Async and Await. Asynchronous programming in depth – Questions and Answers

1. Does using the “async” keyword on a method force all invocations of that method to be asynchronous?

No. When you invoke a method marked as “async”, it begins running synchronously on the current thread. So, if you have a synchronous method that returns void and all you do to change it is mark it as “async”, invocations of that method will still run synchronously. This is true regardless of whether you leave the return type as “void” or change it to “Task”. Similarly, if you have a synchronous method that returns some TResult, and all you do is mark it as “async” and change the return type to be “Task<TResult>”, invocations of that method will still run synchronously.

Marking a method as “async” does not affect whether the method runs to completion synchronously or asynchronously. Rather, it enables the method to be split into multiple pieces, some of which may run asynchronously, such that the method may complete asynchronously. The boundaries of these pieces can occur only where you explicitly code one using the “await” keyword, so if “await” isn’t used at all in a method’s code, there will only be one piece, and since that piece will start running synchronously, it (and the whole method with it) will complete synchronously.

2. Can I mark any method as “async”?

No, Constructors, property accessors and event accessors cannot be `async`.

3. In which cases can we use void return type in async functions?

C# also supports `void` return type for `async` functions. It can be helpful when you want to create an async event handler.

4. Is “await task;” the same thing as “task.Wait()”?

No.
“task.Wait()” is a synchronous, potentially blocking call: it will not return to the caller of Wait() until the task has entered a final state, meaning that it’s completed in the RanToCompletion, Faulted, or Canceled state. In contrast, “await task;” tells the compiler to insert a potential suspension/resumption point into a method marked as “async”, such that if the task has not yet completed when it’s awaited, the async method should return to its caller, and its execution should resume when and only when the awaited task completes. Using “task.Wait()” when “await task;” would have been more appropriate can lead to unresponsive applications and deadlocks.

5. What are the benefit of using async/await with I/O-bound operations?

This model works well with a typical server scenario workload. Because there are no threads dedicated to blocking on unfinished tasks, the server thread pool can service a much higher volume of web requests.

Consider two servers: one that runs async code, and one that does not. For the purpose of this example, each server only has 5 threads available to service requests. Note that these numbers are imaginarily small and serve only in a demonstrative context.

Assume both servers receive 6 concurrent requests. Each request performs an I/O operation. The server without async code has to queue up the 6th request until one of the 5 threads have finished the I/O-bound work and written a response. At the point that the 20th request comes in, the server might start to slow down, because the queue is getting too long.

The server with async code running on it still queues up the 6th request, but because it uses async and await, each of its threads are freed up when the I/O-bound work starts, rather than when it finishes. By the time the 20th request comes in, the queue for incoming requests will be far smaller (if it has anything in it at all), and the server won't slow down.

Although this is a contrived example, it works in a very similar fashion in the real world. In fact, you can expect a server to be able to handle an order of magnitude more requests using `async` and `await` than if it were dedicating a thread for each request it receives.

The biggest gain for using `async` and `await` for a client app is an increase in responsiveness. Although you can make an app responsive by spawning threads manually, the act of spawning a thread is an expensive operation relative to just using `async` and `await`. Especially for something like a mobile game, impacting the UI thread as little as possible where I/O is concerned is crucial.

More importantly, because I/O-bound work spends virtually no time on the CPU, dedicating an entire CPU thread to perform barely any useful work would be a poor use of resources.

Additionally, dispatching work to the UI thread (such as updating a UI) is very simple with `async` methods, and does not require extra work (such as calling a thread-safe delegate).

6. What are the benefit of using async/await with CPU-bound operations?

The use of **async**/**await** provides you with a clean way to interact with a background thread and keep the caller of the async method responsive.

Note that this does not provide any protection for shared data. If you are using shared data, you will still need to **apply an appropriate synchronization strategy**.

7. How async actions helps to increase throughput of the system?

When a request arrives, a thread from the pool is dispatched to process that request.

If the request is processed synchronously, the thread that processes the request is busy while the request is being processed, and that thread cannot service another request.

An asynchronous request takes the **same amount of time** to process as a synchronous request. However, a **thread is not blocked** from responding to other requests.

Asynchronous requests prevent **request queuing** and **thread pool growth** when there are many concurrent requests that invoke long-running operations.