JavaScript Part 8: Asynchronous JavaScript and fetch

Some tasks take time, such as asking a server for data, and a page must not freeze while it waits. Asynchronous code lets JavaScript start such a task and carry on with other work until the answer arrives. Four steps make it manageable: wait, promise, await and fetch. Each step answers one simple question, and the browser console can run every line. In this part, Bassam, a front-end developer at a shipping company, uses them to build a live tracking panel that shows the status of a parcel.

1. Wait

A page runs its JavaScript on a single thread, which means one task at a time, so a slow task could freeze everything. The question is: can other work continue while we wait? With asynchronous code, JavaScript starts a slow task, such as a timer or a request to a server, and moves on. When the slow task finishes, the browser runs the code that was prepared for the result. This is why a page stays responsive while the data is on its way. A common mistake is to write the next line as if the answer had already arrived, when it has not.

Example: A cook puts rice on the stove and sets a timer. Instead of staring at the pot, the cook chops vegetables and lays the table. The rice is the slow task, the timer is the code that was prepared for the result, and the chopping is the rest of the page carrying on. When the timer rings, the cook returns to the pot, and no minute was wasted while waiting.

At work: Bassam's tracking panel asks the tracking service of the company for the status of a parcel, which takes a moment. He asks the Wait question and lets the rest of the page keep working. The visitor can still scroll, open the menu and read other details. Only the status box waits for its answer.

2. Promise

A promise is an object that stands for a result that is not ready yet. It starts as pending, and later it becomes fulfilled with a value or rejected with an error. The question is: has the promise finished, and did it succeed or fail? We attach the then method to receive the value when it succeeds, and the catch method to handle the error when it fails. The finally method runs in both cases, which makes it a good place to remove a loading message. Promises can be chained, so that the result of one step becomes the input of the next.

Example: A restaurant gives a customer a pager after the order is placed. The pager is the promise: it is pending while the meal is being prepared, and the customer is free to sit down and read. When the meal is ready, the pager rings, which is a fulfilled promise. If an item has run out, the staff explain it instead, which is a rejected promise, so the customer always gets one clear answer.

At work: Bassam shows a loading message as soon as the request starts. He asks the Promise question and uses then to display the status when the promise is fulfilled, and catch to show a friendly message when it is rejected. The finally method hides the loading message in both cases. Visitors never stare at an empty box or at a spinner that never stops.

3. Await

The async and await keywords make promise code read like normal step-by-step code. We write async before a function, and inside it we write await in front of a promise to pause that function until the promise is settled. The pause affects only that function, not the whole page. The question is: which step must finish before the next line can run? To handle failures, we place the awaited steps inside a try block and catch

This document is a property of Asaseeyat Academy - © Copy Right Protected.

the error in a catch block. An async function always returns a promise itself, so the code that calls it can wait for it in turn. Independent requests should not wait for each other, so Promise.all waits for several promises that run together until all of them are done.

Example: A baker must wait for the dough to rise before baking, because baking depends on it. While the dough rises, the baker heats the oven and greases the tray, because neither of those depends on the dough. The baker waits only where a step needs the one before it, and starts everything else together. That difference between dependent and independent steps is exactly what await and Promise.all express.

At work: Bassam needs the status of the parcel and the details of the driver, and neither depends on the other. He asks the Await question, uses await for each step inside a try and catch block, and wraps the two requests in Promise.all. The panel is ready after the slower of the two requests instead of after the sum of both. If one request fails, the catch block shows one clear message.

4. Fetch

The fetch function sends a request to a web address and returns a promise for the response. The question is: what did the server send back, and is it OK? The promise rejects only when the request could not be made at all, for example when the network is down, but it does not reject for an error status such as page not found. So we check the ok property of the response first. Then the json method turns the body of the response into a JavaScript object, and since it also returns a promise, it needs await as well. For example, `const response = await fetch(trackingUrl)` waits for the response from the address stored in trackingUrl. Only then do we use the object to fill the page.

Example: A reader opens a news page, and the browser asks a server for the latest headlines. If the server answers with an apology such as page not found, instead of headlines, the page must not show the apology as if it were news. So the page first checks whether the answer is fine, which is what the ok property tells us, and only then reads the headlines and shows them. A page that skips this check can display an error message in the place of the news.

At work: Bassam fetches the tracking address for a parcel. He asks the Fetch question and checks the ok property before anything else, so a missing parcel produces a clear message instead of a broken panel. When the response is fine, he reads it with json and fills the status box. The panel now shows live data and survives a failed request.

Conclusion

Bassam's tracking panel now loads live data without freezing the page and without confusing the visitor. Wait asks whether other work can continue meanwhile, and promise asks whether the result has finished and whether it succeeded. Await asks which step must finish before the next line can run, and fetch asks what the server sent back and whether it is OK. In the next part, we organise code into modules and learn to handle errors and debug.

Sources

- MDN Web Docs: Promise – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- MDN Web Docs: async function – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- MDN Web Docs: await operator – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await
- MDN Web Docs: Using the Fetch API – https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
- MDN Web Docs: Response.json() and Response.ok, checked in Using the Fetch API – https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

This document is a property of Asaseeyat Academy - © Copy Right Protected.