

* **Synchronous Programming:**In synchronous programming, tasks are performed one at a time and only when one is completed the next is unblocked.
* **Asynchronous Programming:**In asynchronous programming, multiple tasks can be executed simultaneously. You can move to another task before the previous one finishes.

In spring boot, we can achieve asynchronous behaviour using @Async annotation. But just @Async annotation will not work. For that, you need to understand how @Async internally works.

@EnableAsync annotation. By default mode is Proxy and another one is AspectJ. In this post, we are going to discuss Proxy mode. Proxy mode allows for the interception of the call through proxy only. Never call the async method from the same class where it is defined, it will not work.

**How does @Async annotation work?**

First, annotate the method with @Async. When you annotate a method with @Async annotation, it creates a proxy for that object based on “proxyTargetClass” property.

When spring executes this method, by default it will be searching for associated thread pool definition. Either a unique spring framework TaskExecutor bean in the context or Executor bean named “taskExecutor”. If neither of these two is resolvable, default it will use spring framework SimpleAsyncTaskExecutor to process async method execution.

**ThreadPoolTaskExecutor** is a class provided by the Spring Framework that serves as an implementation of the **TaskExecutor** interface. It's particularly useful for managing and controlling the execution of concurrent tasks in a Spring application. This class is an implementation of the **Executor** interface, providing a thread pool for managing the execution of tasks asynchronously.

Here are some key details about **ThreadPoolTaskExecutor**:

1. **Task Execution:**
   * **ThreadPoolTaskExecutor** is designed to execute tasks concurrently in a pool of worker threads.
   * It's typically used for parallelizing and managing the execution of asynchronous or background tasks within a Spring application.
2. **Thread Pool Configuration:**
   * You can configure various properties of the thread pool, such as the core pool size, maximum pool size, and the queue capacity.
   * The core pool size is the number of threads to keep in the pool even if they are idle.
   * The maximum pool size is the maximum number of threads to allow in the pool.
   * The queue capacity determines how many tasks can be queued up if the pool is saturated.
3. **Queueing Mechanism:**
   * When there are more tasks than the core pool size, additional tasks are placed in a queue.
   * Depending on the configuration, if the queue is full and the maximum pool size is reached, the **ThreadPoolTaskExecutor** can either reject the task or execute it in the calling thread.
4. **Thread Naming:**
   * You can customize the thread names of the threads in the pool, making it easier to identify threads in logs and debugging.
5. **Timeouts and Rejection Policies:**
   * You can configure timeout settings for tasks and specify rejection policies in case the pool is saturated.
   * Common rejection policies include aborting, discarding, or calling the task in the caller's thread.
6. **Integration with Spring:**
   * **ThreadPoolTaskExecutor** is often used in the context of Spring's **@Async** annotation, allowing methods to be executed asynchronously by the thread pool.
7. **Monitoring and Metrics:**
   * Spring provides various monitoring and management features for thread pools, and **ThreadPoolTaskExecutor** integrates with these features.