Mono. fromFuture(), its relation with CompletableFuture and the common ForkJoinPool

**Mono.fromFuture(CompletableFuture<? extends T> future) in Reactor**

Mono.fromFuture() is a static factory method in Reactor that bridges the gap between the asynchronous programming model of Java's java.util.concurrent.CompletableFuture and Reactor's reactive Mono.

**Functionality:**

1. **Takes a CompletableFuture:** It accepts a CompletableFuture as input. A CompletableFuture represents the result of an asynchronous computation that may not yet be available. It can be explicitly completed with a value or completed exceptionally with an error.
2. **Creates a Mono<T>:** It returns a Mono that will eventually emit at most one item (T) representing the result of the CompletableFuture.
3. **Lazy Subscription:** Similar to other Reactor Mono creators, the interaction with the CompletableFuture (specifically, getting its result or error) only begins when a Subscriber subscribes to the Mono.
4. **Value Emission:**
   * If the CompletableFuture completes successfully with a value, the Mono will emit that value as an onNext() signal followed by an onComplete() signal.
   * If the CompletableFuture completes without a value (i.e., a CompletableFuture<Void>), the Mono will simply emit an onComplete() signal.
5. **Error Propagation:** If the CompletableFuture completes exceptionally (due to an error), the Mono will emit an onError(Throwable) signal with the exception from the CompletableFuture.

**Relationship with CompletableFuture:**

* **Interoperability:** Mono.fromFuture() is a crucial tool for integrating existing asynchronous operations that are based on CompletableFuture into a Reactor-based reactive pipeline. This allows you to leverage the benefits of Reactor's declarative style, operator chaining, and backpressure handling with asynchronous computations managed by CompletableFuture.
* **Reactive Wrapper:** It essentially wraps a CompletableFuture within a reactive Mono, making its asynchronous result consumable within a Reactor flow.
* **No Direct Control:** Once the CompletableFuture is passed to Mono.fromFuture(), Reactor observes its state. Reactor doesn't directly control the execution or completion of the CompletableFuture itself. The CompletableFuture is typically managed by the code that created it.

**Common ForkJoinPool:**

The java.util.concurrent.ForkJoinPool is an implementation of the ExecutorService interface that is particularly well-suited for parallel tasks that can be broken down into smaller, recursive subtasks (fork-join parallelism). It uses a work-stealing algorithm to efficiently distribute and execute these tasks across a pool of threads.

**How CompletableFuture Relates to ForkJoinPool (by Default):**

By default, many of the asynchronous operations provided by CompletableFuture (like thenApplyAsync(), thenComposeAsync(), etc.) that require execution on a separate thread will utilize the **common ForkJoinPool**.

* **Shared Resource:** The common ForkJoinPool is a static, JVM-wide thread pool that is shared by all CompletableFuture operations that don't explicitly specify an Executor.
* **Convenience and Efficiency:** This default behavior provides a convenient way to execute asynchronous tasks without the need for explicit thread management. The ForkJoinPool is designed to be efficient for compute-intensive, fork-join style tasks.

**Implications for Mono.fromFuture() and ForkJoinPool:**

When you use Mono.fromFuture() with a CompletableFuture that internally relies on the common ForkJoinPool for its asynchronous execution, the reactive Mono will indirectly be tied to this pool.

* **No Direct Reactor Control over Threading:** Reactor itself has its own mechanisms for managing concurrency (e.g., Schedulers). 1 When you use Mono.fromFuture(), the threading of the underlying CompletableFuture is managed by the CompletableFuture and its associated Executor (which is often the common ForkJoinPool by default).

[1. Context Propagation with Project Reactor 2 - The bumpy road of Spring Cloud Sleuth](https://spring.io/blog/2023/03/29/context-propagation-with-project-reactor-2-the-bumpy-road-of-spring-cloud/" \t "_blank)

[[Source icon](https://spring.io/blog/2023/03/29/context-propagation-with-project-reactor-2-the-bumpy-road-of-spring-cloud/)](https://spring.io/blog/2023/03/29/context-propagation-with-project-reactor-2-the-bumpy-road-of-spring-cloud/" \t "_blank)

[spring.io](https://spring.io/blog/2023/03/29/context-propagation-with-project-reactor-2-the-bumpy-road-of-spring-cloud/" \t "_blank)

* **Potential for Blocking Issues:** If the CompletableFuture performs long-blocking operations on a thread from the common ForkJoinPool, it can potentially impact the performance of other tasks also using the same pool. For I/O-bound or long-blocking operations, it's generally recommended to use a dedicated ExecutorService optimized for that purpose and provide it explicitly to the CompletableFuture's asynchronous methods.
* **Context Propagation:** Reactor's context propagation might not automatically flow into the execution context of the CompletableFuture if it's running on the common ForkJoinPool. You might need to take explicit steps to propagate context if required.

**Best Practices:**

* **Be Mindful of Blocking:** If the CompletableFuture passed to Mono.fromFuture() performs blocking operations, consider executing those operations on a dedicated, appropriately sized thread pool to avoid starving the common ForkJoinPool or Reactor's event loops.
* **Explicit Executors:** For better control over threading and resource management, consider creating and managing your own ExecutorService instances and providing them to the asynchronous methods of CompletableFuture.
* **Context Awareness:** If your reactive flow relies on Reactor's context, be aware that it might not automatically propagate to the CompletableFuture's execution context. You might need to use techniques like Context.current() and pass it explicitly if needed.

**In Summary:**

Mono.fromFuture() is a vital tool for integrating CompletableFuture-based asynchronous operations into Reactor. It wraps the eventual result or error of a CompletableFuture into a reactive Mono. By default, CompletableFuture often utilizes the common ForkJoinPool for its asynchronous tasks. While convenient, be aware of the potential implications of shared thread pools and blocking operations when combining these technologies. Consider using dedicated ExecutorService instances for better control and performance, especially for I/O-bound or long-blocking tasks.