Javascript book

**async function**

The **async function** declaration declares an async function where the await keyword is permitted within the function body. The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

Async functions may also be defined [as expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/async_function).

[**Try it**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#try_it)

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#syntax)

async function name(param0) {

statements

}

async function name(param0, param1) {

statements

}

async function name(param0, param1, /\* … ,\*/ paramN) {

statements

}

Copy to Clipboard

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#parameters)

name

The function's name.

param Optional

The name of an argument to be passed to the function.

statements Optional

The statements comprising the body of the function. The await mechanism may be used.

[**Return value**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#return_value)

A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) which will be resolved with the value returned by the async function, or rejected with an exception thrown from, or uncaught within, the async function.

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#description)

Async functions can contain zero or more [await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await) expressions. Await expressions make promise-returning functions behave as though they're synchronous by suspending execution until the returned promise is fulfilled or rejected. The resolved value of the promise is treated as the return value of the await expression. Use of async and await enables the use of ordinary try / catch blocks around asynchronous code.

**Note:** The await keyword is only valid inside async functions within regular JavaScript code. If you use it outside of an async function's body, you will get a [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError).

await can be used on its own with [JavaScript modules.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules)

**Note:** The purpose of async/await is to simplify the syntax necessary to consume promise-based APIs. The behavior of async/await is similar to combining [generators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators) and promises.

Async functions always return a promise. If the return value of an async function is not explicitly a promise, it will be implicitly wrapped in a promise.

For example, consider the following code:

async function foo() {

return 1;

}

Copy to Clipboard

It is similar to:

function foo() {

return Promise.resolve(1);

}

Copy to Clipboard

**Note:**

Even though the return value of an async function behaves as if it's wrapped in a Promise.resolve, they are not equivalent.

An async function will return a different *reference*, whereas Promise.resolve returns the same reference if the given value is a promise.

It can be a problem when you want to check the equality of a promise and a return value of an async function.

const p = new Promise((res, rej) => {

res(1);

});

async function asyncReturn() {

return p;

}

function basicReturn() {

return Promise.resolve(p);

}

console.log(p === basicReturn()); // true

console.log(p === asyncReturn()); // false

Copy to Clipboard

The body of an async function can be thought of as being split by zero or more await expressions. Top-level code, up to and including the first await expression (if there is one), is run synchronously. In this way, an async function without an await expression will run synchronously. If there is an await expression inside the function body, however, the async function will always complete asynchronously.

For example:

async function foo() {

await 1;

}

Copy to Clipboard

It is also equivalent to:

function foo() {

return Promise.resolve(1).then(() => undefined);

}

Copy to Clipboard

Code after each await expression can be thought of as existing in a .then callback. In this way a promise chain is progressively constructed with each reentrant step through the function. The return value forms the final link in the chain.

In the following example, we successively await two promises. Progress moves through function foo in three stages.

1. The first line of the body of function foo is executed synchronously, with the await expression configured with the pending promise. Progress through foo is then suspended and control is yielded back to the function that called foo.
2. Some time later, when the first promise has either been fulfilled or rejected, control moves back into foo. The result of the first promise fulfillment (if it was not rejected) is returned from the await expression. Here 1 is assigned to result1. Progress continues, and the second await expression is evaluated. Again, progress through foo is suspended and control is yielded.
3. Some time later, when the second promise has either been fulfilled or rejected, control re-enters foo. The result of the second promise resolution is returned from the second await expression. Here 2 is assigned to result2. Control moves to the return expression (if any). The default return value of undefined is returned as the resolution value of the current promise.

async function foo() {

const result1 = await new Promise((resolve) =>

setTimeout(() => resolve("1"))

);

const result2 = await new Promise((resolve) =>

setTimeout(() => resolve("2"))

);

}

foo();

Copy to Clipboard

Note how the promise chain is not built-up in one go. Instead, the promise chain is constructed in stages as control is successively yielded from and returned to the async function. As a result, we must be mindful of error handling behavior when dealing with concurrent asynchronous operations.

For example, in the following code an unhandled promise rejection error will be thrown, even if a .catch handler has been configured further along the promise chain. This is because p2 will not be "wired into" the promise chain until control returns from p1.

async function foo() {

const p1 = new Promise((resolve) => setTimeout(() => resolve("1"), 1000));

const p2 = new Promise((\_, reject) => setTimeout(() => reject("2"), 500));

const results = [await p1, await p2]; // Do not do this! Use Promise.all or Promise.allSettled instead.

}

foo().catch(() => {}); // Attempt to swallow all errors...

Copy to Clipboard

async function declarations are [hoisted](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting) to the top of their scope and can be called anywhere in their scope.

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#examples)

[**Async functions and execution order**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#async_functions_and_execution_order)

function resolveAfter2Seconds() {

console.log("starting slow promise");

return new Promise((resolve) => {

setTimeout(() => {

resolve("slow");

console.log("slow promise is done");

}, 2000);

});

}

function resolveAfter1Second() {

console.log("starting fast promise");

return new Promise((resolve) => {

setTimeout(() => {

resolve("fast");

console.log("fast promise is done");

}, 1000);

});

}

async function sequentialStart() {

console.log("==SEQUENTIAL START==");

// 1. Execution gets here almost instantly

const slow = await resolveAfter2Seconds();

console.log(slow); // 2. this runs 2 seconds after 1.

const fast = await resolveAfter1Second();

console.log(fast); // 3. this runs 3 seconds after 1.

}

async function concurrentStart() {

console.log("==CONCURRENT START with await==");

const slow = resolveAfter2Seconds(); // starts timer immediately

const fast = resolveAfter1Second(); // starts timer immediately

// 1. Execution gets here almost instantly

console.log(await slow); // 2. this runs 2 seconds after 1.

console.log(await fast); // 3. this runs 2 seconds after 1., immediately after 2., since fast is already resolved

}

function concurrentPromise() {

console.log("==CONCURRENT START with Promise.all==");

return Promise.all([resolveAfter2Seconds(), resolveAfter1Second()]).then(

(messages) => {

console.log(messages[0]); // slow

console.log(messages[1]); // fast

}

);

}

async function parallel() {

console.log("==PARALLEL with await Promise.all==");

// Start 2 "jobs" in parallel and wait for both of them to complete

await Promise.all([

(async () => console.log(await resolveAfter2Seconds()))(),

(async () => console.log(await resolveAfter1Second()))(),

]);

}

sequentialStart(); // after 2 seconds, logs "slow", then after 1 more second, "fast"

// wait above to finish

setTimeout(concurrentStart, 4000); // after 2 seconds, logs "slow" and then "fast"

// wait again

setTimeout(concurrentPromise, 7000); // same as concurrentStart

// wait again

setTimeout(parallel, 10000); // truly parallel: after 1 second, logs "fast", then after 1 more second, "slow"

Copy to Clipboard

**await and parallelism**

In sequentialStart, execution suspends 2 seconds for the first await, and then another second for the second await. The second timer is not created until the first has already fired, so the code finishes after 3 seconds.

In concurrentStart, both timers are created and then awaited. The timers run concurrently, which means the code finishes in 2 rather than 3 seconds, i.e. the slowest timer. However, the await calls still run in series, which means the second await will wait for the first one to finish. In this case, the result of the fastest timer is processed after the slowest.

If you wish to safely perform two or more jobs in parallel, you must await a call to [Promise.all](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all), or [Promise.allSettled](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled).

**Warning:** The functions concurrentStart and concurrentPromise are not functionally equivalent.

In concurrentStart, if promise fast rejects before promise slow is fulfilled, then an unhandled promise rejection error will be raised, regardless of whether the caller has configured a catch clause.

In concurrentPromise, Promise.all wires up the promise chain in one go, meaning that the operation will fail-fast regardless of the order of rejection of the promises, and the error will always occur within the configured promise chain, enabling it to be caught in the normal way.

[**Rewriting a Promise chain with an async function**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#rewriting_a_promise_chain_with_an_async_function)

An API that returns a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) will result in a promise chain, and it splits the function into many parts. Consider the following code:

function getProcessedData(url) {

return downloadData(url) // returns a promise

.catch((e) => downloadFallbackData(url)) // returns a promise

.then((v) => processDataInWorker(v)); // returns a promise

}

Copy to Clipboard

it can be rewritten with a single async function as follows:

async function getProcessedData(url) {

let v;

try {

v = await downloadData(url);

} catch (e) {

v = await downloadFallbackData(url);

}

return processDataInWorker(v);

}

Copy to Clipboard

Alternatively, you can chain the promise with catch():

async function getProcessedData(url) {

const v = await downloadData(url).catch((e) => downloadFallbackData(url));

return processDataInWorker(v);

}

Copy to Clipboard

In the two rewritten versions, notice there is no await statement after the return keyword, although that would be valid too: The return value of an async function is implicitly wrapped in [Promise.resolve](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve) - if it's not already a promise itself (as in the examples).