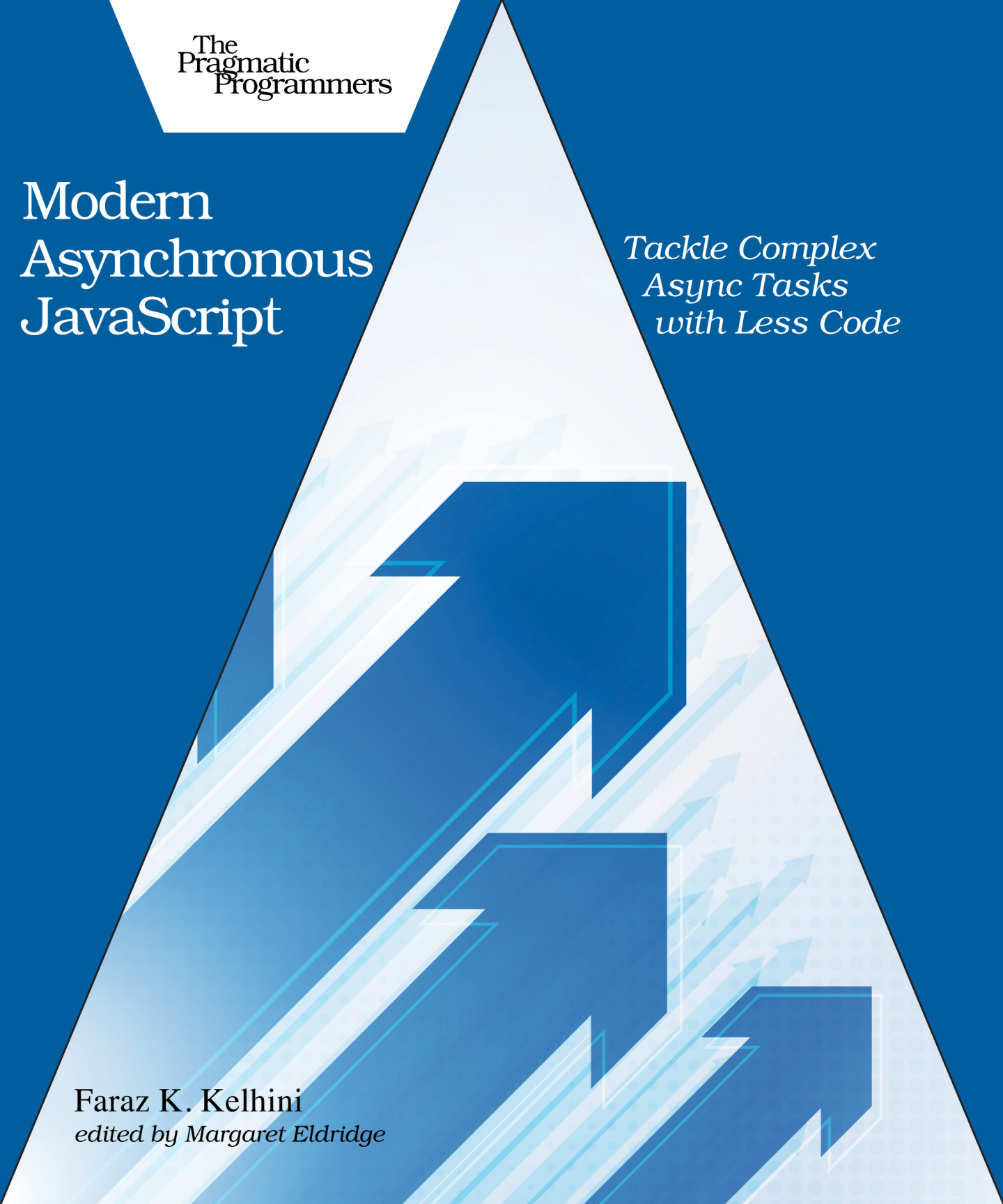
The Pragmatic Programmers

Modern Asynchronous JavaScript

Tackle Complex Async Tasks with Less Code



Faraz K. Kelhini
edited by Margaret Eldridge

Early Praise for *Modern Asynchronous JavaScript*

For developers looking to level up on both their command and understanding of asynchronous JavaScript, especially newer and less familiar methods on the Promise API, this is an indispensable reference and solid guide. Faraz Kelhini walks you through the thorny problems around working asynchronously with iterables and provides excellent and inspiring examples and easy-to-study source code.

➤ **Karl Stolley**
Web Developer, Researcher, Professor, and Author of *Programming WebRTC*, Illinois Institute of Technology

Faraz Kelhini's efficient organization and fine writing make *Modern Asynchronous JavaScript* a valuable introduction to the topic. The dozens of succinct code examples that are included round out an indispensable package for the intermediate-to-advanced JavaScripter.

➤ **Victor Gavenda**
Former Executive Editor, Pearson Education/Peachpit Press

JavaScript moves fast, and even seasoned developers need to pay attention to keep up with the language. *Modern Asynchronous JavaScript* is a pithy primer on newly introduced JavaScript features that help make asynchronous code more concise and safer. Anyone writing asynchronous JavaScript code, whether it runs in a browser or in Node.js, should pay attention. All of the ugly JavaScript code I've written recently wishes I'd read this book sooner.

➤ **Lukas Mathis**
Software Engineer, UI Designer, and Author of *Designed for Use*, Appway

I've worked with countless technical writers throughout my career, and Faraz stands out among them for his ability to clearly and concisely explain complex topics. His work in *Modern Asynchronous JavaScript* meets those high standards and then some. Even for a non-dev like myself, Faraz makes it easy to grasp the concepts at work.

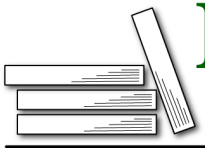
➤ **Matt Angelosanto**
Managing Editor, LogRocket Blog

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

Contents

Acknowledgments

Writing and producing a book requires a team effort. My sincere thanks to the entire Pragmatic Bookshelf team for their great efforts to get this book to fruition. In particular, I would like to thank my editor, Margaret Eldridge, whose watchful eyes identified gaps in my writing and saved me from mistakes.

A big thanks to experts who reviewed the book prior to publication, including Karl Stolley, Lukas Mathis, Michael Fazio, Trevor Burham, Kevin Gisi, Matthew Margolis, Mike Riley, Francesco Piccoli, and Stefan Turalski. These wonderful developers offered very helpful insight on the code quality and suggested ways to improve.

My love to my family for their support and putting up with an author's crankiness. I am grateful for that. And special thanks to my dear friends, Mahsa and Asal, for their encouragement during the writing of this book. You're amazing!

Finally, thanks to the eager readers who bought the book while it was in beta. You put your trust in me, and I appreciate that.

Preface

Modern applications increasingly rely on asynchronous programming to perform multiple tasks at the same time, and JavaScript is quickly evolving to address this need. Many new features of JavaScript are designed to only work asynchronously. As a result, gaining the knowledge to perform async tasks effectively is a must for today's JavaScript programmers.

Designing responsive asynchronous programs might be challenging at first, but once you get the hang of it, the outcome is rewarding. This book is here to help you with that. You'll find multiple examples to help you write advanced programs using the new capabilities of JavaScript introduced in ES2020, ES2021, and ESNext. You'll also discover various techniques to manage and coordinate the asynchronous parts of your code efficiently.

Make sure you actually type and execute the code examples as you follow along in the book. Some examples may appear simple, but there's a big difference between reading the code and being able to write it on your own.

Who Is This Book For?

If you're an intermediate to advanced JavaScript programmer or web developer, this book is for you. We'll focus squarely on the practical aspects of asynchronous programming—that is, what each technique is designed to accomplish and how to use it in your program. If you've been using third-party libraries to manage asynchronous code and now want to switch to native JavaScript APIs, you'll benefit from this guide.

What You Should Know

To use this book, you should already know JavaScript and HTML. Use of HTML will be infrequent and fairly basic, and I'll explain each JavaScript example in detail. So even if your JavaScript or HTML is rusty, you'll be able to understand how the code is working.

What's in This Book?

Modern Asynchronous JavaScript is deliberately succinct. You won't learn everything about asynchronous programming so that you can quickly pick up key tips and tricks. In each chapter, we'll dive straight into a different topic, so feel free to jump to any chapter in the book you want.

Chapter 1 shows you how to define or customize the iteration behavior of JavaScript objects using custom iterators.

Chapter 2 is where you learn to use a generator function as a shortcut to create iterators. Generators are useful when you don't need to manipulate the state-maintaining behavior of an iterator.

Chapter 3 introduces the `Promise.allSettled()` method and compares it to its older sibling `Promise.all()`. You'll use `Promise.allSettled()` to execute multiple async tasks simultaneously and process the outcome even if some of them fail.

Chapter 4 gives you tips to protect your app against server downtime while improving its performance with `Promise.any()` from ES2021.

Chapter 5 explains how to use the `Promise.race()` method to set a time limit for async tasks to avoid entering a state of prolonged or endless waiting.

Chapter 6 is all about the AbortController API. You'll learn to use this API to cancel pending async requests when the user clicks a cancel button.

Chapter 7 covers top-level `await` and how to make use of it to initialize resources, define dependency paths dynamically, and load dependencies with a fallback implementation.

Online Resources

To download the example code used in the book, please visit the Pragmatic Bookshelf website.[1] You can submit feedback and errata entries, get up-to-date information, and join in the discussions on the book's forum page. If you're reading the book in PDF format, you can view or download a specific example by clicking on the little gray box above the code.

Next up is the Introduction. If you're an experienced JavaScript programmer, most of the concepts in the Introduction will be familiar, whereas if you're more intermediate, you might find the discussion of callbacks and promises helpful.

1. https://www.pragprog.com/titles/fkajs

Introduction

The introduction of the *promise* object in ES2015 changed the way we write asynchronous programs in JavaScript. Similar to callbacks and events, a promise defines a block of code to be executed once an operation is finished. But unlike the old approaches, it gives us a robust mechanism to track the state of multiple asynchronous tasks and verify whether they are all successful.

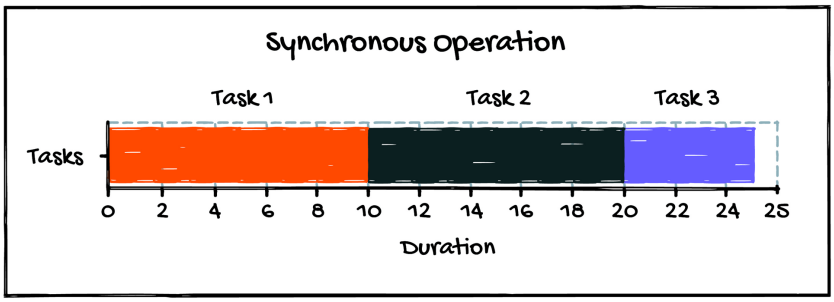
But what exactly do we mean when we say a program is asynchronous?

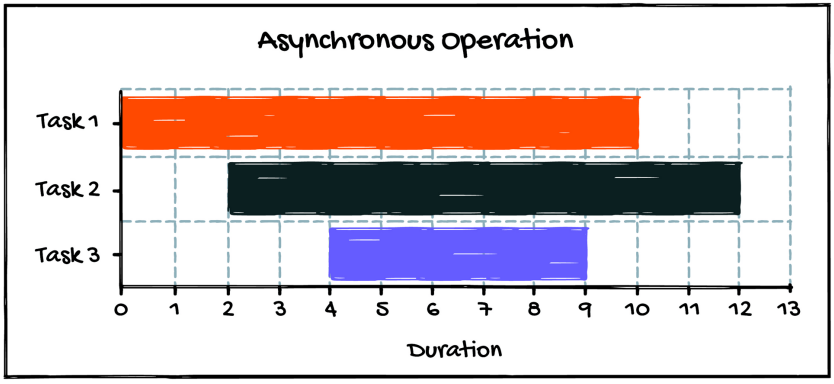
Demystifying Asynchronous Execution

The concept of asynchrony determines whether a task can start executing before another task is finished. In a synchronous execution, the program pauses until the current task is completed before moving to the next task. But in an asynchronous execution, the program continues executing even when the previous operation hasn't finished yet.

It helps to think of synchronous executions as a line of people waiting to buy movie tickets. If you are at the end of the line, you can't buy a ticket until all the people in front of you have bought theirs. Think of asynchronous executions like ordering food in a restaurant. You don't have to wait for other people who have come earlier to get their food before you can order yours. Everyone can order food at any time and receive it when it's ready. Depending on the type of food you order, you may get your food sooner or later than other people.

A bar chart can better illustrate the difference:





It's important to understand that asynchrony and multithreading are two completely different concepts. JavaScript is often considered a single-threaded language, mostly because web browsers run one thread per global environment.

But JavaScript, as a programming language, isn't single-threaded. And there are some JavaScript environments that are multi-threaded. With the introduction of Web Workers, you can even have multiple threads on web browsers (they don't run on the same global environment though).

But even on a single thread, JavaScript is capable of executing asynchronous code. Threads aren't the only way to perform tasks in parallel. Imagine a restaurant with only one cook. The cook can start a burger cooking and set a timer, then put some pizza in the oven and set a timer. He can clean the kitchen while the food is cooking. When the timer goes off, he takes the food out and serves it. If the switch between tasks is efficient enough, you won't notice any lags.

You can imagine a multi-threaded environment like a restaurant that has multiple cooks. One cook is responsible for cooking burgers, another one is responsible for cooking pizzas. But now you have to pay more to keep the

other cook and make sure they share the resources properly so there will be no conflict in the kitchen. In other words, threading describes the number of workers, but asynchrony is about tasks.

Working with Events

The JavaScript language was created to add interactivity to web pages, so it needed a way to detect user actions and react to them. JavaScript's solution for this need was events: whenever you interact with a web page, such as when clicking a button, an event takes place, allowing JavaScript code to react to the action.

Although events have enabled JavaScript programs to detect interaction with objects and react to them, its lack of flexibility has been a significant problem for some developers. For example, events can happen before the program starts listening to them.

If the user moves the cursor over the box before the `onmouseover` property is assigned, the code won't be executed. Therefore, it's always necessary to make sure the event handlers are assigned before an event occurs.

Additionally, events can be frustrating to use when working with more than one element. For instance, there's no easy way to detect whether a collection of images have been loaded, or check the order in which they have loaded.

Working with Callback Functions

Perhaps the simplest asynchronous execution in JavaScript is the `setTimeOut()` function. This function defines a callback function to be executed in the future independently of the main program flow, so it doesn't block the execution of the program.

Another common asynchronous execution in JavaScript is Ajax. Similar to `setTimeOut()`, an Ajax call doesn't stop the execution flow of the program. It specifies a piece of code to run as soon as the code receives data from a server.

The main advantage of using callbacks is that the program can continue doing useful work while other tasks are running, so it feels more responsive and there will be fewer "hangs."

Nesting callbacks is a common practice in JavaScript. But nesting too many callbacks can make the code hard to understand and lead to a maintainability issue known as callback hell. The following code is an example of callback hell:

intro/intro_ex01.js

```
firstFunction((x) => {
  // process...
  secondFunction(x, (y) => {
    // process...
    thirdFunction(y, (z) => {
      // And so on...
    });
  });
});
```

Callback hell is the result of poor coding practice. If you nest more than a few callbacks, your code will quickly become unmanageable. One way to fix this code is to define each function separately:

intro/intro_ex02.js

```
firstFunction((x) => {
  // process...
  secondFunction(x);
});

secondFunction(x, (y) => {
  // process...
  thirdFunction(y);
});

thirdFunction(y, (z) => {
  // process...
  fourthFunction(z);
  // And so on...
});
```

By moving functions to the top level, we'll have a shallower code that is separated into small logical sections. This small change results in a more manageable code.

Still, the callback model is difficult to work with when more complex functionality is needed. With promises, you can easily chain multiple asynchronous tasks dynamically. For example, you can make two async requests simultaneously, wait for the results, and then determine what other async task to do based on the intermediate result. Or you can use a promise to track the state of multiple async operations and react as soon as one of them is completed.

That doesn't mean you should stop using callbacks though. Callbacks are still useful when your code may receive a notification more than once. For instance, the setInterval() method defines a callback function to be executed repeatedly, with a fixed time delay between each call. You can't call a promise again once it's executed, but you can call a callback function multiple times.

Introducing Promises

Originally, the promise construct was used by libraries such as Q, RSVP.js, and WinJS. But it quickly became popular enough to encourage the Ecma Technical Committee to take advantage of it in the ES2015 standard.

Newer JavaScript APIs use the promise object rather than the old-school callback function. For example, the Fetch API not only provides a simpler syntax compared to the complex API of XMLHttpRequest but also prevents callback hell by returning a promise. Let's look at a simple example:

intro/intro_ex03.js

```
const promise = fetch('https://eloux.com/async_js/examples/1.json');

promise.then((result) => {
  // process
}, (error) => {
  console.log(error);
});
```

The fetch() method allows us to retrieve files across the network. This method returns a promise object that acts as a placeholder for the future result of the operation. To react to the result, we use the then() method.

then() accepts two functions as parameters. The first function is called once the promise is succeeded, and the fulfillment value is passed to the function as an argument. The second function is called only if the promise is failed, with the rejection reason passed as its argument.

The promise returned by fetch() remains in the pending state until the Ajax request is completed. The spec uses the term *unsettled* to describe a promise that's pending. After receiving data, the promise transitions either to fulfilled or rejected state. At this point, the promise is considered *settled*.

It's worth noting that a promise cannot succeed or fail more than once. It also cannot switch from failure to success or vice versa. Both arguments of then() are optional. So when we don't need to listen for fulfillment or rejection, we can omit the related argument.

Creating Settled Promises

When working your way through the examples in this book, you may want to key in and alter each example to gain practice. The static Promise.resolve() and Promise.reject() methods allow you to quickly create settled promises and see how the code works when you give it a different value.

For example, the following code creates a promise that's already fulfilled with the value 10:

intro/intro_ex05.js
```
const promise = Promise.resolve(10);

promise.then((data) => {
  console.log(data);    // ⇒ 10
});
```

Here, we have a settled promise that represents only a known value. This promise will never be in the rejected state, so adding a rejection handler is pointless. To create a promise in the rejected state, we can pass a value to the Promise.reject() method, like this:

intro/intro_ex06.js
```
const promise = Promise.reject('Error!');

promise.then(null, (error) => {
  console.error(error);    // ⇒ Error!
});
```

This code creates a settled promise that's rejected with a predefined value. If we add a fulfillment handler to this code, it will never be called.

Note that we won't be covering testing or debugging extensively in this book. Don't forget to take advantage of these methods when debugging your code.

Handling Rejection

There are two primary ways to handle a rejected promise. In the previous example, we used the pattern then(fulfill, reject), but we can also use the catch() method:

intro/intro_ex07.js
```
const promise = Promise.reject('Error!');

promise.catch((error) => {
  console.error(error);    // ⇒ Error!
});
```

When chaining promises and an error occurs, the interpreter skips all then() methods that follow and executes the first catch() method it can find. Consider the following code:

intro/intro_ex08.js
```
const promise = Promise.reject('Error!');

promise.then((value) => {
  // this won't be executed
  console.log('Hi!');
}).then((value) => {
```

```
  // this won't be executed either
}).catch((error) => {
  console.error(error);
});

// logs:
// ⇒ Error!
```

Here, we've used catch() to combine multiple rejection handlers into one case at the end of the chain.

It's important to understand that the pattern then(fulfill, reject) isn't always equivalent to then(fulfill).catch(reject). Using these patterns interchangeably could potentially lead to an error. For example:

intro/intro_ex09.js

```
const promise = Promise.resolve(10);

promise.then((result) => {
  throw new Error();
}, (error) => {
  // this won't be executed
  console.error('An error occurred in the fulfillment handler');
});

// logs:
// ⇒ Uncaught (in promise) Error
```

The fulfillment handler in this code throws an error, but the rejection handler isn't executed. Switching to catch() can fix this problem:

intro/intro_ex10.js

```
const promise = Promise.resolve(10);

promise.then((result) => {
  throw new Error();
}).catch((error) => {
  console.error('An error occurred in the fulfillment handler');
});

// logs:
// ⇒ An error occurred in the fulfillment handler
```

The rejection handler in the then() method cannot handle errors that occur in the fulfillment handler. You'd need to chain an additional then() to do that or, better yet, use the catch() method.

Managing Multiple Concurrent Promises

We can chain multiple promises to perform additional asynchronous operations one after another. But what if we want to execute multiple promises at

the same time and react as soon as one or all of them are settled? JavaScript provides the following methods for this purpose:

- `Promise.race()` – lets you know as soon as one of the given promises either fulfills or rejects
- `Promise.allSettled()` – lets you know when all of the given promises either fulfill or reject
- `Promise.all()` – lets you know as soon as one of the given promises rejects or when all of them fulfill
- `Promise.any()` – lets you know as soon as one of the given promises fulfills or when none of them fulfills

`Promise.race()` and `Promise.all()` have been around since the introduction of the promise object in ES2015, while `Promise.allSettled()` and `Promise.any()` were introduced in ES2020 and ES2021, respectively. Later in the book, we'll learn how to take advantage of each of these methods in our asynchronous programs. But before we do that, let's look at an important technique for processing data from external sources: asynchronous iteration.

CHAPTER 1

Creating Custom Asynchronous Iterators

Iterating over collections is one of the most common tasks in programming. That's why almost every new edition of ECMAScript introduces features to improve the iteration capabilities of the language. These new features make coding easier and more efficient, and they allow you to perform tasks that would previously require external libraries.

As a JavaScript developer, you'll often work with synchronous data like customer order information stored via in-memory lists. *Iterators* give you a neat way to process them, allowing you to move through the elements in the data structure. What about when you need to process asynchronous data via web APIs, like stock prices? Synchronous iterators cannot represent such data sources, so that's where you'll need to use asynchronous iteration.

The process of asynchronous iteration is a bit like cooking pancakes on a griddle. The first step is to heat a lightly oiled griddle. Then you pour the batter onto the griddle, filling it up with pancakes. You wait until the edges start to bubble, indicating they're ready to flip. Each pancake will be ready at a different time, depending on when you poured the batter and how even the heat source is. You flip the pancakes and take them off as they're ready. So while preparing this simple meal, in a sense, you are acting as an *asynchronous iterator* function.

In this chapter, you'll get aquainted with iterators and learn to create your own custom iterators. Because sync and async iterators are closely related, we'll start with synchronous iterators. Then you'll use what you've learned to create async iterators.

Getting Ready

Iterators have been around since ES2015, so browser support is solid. Following are the JavaScript features we'll discuss in this chapter along with some links to up-to-date sources for browser support:

- Async functions[1]
- for await...of loops[2]
- async and await keywords[3]

In the Node environment, you'll need a minimum Node version of 7.6.0 to use the async and await keywords and a minimum version of 10.0.0 to use for await...of. You can also use a Babel plugin to make use of the features in older browsers or Node versions.[4]

Creating a Custom Iterator

Collection objects (including Array, Set, and Map) come with built-in iterators that allow us to navigate their values. So, we don't have to create our own iterators. But sometimes these objects don't serve our purpose. What if we want to customize the iteration behavior of an object to return values backward or randomly? Or iterate over a plain object or class, both being not iterable by default? In that case, we'll need to define our own Symbol.iterator.

Set and Map

ES2015 introduced two new data structures: Set and Map. A Set is like an array, a collection of values, but it can't contain duplicates and the items cannot be accessed individually. Instead, a Set provides methods to iterate over its elements or assert if it contains a value. A Map is similar to an object, mapping keys to values, but provides more functionality in certain situations. Maps are usually used as caches for storing data, which then can be readily accessed when needed.

1. https://caniuse.com/async-functions
2. https://caniuse.com/mdn-javascript_statements_for_await_of
3. https://caniuse.com/mdn-javascript_operators_await
4. https://babeljs.io/docs/en/babel-plugin-proposal-async-generator-functions

Iterable or Iterator?

Iterable is an object that allows its values to be looped over in a for...of construct. It does so by providing a method whose key is Symbol.iterator. That method should be able to produce any number of iterators. Iterator, on the other hand, is an object that's used to obtain the values to be iterated.

Remember, to be classified as an iterable, an object must come with a Symbol.iterator property and specify the return value for each iteration. In the following example, we have a plain object that's iterable because we've defined an iterable protocol that allows us to access the items of the object one at a time:

iterators/iterator_ex03.js

```
const collection = {
  a: 10,
  b: 20,
  c: 30,
  [Symbol.iterator]() {
    let i = 0;
    const values = Object.keys(this);
    return {
      next: () => {
        return {
          value: this[values[i++]],
          done: i > values.length
        }
      }
    };
  }
};

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}
```

The next() method returns the iteration result of the object. This method always returns an object with two properties: value and done. The value property holds the value returned by the iterator. And the done property holds a Boolean value, which is set to true only when there is no more value to return.

We need a way to keep track of the values we want to return, so we define a counter variable with the initial value of 0. To define the return value, we use Object.keys(), which obtains an array of the object's property names. Now if we

call collection[Symbol.iterator](), it returns an object containing a next() method. And with each call to next(), the method returns a {value, done} pair.

Using next() isn't the only way to iterate over iterable objects. The for...of statement lets us create a loop and easily repeat the same function on iterable objects. for...of works better if we want to quickly get the values of all items in the object. next(), on the other hand, is more verbose but allows us to see what's happening in each iteration. Let's rewrite this example using for...of to simplify the code:

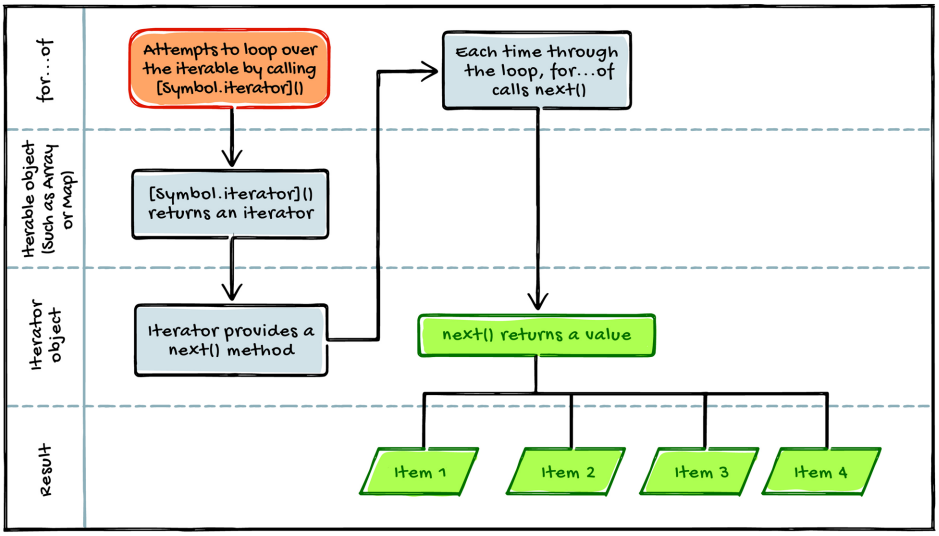
iterators/iterator_ex03_with_for-of.js

```
const collection = {
  a: 10,
  b: 20,
  c: 30,
  [Symbol.iterator]() {
    let i = 0;
    const values = Object.keys(this);
    return {
      next: () => {
        return {
          value: this[values[i++]],
          done: i > values.length
        }
      }
    };
  }
};

for (const value of collection) {
  console.log(value);
}

// logs:
// ⇒ 10
// ⇒ 20
// ⇒ 30
```

[Symbol.iterator]() works like any other method except that it's automatically called if we use for...of on the object. The following image shows how for...of works behind the scenes:



The iterator object is designed to maintain an internal pointer to a position in the items; and each time through the loop, it gives the succeeding value.

Now, what happens when we use for...of to iterate over a native object that's already iterable, like an array? Calling [Symbol.iterator]() on an array will return the result of the values() method because that's the default iterator of arrays.

While values() is the default iterator of sets and arrays, entries() is the default iterator of maps. An object may have several iterators, such as keys(), values(), and entries(), but only one of them serves as the default iterator. Built-in iterators make it possible to iterate over collection objects easily, and custom iterators allow us to define or customize the iteration behavior of objects. But if we want to work with asynchronous sources, we'll need to create custom asynchronous iterators.

Creating a Custom Asynchronous Iterator

Suppose we want a function that retrieves several URLs and processes the result of each URL separately before moving on to the next. In other words, we want to retrieve and parse the URLs asynchronously, but not the results. That's one scenario where an asynchronous iterator is useful.

An asynchronous iterator is very similar to a regular non-async iterator except that its next() method returns a promise rather than a plain object. Thus, instead of immediately returning the result, the promise will provide the value (or failure reason) at some point in the future (like the griddle from our opening analogy, which yields a pancake at some point).

An object is classified as asynchronous iterable when it has a Symbol.asyncIterator method that returns an asynchronous iterator. Without further ado, let's look at a simple example of an async iterable object:

iterators/iterator_ex04.js

```
const collection = {
  a: 10,
  b: 20,
  c: 30,
  [Symbol.asyncIterator]() {
    const keys = Object.keys(this);
    let i = 0;
    return {
      next: () => {
        return new Promise((resolve, reject) => {
          setTimeout(() => {
            resolve({
              value: this[keys[i++]],
              done: i > keys.length
            });
          }, 1000);
        });
      }
    };
  }
};

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

iterator.next().then(result => {
  console.log(result);  // ⇒ {value: 10, done: false}
});

iterator.next().then(result => {
  console.log(result);  // ⇒ {value: 20, done: false}
});

iterator.next().then(result => {
  console.log(result);  // ⇒ {value: 30, done: false}
});

iterator.next().then(result => {
  console.log(result);  // ⇒ {value: undefined, done: true}
});
```

Typically, a sync iterator returns an object containing a next() method. With each call to next(), a {value, done} pair is returned with the value property containing the yielded value. Similarly, an async iterator returns an object containing a next() method. But rather than a plain object, next() returns a promise (line 10) that fulfills to {value, done}.

For the sake of simplicity, we've used the setTimeout() method to resolve the promise after one second. But in a real-world example we're more likely to make a call to an API and wait for a response.

Retrieving URLs Separately

Let's look at a more concrete example. Remember the use case in Creating a Custom Asynchronous Iterator, on page 5, for an asynchronous iterator that retrieves several URLs and processes them separately? Here's how we can implement it:

iterators/iterator_ex05.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',
5       ];

        srcArr[Symbol.asyncIterator] = function() {
          let i = 0;
          return {
10          async next() {
              if (i === srcArr.length) {
                return {
                  done: true
                };
15            }
              const url = srcArr[i++];
              const response = await fetch(url);
              if (!response.ok) {
                throw new Error('Unable to retrieve URL: ' + url);
20            }
              return {
                value: await response.json(),
                done: false
              };
25          }
          };
        };

        const iterator = srcArr[Symbol.asyncIterator]();
30
        iterator.next().then(result => {
          console.log(result.value.firstName);  // ⇒ John
        });

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

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

We begin with replacing the default iterator of srcArr (line 7). By assigning a new function to the Symbol.asyncIterator property of the array, we can define our custom iterator. Within the function, we create a counter variable to keep track of the array index. Then we return an object containing an async next() method.

It's essential to use the async keyword here so that the function returns a promise each time it's called. Line 11 checks whether the end of array has been reached by comparing the value of the counter variable to the length of the array. If that's true, there's no point in continuing the iteration.

It's also important to ensure the response was successful (status in the range 200–299) before proceeding further. Check the value of response.ok (line 18). If it doesn't have a value of true, then there's been an error fetching the URL.

Async iterators are invaluable tools when working with web APIs. Often, the data can only be retrieved in the form of stream or pagination. Iterators make it possible to gracefully obtain the amount of data we need and process them. We'll see an example of this in the next chapter where we write a function to retrieve a specified number of commits from the GitHub API.

Now that you have a foundation in the mechanics of iterators, let's find out how to quickly loop over their items.

Iterating over Async Iterables with for…await…of

In the previous example, we called the [Symbol.asyncIterator]() of the iterable to get an iterator object and called its next() method to resume the execution of the iterator. But sometimes we want a more straightforward way of accessing the items of an async iterable. We want to quickly get the result of all promises and terminate the loop automatically once the done property has a value of true.

The for..of loop does allow you to loop over iterable objects, but it doesn't work with asynchronous iterables (returns undefined). ES2018 introduced for...await...of as a variant of for...of that can iterate over both sync and async iterables.

To see this statement in action, let's look at this rewritten version of the previous example. Notice how for...await...of saves lines of code by executing the same statement for the value of each property:

iterators/iterator_ex06.js

```
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] = function() {
  let i = 0;
  return {
    async next() {
      if (i === srcArr.length) {
        return {
          done: true
        };
      }
      const url = srcArr[i++];
      const response = await fetch(url);
      if (!response.ok) {
        throw new Error('Unable to retrieve URL: ' + url);
      }
      return {
        value: await response.json(),
        done: false
      };
    }
  };
};

(async function() {
  for await (const url of srcArr) {
    console.log(url.firstName);
  }
})();

// logs:
// ⇒ John
// ⇒ Peter
// ⇒ Anna
```

When we run this code, the JavaScript engine executes the Symbol.asyncIterator method of the object to obtain an asynchronous iterator. With each iteration of the loop, the iterator executes the next() method and returns a promise (this happens behind the scenes). As soon as the promise is fulfilled, the value of the value property is assigned to url. As with for...of, the loop will continue until done has a value of true.

Because we can use for...await...of only inside asynchronous functions and generators, we've wrapped the statement in an IIFE (immediately invoked function expression). Otherwise, the code would throw a SyntaxError.

It's a common practice to enclose for...await...of in a try...catch statement. This way when a promise rejects, we can gracefully handle the rejection:

iterators/iterator_ex07.js

```
const collection = {
  [Symbol.asyncIterator]() {
    return {
      next: () => {
        return Promise.reject(new Error('Something went wrong.'))
      }
    };
  }
};

(async function() {
  try {
    for await (const value of collection) {}
  } catch (error) {
    console.error('Caught: ' + error.message);
  }
})();

// logs:
// ⇒ Caught: Something went wrong.
```

This iterator returns a Promise object that is rejected. Without try...catch, we would see an Uncaught (in promise) Error in the browser's console. The try...catch statement allows us to specify a response should an exception be thrown.

The for...await...of statement provides a convenient, concise way of accessing the items of an async iterable. By wrapping it in a try...catch statement, we have the ability to handle promise rejections the way we want.

To Infinity and Beyond!

An interesting aspect of iterators is that they are infinite. For instance, you may have a Fibonacci iterator that delivers an infinite sequence.

Detecting Whether an Object Is Iterable

So far we've been working with iterable objects that we "own." Before iterating over an object that we haven't created, it's important to ensure that the object is iterable; otherwise, the code may throw a TypeError:

iterators/iterator_ex08.js

```
// a plain object
const collection = {
  a: 10,
  b: 20,
```

```
  c: 30
};

for (let value of collection) {  // ⇒ TypeError: collection is not iterable
  console.log(value);
}
```

Detecting whether an object is iterable isn't complicated: check for the existence of [Symbol.iterator] on the object and ensure it's a function (the for...of construct performs a similar check before execution):

iterators/iterator_ex09.js

```
function isIterable(object) {
  return typeof object[Symbol.iterator] === "function";
}

console.log(isIterable({a: 10, b: 20}));  // ⇒ false
console.log(isIterable(123));             // ⇒ false

console.log(isIterable("abc"));           // ⇒ true
console.log(isIterable([10, 20, 30]));    // ⇒ true
```

The process of detecting async iterables is almost the same, except that you'll need to look for [Symbol.asyncIterator] on the object like this:

iterators/iterator_ex10.js

```
const collection = {
  [Symbol.asyncIterator]: async function() {
    // ...
  }
};

function isAsyncIterable(object) {
  return typeof object[Symbol.asyncIterator] === "function";
}

console.log(isAsyncIterable(collection));  // ⇒ true
```

When working with objects originating from external sources, you might expect that they will always remain in a certain shape. But writing your code based on this assumption is a recipe for error. Before using the objects you haven't created, always ensure they have the property/method you're looking for.

Wrapping Up

'Symbol.iterator' and 'Symbol.asyncIterator' are the cornerstone of iterables, and you can take advantage of them to create custom iterators. Custom iterators allow you to define or customize the iteration behavior of JavaScript objects. You can't use 'for...of' to iterate over async iterables; instead you should use 'for...await...of'. Next up, we'll work with generator functions to enhance your JavaScript code.

CHAPTER 2

Enhancing Custom Iterators with Generators

Custom iterators are powerful tools that allow us to define how an object should be navigated. However, they require meticulous programming to maintain their internal state. What if you want a quick way of defining an iterator without going through the hassle of implementing the iterable protocol? Luckily, JavaScript provides *generator* functions as a shortcut to create iterators.

Every generator function is an iterator, but the opposite is not true. You may want to define a custom iterator protocol directly when you need an object with complicated state-maintaining behavior or you want to provide other methods besides `next()`. But in most other cases, you are best suited to define a generator that returns an iterator because state maintenance is mainly done for you.

In this chapter, we'll cover how synchronous and asynchronous generators work by adapting the examples from the previous chapter. Then we'll look at a real-world example so you can see for yourself where asynchronous generators are useful.

Getting Ready

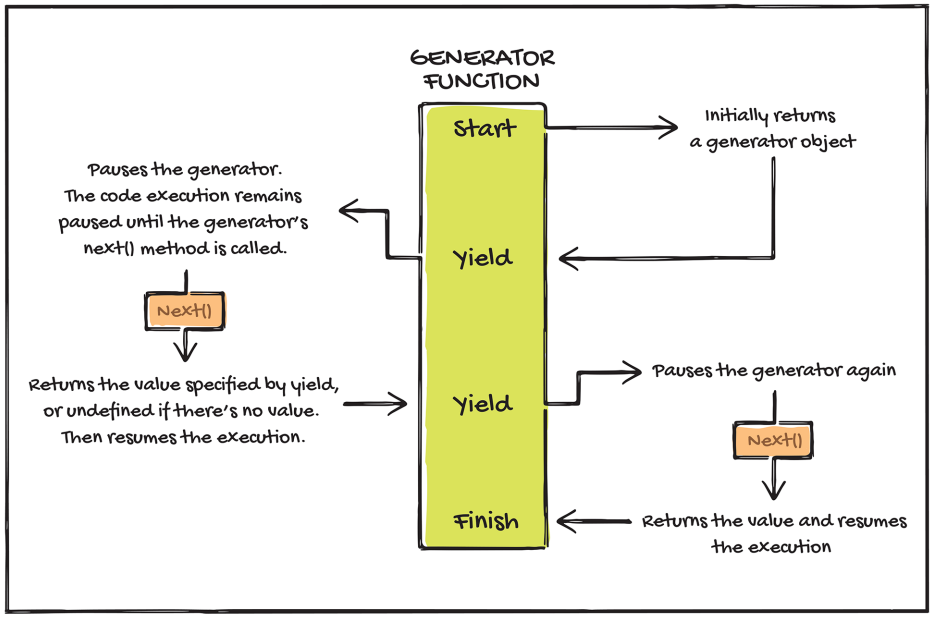
As with iterators, generator functions have been available since ES2015 and browser support is solid.[1] In the Node environment, you'll need a minimum

1. https://caniuse.com/es6-generators

Node version of 4.0.0. And if you want to use generators in older browsers or Node versions, you can use a Babel plugin.[2]

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 2. 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*() {
```

2. https://babeljs.io/docs/en/babel-plugin-proposal-async-generator-functions#installation

```
    for (let key in this) {
      yield this[key];
    }
  }
};

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 7, 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;
    }
  }
```

```
- }
45
- 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

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

the result of multiple promises that are not dependent on each other by using the ES2020 Promise.allSettled() method.

CHAPTER 3

Fetching Multiple Resources

Suppose you want to take an action after multiple async requests have completed, regardless of their success or failure. For example, you need to obtain data from four separate web APIs and process the result, but there might be a network error for a resource that you can live without. The Promise.all() method isn't suitable for this task because a single network error will cause the method to reject immediately.

Fortunately, ECMAScript provides a newer tool that's designed to report the outcome of all requests: Promise.allSettled(). With this method, we can track the state of multiple promises without letting any promise spoil the result of others. We'll start this chapter by examining a common async task: executing multiple promises and handling the result. Once you've learned about potential pitfalls, we'll look at the Promise.allSettled() method and see how it compares to Promise.all().

Getting Ready

Although Promise.allSettled() is relatively new, all modern browsers already support it. But before running the examples in older browsers, you'll want to ensure the browser supports it.[1] In the Node environment, you'll need a minimum Node version of 12.9.0. You can also use a Babel plugin to make use of the feature in older browsers or Node versions.[2]

1. https://caniuse.com/mdn-javascript_builtins_promise_allsettled
2. https://www.npmjs.com/package/babel-plugin-polyfill-es-shims

Executing Multiple Promises

When creating complex JavaScript applications, you'll inevitably encounter circumstances where you need to execute multiple promises. Say you have an async function that retrieves a blog post, like this:

promise.allSettled/tracking_promises_ex01.js

```
async function getPost(id = 1) {
  try {
    return await Utility.loadPost(id);
  } catch (error) {
    // handle error
  }
}
```

This code works great to retrieve a single blog post. But what if you need to retrieve multiple posts? Easy! Create a loop to get the posts you desire:

promise.allSettled/tracking_promises_ex02.js

```
const postIds = ['1', '2', '3', '4'];

postIds.forEach(async id => {
  const post = await getPost(id);

  // process the post
})
```

But there's a problem here: the `await` keyword will pause the loop until it gets a response from `getPost()`. In other words, this code will load the posts sequentially rather than making multiple requests at the same time.

One way to fix this issue is to use the `Promise.all()` method. `Promise.all()` returns a single promise that resolves once all the promises in the iterable have resolved:

promise.allSettled/tracking_promises_ex03.js

```
const postIds = ['1', '2', '3', '4'];

const promises = postIds.map(async (id) => {
  return await getPost(id);
});

const arr = Promise.all(promises);
```

But there's a catch here, too! If one of the promises in the iterable rejects, `Promise.all()` immediately rejects, causing every other post not to load. It's not fair to honest objects who have kept their promise, right?

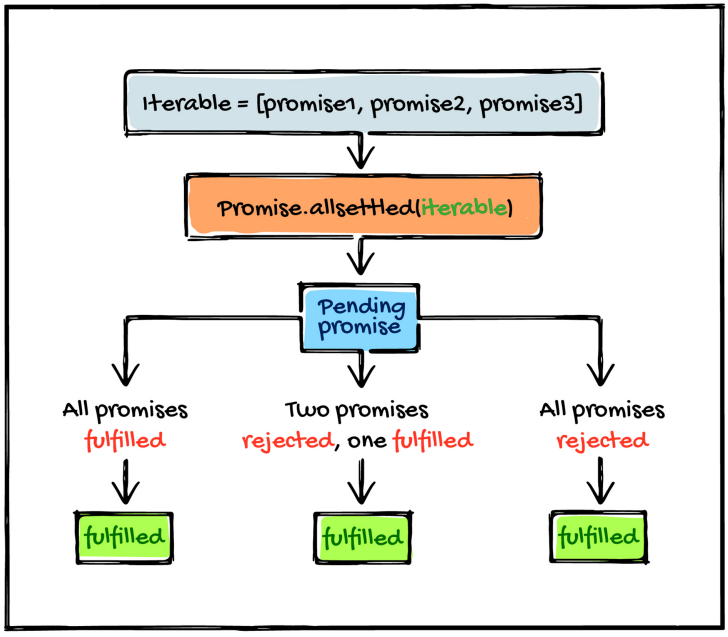
Until fairly recently, JavaScript didn't provide a built-in method to wait for all promises to settle (either fulfilled or rejected). Fortunately, ES2020 is here

to change that by introducing the Promise.allSettled() method. With this method, we can get the result of all promises passed to the method.

Using Promise.allSettled() to Fetch Multiple Resources

The Promise.allSettled() method returns a pending promise that resolves when all of the given promises have either successfully fulfilled or rejected ("settled," in other words). This behavior is very useful to track multiple asynchronous tasks that are not dependent on one another to complete.

The following image shows how the Promise.allSettled() method resolves a pending promise:



In the following example, we attempt to fetch three resources, one of which doesn't exist. Notice how Promise.allSettled() reports the result of every promise:

promise.allSettled/tracking_promises_ex04.js

```
const promises = [
  fetch('https://picsum.photos/200', {mode: "no-cors"}),
  fetch('https://does-not-exist', {mode: "no-cors"}),
  fetch('https://picsum.photos/100/200', {mode: "no-cors"})
];

Promise.allSettled(promises).
  then((results) => results.forEach((result) => console.log(result)));

// logs:
```

```
// => { status: "fulfilled", value: Response }
// => { status: "rejected", reason: TypeError }
// => { status: "fulfilled", value: Response }
```

Rather than immediately rejecting when one of the promises fails, Promise.allSettled() waits until they all have completed.

Notice how the result of all promises is passed as an array to then() and how they are in the same order as the iterable that was given even though they settled out of order. The outcome of each promise has a status property, indicating whether the promise has fulfilled. When a promise is rejected, the result won't have a value property. Instead, it has a reason property containing the rejection reason.

Keep in mind that the promise returned by Promise.allSettled() will almost always be fulfilled. The promise will reject if and only if we pass a value that's not iterable, such as a plain object.

Let's look at the rewritten version of this code, this time with the Promise.all() method:

promise.allSettled/tracking_promises_ex05.js

```
const promises = [
  fetch('https://picsum.photos/200', {mode: "no-cors"}),
  fetch('https://does-not-exist', {mode: "no-cors"}),
  fetch('https://picsum.photos/100/200', {mode: "no-cors"})
];

Promise.all(promises).
  then((results) => results.forEach((result) => console.log(result)));

// logs:
// => Uncaught (in promise) TypeError: Failed to fetch
```

This time, the promise rejects immediately upon the second input promise rejecting. One important difference between these two methods is that Promise.allSettled() has an extra property that Promise.all() doesn't: status. In fact, Promise.all() returns the raw value that Promise.allSettled() tucks into its resulting object. Compare:

promise.allSettled/tracking_promises_ex06.js

```
const promises = [
  Promise.resolve(1),
  Promise.resolve(2)
];

Promise.allSettled(promises).
  then((results) => results.forEach((result) => console.log(result)));

// logs:
// => { status: "fulfilled", value: 1 }
```

```
// => { status: "fulfilled", value: 2 }

Promise.all(promises).
  then((results) => results.forEach((result) => console.log(result)));

// logs:
// => 1
// => 2
```

Notice how Promise.all() directly returns the response. If you're in an old JavaScript environment that doesn't support Promise.allSettled() or if you'd like to directly return the promises, there's a simple workaround for you. Consider the following code:

promise.allSettled/tracking_promises_ex07.js

```
const promises = [
  fetch('https://picsum.photos/200', {mode: "no-cors"}),
  fetch('https://does-not-exist', {mode: "no-cors"}),
  fetch('https://picsum.photos/100/200', {mode: "no-cors"})
].map(p => p.catch(e => e));

Promise.all(promises).
  then((results) => results.forEach((result) => console.log(result)));
```

Here, we've applied the map() method to an iterable of promises. Within the method, we use catch() to return promises that resolve with an error value. This way, we can simulate the behavior of Promise.allSettled() while being able to directly access the result of promises.

Often, we use Promise.all() and Promise.allSettled() with similar types of requests, but there's no written rule that we should. You may find yourself in a situation where you need to read a local file, retrieve a JSON document from a web API, and load an XML document from another API. Once you obtain data from all three async requests, you want to process them. Promise.all() and Promise.allSettled() are ideal for such scenarios.

Keep in mind that you will want to use these methods only when you need to process the result of multiple async requests together. If it's possible to process the result of each async request individually, then handle each promise with its own then() handler. This way, you can execute your code as soon as each promise is resolved.

Wrapping Up

In this chapter, we looked at potential pitfalls when executing multiple promises at the same time. We learned why looping over asynchronous tasks could be a bad idea because it will cause the promises to run sequentially.

Then we learned about the Promise.allSettled() method and compared it to Promise.all().

While Promise.all() is very strict in its execution policy, Promise.allSettled() is forgiving. That doesn't mean Promise.allSettled() is superior to Promise.all(): they complement each other. Using Promise.all() is more appropriate when you have essential async tasks that are dependent on each other. On the other hand, Promise.allSettled() is more suitable for async tasks that might fail but are not essential for your program to function.

As of ES2021, the ECMAScript standard includes one more method for the promise object: Promise.any(). This method is the opposite of Promise.all(). In the next chapter, we're going to learn how Promise.any() can help you when you need to focus on the promise that resolves first.

CHAPTER 4

Improving Reliability and Performance

Nobody likes slow applications. As a programmer, you want to always strive to build apps that provide a snappy user experience. But what's more important is designing apps that are able to recover quickly from difficult conditions. When working with web APIs, for instance, you need to be prepared for server downtime. How would your application react if it couldn't reach a server?

Building apps that are responsive and reliable should be at the top of the list of every developer. ES2021 `Promise.any()` is a recent addition to ECMAScript that helps us achieve both of these goals at the same time. We can protect our app from potential API downtimes by making network requests to multiple APIs asynchronously and using the result of the one that's accessible. What's more, we can improve the performance of critical application services by using the API that responds first.

We'll start this chapter by looking at how `Promise.any()` works. Once you get the hang of it, the next step is to learn how to make our app resilient against API issues and enhance its performance.

Getting Ready

The `Promise.any()` method is a newcomer to the ECMAScript standard, so before running the examples, you'll want to ensure your browser supports it.[1] In the Node environment, you'll need a minimum Node version of 15.0.0. You can also use a Babel plugin to make use of the feature in older browsers or Node versions.[2]

1. https://caniuse.com/mdn-javascript_builtins_promise_any
2. https://www.npmjs.com/package/babel-plugin-polyfill-es-shims

Using the Promise.any() Method

Promise.any() returns a pending promise that resolves asynchronously as soon as one of the promises in the given iterable fulfills. All right, let's execute the following code and see what happens:

promise.any/promise.any_ex01.js

```
const promises = [
  Promise.reject(new Error('failure #1')),
  Promise.reject(new Error('failure #2')),
  Promise.resolve('YES!')
];

Promise.any(promises).
  then((result) => console.log(result));

// logs:
// => YES!
```

Here we have an array of three promise objects passed to Promise.any(). Because Promise.resolve() returns a promise that's already fulfilled, the promise returned by Promise.any() is immediately fulfilled with the given value.

But what happens if all input promises reject?

promise.any/promise.any_ex02.js

```
const promises = [
  Promise.reject(new Error('failure #1')),
  Promise.reject(new Error('failure #2')),
  Promise.reject(new Error('failure #3'))
];

Promise.any(promises).then(
  (result) => {console.log(result)},
  (error) => {console.error(error)}
);

// logs:
// => AggregateError: No Promise in Promise.any was resolved
```

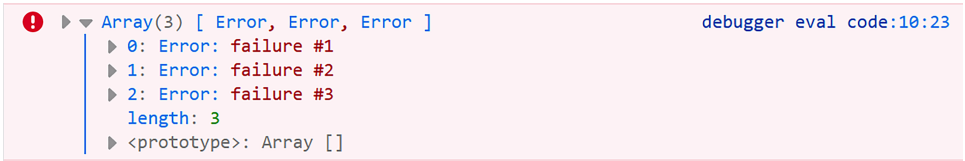
The AggregateError object wraps all the rejection reasons of all the input promises in a single error, and we can read them using the errors property. Edit line 9 of the preceding code and replace console.error(error) with console.error(error.errors), like this:

promise.any/promise.any_ex03.js

```
const promises = [
  Promise.reject(new Error('failure #1')),
  Promise.reject(new Error('failure #2')),
  Promise.reject(new Error('failure #3'))
];
```

```
Promise.any(promises).then(
  (result) => {console.log(result)},
  (error) => {console.error(error.errors)}
);
```

You should see a message like this in your browser's console showing all the rejection reasons we've specified in the input promises:



Only one other case causes Promise.any() to reject: passing an iterable that's empty. Here's an example:

promise.any/promise.any_ex04.js

```
Promise.any([]).then(
  (result) => {console.log(result)},
  (error) => {console.log(error)}
);

// logs:
// => AggregateError: No Promise in Promise.any was resolved
```

Notice the empty array at line 1 that causes Promise.any() to reject. The error message is exactly the same as when all promises reject, which is something to be wary of because you don't want to misunderstand the reason for the failure.

Now that you know how the Promise.any() method works, it's time to look at some real-world examples. Up next, we're going to learn how to take advantage of this method to make our app more resilient.

Avoiding the Single Point of Failure

A *single point of failure* (SPOF) is a component of a system that with just one malfunction or fault will stop the entire system from working. If you want to have a reliable application, you should be able to identify and avoid potential SPOFs in the system.

A common SPOF in web applications occurs when fetching critical resources, such as data for financial markets, from external APIs. If the API is inaccessible, the app will stop working. The Promise.any() method is extremely useful in this regard. It enables us to request data from multiple APIs and use the result of the first successful promise.

Let's look at an example. In the following code, we have an array containing the URL of two APIs, both of which return the same info. Promise.any() will attempt to fetch the two URLs at the same time; so as long as one of the APIs is available, the code works fine:

promise.any/promise.any_ex05.js

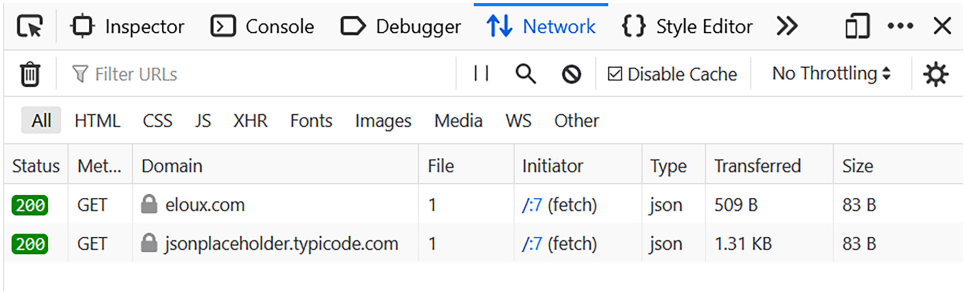
```
const apis = [
  'https://eloux.com/todos/1',
  'https://jsonplaceholder.typicode.com/todos/1'
];

async function fetchData(api) {
  const response = await fetch(api);
  if (response.ok) {
    return response.json();
  } else {
    return Promise.reject(new Error('Request failed'));
  }
}

function getData() {
  return Promise.any([
    fetchData(apis[0]),
    fetchData(apis[1])
  ]);
}

getData().then((response) => console.log(response.title));
```

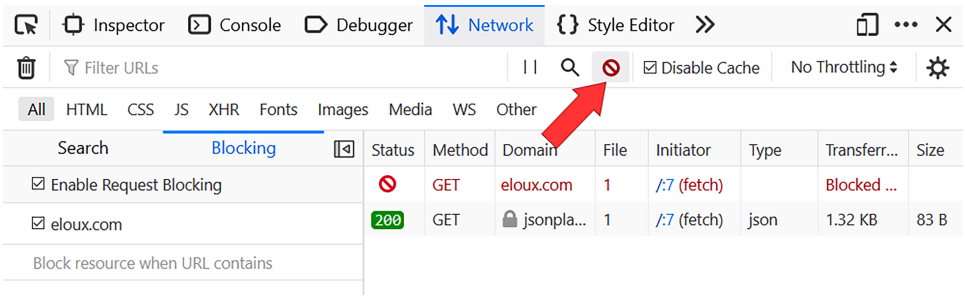
Now, to see what actually happens behind the scenes, check the network tab of your browser's console. Press F12 on your keyboard to open developer tools and then navigate to the network. The following image shows the Network tab of Mozilla Firefox:



You can see that the status of both requests is 200, which means they are successfully fulfilled. But remember, Promise.any() only uses the result of the first promise that fulfills, so the other result is ignored.

To see what happens if one of the APIs fails, we're going to block one of the requests. If you're using Firefox, click on the icon that says Request Blocking

and enter eloux.com in the given field and press enter. This step will block any request to eloux.com, allowing you to simulate that the API is inaccessible, as shown in the image that follows:



If you refresh the page, you'll see that the request to eloux.com is unsuccessful. But since the other API is available, our code works fine. Now, what happens if both APIs are unavailable? Go ahead and enter jsonplaceholder.typicode.com, the URL of the other API, to block it. Switch back to the Console tab and run the code again. You should see an error message like this:

```
Uncaught (in promise) AggregateError: No Promise in Promise.any was resolved
```

We're getting an AggregateError that denotes all promises passed to Promise.any() are rejected. But this error message isn't the best way to handle a rejected promise. Let's chain a catch() method and output a custom error message:

promise.any/promise.any_ex06.js

```
const apis = [
  'https://eloux.com/todos/1',
  'https://jsonplaceholder.typicode.com/todos/1'
];

async function fetchData(api) {
  const response = await fetch(api);
  if (response.ok) {
    return response.json();
  } else {
    return Promise.reject(new Error('Request failed'));
  }
}

function getData() {
  return Promise.any([
    fetchData(apis[0]),
    fetchData(apis[1])
  ]).catch(() => {
    return Promise.reject(
      new Error('Unable to access the API')
    );
  });
```

```
}

getData().then(
  (response) => console.log(response.title),
  (error) => console.error(error)
);
```

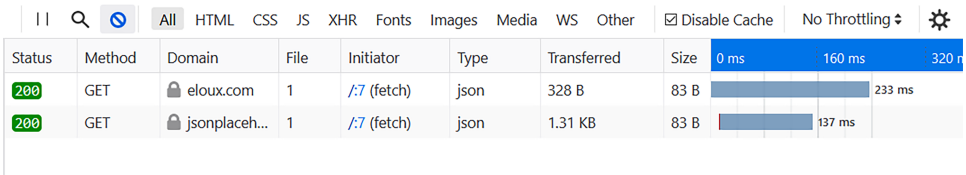
Here, we're using the catch() method to take over error handling. We return a promise that is rejected and give a reason as to why. Now run the code again. You should see our custom error message logged to the console, as shown in the screenshot that follows:

So far we've learned how to execute multiple promises at the same time to make our app more reliable. But this approach has one more benefit: performance.

Improving the Performance of Your App

As a programmer, it's always in your best interest to build apps that respond quickly to user requests. Promise.any() allows you to improve the performance of critical app services by using the data from the API that responds first. As we saw in Avoiding the Single Point of Failure, on page 29, if more than one promise is fulfilled, we'll get back the first fulfillment value. The other fulfillment values are ignored.

Open the network tab again and check the timeline column. It provides the amount of time it took (in milliseconds) to get a response from each API. Remember you turned on request blocking in the last example to simulate the failure of the requests—turn request blocking off for this example. Also, mark the Disable Cache checkbox so that the browser retrieves a fresh copy of each file rather than loading them from the cache. Now run the code from the last example again. You should see a result like this:



So if we only used the first API, it would take our app at least 233 milliseconds to retrieve and handle the data. With the second API added, it's now 137 ms, which is about 40 percent faster. This is a nice improvement in load time. Obviously, you'll see different results depending on your distance from the servers and internet connection speed.

So in addition to avoiding the single point of failure issue, you can use Promise.any() to improve the performance of your application—and that's a win-win.

Wrapping Up

Promise.any() is a valuable addition to the Promise object. This method allows us to execute multiple promises asynchronously and use the result of the one that's accessible to respond faster. Make use of the Promise.any() method in your applications to avoid the dreaded SPOF and you'll potentially see some welcome performance improvement too. Faster applications equal happier users, which is the promise of Promise.any().

So far, we've talked about Promise.any(), Promise.allSettled(), and Promise.all(). One more method that you should know about is Promise.race(). This method comes in handy when you need to set a time limit for computationally expensive tasks.

CHAPTER 5

Setting a Time Limit for Async Tasks

Internet users are impatient. When working with asynchronous network requests, how long should we keep them waiting until we get a response from a server and settle a promise? If the app takes longer than a few seconds to respond, most users will leave it and find somewhere else to go. The amount of time it takes for an API to respond is usually outside the control of our code due to things like server overload. We need a way to set a time limit to avoid entering a state of prolonged or endless waiting.

Fortunately, the `Promise` object comes with a method called `Promise.race()` that we can take advantage of to enforce a time limit for the latency of async tasks. This method is designed to race several promises against each other and return the result of the promise that settles first. By supplying it with a promise that's going to be rejected after an allotted time, we can define how long the code should wait to get a response from a server.

We'll start this chapter by comparing `Promise.race()` to a method we already know: `Promise.any()`. Once we get the hang of how it works, we'll use it to concurrently fetch data from an external API and execute a timer that logs an error after two seconds. We then revise the code to replace the error message with a function that pulls data from a cache to use in case the API is inaccessible or doesn't respond quickly enough.

Getting Ready

The `Promise.race()` method has been around for a while, so most modern browsers already support it.[1] Node.js support for `Promise.race()` dates back to as early as

1. https://caniuse.com/mdn-javascript_builtins_promise_race

version 0.12.0. But if you want to make use of the feature in even older Node versions or browsers you can use a Babel plugin.[2]

Comparing Promise.race() to Promise.any()

Recall from Using the Promise.any() Method, on page 28, that Promise.any() uses the value of the first promise that fulfills. Promise.race() behaves exactly the same as far as promise fulfillment. However, when it come to rejection, Promise.race() is completely different: it settles as soon as one of the given promises rejects. In other words, while Promise.any() rejects if all of the given promises reject, Promise.race() rejects if the first promise that settles is rejected.

To demonstrate, let's create two promises. Promise A will reject after one second and promise B will succeed after two seconds:

latency/latency_ex01.js

```
const promiseA = new Promise((resolve, reject) => {
  setTimeout(reject, 1000, 'A');
});

const promiseB = new Promise((resolve) => {
  setTimeout(resolve, 2000, 'B');
});

Promise.race([
  promiseA,
  promiseB
]).then((response) => {
  console.log(response);
}).catch((error) => {
  console.error(error);  // => A
});

Promise.any([
  promiseA,
  promiseB
]).then((response) => {
   console.log(response);  // => B
}).catch((error) => {
  console.error(error);
});
```

Run this code in your browser's console. You'll be passing the same array of promises to both methods. But notice that Promise.race() rejects after one second and logs A to the console, while Promise.any() succeeds after two seconds and logs B.

2. https://www.npmjs.com/package/babel-plugin-polyfill-es-shims

Another difference between the two methods is that passing an empty array (or any other empty iterable) to Promise.race() results in a promise that remains in pending state:

latency/latency_ex02.js

```
Promise.race("").then((response) => {
  // this will never be executed
}).catch((error) => {
  // neither this one
});
```

If we pass Promise.race() an iterable containing nothing, then the first of nothing can never be determined. So if the returned promise is stuck in the pending state, the first thing to check is the iterable we're passing to the method.

Okay, now that we know how Promise.race() works, let's go ahead and use it in a more practical way.

Enforcing a Time Limit for Async Tasks

The Promise.race() method can be useful when fetching an external resource that may take a while to complete. With this method, we can race an async task against a promise that's going to be rejected after a number of milliseconds. Depending on the promise that settles first, we either obtain the result or report an error message.

Let's create a function that attempts to pull data from an API and reports an error if the server doesn't respond quickly enough. We'll represent each outcome by a promise and use Promise.race() to select the first accessible result:

latency/latency_ex03.js

```
function fetchData() {
  const timeOut = 2000;    // two seconds
  const data = fetch('https://jsonplaceholder.typicode.com/todos/1');
  const failure = new Promise((resolve, reject) => {
    setTimeout(() => {
      reject(new Error(`Failed to retrieve data after ${timeOut} milliseconds`));
    }, timeOut);
  });
  return Promise.race([data, failure]);
}

fetchData().then((response) => {
  console.log(response);
}).catch((error) => {
  console.error(error);
});
```

Here, we've set the timeout to two seconds, which is usually more than enough to receive a response from an API. To see the timeout message, go ahead and set a one-millisecond timeout and run the code again. Sure enough we can see the timeout message:

```
Error: Failed to retrieve data after 1 milliseconds        debugger eval code:15:11
```

This code works fine but simply logging an error doesn't offer the best user experience. We can further improve this code by using cached data if fresh data isn't available in allotted time. Let's revise the code a little:

latency/latency_ex04.js

```
function loadFromCache() {
  const data = {
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
  };
  return new Promise((resolve) => {
    resolve(data);
  })
}

function fetchData() {
  const timeOut = 2000;    // two seconds
  const cache = loadFromCache().then((data) => {
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve(data);
      }, timeOut);
    });
  });
  const freshData = fetch('https://jsonplaceholder.typicode.com/todos/1');
  return Promise.race([freshData, cache]);
}

fetchData().then((response) => {
  console.log(response);
}).catch((error) => {
  console.error(error);
});
```

In this version of the code, we concurrently fetch data from an external API and pull data from a cache to use in case the API is inaccessible or doesn't respond quickly enough. For the sake of simplicity, the loadFromCache() function returns a predefined object, but in a real-world app you'll probably load the cached data from a database.

Of course, using cached data would work only for certain types of information that doesn't change frequently. If you're retrieving data like stock prices or exchange rates, then using Promise.any() is a better choice (see Avoiding the Single Point of Failure, on page 29) since it allows you to request data from multiple APIs and use the result of the one that's accessible.

An interesting use case for Promise.race() is to batch async requests. As explained by Chris Jensen,[3] if you have to make a large number of async requests and don't want the pending requests to get out of hand, you can use Promise.race() "to keep a fixed number of parallel promises running and add one to replace whenever one completes." Using Promise.race() in this way lets you run multiple jobs in a batched way while preventing too much work from happening at one time.

You can also apply Promise.race() to a computationally expensive background task. It's easy to imagine cases where some task might be attempted in the background, such as rendering a complex canvas while the user is occupied with something else. Using Promise.race() there, again gives you some knowable time to work with—and the opportunity to introduce some logic of what to do should the task fail.

Wrapping Up

You can't control how long it takes for a server to respond to requests, but that shouldn't stop you from designing apps that are responsive to user requests. By taking advantage of the Promise.race() method, you can set a timeout for async requests and react if they take too long to complete.

In the next chapter, you're going to learn about canceling pending async requests using the AbortController API.

3. https://stackoverflow.com/a/48820037

CHAPTER 6

Canceling Pending Async Requests

Applications today must work with information on remote servers, and the Fetch API allows you to easily retrieve resources asynchronously across the network. But sometimes you may want to cancel a pending async request before it has completed. Perhaps you have a network-intensive application and async requests are taking too long to fulfill, or maybe the user clicked a Cancel button.

The AbortController API provides a generic interface that allows you to cancel a fetch request. The cornerstone of the API is the AbortController interface, which provides an abort() method. You can create a cancelable fetch request by passing the signal property of AbortController as an option to fetch(). Later, when you want to abort the fetch, simply call the abort() method to terminate the network transmission.

We'll start this chapter by setting a time limit for a fetch request. This example should give you a clear idea of how to implement an AbortController. Then we'll move on to more advanced topics like deregistering multiple event listeners in one statement and creating an abort button to let users cancel async requests.

Getting Ready

All modern browsers support the AbortController API.[1] In the Node environment, you'll need a minimum Node version of 15.0.0. To run your program in older browsers or Node versions, you can use a Babel plugin.[2]

1. https://caniuse.com/abortcontroller
2. https://www.npmjs.com/package/yet-another-abortcontroller-polyfill

Canceling Async Tasks After a Period of Time

Remember Enforcing a Time Limit for Async Tasks, on page 37, where we used Promise.race() to concurrently fetch data from an external API and execute a timer? With the AbortController API, we can achieve a similar outcome: cancel a request that we have already issued but don't want to wait for the operation to finish.

Promise.race() vs. AbortController

When you only want to automatically cancel a fetch request after a specified amount of time, you can use either Promise.race() or AbortController. It's a matter of preference. AbortController is easier to use if you want to cancel an async request once a specific DOM event is fired, such as when the user clicks a cancel button. Promise.race(), on the other hand, works better if you want to concurrently fetch data and perform another task, such as pulling backup from a database.

Consider the following example. When we run this code, the setTimeout() method sets a two-second timer to execute abort(). If the fetch is complete in the allotted time, the abort will have no effect. If not, an error is thrown, as follows:

abort/abort_ex01.js

```
const controller = new AbortController();
const signal = controller.signal;

fetch('https://eloux.com/todos/1', {signal})
  .then(response => {
    return response.json();
  }).then(response => {
    console.log(response);
  });

setTimeout(() => controller.abort(), 2000);
```

To be able to abort a fetch() request, we first need to create an instance of AbortController (line 1). It's essential to have an AbortController before initiating the fetch request, so we execute this statement first.

Next, we obtain a signal object that allows us to exchange information with the fetch() method:

```
const signal = controller.signal;
```

We then pass this signal object as an option to fetch():

```
fetch('https://eloux.com/todos/1', {signal});
```

Here, we're connecting signal and controller with the request. This connection enables us to abort the request by calling the abort() method of controller:

```
controller.abort();
```

abort() is the only method of controller and will cause the promise object returned by fetch to reject with an exception, like this:

```
▶ Uncaught (in promise) DOMException: The operation was aborted.   debugger eval code:5
```

At this point, the control will be passed to the catch() method (if it exists). Upon calling abort(), the API will notify the signal, which if you want, you can listen to by attaching an event handler:

```
signal.addEventListener('abort', () => {
  console.log(signal.aborted);
});

// logs:
// => true
```

Notice how after aborting succeeds, the aborted property of signal has a value of true.

Aborting a request might take only a few lines of code, but if you have to do it a dozen times, you're making your app unnecessarily hefty. So why not turn the code into a function?

abort/abort_ex03.js

```
function fetchWithTimeout(url, settings, timeout) {
  // If the timeout argument doesn't exists
  if (timeout === undefined) {
    return fetch(url, settings);
  }

  // if timeout isn't an integer, throw an error
  if (!Number.isInteger(timeout)) {
    throw new TypeError('The third argument is not an integer')
  }

  const controller = new AbortController();
  setTimeout(() => controller.abort(), timeout);
  settings.signal = controller.signal;
  return fetch(url, settings);
}
```

This function works like a fetch() method but provides the option of setting a timeout. If we pass an integer (in milliseconds) as the third argument, the request will abort after the time expires. If not, it will retrieve the resource like a regular fetch().

A server might take a longer time than expected to respond to requests, or might not respond at all. It's always a good idea to set a time limit for async requests to avoid keeping your users waiting. But how do you distinguish a fetch request that's intentionally aborted from the ones that are terminated because of an error? We'll explore that question next.

Handling an Aborted Request

When abort() successfully cancels a request, the pending promise rejects with a DOMException error. But you don't want to show the default error message if the operation is canceled by the user. After all, it's not considered an error if the cancelation is intentional.

So let's add a catch() method to the promise chain to handle the rejection:

abort/abort_ex04.js

```
const src = 'https://eloux.com/todos/1';
const controller = new AbortController();
const signal = controller.signal;

fetch(src, {signal})
  .then(response => {
    return response.json();
  })
  .then(json => {
    console.log(json);
  })
  .catch(error => {
    if (error.name === 'AbortError') {
      console.log('Request successfully canceled');
    } else {
      console.error('Fetch failed!', error);
    }
  });

controller.abort();

// logs:
// => Request successfully cancelled
```

To ensure we're handling the abort error separately, we can check the name property of the error. If it has a value of AbortError, we know it's thrown by the AbortController API. If not, then we handle it like any other error.

Fetch requests aren't the only type of request that can be canceled with the AbortController API. We can cancel event listeners too!

Removing Multiple Event Listeners

In client-side JavaScript programming, the flow of the code is determined by events. Whenever something happens to the page or web browser, the browser emits an event, such as when the user clicks a link or a given resource has loaded. If we want to do something when a particular event occurs, we can register one or more functions to be called using the addEventListener() method.

We can later remove an event handler function from an object using the removeEventListener() method. If we register dozens of event handlers, we'll need the exact same number of removeEventListener() methods to deregister them, which unnecessarily bloats the code. With AbortController we can deregister multiple event listeners in only one statement.

Let's look at a simple example. Say we want to temporarily give an element some special effect. We want to change the text of the element when the user's pointing device (usually a mouse or trackpad) moves the cursor onto the element and reverse the effect when it moves past the element. We also want to change the background color of the element when the device is pressed (mousedown) or released (mouseup). Here's how we can implement this effect using the old approach:

abort/abort_ex05.html

```
<!doctype html>
<html lang="en-us">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script src="abort_ex05.js" defer></script>
</head>

<body>
  <div class="container">Mouse over me!</div>
</body>

</html>
```

abort/abort_ex05.js

```
const container = document.querySelector('.container');

function sayHello() {
  container.textContent = 'Hello';
}

function sayBye() {
  container.textContent = 'Bye!';
}
```

```
function depress() {
  container.style.backgroundColor = 'aqua';
}

function release() {
  container.style.backgroundColor = 'transparent';
}

container.addEventListener('mouseenter', sayHello);
container.addEventListener('mouseout', sayBye);
container.addEventListener('mousedown', depress);
container.addEventListener('mouseup', release);
```

Here, we're attaching four event handlers to an element with a class of container. Now if we want to stop the effect, we'll have to remove each handler one by one. Note that the arguments must be the same—the type of event and the function of the event handler.

abort/abort_ex06.js

```
container.removeEventListener('mouseenter', sayHello);
container.removeEventListener('mouseout', sayBye);
container.removeEventListener('mousedown', depress);
container.removeEventListener('mouseup', release);
```

But it's possible to use the AbortController API to achieve the same result without having to remove each handler separately. The addEventListener() method now accepts an abort signal as the third argument. Create a controller object and pass its signal property to addEventListener(), like this:

abort/abort_ex07.js

```
const container = document.querySelector('.container');
const controller = new AbortController();
const signal = controller.signal;

function sayHello() {
  container.textContent = 'Hello';
}

function sayBye() {
  container.textContent = 'Bye!';
}

function depress() {
  container.style.backgroundColor = 'aqua';
}

function release() {
  container.style.backgroundColor = 'transparent';
}

container.addEventListener('mouseenter', sayHello, {signal});
container.addEventListener('mouseout', sayBye, {signal});
container.addEventListener('mousedown', depress, {signal});
```

```
container.addEventListener('mouseup', release, {signal});
```

Now, we can abort all addEventListener() methods with a single AbortSignal:

```
controller.abort();
```

Removing each event handler separately might not be a big issue when we have a few event handlers, but it becomes unpleasant if we have dozens. For example, suppose we have a long list of elements, and we want to enable the user to sort the list by dragging and dropping the elements.

For each element, there's an event attached to check for the state of dragging. We want to disable drag and drop once the user clicks Save. Using AbortController can be a time-saver in this situation: rather than removing each event handler separately, we can call abort() to remove them all.

So far, we've been canceling requests programmatically, but what if you want to give users the option to cancel requests themselves, like when they decide not to wait for a download?

Making a User-Cancelable Async Request

When including large files on your page, you should take into account the fact that some users will be on limited bandwidth or mobile devices with expensive data plans. Therefore, the ability for a user to load and cancel loading large items is valuable.

Say you need to load a very large photo (in this case, 22 MB in size) from Wikipedia. You want to define a button that fetches the photo and another button that aborts the loading. Here's how the program will look:

You can see a live example of this program here:

https://eloux.com/async_js/examples/abort_ex08_complete.html

First, define an HTML <image> element on the page. The src attribute of this element will be filled once the image is loaded. We also need an element to inform the user about the outcome, so define a element with a class of result. Next, create the buttons. We're going to disable the abort button until the load button is clicked, so give it a disabled attribute:

abort/abort_ex08.html

```
<!doctype html>
<html lang="en-us">

<head>
  <meta charset="utf-8">
  <title>Making a User Cancelable Async Request</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script src="abort_ex08.js" defer></script>
</head>

<body>
  <image class="image">
  <span class="result"></span>
  <button class="loadBtn">Load Photo</button>
  <button class="abortBtn" disabled="disabled">Cancel Loading</button>
</body>

</html>
```

Now, in the JavaScript file, we need to set up two functions: one to call when the Load Photo button is clicked and the other to call when the Cancel Loading button is clicked:

abort/abort_ex08.js

```
Line 1 // create a reference to each HTML element
- const loadBtn = document.querySelector('.loadBtn');
- const abortBtn = document.querySelector('.abortBtn');
- const image = document.querySelector('.image');
5 const result = document.querySelector('.result');
-
- const controller = new AbortController();
-
- // abort the request
10 abortBtn.addEventListener('click', () => controller.abort());
-
- // load the image
- loadBtn.addEventListener('click', async () => {
-   loadBtn.disabled = true;
15   abortBtn.disabled = false;
-
-   result.textContent = 'Loading...';
-
-   try {
20     const response = await fetch(`https://upload.wikimedia.org/wikipedia/com
- mons/a/a3/Kayakistas_en_Glaciar_Grey.jpg`, {signal: controller.signal});
-     const blob = await response.blob();
-     image.src = URL.createObjectURL(blob);
-
25     // remove the "Loading.." text
-     result.textContent = '';
-   }
-   catch (err) {
-     if (err.name === 'AbortError') {
30       result.textContent = 'Request successfully canceled';
-     } else {
-       result.textContent = 'An error occurred!'
-       console.error(err);
-     }
35   }
-
-   loadBtn.disabled = false;
-   abortBtn.disabled = true;
- });
```

Notice how line 13 of the code registers an async function to be called when the Load Photo button is clicked. Within the function, we disable the Load button to prevent another click and enable the Cancel Loading button. Next we attempt to retrieve the image using the standard fetch() function.

To be able to display the image we've retrieved, we need to convert it into an object URL. First use the Blob() constructor to get a Blob object (line 22). Now you can create a URL that refers to the Blob by passing the object into the URL.createObjectURL() method (line 23). All that's left to do to display the image is insert the resulting data into the src attribute of the image tag. At the end of the code, we revert the buttons to their original state.

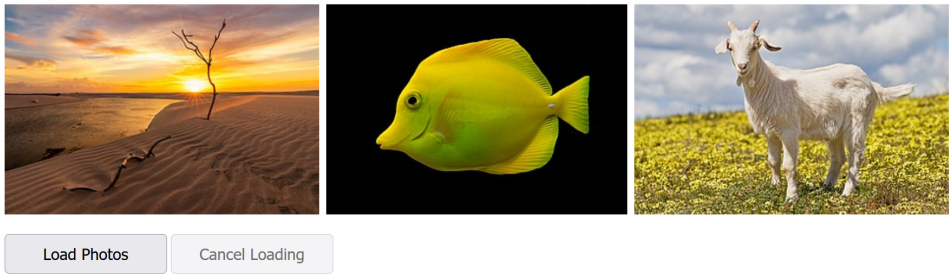
What's a Blob?

Blob stands for *binary large object*, which is a data type containing a collection of binary data. In JavaScript, Blob serves as an essential data interchange method for several APIs. They're often used when working with data that isn't in a JavaScript-native format, such as images, audio, or other multimedia objects.

Now, what if we need to fetch multiple images and want to let the user abort them all at the same time?

Aborting Multiple Fetch Requests with One Signal

Just as we can abort multiple addEventListener() methods, we can abort multiple fetch requests with a single AbortSignal. Let's revise the previous example to fetch an array of images rather than a single image. The following image shows how our program will look:



Here's a live example:

https://eloux.com/async_js/examples/abort_ex09_complete.html

This time, we'll first construct a new array of fetch requests out of the image URLs using the map() method (line 25). Then we execute them all by passing the array to Promise.all(). For each image that's loaded, we convert it into a data URL, create an image element, and insert it into the page.

abort/abort_ex09.html

```
<!doctype html>
<html lang="en-us">
```

```
<head>
  <meta charset="utf-8">
  <title>Aborting Multiple Fetch Requests With One Signal</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script src="abort_ex09.js" defer></script>
</head>

<body>
  <div class="gallery"></div>
  <span class="result"></span>
  <button class="loadBtn">Load Photos</button>
  <button class="abortBtn" disabled="disabled">Cancel Loading</button>
</body>

</html>
```

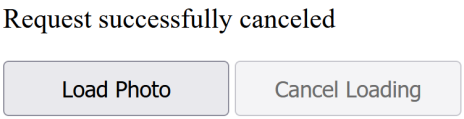
abort/abort_ex09.js

```
Line 1  const loadBtn = document.querySelector('.loadBtn');
-       const abortBtn = document.querySelector('.abortBtn');
-       const gallery = document.querySelector('.gallery');
-       const result = document.querySelector('.result');
5
-       const controller = new AbortController();
-
-       const urls = [
-         `https://upload.wikimedia.org/wikipedia/commons/thumb/7/70/Por_do_Sol_em
10      _Baixa_Grande.jpg/320px-Por_do_Sol_em_Baixa_Grande.jpg`,
-         `https://upload.wikimedia.org/wikipedia/commons/thumb/5/5e/Zebrasoma_fla
-       vescens_Luc_Viatour.jpg/320px-Zebrasoma_flavescens_Luc_Viatour.jpg`,
-         `https://upload.wikimedia.org/wikipedia/commons/thumb/f/ff/Domestic_goat
-       _kid_in_capeweed.jpg/320px-Domestic_goat_kid_in_capeweed.jpg`
15      ];
-
-       abortBtn.addEventListener('click', () => controller.abort());
-
-       loadBtn.addEventListener('click', async () => {
20        loadBtn.disabled = true;
-         abortBtn.disabled = false;
-
-         result.textContent = 'Loading...';
-
25        const tasks = urls.map(url => fetch(url, {signal: controller.signal}));
-
-         try {
-           const response = await Promise.all(tasks);
-           response.forEach(async (r) => {
30           const img = document.createElement('img');
-            const blob = await r.blob();
-            img.src = URL.createObjectURL(blob);
-            gallery.appendChild(img);
-          });
35        result.textContent = '';
-       } catch (err) {
```

```
    if (err.name === 'AbortError') {
      result.textContent = 'Request successfully canceled';
    } else {
      result.textContent = 'An error occurred!'
      console.error(err);
    }
  }

  loadBtn.disabled = false;
  abortBtn.disabled = true;
});
```

Now if we press the Cancel Loading button while the requests are in progress, it aborts every fetch and throws an error. In the `catch` block, we intercept the error and insert a custom message into the page informing the user that cancelation was successful:



Note that `Promise.allSettled()` wouldn't be suitable for this task. `Promise.allSettled()` is designed to wait for all promises to settle, so it doesn't make sense to use this method when you need to abort requests before they're completed.

Another use case for `abort()` could be live search: when the user types a character in the input, it triggers a search request; when that promise resolves, you want to show the search results. But if the user presses multiple keys, the first search might resolve before the last. Aborting the "stale" request ensures that the search results reflect the most recent query.

Wrapping Up

An interesting aspect of the AbortController API is that it's provided by the DOM standard and designed to be generic. That means soon we'll see it adopted by other standards and libraries as well, lowering the learning curve for developers who want to use those platforms.

Make use of the AbortController API in your programs to cancel async requests that are no longer needed or taking too long to complete. You can do that by calling `abort()` directly, setting a timer to call `abort()`, or providing a cancel button for users to abort requests whenever they want. You can even use an `AbortController` to deregister an event listener, or multiple event listeners, which is an ability that JavaScript previously lacked.

The final chapter of this book is about the top-level `await`. In modern JavaScript programming, it's a common practice to separate the functionality of a program into independent modules. Modular programming provides several benefits like the ability to use existing assets in other programs. It also makes testing and debugging easier because when we need to fix a specific function, we only have to do it in one module.

Top-level `await` is a feature of modules that improves on the regular `await` keyword to enable developers to access the result of a promise from another module without having to use a workaround. In the next chapter, we're going to cover this feature.

CHAPTER 7

Accessing Promise Results from Another Module

Often in modular programs, you need the result from another module before you run the main module. Consider a recipe suggester (recipe.js) that checks a list of on-hand ingredients asynchronously (pantry.js). Code for recipe.js should not execute until code for pantry.js has run. With top-level await we could tell recipe.js to halt its execution until pantry.js is fully executed. If you used the standard await keyword in an async function, recipe.js would attempt to access the result of pantry.js before it was ready.

A number of workarounds could be used, but they make the code more complicated. Top-level await is an addition to the language that provides a straightforward way to use the await keyword outside of async functions so that we can perform async tasks directly at the top level of the module. Now, modules can act as a big async function and importing modules will wait for the child modules to fully execute before they themselves start executing.

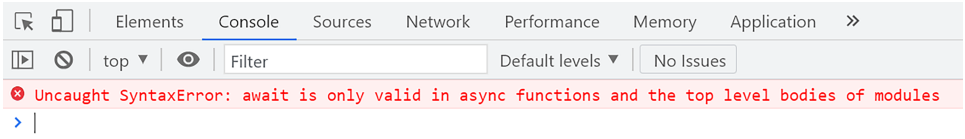
In this chapter, we'll find out how top-level await works and how to make use of it to initialize resources, define dependency paths dynamically, and load dependencies with a fallback implementation.

Getting Ready

Before using top-level await, make sure your browser or Node version supports it.[1] You'll need a minimum Node version of 14.8.0. You can also use a Babel plugin to make use of the feature in older browsers or Node versions.[2]

1. https://caniuse.com/mdn-javascript_operators_await_top_level
2. https://babeljs.io/docs/en/babel-plugin-syntax-top-level-await

Keep in mind you can't use top-level await in classic scripts—it only works in module scripts and browser dev tools. If you get a SyntaxError like the following, that means you're not using it in a module:



Adding a module to an HTML file is the same as adding a regular script except that you should add a type attribute with the value of module, like this:

```
<script type="module" src="module1.js"></script>
```

Also, modules are subject to same-origin policy, meaning that you can't import them from the file system. You'll need to run the code examples in this chapter on a server.

Using Top-Level await

When the await keyword was first introduced, it wasn't possible to use it outside of async functions. You've probably encountered the fatal syntax error await is only valid in async function, too, when attempting to use it for the first time:

await/Await_ex01.js

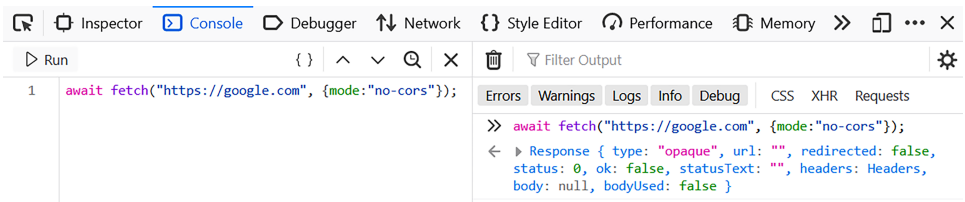
```
const res = await fetch('https://example');
// SyntaxError: await is only valid in async function
```

As a way to get access to the feature, I often wrapped the await statements in an immediately invoked async function expression, like this:

await/Await_ex02.js

```
(async function() {
  const res = await fetch('https://example');
}());
```

It's unpleasant but functional! With top-level await we no longer have to do this, because the await keyword works outside of async functions as well. Go ahead and execute await fetch("https://google.com", {mode:"no-cors"}); in your browser console. You should see a response like this:



But there's a bigger problem that top-level await allows us solve. Bear with me for a moment. When working with ES modules, we can make variables and functions available outside the module using the export keyword. Then other modules in separate files can use the import keyword to access those variables and functions. Any export or import statement must be expressed at the top level of the code.

Say we have a module that retrieves weather data for Tokyo, Japan, from an external API, and we want to make the result available to other modules:

await/Await_ex03.js

```
let result;
const api = `http://api.openweathermap.org/data/2.5/weather?
q=Tokyo,Japan&APPID=1b1b3e9e909416e5bbe365a0a8505fbb`;
// use your own app id in production

(async () => {
  const response = await fetch(api);
  result = await response.json();
})();

export {result};
```

In another module, we'd like to import the result and extract the temperature:

await/Await_ex04.js

```
import {result} from './module1.js';

console.log(result.main.temp);

// logs:
// => TypeError: Cannot read property 'main' of undefined
```

But this code produces a TypeError because we're trying to access the export before the async function finishes executing. We still have a promise waiting to be settled; until then result has a value of undefined.

Now let's delay the console.log() method and see what happens. Go ahead and encapsulate the method in a setTimeout(), like this:

await/Await_ex05.js

```
import {result} from './module1.js';

// don't do this in production
setTimeout(() => {
  console.log(result.main.temp);
}, 2000);

// logs:
// => 292.94
```

This time we get the result we're looking for (note that the temperature is in kelvin units). This means that exported variables are undefined until the promise

is settled. But this is a bad way of coding our module because now every consumer of module1.js needs to know what a reasonable wait would be.

We can't use the export keyword inside functions, and prior to the introduction of top-level await, we couldn't use the await keyword outside of async functions either.

One workaround is to export the entire async function as the default export value:

await/Await_ex06.js

```
let result;
const api = `http://api.openweathermap.org/data/2.5/weather?
q=Tokyo,Japan&APPID=1b1b3e9e909416e5bbe365a0a8505fbb`;
// use your own app id in production

export default (async () => {
  const response = await fetch(api);
  result = await response.json();
})();

export {result};
```

Then we could wait for the async function to settle before accessing the variable:

await/Await_ex07.js

```
import p, {result} from './module1.js';

p.then(()=>{
  console.log(result.main.temp);
})

// logs:
// => 292.94
```

But as the code becomes more complicated, it will become more difficult to manage the modules this way. Other workarounds could work as well, but they come with their own limitations.

Top-level await aims to solve this problem by enabling developers to use the await keyword outside async functions. You don't need to do anything special to start using top-level await except having a modern browser that supports the feature:

await/Await_ex08.js

```
const api = `http://api.openweathermap.org/data/2.5/weather?
q=Tokyo,Japan&APPID=1b1b3e9e909416e5bbe365a0a8505fbb`;
// use your own app id in production

const response = await fetch(api);
const result = await response.json();
```

```
export {result};
```

Then in another module we can access the exported variable directly:

await/Await_ex09.js

```
import {result} from './module1.js';

console.log(result.main.temp);

// logs:
// => 292.94
```

With top-level await, ECMAScript modules can await resources, causing other modules who import them to wait before they start evaluating their own code. Top-level await allows us to do some cool things that would previously require a lot of effort to achieve. Let's take a look at a few practical uses.

Putting Top-Level await to Work

When designing a program to support multiple languages and regions, you may want to use a runtime value to determine the language to use. Say you have an ES module and want to load a language pack dynamically, based on the preferred language of the user set in the browser. You can take advantage of top-level await to import the messages:

await/Await_ex10.js

```
const messages = await import(`./packs/messages-${navigator.language}.js`);
```

The navigator.language property allows us to access the preferred language of the user, which is usually the language of the browser UI. To embed the value of the property within the string, we've put it inside ${...}. The module will be waiting for the language pack to be imported and can only evaluate the rest of the body once the pack has been loaded.

We can also use top-level await to load dependencies with a fallback implementation. As we learned in Avoiding the Single Point of Failure, on page 29, it's important to protect our app against external server issues. Network requests to a server might fail. In critical applications, you can provide dependency fallbacks to mitigate such failures using top-level await. Here's an example:

await/Await_ex11.js

```
let d3;

try {
  d3 = await import('https://cdnjs.cloudflare.com/ajax/libs/d3/6.7.0/d3.min.js');
} catch {
  d3 = await import('https://ajax.googleapis.com/ajax/libs/d3js/6.7.0/d3.min.js');
}
```

In this code, we first attempt to load the D3 JavaScript library from Cloudflare. If the import fails, we try an alternate CDN. Alternatively, we can use Promise.any() to execute both requests asynchronously and use the result of the one that responds faster:

await/Await_ex12.js
```
const CDNs = [
  'https://cdnjs.cloudflare.com/ajax/libs/d3/6.7.0/d3.min.js',
  'https://ajax.googleapis.com/ajax/libs/d3js/6.7.0/d3.min.js'
];

const d3 = await Promise.any(CDNs);
```

Another use case for top-level await is resource initialization. For example:

await/Await_ex13.js
```
import {dbConnector} from './utilities.js';

const connection = await dbConnector.connect();
```

By using top-level await, we can make the module behave like a big async function. We can now represent resources with await and handle errors if the module can't be used.

Remember, a module won't start evaluating its body until the module that's being imported has finished executing its body. So if the other module has a top-level await, it must be completed before the module that's importing it begins executing.

Wrapping Up

Previously, await was valid only in async functions. Times changed and await got easier. We can use top-level await in modules to avoid wrapping code in an async IIFE. We no longer need a workaround when accessing the result of a promise from another module. And we can use await for tasks like initializing resources, defining dependency paths dynamically, and loading dependencies with a fallback implementation.

Thank you for reading!

We've reached the end of the book! The beauty of JavaScript is that it is flexible enough to let programmers determine how the language will evolve. As a result, the inputs and proposals from the JavaScript community have played an important role in the way the language has progressed. Anyone can get involved by drafting a proposal (read https://github.com/tc39/proposals).

An interesting aspect of the process is that new features are implemented by browsers before they're added to the specification. If you want to tune into the features that are coming down the pike, here are some resources:

- Check out the proposals repository on GitHub (https://github.com/tc39/proposals). Particularly, look for features that are at stage 3 and stage 4. When a feature is at stage 3, that means the feature's semantics, syntax, and API are completed and it's very likely to be added to the specification. A stage 4 proposal will be included in the soonest practical standard revision of the specification.
- Watch the Google's V8 JavaScript engine blog. If something gets added to Chrome, this is the place you'll hear about the feature (https://v8.dev/).
- Follow and take part in the TC39 Discourse group (https://es.discourse.group/). TC39 is the committee responsible for evolving the definition of JavaScript.

Thank you for reading! It was my pleasure to write this book. I hope you've found it useful.

Thank you!

We hope you enjoyed this book and that you're already thinking about what you want to learn next. To help make that decision easier, we're offering you this gift.

Head over to https://pragprog.com right now and use the coupon code BUYANOTHER2021 to save 30% on your next ebook. Void where prohibited or restricted. This offer does not apply to any edition of the *The Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propose a writing idea to us? After all, many of our best authors started off as our readers, just like you. With a 50% royalty, world-class editorial services, and a name you trust, there's nothing to lose. Visit https://pragprog.com/become-an-author/ today to learn more and to get started.

We thank you for your continued support, and we hope to hear from you again soon!

The Pragmatic Bookshelf

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by professional developers for professional developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/fkajs
Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up to Date
https://pragprog.com
Join our announcement mailing list (low volume) or follow us on twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this ebook, perhaps you'd like to have a paper copy of the book. Paperbacks are available from your local independent bookstore and wherever fine books are sold.

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764