Extracted from:

Modern Asynchronous JavaScript

Tackle Complex Async Tasks with Less Code

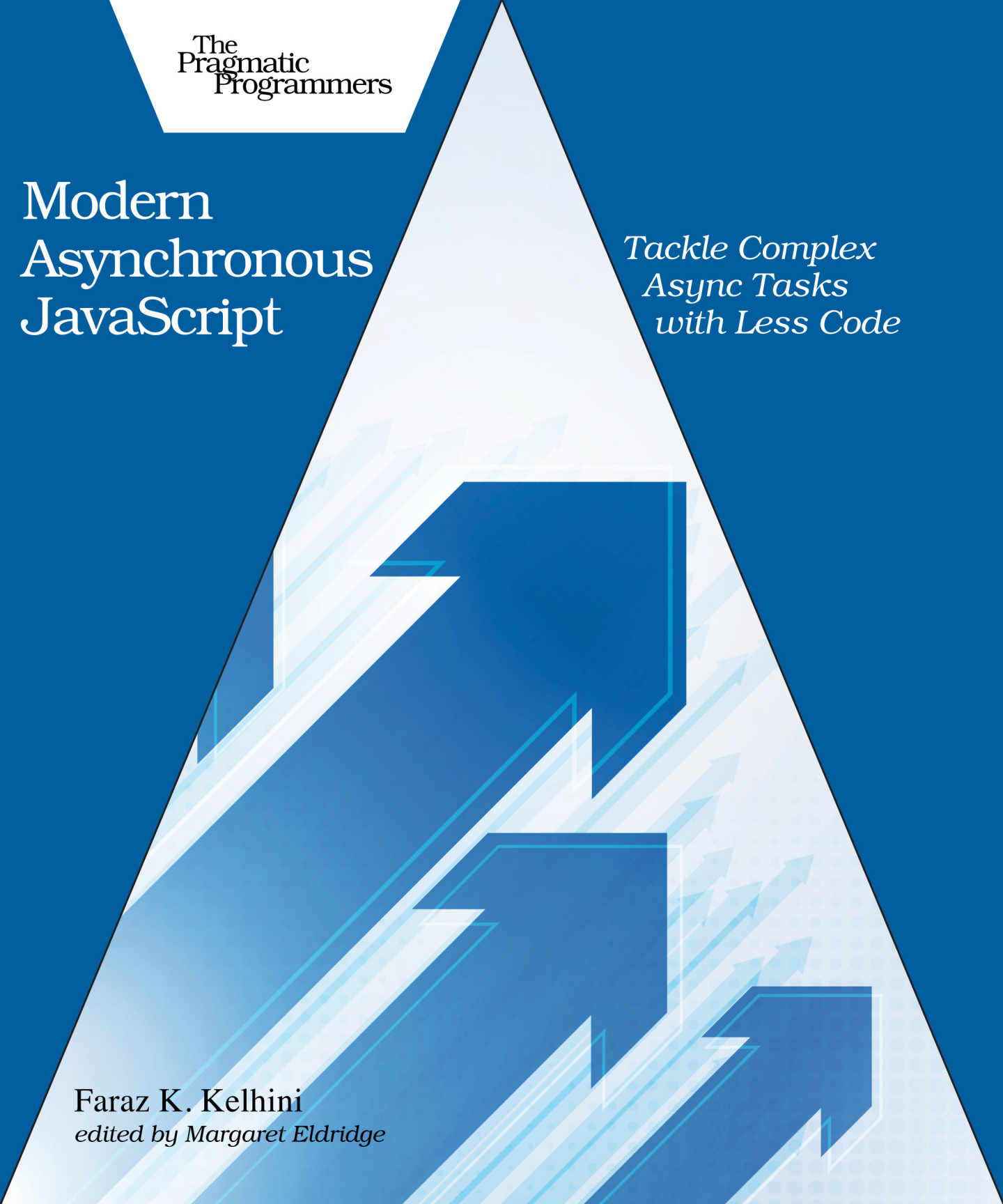
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Modern Asynchronous JavaScript

Tackle Complex Async Tasks with Less Code



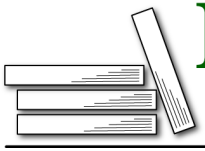
Faraz K. Kelhini
edited by Margaret Eldridge

Modern Asynchronous JavaScript

Tackle Complex Async Tasks with Less Code

Faraz K. Kelhini

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Margaret Eldridge
Copy Editor: L. Sakhi MacMillan
Founders: Andy Hunt and Dave Thomas

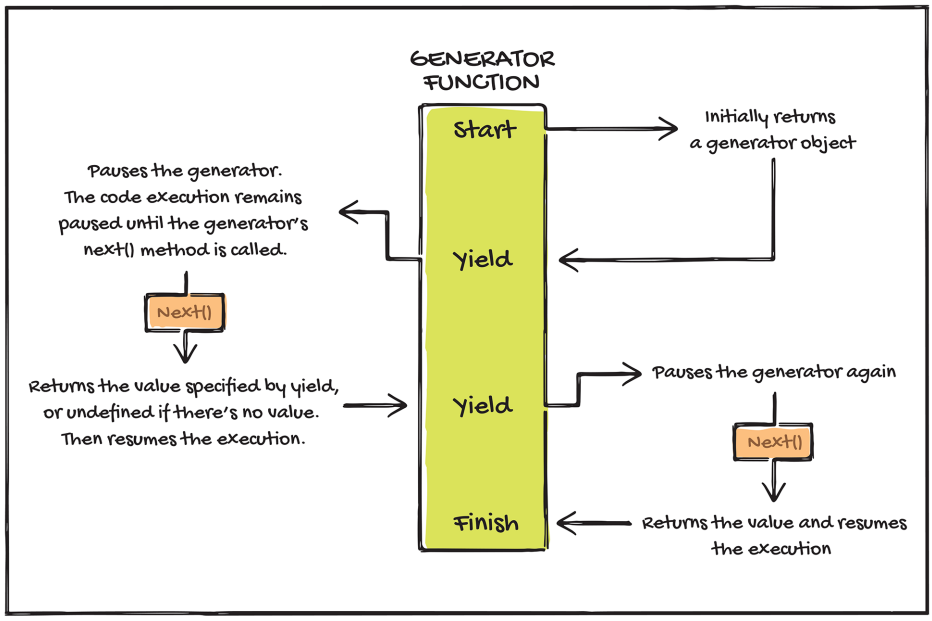
For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-904-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2, 2021

Using a Generator to Define a Custom Iterator

Generator functions enhance the process of defining the iterable protocol by providing an iterative algorithm. When called, a generator function doesn't execute its body immediately. Instead, it returns a special type of iterator known as a *generator object*, as shown in the following image.



We can run the generator function's body by calling its next() method. The yield keyword pauses the generator and specifies the value to be returned. With that in mind, let's adapt the example in Creating a Custom Iterator, on page ?. The result of this code is identical, but it's much easier to implement.

Notice the asterisk following the function keyword at line 5. This is our generator function and defines a custom iterator for collection:

generators/gen_ex01.js

```
Line 1  const collection = {
-         a: 10,
-         b: 20,
-         c: 30,
5         [Symbol.iterator]: function*() {
-           for (let key in this) {
-             yield this[key];
-           }
-         }
10      };
```

```
const iterator = collection[Symbol.iterator]();

console.log(iterator.next());  // ⇒ {value: 10, done: false}
console.log(iterator.next());  // ⇒ {value: 20, done: false}
console.log(iterator.next());  // ⇒ {value: 30, done: false}
console.log(iterator.next());  // ⇒ {value: undefined, done: true}
```

We've used a for...in loop inside the generator to iterate over the object's properties. With each iteration, the yield keyword halts the loop's execution and returns the value of the succeeding property to the caller.

It's possible to call a generator *function* as many times as needed, and each time it returns a new generator object. But a generator *object* can be iterated only once. Since the object returned by a generator is always an iterator, we can use the for...of syntax to iterate over the result as well.

Now that we know how synchronous generators work, we're ready to look at its asynchronous counterpart.

Creating an Asynchronous Generator

An async generator is similar to a sync generator in that calling next() resumes the execution of the generator until reaching the yield keyword. But rather than returning a plain object, next() returns a promise.

You can think of an async generator as a combination of an async function and a generator function. Let's rewrite the example from Retrieving URLs Separately, on page ?, using a generator function. Notice the async keyword and the asterisk symbol (*) at line 7 indicating an asynchronous generator function:

generators/gen_ex02.js

```
Line 1 const srcArr = [
  'https://eloux.com/async_js/examples/1.json',
  'https://eloux.com/async_js/examples/2.json',
  'https://eloux.com/async_js/examples/3.json',
];

srcArr[Symbol.asyncIterator] = async function*() {
  let i = 0;
  for (const url of this) {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error('Unable to retrieve URL: ' + response.status);
    }
    yield response.json();
  }
};
```

```
const iterator = srcArr[Symbol.asyncIterator]();

iterator.next().then(result => {
  console.log(result.value.firstName);  // ⇒ John
});

iterator.next().then(result => {
  console.log(result.value.firstName);  // ⇒ Peter
});

iterator.next().then(result => {
  console.log(result.value.firstName);  // ⇒ Anna
});
```

Within this generator, we've used the await keyword to wait for the fetch operation to complete. As with non-async generator functions, yield returns the result to the function's caller. Notice how this asynchronous generator simplifies the process of defining the asynchronous iterable protocol. It's not only easier to write but also less error-prone.

In production, you'll also want to use catch() to handle errors and rejected cases during the iteration. A well-designed program should be able to recover from common errors without terminating the application. You can chain a catch() method the same way as its sister method then(). For example:

```
iterator.next()
  .then(result => {
    console.log(result.value.firstName);
  })
  .catch(error => {
    console.error('Caught: ' + error.message);
  });
```

If an error occurs, catch() will be executed with the rejection reason passed as its argument. Now let's look at a more complex example of an async generator.

Iterating over Paginated Data

One situation we want to use asynchronous iteration over synchronous is when working with web APIs that provide paginated data. By using an asynchronous iterator, we can seamlessly make multiple network requests and iterate over the results. For example, GitHub provides an API that allows us to retrieve commits for a repository. The response is in JSON format and contains the data for the last 30 commits of the repository. The API will also provide pagination link headers for the remaining commits.

Say we want to retrieve info for the last 90 commits of a particular GitHub repository. We can achieve that using an asynchronous iterator or, better yet, a generator. Let's create an asynchronous generator and program it to handle the pagination:

generators/gen_ex03.js

```
// create an async generator function
async function* generator(repo) {

  // create an infinite loop
  for (;;) {

    // fetch the repo
    const response = await fetch(repo);

    // parse the body text as JSON
    const data = await response.json();

    // yield the info of each commit
    for (let commit of data) {
      yield commit;
    }

    // extract the URL of the next page from the headers
    const link = response.headers.get('Link');
    repo = /<(.*?)>; rel="next"/.exec(link)?. [1];

    // if there's no "next page", break the loop.
    if (repo === undefined) {
      break;
    }
  }
}

async function getCommits(repo) {

  // set a counter
  let i = 0;

  for await (const commit of generator(repo)) {

    // process the commit
    console.log(commit);

    // break at 90 commits
    if (++i === 90) {
      break;
    }
  }
}

getCommits('https://api.github.com/repos/tc39/proposal-temporal/commits');
```

Here, we've created two async functions, one of which is a generator. The generator function is responsible for retrieving the resource, parsing it as JSON, and sending the info of each commit to the generator's caller.

In order to fetch the last 90 commits, not just 30, we put these tasks in a loop within the generator. And each time through the loop, we fetch the next batch of commits. The expression `response.headers.get('Link')` at line 19 extracts the URL of the next page from the headers and assigns it to the `repo` variable so that we can access the new URL in the next loop.

If there's no "next page" in the headers, that means there are no more commits to fetch, so we break the loop (line 24).

Within the `getCommits()` function, we define a counter variable to keep track of the number of fetched commits. When the number reaches 90, we stop calling the generator (line 40). The takeaway from this example is that asynchronous generators allow us to smoothly and continuously make several network requests and iterate over the results.

Another interesting use case for asynchronous generator would be fetching images from a photo sharing website like Flickr. The Flickr API provides an endpoint for fetching images based on given keywords.[3] Say you want to create a program that retrieves and processes photos taken in London. Since there are millions of photos of London on Flickr, the API cannot return them all at once. Instead, it returns photos in batches of 100. With an async generator function, you can fetch and navigate the batches asynchronously. Using an async generator would also open up the possibility to seamlessly aggregate photos from several sources.

Wrapping Up

Generators enhance the process of creating iterables by providing an iterative algorithm. An async generator is similar to a sync generator except that it returns a promise rather than a plain object. Use a generator function when you don't want to manipulate the state-maintaining behavior of the object.

Armed with the foundation of asynchronous iterators and generators, you can now make more powerful asynchronous programs. Up next, you'll get the result of multiple promises that are not dependent on each other by using the ES2020 `Promise.allSettled()` method.

3. https://www.flickr.com/services/api/flickr.photos.search.htm