If you’re reading this blog, you probably have some familiarity with asynchronous programming in JavaScript, and you may be wondering how it works in TypeScript.

Since TypeScript is a superset of JavaScript, async/await works the same, but with some extra goodies and type safety. TypeScript enables you to type-safe the expected result and even type-check errors, which helps you detect bugs earlier on in the development process.

async/await is essentially a [syntactic sugar](https://medium.com/@matt.mcalister93/async-and-await-syntactic-sugar-for-promises-in-javascript-aee7ace36d14) for promises, which is to say the async/await keyword is a wrapper over promises. An async function always returns a promise. Even if you omit the Promise keyword, the compiler will wrap your function in an immediately resolved promise.

Allow me to demonstrate:

const myAsynFunction = async (url: string): Promise<T> => {

const { data } = await fetch(url)

return data

}

const immediatelyResolvedPromise = (url: string) => {

const resultPromise = new Promise((resolve, reject) => {

resolve(fetch(url))

})

return resultPromise

}

Although they look totally different, the code snippets above are more or less equivalent. [Async/await](https://blog.logrocket.com/promise-chaining-is-dead-long-live-async-await-445897870abc/) simply enables you to write the code in a more synchronous manner and unwraps the promise in-line for you. This is powerful when you’re dealing with complex asynchronous patterns.

To get the most out of the async/await syntax, you’ll need a basic understanding of promises. Let’s take a closer look at Promises on a fundamental level.

**What is a promise in TypeScript?**

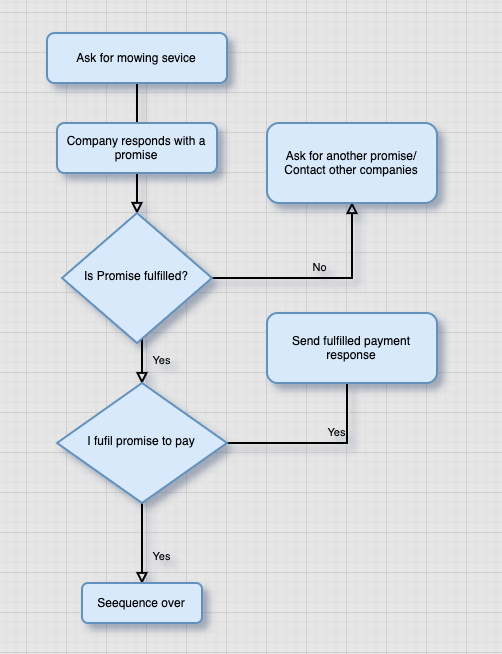
According to [Lexico](https://www.lexico.com/en/definition/promise" \t "_blank), a promise, in the English language, is “a declaration or assurance that one will do a particular thing or that a particular thing will happen.” In JavaScript, a promise refers to the expectation that something will happen at a particular time, and your app relies on the result of that future event to perform certain other tasks.

To show what I mean, I’ll break down a real-world example and commute it into pseudocode and then actual TypeScript code.

Let’s say I have a lawn to mow. I contact a mowing company that promises to mow my lawn in a couple of hours. I, in turn, promise to pay them immediately afterward, provided the lawn is properly mowed.

Can you spot the pattern? The first obvious thing to note is that the second event relies entirely on the previous one. If the first event’s promise is fulfilled, the next event’s will execute. The promise in that event is then either fulfilled or rejected or remains pending.

Let’s look at this sequence step by step and then code it out.



**The promise syntax**

Before we write out the full code, it makes sense to examine the syntax for a promise — specifically, an example of a promise that resolves into a string.

We declared a promise with the new + Promise keyword, which takes in the resolve and reject arguments. Now let’s write a promise for the flow chart above.

// I send a request to the company. This is synchronous

// company replies with a promise

const angelMowersPromise = new Promise<string>((resolve, reject) => {

// a resolved promise after certain hours

setTimeout(() => {

resolve('We finished mowing the lawn')

}, 100000) // resolves after 100,000ms

reject("We couldn't mow the lawn")

})

const myPaymentPromise = new Promise<Record<string, number | string>>((resolve, reject) => {

// a resolved promise with an object of 1000 Euro payment

// and a thank you message

setTimeout(() => {

resolve({

amount: 1000,

note: 'Thank You',

})

}, 100000)

// reject with 0 Euro and an unstatisfatory note

reject({

amount: 0,

note: 'Sorry Lawn was not properly Mowed',

})

})

In the code above, we declared both the company’s promises and our promises. The company promise is either resolved after 100,000ms or rejected. A Promise is always in one of three states: resolved if there is no error, rejected if an error is encountered, or pending if the promise has been neither rejected nor fulfilled. In our case, it falls within the 100000ms period.

But how can we execute the task in a sequential and synchronous manner? That’s where the then keyword comes in. Without it, the functions simply run in the order in which they resolve.

[Over 200k developers use LogRocket to create better digital experiences](https://lp.logrocket.com/blg/learn-more)

**[[](https://lp.logrocket.com/blg/learn-more)Learn more →](https://lp.logrocket.com/blg/learn-more)**

**Sequential execution with .then**

Now we can chain the promises, which allows them to run in sequence with .then. This functions like a normal human language — do this and then that and then that, and so on.

angelMowersPromise

.then(() => myPaymentPromise.then(res => console.log(res)))

.catch(error => console.log(error))

The code above will run the angelMowersPromise. If there is no error, it’ll run the myPaymentPromise. If there is an error in either of the two promises, it’ll be caught in the catch block.

Now let’s look at a more technical example. A common task in frontend programming is to make network requests and respond to the results accordingly.

Below is a request to fetch a list of employees from a remote server.

const api = 'http://dummy.restapiexample.com/api/v1/employees'

fetch(api)

.then(response => response.json())

.then(employees => employees.forEach(employee => console.log(employee.id)) // logs all employee id

.catch(error => console.log(error.message))) // logs any error from the promise

There may be times when you need numerous promises to execute in parallel or in sequence. Constructs such as Promise.all or Promise.race are especially helpful in these scenarios.

Imagine, for example, that you need to fetch a list of 1,000 GitHub users, then make an additional request with the ID to fetch avatars for each of them. You don’t necessarily want to wait for each user in the sequence; you just need all the fetched avatars. We’ll examine this in more detail later when we discuss Promise.all.

Now that you have a fundamental grasp of promises, let’s look at the async/await syntax.

**async/await**

Async/await is a surprisingly easy syntax to work with promises. It provides an easy interface to read and write promises in a way that makes them appear synchronous.

An async/await will always return a Promise. Even if you omit the Promise keyword, the compiler will wrap the function in an immediately resolved Promise. This enables you to treat the return value of an [async function](https://blog.logrocket.com/the-visual-learners-guide-to-async-js-62a0a03d1d57/) as a Promise, which is quite useful when you need to resolve numerous asynchronous functions.

As the name implies, async always goes hand in hand with await. That is, you can only await inside an async function. The async function informs the compiler that this is an asynchronous function.

If we convert the promises from above, the syntax looks like this:

const myAsync = async (): Promise<Record<string, number | string>> => {

await angelMowersPromise

const response = await myPaymentPromise

return response

}

As you can see immediately, this looks more readable and appears synchronous. We told the compiler on line 3 to await the execution of angelMowersPromise before doing anything else. Then, we return the response from the myPaymentPromise.

You may have noticed that we omitted error handling. We could do this with the catch block after the .then in a promise. But what happens if we encounter an error? That leads us to try/catch.

**Error handling with try/catch**

We’ll refer to the employee fetching example to the error handling in action, since it is likely to encounter an error over a network request.

Let’s say, for instance, that the server is down, or perhaps we sent a malformed request. We need to pause execution to prevent our program from crashing. The syntax will look like this:

interface Employee {

id: number

employee\_name: string

employee\_salary: number

employee\_age: number

profile\_image: string

}

const fetchEmployees = async (): Promise<Array<Employee> | string> => {

const api = 'http://dummy.restapiexample.com/api/v1/employees'

try {

const response = await fetch(api)

const { data } = await response.json()

return data

} catch (error) {

if (error) {

return error.message

}

}

}

We initiated the function as an async function. We expect the return value to be of the typeof array of employees or a string of error messages. Therefore, the type of Promise is Promise<Array<Employee> | string>.

Inside the try block are the expressions we expect the function to run if there are no errors. The catch block captures any error that arises. In that case, we’d just return the message property of the error object.

The beauty of this is that any error that first occurs within the try block is thrown and caught in the catch block. An uncaught exception can lead to hard-to-debug code or even break the entire program.