Module 0335: What is 'await'?

Tak Auyeung, Ph.D.

October 10, 2021

Contents

1 About this module

- Prerequisites:
- Objectives: This module explains the important concept of 'await' in the context of Node.

2 An analogy

Let's think about a restaurant (analoguous to a server environment). In this context, a customer represents a HTTP client. A waiter is an instance of a HTTP server (your Express+node app). The kitchen represents a "back-end" server.

When there is no customer, the waiter and the kitchen both have nothing to do.

Upon the arrival of a customer, the waiter takes the order (in the form of an HTTP request). You can look at an "end-point" as a particular item that the restaurant can do for its customers. Obviously, most of the time, most end-points correspond to dishes.

However, some end-points may correspond to simple information that a waiter can respond without delay, such as inquries of opening and closing times.

Most requests, however, take time to process. For example, a medium rare steak may take the kitchen a good amount of time to prepare.

2.1 Taking an order that is slow to prepare

Let us look at things from the perspective of the waiter because the waiter represents an Express+node app. A customer walks in and order a medium rare steak, what is the waiter going to do?

2.1.1 Option 1: synchronous processing

In PHP and many other scripting languages, the waiter takes the order (parses the URL for GET or POST parameters), and go to the kitchen (a database server, for example), and inform the kitchen that a customer would like to have a medium rare steak.

And then the waiter simply waits for the kitchen to finish preparing the steak.

20 minutes later, the kitchen finishes preparing the steak, hand the steak to the waiter, and then the waiter in return serves the customer.

When the waiter was waiting for the kitching to prepare the steak, the waiter cannot do anything else. The waiter cnanot serve another customer (representing another incoming HTTP request). Even if there are a few other things that the waiter could have done with the same customer, such as serving water, the waiter cannot do that.

2.1.2 Option 2: asynchronous processing

In Express+node, the waiter sends the steak order to the kitchen on a piece of paper. In addition to the steak order itself, this piece of paper also includes additional information, such as which customer originates this order, and a method to notify the waiter when the dish is ready.

As soon as this piece of paper is passed on to the kitchen, the waiter is again available to do other things. For example, the waiter can now seat another customer (accepting an HTTP request), or continue to serve additional dishes or some other customers in the restaurant.

When the kitchen is done with the said steak order, it pages the waiter and informs the waiter that the steak is done. As soon as the waiter is ready, the waiter goes to the kitchen to pick up the finished steak order. Then the waiter can now resume serving the customer who ordered the steak in the first place.

2.1.3 General idea

The general idea is that Expres+node only has one waiter to serve potentially many customers (each customer corresponds to an HTTP connection/request). This is how a cost-effective restaurant operates.

On the other hand, languages like PHP rely on allocating one waiter per customer. Such an approach means a restaurant can hire waiters who do not multitask. However, the overhead of having a large number of waiters is expensive. Furthermore, this approach relies on the ability of a restaurant (the server environment, namely the operating system and the HTTP server daemon) being able to create a new waiter when a customer walks in.

The Express+node method is far more efficient and portable. In fact, the Express+node method relies little on the server environment. As such, Express+node apps can potentially run in a wide variety of environments.

In order for this single-waiter method to work, the waiter has to be able to identify steps that may take a long time.

For any steps that may take a long time, the waiter needs the following:

- a way to start the step.
- a way to be notified when the step is completed
- and a way to remember what to continue to do after the step is completed

Through the relatively brief history Express/node, several mechanisms were used to this end:

- In the beginning, the concept of "call-back" functions was used. Suffice to say that this method worked, but it also causes some major issues with the expression of sequential logic.
- Next, the concept of "Promises" was used to encapsulate the lower-level mechanism of "call-back" functions. The use of Promises is an object-oriented approach, and it affords a great deal of flexibilities. *However*, the specification of sequential steps continue to be quite obscure.

- In a later specification of ECMAScript (2017), the concepts of `await` and `async` were added as "syntactic eye candy" of Promises.

It is important to note that call-back functions, at the lowest level, form the foundation of Express+node handling multiple requests efficiently. Promises are constructs built on top of call-back functions, and then `async`/`await` are constructs that are built on top of Promises.

3 An non-Express illustration

To understand the involved concepts, let us example a simple node script that is not in an Express context.

3.1 No delay

We will start with the following code to simply print "5, 4, 3, 2, 1, Thunderbirds are go!"

Listing 1: Boring without pauses

```
console.log(5)
console.log(4)
console.log(3)
console.log(2)
console.log(1)
console.log('Thunderbirds are go!')
```

This is no fun! The program quickly prints 6 lines of text and it is done! We want to add the drama of pausing one second between printing the lines.

In this example, we are *intentionally* introducing a specific delay. However, in an Express+node script, the intention delay is replaced by lengthy tasks that cannot be performed by the Express+node script. For example, accessing a database via queries is a common task that takes a bit to time to complete. For complex apps, using a RESTful API to communicate with another web server is also an example of a lengthy task.

The mechanism to introduced an artificial delay and to perform a lengthy task is the same, and that is the reason why we are exploring this example.

Some people may think of using a loop that counts to a very large number as a delay mechanism. There are many problems with this "busy delay loop" approach. It uses up a lot of processor cycles, and the actual amount of delay depends on many factors. Suffice to say a busy delay loop is not an acceptable solution.

3.2 Using functions

node offers a delay mechanism `setTimeout`. In the simplest form, `setTimeout` use a provided function that requires two pieces of information:

- a call back function that is called by node, and
- the amount of milliseconds (ms) to set the timer before the said call back function is called

This means that *each step* that is delayed needs to be a function. Without the use of anonymous functions, now we have the following code to get ready for the delay mechanism:

Listing 2: Still boring and fragmented

```
function tb5()
{
  console.log(5)
}

function tb4()
{
  console.log(4)
}
```

```
function tb3()
{
  console.log(3)
}

function tb2()
{
  console.log(2)
}

function tb1()
{
  console.log(1)
}

function tbGo()
{
  console.log('Thunderbirds are go!')
}

tb5()
tb4()
tb3()
tb2()
tbGo()
```

This script does the same thing as the previous one, but it now ready to utilize setTimeout to have a delay. The following is the code that has a 1 second delay between the printing of each line:

3.3 Using setTimeout and call-back functions

Listing 3: Using call-back to pause

```
function tb5()
{
  console.log(5)
  setTimeout(tb4, 1000)
}

function tb4()
{
  console.log(4)
  setTimeout(tb3, 1000)
}
function tb3()
{
  console.log(3)
  setTimeout(tb2, 1000)
}

function tb2()
{
  console.log(2)
  setTimeout(tb1, 1000)
}

function tb1()
{
```

```
  console.log(1)
  setTimeout(tbGo, 1000)
}

function tbGo()
{
  console.log('Thunderbirds are go!')
}

tb5()
```

This program does introduce the delay. However, you can see how the original sequential logic is not broken up into pieces. The use of Promises makes it possible to encapsulate the delay mechanism itself into a reusable manner.

3.4 Using Promises

Listing 4: Using Promises to pause

```
function tb5()
{
  console.log(5)
  return delay(1000)
}

function tb4()
{
  console.log(4)
  return delay(1000)
}

function tb3()
{
  console.log(3)
  return delay(1000)
}

function tb2()
{
  console.log(2)
  return delay(1000)
}

function tb1()
{
  console.log(1)
  return delay(1000)
}

function tbGo()
{
  console.log('Thunderbirds are go!')
}


function delay(msPeriod)
{
  function delayPromiseHandler(resolve, reject)
```

```
  {
    function timeoutHandler(delayMs)
    {
      resolve()
    }
    setTimeout(timeoutHandler, msPeriod)
  }
  return new Promise(delayPromiseHandler)
}

tb5().then(tb4).then(tb3).then(tb2).then(tb1).then(tbGo)
```

(You can download this script here.)

This program uses the Promise concept. While it is more obvious how the functions are "chained", the necessity to break up the steps into functions makes it difficult to follow the sequential logic.

3.5 Using async/await

Listing 5: Using async/await to pause

```
function delay(msPeriod)
{
  function delayPromiseHandler(resolve, reject)
  {
    function timeoutHandler(delayMs)
    {
      resolve()
    }
    setTimeout(timeoutHandler, msPeriod)
  }
  return new Promise(delayPromiseHandler)
}

async function main()
{
  console.log(5)
  await delay(1000)
  console.log(4)
  await delay(1000)
  console.log(3)
  await delay(1000)
  console.log(2)
  await delay(1000)
  console.log(1)
  await delay(1000)
  console.log("Thunderbirds are go!")
}

main()
```

(You can download this script here.)

This version utilizes the `async`/`await` constructs. It is quite obvious how much cleaner this version of code looks.

4 Async/await

`async` is a reserved word used to qualify a function *definition*. When a function `async` qualified, execution of its code can potentially pause when `await` is used to qualify a function *call*.

Note that `await` applies to a value, which potentially can be one that is returned by a call to a function. Technically, the reserved word `await` only has meaning when the value being `awaited` is a Promise object (in which case execution is paused until the Promise is resolved or rejected). However, `awaiting` a value that is not a promise simply does not do anything special, it has no ill effect.

However, *forgetting* to use `await` when one is needed can cause some strange behavior. Specifically, you may find some steps being performed out of order.

4.1 Technically, what does await do?

`await` instructs node to block (aka pause) execution in an `async function` until the (singular) `awaited` `Promise` object resolves. The resolution value of the `Promise` becomes the value of the `awaited` expression.

In general, a `Promise` object resolves when the underlying call-back `function` is called with potentially a resolution value.

As a result, `await` utlimately waits for a call-back function to be called. Call-back functions are usually called when a time consuming asychronous (not something to be performed by the node script) operation is completed.

4.2 What happens without "await"?

Let's say that there 3 operations. `op1`, `op2` and `op3` must be performed in this order due to dependency between the steps. Let us also awsume the 3 operations are all `async functions`.

The following code performs the operations, but without using `await` at all.

Listing 6: No await to synchronouse

```
function delay(msPeriod)
{
  function delayPromiseHandler(resolve, reject)
  {
    function timeoutHandler(delayMs)
    {
      resolve()
    }
    setTimeout(timeoutHandler, msPeriod)
  }
  return new Promise(delayPromiseHandler)
}

async function op1()
{
  return delay(Math.random()*100+300)
}

async function op2()
{
  return delay(Math.random()*100+300)
}

async function op3()
{
  return delay(Math.random()*100+300)
}

async function main()
{
  op1().then(() => console.log("op1 is done"))
  op2().then(() => console.log("op2 is done"))
  op3().then(() => console.log("op3 is done"))
```

```
  console.log("main is done")
}

main()
```

(You can download this script here.)

For now, pay no attention to the weird construct in the definition of the `main function`. The `.then(...)` method is only there so that we can see when an operation is completed.

The definitions of op1, op2 and op3 make use of a random number to determine how much time to delay. The actual amount of time to delay (for each instance of running this program) is a random number between 300ms and 399ms. This is done to emulate the unpredicatable nature of how long it may take to complete an asynchronous operation.

Running this program reveals the strange nature of asynchronous operations.

4.2.1 'main' is done first!

First of all, `main` is actually completed *first*! This is because without `await`, the *calls* to op1, op2 and op3 merely informs node to "when you get a chance, start the operation." In other words, this `main function` is analogous to a waiter dropping off 3 orders of dishes with the kitchen where each dish takes a long time to prepare.

As far as the waiter is concerned, dropping off the orders is quick, and that is all to be done in `main`!

4.2.2 There is no predictable ordering of the completion of op1, op2 and op3

Try to run this code several times. Except for some unlikely coincidence, you will find that each time you run this program, the ordering of the completion of op1, op2 and op3 is unpredictable.

This because all 3 operations are started approximately at the same time (differing maybe by hundreds of microseconds). This program specifically varies the completion time of each operation to emulate the nature of asynchronous calls.

4.3 Awaiting at the wrong places

The following code is one attempt to fix the problem.

Listing 7: Awaits after all operations are started

```
function delay(msPeriod, name)
{
  function delayPromiseHandler(resolve, reject)
  {
    function timeoutHandler(delayMs)
    {
      resolve()
    }
    console.log(`${name} is started`)
    setTimeout(timeoutHandler, msPeriod)
  }
  return new Promise(delayPromiseHandler)
}

async function op1()
{
  return delay(Math.random()*100+300, 'op1')
}

async function op2()
{
  return delay(Math.random()*100+300, 'op2')
}
```

```
async function op3()
{
  return delay(Math.random()*100+300, 'op3')
}

async function main()
{
  let op1Promise = op1()
  let op2Promise = op2()
  let op3Promise = op3()

  await op1Promise
  console.log('op1 is done')
  await op2Promise
  console.log('op2 is done')
  await op3Promise
  console.log('op3 is done')
  console.log("main is done")
}

main()
```

(You can download this script here.)

When this code is run, you will find that all operations are started closely next to each other, and this time, they are completed in the right order. It *appears* that the problem is solved, but that is not the case.

Recall that we assumed interdependencies between the operations. This means that op2 should not start until op1 is completed, and similarly for op2 and op3. Having op2 started before the confirmed completion of op1 is incorrect.

Note that in this version, the `delay function` is changed slightly so that a name can be printed to identify which operation has started.

This program illustrates two important points. Just the use of `await` may not solve the problem of having operations sequenced. However, it also indicates the placement of `await` can potentially allow asynchronous operations to occur in parallel when it makes sense (just not in this program based on our assumptions).

4.4 Awaiting each sequential step

By moving statement a little bit, we end up with the following code.

Listing 8: Awaits between steps to synchronouse operations

```
function delay(msPeriod, name)
{
  function delayPromiseHandler(resolve, reject)
  {
    function timeoutHandler(delayMs)
    {
      resolve()
    }
    console.log(`${name} is started`)
    setTimeout(timeoutHandler, msPeriod)
  }
  return new Promise(delayPromiseHandler)
}

async function op1()
{
```

```
  return delay(Math.random()*100+300, 'op1')
}

async function op2()
{
  return delay(Math.random()*100+300, 'op2')
}

async function op3()
{
  return delay(Math.random()*100+300, 'op3')
}

async function main()
{
  let op1Promise = op1()
  await op1Promise
  console.log('op1 is done')

  let op2Promise = op2()
  await op2Promise
  console.log('op2 is done')

  let op3Promise = op3()
  await op3Promise
  console.log('op3 is done')

  console.log("main is done")
}

main()
```

(You can download this script here.)

Note how the calls to `op1`, `op2` and `op3` are *after* the await of the `Promise` of the previous operation. This makes sense because the calling of `op1` is what *starts* the operation associated with `op1`, but the call actually returns right away to perform whatever is after the call to `op1` in `main`.

It is the `await` that blocks further execution until the `Promise` returned by `op1` is resolved. Recall that when a `Promise` is resolved, it usually means the underlying call-back function has been resolved (aka concluded).