTaskExecutor* is used.

AysncExecutionInterceptor.java

```
@Nullable
protected Executor getDefaultExecutor(@Nullable BeanFactory beanFactory) {
    Executor defaultExecutor = super.getDefaultExecutor(beanFactory);
    return (Executor) (defaultExecutor != null ? defaultExecutor : new SimpleAsyncTaskExecutor());
}
```

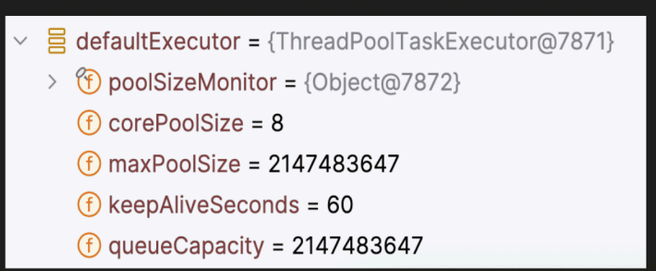
UseCase1:

```java
@Configuration
public class AppConfig {

}
```

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

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

```java
@Component
public class UserService {

    @Async
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

During Application startup, Spring boot sees that, no **ThreadPoolTaskExecutor** Bean present, so it creates its default "**ThreadPoolTaskExecutor**" with below configurations.



ThreadPoolTaskExecutor is nothing but a Spring boot Object, which is just a wrapper around Java ThreadPoolExecutor.

ThreadPoolTaskExecutor.java

```java
protected ExecutorService initializeExecutor(ThreadFactory threadFactory, RejectedExecutionHandler rejectedExecutionHandler) {
    BlockingQueue<Runnable> queue = this.createQueue(this.queueCapacity);
    ThreadPoolExecutor executor = new ThreadPoolExecutor(this.corePoolSize, this.maxPoolSize, (long) this.keepAliveSeconds, TimeUnit.SECONDS, queue, threadFactory, rejectedExecutionHandler) {
        public void execute(Runnable command) {...}

        protected void beforeExecute(Thread thread, Runnable task) {...}

        protected void afterExecute(Runnable task, Throwable ex) {...}
    };
    if (this.allowCoreThreadTimeOut) {...}

    if (this.prestartAllCoreThreads) {...}

    this.threadPoolExecutor = executor;
    return executor;
}
```

And its not recommended at all, why?

1. Underutilization of Threads: With Fixed Min pool size and Unbounded Queue(size is too big), its possible that most of the tasks will sit in the queue rather than creating new thread.

2. High Latency: Since queue size is too big, tasks will queue up till queue is not fill, high latency might occur during high load.

3. Thread Exhaustion: Lets say, if Queue also get filled up, then Executor will try to create new threads till Max pools size is not reached, which is Integer.MAX_VALUE. This can lead to thread exhaustion. And server might go down because of overhead of managing so many threads.

4. High Memory Usage: Each threads need some memory, when we are creating these many threads, which may consume large amount of memory too, which might lead to performance degradation too.

UseCase2: Creating our own custom, ThreadPoolTaskExecutor

```java
@Configuration
public class AppConfig {

    @Bean(name = "myThreadPoolExecutor")
    public Executor taskPoolExecutor() {

        int minPoolSize = 2;
        int maxPoolSize = 4;
        int queueSize = 3;

        ThreadPoolTaskExecutor poolTaskExecutor = new ThreadPoolTaskExecutor();
        poolTaskExecutor.setCorePoolSize(minPoolSize);
        poolTaskExecutor.setMaxPoolSize(maxPoolSize);
        poolTaskExecutor.setQueueCapacity(queueSize);
        poolTaskExecutor.setThreadNamePrefix("MyThread-");
        poolTaskExecutor.initialize();
        return poolTaskExecutor;
    }
}
```

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

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

```java
@Component
public class UserService {

    @Async
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

or

```java
@Component
public class UserService {

    @Async("myThreadPoolExecutor")
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

During Application startup, Spring boot sees that, **ThreadPoolTaskExecutor** Bean present, so it makes it default only.

And even when we use @Async without any name, our "***myThreadPoolExecutor***" will get picked only.

Output:

```
2024-08-11T17:05:14.403+05:30  INFO 47918 --- [nio-8080-exec-1] o.s.web.servlet.
inside getUserMethod: http-nio-8080-exec-1
inside asyncMethodTest: MyThread-1
inside getUserMethod: http-nio-8080-exec-2
inside asyncMethodTest: MyThread-2
inside getUserMethod: http-nio-8080-exec-3
inside asyncMethodTest: MyThread-1
inside getUserMethod: http-nio-8080-exec-4
inside asyncMethodTest: MyThread-2
inside getUserMethod: http-nio-8080-exec-5
inside asyncMethodTest: MyThread-1
```

Output: (after putting sleep in async method, to simulate load)

```
@Component
public class UserService {

    @Async("myThreadPoolExecutor")
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
        try {
            Thread.sleep( millis: 50000);
        }catch (Exception e){

        }
    }
}
```

```
2024-08-11T20:43:02.142+05:30  INFO 49592 --- [nio-8080-exec-1]
inside getUserMethod: http-nio-8080-exec-1
inside asyncMethodTest: MyThread-1
inside getUserMethod: http-nio-8080-exec-2
inside asyncMethodTest: MyThread-2
inside getUserMethod: http-nio-8080-exec-3
inside getUserMethod: http-nio-8080-exec-4
inside getUserMethod: http-nio-8080-exec-5
inside getUserMethod: http-nio-8080-exec-6
inside asyncMethodTest: MyThread-3
inside getUserMethod: http-nio-8080-exec-7
inside asyncMethodTest: MyThread-4
inside getUserMethod: http-nio-8080-exec-8
java.util.concurrent.RejectedExecutionException Create breakpoint
```

Min Pool threads used

Queue got full

New Threads created, till Max Pool Capacity

Any New Request got Rejected

Its recommended, as its solve all the issues existing with the previous use case

UseCase3: Creating our own custom, ThreadPoolExecutor (java one)

```
@Configuration
public class AppConfig implements AsyncConfigurer {

    @Bean(name = "myThreadPoolExecutor")
    public Executor taskPoolExecutor() {

        int minPoolSize = 2;
        int maxPoolSize = 4;
        int queueSize = 3;

        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(minPoolSize, maxPoolSize,
                keepAliveTime: 1, TimeUnit.HOURS, new ArrayBlockingQueue<>(queueSize), new CustomThreadFactory());

        return poolExecutor;
    }

    class CustomThreadFactory implements ThreadFactory {

        private final AtomicInteger threadNo = new AtomicInteger( initialValue: 1);
        @Override
        public Thread newThread(Runnable r) {
            Thread thread = new Thread(r);
            thread.setName("MyThread-" + threadNo.getAndIncrement());
            return thread;
        }
    }
}
```

```
@SpringBootApplication
@EnableAsync
public class SpringbootApplication {

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

```
@Component
public class UserService {

    @Async
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

During Application startup, Spring boot sees that, **ThreadPoolExecutor (java one)** Bean is present, so it do not create its own default ***ThreadPoolTaskExecutor (spring wrapper one)***
instead it set the default executor is "***SimpleAsyncTaskExecutor***"

Now, if we run the above code, what we see.

```
2024-08-11T23:53:52.093+05:30  INFO 58643 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
inside getUserMethod: http-nio-8080-exec-1
2024-08-11T23:53:52.124+05:30  INFO 58643 --- [nio-8080-exec-1] .s.a.AnnotationAsyncExecutionInterceptor
inside asyncMethodTest: SimpleAsyncTaskExecutor-1
inside getUserMethod: http-nio-8080-exec-2
inside asyncMethodTest: SimpleAsyncTaskExecutor-2
inside getUserMethod: http-nio-8080-exec-3
inside asyncMethodTest: SimpleAsyncTaskExecutor-3
inside getUserMethod: http-nio-8080-exec-4
inside asyncMethodTest: SimpleAsyncTaskExecutor-4
inside getUserMethod: http-nio-8080-exec-5
inside asyncMethodTest: SimpleAsyncTaskExecutor-5
inside getUserMethod: http-nio-8080-exec-6
inside asyncMethodTest: SimpleAsyncTaskExecutor-6
inside getUserMethod: http-nio-8080-exec-7
inside asyncMethodTest: SimpleAsyncTaskExecutor-7
inside getUserMethod: http-nio-8080-exec-8
inside asyncMethodTest: SimpleAsyncTaskExecutor-8
inside getUserMethod: http-nio-8080-exec-9
inside asyncMethodTest: SimpleAsyncTaskExecutor-9
```

And its not recommended at all to use ***"SimpleAsyncTaskExecutor"***, why?

It just creates new thread every time. So it may lead to

1. Thread Exhaustion: just blindly creating new thread with every Async request, might lead up to thread exhaustion.

2. Thread Creation Overhead: Since Threads are not reused, so thread management (creation, destroying) is an additional overhead.

3. High Memory Usage: Each threads need some memory, when we are creating these many threads, which may consume large amount of memory too, which might lead to performance degradation too.

So, whenever we have defined our own **ThreadPoolExecutor** (Java one), always specify the name also with Async annotation.

```java
@Configuration
public class AppConfig implements AsyncConfigurer {

    @Bean(name = "myThreadPoolExecutor")
    public Executor taskPoolExecutor() {

        int minPoolSize = 2;
        int maxPoolSize = 4;
        int queueSize = 3;

        ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(minPoolSize, maxPoolSize,
                keepAliveTime: 1, TimeUnit.HOURS, new ArrayBlockingQueue<>(queueSize), new CustomThreadFactory());

        return poolExecutor;
    }

    class CustomThreadFactory implements ThreadFactory {

        private final AtomicInteger threadNo = new AtomicInteger( initialValue: 1);
        @Override
        public Thread newThread(Runnable r) {
            Thread thread = new Thread(r);
            thread.setName("MyThread-" + threadNo.getAndIncrement());
            return thread;
        }
    }
}
```

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

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

```java
@Component
public class UserService {

    @Async("myThreadPoolExecutor")
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

Output:

```
2024-08-11T17:05:14.403+05:30  INFO 47918 --- [nio-8080-exec-1] o.s.web.servlet.
inside getUserMethod: http-nio-8080-exec-1
inside asyncMethodTest: MyThread-1
inside getUserMethod: http-nio-8080-exec-2
inside asyncMethodTest: MyThread-2
inside getUserMethod: http-nio-8080-exec-3
inside asyncMethodTest: MyThread-1
inside getUserMethod: http-nio-8080-exec-4
inside asyncMethodTest: MyThread-2
inside getUserMethod: http-nio-8080-exec-5
inside asyncMethodTest: MyThread-1
```

Hey, I don't want all this confusion, Usecase1, Usecase2 or Usecase3.

I always want to set my executor as default one, even if anyone use @Async, my executor only should be picked.

```java
@Configuration
public class AppConfig implements AsyncConfigurer {

    private ThreadPoolExecutor poolExecutor;
    @Override
    public synchronized Executor getAsyncExecutor() {

        if (poolExecutor == null) {
            int minPoolSize = 2;
            int maxPoolSize = 4;
            int queueSize = 3;
            poolExecutor = new ThreadPoolExecutor(minPoolSize, maxPoolSize, keepAliveTime: 1,
                    TimeUnit.HOURS, new ArrayBlockingQueue<>(queueSize), new CustomThreadFactor());
        }
        return poolExecutor;
    }
}

class CustomThreadFactor implements ThreadFactory {
    private final AtomicInteger threadNumber = new AtomicInteger( initialValue: 1);
    @Override
    public Thread newThread(Runnable r) {
        Thread th = new Thread(r);
        th.setName("MyThread-" + threadNumber.getAndIncrement());
        return th;
    }
}
```

Using Java ThreadPoolExecutor

```java
@Component
public class UserService {

    @Async
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

No, Executor name provided

Still, default executor configuration picked is mine one, not SimpleAsycTaskExecutor

Output:

```
2024-08-12T00:11:42.078+05:30  INFO 58958 --- [nio-8080-exec-1]
inside getUserMethod: http-nio-8080-exec-1
inside asyncMethodTest: MyThread-1
inside getUserMethod: http-nio-8080-exec-2
inside asyncMethodTest: MyThread-2
inside getUserMethod: http-nio-8080-exec-3
inside asyncMethodTest: MyThread-1
inside getUserMethod: http-nio-8080-exec-4
inside asyncMethodTest: MyThread-2
inside getUserMethod: http-nio-8080-exec-5
inside asyncMethodTest: MyThread-1
```

Conditions for @Async Annotation to work properly?

1. Different Class :
   If @Async annotation is applied to the method within the same class from which it is being called, then Proxy mechanism is skipped because internal method calls are **NOT INTERCEPTED**.

2. Public method:
   Method annotated with @Async must be public. And again, AOP interception works only on Public methods.

Both in same class, Proxy will get bypassed

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

    @GetMapping(path = "/getuser")
    public String getUserMethod(){
        System.out.println("inside getUserMethod: " + Thread.currentThread().getName() );
        asyncMethodTest();
        return null;
    }

    @Async
    public void asyncMethodTest() {
        System.out.println("inside asyncMethodTest: " + Thread.currentThread().getName());
    }
}
```

This is wrong way of doing it, correct way is, @Async method should be in different class and should be public

Output:

```
2024-08-18T12:46:04.532+05:30  INFO 60618 ---
inside getUserMethod: http-nio-8080-exec-1
inside asyncMethodTest: http-nio-8080-exec-1
```

@Aysn and Transaction Management

Usecase1:  **Transaction Context do not transfer from caller thread to new thread which got created by Async.**

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

    @Autowired
    UserService userService;

    @PostMapping(path = "/updateuser")
    public String updateUserMethod(){
        userService.updateUser();
        return null;
    }
}
```

```
@Component
public class UserService {

    @Autowired
    UserUtility userUtility;

    @Transactional
    public void updateUser(){

        //1. update user status
        //2. update user first name

        //3. update user
        userUtility.updateUserBalance();
    }
}
```

```
@Component
public class UserUtility {

    @Async
    public void updateUserBalance(){
        //updating user balance amount.
    }
}
```

Usecase2: **Use with Precaution, as new thread will be created and have transaction management too but context is not same as parent thread. So Propagation will not work as expected.**

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

    @Autowired
    UserService userService;

    @PostMapping(path = "/updateuser")
    public String updateUserMethod(){
        userService.updateUser();
        return null;
    }
}
```

```java
@Component
public class UserService {

    @Transactional
    @Async
    public void updateUser(){

        //1. update user status
        //2. update user first name
        //3. update user
    }
}
```

Usecase3: ✓

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

    @Autowired
    UserService userService;

    @PostMapping(path = "/updateuser")
    public String updateUserMethod(){
        userService.updateUser();
        return null;
    }
}
```

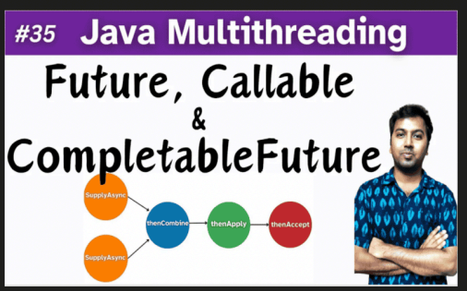
```java
@Component
public class UserService {

    @Autowired
    UserUtility userUtility;

    @Async
    public void updateUser(){
       userUtility.updateUser();
    }
}
```

```java
@Component
public class UserUtility {

    @Transactional
    public void updateUser(){
        //1. update user status
        //2. update user first name
        //3. update user
    }
}
```

@Aysn Method return type

Both Future and Completable Future can be the return type of the Async method

Checkout Java Playlist to learn more in depth of Future and CompletableFuture



Using Future:

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

    @Autowired
    UserService userService;

    @GetMapping(path = "/getuser")
    public String getUserMethod(){
       Future<String> result =  userService.performTaskAsync();
        String output = null;
       try {
           output = result.get();
           System.out.println(output);
       }catch (Exception e) {
            System.out.println("some exception");
       }
        return output;
    }
}
```

```java
@Component
public class UserService {

    @Async
    public Future<String> performTaskAsync(){
        return new AsyncResult<>( value: "async task result");
    }
}
```

S.No.	Method Available in Future Interface	Purpose
1.	boolean cancel(boolean mayInterruptIfRunning)	• Attempts to cancel the execution of the task. • Returns false, if task can not be cancelled (typically bcoz task already completed); returns true otherwise.
2.	boolean isCancelled()	• Returns true, if task was cancelled before it get completed.
3.	boolean isDone()	• Returns true if this task completed. Completion may be due to normal termination, an exception, or cancellation -- in all of these cases, this method will return true.
4.	V get()	• Wait if required, for the completion of the task. • After task completed, retrieve the result if available.
5.	V get(long timeout, TimeUnit unit)	• Wait if required, for at most the given timeout period. • Throws 'TimeoutException' if timeout period finished and task is not yet completed.

Using CompletableFuture:

Introduced in Java8

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

    @Autowired
    UserService userService;

    @GetMapping(path = "/getuser")
    public String getUserMethod(){
       CompletableFuture<String> result =  userService.performTaskAsync();
        String output = null;
       try {
           output = result.get();
           System.out.println(output);
       }catch (Exception e) {
            System.out.println("some exception");
       }
        return output;
    }
}
```

```java
@Component
public class UserService {

    @Async
    public CompletableFuture<String> performTaskAsync(){
        return CompletableFuture.completedFuture( value: "async task result");
    }
}
```

Exception Handling

Method which has Return type

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

    @Autowired
    UserService userService;

    @GetMapping(path = "/getuser")
    public String getUserMethod(){
        CompletableFuture<String> result =  userService.performTaskAsync();
        String output = null;
        try {
            output = result.get();
            System.out.println(output);
        }catch (Exception e) {
            System.out.println("some exception");
        }
        return output;
    }
}
```

During Get call, Exception is thrown and we can catch it and handled accordingly

```
@Component
public class UserService {

    @Async
    public CompletableFuture<String> performTaskAsync(){
        return CompletableFuture.completedFuture( value: "async task result");
    }
}
```

Method which do not return anything

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

    @Autowired
    UserService userService;

    @GetMapping(path = "/getuser")
    public String getUserMethod(){
        userService.performTaskAsync();
        return "";
    }
}
```

```
@Component
public class UserService {

    @Async
    public void performTaskAsync(){
        //perform some task
    }
}
```

How to handle this?

1. Within Async Method itself

```
@Component
public class UserService {

    @Async
    public void performTaskAsync(){
        try {
            //perform some task
        } catch (Exception e) {
            //hanlde the exception here
        }
    }
}
```

2. Implement Custom AsyncExceptionHandler

```java
@Configuration
public class AppConfig implements AsyncConfigurer {

    @Autowired
    private AsyncUncaughtExceptionHandler asyncUncaughtExceptionHandler;

    @Override
    public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
        return this.asyncUncaughtExceptionHandler;
    }
}

@Component
class DefaultAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler {

    @Override
    public void handleUncaughtException(Throwable ex, Method method, Object... params) {
        System.out.println("in default Uncaugh Exception method");
        //logging can be done here.
    }
}
```

```java
@Component
public class UserService {

    @Async
    public void performTaskAsync(){
        int i = 0;
        System.out.println(5/i);
    }
}
```

Output:

```
2024-08-18T20:15:14.082+05:30  INFO 70455 --- [nio-8080-exec-1]
in default Uncaugh Exception method
```

If , we will not handle it, then, Spring boot default SimpleAsyncUncaughtExceptionHandler will get invoked

```java
@Configuration
public class AppConfig  {

}
```

```java
@Component
public class UserService {

    @Async
    public void performTaskAsync(){
        int i = 0;
        System.out.println(5/i);
    }
}
```

Spring boot framework code..

```java
public class SimpleAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler {
    private static final Log logger = LogFactory.getLog(SimpleAsyncUncaughtExceptionHandler.class);

    public SimpleAsyncUncaughtExceptionHandler() {
    }

    public void handleUncaughtException(Throwable ex, Method method, Object... params) {
        if (logger.isErrorEnabled()) {
            logger.error( message: "Unexpected exception occurred invoking async method: " + method, ex);
        }

    }
}
```

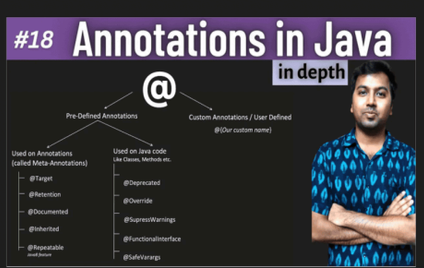
Output:

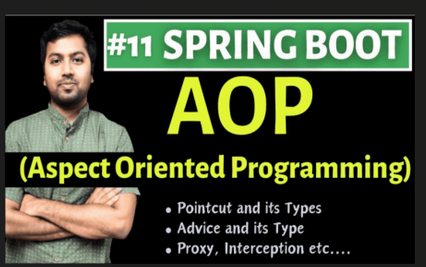
```
 task-5] .a.i.SimpleAsyncUncaughtExceptionHandler : Unexpected exception occurred invoking async method:

java.lang.ArithmeticException Create breakpoint : / by zero
    at com.conceptandcoding.learningspringboot.AsyncAnnotationLearn.UserService.performTaskAsync(UserService.java:18)
```

Interceptor:
it's a mediator, which get invoked before or after your actual code.

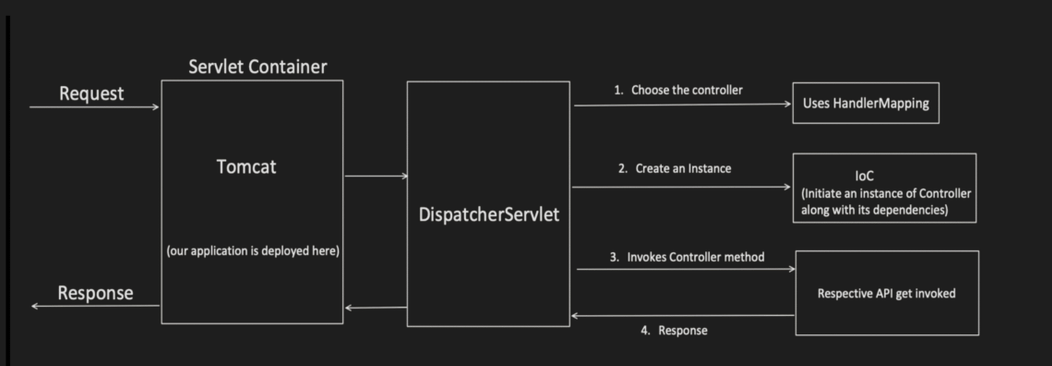
In future topics below, we might need to write our custom interceptors:
- *Springboot Caching,*
- *Springboot logging,*
- *Springboot Authentication etc..*

Pre-requisite to understand custom interceptor better:





Custom Interceptor for Requests before even reaching to specific Controller class



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

    @Autowired
    User user;

    @GetMapping(path = "/getUser")
    public String getUser() {
        user.getUser();
        return "success";
    }
}
```

```java
@Component
public class MyCustomInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        System.out.println("inside pre handle Method");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable ModelAndView modelAndView) throws Exception {
        System.out.println("inside post handle method");

    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable Exception ex) throws Exception {
        System.out.println("inside after completion method");
    }
}
```

```java
@Configuration
public class AppConfig implements WebMvcConfigurer {

    @Autowired
    MyCustomInterceptor myCustomInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(myCustomInterceptor)
                .addPathPatterns("/api/*") // Apply to these URL patterns
                .excludePathPatterns("/api/updateUser", "/api/deleteUser"); // Exclude for these URL patterns
    }
}
```

http://localhost:8080/api/getUser

```
2024-08-31T23:01:49.217+05:30  INFO 4417 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
inside pre handle Method
hitting db to get the userdata
inside post handle method
inside after completion method
```

Custom Interceptor for Requests after reaching to specific Controller class

Step1: Creation of custom annotation

We can create Custom Annotation using keyword *"@interface"* java Annotation.

```java
public @interface MyCustomAnnotation {
}
```

```java
public class User {

    @MyCustomAnnotation
    public void updateUser(){
        //some business logic
    }
}
```

2 Important Meta Annotation properties are:

- @Target :
  this meta annotation, tells where we can apply the particular annotation on method or class or constructor etc.

```java
@Target(ElementType.METHOD)
public @interface MyCustomAnnotation {
}
```

```java
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD})
public @interface MyCustomAnnotation {
}
```

@Retention:
this meta annotation tell, how the particular annotation will be stored in java.

- RetentionPolicy.*SOURCE :*
  Annotation will be discarded by compiler itself and its not even recorded in .class file.

  RetentionPolicy.*CLASS :*
  Annotation will be recorded in .class file but ignored by JVM during run time.

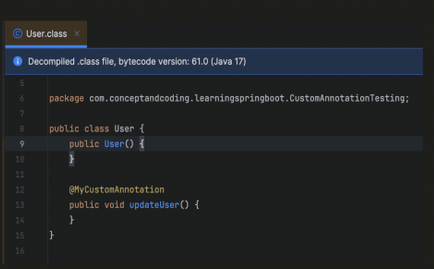
  RetentionPolicy.*RUNTIME :*
  Annotation will be recorded in .class file and also available during run time.

```java
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.SOURCE)
public @interface MyCustomAnnotation {
}
```

```java
public class User {

    @MyCustomAnnotation
    public void updateUser(){
        //some business logic
    }
}
```


```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation {
}
```

```java
public class User {

    @MyCustomAnnotation
    public void updateUser(){
        //some business logic
    }
}
```



How to create Custom Annotation with methods (more like a fields):

- No parameter, no body
- Return type is restricted to:
  - Primitive type (int, boolean, double etc.)
  - String
  - Enum
  - Class<?>
  - Annotations
  - Array of above types

```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation {

    String key() default "defaultKeyName";
}
```

```java
public class User {

    @MyCustomAnnotation(key = "userKey")
    public void updateUser(){
        //some business logic
    }
}
```

```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation {

    int intKey() default 0;

    String stringKey() default "defaultString";

    Class<?> classTypeKey() default String.class;

    MyCustomEnum enumKey() default MyCustomEnum.ENUM_VAL1;

    String[] stringArrayKey() default {"default1", "default2"};

    int[] intArrayKey() default {1, 2};

}
```

```java
public class User {

    @MyCustomAnnotation(intKey = 10, stringKey = "user", classTypeKey = User.class, enumKey = MyCustomEnum.ENUM_VAL2)
    public void updateUser(){
        //some business logic
    }
}
```

Step2: Creation of Custom Interceptor

```java
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation {

    String name() default "";

}
```

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

    @Autowired
    User user;

    @GetMapping(path = "/getUser")
    public String getUser(){
        user.getUser();
        return "success";
    }
}
```

```java
@Component
public class User {

    @MyCustomAnnotation(name = "user")
    public void getUser() {
        System.out.println("get the user details");
    }
}
```

```java
@Component
@Aspect
public class MyCustomInterceptor {

    @Around("@annotation(com.conceptandcoding.learningspringboot.CustomInterceptor.MyCustomAnnotation)") //pointcut expression
    public void invoke(ProceedingJoinPoint joinPoint) throws Throwable{ //advice

        System.out.println("do something before actual method");

        Method method = ((MethodSignature)joinPoint.getSignature()).getMethod();
        if(method.isAnnotationPresent(MyCustomAnnotation.class)){
            MyCustomAnnotation annotation = method.getAnnotation(MyCustomAnnotation.class);
            System.out.println("name from annotation: " + annotation.name());
        }

        joinPoint.proceed();
        System.out.println("do something after actual method");
    }
}
```

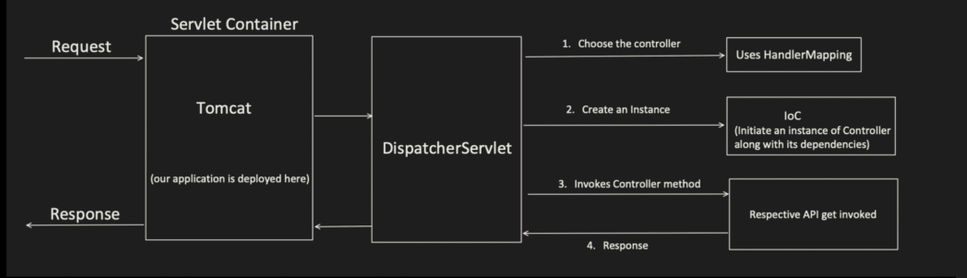
Output:

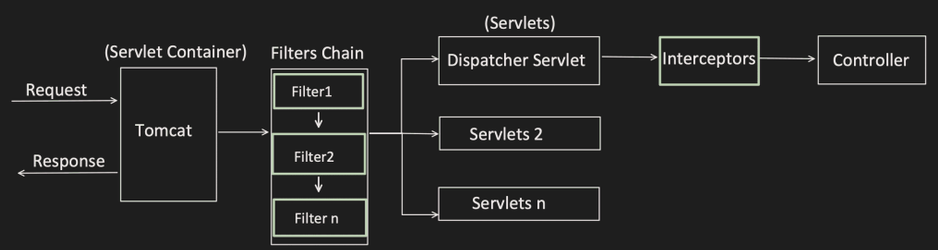
```
2024-08-31T17:51:51.344+05:30  INFO 3464 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
do something before actual method
name from annotation: user
get the user details
do something after actual method
```

Filter:
It intercept the HTTP Request and Response, before they reach to the servlet.

Interceptor:
Its specific to Spring framework, and intercept HTTP Request and Response, before they reach to the Controller.

Lets see this diagram again:





What is servlet:

Servlet is nothing but a Java class, which accepts the incoming request, process it and returns the response.

We can create multiple servlets like:

Servlet 1: can be configured to handle REST APIs
Servlet 2 : can be configured to handle SOAP APIs etc....

Similarly like this, **"DispatcherServlet"** is kind of servlet provided by spring, and by default its configured to handle all APIs "/*".

Filter:
Is used when we want to intercept HTTP Request and Response and add logic agnostic of the underlying servlets.
We can have many filters and have ordering between them too.

Interceptors:
Is used when we want to intercept HTTP request and response and add logic specific to a particular servlet.
We can have many Interceptors and have ordering between them too.

Multiple Interceptors and its Ordering:

In previous video, we already saw, how to add 1 interceptor.
How "preHandle" , "postHandle" and "afterCompletion" comes into the picture.

Now here, will show how to add more than 1.

Also, if "preHandle" returns false, next interceptor and controller will not get invoked itself.

```
@Configuration
public class AppConfig implements WebMvcConfigurer {

    @Autowired
    MyCustomInterceptor1 myCustomInterceptor1;

    @Autowired
    MyCustomInterceptor2 myCustomInterceptor2;


    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //below order is maintained while calling interceptors
        registry.addInterceptor(myCustomInterceptor1)
                .addPathPatterns("/api/*") // Apply to these URL patterns
                .excludePathPatterns("/api/updateUser", "/api/deleteUser"); // Exclude for these URL patterns

        registry.addInterceptor(myCustomInterceptor2)
                .addPathPatterns("/api/*") // Apply to these URL patterns
                .excludePathPatterns("/api/updateUser");
    }
}
```

Output:

```
2024-09-02T17:01:11.719+05:30  INFO 7162 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
2024-09-02T17:01:11.720+05:30  INFO 7162 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
inside pre handle Method - MyCustomInterceptor1
inside pre handle Method - MyCustomInterceptor2
hitting db to get the userdata
inside post handle method- MyCustomInterceptor2
inside post handle method- MyCustomInterceptor1
inside after completion method - MyCustomInterceptor2
inside after completion method- MyCustomInterceptor1
```

How to Add Filters:

```java
import jakarta.servlet.*;
import java.io.IOException;

public class MyFilter1 implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        Filter.super.init(filterConfig);
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
            throws IOException, ServletException {

        System.out.println("MyFilter1 inside");
        filterChain.doFilter(servletRequest, servletResponse);
        System.out.println("MyFilter1 completed");
    }

    @Override
    public void destroy() {
        Filter.super.destroy();
    }
}
```

```java
import jakarta.servlet.*;
import java.io.IOException;

public class MyFilter2 implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        Filter.super.init(filterConfig);
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
            throws IOException, ServletException {

        System.out.println("MyFilter2 inside");
        filterChain.doFilter(servletRequest, servletResponse);
        System.out.println("MyFilter2 completed");
    }

    @Override
    public void destroy() {
        Filter.super.destroy();
    }
}
```

```java
@Configuration
public class AppConfig {

    @Bean
    public FilterRegistrationBean<MyFilter1> myFirstFilter() {
        FilterRegistrationBean<MyFilter1> filterRegistrationBean = new FilterRegistrationBean<>();
        filterRegistrationBean.setFilter(new MyFilter1());
        filterRegistrationBean.addUrlPatterns("/*");
        filterRegistrationBean.setOrder(2);
        return filterRegistrationBean;
    }

    @Bean
    public FilterRegistrationBean<MyFilter2> mySecondFilter() {
        FilterRegistrationBean<MyFilter2> filterRegistrationBean = new FilterRegistrationBean<>();
        filterRegistrationBean.setFilter(new MyFilter2());
        filterRegistrationBean.addUrlPatterns("/*");
        filterRegistrationBean.setOrder(1);
        return filterRegistrationBean;
    }
}
```

Output:

```
2024-09-02T17:30:39.279+05:30  INFO 7350 --- [nio-8080-exec-1]
2024-09-02T17:30:39.280+05:30  INFO 7350 --- [nio-8080-exec-1]
MyFilter2 inside
MyFilter1 inside
hitting db to get the userdata
MyFilter1 completed
MyFilter2 completed
```

If both Interceptor and Filter used together, Output will look like this.

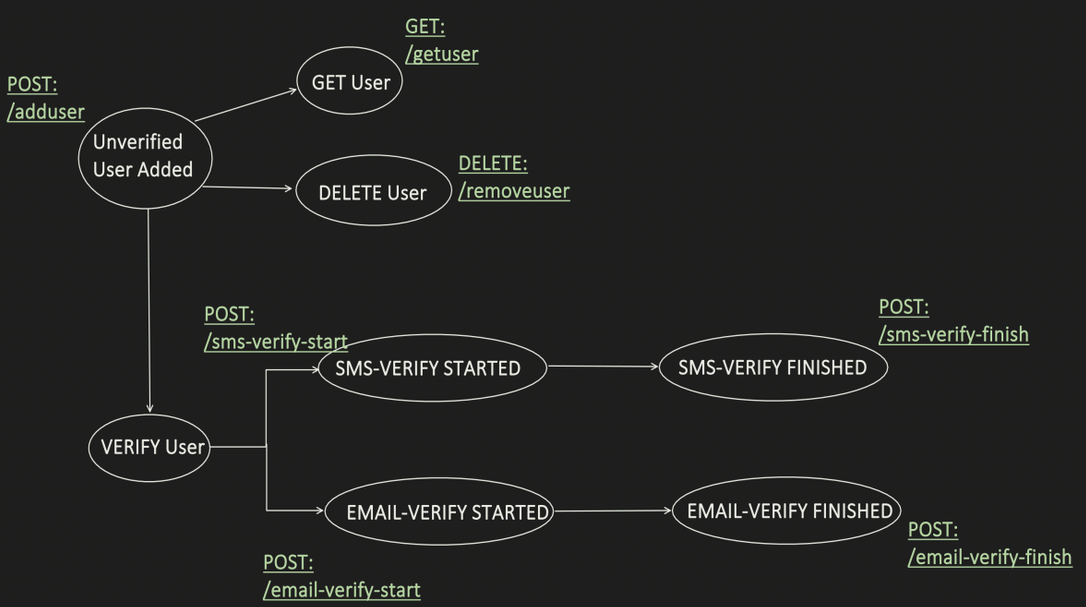
```
2024-09-02T17:32:25.643+05:30  INFO 7373 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
2024-09-02T17:32:25.644+05:30  INFO 7373 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
MyFilter1 inside
MyFilter2 inside
inside pre handle Method - MyCustomInterceptor1
inside pre handle Method - MyCustomInterceptor2
hitting db to get the userdata
inside post handle method- MyCustomInterceptor2
inside post handle method- MyCustomInterceptor1
inside after completion method - MyCustomInterceptor2
inside after completion method- MyCustomInterceptor1
MyFilter2 completed
MyFilter1 completed
```

HATEOAS

Hypermedia As The Engine Of Application State

It tells the client, what the next action you can perform on particular item.

For example:



API Response, after *"Unverified User Added"* POST: /adduser API

Without *HATEOAS* link

```
{
    "userID": "123456",
    "name": "SJ",
    "verifyStatus": "UNVERIFIED"
}
```

With *HATEOAS* link

```
{
    "userID": "123456",
    "name": "SJ",
    "verifyStatus": "UNVERIFIED",
    "links": [
        {
            "rel": "self",
            "href": "http://localhost:8080/api/getUser/123456",
            "type": "GET"
        }
    ]
}
```

Before understanding HOW to do this, lets understand WHEN to use and WHY to use HATEOS link?

2 Major Purpose of using HATEOAS link is to achieve

- "LOOSE COUPLING" and
- "API DISCOVERY"

To achieve above, server provides the next set of APIs (actions) in the Response itself, which client can take. So that client have less business logic around APIs (which API to invoke, when to invoke, how to invoke etc....)

But, Adding all next set of ACTIONS can make our API Response Bloat up and has several disadvantages:

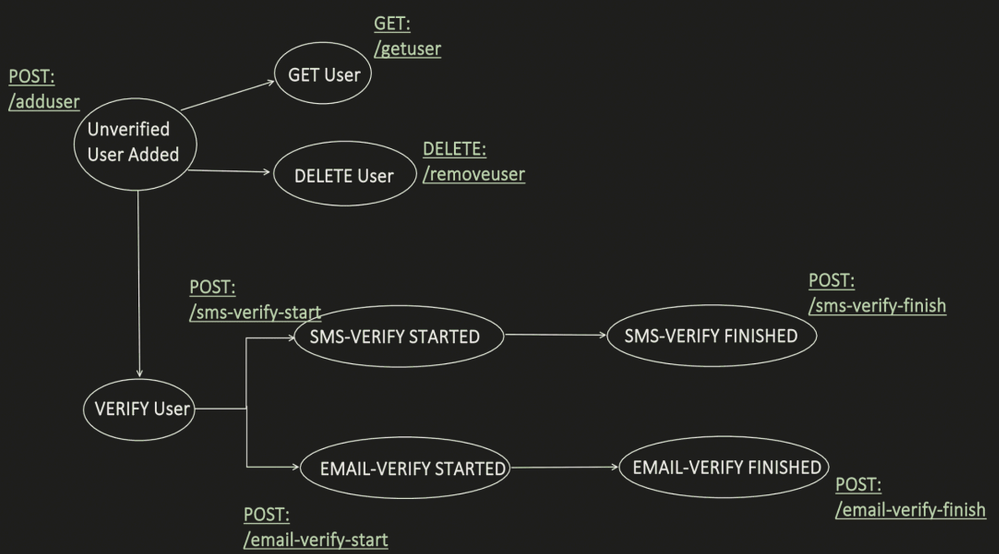
- Increase complexity at server side.
- Latency impact.
- Increase Payload size.


```
{
    "userID": "123456",
    "name": "SJ",
    "verifyStatus": "UNVERIFIED",
    "links": [
        {
            "rel": "self",
            "href": "http://localhost:8080/api/getUser/123456",
            "type": "GET"
        },
        {
            "rel": "remove",
            "href": "http://localhost:8080/api/removeuser/123456",
            "type": "DELETE"
        },
        {
            "rel": "update",
            "href": "http://localhost:8080/api/updateuser/123456",
            "type": "PATCH"
        },
        {
            "rel": "verify-start",
            "href": "http://localhost:8080/api/sms-verify-start/123456",
            "type": "POST"
        },
        {
            "rel": "verify-finish",
            "href": "http://localhost:8080/api/sms-verify-finish/123456",
            "type": "POST"
        }
    ]
}
```


Never Ever add all possible next set of actions (API) just like that.

So, proper analysis need to be done, what actually will help us to achieve LOOSE COUPLING

Now, if we see the above diagram again



I see, the tight coupling lies during VERIFY process.

Client need some info, before it can decide which Verify API to invoke. For example:

```
{
    "userID": "123456",
    "name": "SJ",
    "verifyStatus": "UNVERIFIED",
    "verifyType": "SMS",
    "verifyState": "NOT_YET_STARTED"
}
```

Client need to put business logic, that

```
if(VerifyStatus == "UNVERIFIED")
{
    if(verifyType== "SMS")
      {
          if(verifyState== "NOTE_YET_STARTED")
          {
                Call POST: /sms-verify-start
          }
          Else if (verifyState == "STARTED")
          {
                Call POST: /sms-verify-finish
          }
      }
      Else if(verifyType == "EMAIL")
      {
          if(verifyState== "NOTE_YET_STARTED")
          {
                Call POST: /email-verify-start
          }
          Else if (verifyState == "STARTED")
          {
                Call POST: /email-verify-finish
          }
      }
```

This dependency, can be removed by HATEOAS link

```
{
    "userID": "123456",
    "name": "SJ",
    "verifyStatus": "UNVERIFIED",
    "links": [
        {
            "rel": "verify",
            "href": "http://localhost:8080/api/sms-verify-finish/123456",
            "type": "POST"
        }
    ]
}
```

Now, we have achieved LOOSE COUPLING and Client code looks like this.

```
if(verifyStatus == "UNVERIFIED") {

    //invoke the verify URI, given in HATEOAS link

}
```

Dependency Required:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-hateoas</artifactId>
<version>2.6.4</version>
</dependency>
```

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

    @Autowired
    User user;

    @PostMapping(path = "/adduser")
    public ResponseEntity<UserResponse> addUser() {

        UserResponse response = user.getUser();

        //our business logic to determine which Verify API need to be invoked.
        Link verifyLink =  WebMvcLinkBuilder.linkTo(UserController.class) WebMvcLinkBuilder
                .slash( object: "sms-verify-finish")
                .slash(response.getUserID())
                .withRel("verify") Link
                .withType("POST");

        response.addLink(verifyLink);

        return new ResponseEntity<>(response, HttpStatus.OK);
    }
}
```

```
public class HateoasLinks {

    private List<Link> links = new ArrayList<>();

    public void addLink(Link link) {
        links.add(link);
    }
}
```

```
public class UserResponse extends HateosLinks {

    private String userID;
    private String name;
    private String verifyStatus;

    //getters and setters here
}
```

Other way to create Link:

```
Link verifyLink =  Link.of( href: "/api/sms-verify-finish/" + response.getUserID())
        .withRel( relation: "verify")
        .withType("POST");
```


```
{
    "userID": "123456",
    "name": "SJ",
    "verifyStatus": "UNVERIFIED",
    "links": [
        {
            "rel": "verify",
            "href": "http://localhost:8080/api/sms-verify-finish/123456",
            "type": "POST"
        }
    ]
}
```

Response Generally contains 3 parts:

Status Code: HTTP return code like 200 OK, 500 INTERNAL_ERROR etc.
Header: Additional information (Optional)
Body: Data need to be sent in response.

We can use "ResponseEntity<T>" to create Response and in this 'T' represents the type of the 'Body'.

```java
@GetMapping (path = "/get-user")
public ResponseEntity<String> getUser() {

    return ResponseEntity.ok( body: "My Response body Object can go here");
}
```

OR

```java
@GetMapping (path = "/get-user")
public ResponseEntity<String> getUser() {

    HttpHeaders headers = new HttpHeaders();
    headers.add( headerName: "My-Header1",  headerValue: "SomeValue1");
    headers.add( headerName: "My-Header2",  headerValue: "SomeValue2");

    return ResponseEntity.status(HttpStatus.OK)
            .headers(headers)
            .body("My Response body Object can go here");

}
```

body should be last, actually its kind of using Builder design pattern, so 'status' , 'headers' all are returning Builder object and 'body' method call returns the ResponseEntity object.

```
public static BodyBuilder status(HttpStatusCode status) {
    Assert.notNull(status, message: "HttpStatusCode must not be null");
    return new DefaultBuilder(status);
}
```

```
B headers(@Nullable HttpHeaders headers);
```

```
<T> ResponseEntity<T> body(@Nullable T body);
```

So, what to do, when we don't want to add any body in the response:

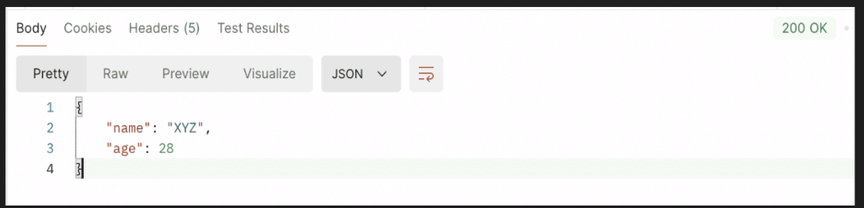
Then, we should use 'build' method

```
<T> ResponseEntity<T> build();
```

```
@GetMapping (path = "/get-user")
public ResponseEntity<Void> getUser() {

    HttpHeaders headers = new HttpHeaders();
    headers.add( headerName: "My-Header1", headerValue: "SomeValue1");
    headers.add( headerName: "My-Header2", headerValue: "SomeValue2");

    return ResponseEntity.status(HttpStatus.OK)
            .headers(headers).build();

}
```

by-default, 200 Ok is the status code set:

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

    @GetMapping (path = "/get-user")
    public User getUser() {
       User responseObj = new User( name: "XYZ", age: 28);
        return responseObj;
    }
}
```



@ResponseBody

When we return Plain string or POJO directly from the class, then @ResponseBody annotation is required.

Why?
It tells to considered value as Response Body and not the View.

But in above example (mentioning below also), we did not use @ResponseBody

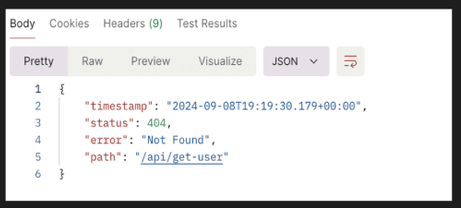
```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping (path = "/get-user")
    public User getUser() {
       User responseObj = new User( name: "XYZ", age: 28);
        return responseObj;
    }
}
```

Its because, @RestController, automatically puts @ResponseBody to all the methods

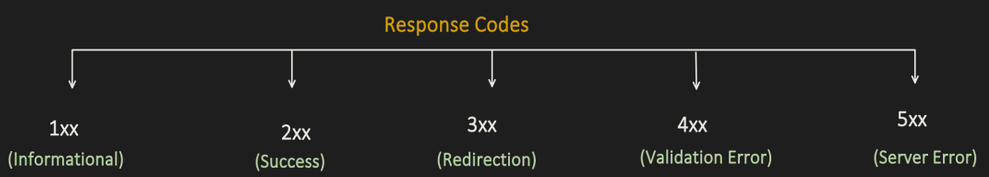
```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Controller
@ResponseBody
public @interface RestController {
    @AliasFor(
            annotation = Controller.class
    )
    String value() default "";
}
```

But lets say, if you use @Controller instead, then below code will throw exception

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

    @GetMapping (path = "/get-user")
    public String getUser() {
        return "XYZ";
    }
}
```



Because, Return value is treated as "view" and spring boot will try to look for file with the given name "XYZ" which do not exist.



2xx (Success)
Request received from Client is received and processed successfully.

Status Code	Reason	Mostly used in	More details
200	Ok	GET, POST (Idempotent calls)	Request is successful and we are returning the response body.
201	Created	POST	Request is successful and new resource is created.
202	Accepted	POST	Request is successfully accepted but processing is not yet completed. Batch processing like Export, Import etc.
204	No Content	DELETE	Request is successful and we are NOT returning any data in response body.
206	Partial Content	POST	Request is partial successful, say during Bulk Addition of 100 Users , 95 passed and 5 requests failed, so this Response code can be used.

3xx (Redirection)

Client must take additional action to complete the request

Status Code	Reason	Mostly used in	More details
301	Moved Permanently	When we migrate from Legacy API to new API (Old Status code, new one is 308)	All request should directed to the new URI.
308	Permanent Redirect	When we migrate from Legacy API to new API	Same as 301, but it do not allow HTTP Method to change while redirect (for ex: if Old API call is POST, then NEW API should also be POST, which is relaxed in 301)
304	Not Modified	GET PATCH  (try to avoid using it with PATCH for example: you trying to update a name of the USER, but lets say name is already same in the DB, so no update required. In that case, we should not throw 304 (NOT_MODIFIED) error code, instead 204(NO_CONTENT) or 200(OK) is more appropriate.)	1. Client makes a GET call, Server returned it with Last-Modified time in header. 2. Client cache the response. 3. Client make a GET call, pass this Last modified time in "If-Modified-Since" header. 4. Server check the particular resource last update time with what client provided, if resource is not updated, server simply returns 304 (NOT_MODIFIED). 5. If Modified, server process the request as usual and returns the new values.

4xx (Validation Errors)

Client need to pass correct request to server

Status Code	Reason	Mostly used in	More details
400	Bad Request	GET, POST, PATCH, DELETE	Client is not passing the required details to process the request.
401	Unauthorized	GET, POST, PATCH, DELETE	Any API, which require Authentication(like Bearer token, Basic authentication etc..) and client try to access it without providing authentication details.
403	Forbidden	GET, POST, PATCH, DELETE	Lets say, only ADMIN can perform certain operation. But if API get invoked apart from ADMIN. We should throw 401 status code as clients (apart from ADMIN) do not have permission to access the resource.
404	Not Found	GET, PATCH, DELETE	The requested resource which client passes, is not found in DB by the server. For ex: GET the user details with ID: 123, but in DB there is not such ID present.
405	Method Not Allowed	GET, POST, PATCH, DELETE	Ex: Hitting GET API, but with POST HTTP Method. In Springboot, dispatcher servlet might throw this error, as control not even reach to controller.
422	Un-processable Entity	GET, POST, PATCH, DELETE	Your application Business validation: Like France Users should not be allowed to open an account. (as country is not supported yet)
429	Too Many Requests	GET, POST, PATCH, DELETE	Lets say: our rule is: 1 user can max make 10 calls in a minute. if User:12345 makes the 11th call in a minute then this 11th call should get failed and we can throw 429 error code.

5xx (Server Errors)

Request got failed at Server, even though client passed the valid request. Means Something wrong at Server.

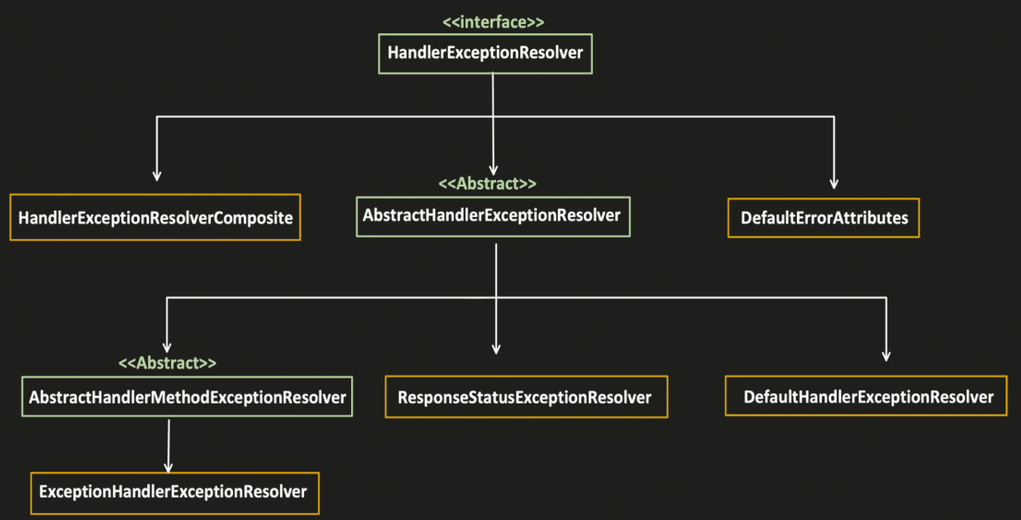
Status Code	Reason	Mostly used in	More details
500	Internal Server Error	GET, POST, PATCH, DELETE	Generic error code when no more specific error code is suitable.
501	Not Implemented	GET, POST, PATCH, DELETE	API lacks the ability to fulfill the request. Or say, API is in development and in future it will be available.
502	Bad Gateway	GET, POST, PATCH, DELETE	Server acting as a proxy and while calling upstream got invalid response. Example: My application is deployed behind Reverse Proxy (Nginx). If NginX is not able to communicate with my application (because of misconfiguration of port number or something), then it is eligible to throw 502 Bad Gateway.

1xx (Informational)

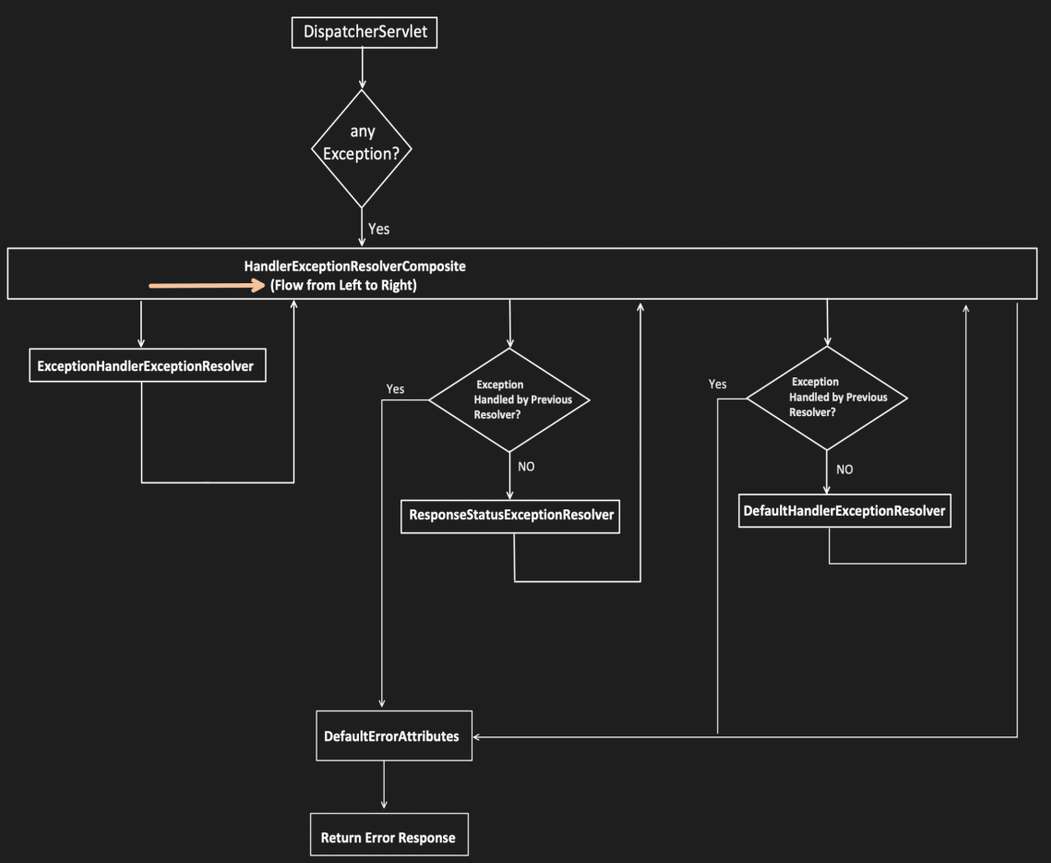
Interim response to communicate request progress or its status before processing the final request.

Status Code	Reason	Mostly used in	More details
100	Continue	POST	Before sending the request, client check with server, if it can handle the request and ready: 1. Client add few things in the header first, like: - content length : 1048576 - content type : multipart/form-data - Expect: 100-continue 2. Server, checks that in header, 'Expect:100-continue' is present, means, client is just checking. So server validate everything (authentication, authorization, content type, length etc.) 3. If Server is okay, it return 100 CONTINUE status code 4. Client receives it and then invokes the API again without Expect and server process the request.

Classes involved in handling an Exception:



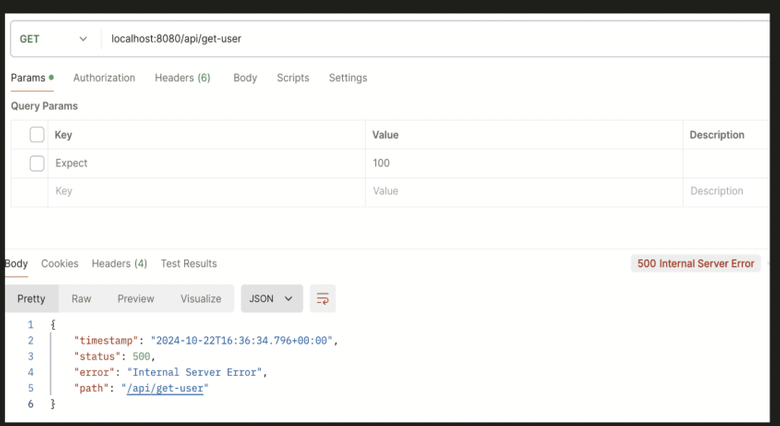
Let's understand the sequence, when any exception occurs:



Let's see the flow with an example:

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

    @GetMapping(path = "/get-user")
    public String getUser() {
        throw new NullPointerException("throwing null pointer exception for testing");
    }
}
```

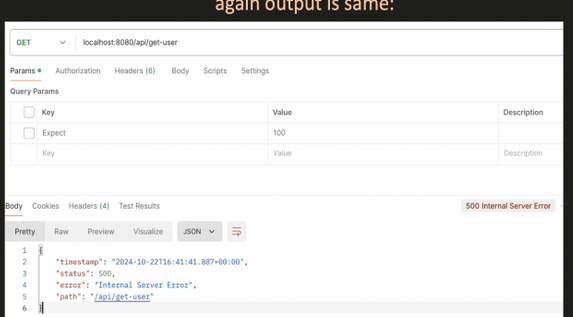
Output:




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

    @GetMapping(path = "/get-user")
    public String getUser() {

        throw new CustomException(HttpStatus.BAD_REQUEST,
                "request is not correct, UserID is missing");
    }
}
```

```java
public class CustomException extends RuntimeException{

    HttpStatus status;
    String message;

    CustomException(HttpStatus status, String message) {
        this.status = status;
        this.message = message;
    }

    public HttpStatus getStatus() {
        return status;
    }

    @Override
    public String getMessage() {
        return message;
    }
}
```

again output is same:



Why for both output is same?
Why my Return Code "BAD_REQUEST" i.e. 400 and my error message is not shown in output?

In both the example, we are not creating the **ResponseEntity** object, we are just returning the exception, some other class is creating the ResponseEntity Object.

If we need full control and don't want to rely on Exception Resolvers, then we have to create the ResponseEntity Object.

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

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {
        try {
            //your business logic and validations...
            throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
        }
        catch (CustomException e) {
            ErrorResponse errorResponse = new ErrorResponse(new Date(), e.getMessage(), e.getStatus().value());
            return new ResponseEntity<>(errorResponse, e.getStatus());
        }
        catch (Exception e) {
            ErrorResponse errorResponse = new ErrorResponse(new Date(), e.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR.value());
            return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}
```

```java
public class CustomException extends RuntimeException{

    HttpStatus status;
    String message;

    CustomException(HttpStatus status, String message) {
        this.status = status;
        this.message = message;
    }

    public HttpStatus getStatus() {
        return status;
    }

    @Override
    public String getMessage() {
        return message;
    }
}
```

```java
public class ErrorResponse {

    private Date timestamp;
    private String msg;
    private int status;

    public ErrorResponse(Date timestamp, String msg, int status) {
        this.msg = msg;
        this.status = status;
        this.timestamp = timestamp;
    }


    public Date getTimestamp() {
        return timestamp;
    }

    public String getMessage() {
        return msg;
    }

    public int getStatus() {
        return status;
    }
}
```

Body Cookies Headers (4) Test Results — 400 Bad Request

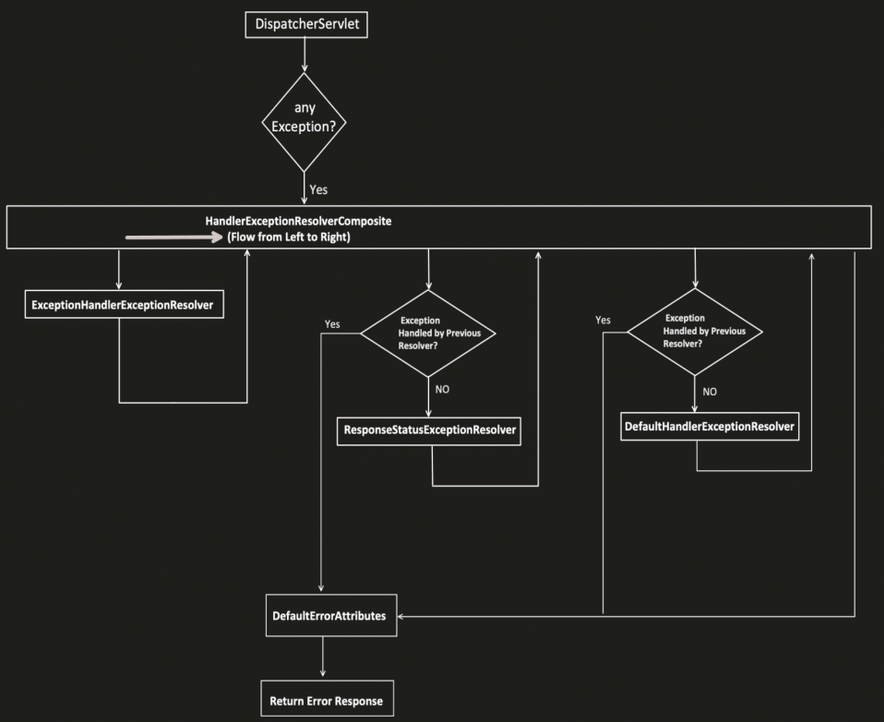
Pretty Raw Preview Visualize JSON

```
{
    "timestamp": "2024-10-25T12:13:02.318+00:00",
    "status": 400,
    "message": "UserID is missing"
}
```

So, When we don't return ResponseEntity like below:

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

    @GetMapping(path = "/get-user")
    public String getUser() {

        throw new CustomException(HttpStatus.BAD_REQUEST,
                "request is not correct, UserID is missing");
    }
}
```

then in Exception scenario, exception passes through each Resolver like "ExceptionHandlerExceptionResolver", "ResponseStatusExceptionResolver" and "DefaultHandlerExceptionResolver" in sequence.



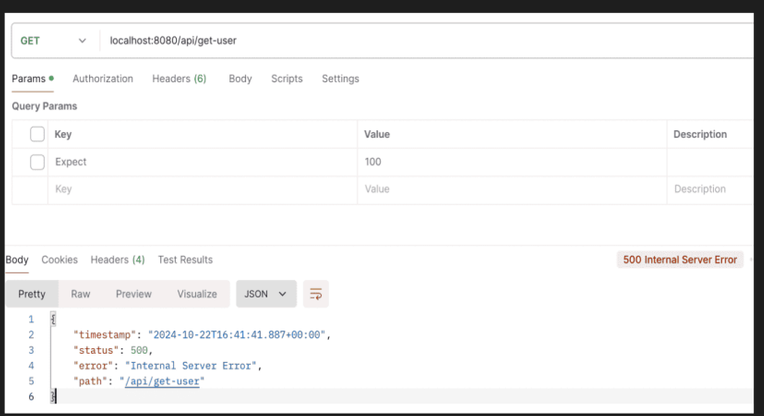
Each Resolver set the proper **status** and **message** in HTTP response for exceptions which they are taking care of.
and
NullPointerException and CustomException all the 3 Resolvers, do not understand, so Status and Message is not set.

So, when control reaches to "**DefaultErrorAttributes" class, it fills the values in HTTP Response with default values.**

DefaultErrorAttributes

```java
@Override
public Map<String, Object> getErrorAttributes(WebRequest webRequest, ErrorAttributeOptions options) {
    Map<String, Object> errorAttributes = getErrorAttributes(webRequest, options.isIncluded(Include.STACK_TRACE));
    options.retainIncluded(errorAttributes);
    return errorAttributes;
}

private Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean includeStackTrace) {
    Map<String, Object> errorAttributes = new LinkedHashMap<>();
    errorAttributes.put("timestamp", new Date());
    addStatus(errorAttributes, webRequest);
    addErrorDetails(errorAttributes, webRequest, includeStackTrace);
    addPath(errorAttributes, webRequest);
    return errorAttributes;
}
```

```java
@RequestMapping
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
    HttpStatus status = this.getStatus(request);
    if (status == HttpStatus.NO_CONTENT) {
        return new ResponseEntity(status);
    } else {
        Map<String, Object> body = this.getErrorAttributes(request, this.getErrorAttributeOptions(request, MediaType.ALL));
        return new ResponseEntity(body, status);
    }
}
```



So, now question is: what exception does "ExceptionHandlerExceptionResolver", "ResponseStatusExceptionResolver" and "DefaultHandlerExceptionResolver" handles?

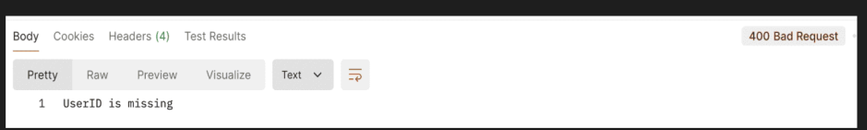
1. ExceptionHandlerExceptionResolver

Responsible for handling below annotations:

- @ExceptionHandler
- @ControllerAdvice

Controller level Exception handling:

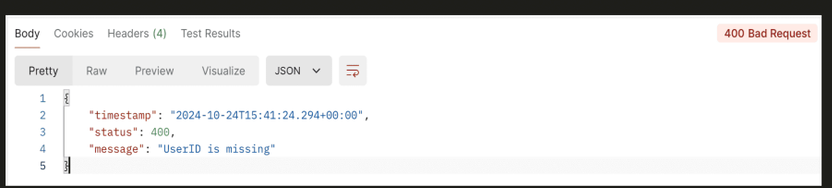
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

            //your business logic and validations...
            throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(ex.getMessage(), ex.getStatus());
    }
}
```



Use-case just to show returning Error Response object instead of just message:

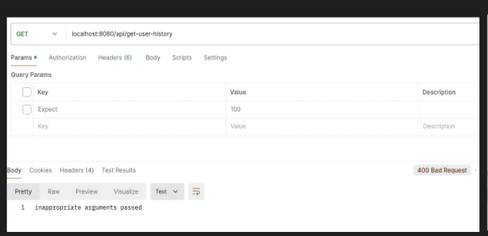
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

            //your business logic and validations...
            throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<Object> handleCustomException(CustomException ex) {
        ErrorResponse errorResponse = new ErrorResponse(new Date(), ex.getMessage(), ex.getStatus().value());
       return new ResponseEntity<>(errorResponse, ex.getStatus());
    }
}
```

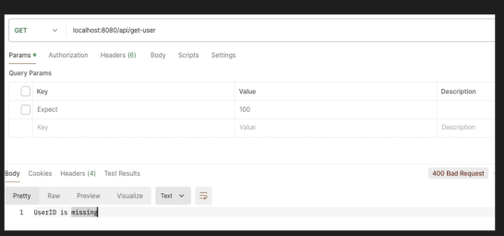
```java
public class ErrorResponse {

    private Date timestamp;
    private String msg;
    private int status;

    public ErrorResponse(Date timestamp, String msg, int status) {
        this.msg = msg;
        this.status = status;
        this.timestamp = timestamp;
    }


    public Date getTimestamp() {
        return timestamp;
    }

    public String getMessage() {
        return msg;
    }

    public int getStatus() {
        return status;
    }
}
```



Use-case just to show multiple @ExceptionHandler in single Controller class:

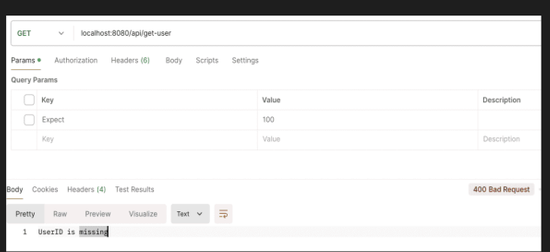
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

           //your business logic and validations...
           throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @GetMapping(path = "/get-user-history")
    public ResponseEntity<?> getUserHistory() {

       //your business logic and validations...
       throw new IllegalArgumentException("inappropriate arguments passed");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
      return new ResponseEntity<>(ex.getMessage(), ex.getStatus());
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleCustomException(IllegalArgumentException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```

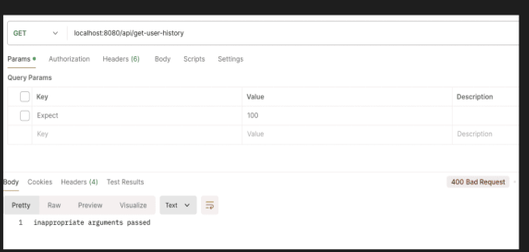




Use-case just to show 1 @ExceptionHandler handling multiple exceptions:

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

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

          //your business logic and validations...
          throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @GetMapping(path = "/get-user-history")
    public ResponseEntity<?> getUserHistory() {

        //your business logic and validations...
        throw new IllegalArgumentException("inappropriate arguments passed");
    }

    @ExceptionHandler({CustomException.class, IllegalArgumentException.class})
    public ResponseEntity<String> handleCustomException(Exception ex) {
       return new ResponseEntity<>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```





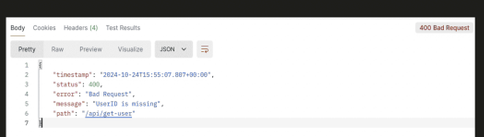
Use-case just to show @ExceptionHandler not returning ResponseEntity and let "**DefaultErrorAttributes**" to create the ResponseEntity.

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

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

          //your business logic and validations...
          throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @GetMapping(path = "/get-user-history")
    public ResponseEntity<?> getUserHistory() {

        //your business logic and validations...
        throw new IllegalArgumentException("inappropriate arguments passed");
    }

    @ExceptionHandler(CustomException.class)
    public void handleCustomException(HttpServletResponse response, CustomException ex) throws IOException {
        response.sendError(HttpStatus.BAD_REQUEST.value(), ex.message);
    }
}
```

Without this DefaultErrorAttributes, filter out the message field in response

application.properties

```
server.error.include-message=always
```

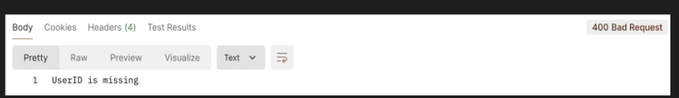


Global Exception handling:

Problem with Controller level @ExceptionHandler is:

- if multiple controller has the same type of Exceptions then same handling we might to do in multiple controller
- which is nothing but a code duplication.

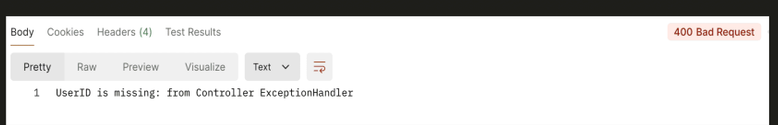
```java
@ControllerAdvice
public class GlobalExceptionHandling {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(ex.message, ex.getStatus());
    }
}
```



What if, I provide both Controller level and Global level @ExceptionHandler, which one has more priority?

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

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

            //your business logic and validations...
            throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>( body: ex.message + ": from Controller ExceptionHandler", ex.getStatus());
    }
}
```

```java
@ControllerAdvice
public class GlobalExceptionHandling {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>( body: ex.message + ": from Global ExceptionHandler", ex.getStatus
    }
}
```



What if there are 2 handlers which can handle an exception, which one will be given priority:

It always follow an hierarch, from bottom to up (first look for exact match if not, check for its parent and so on...)

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

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

            //your business logic and validations...
            throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }
}
```

```java
@ControllerAdvice
public class GlobalExceptionHandling {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(ex.message , ex.getStatus());
    }

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> handleRuntimeException(RuntimeException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

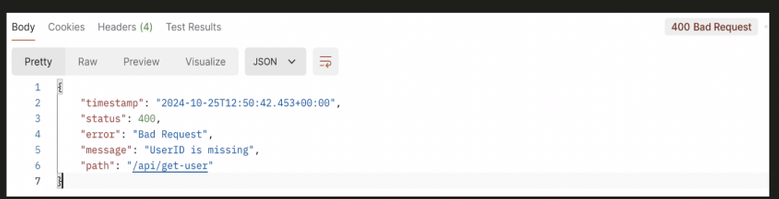
2. ResponseStatusExceptionResolver

Handles Uncaught exception annotated with **@ResponseStatus** annotation.

Use-case1: Used above an Exception class

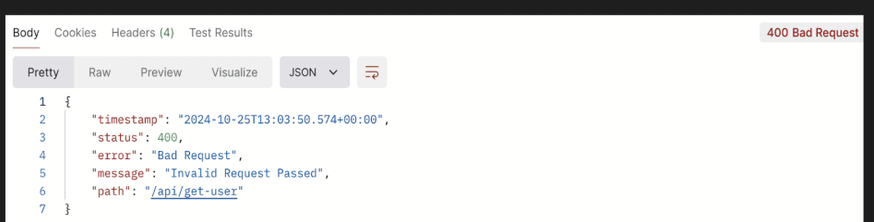
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

            //your business logic and validations...
            throw new CustomException("UserID is missing");
    }
}
```

```java
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class CustomException extends RuntimeException {

    CustomException(String message) {
        super(message);
    }
}
```




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

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

            //your business logic and validations...
            throw new CustomException("UserID is missing");
    }
}
```

```java
@ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Passed")
public class CustomException extends RuntimeException {

    HttpStatus status;

    CustomException(HttpStatus status, String message) {
        super(message);
        this.status = status;
    }
}
```

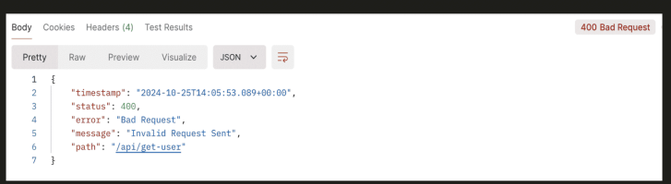


Use-case2: Used above an @ExceptionHandler method

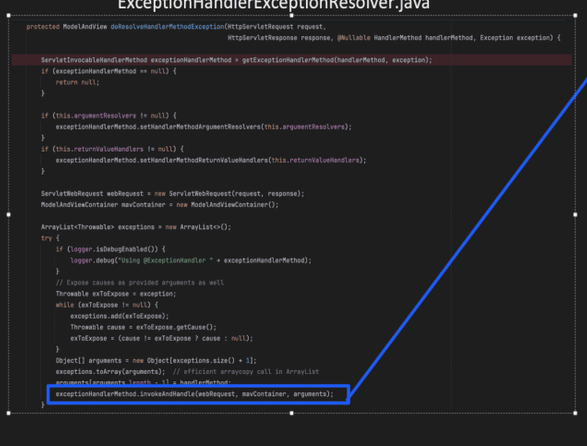
Again ***ResponseStatusExceptionResolver*** handles Uncaught exception annotated with **@ResponseStatus** annotation but if used with ***@ExceptionHandler*** then it will not be handled by "***ResponseStatusExceptionResolver ",*** it will be handled by Spring request handling mechanism itself.

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

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

            //your business logic and validations...
            throw new CustomException(HttpStatus.INTERNAL_SERVER_ERROR, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Sent")
    public ResponseEntity<Object> handleCustomException(CustomException e) {
        return new ResponseEntity<>( body: "you are not authorized", HttpStatus.FORBIDDEN);
    }
}
```

```java
public class CustomException extends RuntimeException {

    HttpStatus status;

    CustomException(HttpStatus status, String message) {
        super(message);
        this.status = status;
    }
}
```

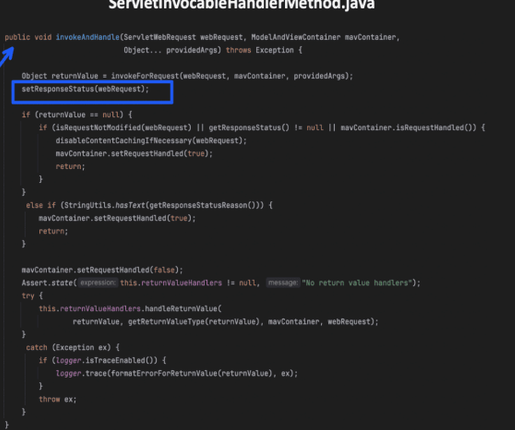


ExceptionHandlerExceptionResolver.java



```java
protected ModelAndView doResolveHandlerMethodException(HttpServletRequest request,
                                                       HttpServletResponse response, @Nullable HandlerMethod handlerMethod, Exception exception) {

    ServletInvocableHandlerMethod exceptionHandlerMethod = getExceptionHandlerMethod(handlerMethod, exception);
    if (exceptionHandlerMethod == null) {
        return null;
    }

    if (this.argumentResolvers != null) {
        exceptionHandlerMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
    }
    if (this.returnValueHandlers != null) {
        exceptionHandlerMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
    }

    ServletWebRequest webRequest = new ServletWebRequest(request, response);
    ModelAndViewContainer mavContainer = new ModelAndViewContainer();

    ArrayList<Throwable> exceptions = new ArrayList<>();
    try {
        if (logger.isDebugEnabled()) {
            logger.debug("Using @ExceptionHandler " + exceptionHandlerMethod);
        }
        // Expose causes as provided arguments as well
        Throwable exToExpose = exception;
        while (exToExpose != null) {
            exceptions.add(exToExpose);
            Throwable cause = exToExpose.getCause();
            exToExpose = (cause != exToExpose ? cause : null);
        }
        Object[] arguments = new Object[exceptions.size() + 1];
        exceptions.toArray(arguments);  // efficient arraycopy call in ArrayList
        arguments[arguments.length - 1] = handlerMethod;
        exceptionHandlerMethod.invokeAndHandle(webRequest, mavContainer, arguments);
    }
```

ServletInvocableHandlerMethod.java



```java
public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,
                            Object... providedArgs) throws Exception {

    Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);
    setResponseStatus(webRequest);

    if (returnValue == null) {
        if (isRequestNotModified(webRequest) || getResponseStatus() != null || mavContainer.isRequestHandled()) {
            disableContentCachingIfNecessary(webRequest);
            mavContainer.setRequestHandled(true);
            return;
        }
    }
    else if (StringUtils.hasText(getResponseStatusReason())) {
        mavContainer.setRequestHandled(true);
        return;
    }

    mavContainer.setRequestHandled(false);
    Assert.state( expression: this.returnValueHandlers != null, message: "No return value handlers");
    try {
        this.returnValueHandlers.handleReturnValue(
                returnValue, getReturnValueType(returnValue), mavContainer, webRequest);
    }
    catch (Exception ex) {
        if (logger.isTraceEnabled()) {
            logger.trace(formatErrorForReturnValue(returnValue), ex);
        }
        throw ex;
    }
}
```

What if @ExceptionHandler method, set Response status and message itself instead of returning the response entity:

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

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

            //your business logic and validations...
            throw new CustomException(HttpStatus.INTERNAL_SERVER_ERROR, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Sent")
    public void handleCustomException(CustomException e, HttpServletResponse response) throws IOException{
        response.sendError(HttpStatus.FORBIDDEN.value(), s: "you are not authorized");
    }
}
```

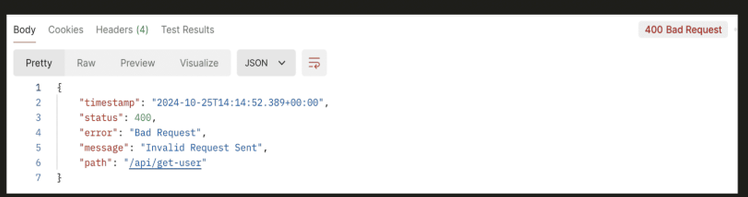
- Its because, Response.sendError first set the status and message in response and do COMMIT.

  2nd ResponseStatus method will try to do the same thing, and Exception will occur in ExceptionHandlerResolver class as we try to reset already committed status field.

So its advisable not to use together @ExceptionHandler and @ResponseStatus together to avoid confusion.

But if you have to, use like below:

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

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

            //your business logic and validations...
            throw new CustomException(HttpStatus.INTERNAL_SERVER_ERROR, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Sent")
    public void handleCustomException(CustomException e) {
        //do nothing here
    }
}
```



3. DefaultHandlerExceptionResolver

Handles Spring framework related exceptions only like MethodNotFound, NoResourceFound etc..