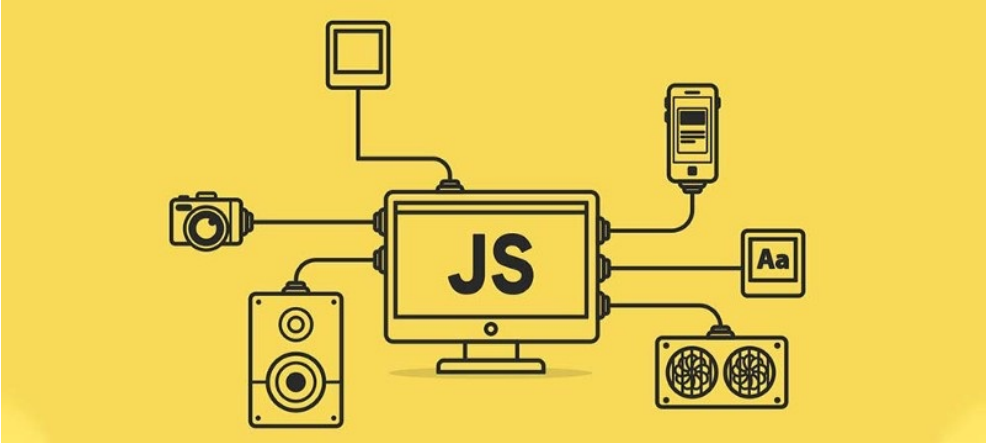
JavaScript with Dimos: Intro to Async Functions

Hey there I am Dimos, a senior software engineer and software architect at arconsis. From time to time, I publish articles into the series "JavaScript with Dimos", where I am trying to analyze a JavaScript topic and to focus on the core concepts of JavaScript.



JavaScript is a **single threaded** language and executes code blocks by order from top to bottom, therefore, it is **synchronous** in nature. Sometimes we may need to use an asynchronous function for example to fetch data from an API. An operation like this takes time to complete, and a blocking approach would negatively impact user experience. Thanks to **JavaScript event loop**, we are able to do **non-blocking** processes by using a callback function and/or a promise function. Using a non-blocking approach we can reduce the JavaScript main thread load and improve our code performance.
JavaScript introduced **async** and **await** keywords as an addition to EcmaScript 2015. In this article, we are going to talk about async functions, and how they work.

Intro to JavaScript Promises

> A Promise is a proxy for a value not necessarily known when the promise is created. It allows us to associate handlers with an asynchronous action's eventual success value or failure reason.

Therefore, a promise is a special JavaScript **object** which represents a **future result** of an asynchronous action.

A promise has three states:

1. **pending**—Initial state which indicates that the request is pending, and is neither fulfilled nor rejected.
2. **fulfilled**—Indicates that the operation was completed successfully.
3. **rejected** —Indicates that the operation failed.

Intro to async functions

The async function is a function that is declared using the async keyword and returns a AsyncFunction object. When we call an async function, it always returns a promise. Hence, async functions allow us to write **promise-based code** (async/await is built on top of promises) **as if it were synchronous**, but **without blocking the main thread**. Below we can find some ways to **declare** async functions:

```
async function functionName(params) {
   # code block
}

const functionName = async function functionName(params) {
   # code block
}

const functionName = async (params) => {
   # code block
};
```

As we can see, we can declare async functions as every other function. The only difference is the **usage** of the async keyword.

Below we can find out how we can **call** an async function:

```
try {
   const result = await functionName(params)
} catch (error) {
   console.error(`Async function threw an error: ${error}`)
}
```

As we can see above, the execution of an async function is equivalent to execution of a regular synchronous JavaScript function and the only difference is the **usage** of await keyword.

How async functions work

The await keyword tells JavaScript to "**wait**" until the promise is settled before executing the rest of the code block. So they are simply *syntactic sugar* for creating functions that return and wait for Promises.
The async/await syntax improves readability by letting us write code as synchronously.

- Async function is declared by using the word **async** in their declaration.
- await keyword is permitted within async functions and can only be used **inside** an async function, otherwise we will get Error.
- Always **returns Promises**.
- Code can be "**paused**" waiting using await keyword.
- await returns whatever the async function returns when it is done. It will return **result** if the Promise is fulfilled or will throw an **error** if the Promise is rejected.
- If an async function throws an exception, the exception will bubble up to the parent functions just like in normal JavaScript, and can be caught with try/catch or using .catch() method directly in await asyncFunc.catch(error => { }) just like in Promises.

Fulfilling the Promise of an async function:

```
async function asyncFunc() {
   return 'result';
}
```

which is *equivalent to*:

```
function asyncFunc() {
  return Promise.resolve('result');
}
```

which is same to:

```
function asyncFunc() {
  return new Promise((resolve, reject) => {
    return resolve('result');
  });
}
```

Rejecting the Promise of an async function:

```
async function asyncFunc() {
  return Promise.reject(new Error('We have error!'));
}
```

which is equivalent to:

```
async function asyncFunc() {
  return Promise.reject(new Error('We have error!'));
}
```

which is same to:

```
async function asyncFunc() {
  return new Promise((resolve, reject) => {
    return reject(new Error('We have error!'));
  });
}
```

The await keyword is used to handle Promises inside the async function. Therefore, handling **results and errors** of asynchronous computations takes place via await. The operator await "**waits"** for the Promise to be settled. It "pauses" the function until *asyncFunc* is done (fulfilled or rejected), and returns the result (resolved result or error):

- If the Promise is **fulfilled**, the result of await is the *fulfillment value*.

```
async function asyncFunc() {
  const result = await otherAsyncFunc();
  console.log(result);
}

// which is equivalent to:
function asyncFunc() {
  return otherAsyncFunc()
  .then(result => {
    console.log(result);
  });
}
```

- If the Promise is **rejected**, await throws the *rejection value*.

```
async function asyncFunc() {
  try {
    await otherAsyncFunc();
  } catch (err) {
    console.error(err);
  }
}

// which is equivalent to:
function asyncFunc() {
```

```
  return otherAsyncFunc()
    .catch(err => {
      console.error(err);
    });
}
```

Error Handling

An async function has two possible return values: *resolved* value, and *rejected* value. We can use **.then()** for normal cases and **.catch()** for exceptional cases. Let's explore how we handle errors in async functions:

1) try...catch statement

The most standard way is to use try...catch statement. When await a call, any rejected value will be thrown as an exception. Here is an example:

```
async function asyncFunc() {
  try {
    await otherAsyncFunc();
  } catch (err) {
    console.error(err);
  }
}
```

The error is exactly the rejected value. After we caught the exception, we have **several ways to deal** in catch(err) {...}block with it:

- Handle the exception, and return a **normal value** (not an error). Not using any return statement in the catch block is equivalent to using return undefined; and is a normal value as well. catch(err) { return 'fail' }
- **Throw error**, if you want the caller to handle the exception. catch (err) { throw err }
- **Reject it**, like return Promise.reject(error). This is equivalent to throwing an error. catch (err) { return Promise.reject(err) }

2) Using .catch

As we declared before, the functionality of await is to "wait" for a promise to be fulfilled or rejected. That's why we can handle the await otherAsyncFunc as a Promise using then or catch method (as an async function **always returns a Promise**). Therefore we can write error handling like this:

```
function asyncFunc() {
  const result = await otherAsyncFunc()
    .catch((error) => { console.log(error); });
}
```

Serial execution of asynchronous functions

The below code block runs two asynchronous functions, asyncFunc1() and asyncFunc2(in sequence. The code execution will wait until asyncFunc1() returns (resolved / rejected), via await asyncFunc1() before starting asyncFunc2() execution. Notice that asyncFunc2() does not depend on the result of asyncFunc1() as using of await runs these two functions **sequentially**.

```
async function asyncFunc1() {
  return 1;
}

async function asyncFunc2() {
  return 2;
}

(async() => {
   const result1 = await asyncFunc1();
   const result2 = await asyncFunc2();
})();
```

Below we can see how we can iterate over async iterable arrays sequentially. As we can see there are 2 ways:

- Using for await...of statement, which creates a loop *iterating* over async iterable objects as well as on sync iterables.
- Using Array.prototype.reduce() method, which executes a *reducer* function for array elements and returns a single value.

```
const arr = [1, 2, 3, 4, 5];

async function asyncFunc(item) {
   return item;
}

(async() => {
   const results = await arr.reduce(async (promise, item) => {
     const accumulator = await promise;
     const result = await asyncFunc(item)
     return [...accumulator, result]
   }, Promise.resolve([]));
   console.log('results', results); // Output: [1, 2, 3, 4, 5]
})();

// which is equivalent to:
(async() => {
   const results = []
   for await (const item of arr) {
     results.push(await asyncFunc(item));
   }
   console.log('results', results); // Output: [1, 2, 3, 4, 5]
})();
```

Concurrent execution of asynchronous functions

As we can see below, using Promise.all() method we can execute these async functions (asyncFunc1(), asyncFunc2()) **concurrently**. This method combines all the promises and returns a *single promise*, resolved to an array of the results of the input promises. If one promise throws an error, then Promise.all() will be **rejected immediately** with the first rejection message / error.

```
async function asyncFunc1() {
  return 1;
}

async function asyncFunc2() {
  return 2;
}

(async() => {
   const results = await Promise.all([asyncFunc1(), asyncFunc2()]);
   console.log(results); // Output: [1, 2]
})();
```

Below we can see how we can iterate over *async iterable arrays* **concurrently**. We can combine Array.prototype.map() built-in array's method which uses a callback to apply the function over each array's and returns an array of values, with Promise.all() method. The Array.prototype.map() method is a synchronous operation, but combined with async/await leads to an array of promises. The final resolved value is an array of the results of the input promises, and because of using Promise.all() the execution will be rejected immediately, if a promise throws an error.

```
const arr = [1, 2, 3, 4, 5];

async function asyncFunc(item) {
  return item;
}

(async() => {
  const results = await Promise.all(arr.map(async (item) => {
    const res = await asyncFunc(item);
    return res;
  }));
  console.log('results', results); // Output: [1, 2, 3, 4, 5]
})();
```

Advantages of async/await

- **Readability:** As we can see from above examples, async/await makes the code cleaner and more readable compared to promises style.
- **Error handling:** We can handle both asynchronous and synchronous errors using try/catch.

Avoiding common pitfalls

- Avoid abuse await, when we can execute async calls in **parallel** unless they need to be done synchronously.
- Avoid combining await and return statements together, e.g. await return Promise.resolve("result") . Instead we can just return a promise via:return Promise.resolve("result") and "await" the promise on execution.

Conclusions

- An **async function always returns Promises**, whether you **use** await **or not**. They are simply *syntactic sugar* for creating functions that return and wait for Promises.
- If **no** await is present the execution is **not "paused"** and the result of it settling is never handled, as the promise starts its execution when we invoke it. Therefore it synchronously returns a pending promise as didn't tell JavaScript to "**wait**".
- The async keyword just declares that the async function returns a value which is guaranteed **to be a promise**, so that callers can call asyncFunc().then(...) or await asyncFunc() safely. Therefore when we use async keyword we create an **asynchronous functions that always return a Promise**.
- We can handle an async function when we call it, exactly **like Promise**.
- await operator "**pauses**" an async function.
- await returns whatever the async function returns when it is done. It will return result if Promise is **fulfilled** or will throw an error if Promise is **rejected.**
- Best approach to **handle errors** of an async function is using try...catch statement, but of course we can use .catch() method directly in async function when we call it.

Those are all we need to know to start using an async function in JavaScript. Meanwhile, please feel free to post any questions, comments, or feedback in the comment section below.

Follow on Twitter here!

Follow on Github here!

Further Reading

1. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
2. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await
3. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
4. https://javascript.info/async
5. https://exploringjs.com/impatient-js/ch_async-functions.html
6. https://www.tabnine.com/academy/javascript/how-to-use-asynchronous-functions/

By Dimosthenis Botsaris on May 2, 2022.

Canonical link

Exported from Medium on May 12, 2022.