



# How to speed up Puppeteer scraping with parallelization

Reduce the total time by running multiple jobs in parallel



 Code is available on GitHub

## Scraping with Puppeteer

A typical scraping job gets a bunch of URLs and it needs to open a page, interact with the site, possibly by logging in and navigating around, and extract a piece of data. Puppeteer is the perfect choice for this, as it uses an actual browser that solves a whole array of edge cases.

The trivial solution is to run a loop that gets the next item, run the browser to interact with the site, and write its result to some collection value. For example, the following code starts the browser, then runs the scraper code sequentially, each job in a new tab:

```
const browser = await puppeteer.launch({/* ... */});

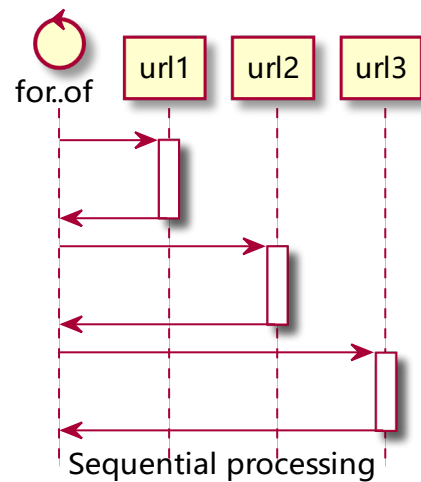
const urls = [/* ... */];
const results = [];
for (const url of urls) {
  const page = await browser.newPage();
  await page.goto(url);

  // run test code
  const result = ...;
  results.push(result);

  await page.close();
}

await browser.close();
```

This processes one job at a time, each waiting for the previous one to finish. This limits the memory and CPU usage during the run, but it leads to terrible performance. But since the jobs are independent it is possible to shorten the total runtime by using multiple tabs.



But first, let's do some refactoring!

## Disposing resources

The above code works for a dummy project but is extremely fragile. If there is an error, it leaves the page and the browser open, which can easily lead to a memory leak. Fortunately, there is an elegant solution for this: the **async disposer pattern**. This structure implements automatic lifecycle management for a resource that handles cleaning it up when it's not needed.

To manage the browser and the page in a safer way, you can use this code:

```
const withBrowser = async (fn) => {
  const browser = await puppeteer.launch({/* ... */});
  try {
    return await fn(browser);
  } finally {
    await browser.close();
  }
}

const withPage = (browser) => async (fn) => {
  const page = await browser.newPage();
  try {
    return await fn(page);
  } finally {
    await page.close();
  }
}

const urls = [/* ... */];
const results = [];

await withBrowser(async (browser) => {
  for (const url of urls) {
    const result = await withPage(browser)(async (page) => {
      await page.goto(url);

      // run test code
      return ...;
    });

    results.push(result);
  }
});
```

With this structure, starting and stopping the browser and the page is handled by the utility functions. This leads to a safer code.



The easiest way in Javascript to run multiple things in parallel is to use `Promise.all`. Using this built-in yields a simple structure to process an array of inputs with async functions.

```
const results = await withBrowser(async (browser) => {
  return Promise.all(urls.map(async (url) => {
    return withPage(browser)(async (page) => {
      await page.goto(url);

      // run test code
      return ...;
    });
  }));
});

// starting url1
// starting url2
// starting url3
// url1 finished
// url3 finished
// url2 finished
```

This also collects the results in an array, so there is no need to push the elements for every new result. And it also yields a nice, functional code with no loops and variables.



<https://advancedweb.hu/how-to-speed-up-puppeteer-scraping-with-parallelization/>

Let’s see how to put a limit on the parallelization!

# Bluebird Promise.map

The **Bluebird** promise implementation has a **map** function that supports a **concurrency setting**. By using that, you can define how many elements are processed in parallel, and it automatically starts a new one when a previous one is finished. This coordination happens in the background, and it returns the results in an array, same as the **Promise.all** construct.

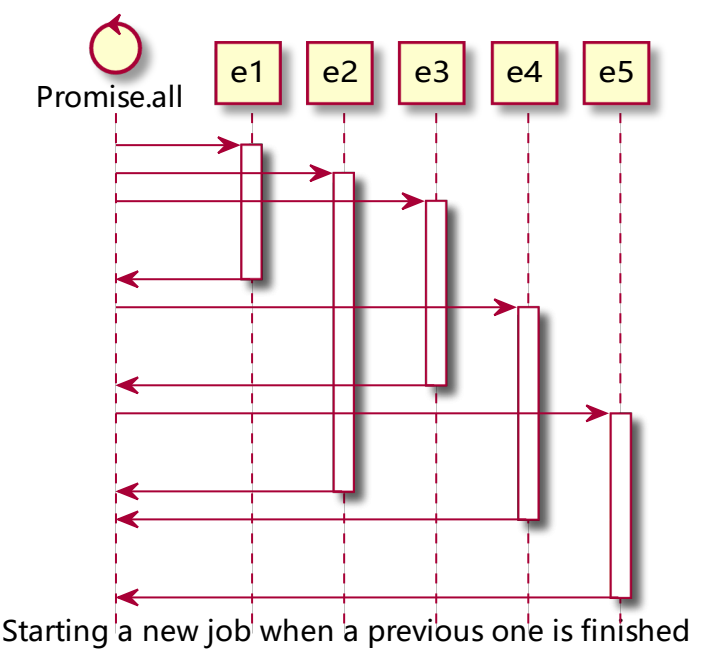
This code processes the urls 3 at a time:

```
const bluebird = require("bluebird");

const results = await withBrowser(async (browser) => {
  return bluebird.map(urls, async (url) => {
    return withPage(browser)(async (page) => {
      await page.goto(url);

      // run test code
      return ...;
    });
  }, {concurrency: 3});
});

// starting url1
// starting url2
// starting url3
// url1 finished
// starting url4
// url3 finished
// starting url5
// url2 finished
// url4 finished
// url5 finished
```



# RxJS mergeMap

Another solution is to use RxJS and Observables to process the input array. This library has a **mergeMap** operator that also supports a concurrency setting. It works the same as Bluebird.map: it processes **n** items in parallel, starting work on a new one when a previous one is finished.

One difference is that RxJS works with *streams of elements*, so to get an array with the results, you need to use the **toArray** operator.

This code processes 3 items at a time:

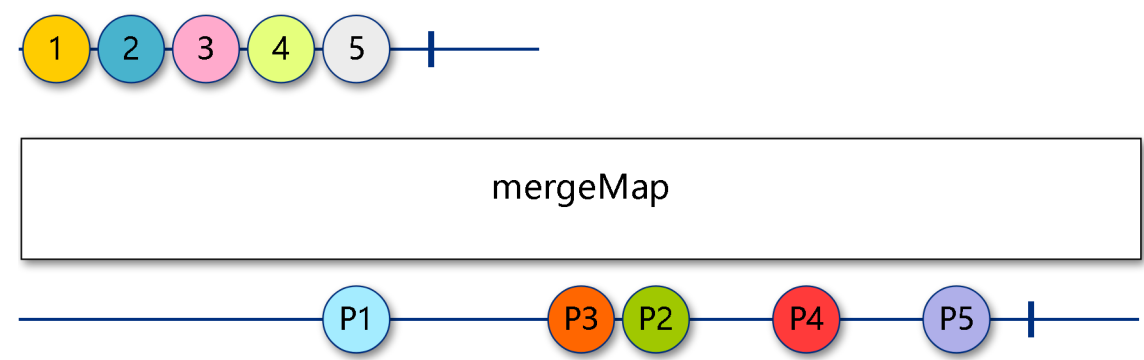
```
const rxjs = require("rxjs");
const {mergeMap, toArray} = require("rxjs/operators");

return rxjs.from(urls).pipe(
  mergeMap(async (url) => {
    return withPage(browser)(async (page) => {
      console.log(`Scraping ${url}`);
      await page.goto(`${host}/${url}`);

      const result = await page.evaluate(e => e.textContent, await page.$("#result"));
      console.log(`Scraping ${url} finished`);
      return result;
    });
  }, 3),
  toArray(),
).toPromise();

// starting url1
// starting url2
// starting url3
// url1 finished
// starting url4
// url3 finished
// starting url5
// url2 finished
// url4 finished
// url5 finished

// [url1, url3, url2, url4, url5]
```



## orderedMergeMap

The problem with the `mergeMap` operator is that it does not keep the ordering of the elements in the result array. This may or may not be a problem, but fortunately, it's not hard to make a variation of it that makes sure that the result array has the same order as the input array. You can see the implementation in [this article](#).

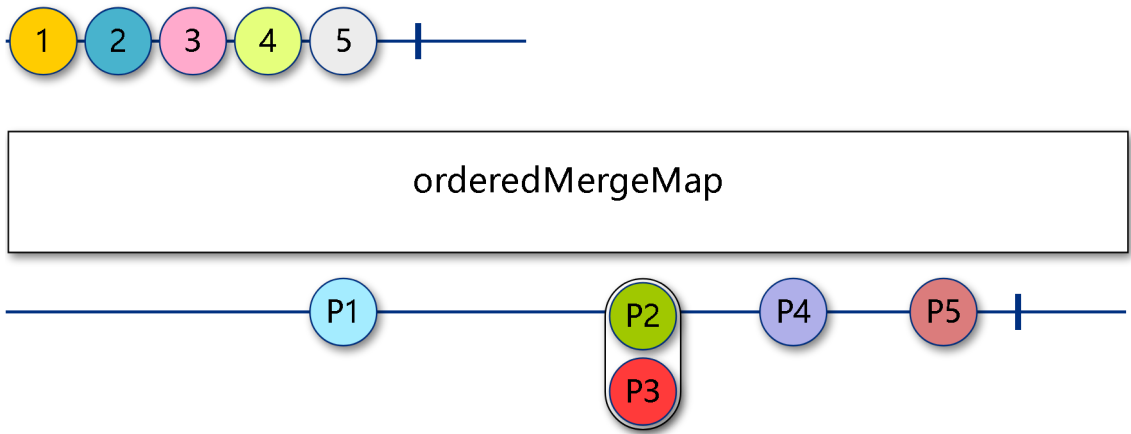
To use it, switch the operator to the ordered implementation:

```
return rxjs.from(urls).pipe(
  orderedMergeMap(async (url) => {
    return withPage(browser)(async (page) => {
      console.log(`Scraping ${url}`);
      await page.goto(`${host}/${url}`);

      const result = await page.evaluate(e => e.textContent, await page.$("#result"));
      console.log(`Scraping ${url} finished`);
      return result;
    });
  }, 3),
  toArray(),
).toPromise();

// starting url1
// starting url2
// starting url3
// url1 finished
// starting url4
// url3 finished
// starting url5
// url2 finished
// url4 finished
// url5 finished

// [url1, url2, url3, url4, url5]
```



# Error handling

The above solutions all produce a result array when all individual scraping jobs run to completion. But even one error case blows up the whole process.

Due to error bubbling, any exception thrown inside a handler terminates the processing and discards all the results. Especially with a long job running through a long list of urls this is a waste and retrying should happen only for the failed elements.

Let's modify the scraper code to return an object which has either a `result` or an `error` field. This allows the caller to decide how to handle errors, either by discarding those results or retrying them.

The easiest way is to handle this on the page-level since each job has its own browser tab:

```
return withPage(browser)(async (page) => {
  await page.goto(url);

  // run test code
  return ...;
}).then((r) => ({result: r}), (e) => ({error: e}));

// [
//   {result: url1},
//   {result: url2},
//   {error: "Not found"},
//   {result: url4}
// ]
```

# Conclusion

Scraping with Puppeteer is essentially an async operation as it needs to communicate with a remote process (the browser). This makes it easy to run jobs in parallel, speeding up the scraping.

But as each job needs a tab in the browser, it consumes a lot of memory. Unbounded parallelization can quickly use up all the available RAM which then either terminates the process or slows it down.

By using a library that allows collection processing with a cap on the available concurrency, you can control the memory usage. This makes it possible to get the benefits of parallelization while also keeping the process within the resources of the executing machine.


In this article, we’ve covered Bluebird `Promise.map` and RxJS’s `mergeMap` to solve this problem as they both have an option to set the parallelization limit.

Javascript<sup>68</sup>

async/await<sup>21</sup>

RxJS<sup>4</sup>





22 January 2021