HOW SOFTWARE WORKS · BUILDER · 15 MIN READ

# Async JavaScript: promises, async and await

Why some code has to wait, what a promise is, how async and await read, how to catch errors and run things side by side, and the async bugs AI-written code makes most.

Pass the check (4 of 5) online to earn the “Promise Keeper” badge:
https://digitalbuilders757.com/university/study/async-javascript-promises-and-await

## Why some code has to wait

A page's JavaScript does one thing at a time. Loading data from an API, asking a database or waiting for a timer can take a second or more, and if the code simply stopped and waited, the whole page would freeze: no scrolling, no clicks.

So slow work happens in the background, and your code gets a promise straight away: an object standing in for a value that isn't ready yet. A promise starts pending, then is either fulfilled with a value (the data arrived) or rejected with an error (the network failed).

## async and await

await waits for a promise and hands you its value: const response = await fetch("/api/notes"). Only the function it's in pauses; the rest of the page keeps working. You can use await inside a function marked async, and at the top level of a module.

An async function always returns a promise, so whoever calls it needs await too: const notes = await getNotes(). Forget it and you get the promise itself, which shows up as Promise { <pending> } or [object Promise] where your data should be.

Older code chains .then() instead: fetch(url).then((response) => response.json()).then((data) => show(data)). It's the same promises underneath, and AI tools write both styles.

## When things go wrong

- A rejected promise throws an error at the await, so wrap the awaits in try { ... } catch (error) { ... } and show the user a friendly message.
- fetch only rejects when the request can't happen at all, such as when you're offline. A 404 or a 500 still arrives as a response, with response.ok set to false, so check it before you read the body.
- finally runs whether it worked or not: the place to hide a loading spinner.

- An error nobody catches becomes an 'Uncaught (in promise)' error in the console, and often a blank or frozen screen for the user.

## One after another, or side by side

Two awaits in a row run one after the other. That's right when the second needs the first's result, like loading a user and then their orders. When the jobs don't depend on each other, start them together and wait once: const [user, settings] = await Promise.all([getUser(), getSettings()]). Two one-second calls then take about one second, not two.

- Promise.all fails as soon as any promise fails. Promise.allSettled waits for every one and tells you which worked.
- for (const id of ids) { await save(id) } handles one item at a time, in order.
- ids.forEach(async (id) => { await save(id) }) does not wait at all: forEach starts every call and moves straight on. Use for...of, or await Promise.all(ids.map(save)).
- Hundreds of requests at once can hit rate limits. For big lists, work in small batches.

## Async bugs to spot in AI-written code

- A missing await: data shows up as [object Promise], or .json is not a function.
- forEach with an async callback, followed by code that expects the work to be done.
- No response.ok check, so an error page gets parsed as JSON ('Unexpected token <').
- No try/catch around awaits that can fail, so one network blip breaks the page.
- In React, useEffect's function can't be async itself: define an async function inside the effect and call it. In Next.js server components, you can await data directly.
- Independent awaits written one after another, making a page slower than it needs to be.

## Key terms

**Async / await**
Code for slow tasks like loading data: await waits for the answer.

**Promise**
An object standing in for a value that isn't ready yet. It's pending, then fulfilled with a value or rejected with an error.

**Promise.all**
Starts several promises side by side and waits for all of them; it fails as soon as any one fails.

**Callback**
A function you hand to other code to call later, like the one you pass to addEventListener or setTimeout.

**fetch**
The browser's built-in way to request a URL. It returns a promise of a response; check response.ok before using it.

**API**
An agreed way for one piece of software to ask another for something.

**JSON**
A simple text format for data that apps send to each other: names and values in curly braces.

**Error message**
The computer's description of what went wrong. Copy it exactly when asking for help.

## Try it: Watch code wait

1. Open the 'Wait with async and await' starter in the code playground and run it. Remove one await: what changes?
2. Open 'Side by side with Promise.all'. Compare the two times it prints, then add the failing call and catch the error.
3. Open 'The forEach that doesn't wait' and fix it two ways: with for...of, and with Promise.all and map.
4. Find an await in code your AI wrote. What happens if that promise rejects? Is there a try/catch, and is response.ok checked?
5. Ask your AI helper: 'Which of these awaits could run side by side?' Check its answer: do the calls really not depend on each other?

KNOWLEDGE CHECK

# Async JavaScript: promises, async and await

5 questions. Circle one answer for each; 4 or more correct passes. Answers are at the back.

### 1. const data = getNotes() prints Promise { <pending> } instead of the notes. Why?

A. The notes are too big to print, so the console shows a placeholder
B. getNotes has a typo, so JavaScript printed a promise instead of an error
C. Promises only work inside loops, and this call isn't inside one
D. getNotes is async, so it returns a promise; it needs await

### 2. Your API returns a 404. What does await fetch(url) do?

A. It returns a response with response.ok set to false
B. It throws, so the code jumps straight into your catch block
C. It returns undefined, and the next line carries on as normal
D. It waits forever, because 404 pages never finish loading

### 3. getUser() and getWeather() each take a second, and neither needs the other. The fastest way to get both?

A. await getUser(), then await getWeather() on the next line
B. await Promise.all([getUser(), getWeather()])
C. Call both without await, and read the results a second later
D. Put both calls in a forEach, so they run in a loop

### 4. This logs 0 even though every save works. Why? ids.forEach(async (id) => { await save(id); done++ }); console.log(done)

A. done must be declared with const for the count to change
B. save needs to be called without await when it's inside a loop
C. forEach doesn't wait for async callbacks, so the log runs before any save finishes
D. console.log can't print numbers that change inside a loop

### 5. Where's the best place to hide a loading spinner after an await that might fail?

A. Right after the await, as the next line inside try
B. Only inside catch, because the spinner can only stop when there's an error
C. Before the await, so it never shows at all
D. In a finally block, which runs whether it worked or not

## Official sites

MDN: Using promises: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
MDN: Promise.all(): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

Last reviewed 2026-11-25. AI tools change fast; check each tool's own site for current features, plans and prices.

ANSWER KEY

# Async JavaScript: promises, async and await

**1. D: getNotes is async, so it returns a promise; it needs await**
An async function always returns a promise. const data = await getNotes() waits for it and gives you the notes.

**2. A: It returns a response with response.ok set to false**
fetch only rejects when the request can't happen, like when you're offline. Check response.ok before reading the body.

**3. B: await Promise.all([getUser(), getWeather()])**
Promise.all starts both and waits once: about one second in all. Two awaits in a row take about two.

**4. C: forEach doesn't wait for async callbacks, so the log runs before any save finishes**
forEach starts every callback and moves straight on. Use for (const id of ids) { await save(id) } or await Promise.all(ids.map(save)).

**5. D: In a finally block, which runs whether it worked or not**
finally always runs, so the spinner goes away whether the data loaded or the request failed.