The 80/20 Guide to ES2015 Generators

Valeri Karpov

Table of Contents

Chapter 1: Getting Started

Generators are a powerful new feature in ES2015. Generators are far from a new programming construct - they first appeared in 1975 and Python has had them since Python 2.2 in 2001. However, as you'll see, generators are even more powerful in an event-driven language like JavaScript. In JavaScript (assuming Node.js >= 4.0.0), a **generator function** is defined as shown below.

```
const generatorFunction = function*() {
  console.log('Hello, World!');
};
```

However, if you run `generatorFunction`, you'll notice that the return value is an object.

```
$ node
> var generatorFunction = function*() { console.log('Hello, World!'); };
undefined
> generatorFunction()
{}
```

That's because a generator function creates and returns a **generator object**. Typically, the term **generator** refers to a generator object rather than a generator function. A generator object has a single function, `next()`. If you execute the generator object's `next()` function, you'll notice that Node.js printed 'Hello, World!' to the screen.

```
$ node
> var generatorFunction = function*() { console.log('Hello, World!'); };
undefined
> generatorFunction()
{}
> generatorFunction().next()
Hello, World!
{ value: undefined, done: true }
>
```

Notice that `next()` returned an object, `{ value: undefined, done: true }`. The meaning of this object is tied to the `yield` keyword. To introduce you to the `yield` keyword, consider the following generator function.

```
const generatorFunction = function*() {
  yield 'Hello, World!';
};
```

Let's see what happens when you call `next()` on the resulting generator.

```
$ node
> var generatorFunction = function*() { yield 'Hello, World!'; };
undefined
> var generator = generatorFunction();
undefined
> generator.next();
{ value: 'Hello, World!', done: false }
> generator.next();
{ value: undefined, done: true }
>
```

Notice that, the first time you call `generator.next()`, the `value` property is equal to the string your generator function yielded. You can think of `yield` as the generator-specific equivalent of the `return` statement.

You might be wondering why the return value of `generator.next()` has a `done` property. The reason is tied to why `yield` is different from `return`.

`yield` vs `return`

The `yield` keyword can be thought of as a `return` that allows **re-entry**. In other words, once `return` executes, the currently executing function is done forever. However, when you call `generator.next()`, the JavaScript interpreter executes the generator function until the first `yield` statement. When you call `generator.next()` again, the generator function picks up where it left off. You can think of a generator as a function that can "return" multiple values.

```
const generatorFunction = function*() {
  let message = 'Hello';
  yield message;
  message += ', World!';
  yield message;
};

const generator = generatorFunction();
// { value: 'Hello', done: false };
const v1 = generator.next();
// { value: 'Hello, World!', done: false }
const v2 = generator.next();
// { value: undefined, done: true }
const v3 = generator.next();
```

Re-entry

The most important detail from the above example is that, when `yield` executes, the generator function stops executing until the next time you call `generator.next()`. You can call `generator.next()` whenever you want, even in a `setTimeout()`. The JavaScript interpreter will re-enter the generator function with the same state that it left off with.

```
const generatorFunction = function*() {
  let i = 0;
  while (i < 3) {
    yield i;
    ++i;
  }
};

const generator = generatorFunction();

let x = generator.next(); // { value: 0, done: false }
setTimeout(() => {
  x = generator.next(); // { value: 1, done: false }
  x = generator.next(); // { value: 2, done: false }
  x = generator.next(); // { value: undefined, done: true }
}, 50);
```

`yield` vs `return` revisited

You may be wondering what happens when you use `return` instead of `yield` in a generator. As you might expect, `return` behaves similarly to `yield`, except for done is set to true.

```
const generatorFunction = function*() {
  return 'Hello, World!';
};

const generator = generatorFunction();

// { value: 'Hello, World!', done: true }
const v = generator.next();
```

Case Study: Async Fibonacci

The fact that you can execute `generator.next()` asynchronously hints at why generators are so useful. You can execute `generator.next()` synchronously or asynchronously without changing the implementation of the generator function.

For instance, lets say you wrote a generator function that computes the Fibonacci Sequence. Note that generator functions can take parameters like any function.

```
const fibonacciGenerator = function*(n) {
  let back2 = 0;
  let back1 = 1;
  let cur = 1;
  for (let i = 0; i < n - 1; ++i) {
    cur = back2 + back1;
    back2 = back1;
    back1 = cur;
    yield cur;
  }

  return cur;
};
```

You could compute the n-th Fibonacci number synchronously using the code below.

```
const fibonacci = fibonacciGenerator(10);
let it;
for (it = fibonacci.next(); !it.done; it = fibonacci.next()) {}
it.value; // 55, the 10th fibonacci number
```

However, computing the n-th Fibonacci number synchronously is not a hard problem without generators. To make things interesting, let's say you wanted to compute a very large Fibonacci number **without blocking the event loop**. Normally, a JavaScript for loop would block the event loop. In other words, no other JavaScript code can execute until the for loop in the previous example is done. This can get problematic if you want to compute the 100 millionth Fibonacci number in an Express route handler. Without generators, breaking up a long-running calculation can be cumbersome.

However, since you have a generator function that yields after each iteration of the for loop, you can call `generator.next()` in a `setInterval()` function. This will compute the next Fibonacci number with each iteration of the event loop, and so won't prevent Node.js from responding from incoming requests. You can make your Fibonacci calculation asynchronous without changing the generator function!

```
const fibonacci = fibonacciGenerator(10);
// And compute one new Fibonacci number with each iteration
// through the event loop.
const interval = setInterval(() => {
  const res = fibonacci.next();
  if (res.done) {
    clearInterval(interval);
    res.value; // 55, the 10th fibonacci number
  }
}, 0);
```

For/Of Loops

Remember the for loop you saw for exhausting the Fibonacci generator?

```
for (it = fibonacci.next(); !it.done; it = fibonacci.next()) {}
```

This for loop is a perfectly reasonable way of going through every value of the generator. However, ES2015 introduces a much cleaner mechanism for looping through generators: the `for-of` loop.

```
let fibonacci = fibonacciGenerator(10);
for (const x of fibonacci) {
  x; // 1, 1, 2, 3, 5, ..., 55
}
```

Iterators and Iterables

For/Of loops aren't just for generators. A generator is actually an instance of a more general ES2015 concept called an iterator. An **iterator** is any JavaScript object that has a `next()` function that returns `{ value: Any, done: Boolean }`. A generator is one example of an iterator. You can also iterate over arrays:

```
for (const x of [1, 2, 3]) {
  x; // 1, 2, 3
}
```

However, For/Of loops don't operate on iterators, they operate on iterables. An **iterable** is an object that has a `Symbol.iterator` property which is a function that returns an iterator. In other words, when you execute a For/Of loop, the JavaScript interpreter looks for a `Symbol.iterator` property on the object you're looping `of`.

```
let iterable = {};
for (const x of iterable) {} // Throws an error

// But once you add a Symbol.iterator property, everything works!
iterable[Symbol.iterator] = function() {
  return fibonacciGenerator(10);
};
for (const x of iterable) {
  x; // 1, 1, 2, 3, 5, ..., 55
}
```

A Brief Overview of Symbols

Symbols are another new feature in ES2015. Since this book is about generators, we won't explore symbols in depth, just enough to understand what the mysterious `iterable[Symbol.iterator]` code in the previous example is about.

You can think of a symbol as a unique identifier for a key on an object. For instance, suppose you wrote your own programming language and defined an iterable as an object that had a property named `iterator`. Now, every object that has a property named `iterator` would be an iterable, which could lead to some unpredictable behavior. For instance, suppose you added a property named `iterator` to an array - now you've accidentally broken for/of loops for that array!

Symbols protect you from the issue of accidental string collision. No string key is equal to `Symbol.iterator`, so you don't have to worry about accidentally breaking an iterable. Furthermore, symbols don't appear in the output of `Object.keys()`.

```
Symbol.iterator; // Symbol(Symbol.iterator)

let iterable = {};
iterable[Symbol.iterator] = function() {
  return fibonacciGenerator(10);
};

iterable.iterator; // undefined
Object.keys(iterable); // Empty array!
```

Iterables and Generators

The most important detail to note about generators and iterables is that *generator objects* are iterables, not *generator functions*. In order words, you can't run a for/of loop on a generator function.

```
fibonacciGenerator[Symbol.iterable]; // Undefined
fibonacciGenerator(10)[Symbol.iterable]; // Function

for (const x of fibonacciGenerator) {} // Error!
for (const x of fibonacciGenerator(10)) {} // Ok
```

You may find it strange that the generator's `Symbol.iterator` function returns itself given that generator functions are not iterable. One reason for this decision is that a generator function can take parameters. For instance, looping over `fibonacciGenerator(10)` would not give the same results as looping over `fibonacciGenerator(11)`.

The second most important detail to note about generators and iterables is that `generator[Symbol.iterator]` is a function that returns the generator itself. This means that you can't loop over the same generator twice. Once a generator is done, subsequent for/of loops will exit immediately.

```
const fibonacci = fibonacciGenerator(10);
fibonacci[Symbol.iterator]() === fibonacci; // true
for (const x of fibonacci) {
  // 1, 1, 2, 3, 5, ..., 55
}
for (const x of fibonacci) {
  // Doesn't run!
}
```

Error Handling

One detail that has been glossed over so far is how generators handle exceptions. What happens when you divide by zero in a generator? As you might have guessed, the `generator.next()` call throws an error.

```
const generatorFunction = function*() {
  throw new Error('oops!');
};

const generator = generatorFunction();

// throws an error
generator.next();
```

The error's stack trace reflects the fact that `next()` was the function that called the function that threw the error. In particular, if you call `next()` asynchronously, you will lose the original stack trace.

```
const generatorFunction = function*() {
  throw new Error('oops!');
};

const generator = generatorFunction();

setTimeout(() => {
  try {
    generator.next();
  } catch (err) {
    /**
     * Error: oops!
     * at generatorFunction (book.js:2:15)
     * at next (native)
     * at null._onTimeout (book.js:18:21)
     * at Timer.listOnTimeout (timers.js:89:15)
     */
    err.stack;
  }
}, 0);
```

Re-entry With Error

When you think of generators, you need to think of 2 functions: the generator function itself, and the function that's calling `next()` on the generator. When the generator function calls `yield` or `return`, the calling function regains control. When the calling function calls `next()`, the generator function regains starts running again. There's another way the calling function can give control back to the generator function: the `throw()` function.

The `throw()` function is a way for the calling function to tell the generator function that something went wrong. In the generator function, this will look like the `yield` statement threw an error. You can then use try/catch to handle the error in the generator function. As you'll see in the next section, this pattern is indispensable for working with asynchronous code and generators.

```
const fakeFibonacciGenerator = function*() {
  try {
    yield 3;
  } catch (error) {
    error; // Error: Expected 1, got 3
  }
};
const fibonacci = fakeFibonacciGenerator();

const x = fibonacci.next();
fibonacci.throw(new Error(`Expected 1, got ${x.value}`));
// { value: undefined, done: true }
fibonacci.next();
```

Case Study: Handling Async Errors

Remember that there are two functions involved in generator functions: the generator function itself, and the function that calls `next()` on the generator object. So far in this book, the function that calls `next()` hasn't done any real work. The most complex example is the async Fibonacci example, which acted as a scheduler for the Fibonacci generator.

One pivotal feature of generators is that the `next()` function can take a parameter. That parameter then becomes the return value of the `yield` statement in the generator function itself!

```
const generatorFunction = function*() {
  const fullName = yield ['John', 'Smith'];
  fullName; // 'John Smith'
};

const generator = generatorFunction();
// Execute up to the first `yield`
const next = generator.next();
// Join ['John', 'Smith'] => 'John Smith' and use it as the
// result of `yield`, then execute the rest of the generator function
generator.next(next.value.join(' '));
```

Once you combine this feature with the `throw()` function, you have everything you need to have the generator function `yield` whenever it needs to do an asynchronous operation. The calling function can then execute the asynchronous operation, `throw()` any errors that occurred, and return the result of the async operation using `next()`.

This means that your generator function doesn't need to worry about callbacks. The calling function can be responsible for running asynchronous operations and reporting any errors back to the generator function. For instance, the below example shows how to run a generator function that yields an asynchronous function without any errors.

```
const async = function(callback) {
  setTimeout(() => callback(null, 'Hello, Async!'), 10);
};

const generatorFunction = function*() {
  const v = yield async;
  v; // 'Hello, Async!'
};

const generator = generatorFunction();
const fn = generator.next();
fn(function(error, res) {
  generator.next(res);
});
```

Now suppose that the `async` function returns an error. The calling function can then call `throw()` on the generator, and now your generator function can handle this asynchronous operation with try/catch! As you'll see in the coroutines chapter, this idea is the basis of the co library.

```
const async = function(callback) {
  setTimeout(() => callback(new Error('Oops!')), 10);
};

const generatorFunction = function*() {
  try {
    yield async;
  } catch (error) {
    error; // Error: Oops!
  }
};

const generator = generatorFunction();
const fn = generator.next();
fn(function(error, res) {
  generator.throw(error);
});
```