Anton Lavrenov Follow
Making web. https://lavrton.com
Oct 31, 2017 · 3 min read

JavaScript loops—how to handle async/await

How to run async loops in sequence or in parallel?

Before doing asynchronous magic with loops I want to remind you how we write classical synchronous loops.

😐 Synchronous loop

Long time ago I was writing loops like this (probably you too):

```
for(var i = 0; i < array.length; i++) {
  var item = array[i];
  // do something with item
}
```

Old-school loop

It is good, it is fast, but it has many readability and maintenance issues. Then I used to use its better version:

```
array.forEach(item => {
  // do something with item
});
```

Ah. This version of loops is really nice.

JavaScript language is developing very fast. We have more features and new syntax. One of my favorite is async/await. I am using it more frequently now. And sometimes I have a situation where I need to do something with items in an array asynchronously.

😭 Asynchronous loops

How to use await inside a loop? Let's just write async function and await each task.

```
async function processArray(array) {
  array.forEach(item => {
    // define synchronous anonymous function
    // it will throw error here
    await func(item);
  });
}
```

Wrong example. It will throw an error.

This code will **throw a syntax error**. Why? Because we can not use **await** inside synchronous function. As you can see "processArray" is async function. But anonymous function that we use for forEach is **synchronous**.

🤔 1. Don't wait for result

How to fix previous issue? We can define anonymous function as asynchronous as well:

```
async function processArray(array) {
  array.forEach(async (item) => {
    await delayedLog(item);
  });
  console.log('Done!');
}
```

But forEach **will not wait** until all items are finished. It will just run tasks and go next. As a proof let's write simple test:

```
1  function delay() {
2    return new Promise(resolve => setTimeout(resolve, 300));
3  }
4
5  async function delayedLog(item) {
6    // notice that we can await a function that returns promise
7    await delay();
8    // log item only after a delay
9    console.log(item);
10 }
11
12
13 async function processArray(array) {
14   array.forEach(async (item) => {
15     await delayedLog(item);
16   });
17   console.log('Done!');
18 }
19
20 processArray([1, 2, 3]);
```

The output will be:

```
Done!
1
2
3
```

It can be ok if you don't need to wait for the results. But in almost all cases this is not a good logic.

2. Process array in sequence

To wait the result we should return back to old-school "for loop", but this time we can use modern version with *for..of* construction (thanks to Iteration Protocol) for better readability:

```
async function processArray(array) {
  for (const item of array) {
    await delayedLog(item);
  }
  console.log('Done!');
}
```

Run async operation as sequence

This will give us expected output:

```
1
2
3
Done!
```

The code will handle each item one by one in series. But we can run it in parallel!

3. Process array in parallel

We can slightly change the code to run async operations in parallel:

```
async function processArray(array) {
  // map array to promises
  const promises = array.map(delayedLog);
  // wait until all promises are resolved
  await Promise.all(promises)
  console.log('Done!');
}
```

This code will run many "delayLog" tasks in parallel. But be careful with very large array (too many tasks in parallel can be too heavy for CPU or memory).

That is all. Thanks for reading!

. . .

Do you have performance issues with your web-app? I can help you.