Careful Examination of JavaScript Await

Nested Software Apr 6 *Updated on May 09, 2018*

#javascript **#aync** **#await** **#generators**

Recently I found myself getting a bit confused writing some JavaScript code with async/await. I worked through in some detail what happens when we `await`, and I thought it might be helpful to publish an article about it (for my future self as much as other readers!).

> tl;dr: `await` is similar to `yield` but there are some differences.

The following code will hopefully clarify what happens with `async/await` in JavaScript. Can you figure out what it will do?

```
    console.log('asyncTask started')

    const promise = new Promise(resolve => {
        setTimeout(() => {
            console.log('asyncTask resolving promise')
            resolve('1000')
        }, 2000)
    })

    console.log('asyncTask returning promise')

    return promise
}

const asyncFunction = async () => {
    console.log('asyncFunction started')

    const promise = asyncTask()

    const awaitResult = await promise

    console.log('returning from asyncFunction, awaitResult = "'
        + awaitResult + '"')

    return 'I am returning with "' + awaitResult + '"'
}

const timer = () => setInterval(()=>console.log('tick'), 500)

//start of main

const t = timer()

const mainPromise = asyncFunction()

console.log('mainPromise =  ' + mainPromise)

mainPromise.then((result) => {
    console.log('mainPromise has resolved, result = ' + result)

    //stop timer
    clearInterval(t)
```

```
console.log('end of main code')
```

Here is the output:

```
C:\dev>node promises.js
asyncFunction started
asyncTask started
asyncTask returning promise
mainPromise =  [object Promise]
end of main code
tick
tick
tick
asyncTask resolving promise
returning from asyncFunction, awaitResult = "1000"
mainPromise has resolved, result = I am returning with "1000"
```

JavaScript does some tricky things behind the scenes with `await` so I think it may be helpful to carefully go over this code in order to see what happens at each step:

- In the main code, we start a timer.
- Next, we call `asyncFunction`.
- In `asyncFunction`, we call `asyncTask`.
- `asyncTask` creates a promise.
- The promise initiates a `setTimeout`.
- `asyncTask` returns the promise to `asyncFunction`.
- In `asyncFunction`, we now `await` the promise returned from `asyncTask`.

generator function. What happens here is that `asyncFunction` is temporarily suspended and "returns" early back to the "main" code. If `asyncFunction` were a generator function, then we could resume it in our own code by calling its `next` method. However, we will see that is not quite what happens in this case.

- What is yielded when `asyncFunction` is suspended? It turns out that the JavaScript runtime creates a new promise at this point and that's what is assigned to the `mainPromise` variable. It's important to realize this promise is different from the one that `asyncTask` returns.
- Now the rest of the "main" code runs and we see "end of main code" printed to the console. However, the JavaScript runtime doesn't exit because it still has work to do! After all, we still have a `setTimeout` pending (as well as our timer's `setInterval`) .
- Once two seconds have gone by (we can see this happening via our timer's "ticks"), `setTimeout` 's callback function is invoked.
- This callback function in turn resolves the promise that is currently being awaited by `asyncFunction` .
- When the promise is resolved, the JavaScript runtime resumes `asyncFunction` from where it was suspended by `await` . This is very similar to calling `next` on a generator function, but here the runtime does it for us.
- Since there are no more `await` statements, `asyncFunction` now runs to completion and actually properly returns.

was already suspended earlier, and at that point, it yielded a promise that was assigned to the `mainPromise` variable.

- What happens is that the JavaScript engine intercepts the return and uses whatever value is in the return statement to fulfill the promise it created earlier.
    - We can see that this happens, because now the callback supplied to `mainPromise.then` is actually executed.
    - We returned a string from `asyncFunction` that included the value of the resolved promise from asyncTask: Therefore that's the string that is passed as `result` to the callback in `mainPromise.then((result) => { console.log('mainPromise has resolved, result = ' + result) })`

Since this stuff can easily get confusing, let's summarize:

- `await` in an `async` function is very similar to `yield` in a generator function: In both cases the function is suspended and execution returns to the point from which it was called.
- However, `await` is different in the following ways:
    - The JavaScript runtime will create a new promise and that's what is yielded when the function is suspended.
    - When the promise that is being `await` ed is fulfilled, the JavaScript runtime will automatically resume the `async` function

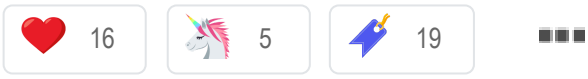
JavaScript runtime will use the function's return value to fulfill the promise that the runtime created earlier.

References:

Aync function
Await
Generator function
Iterators and generators

Related:

- Lazy Evaluation in JavaScript with Generators, Map, Filter, and Reduce
- How to Serialize Concurrent Operations in JavaScript: Callbacks, Promises, and Async/Await
- The Iterators Are Coming! [Symbol.iterator] and [Symbol.asyncIterator] in JavaScript
- Asynchronous Generators and Pipelines in JavaScript



Add to the discussion

PREVIEW SUBMIT

Agus Arias

Apr 8

Thank you! I found this really useful :)

2

REPLY

Nested Software

Apr 8

Thank you!

1

REPLY

code of conduct - report abuse

Classic DEV Post from Mar 11

If you could start over from scratch, how would CSS work?

Ben Halpern

CSS has a lot of issues. Now that we have a few decades of knowledge, how would...

74 53

READ POST SAVE FOR LATER

From one of our Community Sponsors

with Pusher

Jess Lee

The theme of this contest is to create a real-time app or a hack that uses Pusher Channels real-time API.

281 77

READ POST SAVE FOR LATER

Another Post You Might Like

How Did You Start Coding?

Sai gowtham

How Did You Start Coding?

67 34

READ POST SAVE FOR LATER

Javascript needs competition on the front end. Thoughts?
Sasa Blagojevic - Jun 2

Play with the React Router Part-2
Sai gowtham - Jun 2

Passing data to a router-link in Vue.JS
Ali GOREN - Jun 2

Building a Pomodoro Timer with Vue.js on CodePen
Tori Pugh - May 31