O'REILLY®

Multithreaded JavaScript

Concurrency Beyond the Event Loop

Early Release

RAW & UNEDITED

Thomas Hunter II
& Bryan English

Multithreaded JavaScript

Concurrency Beyond the Event Loop

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Thomas Hunter II
Bryan English

Multithreaded JavaScript

by Thomas Hunter II and Bryan English

Copyright © 2021 Thomas Hunter II and Bryan English. All rights reserved.

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Amanda Quinn

Development Editor=: Corbin Collins

Production Editor: Daniel Elfanbaum

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

November 2021: First Edition

Revision History for the Early Release

- 2021-04-19: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098104436 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Multithreaded JavaScript*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent

the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10436-8

[LSI]

Chapter 1. Introduction

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Computers used to be much simpler. That's not to say they were easy to use or write code for, but conceptually there was a lot less to work with. PCs in the 1980s typically had a single 8-bit CPU core, and not a whole lot of memory. You typically could only run a single program at one time. What we think of these days as operating systems would not even be running at the same time as the program the user was interacting with.

Eventually, people wanted to run more than one program at once, and multitasking was born. This allowed operating systems to run several programs at the same time by switching execution between them. Programs could decide when it would be an appropriate time to let another program run by yielding execution to the operating system. This approach is called *cooperative multitasking*.

In a cooperative multitasking environment, when a program fails to yield execution for any reason, no other program can continue executing. This interruption to other programs is not desirable, so eventually operating systems moved toward *preemptive multitasking*. In this model, the operating system would determine which program would run on the CPU at which time, using its own notion of scheduling, rather than relying on the programs themselves to be

the sole deciders of when to switch execution. To this day, almost every operating system uses this approach, even on multi-core systems, because we generally have more programs running that we have CPU cores.

Running multiple tasks at once is extremely useful for both programmers and users. Before threads, a single program (that is, a single *process*) could not have multiple tasks running at the same time. Instead, programmers wishing to perform tasks concurrently would either have to split up the task into smaller chunks and schedule them inside the process, or run separate tasks in separate processes and have them communicate with each other.

Even today, in some high-level languages the appropriate way to run multiple tasks at once is to run additional processes. In some languages, like Ruby and Python, there's a *global interpreter lock (GIL)*, meaning only one thread can be executing at a given time. While this makes memory management far more practical, it makes multithreaded programming not as attractive to programmers, and instead multiple processes are employed.

Until fairly recently, JavaScript was a language where the only multitasking mechanisms available were splitting tasks up and scheduling their pieces for later execution, and in the case of Node.js, running additional processes. Today, in all major JavaScript environments, we have access to threads, and unlike Ruby and Python, we don't have a GIL making them effectively useless for performing CPU-intensive tasks. Instead, other trade-offs are made, like not sharing JavaScript objects across threads (at least not directly). Still, threads are useful to JavaScript developers for cordoning off CPU-intensive tasks. In the browser, there are also special-purpose threads that have feature sets available to them that are different from the main thread.

The purpose of this book is to explore and explain JavaScript threads as a programming concept and tool. You'll learn how to use them, and more importantly, when to use them. Not every problem needs to be solved with threads. Not even every CPU-intensive problem needs to be solved with threads. It's the job of software developers to evaluate problems and tools to determine the most appropriate solutions. The aim here is to give you another tool, and enough knowledge around it to knew when to use it and how.

What Are Threads?

In all modern operating systems, all units of execution outside the kernel are organized into processes and threads. Developers can use processes and threads, and communication between them, to add concurrency to a project. On systems with multiple CPU cores, this also means adding parallelism.

When you execute a program, such as Node.js or a code editor, you're initiating a process. This means that code is loaded into a memory space unique to that process, and no other memory space can be addressed by the program without either asking the kernel for more memory, or for a different memory space to be mapped in. Without adding threads or additional processes, only one instruction is executed at a time, in the appropriate order as prescribed by the program code.

A program may spawn additional processes, which have their own memory space. These processes do not share memory (unless it's mapped in via additional system calls), and have their own instruction pointers, meaning each one can be executing a different instruction at the same time. If the processes are being executed on the same core, the processor may switch back and forth between processes, temporarily stopping execution for that one process while another one executes.

A process may also spawn threads, rather than full-blown processes. A thread is just like a process, except that it shares memory space with the process that it belongs to. A process can have many threads, and each one has its own instruction pointer. All the same properties about execution of processes apply to threads as well. Because they share a memory space, it's easy to share program code and other values between threads. This makes them more valuable than processes for adding concurrency to programs, but at the cost of some complexity in programming, which we'll cover later on in this book.

A typical way to take advantage of threads is to offload CPU-intensive work, like mathematical operations, to an additional thread or pool of threads while the main thread is free to interact externally with the user or other programs by checking for new interactions inside an infinite loop. Many classic web server programs, such as Apache, use a system like this in order to handle large loads of HTTP requests. This might end up looking something like Figure 1-1. In this model, HTTP request data is passed to a worker thread for processing, and then

when the response is ready, it’s handed back to the main thread to be returned back to the user agent.

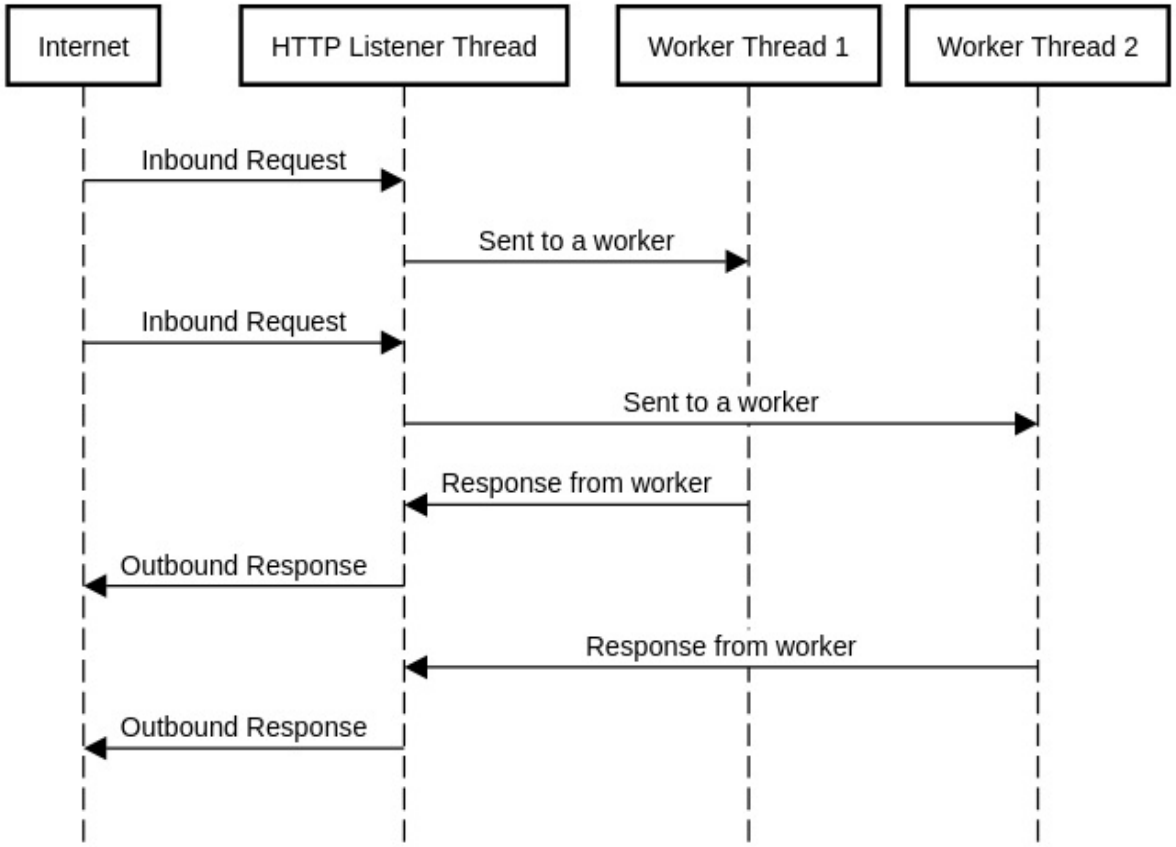

Figure 1-1. Worker threads as they might be used in an HTTP server.

In order for threads to be useful, they need to be able to coordinate with each other. This means they have to be able to do things like wait for things to happen on other threads, and get data from them. As discussed, we have a shared memory space between threads, and with some other basic primitives, systems for passing messages between threads can be constructed. In many cases, these sorts of constructs are available at the language or platform level.

Concurrency vs. Parallelism

It’s important to distinguish between concurrency and parallelism, since they’ll come up fairly often when programming in a multithreaded manner. These are very related terms, and can mean very similar things depending on the circumstances. Let’s start with some definitions.

Concurrency

: Tasks are run in overlapping time.

Parallelism

: Tasks are run at exactly the same time.

While it may seem like these mean they same thing, consider that tasks may be broken up into smaller parts and then interleaved. In this case, concurrency can be achieved without parallelism, since the time-frames that the tasks run in can be overlapped. For tasks to be running with parallelism, they must be running at *exactly the same time*. Generally, this means they must be running on separate CPU cores at exactly the same time.

Threads do not automatically provide parallelism. The system hardware must allow for this by having multiple CPU cores, and the operating system scheduler must decide to run the threads on separate CPU cores. On single-core systems, or systems with more threads running than CPU cores, multiple threads may be run on a single CPU *concurrently* by switching between them at appropriate times. Also, in languages with a GIL like Ruby and Python, threads are explicitly prevented from offering parallelism because only one instruction can be executed at a time throughout the entire runtime.

It’s important to also think about this in terms of timing, since threads are typically added to a program to increase performance. If your system is only allowing for concurrency, due to only having a singly CPU core available, or be being already loaded with other tasks, then there may not be any perceived benefit to using extra threads. In fact, the overhead of synchronization and context-switching between the threads may end up making the program perform even worse. Always measure the performance of your application under the conditions it’s expected to run in. That way you verify whether a multithreaded programming model will actually be beneficial to you.

Single-Threaded JavaScript

Historically, the platforms JavaScript runs on did not provide any thread support,

so the language was thought of as a single-threaded environment. Whenever you hear someone say that JavaScript is single-threaded, they're referring to this historical background and the programming style that it naturally lent itself to. It's true though, that despite the title of this book, the language itself does not have any built-in functionality to create threads. This shouldn't be that much of a surprise, because it also doesn't have any built-in functionality to interact with the network, devices, file system, or make any system calls. Indeed, even such basics as `setTimeout()` aren't actually JavaScript features. Instead environments the VM is embedded in, such as Node.js or browsers, provide these via environment-specific APIs.

Instead of threads as a concurrency primitive, most JavaScript code is written in an event-oriented manner operating on a single execution thread. As various events like user interactions or IO happen, they trigger the execution of functions previously set to run upon these events. These functions are typically called *callbacks* and are at the core of how asynchronous programming is done in Node.js and the browser. Even in promises or the `async`/`await` syntax, callbacks are the underlying primitive. It's important to recognize that callbacks are not running in parallel, or alongside any other code. When code in a callback is running, that's the only code that's currently running. Put another way, only one call stack is active at any given time.

It's often easy to think of operations happening in parallel, when in fact they're happening concurrently. For example, imagine you want to open three files containing numbers, named *1.txt*, *2.txt*, and *3.txt*, and then add up the results and print them. In Node.js, you might do something like Example 1-1.

Example 1-1. Reading from files concurrently in Node.js.

```
const fs = require('fs').promises;

async function getNum(filename) {
  return parseInt(await fs.readFile(filename, 'utf8'), 10);
}

(async () => {
  try {
    const numberPromises = [1, 2, 3].map(i => getNum(`${i}.txt`));
    const numbers = await Promise.all(numberPromises);
    console.log(numbers[0] + numbers[1] + numbers[2]);
  } catch (err) {
    console.error('Something went wrong:');
```

```
    console.error(err);
  }
})();
```

Since we're using `Promise.all()`, we're waiting for all three files to be read and parsed. If you squint a bit, it may even look similar to the `pthread_join()` from the C example later in this chapter. However, just because the promises are being created together and waited upon together doesn't mean that the code resolving them runs at the same time, it just means their timeframes are overlapping. There's still only one instruction pointer, and only one instruction is being executed at a time.

In the absence of threads, there's only one JavaScript environment to work with. This means one instance of the VM, one instruction pointer, and one instance of the garbage collector. By one instruction pointer, we mean that the JavaScript interpreter is only executing on instruction at any given time. That doesn't mean we're restricted to one global object though. In both the browser and Node.js, we have Realms at our disposal.

Realms can be thought of as instances of the JavaScript environment as provided to JavaScript code. This means that each Realm gets its own global object, and all of the associate properties of the global object, such as built-in classes like `Date` and other objects like `Math`. The global object is referred to as `global` in Node.js and `window` in browsers, but in modern versions of both, you can refer to the global object as `globalThis`.

In browsers each frame in a web page has a Realm for all of the JavaScript within it. Since each frame has its own copy of `Object` and other primitives within it, you'll notice that they have their own inheritance trees, and `instanceof` might not work as you expect it to when operating on objects from different Realms. This is demonstrated in Example 1-2.

Example 1-2. Objects from a different from in a browser.

```
const iframe = document.createElement('iframe');
document.body.appendChild(iframe);
const FrameObject = iframe.contentWindow.Object; ❶

console.log(Object === FrameObject); ❷
console.log(new Object() instanceof FrameObject); ❸
console.log(FrameObject.name); ❹
```

❶ The global object inside the `iframe` is accessible with the

`contentWindow` property.

❷ This returns false, so the `Object` inside the frame is not the same as in the main frame.

❸ `instanceof` fails, as expected since they're not the same `Object`.

❹ Despite all this, the constructors have the same `name` property.

In Node.js, Realms can be constructed with the `vm.createContext()` function, as shown in Example 1-3. In the parlance of Node.js, Realms are called Contexts. All the same rules and properties applying to browser frames also apply to Contexts, but in Contexts, you don't have access to any global properties or anything else that might be in scope in your Node.js files. If you want to use these features, they need to be manually passed in to the Context.

Example 1-3. Objects from a new Context in Node.js.

```
const vm = require('vm');
const ContextObject = vm.runInNewContext('Object'); ❶

console.log(Object === ContextObject); ❷
console.log(new Object() instanceof ContextObject); ❸
console.log(ContextObject.name); ❹
```

❶ We can get objects from a new context using `runInNewContext`.

❷ This returns false, so as with browser iframes, `Object` inside the context is not the same as in the main context.

❸ Similarly, `instanceof` fails.

❹ Once again, the constructors have the same `name` property.

In any of these Realm cases, it's important to note that we still only have instruction pointer, and code from only one Realm is running at a time, because we're still only talking about single-threaded execution.

Hidden Threads

While your JavaScript code may run, at least by default, in a single-threaded environment, that doesn't mean the process running your code is single-threaded. In fact, many threads might be used in order to have that code running

smoothly and efficiently. It's a common misconception that Node.js is a single-threaded process.

Modern JavaScript engines like V8 use separate threads to handle garbage collection and other features that don't need to happen in-line with JavaScript execution. In addition, the platform runtimes themselves may use additional threads to provide other features.

In Node.js, `libuv` is used as an OS-independent asynchronous IO interface, and since not all system-provided IO interfaces are asynchronous, it uses a pool of worker threads to avoid blocking program code when using otherwise-blocking APIs, such as file system APIs. By default, four of these threads are spawned, though this number is configurable via the `UV_THREADPOOL_SIZE` environment variable, and can be up to 1024.

On UNIX-like systems, you can see these extra threads by using `top -H` on a given process. In Example 1-4, a simple Node.js web server was started, and the PID was noted and passed to `top`. You can see the various V8 and `libuv` threads add up to 7 threads, including the one that the JavaScript code runs in. You can try this with your own Node.js programs, and even try changing the `UV_THREADPOOL_SIZE` environment variable to see the number of threads change.

Example 1-4. Output from `top`, showing the threads in a Node.js process.

```
$ top -H -p 81862
top - 14:18:49 up 1 day, 23:18,  1 user,  load average: 0.59, 0.82, 0.83
Threads:   7 total,   0 running,   7 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.2 us,  0.0 sy,  0.0 ni, 97.8 id,  0.0 wa,  0.0 hi,  0.0 si,
0.0 st
MiB Mem :  15455.1 total,   2727.9 free,   5520.4 used,   7206.8
buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   8717.3 avail
Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+
COMMAND
  81862 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.03
node
  81863 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00
node
  81864 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00
node
  81865 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00
```

```
node
  81866 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00
node
  81867 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00
node
  81868 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00
node
```

Browsers similarly perform many tasks, such as DOM rendering, in threads other than the one used for JavaScript execution. An experiment with `top -H` like we did for Node.js would result in a similar handful of threads. Modern browsers take this even further by using multiple processes in order to add a layer of security by isolation.

It's important to think about these extra threads when going through a resource-planning exercise for your application. You should never assume that just because JavaScript is single-threaded that only one thread will be used by your JavaScript application. For example, in production Node.js applications, measure the number of threads used by the application and plan accordingly. Don't forget that many of the native addons in the Node.js ecosystem spawn threads of their own as well, so it's important to go through this exercise on an application-by-application basis.

Threads in C: Get Rich with Happycoin

Threads are obviously not unique to JavaScript. They're a long-standing concept at the operating system level, independent of languages. Let's explore how a threaded program might look in C. C is an obvious choice here, since the C interface for threads is what underlies most thread implementations in higher-level languages, even if there may seem to be different semantics.

Let's start with an example. Imagine a proof-of-work algorithm for a simple and impractical cryptocurrency called Happycoin, as follows:

1. Generate a random unsigned 64-bit integer.
2. Determine whether or not the integer is happy.
3. If it's not happy, it's not a Happycoin.
4. If it's not divisible by 10,000, it's not a Happycoin.

5. Otherwise, it's a Happycoin.

A number is happy if it eventually goes to 1 when replacing it with the sum of the squares of its digits, and looping until either the 1 happens, or a previously-seen number arises. Wikipedia defines it clearly, and also points out that if any previously seen numbers arise, then 4 will arise, and vice versa. You may notice that our algorithm is needlessly too expensive, because we could check for divisibility before checking for happiness. This is intentional, because we're trying to demonstrate a heavy workload.

Let's build a simple C program that runs the proof-of-work algorithm 10,000,000 times, printing any Happycoins found, and a count of them.

> **NOTE**
>
> The `cc` in the compilation steps here can be replaced with `gcc` or `clang`, depending on which is available to you. On most systems, `cc` is an alias for either `gcc` or `clang`, so that's what we'll use here.
>
> Windows users may have to do some extra work here to get this going in Visual Studio, and the threads example won't work out-of-the-box on Windows since it uses POSIX threads rather than Windows threads, which are different. To simplify trying this on Windows, the recommendation is to use Windows Subsystem for Linux so that you have a POSIX-compatible environment to work with.

With Only the Main Thread

Create a file called *happycoin.c*, in a directory called *ch1-c-threads/*. We'll build up this file over the course of this section. To start off, add the code as shown in Example 1-5.

Example 1-5. ch1-c-threads/happycoin.c

```
#include <inttypes.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

uint64_t random64(unsigned int * seed) {
  uint64_t result;
  uint8_t * result8 = (uint8_t *)&result;
  for (int i = 0; i < 8; i++) {
```

```
    result8[i] = rand_r(seed);
  }
  return result;
}
```

We've added a bunch of `includes` which give us handy things like types, IO functions, and the time and random number functions we'll be needing. Since the algorithm requires the generation of a random 64-bit unsigned integer (i.e. a `uint64_t`), we need eight random bytes, which `random64()` gives us by calling `rand_r()` until we have enough bytes. Since `rand_r()` also required a reverence to a seed, we'll pass that into `random64()` as well.

Now let's add our happy number calculation as shown in Example 1-6.

Example 1-6. ch1-c-threads/happycoin.c

```
uint64_t sum_digits_squared(uint64_t num) {
  uint64_t total = 0;
  while (num > 0) {
    uint64_t num_mod_base = num % 10;
    total += num_mod_base * num_mod_base;
    num = num / 10;
  }
  return total;
}

bool is_happy(uint64_t num) {
  while (num != 1 && num != 4) {
    num = sum_digits_squared(num);
  }
  return num == 1;
}

bool is_happycoin(uint64_t num) {
  return is_happy(num) && num % 10000 == 0;
}
```

To get the sum of the squares of the digits in `sum_digits_squared`, we're using the mod operator, %, to get each digit from right to left, squaring it, then adding it to our running total. We then use this function in `is_happy` in a loop, stopping when the number is 1 or 4. We stop at 1, since that indicates the number is happy. We also stop at 4, because that's indicative of an infinite loop where we never end up at 1. Finally, in `is_happycoin()`, we do the work of checking whether a number is happy and also divisible by 10,000.

Let's wrap this all up in our `main()` function as shown in Example 1-7.

Example 1-7. ch1-c-threads/happycoin.c

```
int main() {
  unsigned int seed = time(NULL);
  int count = 0;
  for (int i = 1; i < 10000000; i++) {
    uint64_t random_num = random64(&seed);
    if (is_happycoin(random_num)) {
      printf("%" PRIu64 " ", random_num);
      count++;
    }
  }
  printf("\ncount %d\n", count);
  return 0;
}
```

First, we need a seed for the random number generator. The current time is as suitable a seed as any, so we'll use that via `time()`. Then, we'll loop 10,000,000 times, first getting a random number from `random64()`, then checking if it's happy. If it is, we'll increment the count, and print the number out. The weird `PRIu64` syntax in the `printf()` call is necessary for properly printing out 64-bit unsigned integers. When the loop completes, we print out the count and exit the program.

To compile and run this program use the following commands in your *ch1-c-threads* directory.

```
$ cc -o happycoin happycoin.c
$ ./happycoin
```

You'll get a list of Happycoins found on one line, and the count of them on the next line. For a given run of the program, it might look something like Example 1-8.

Example 1-8. Example output of `./happycoin`.

```
$ ./happycoin
11023541197304510000 [... 167 entries redacted for brevity ...]
7705413983788400000
count 169
```

It takes a non-trivial amount of time to run this program; about 2 seconds on a run-of-the-mill computer. This is a case where threads can be useful to speed

things up, because many iterations of the same largely mathematical operation are being run. Let's go ahead and convert this example to a multithreaded program.

With Four Worker Threads

We'll set up four threads that will each run a quarter of the iterations of the loop that generates a random number and tests if it's a Happycoin.

In POSIX C, threads are managed with the `pthread_*` family of functions. The `pthread_create()` function is use to create a thread. A function is passed in that will be executed on that thread. Program flow continues on the main thread. The program can wait for a thread's completion by calling `pthread_join()` on it. You can pass arguments to the function being run on the thread via `pthread_create()` and get return values from `pthread_join()`.

In our program, we'll isolate the generation of Happycoins in a function called `get_happycoins()` and that's what will run in our threads. We'll create the 4 threads, and then immediately wait for the completion of them. Whenever we get the results back from a thread, we'll output them, and store the count so we can print the total at the end. To help in passing the results back, we'll created a simple `struct` called `happy_result`.

Make a copy of your existing *happycoin.c* and name it *happycoin-threads.c*. Then in the new file, insert the code in Example 1-9 under the last `#include` currently in the file.

Example 1-9. ch1-c-threads/happycoin-threads.c

```
#include <pthread.h>

struct happy_result {
  int count;
  uint64_t * nums;
};
```

The first line includes `pthread.h`, which gives us access to the various thread functionality we'll need. Then `struct happy_result` is defined, which we'll use as the return value for our thread function `get_happycoins()` later on. It stores an array of found happycoins, represented here by a pointer, and the

count of them.

Now, go ahead and delete the whole `main()` function, since we're about to replace it. First, lets add our `get_happycoins()` function in Example 1-10, which is the code that will run on our worker threads.

Example 1-10. ch1-c-threads/happycoin-threads.c

```
void * get_happycoins(void * arg) {
  int attempts = *(int *)arg;
  int limit = attempts/10000;
  unsigned int seed = time(NULL);
  uint64_t * nums = malloc(limit * sizeof(uint64_t));
  struct happy_result * result = malloc(sizeof(struct happy_result));
  result->nums = nums;
  result->count = 0;
  for (int i = 1; i < attempts; i++) {
    if (result->count == limit) {
      break;
    }
    uint64_t random_num = random64(&seed);
    if (is_happycoin(random_num)) {
      result->nums[result->count++] = random_num;
    }
  }
  return (void *)result;
}
```

You'll notice that this function takes in a single `void *` and returns a single `void *`. That's the function signature expected by `pthread_create()`, so we don't have a choice here. This means we have to cast our arguments to what we want them to be. We'll be passing in the number of attempts, so we'll cast the argument to an `int`. Then, we'll set the seed as we did in the previous example, but this time it's happening in our thread function, so we get a different seed per thread.

After allocating enough space for our array and `struct happy_result`, we go ahead into the same loop that we did in `main()` in the single-threaded example, only this time we're putting the results into the struct instead of printing them. Once the loop is done, we return the struct as a pointer, which we cast as `void *` to satisfy the function signature. This is how information is passed back to the main thread, which will make sense of it.

This demonstrates one of the key properties of threads that we don't get from

processes, which is the shared memory space. If, for example, we were using processes instead of threads, and some *interprocess communication (IPC)* mechanism to transfer results back, we wouldn't be able to simply pass a memory address back to the main process, since the main process wouldn't have access to memory of the worker process. Thanks to virtual memory, the memory address might refer to something else entirely in the main process. Instead of passing a pointer, we'd have to pass the entire value back over the IPC channel, which can introduce performance overhead. Since we're using threads instead of processes, we can just use the pointer, so that the main thread can use it just the same.

Shared memory isn't without its trade-offs though. In our case, there's no need for the worker thread to make any use of the memory it has now passed to the main thread. This isn't always the case with threads. In a great multitude of cases, it's necessary to properly manage how threads access shared memory via synchronization, otherwise some unpredictable results may occur. We'll go into how this works in JavaScript in detail in [Link to Come] and [Link to Come].

Now, let's wrap this up with the `main()` function in Example 1-11.

Example 1-11. ch1-c-threads/happycoin-threads.c

```
#define THREAD_COUNT 4

int main() {
  pthread_t thread [THREAD_COUNT];

  int attempts = 10000000/THREAD_COUNT;
  int count = 0;
  for (int i = 0; i < THREAD_COUNT; i++) {
    pthread_create(&thread[i], NULL, get_happycoins, &attempts);
  }
  for (int j = 0; j < THREAD_COUNT; j++) {
    struct happy_result * result;
    pthread_join(thread[j], (void **)&result);
    count += result->count;
    for (int k = 0; k < result->count; k++) {
      printf("%" PRIu64 " ", result->nums[k]);
    }
  }
  printf("\ncount %d\n", count);
  return 0;
}
```

First, we'll declare our four threads as an array on the stack. Then, we divide our desired number of attempts (10,000,000) by the number of threads. This is what will be passed to `get_happycoins()` as an argument, which we see inside the first loop, which creates each of the threads with `pthread_create()`, passing in the number of attempts per thread as an argument. In the next loop, we wait for each of the threads to finish their execution with `pthread_join()`. Then we can print the results and the total from all the threads, just like we would in the single-threaded example.

> **NOTE**
>
> This program leaks memory. One hard part of multithreaded programming in C and some other languages is that it can be very easy to lose track of where and when memory is allocated and where and when it should be freed. See if you can modify the code here to ensure the program exits with all heap-allocated memory freed.

With the changes complete, you can compile and run this program with the following commands in your *ch1-c-threads* directory.

```
$ cc -pthread -o happycoin-threads happycoin-threads.c
$ ./happycoin-threads`
```

The output should look something like Example 1-12.

Example 1-12. Example output of `./happycoin-threads`.

```
$ ./happycoin-threads
2466431682927540000 [... 154 entries redacted for brevity ...]
15764177621931310000
count 156
```

You'll notice output similar to the single-threaded example.[1] You'll also notice that it's a bit faster. On my computer it finishes in about 0.8 seconds. This isn't *quite* four times as fast, since there's some initial overhead in the main thread, and also the cost of printing of results. We could print the results as soon as they're ready on the thread that's doing the work, but if we do that, the results may clobber each other in the output, since nothing stops two threads from printing to the output stream at the same time. By sending the results to the main thread, we can coordinate the printing of results there so that nothing gets

clobbered.

This illustrates the primary advantage and one drawback of threaded code. On one hand, it's useful for splitting up computationally expensive tasks so that they can be run in parallel. On the other hand, we need to ensure that some events are properly synchronized so that weird errors don't occur. When adding threads to your code in any language, it's worth making sure that the use is appropriate. Also, as with any exercise in attempting to make faster programs, always be measuring. You don't want to have the complexity of threaded code in your application if it doesn't turn out to give you any actual benefit.

Any programming language supporting threads is going to provide some mechanisms for creating and destroying threads, passing messages in between, and interacting with data that's shared between the threads. This may not look the same in every language, because as languages and their paradigms are different, so are their programmatic models of parallel programming. Now that we've explored what threaded programs look like in a low-level language like C, let's dive in to JavaScript. Things will look a little different, but as you'll see, the principles remain the same.

---

1 The fact that the total count from the multithreaded example is different from the single-threaded example is irrelevant, since the count is dependent on how many random numbers happened to be Happycoins. The result will be completely different between two different runs.

Chapter 2. Browsers

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

JavaScript doesn't have a single, bespoke implementation like most other programming languages do. For example, with Python, you're probably going to run the Python binary provided by the language maintainers. JavaScript, on the other hand, has many different implementations. This includes the JavaScript engine that ships with different web browsers, such as V8 in Chrome, SpiderMonkey in Firefox, and Nitro in Safari. The V8 engine is also used by Node.js, a popular JavaScript runtime for building server software.

These separate implementations each start off by implementing some facsimile of the ECMAScript specification. As the compatability charts that we so often need to consult suggest, not every engine implements JavaScript the same way. At the language level, there are some concurrency primitives that have been made available, which are covered in more detail in [Link to Come] and [Link to Come].

Other APIs are also added in each implementation to make the JavaScript that can be run even more powerful. This chapter focuses entirely on the multithreaded APIs that are provided by modern web browsers, the most approachable of which is the web worker.

Dedicated Workers

Web workers allow you to spawn a new environment for executing JavaScript in. JavaScript that is executed in this way is allowed to run in a separate thread from the JavaScript that spawned it. Communication occurs between these two environments by using a pattern called *message passing*. Recall that it's JavaScript's nature to be single threaded. Web workers play nicely with this nature and expose message passing by way of triggering functions to be run in the event loop.

It's possible for a JavaScript environment to spawn more than one web worker, and a given web worker is free to spawn even more web workers. That said, if you find yourself spawning massive hierarchies of web workers, you might need to reevaluate your application.

There is more than one type of web worker, the simplest of which is the dedicated worker.

Dedicated Worker Hello World

The best way to learn a new technology is to actually work with it. The relationship between page and worker that you are building is displayed in Figure 2-1. In this case you'll create just a single worker, but a hierarchy of workers is also achieveable.

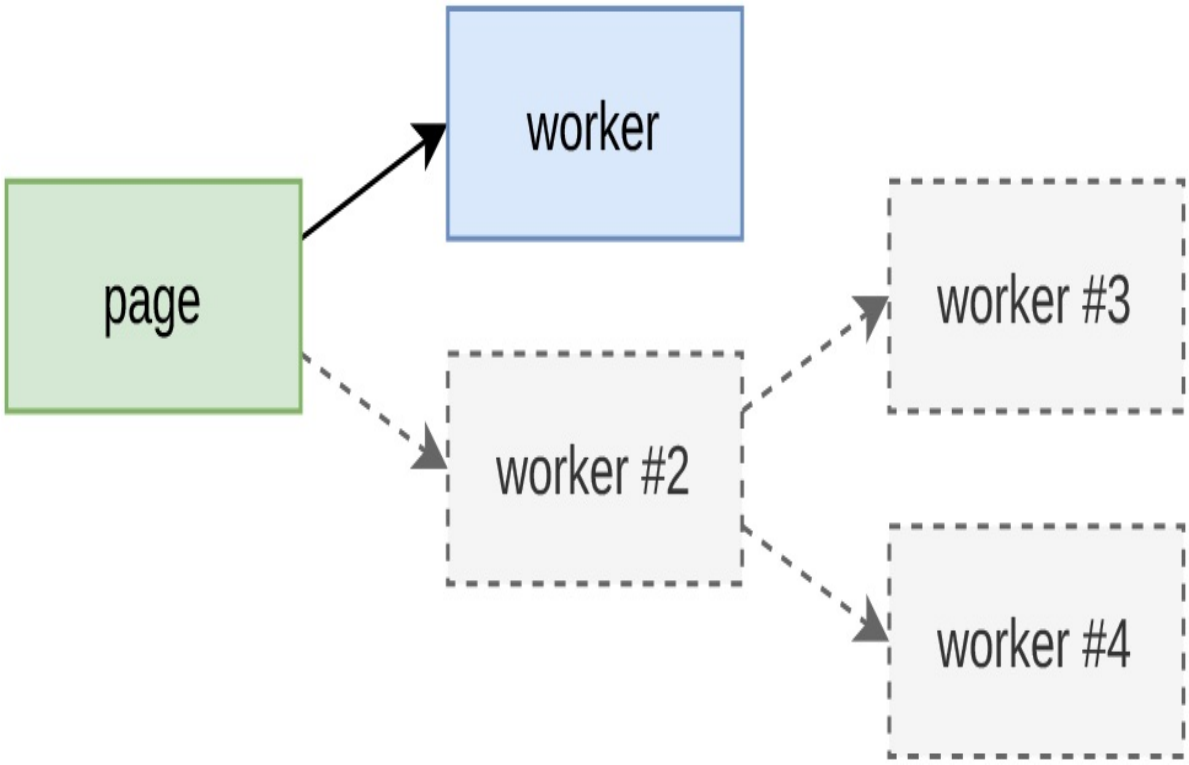

Figure 2-1. Dedicated Worker relationship

First, create a directory named *ch2-web-workers/*. You'll keep the three example files required for this project in there. Next, create an *index.html* file inside of the directory. JavaScript that runs in the browser needs to first be loaded by a web page, and this file represents the basis of that web page. Add the content from listing Example 2-1 to this file to kick things off.

Example 2-1. ch2-web-workers/index.html

```
<html>
  <head>
    <title>Web Workers Hello World</title>
    <script src="main.js"></script>
  </head>
</html>
```

As you can see, this file is super basic. All it is doing is setting a title and loading a single JavaScript file named *main.js*. The remaining sections in this chapter follow a similar pattern. The more interesting part is what's inside of the *main.js* file.

In fact, create that *main.js* file now, and add the content from Example 2-2 to it.

Example 2-2. ch2-web-workers/main.js

```
console.log('hello from main.js');

const worker = new Worker('worker.js'); ❶

worker.onmessage = (msg) => { ❷
  console.log('message received from worker', msg.data);
};

worker.postMessage('message sent to worker'); ❸

console.log('hello from end of main.js');
```

❶ Instantiation of a new dedicated worker.

❷ A `message` handler is attached to the worker.

❸ A message is passed into the worker.

The first thing that happens in this file is that a call to `console.log()` is made. This is to make it obvious the order in which files get executed. The next thing that happens is that a new dedicated worker gets instantiated. This is done by calling `new Worker(filename)`. Once called, the JavaScript engine begins the download (or cache lookup) for the appropriate file in the background.

Next, a handler for the `message` event is attached to the worker. This is done by assigning a function to the `.onmessage` property of the dedicated worker. When a message is received, that function gets called. The argument provided to the function is an instance of `MessageEventPrototype`. It comes with a bunch of properties but the one that's most interesting is the `.data` property. This represents the object that was returned from the dedicated worker.

Finally, a call to the dedicated worker's `.postMessage()` method is made. This is how the JavaScript environment that instantiates the dedicated worker is able to communicate with the dedicated worker. In this case a basic string has been passed into the dedicated worker. There are restrictions on what kind of data can be passed into this method; see "Structured Clone Algorithm" for more details.

Now that your main JavaScript file is finished, you're ready to create the file that will be executed within the dedicated worker. Create a new file named *worker.js* and add the contents of to Example 2-3 to it.

Example 2-3. ch2-web-workers/worker.js

```
console.log('hello from worker.js');

self.onmessage = (msg) => {
  console.log('message from main', msg.data);

  postMessage('message sent from worker');
};
```

In this file a single global named `onmessage` is defined and a function is assigned to it. This `onmessage` function, inside of the dedicated worker, is called when the `worker.postMessage()` method is called from outside the dedicated worker. This assignment could also have been written as `onmessage =` or even `var onmessage =`, but using `const onmessage =` or `let onmessage =` or even declaring `function onmessage` won't work. The `self` identifier is intented to be a way to represent the global object across multiple JavaScript environments. The name `window` just doesn't make a lot of sense in Node.js, which runs on a server, or even inside of a dedicated worker, where the `window` object isn't available.

Inside of the `onmessage` function, the code first prints the message that was received from outside of the dedicated worker. After that, it calls the `postMessage()` global function. This method takes an argument, and the argument is then provided to the calling environment by triggering the dedicated worker's `onmessage()` method. The same rules about message passing and object cloning also apply here. Again, the example is just using a simple string for now.

There are some additional rules when it comes to loading a dedicated worker script file. The file that is loaded must be in the same origin that the main JavaScript environment is running in. Also, browsers won't allow you to run dedicated workers when JavaScript runs using the `file://` protocol, which is a fancy way of saying you can't simply double-click the *index.html* file and view the application running. Instead, you'll need to run your application from a web server. Luckily, if you have a recent Node.js installed, you can run the following command to start a very basic web server locally:

```
$ npx serve .
```

Once executed, this command spins up a server that hosts files from the local filesystem. It also displays the URL that the server is available as. On Thomas's machine, the command outputs the following URL:

```
http://localhost:5000
```

Copy whatever URL was provided to you and open it using a web browser. When the page first opens you'll most likely see a plain, white screen. But, that's not a problem, as all of the output is being displayed in the web developer console. Different browsers make the console available in different ways, but usually you can right-click somewhere in the background and click the Inspect menu option, or you can press Ctrl + Shift + I (or Cmd + Shift + I) to open up the inspector. Once in the inspector, click on the Console tab, and then refresh the page just in case any console messages weren't captured. Once that's done you should see the messages that are displayed in Table 2-1.

Table 2-1. Example Console output

Log	Location
hello from main.js	main.js:1:9
hello from end of main.js	main.js:11:9
hello from worker.js	worker.js:1:9
message from main, message sent to worker	worker.js:4:11
message received from worker, message sent from worker	main.js:6:11

This output confirms the order in which the messages have been executed. First, the *main.js* file is loaded, and it's output is printed. The worker is instantiated and configured, its `postMessage()` method is called, and then the last message gets printed as well. Next, the *worker.js* file is run, and its message handler is called, printing a message. It then calls `postMessage()` to send a message back to *main.js*. Finally, the `onmessage` handler for the dedicated worker is called in *main.js*, and the final message is printed.

Advanced Dedicated Worker Usage

Now that you're familiar with the basics of dedicated workers, you're ready to work with some of the more complex features.

When you work with JavaScript that doesn't involve dedicated workers, all the code you end up loading is available in the same realm. Loading new JavaScript code is done either by loading a script with a `<script>` tag, or by making an XHR request and using the `eval()` function with a string representing the code. When it comes to dedicated workers, you can't inject a `<script>` tag into the DOM since there's no DOM associated with the worker.

Instead, you can make use of the `importScripts()` function, which is a global function that is only available within dedicated workers. This function accepts one or more arguments that represent the paths to scripts to be loaded. These scripts will be loaded from the same origin as the web page. These scripts are loaded in a synchronous manner, so code that follows the function call will run after the scripts are loaded.

Instances of `Worker` inherit from `EventTarget`, and have some generic methods on it for dealing with events. However, the `Worker` class provides the most important methods on the instance. The following is a list of these methods, some of which you've already worked with, some of which are new:

`worker.postMessage(msg)`

Invokes the `self.onmessage` function inside the worker, passing in `msg`.

`worker.onmessage`

If assigned, is invoked when the `self.postMessage` function inside the worker is called.

`worker.onerror`

If assigned, is invoked when an error is thrown inside the Worker. A single `ErrorEvent` argument is provided, having a `.colno`, `.lineno`, `filename`, and `.message` property. This error will bubble up unless you call `err.preventDefault()`.

`worker.onmessageerror`

If assigned, it is invoked when the worker receives an un-deserializable messae.

`worker.terminate()`

If called, the worker terminates immediately. Future calls to `worker.postMessage()` will silently fail.

Inside of the dedicated worker, the global `self` variable is an instance of `WorkerGlobalScope`. The most notable addition is the `loadScripts()` function for injecting new JavaScript files. Some of the high-level communication APIs like `XMLHttpRequest`, `WebSocket`, and `fetch()` are available. Useful functions that aren't necessarily part of JavaScript but are rebuilt by every major engine, like `setTimeout()`, `setInterval()`, `atob()` and `btoa()` are also available. The two data-storage APIs, `localStorage` and `indexedDB` are available.

When it comes to APIs that are missing, though, you'll need to experiment and see what you have access to. Generally, APIs that modify the global state of the web page aren't available. In the main JavaScript realm, the global `location` is available, and is an instance of `Location`. Inside of a dedicated worker, `location` is still available, but it's an instance of `WorkerLocation` and is a little different, notably missing a `.reload()` method that can cause a page refresh. The `document` global is also missing, which is the API for accessing the page DOM.

When instantiating a dedicated worker, there is an optional second argument for specifying the options for the worker. The instantiation takes on the following signature:

```
const worker = new Worker(filename, options);
```

The `options` argument is an object which can contain the properties listed here:

`type`

Either `classic` (default), for a classic JavaScript file, or `module`, to specify an ESM Module.

`credentials`

TODO

`name`

This names a dedicated worker and is mostly used for debugging. The value is provided in the worker as a global named `name`.

Structured Clone Algorithm

The *structured clone algorithm* is a mechanism that browsers use when copying objects using certain APIs. Most notably, it's used when passing data between workers, though other APIs use it as well. With this mechanism, data is serialized and then later deserialized in a manner such that the resulting deserialized object should be compatible with any given JavaScript realm.

As a quick rule of thumb, any data that can be cleanly represented as JSON can be safely cloned in this manner. Sticking to data represented in this manner will certainly lead to very few surprises. That said, the structured clone algorithm supports many other types of data as well.

First off, all of the primitive data types available in JavaScript, with the exception of the `Symbol` type, can be represented. This includes the `Boolean`, `Null`, `Undefined`, `Number`, `BigInt`, and `String` types.

Instances of `Array`, `Map`, and `Set`, which are each used for storing collections of data, can also be cloned in this manner. Even `ArrayBuffer`, `ArrayBufferBiew`, and `Blob` instances, which store binary data, can be passed along.

Instances of some more complex objects, as long as they are quite universal and well understood, can also be passed through. This includes objects created using the `Boolean` and `String` contructor, `Date`, and even `RegExp` instances[1].

Some more complex and lesser-known object instances, like those for `File`, `FileList`, `ImageBitmap`, and `ImageData` can also be cloned.

Another notable difference that works with the structured clone algorithm, but doesn't work with JSON objects, is that recursive objects, those with nested properties that reference another property, can also be cloned. The algorithm is smart enough to stop serializing an object once it encounters a duplicate, nested object.

There are several "shortcomings" that may affect your implementations. First, a `Function` cannot be cloned in this manner. Functions can be pretty complex things. For example, they have a scope available and can access variables declared outside of them. Passing something like that between realms wouldn't make a whole lot of sense.

Another missing feature, that will likely affect your implementations, is that DOM elements cannot be passed along. Does this mean that the work that a web worker performs can't be displayed to the user in the DOM? Absolutely not. Instead, you'll need to have a web worker return a value that the main JavaScript realm is then able to transform and display to the user. For example, if you were to calculate 1,000 iterations of fibonacci in a web worker, the numberic value could be returned, and the calling JavaScript code could then take that value and place it in the DOM.

Objects in JavaScript are fairly complex. Sometimes they can be created using the object literal syntax. Other times they can be created by instantiating a base class. And still, other times they can be modified, by setting property descriptors and setters and getters. When it comes to the structured clone algorithm, only the basic values of objects are retained.

Most notably, this means when you define a class of your own and pass an instance to be cloned, only the own properties of that instance will be cloned, and the resulting object will be an instance of `Object`. Properties defined in the prototype will not be cloned either. Even if you define `class Foo {}` both on the calling side and inside of the web worker, the value will still be an instance of `Object`. This is because there's no real way to guarantee that both sides of the clone are dealing with the exact same `Foo` class.

Certain objects will entirely refuse to cloned. For example, if you try to pass `window` from *main.js* to *worker.js*, or if you try to return `self` in the opposite direction, you may receive one of the following errors, depending on the

browser:

```
Uncaught DOMException: The object could not be cloned.
DataCloneError: The object could not be cloned.
```

There are some inconsistencies across JavaScript engines, so it's best to test your code in multiple browsers. For example, some browsers support cloning `Error` instances, and others do not. The general rule of thumb is that JSON-compatible objects should never be a problem, but more complex data might be. For that reason, passing around simpler data is usually best.

Shared Workers

A *shared worker* is another type of web worker, but what makes it special is that a shared worker can be accessed by different browser environments, such as different windows (tabs), across iframes, and even from different web workers. They also have a different `self` within the worker, being an instance of `SharedWorkerGlobalScope`. A shared worker can only be accessed by JavaScript running on the same origin. For example, a window running on *`http://localhost:5000`* cannot access a shared worker running on *`http://google.com:80`*.

Before diving into code, it's important to consider a few gotcha's. One thing that makes shared workers a little hard to reason about is that they aren't necessarily attached to a particular window (environment). Sure, they're initially spawned by a particular window, but after that they can end up "belonging" to multiple windows. That means that when the first window is closed, the shared worker is kept around.

> **TIP**
>
> Since shared workers don't belong to a particular window, one interesting question is where should `console.log` output go? As of Firefox v85, the output is associated with the first window that spawns the shared worker. Open another window and the first still gets the logs. Close the first window and the logs are now invisible. Open another window and the historical logs then appear in the newest window. Chrome v87, on the other hand, doesn't display shared worker logs at all. Keep this in mind when debugging.

DEBUGGING SHARED WORKERS

Both Firefox and Chrome offer a dedicated way to debug shared workers. In Firefox, visit `about:debugging` in the address par. Next, click This Firefox in the left column. Then, scroll down until you see the Shared Workers section with a list of shared worker scripts. In our case we see an Inspect button next to an entry for the *shared-worker.js* file. With Chrome, visit `chrome://inspect/#workers`, find the *shared-worker.js* entry, and then click the "inspect" link next to it. With both browsers you'll be taken to a dedicated console attached to the worker.

Shared workers can be used to hold semi-persistent state that is then maintained when other windows connect to it. For example, if window 1 tells the shared worker to write a value, then window 2 can ask the shared worker for that value back. Refresh window 1 and the value is still maintained. Refresh window 2 and it's also retained. Close window 1 and it's still retained. However, once you close or refresh the final window that is still using the service worker, the state will be lost and the shared worker script will be evaluated again.

WARNING

A shared worker JavaScript file is cached while multiple windows are using it; refreshing a page won't necessarily reload your changes. Instead, you'll need to close other open browser windows, then refresh the remaining window, to get the browser to run your new code.

With these caveats in mind you're now ready to build a simple application that uses shared workers.

Shared Worker Hello World

A shared worker is "keyed" based on its location in the current origin. For example, the shared worker you'll work with in this example is located somewhere like *`http://localhost:5000/shared-worker.js`*. Whether the worker is loaded from an HTML file located at `/red.html`, `/blue.html`, or even `/foo/index.html`, the shared worker instance will

always remain the same. There is a way to create different shared worker instances using the same JavaScript file, and that's covered in "Advanced Shared Worker Usage".

The relationship between the page and the worker that you are building is displayed in Figure 2-2.

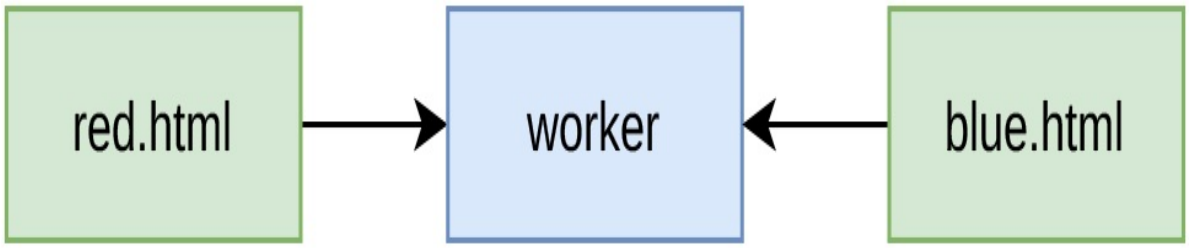

Figure 2-2. Shared Worker relationship

Now, it's time to create some files. For this example, create a directory named *ch2-shared-workers/*, and all the files necessary will live in this directory. Once that's done, create an HTML file containing the content in Example 2-4.

Example 2-4. ch2-shared-workers/red.html

```
<html>
  <head>
    <title>Shared Workers Red</title>
    <script src="red.js"></script>
  </head>
</html>
```

Much like the HTML file you created in the previous section, this one just sets a title and loads a JavaScript file. Once that's done, create another HTML file containing the content in Example 2-5.

Example 2-5. ch2-shared-workers/blue.html

```
<html>
  <head>
    <title>Shared Workers Blue</title>
    <script src="blue.js"></script>
  </head>
</html>
```

For this example you're going to work with two separate HTML files, each representing a new JavaScript environment that will be available on the same origin. Technically, you could have reused the same HTML file in both windows, but we want to make it very explicit that none of the state is going to

be associated with the HTML files or the main JavaScript files.

Next, you're ready to create the first JavaScript file loaded directly by an HTML file. Create a file containing the content in Example 2-6.

Example 2-6. ch2-shared-workers/red.js

```
console.log('red.js');

const worker = new SharedWorker('shared-worker.js'); ❶

worker.port.onmessage = (event) => { ❷
  console.log('EVENT', event.data);
};
```

❶ Instantiate the shared worker

❷ Note the `worker.port` property for communications

This JavaScript file is rather basic. What it does is instantiate a shared worker instance by calling `new SharedWorker()`. After that it adds a handler for message events that are emitted from the shared worker. When a message is received it is simply printed to the console.

Unlike with `Worker` instances, where you called `.onmessage` directly, with `SharedWorker` instances you'll make use of the `.port` property.

Next, copy and paste the *red.js* file that you created in Example 2-6 and name it *blue.js*. The content will remain exactly the same.

Finally, create a *shared-worker.js* file, containing the content in Example 2-7. This is where most of the magic will happen.

Example 2-7. ch2-shared-workers/shared-worker.js

```
const ID = Math.floor(Math.random() * 999999); ❶
console.log('shared-worker.js', ID);

const ports = new Set(); ❷

self.onconnect = (event) => { ❸
  const port = event.ports[0];
  ports.add(port);
  console.log('CONN', ID, ports.size);

  port.onmessage = (event) => { ❹
    console.log('MESSAGE', ID, event.data);
```

```
    for (let p of ports) { ❺
      p.postMessage([ID, event.data]);
    }
  };
};
```

❶ Random ID for debugging

❷ Singleton list of ports

❸ Connection event handler

❹ Callback when a new connection occurs

❺ Messages are dispatched to each window

The first thing that happens in this file is that a random ID value is generated. This value is printed in the console, and later passed to the calling JavaScript environments. It's not particularly useful with a real application, but it does a great job proving that state is retained, and when state is lost, when dealing with this shared worker.

Next, a singleton `Set` named `ports` is created.[2] This will contain a list of all of the ports that are made available to the worker. These port objects, both the `worker.port` available in the window and the `port` provided in a service worker, are an instance of the `MessagePort` class.

The final thing that happens in the outer scope of this shared worker file is that a listener for the *connect* event is established. This function is called every time a JavaScript environment creates a `SharedWorker` instance that references this shared worker. When this listener is called, an instnace of `MessageEvent` is provided as the argument.

There are several properties available on the *connect* event, but the most important one is the `ports` property. This property is an array that contains a single element which is a reference to the `MessagePort` instance that allows communication with the calling JavaScript environment. This particular port is then added to the `ports` set.

An event listener for the *message* event is also attached to the port. Much like the `onmessage` method you previously used with the `Worker` instance, this method is called when one of the external JavaScript environments calls the applicatble `.postMessage()` method. When a message is received, the code

prints the ID value and the data that was received.

The event listener also dispatches the message back to the calling environments. It does this by iterating the `ports` set, calling the `.postMessage()` method for each of the encountered ports. Since this method only takes a single argument, an array is passed in to sort of emulate multiple arguments. The first element of this array is the ID value again, and the second is the data that was passed in.

If you've worked with Web Sockets using Node.js before, then this code pattern should feel very familiar. With most popular Web Sockets packages, an event is triggered when a connection is made, and the connection argument can then have a message listener attached to it.

At this point you're ready to test your application again. First, run the following command inside of your *ch2-shared-workers/* directory, and then copy and paste the URL that is displayed:

```
$ npx serve .
```

Again, in our case, we're given the URL ***`http://localhost:5000`***. This time though, instead of opening the URL directly, you'll want to first open the web inspector in your browser, and then open a modified version of the URL.

Switch to your browser and open a new tab. It's fine if this opens your home page, a blank tab, or whatever your default page is. Then, open the web inspector again and navigate to the console tab. Once that's done, paste the URL that was given to you, but modify it to open the */red.html* page. The URL that you enter might look something like this:

```
http://localhost:5000/red.html
```

Press Enter to open the page. The `serve` package will probably redirect your browser from */red.html* to */red*, but that's fine.

Once the page has loaded, you should see the messages listed in Table 2-2 displayed in your console. If you open the inspector after loading the page then you probably won't see any logs, though doing so then refreshing the page should display the logs. Note that at the time of writing, only Firefox will display

messages generated in *shared-worker.js* — while Chrome will not.

Table 2-2. First window console output

Log	Location
red.js	red.js:1:9
shared-worker.js 592204	shared-worker.js:2:9
CONN 592204 1	shared-worker.js:9:11

In our case we can see that the *red.js* file was executed, that this particular *shared-worker.js* instance generated an ID of 592204, and that there is currently a single window connected to this shared worker.

Next, open another browser window. Again, open the web inspector first, switch to the console tab, and then paste the base URL that was provided by the `serve` command, and then add */blue.html* to the end of the URL. In our case the URL looks like this:

```
http://localhost:5000/blue.html
```

Press Enter to open the URL. Once the page loads, you should only see a single message printed in the console output stating that the *blue.js* file was executed. At this point it's not too interesting. But, switch back to the previous window you had opened for the *red.html* page. You should see that the new log listed in Table 2-3 has been added.

Table 2-3. First window console output, continued

Log	Location
CONN 278794 2	shared-worker.js:9:11

Now things are getting a little exciting. At this point, the shared worker environment has two references to a `MessagePort` instance pointing to two

separate windows. At the same time, two windows have references to `MessagePort` instances for the same shared worker.

At this point you're ready to send a message to the shared worker from one of the windows. Switch focus to the console window and type in the following command:

```
worker.port.postMessage('hello, world');
```

Press Enter to execute that line of JavaScript. You should see a message in the first console that is generated in the shared worker, a message in the first console from *red.js*, and a message in the second window's console from *blue.js*. In our case we see the outputs listed in Table 2-4.

Table 2-4. First and second window console output

Log	Location	Console
MESSAGE 278794 hello, world	shared-worker.js:12:13	1
EVENT Array [278794, "hello, world"]	red.js:6:11	1
EVENT Array [278794, "hello, world"]	blue.js:6:11	2

At this point you've successfully sent a message from the JavaScript environment available in one window, to the JavaScript environment in a shared worker, and then passed a message from the worker to two separate windows.

Advanced Shared Worker Usage

Shared workers are goverend by the same object cloning rules described in "Structured Clone Algorithm". Also, like their dedicated worker equivalent, shared workers also have access to the `importScripts()` function for loading external JavaScript files. As of Firefox v85/Chrome v87, Thomas personally finds Firefox more convenient to debug shared workers with due to the output of `console.log()` from the shared worker being available.

The shared worker instances do have access to a *connect* event, which can be handled with the `self.onconnect()` method. Notably missing, especially if

you're familiar with Web Sockets, is a *disconnect* or *close* event.

When it comes to creating a singleton collection of `port` instances, like in the sample code in this section, it's very easy to create a memory leak. In this case, just keep refreshing one of the windows, and each refresh adds a new entry to the set.

This is far from ideal. One thing you can do to address this is to add an event listener in your main JavaScript environments (i.e. *red.js* and _blue.js) that fires when the page is being torn down. Have this event listener pass a special message to the shared worker. Within the shared worker, when the message is received, remove the port from the list of ports. Here's an example of how to do this:

```
// main JavaScript file
window.addEventListener('beforeunload', () => {
  worker.port.postMessage('close');
});

// shared worker
port.onmessage = (event) => {
  if (event.data === 'close') {
    ports.delete(port);
    return;
  }
};
```

Unfortunately, there are still situations where a port can stick around. If the *beforeunload* event doesn't fire, or if an error happens when it's fired, or if the page crashes in an unanticipated way, this can lead to expired port references sticking around in the shared worker.

A more robust system would also need a way for the shared worker to occasionally "ping" the calling environments, sending a special messiage via `port.postMessage()`, and have the calling environments reply. With such an approach the shared worker can delete port instances if it doesn't receive a reply within a certain amount of time. But, even this approach isn't perfect, as a slow JavaScript environment can lead to long response times. Luckily, interacting with ports that no longer have a valid JavaScript associated with them doesn't have much of a side effect.

The full constructor for the `SharedWorker` class looks like this:

```
const worker = new SharedWorker(filename, nameOrOptions);
```

The signature is slightly different than when instantiating a `Worker` instance, notably that the second argument can either be an options object, or the name of the worker. Much like with a `Worker` instance, the name of the worker is available inside of the worker as `self.name`.

At this point you may be wondering how that works. For example, could the shared worker be declared in *red.js* with a name of “red worker” and in *blue.js* with a name of “blue worker”? In this case, two *separate* workers will be created, each with a different global environment, a different ID value, and the appropriate `self.name`.

You can think of these shared worker instances as being “keyed” by not only their URL but also their name. This may be why the signature changes between a `Worker` and a `SharedWorker`, as the name is much more important for the latter.

Other than the ability to replace the options argument with a string name, the options argument for a `SharedWorker` is exactly the same as it is for a `Worker`.

In this example, you’ve only created a single `SharedWorker` instance per window, assigned to `worker`, but there is nothing from stopping you from creating multiple instances. In fact, you can even create multiple shared workers that point to the same instance, assuming the URLs and names match. When this happens, both of the shared worker instance’s `.port` properties are able to receive messages.

These `SharedWorker` instances are definitely capable of maintaining state between page loads. You’ve been doing just that, with the `ID` variable holding a unique number, and `ports` containing a list of ports. This state is even persisted through refreshes as long as one window remains open, like if you were to refresh the *blue.html* page followed by the *red.html* page. However, that state would be lost if both pages were refreshed simultaneously, one closed and the other refreshed, or if both were closed. In the next section you’ll work with a

technology that can continue to maintain state—and run code—even when connected windows are closed.

Service Workers

A *service worker* functions as a sort of proxy that sits between one or more web pages running in the browser and the server. Since a service worker isn't associated with just a single web page, but potentially multiple pages, it's more similar to a shared worker than to a dedicated worker. They're even "keyed" in the same manner as shared workers. But, a shared worker can exist and run in the background even when a page isn't necessarily still open. Because of this you can think of a dedicated worker as being associated with 1 page, a shared worker as being associated with 1 or more pages, but a service worker as being associated with 0 or more pages. But, a shared worker doesn't magically spawn into existence. Instead, it does require a web page to be opened first to install the shared worker.

Shared workers are primarily intended for performing cache management of a website or a single page application. They are most commonly invoked when network requests are sent to the server, wherein an event handler inside of the service worker intercepts the network request. The service worker's claim to fame is that it can be used to return cached assets when a browser displays a web page but the computer it's running on no longer has network access. When the service worker receives the request it may consult a cache to find a cached resource, make a request to the server to retrieve some semblence of the resource, or even perform a heavy computation and return the result. While this last option makes it similar to the other web workers you've looked at, you really shouldn't use shared workers just for the purpose of offloading CPU intensive work to another thread.

Service workers expose a larger API than that of the other web workers, though their primary use-case is not for offloading heavy computation from the main thread. Service workers are certainly complex enough to have entire books dedicated to them. That said, as the primary goal of this book is to teach you about the multithreaded capabilities of JavaScript, we won't cover them in their entirety. For example, there's an entire Push API available for receiving

messages pushed to the browser from the server which won't be covered at all.

Much like with the other web workers, a service worker can't access the DOM. They also can't make blocking requests. For example, setting the third argument of `XMLHttpRequest#open()` to `false` is not allowed. Browsers will only allow service workers to run on a web page that has been served using the HTTPS protocol. Luckily for us, there is one notable exception, where the `localhost` host may load service workers using HTTP. This is to make local development easier. Firefox doesn't allow service workers when using its Private Browsing feature. Chrome, however, does allow service workers when using its Incognito feature. That said, a service worker instance can't communicate between a normal and Incognito window.

Both Firefox and Chrome have an Applications panel in the inspector that contains a Service Workers section. You can use this to both view any service workers associated with the current page, and to also perform a very important development action: unregister them. Unfortunately, as of the current browser versions, these browser panels don't provide a way to hop into the JavaScript inspectors for the service workers.

DEBUGGING SERVICE WORKERS

To get into the inspector panels for your service worker instances you'll need to go somewhere else. In Firefox, open the address bar and visit *about:debugging#/runtime/this-firefox*. Scroll down to the service workers and any workers you create today should be visible at the botttom. For Chrome, there are two different screens available for getting access to the browser's service workers. The better one, in Thomas's opinion, is located at *chrome://serviceworker-internals/*. It contains a listing of service workers, their status, and basic log output. The other one is at *chrome://inspect/#service-workers*, and contains a lot less information.

Now that you're aware of some of the gotchas with service workers, you're ready to build one out.

Service Worker Hello World

In this section you're going to build a very basic service worker that intercepts all HTTP requests sent from a basic web page. Most of the requests will pass through to the server unaltered. However, requests made to a specific resource will instead return a value that is calculated by the service worker itself. Most service workers would instead do a lot of cache lookups, but again, the goal is to show off service workers from a multithreaded point of view.

The first file you'll need is again an HTML file. Make a new directory named *ch2-service-workers/*. Then, inside of this directory, create a file with the content from Example 2-8.

Example 2-8. ch2-service-workers/index.html

```
<html>
  <head>
    <title>Dedicated Workers Example</title>
    <script src="main.js"></script>
  </head>
</html>
```

This is a rather basic file that just loads your application's JavaScript file, which comes next. Create a file named *main.js*, and add the content from Example 2-9 to it.

Example 2-9. ch2-service-workers/main.js

```
navigator.serviceWorker.register('/sw.js', { ❶
  scope: '/'
});

navigator.serviceWorker.oncontrollerchange = () => { ❷
  console.log('controller change');
};

async function makeRequest() { ❸
  const result = await fetch('/data.json');
  const payload = await result.json();
  console.log(payload);
}
```

❶ Registers service worker and defines scope.

❷ Listens for a `controllerchange` event.

❸ (Callout text to come.)

Now things are starting to get a little interesting. The first thing going on in this

file is that the service worker is created. Unlike the other web workers you worked with, you aren't using the `new` keyword with a constructor. Instead, this code depends on the `navigator.serviceWorker` object to create the worker. The first argument is the path to the JavaScript file that acts as the service worker. The second argument is an optional configuration object which supports a single `scope` property.

The `scope` represents the directory for the current origin wherein any HTML pages that are loaded in it will have their requests passed through the service worker. By default, the `scope` value is the same as the directory that the service worker is loaded from. In this case, the / value is relative to the *index.html* directory, and since *sw.js* is located in the same directory, we could have omitted the scope and it would behave exactly the same.

Once the service worker has been installed for the page, all outbound HTTP requests will get sent through the service worker. This includes requests made to different origins. Since the scope for this page is set to the uppermost directory of the origin, any HTML page that is opened in this origin will then have to make requests through the service worker for assets. If the `scope` had been set to */foo*, then a page opened at */bar.html* will be unaffected by the service worker, but a page at */foo/baz.html* would be affected.

The next thing that happens is that a listener for the `controllerchange` event is added to the `navigator.serviceWorker` object. When this listener fires, a message is printed to the console. This message is just for debugging when a service worker takes control of a page that has been loaded and which is within the scope of the worker.

Finally, a function named `makeRequest()` is defined. This function makes a GET request to the */data.json* path, decodes the respones as JSON, and prints the result. As you might have noticed, there aren't any references to that function. Instead, you'll manually run it in the console later to test the functionality.

With that file out of the way, you're now ready to create the service worker itself. Create a third file named *sw.js*, and add the content from Example 2-10 to it.

Example 2-10. ch2-service-workers/sw.js

```
let counter = 0;
```

```
self.oninstall = (event) => {
  console.log('service worker install');
};

self.onactivate = (event) => {
  console.log('service worker activate');
  event.waitUntil(self.clients.claim()); ❶
};

self.onfetch = (event) => {
  console.log('fetch', event.request.url);

  if (event.request.url.endsWith('/data.json')) {
    counter++;
    event.respondWith( ❷
      new Response(JSON.stringify({counter}), {
        headers: {
          'Content-Type': 'application/json'
        }
      })
    );
    return;
  }

  // fallback to normal HTTP request
  event.respondWith(fetch(event.request)); ❸
};
```

❶ Allows service worker to claim the opened *index.html* page.

❷ Override for when */data.json* is requested.

❸ Other URLs will fall back to a normal network request.

The first thing that happens in this file is that a global variable `counter` is initialized to zero. Later, when certain types of requests are intercepted, that number will increment. This is just an example to prove that the service worker is running; in a real application you should never store state that's meant to be persistent in this way. In fact, expect any service workers to start and stop fairly frequently, in a manner that's hard to predict and different depending on browser implementation.

After that we create a handler for the `install` event by assigning a function to `self.oninstall`. This function runs when this version of the service worker is installed for the very first time in the browser. Most real-world applications

will perform instantiation work at this stage. For example, there's an object available at `self.caches` which can be used to configure caches that store the result of network requests. However, since this basic application doesn't have much to do in the way of instantiation, it just prints a message and finishes.

Next up is a function for handling the `activate` event. This event is useful for performing cleanup work when new versions of the service worker are introduced. With a real-world application, its probably going to do work like tearing down old versions of caches.

In this case, the `activate` handler function is making a call to the `self.clients.claim()` method. Calling this allows the page instance which first created the service worker, i.e. the *index.html* page you'll open for the first time, to then get controlled by the service worker. If you didn't have this line, the page wouldn't be controlled by the service worker when first loaded. However, refreshing the page or opening *index.html* in another tab would then allow that page to be controlled.

The call to `self.clients.claim()` returns a promise. Sadly, event handler functions used in service workers are not themselves aware of promises. So, you can't, for example, assign an async function to the handlers and expect them to resolve properly. However, the `event` argument is an object with a `.waitUntil()` method, which does work with a promise. Once the promise provided to that method resolves, it will allow the `oninstall` and `onactivate` (and later `onfetch`) handlers to finish. By not calling that method, like in the `oninstall` handler, the step is considered finished once the function exits.

The last event handler is the `onfetch` function. This one is the most complex, and also the one that will be called the most throughout the lifetime of the service worker. This handler is called every time a network request is made by a web page under control of the service worker. It's called `onfetch` to signal that it correlates to the `fetch()` function in the browser, though it's almost a misnomer, as any network request will be passed through it. For example, if an image tag is later added to the page, the request would also trigger `onfetch`.

This function first logs a message to confirm that its being run, and also printing the URL that is being requested. Other information about the requested resource

is also available, such as headers and the HTTP method. In a real-world application this information can be used to consult with a cache to see if the resource already exists. For example, a GET request to a resource within the current origin could be served from the caache, but if it doesn't exist, it could be requested using the `fetch()` function, then inserted into the cache, then returned to the browser.

This basic example just takes the URL, and checks to see if it's for a URL that ends in */data.json*. If it is not, the `if` statement body is skipped, and the final line of the function is called. This line just takes the request object (which is an instance of `Request`), passes it to the `fetch()` method, which returns a promise, and passes that promise to `event.respondWith()`. The `fetch()` method will resolve an object which will then be used to represent the response, which is then provided to the browser. This is essentially a very basic HTTP proxy.

However, circling back to the */data.json* URl check, if it does pass, and then something more complicated happens. In that case the `counter` variable is incremented, and a new response is generated from scratch (which is an instance of `Response`). In this case, a JSON string is constructed that contains the `counter` value. This is provided as the first argument to `Response`, which represents the response body. The second argument contains meta information about the response. In this case the `Content-Type` header is set to `application/json`, which suggests to the browser that the response is a JSON payload.

Now that your files have been created, navigate to the directory where you created them using your console, and run the following command to start another web server:

```
$ npx serve .
```

Again, copy the URL that was provided, open a new web browser window, open the inspector, then paste the URL to visit the page. You should see a single message printed in your console at this point:

```
controller change              main.js:6:11
```

Next, browse to the list of service workers installed in your browser using the aforementioned technique. Within the inspector, you should see the previously logged messages; specifically you should see these two:

```
service worker install         sw.js:4:11
service worker activate        sw.js:8:11
```

Next, switch back to the browser window. While in the console tab of the inspector, run the following line of code:

```
makeRequest();
```

This runs the `makeRequest()` function, which triggers an HTTP GET request to */data.json* of the current origin. Once it completes, you should see the message `Object { counter: 1 }` displayed in your console. That message was generated using the service worker, and the request was never sent to the web server. If you switch to the network tab of the inspector, you should see what looks like an otherwise normal request to get the resource. If you click the request you should see that it replied with a 200 status code, and the `Content-Type` header should be set to `application/json` as well. As far as the web page is concerned, it did make a normal HTTP request. But, you know better.

Switch back to the service worker inspector console. In here, you should see that a third message has been printed containing the details of the request. On our machine we get the following:

```
fetch http://localhost:5000/data.json   sw.js:13:11
```

At this point you've successfully intercepted an HTTP request from one JavaScript environment, performed some computation in another environment, and returned the result back to the main environment. Much like with the other web workers, this calculation was done in a separate thread. Had the service worker done some very heavy and slow calculations, the web page would have been free to perform other actions while it waited for the response.

> **TIP**
>
> In your first browser window, you might have noticed an error that an attempt to download the *favicon.ico* file was made but failed. You might also be wondering why the shared worker console doesn't mention this file. That's because, at the point when the window was first opened, it wasn't yet under control of the service worker, so the request was made directly over the network, bypassing the worker. Debugging service workers can be confusing, and this is one of the caveats to keep in mind.

Now that you've built a working service worker, you're ready to learn about some of the more advanced features they have to offer.

Advanced Service Worker Concepts

Service workers are intended to only be used for performing asynchronous operations. Because of that, the `localStorage` API, which technically blocks when reading and writing, isn't available. However, the `indexedDB` API is available. When it comes to keeping track of state, you'll mostly be using `self.caches` and `indexedDB`. Again, keeping data in a global variable isn't going to be reliable. In fact, while debugging your service workers, you might find that they occasionally end up stopped, at which point you're not allowed to hop into the inspector. The browsers have a button that allow you to start the worker again, allowing you to hop back into the inspector. It's this stopping and starting that flushes out global state.

Service worker scripts are cached rather aggresively by the browser. When reloading the page, the browser may make a request for the script, but unless the script has changed, it won't be considered for being replaced. The Chrome browser does offer the ability to trigger an update to the script when reloading the page; to do this, navigate to the Application tab in the inspector, then click "Service Workers", then click the "Update on reload" checkbox.

Every service worker goes through a state change from the time of it's inception until the time it can be used. This state is available within the service worker by reading the `self.serviceWorker.state` property. Here's a list of the stages it goes through:

parsed

This is the very first state of the service worker. At this point the JavaScript contents of the file has been parsed. This is more of an internal state that you'll probably never encounter in your application.

installing

The installation has begun, but is not yet complete. This happens once per worker version. This state is active after `oninstall` is called and before the `event.respondWith()` promise has resolved.

installed

At this point the installation is complete. The `onactivate` handler is going to be called next. In my testing I find that the service workers jump from `installing` to `activating` so fast that I never see the `installed` state.

activating

This state happens when `onactivate` is called but the `event.respondWith()` promise hasn't yet resolved.

activated

The activation is complete, and the worker is ready to do it's thing. At this point `fetch` events will get intercepted.

redundant

At this point, a newer version of the script has been loaded, and the previous script is no longer necessary. This can also be triggered if the worker script download fails, contains a syntax error, or if an error is thrown.

Philosphically, service workers should be treated as a form of progressive enhancement. This means that any web pages using them should still behave as usual if the service worker isn't used at all. This is important as you might encounter a browser that doesn't support service workers, or the installation phase might fail, or privacy-conciencious users might disable them entirely. That means, if you're only looking to add multithreading capabilities to your

application, then choose once of the other web workers instead.

The global `self` object used inside of service workers is an instance of `ServiceWorkerGlobalScope`. The `importScripts()` function available in other web workers is abailable in this environment as well. Like the other workers, it's also possible to pass message into, and receive message from, a service worker. The same `self.onmessage` handler can be assigned. This can, perhaps, be used to signal to the service worker that it should perform some sort of cache invalidation. Again, messages passed in this way are subject to the same cloning algorithm discussed in "Structured Clone Algorithm".

While debugging your service workers, and the requests that are being made from your browser, you'll need to keep caching in mind. Not only can the service worker implement caches that you control programmatically, but the browser itself also still has to deal with regular network caching. This can mean request sent from your service worker to the server might not always be received by the server. For this reason, keep the `Cache-Control` and `Expires` headers in mind, and be sure to use intentional values.

There are many more features available to service workers than what are covered in this section. Mozilla, the company behind Firefox, was nice enough to put together a cookbook website full of common strategies when building out service workers. This website is available at serviceworkers.rs and we recommend checking it out if you're considering implementing service workers in your next web app.

Service workers, and the other web workers you've looked at, certainly come with a bit of complexity. Lucky for us, there are some convenient libraries available, and communication patterns that you can implement, to make managing them a little easier.

Patterns and Libraries

Each of the web workers covered in this chapter expose an interface for passing messages into, and receiving messages from, a separate JavaScript environment. This allows you to build applications that are capable of running JavaScript simultaneously across multiple cores.

However, you've really only worked with simple contrived examples so far, passing along simple strings and calling simple functions. When it comes to building larger applications it'll be important to pass messages along that can scale, run code in workers that can scale, and simplifying the interface when working with workers will also reduce potential errors.

The RPC Pattern

So far, you've only worked with passing basic strings along to workers. While this is fine for getting a feel for the capabilities of web workers, it's something that isn't going to scale well for a full application.

For example, let's assume you have a web worker that does a single thing, like sum all the square root values from 1 to 1,000,000. Well, you could just call the `postMessage()` for the worker, passing in nothing, then run the slow logic in the `onmessage` handler, and send the message back using the worker's `postMessage()` function. But, what if the worker also needs to calculate fibonacci? In that case you could pass in a string, one for `square_sum`, and one for `fibonacci`. But, what if you need arguments? Well, you could pass in `square_sum|1000000`. But, what if you need argument types? Maybe you get something like `square_sum|num:1000000`, etc. You can probably see what we're getting at.

The RPC (Remote Procedure Call) pattern is a way to take a representation of a function and its arguments, serialize them, and pass them to a remote destination to have them get executed. The string `square_sum|num:1000000` is actually a form of RPC that we just accidentally invented. Perhaps it could ultimately get converted into a function call like `squareNum(1000000)`, but that will be considered next in "The Command Dispatcher Pattern".

There's another bit of complexity that an application needs to worry about as well. If the main thread only sends a single message to a web worker at a time, then when a message is returned from the web worker, you know it's the response to the message. However, if you send multiple messages to a web worker at the same time, there's no easy way to correlate the responses. For example, imagine an application that sends two messages to a web worker, and receives two response:

```
worker.postMessage('square_sum|num:4');
worker.postMessage('fibonacci|num:33');

worker.onmessage = (result) => {
  // Which result belongs to which message?
  // '3524578'
  // 4.1462643
};
```

Luckily, there does exist a standard for passing messages around and fulfilling the RPC pattern that can be draw inspiration from. This standard is called JSON-RPC, and it's fairly trivial to implement. This standard defines JSON representations of request and response objects, "notification" objects, a way to define the method being called and arguments in the request, the result in the response, and a mechanism for associating requests and responses. It even supports error values and batching of requests. For this example you'll only work with a request and response.

Taking the two function calls from our example, the JSON-RPC version of those requests and responses might look like this:

```
// worker.postMessage
{"jsonrpc": "2.0", "method": "square_sum", "params": [4], "id": 1}
{"jsonrpc": "2.0", "method": "fibonacci", "params": [7], "id": 2}

// worker.onmessage
{"jsonrpc": "2.0", "result": "3524578", "id": 2}
{"jsonrpc": "2.0", "result": 4.1462643, "id": 1}
```

In this case there's now a clear correlation between the response messages and their request.

JSON-RPC is intended to use JSON as the encoding when serializing messages, particularly when sending messages over the network. In fact, those `jsonrpc` fields define the version of JSON-RPC that the message is adhering to, which is very important in a network setting. However, since web workers use the structured clone algorithm that allows passing JSON-compatible objects along, an app could just pass objects directly without paying the cost of JSON serialization and deserialization. Also, the `jsonrpc` fields might not be as important in the browser where you have tighter control of both ends of the communication channel.

With these `id` properties correlating request and response objects, it's possible to then correlate which message relates to which. You'll build a solution for correlating these two in "Putting it All Together". But, for now, you need to first determine which function to call when a message is received.

The Command Dispatcher Pattern

While the RPC pattern is useful for defining protocols, it doesn't necessarily provide a mechanism for determining what code path to execute on the receiving end. The Command Dispatcher pattern solves this, and is a way to take a serialized command, find the appropriate function, and then execute it, optionally passing in arguments.

This pattern is fairly straightforward to implement and doesn't require a whole lot of magic. First, we can assume that there are two variables that contain relevant information about the method or "command" that the code needs to run. The first variable is called `method` and is a string. The second variable is called `args` and is an array of values to be passed into the method. Assume these have been pulled from the RPC layer of the application.

The code that ultimately needs to run might live in different parts of the application. For example, maybe the square sum code lives in a third-party library, and the fibonacci code is something that you've declared more locally. Regardless of where that code lives, you'll want to make a single repository that maps these commands to the code that needs to be run. There are several ways to pull this off, for example by using a `Map` object, but since the commands are going to be fairly static a humble JavaScript object will suffice.

Another important concept is that only defined commands should be executed. If the caller wants to invoke a method that doesn't exist, an error should be gracefully generated that can be returned to the caller, without crashing the web worker. And, while the arguments could be passed in to the method as an array, it would be a much nicer interface if the array of arguments were spread out into normal function arguments.

Example 2-11 shows an example implementation of a command dispatcher that you can use in your applications:

Example 2-11. Example Command Dispatcher

```
const commands = { ❶
  square_sum(max) {
    let sum = 0;
    for (let i = 0; i < max; i++) sum += Math.sqrt(i);
    return sum;
  },
  fibonacci(limit) {
    let prev = 1n, next = 0n, swap;
    while (limit) {
      swap = prev; prev = prev + next;
      next = swap; limit--;
    }
    return String(next);
  }
};
function dispatch(method, args) {
  if (commands.hasOwnProperty(method)) { ❷
    return commands[method](...args); ❸
  }
  throw new TypeError(`Command ${method} not defined!`);
}
```

❶ The definition of all supported commands.

❷ Check to see if command exists.

❸ Arguments are spread and method is invoked.

This code defines an object named `commands` that contains the entire collection of commands that are supported by the command dispatcher. In this case the code is inlined but it's absolutely fine, an even encouraged, to reach out to code that lives elsewhere.

The `dispatch()` function takes two arguments, the first being the name of the method and the second being the array of arguments. This function can be invoked when the web worker receives an RPC message representing the command. Within this function the first step is to check if the method exists. This is done using `commands.hasOwnProperty()`. This is much safer than calling `method in commands` or even `commands[method]` since you don't want non-command properties like *`__proto__`* being called.

If the command is determined to exist then the command arguments are spread out, so that the first array element is the first argument, etc., and then the function is called with the arguments, and the result of the call is returned. However, if the command doesn't exist, then a `TypeError` is thrown.

This is about as basic of a command dispatcher that you can create. Other, more advanced dispatchers might do things like type checking, where the arguments validated to adhere to a certain primitive type or that objects follow the appropriate shape, throwing errors generically so that the command method code doesn't need to do it.

These two patterns will definitely help your applications out, but the interface can be streamlined even more.

Putting it All Together

With JavaScript applications, we often think about performing work with outside services. For example, maybe we make a call to a database or maybe we make an HTTP request. When this happens we need to wait for a response to happen. Ideally, we can either provide a callback or treat this lookup as a promise. While the web worker messaging interface doesn't make this straight forward, we can definitely build it out by hand.

It would also be nice to have a more symmetrical interface within a web workers, perhaps by making use of an asynchronous function, one where the resolved value is automatically sent back to the calling environment, without the need to manually call `postMessage()` within the code.

In this section you'll do just that. You'll combine the RPC pattern and the Command Dispatcher pattern and end up with an interface that makes working with web workers much like working with other external libraries you may be more familiar with. For this example we'll use a dedicated worker, but the same thing could be built with a shared worker or service worker.

First, create a new directory named *ch2-patterns/* to house the files you're going to create. In here first create another basic HTML file named *index.html* containing the contents of Example 2-12.

Example 2-12. ch2-patterns/index.html

```
<html>
  <head>
    <title>Worker Patterns</title>
    <script src="rpc-worker.js"></script>
    <script src="main.js"></script>
  </head>
</html>
```

This time the file is loading two JavaScript files. The first is a new library, and the second is the main JavaScript file, which you'll now create. Make a file named *main.js*, and add the contents of Example 2-13 to it.

Example 2-13. ch2-patterns/main.js

```
const worker = new RpcWorker('worker.js');

Promise.allSettled([
  worker.exec('square_sum', 1_000_000),
  worker.exec('fibonacci', 1_000),
  worker.exec('fake_method')
]).then(([square_sum, fibonacci, fake]) => {
  console.log('square sum', square_sum);
  console.log('fibonacci', fibonacci);
  console.log('fake', fake);
});
```

This file represents application code using these new design patterns. First, a worker instance is created, but not by calling one of the web worker classes you've been working with so far. Instead, the code instantiates a new `RpcWorker` class. This class is going to be defined soon.

After that, three calls to different RPC methods are made by calling `worker.exec`. The first one is a call to the `square_sum` method, the second is to the `fibonacci` method, and the third is to a method that doesn't exist called `fake_method`. The first argument is the name of the method, and all the following arguments end up being the arguments that are passed to the method.

The `exec` method returns a promise, one that will resolve if the operation succeeds and will reject if the operation fails. With this in mind, each of the promises has been wrapped into a single `Promise.allSettled()` call. This will run all of them and then continue the execution once each is complete—regardless of success or failure. After that the result of each operation is printed.

Next, create a file named *rpc-worker.js*, and add the contents of Example 2-14 to it.

Example 2-14. ch2-patterns/rpc-worker.js (part 1)

```
class RpcWorker {
  constructor(path) {
    this.next_command_id = 0;
    this.in_flight_commands = new Map();
    this.worker = new Worker(path);
```

```
    this.worker.onmessage = this.onMessageHandler.bind(this);
  }
```

This first part of the file starts the `RpcWorker` class and defines the constructor. Within the constructor a few properties are initialized. First, the `next_command_id` is set to zero. This value is used as the JSON-RPC-style incrementing message identifier. This is used to correlate the request and response objects.

Next, a property named `in_flight_commands` is initialized to an empty `Map`. This contains entries keyed by the command ID, with a value that contains a promise's resolve and reject functions. The size of this map grows with the number of parallel messages sent to the worker and shrinks as their correlating messages are returned.

After that, a dedicated worker is instantiated and assigned to the `worker` property. This class effectively encapsulates a `Worker` instance. After that the `onmessage` handler of the worker is configured to call the `onMessageHandler` for the class (defined in the next chunk of code). The `RpcWorker` class doesn't extend `Worker` as it doesn't really want to expose functionality of the underlying web worker, instead creating a completely new interface.

Continue modifying the file by adding the content from Example 2-15 to it.

Example 2-15. ch2-patterns/rpc-worker.js (part 2)

```
  onMessageHandler(msg) {
    const { result, error, id } = msg.data;
    const { resolve, reject } = this.in_flight_commands.get(id);
    this.in_flight_commands.delete(id);
    if (error) reject(error);
    else if (result) resolve(result);
  }
```

This chunk of the file defines the `onMessageHandler` method, which runs when the dedicated worker posts a message. This code assumes that a JSON-RPC-like message is passed from the web worker to the calling environment, and so, it first extracts the `result`, `error`, and `id` values from the response.

Next, it consults the `in_flight_commands` map to find the matching ID value, retrieving the appropriate rejection and resolving functions, deleting the

entry from the list in the process. If the `error` value was provided then the operation is considered a failure and the `reject()` function is called with the erroneous value. Otherwise, the `resolve()` function is called with the result of the operation.

For a production-ready version of this library you would also want to support a timeout value for these operations. Theoretically, it's possible for an error to be thrown in such a way, or for a promise to never end up resolving in the worker, and the calling environment would want to reject the promise and also clear the data from the map. Otherwise the application might end up with a memory leak.

Finally, finish up this file by adding the remaining content from Example 2-16 to it.

Example 2-16. ch2-patterns/rpc-worker.js (part 3)

```
  exec(method, ...args) {
    const id = ++this.next_command_id;
    let resolve, reject;
    const promise = new Promise((res, rej) => {
      resolve = res;
      reject = rej;
    });
    this.in_flight_commands.set(id, { resolve, reject });
    this.worker.postMessage({ method, params: args, id });
    return promise;
  }
}
```

This last chunk of the file defines the `exec()` method, which is called when the application wants to execute a method in the web worker. The first thing that happens is that a new ID value is generated. Next, a promise is created, which will later be returned by the method. The `reject` and `resolve` functions for the promise are pulled out and are added to the `in_flight_commands` map, associated with the ID value.

After that, a message is posted to the worker. The object that is passed into the worker is an object roughly adhering to the JSON-RPC shape. It contains the `method` property, a `params` property which is the remaining arguments in an array, and the `id` value which was generated for this particular command execution.

This is a fairly common pattern, useful for associating outgoing asynchronous

messages with incoming asynchronous messages. You might find yourself implementing a similar pattern if you needed to, say, put a message onto a network queue and later receive a message. But, again, it does have memory implications.

With the RPC worker file out of the way you're ready to create the last file. Make a file named *worker.js*, and add the contents of Example 2-17 to it.

Example 2-17. ch2-patterns/worker.js

```
const sleep = (ms) => new Promise((res) => setTimeout(res, ms));

function asyncOnMessageWrap(fn) { ❶
  return async function(msg) {
    postMessage(await fn(msg.data));
  }
}

const commands = {
  async square_sum(max) {
    await sleep(Math.random() * 100); ❷
    let sum = 0; for (let i = 0; i < max; i++) sum += Math.sqrt(i);
    return sum;
  },
  async fibonacci(limit) {
    await sleep(Math.random() * 100);
    let prev = 1n, next = 0n, swap;
    while (limit) { swap = prev; prev = prev + next; next = swap; limit-
-; }
    return String(next);
  }
};

self.onmessage = asyncOnMessageWrap(async (rpc) => { ❸
  const { method, params, id } = rpc;

  if (commands.hasOwnProperty(method)) {
    const result = await commands[method](...params);
    return { result, id }; ❹
  }

  return { ❺
    error: {
      code: -32601,
      message: `method ${method} not found`
    },
    id
  };
```

```
});
```

❶ A basic wrapper to convert `onmessage` to an async function.

❷ Artificial random slowdowns are added to the commands.

❸ The `onmessage` wrapper is injected.

❹ A successful JSON-RPC-like message is resolved on success.

❺ An erroneous JSON-RPC-like message is rejected if method doesn't exist.

This file has a lot going on. First, the `sleep` function is just a promise equivalent version of `setTimeout()`. The `asyncOnmessageWrap()` is a function that can wrap an async frunction, and be assigned the `onmessage` handler. This is a convenience to pull out the data property of the incoming message, pass it to the function, await the result, then pass the result to `postMessage()`.

After that the `commands` object from before has made its return. This time, though, artificial timeouts have been added and the functions have been made into async functions. This lets the methods emulate an otherwise slow asynchronous process.

Finally, the `onmessage` handler is assigned using the wrapper function. The code inside of it takes the incoming JSON-RPC-like message and pulls out the `method`, `params`, and `id` properties. Much like before, the commands collection is consulted to see if it has the method. If it doesn't, a JSON-RPC-like error is returned. The `-32601` value is a magic number defined by JSON-RPC to represent a method that doesn't exist. When the command does exist, the command method is executed, then the resolved value is coerced into a JSON-RPC-like successful message and returned.

Once you've got the file created, switch to your browser and open the inspector. Then, launch the web server again using the following command from within the *ch2-patterns/* directory:

```
$ npx serve .
```

Next, switch back to browser and paste in the URL from the output. You won't see anything interesting on the page but in the console you should see the

following messages:

```
square sum    Object { status: "fulfilled", value: 666666166.4588418 }
fibonacci     Object { status: "fulfilled", value: "4346655768..." }
fake          Object { status: "rejected", reason: {...} }
```

In this case you can see that both the `square_sum` and `fibonacci` calls ended successfully, while the `fake_method` command resulted in failure. More importantly, under the hood, the calls to the methods are resolving in different orders, but thanks to the incrementing ID values the response are always properly correlated to their requests.

---

1 There is a small caveat with `RegExp` instances. They contain a `.lastIndex` property, which is used when running a regular expression multiple times over the same string to know where the expression last ended. This property is not passed along.

2 As of Firefox v85, regardless of how many entries are in the `ports` set, calling `console.log(ports)` will always display a single entry. For now, to debug the size, call `console.log(ports.size)` instead.

About the Authors

Thomas Hunter II has contributed to dozens of enterprise Node.js services and has worked for a company dedicated to securing Node.js. He has spoken at several conferences on Node.js and JavaScript, is JSNSD/JSNAD certified, and is an organizer of NodeSchool SF. Thomas has published four books including *Distributed Systems with Node.js* by O'Reilly.

Bryan English is an open source JavaScript and Rust programmer and enthusiast and has worked on large enterprise systems, instrumentation, and application security. Currently he's a Senior Open Source Software engineer at Datadog. He's used Node.js both professionally and in personal projects since not long after its inception. He is also a Node.js core collaborator and has contributed to Node.js in many ways through several of its various Working Groups.

iii. Putting it All Together