**Slide 2**

* For this example, let’s say the requests in the system depend on some external resource, like a database or Web API. When a request comes in, ASP.NET takes one of its thread pool threads and assigns it to that request. Because it’s written synchronously, the request handler will call that external resource synchronously. This blocks the request thread until the call to the external resource returns. Figure 1 illustrates a thread pool with two threads, one of which is blocked waiting for an external resource.
* Eventually, that external resource call returns, and the request thread resumes processing that request. When the request is complete and the response is ready to be sent, the request thread is returned to the thread pool.
* This is all well and good—until your ASP.NET server gets more requests than it has threads to handle. At this point, the extra requests have to wait for a thread to be available before they can run.
* In this situation, the first two requests are assigned threads from the thread pool. Each of these requests calls an external resource, blocking their threads. The third request has to wait for an available thread before it can even start processing, but the request is already in the system. Its timer is going, and it’s in danger of an **HTTP Error 503** (Service unavailable).
* But think about this for a second: That third request is waiting for a thread, when there are two other threads in the system effectively doing nothing. Those threads are just blocked waiting for an external call to return. They’re not doing any real work; they’re not in a running state and are not given any CPU time. Those threads are just being wasted while there’s a request in need. This is the situation addressed by asynchronous requests.

**Slide 3**

* Asynchronous code can scale further than blocking threads because it uses much less memory; every thread pool thread on a modern OS has a **1MB** stack, plus an **unpageable** kernel stack. That doesn’t sound like a lot until you start getting a whole lot of threads on your server. In contrast, the memory overhead for an asynchronous operation is much smaller. So, a request with an asynchronous operation has much less memory pressure than a request with a blocked thread. Asynchronous code allows you to use more of your memory for other things (caching, for example).
* Asynchronous code can scale faster than blocking threads because the thread pool has a limited **injection rate**. Currently, the rate is one thread every two seconds. This injection rate limit is a good thing; it avoids constant thread construction and destruction. However, consider what happens when a sudden flood of requests comes in. Synchronous code can easily get bogged down as the requests use up all available threads and the remaining requests have to wait for the thread pool to inject new threads. On the other hand, asynchronous code doesn’t need a limit like this; it’s “always on,” so to speak. Asynchronous code is more responsive to sudden swings in request volume.

**Slide 4**

**Slide 6**

* as a method marked with contextual keyword async. It doesn't necessarily mean that the method executes asynchronously.
* It doesn't mean that the method is asynchronous at all. It only means that the compiler performs some special transformation to the method.
* Asynchronous methods look something like this:

**async operator**

- is applied to a method to indicate it is asynchronous

- It does not run this method on a thread pool thread

- It runs synchronously until it hits the await

**await operator**

- is applied to task in an asynchronous method to suspend execution of the method until the awaited task completes

**Task**: unit of work with a promise to give you results back in the future; That promise could be backed by IO-operation or represent a computation-intensive operation.

Results of the operation is \*\*self-sufficient\*\* and is a \*\*first-class citizen\*\*

- You can store it in a variable

- Return it from a method or pass it to another method

- You can join the results together to form one

- You can wait for results synchronously or you can "await" the result by adding "continuation" to the "future"

- It takes an argument, an awaitable (an asynchronous operation)

- examines the awaitable to see if it is already completed.

- If the awaitable is not complete, then it acts asynchronously by telling the awaitable to run the remainder of the method when it completes, and then returns the async method.

- Later on, when the awaitable completes, it will execute the remainder of the async method on a context that was captured before the await returned.

**Slide 7**

**Awaitable**

* Task.Yield: returns awaitables that are not tasks
* Task.FromResult - Task object is already computed

**Return Types**

* Async methods can return Task<T>, Task, or void. In almost all cases, you want to return Task<T> or Task, and return void only when you have to.
* Task and Task<T> are awaitable while void is not, however Task acts as a void because it does not return a value.
* You have to return void when you have async event handlers.
* To return a value, the method must be of Task<T> and must return a value of type T.

**Slide 8**

* when you await a built-in awaitable, then the awaitable will capture the current "context" and later apply it to the remainder of the async method
* If SynchronizationContext.Current is not null, then it’s the current SynchronizationContext (UI or ASP.NET context)
* Otherwise it’s a TaskScheduler.Default (thread pool context)
* Most async methods will be designed with composition in mind: they await other operations, and each one represents an asynchronous operation itself.
* If you want to tell the awaiter to not capture the current context, you call the ConfigureAwait and pass false
* The important thing to note is that each level of async method calls has its own context
* A good rule of thumb is to use ConfigureAwait(false) unless you know you do need the context

**Slide 9**

* async method also offers the ability to start several operations and await for one (or all) to complete

**Slide 10**

* The top-level method calls GetJsonAsync (within the UI/ASP.NET context).
* GetJsonAsync starts the REST request by calling HttpClient.GetStringAsync (still within the context).
* GetStringAsync returns an uncompleted Task, indicating the REST request is not complete.
* GetJsonAsync awaits the Task returned by GetStringAsync. The context is captured and will be used to continue running the GetJsonAsync method later. GetJsonAsync returns an uncompleted Task, indicating that the GetJsonAsync method is not complete.
* The top-level method synchronously blocks on the Task returned by GetJsonAsync. This blocks the context thread.
* Eventually, the REST request will complete. This completes the Task that was returned by GetStringAsync.
* The continuation for GetJsonAsync is now ready to run, and it waits for the context to be available so it can execute in the context.
* Deadlock. The top-level method is blocking the context thread, waiting for GetJsonAsync to complete, and GetJsonAsync is waiting for the context to be free so it can complete.

**Slide 12**

* Is this code asynchronous? Yes. Is it a correct way to write asynchronous code? No. The UI thread starts a new CPU-bound thread at “1” and returns. The new thread then starts a new I/O thread at “2” and falls to sleep waiting for its completion.
* So, what happens here? Instead of creating just an I/O thread we create both CPU thread at “1” and I/O thread at “2”. It’s a waste of thread

**Slide 13**

* If you are developing a third-party library, it is always vital to configure await in such a way that the rest of the method will be executed by an arbitrary thread from the thread pool.
* You can slightly increase performance by allowing CLR to execute your code by any thread from the thread pool
* by using the default implementation (or explicitly writing ConfigureAwait(true)), you leave a hole for possible deadlocks.

**Slide 14**

* There are three possible return types for async methods: Task, Task<T> and void, but the natural return types for async methods are just Task and Task<T>.
* When converting from synchronous to asynchronous code, any method returning a type T becomes an async method returning Task<T>,
* and any method returning void becomes an async method returning Task
* Async void methods have different error-handling semantics. When an exception is thrown out of an async Task or async Task<T> method, that exception is captured and placed on the Task object.
* Async void methods have different composing semantics. Async methods returning Task or Task<T> can be easily composed using await, Task.WhenAny, Task.WhenAll and so on. Async methods returning void don’t provide an easy way to notify the calling code that they’ve completed. It’s easy to start several async void methods, but it’s not easy to determine when they’ve finished. Async void methods will notify their SynchronizationContext when they start and finish, but a custom SynchronizationContext is a complex solution for regular application code.
* Async void methods are difficult to test. Because of the differences in error handling and composing, it’s difficult to write unit tests that call async void methods. The MSTest asynchronous testing support only works for async methods returning Task or Task<T>. It’s possible to install a SynchronizationContext that detects when all async void methods have completed and collects any exceptions, but it’s much easier to just make the async void methods return Task instead

Conclusion

- As you can see asynchronous programming has many benefits but does add complexity to your code. Like all programming patterns it is best to weigh the benefits against the added complexity to see if it is the right fit for your application

<http://enterprisecraftsmanship.com/2014/12/20/pitfalls-of-async-await/>

<https://blog.stephencleary.com/2012/07/dont-block-on-async-code.html>

https://msdn.microsoft.com/en-us/magazine/jj991977.aspx