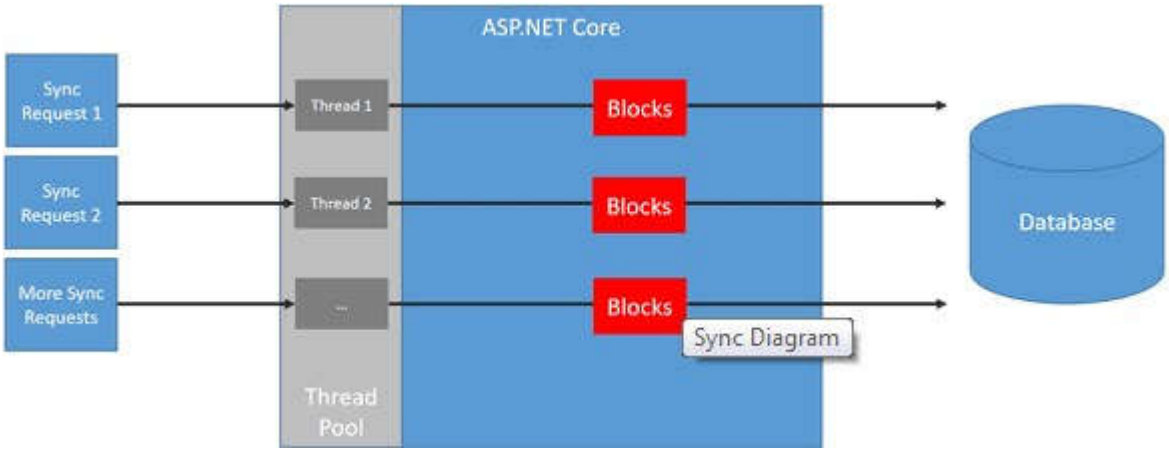
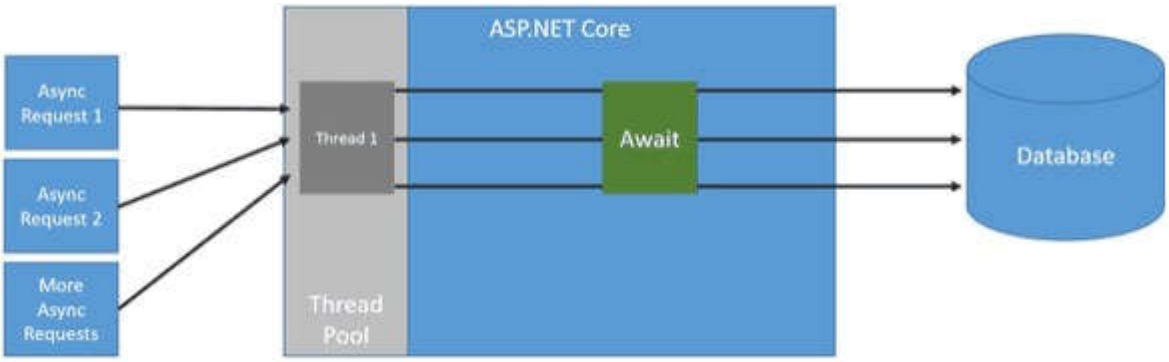
For synchronous API code, when a request is made to the API, a thread from the thread pool will handle the request. If the code makes an I/O call (like a database call) synchronously, the thread will block until the I/O call has finished. The blocked thread can't be used for any other work, it simply does nothing and waits for the I/O task to finish. If other requests are made to our API whilst the other thread is blocked, different threads in the thread pool will be used for the other requests.



There is some overhead in using a thread - a thread consumes memory and it takes time to spin a new thread up. So, really we want our API to use as few threads as possible.

If the API was to work in an asynchronous manner, when a request is made to our API, a thread from the thread pool handles the request (as in the synchronous case). If the code makes an asynchronous I/O call, the thread will be returned to the thread pool at the start of the I/O call and then be used for other requests.

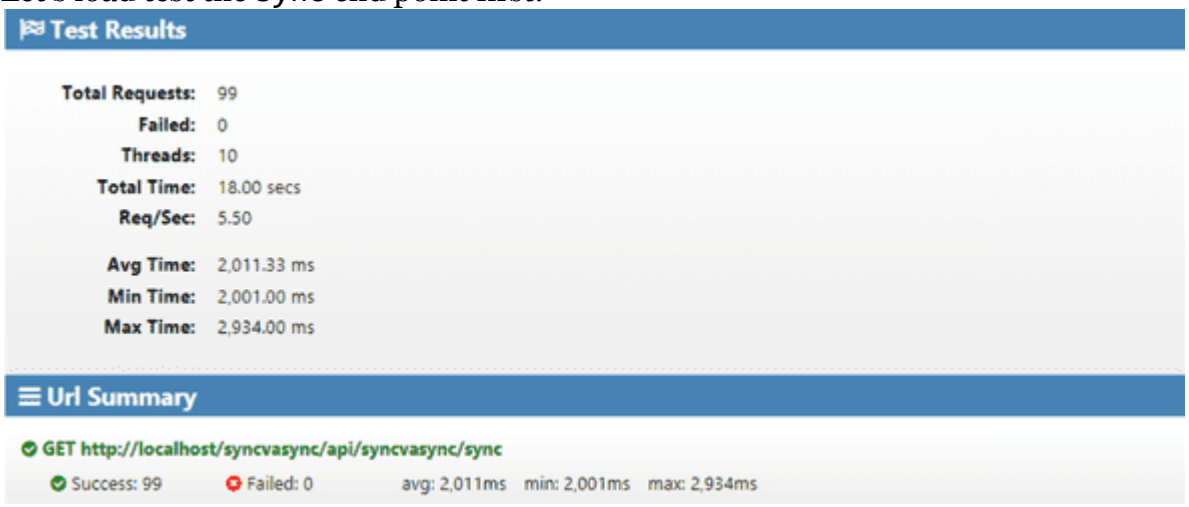


So, making operations asynchronous will allow our API to work more efficiently with the ASP.NET Core thread pool. So, in theory, this should allow our API to scale better

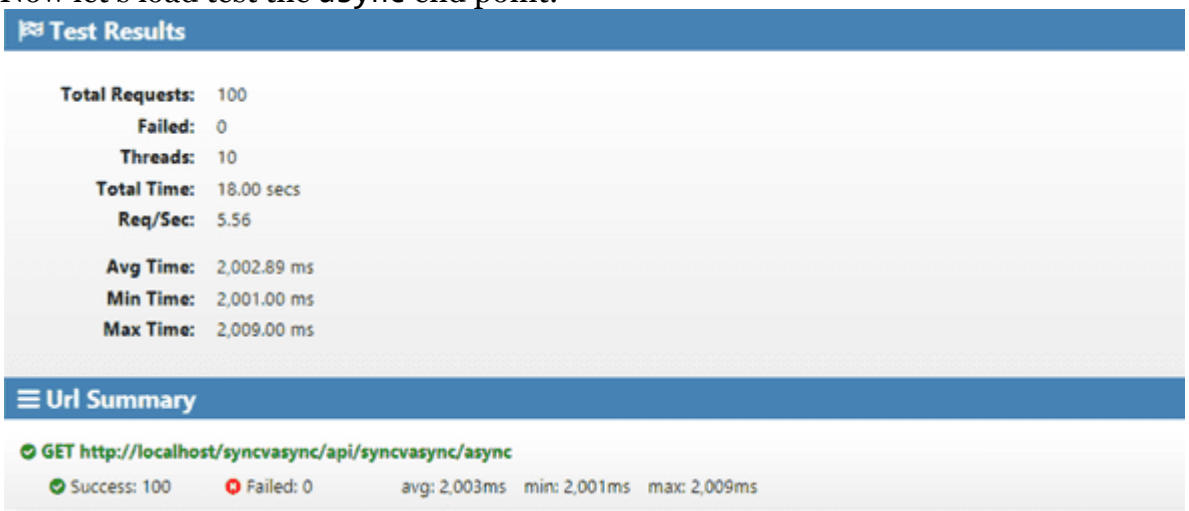
```
[Route("api/syncvasync")]
public class SyncVAsyncController : Controller
{
    private readonly string _connectionString;
    public SyncVAsyncController(IConfiguration configuration)
    {
        _connectionString =
configuration.GetConnectionString("DefaultConnection");
    }

    [HttpGet("sync")]
    public IActionResult SyncGet()
    {
        using (SqlConnection connection = new
SqlConnection(_connectionString))
        {
            connection.Open();

            connection.Execute("WAITFOR DELAY '00:00:02';");
        }

        return Ok();
    }

    [HttpGet("async")]
    public async Task<IActionResult> AsyncGet()
    {
        using (SqlConnection connection = new
SqlConnection(_connectionString))
        {
            await connection.OpenAsync();
```

```
            await connection.ExecuteAsync("WAITFOR DELAY
'00:00:02';");
        }

        return Ok();
    }
}
```

Let's load test the sync end point first:



Now let's load test the async end point:



So, we have better results for the async end point, but only just!

Let's throttle the thread pool:

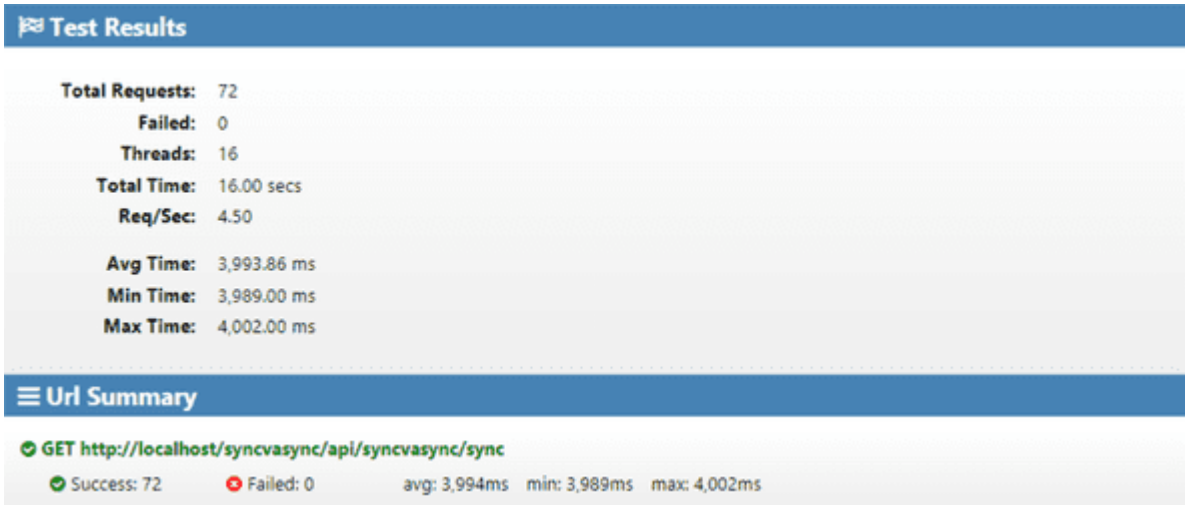
```
public class Program
{
    public static void Main(string[] args)
    {
        ...
 int processorCounter = Environment.ProcessorCount; // 8 on my PC
bool success = ThreadPool.SetMaxThreads(processorCounter,
processorCounter);        ...
    }
}
```

Let's also return the available threads in the thread pool in the responses:

```
[HttpGet("sync")]
public IActionResult SyncGet()
{
    using (SqlConnection connection = new
SqlConnection(_connectionString))
    {
        connection.Open();

        connection.Execute("WAITFOR DELAY '00:00:02';");
    }

    return Ok(GetThreadInfo());}

[HttpGet("async")]
public async Task<IActionResult> AsyncGet()
{
    using (SqlConnection connection = new
SqlConnection(_connectionString))
    {
        await connection.OpenAsync();
```

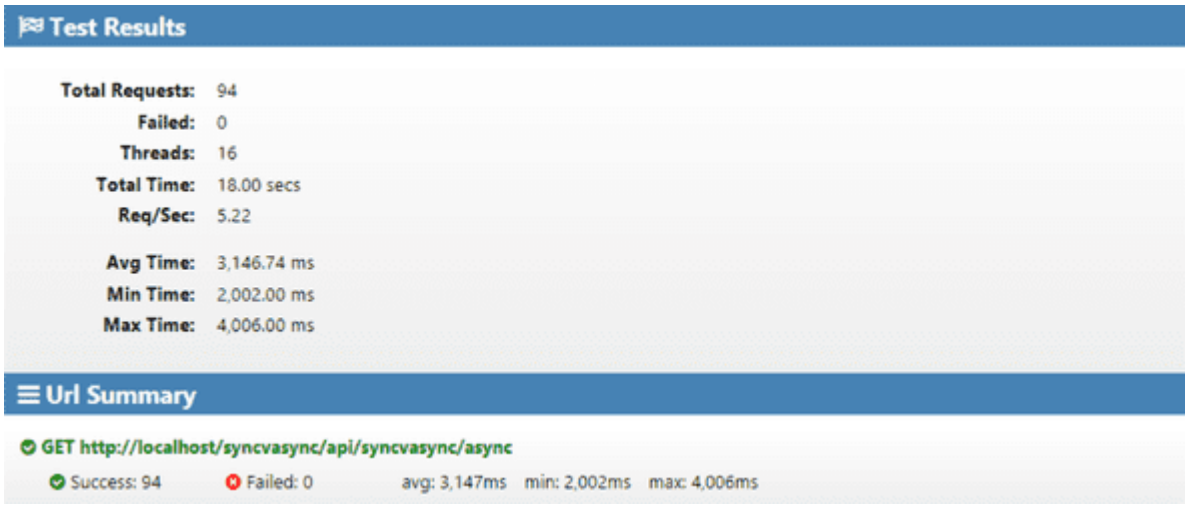
```
        await connection.ExecuteAsync("WAITFOR DELAY '00:00:02';");
    }

    return Ok(GetThreadInfo());}

private dynamic GetThreadInfo(){
  int availableWorkerThreads;
 int availableAsyncIOThreads;
 ThreadPool.GetAvailableThreads(out availableWorkerThreads, out
availableAsyncIOThreads);
  return new { AvailableAsyncIOThreads = availableAsyncIOThreads,
AvailableWorkerThreads = availableWorkerThreads };}
```

Let's load test the sync end point again now that the thread pool has been throttled:



We can see from the responses that all the threads in thread pool are being used:

Load testing the async end point again shows it is more efficient than the sync end point:



We can see from the responses that not all the threads in thread pool are being used - the thread pool is being used more efficiently:

Conclusion

When the API is stressed, async action methods will give the API some much needed breathing room whereas sync action methods will deteriorate quicker.

Async code doesn't come for free - there is additional overhead in context switching, data being shuffled on and off the heap, etc which is why async code can be a bit slower than the equivalent sync code if there is plenty of available threads in thread pool. This difference is usually very minor though.

It's a good idea to write async action methods that are I/O bound even if the API is only currently dealing with a low amount of usage. It only takes typing an extra keyword per I/O call and the usage can grow.

The asynchronous version will always be slower than the synchronous version when there is no concurrency.

- It's doing all of the same work as the non-async version, but with a small amount of overhead added to manage the asynchrony.
- Each request will be slower, but if you make 1000 requests at the same time, the asynchronous implementation will be able to handle them all more quickly (at least in certain circumstances).
- It happens because the asynchronous solution allows the thread that was allocated to handle the request to go back to the pool and handle other requests, whereas the synchronous solution forces the thread to sit there and do nothing while it waits for the asynchronous operation to complete.
- There is overhead in structuring the program in a way that allows the thread to be freed up to do other work, but the advantage is the ability of that thread to do another job.
- If your program has no other work for that thread, then it ends up being a net loss.
- Async/await saves time only when the job is I/O-bound. Any application's job that is CPU-bound will introduce some performance hits.
- That's because if you have some computations that take 10s on your CPU(s), then adding async/await will add X extra time to that 10s for task creation, scheduling and synchronization to get the job done.
- The performance often hits due to introducing async/await are not that large especially if you are careful not to overdo it.