@Async annotation vs completable future:

Both `@Async` (from Spring) and `CompletableFuture` (from Java's concurrency package) are used for asynchronous programming, but they work differently and serve different purposes. Let's break down their differences, use cases, and best practices.

1. `@Async` in Spring @awsravi

What is `@Async`?

- `@Async` is a Spring annotation that allows methods to run asynchronously in a separate thread.
- It is part of **Spring's Task Execution Framework** and is used to improve performance by running tasks concurrently.

How it Works

- When a method is annotated with `@Async`, Spring automatically executes it in a separate thread without blocking the main thread.
- Spring uses an underlying `TaskExecutor` (by default, `SimpleAsyncTaskExecutor`), which can be customized.

```
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;

import java.util.concurrent.CompletableFuture;

@Service
public class MyService {

    @Async
    public CompletableFuture<String> asyncMethod() {
        try {
            Thread.sleep(2000); // Simulate a long-running task
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return CompletableFuture.completedFuture("Task Completed");
    }
}
```

Key Points of `@Async`

- Requires Spring Boot and `@EnableAsync` annotation in the main configuration at Class level.

- Works with **Spring Beans only** (i.e., methods should be called from another Spring-managed bean).
- Uses **thread pool executors**, which can be configured.

- Methods should return `CompletableFuture<T>` or `void` (if you don't need a result).

Advantages

✅ Simple to use with Spring-managed beans.

✅ Handles thread management automatically via Spring's Task Executor.

✅ Works well with `CompletableFuture` for further processing.

Disadvantages

❌ Only works in a Spring-managed environment.

❌ Calling `@Async` methods within the same class doesn't work because Spring proxies the method.

❌ Limited flexibility compared to native Java concurrency utilities.

2. `CompletableFuture` (Java 8+)

What is `CompletableFuture`?

- `CompletableFuture` is part of Java's `java.util.concurrent` package.
- It provides an advanced way to handle asynchronous computations with **more flexibility** than `@Async`.
- Allows chaining, exception handling, combining multiple futures, and more.

EX:

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class CompletableFutureExample {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(2000); // Simulate a long task
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "Task Completed";
        });

        System.out.println(future.get()); // Blocking call to get result
    }
}
```

@awsravi

Key Features of `CompletableFuture`

- **Non-blocking execution** using `supplyAsync()` or `runAsync()`.
- **Chaining operations** using `.thenApply()`, `.thenAccept()`, etc.

- **Combining multiple async operations** using `.thenCombine()`, `.allOf()`, `.anyOf()`, etc.
- **Exception handling** using `.exceptionally()`.

EX:

CompletableFuture.supplyAsync(() -> "Hello")

.thenApply(s -> s + " World")

.thenAccept(System.out::println);

Key Differences: `@Async` vs `CompletableFuture`

Feature	`@Async` (Spring)	`CompletableFuture` (Java)
Scope	Spring Boot apps only	General Java concurrency
Thread Management	Managed by Spring (TaskExecutor)	Uses `ForkJoinPool` by default (or custom `Executor`)
Ease of Use	Simple (just annotate method)	More complex but powerful
Method Invocation	Works with Spring beans	Works everywhere
Chaining Operations	Not directly (can return `CompletableFuture`)	Yes, supports fluent chaining
Exception Handling	Limited to Spring's exception handling	Supports `.exceptionally()`, `.handle()`, etc.
Parallel Execution	Possible but limited	Easily supports parallel execution

When to Use What?

Use Case	Best Choice
Simple async task in a Spring Boot app	`@Async`
Need control over execution (thread pool, exception handling, etc.)	`CompletableFuture`
Chaining multiple async tasks	`CompletableFuture`
Running independent tasks in parallel	`CompletableFuture.allOf()`
Want automatic thread management in Spring	`@Async`
Need fine-grained control over threads	`CompletableFuture` with custom executor

Can We Use Both Together?

Yes! You can use `@Async` to offload tasks to Spring's thread pool while returning a `CompletableFuture` to take advantage of Java's async chaining.

Example: Using `@Async` with `CompletableFuture`

```
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;
import java.util.concurrent.CompletableFuture;

@Service
public class MyService {

    @Async
    public CompletableFuture<String> fetchData() {
        return CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "Data Loaded";
        });
    }
}
```

This method:

1. Runs asynchronously in Spring's managed thread pool.
2. Uses `CompletableFuture` for further processing.

@awsravi

Final Thoughts

- Use `@Async` for **quick and easy async execution** in Spring Boot applications.
- Use `CompletableFuture` when **you need more control, chaining, or parallel execution**.
- In Spring apps, **you can use both together** to get the best of both worlds!