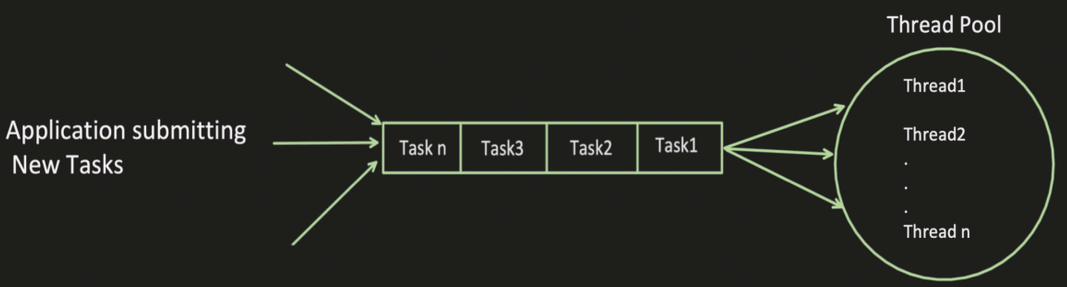
What is ThreadPool:

- 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 {

}
```

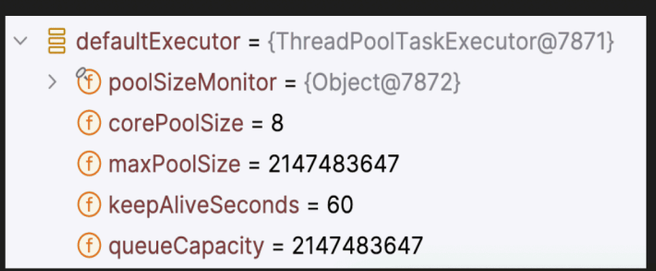
```java
@SpringBootApplication
@EnableAsync
public class SpringbootApplication {

    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
```