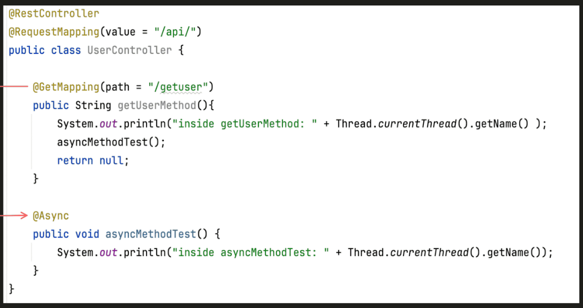
## Conditions for @Async Annotation to work properly?

1. Different Class :
   If @Aync 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.

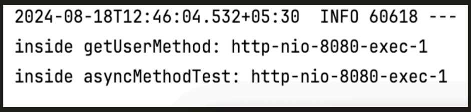
*Both in same class, Proxy will get bypassed*



```java
@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*

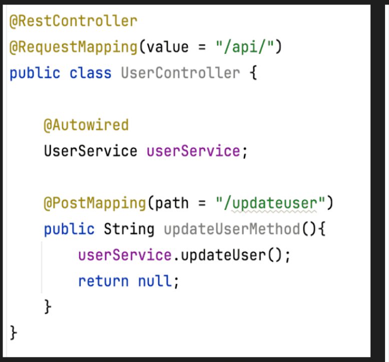
**Output:**



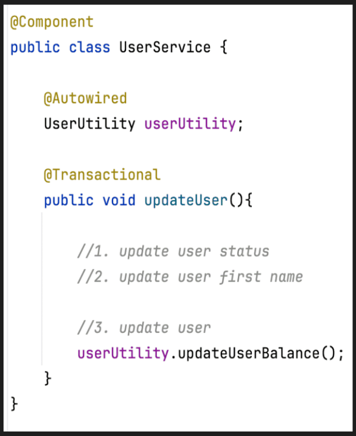
```
2024-08-18T12:46:04.532+05:30  INFO 60618 ---
inside getUserMethod: http-nio-8080-exec-1
inside asyncMethodTest: http-nio-8080-exec-1
```

## @Aysn and Transaction Management

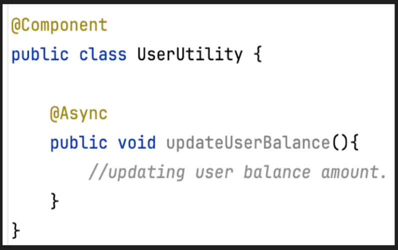
**Usecase1:** ✗ **Transaction Context do not transfer from caller thread to new thread which got created by Async.**



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

    @Autowired
    UserService userService;

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

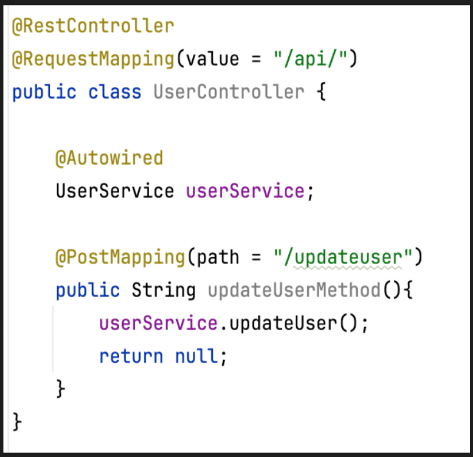


```java
@Component
public class UserService {

    @Autowired
    UserUtility userUtility;

    @Transactional
    public void updateUser(){

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

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

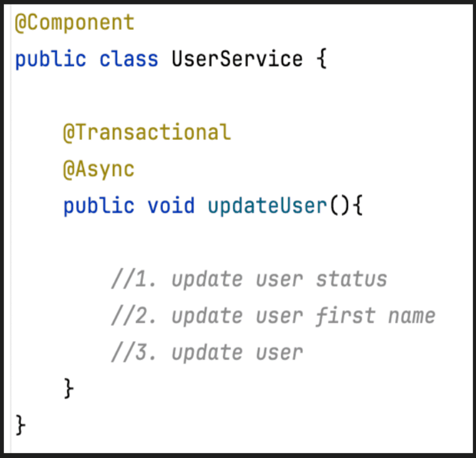


```java
@Component
public class UserUtility {

    @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.

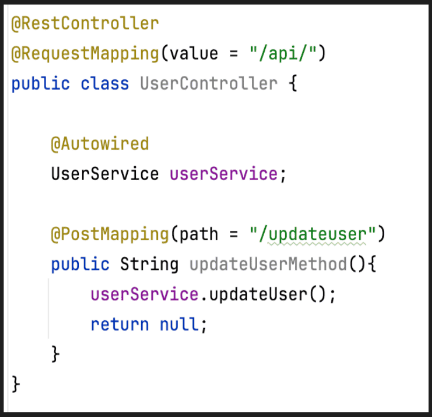


```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserService userService;

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

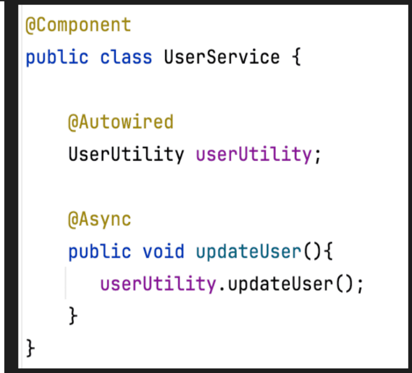


```java
@Component
public class UserService {

    @Transactional
    @Async
    public void updateUser(){

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

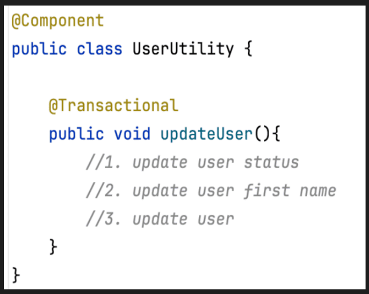
**Usecase3:** ✓



```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();
    }
}
```

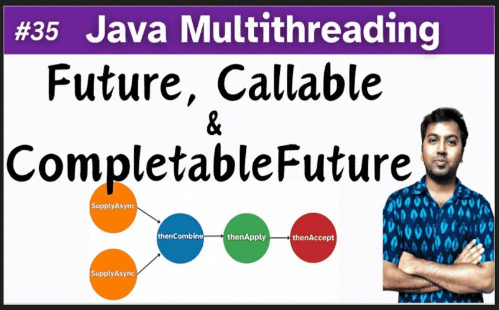


```java
@Component
public class UserUtility {

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

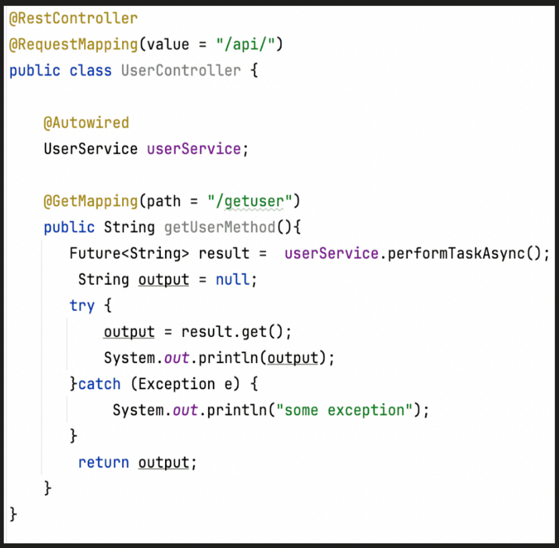
## @Aysn Method return type

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

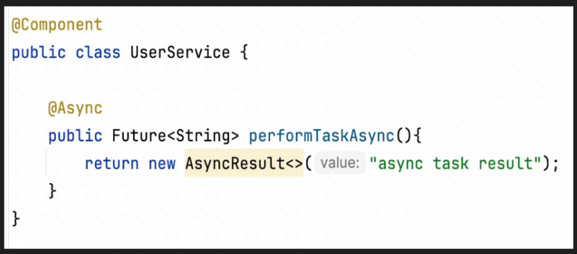
***Checkout Java Playlist to learn more in depth of Future and CompletableFuture***



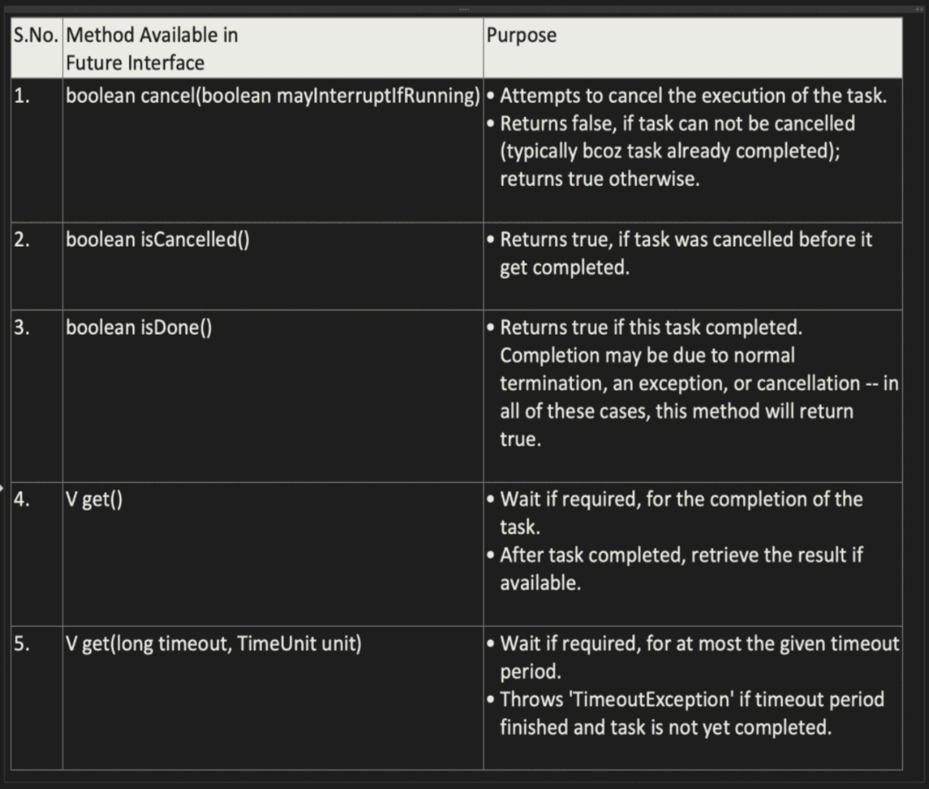
***Using Future:***



```
@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;
    }
}
```



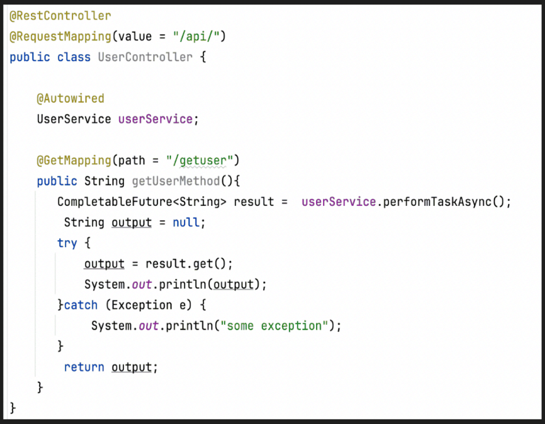
```
@Component
public class UserService {

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



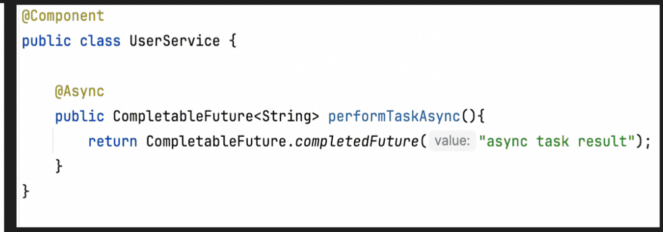
| S.No. | Method Available in Future Interface | Purpose |
|---|---|---|
| 1. | boolean cancel(boolean mayInterruptIfRunning) | • Attempts to cancel the execution of the task.<br>• Returns false, if task can not be cancelled (typically bcoz task already completed); returns true otherwise. |
| 2. | boolean 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.<br>• 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.<br>• Throws 'TimeoutException' if timeout period finished and task is not yet completed. |

## *Using CompletableFuture:*

Introduced in Java8



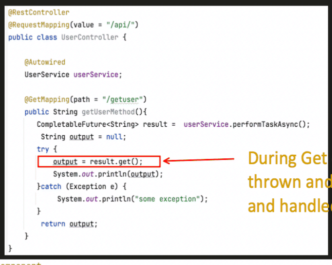
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @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");
    }
}
```

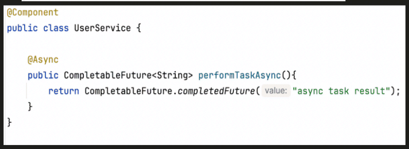
## Exception Handling

Method which has Return type



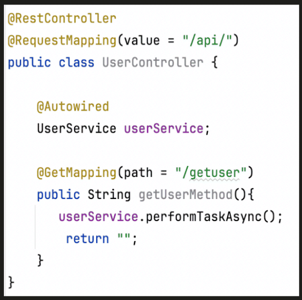
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @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;
    }
}
```

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

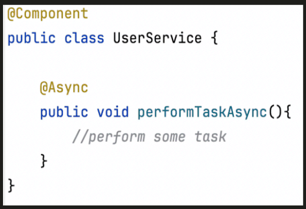


```java
@Component
public class UserService {

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

Method which do not return anything



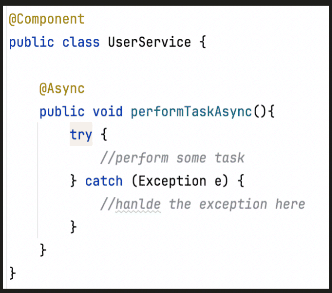
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @Autowired
    UserService userService;

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



```java
@Component
public class UserService {

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

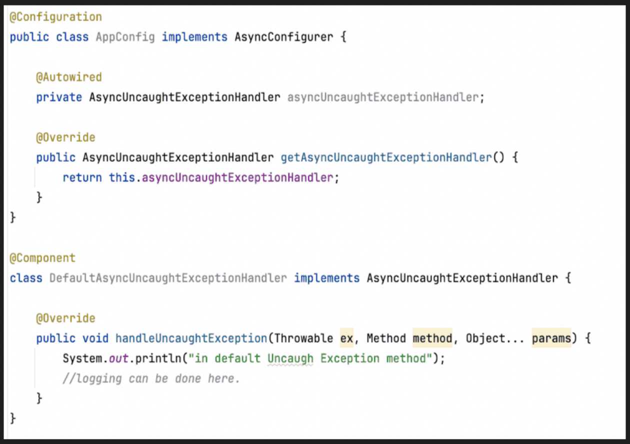
How to handle this?

1. **Within Async Method itself**

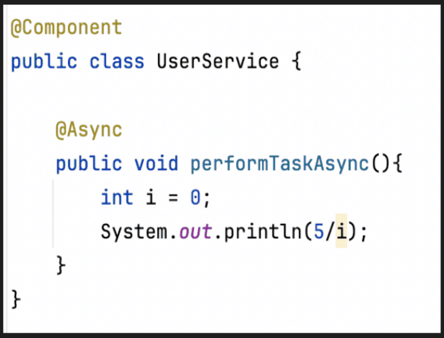


```java
@Component
public class UserService {

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

## 2. Implement Custom AsyncExceptionHandler

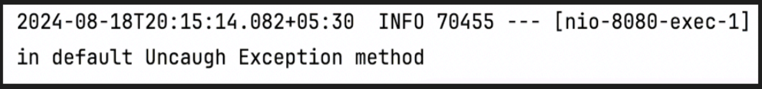


```java
@Configuration
public class AppConfig implements AsyncConfigurer {

    @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.
    }
}
```



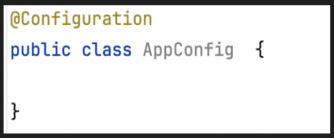
```java
@Component
public class UserService {

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

**Output:**

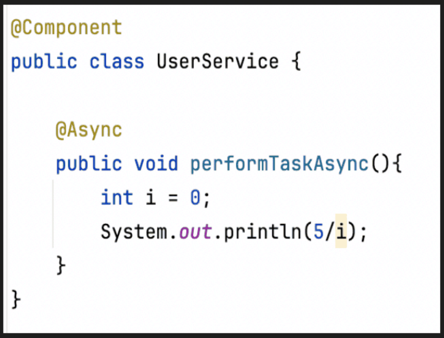


```
2024-08-18T20:15:14.082+05:30  INFO 70455 --- [nio-8080-exec-1]
in default Uncaugh Exception method
```

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

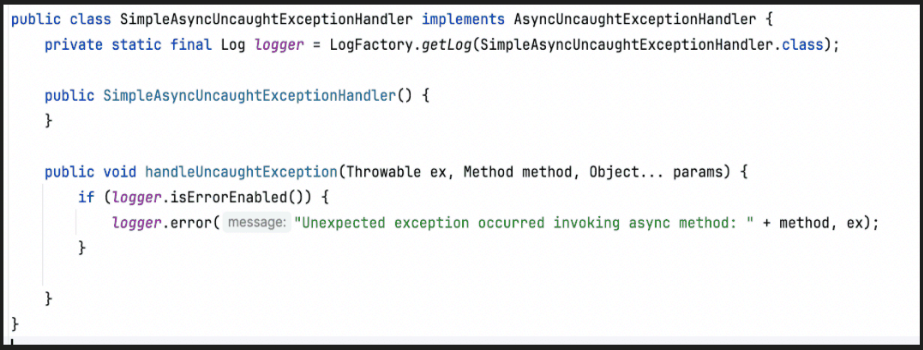


```java
@Configuration
public class AppConfig  {

}
```



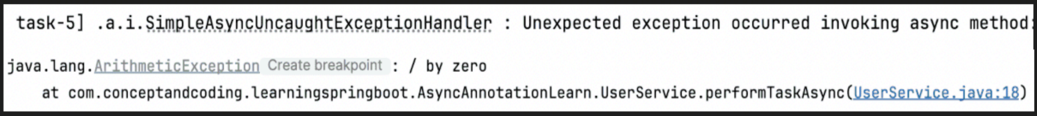
```java
@Component
public class UserService {

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

## Spring boot framework code..



```
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)
```