JavaScript has evolved to adopt concurrent capabilities—one of the reasons why it is still at the forefront of modern web development. This book helps you dive into concurrent JavaScript and demonstrates how to apply its core principles, key techniques, and tools to a range of complex development challenges. Built around the three core principles of concurrency—parallelism, synchronization, and conservation—you will learn everything you need to unlock a more efficient and dynamic JavaScript in order to lay the foundations of even better user experiences.

Throughout the book you will learn how to put these principles into action by using a range of development approaches. Covering everything from JavaScript promises, web workers, and generators to functional programming techniques, everything you learn will have a real impact on the performance of your applications. You will also learn how to move between the client and server for a frictionless and fully realized approach to development. With further guidance on concurrent programming with NodeJS, JavaScript Concurrency is committed to making you a better web developer.

Who this book is written for

This book is written for any JavaScript developer who wants to learn how to write more efficient, powerful, and maintainable applications that utilize the latest developments in JavaScript.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Why JavaScript Concurrency?'
- A synopsis of the book’s content
- More information on JavaScript Concurrency
About the Author

Adam Boduch has been involved with large-scale JavaScript development for nearly 10 years. Before moving to the front-end, he worked on several large-scale cloud computing products, using Python and Linux. No stranger to complexity, Adam has practical experience with real-world software systems, and the scaling challenges they pose. He is the author of several JavaScript books, including JavaScript at Scale, Packt Publishing, and is passionate about innovative user experiences and high performance.
Preface

It wasn't long ago when I would dread using many of the web applications I had come to depend on. When they worked, they were fantastic; when they didn't, it was a nightmare. Especially frustrating was the clear fact that there weren't any bugs in the JavaScript code driving the application. No, the problem was that often there was too much code running, perhaps because it was processing a large data set. The end result was always the same: the UI would freeze, and I would helplessly curse the Web.

Nowadays, this doesn't happen as frequently. We've fixed many of the common JavaScript problems from years ago. Something that hasn't caught on as fast as I had hoped is concurrency in JavaScript. There are little bits of concurrency sprinkled all throughout our applications, but seldom do we see truly concurrent JavaScript code.

Let's change the status quo.

What this book covers

Chapter 1, Why JavaScript Concurrency?, is an introduction to concurrency in JavaScript.

Chapter 2, The JavaScript Execution Model, takes you through the mechanisms that run our JavaScript code.

Chapter 3, Synchronizing with Promises, looks at synchronization techniques using promises.

Chapter 4, Lazy Evaluation with Generators, will get your grips to conserving resources by computing lazily.

Chapter 5, Working with Workers, looks at achieving true parallelism in JavaScript.

Chapter 6, Practical Parallelism, will help you in identifying the right parallelization problems to solve.
Preface

Chapter 7, *Abstracting Concurrency*, will get your hands dirty writing concurrent code that reads like regular code.

Chapter 8, *Evented IO with NodeJS*, will show you how concurrency semantics work in this environment.

Chapter 9, *Advanced NodeJS Concurrency*, is learning about specific Node concurrency issues.

Chapter 10, *Building a Concurrent Application*, is all about putting it all together.
JavaScript is not a language associated with concurrency. In fact, it's frequently associated with the exact opposite—concurrency challenges. This has changed a lot over the past few years, especially with new language features in ES 2015. Promises have been used in JavaScript for many years; only now, they're a native type. Generators are another addition to the language that changes the way we think about concurrency in JavaScript. Web workers have been in browsers for several years, and yet, we don't see them used that often. Perhaps, it has less to do with workers and more about our understanding of the role that concurrency plays in our applications.

The aim of this chapter is to explore some general concurrency ideas, starting with what, exactly, concurrency is. If you don't have any sort of concurrent programming in your background, it's fine because this chapter is a perfect starting point for you. If you've done concurrent programming in the past using JavaScript or some other language, think of this chapter as a refresher, only with JavaScript as the context.

We'll wrap up this chapter with some overarching concurrency principles. These are valuable programming tools that we should keep in the back of our heads while writing concurrent code. Once we have learned to apply these principles, they'll tell us whether we're on the right track with our concurrency design, or that we need to take a step back and ask ourselves what we're really trying to achieve. These principles take a top-down approach to the design of our application. This means that they're applicable from the very start, even before we've started writing any code. Throughout the book, we'll be referring to these principles, so if you only read one section in this chapter, make sure it's *Concurrency principles* at the end.
Synchronous JavaScript

Before we start conjuring large-scale concurrent JavaScript architectures, let’s shift our attention to the good old synchronous JavaScript code that we’re all familiar with. These are the blocks of JavaScript code that are called as the result of a click event, or run as the result of loading a webpage. Once they start, they don't stop. That is to say, they’re run-to-completion. We’ll dig into run-to-completion a little more in the following chapter.

We'll occasionally see the term synchronous and serial used interchangeably throughout the chapters. They’re both referring to code statements that run one after another until there’s nothing more to run.

Despite JavaScript being designed as a single-threaded, run-to-completion environment, the nature of the web complicates this. Think about the web browser, and all it's moving parts. There’s the Document Object Model (DOM) for rendering user interfaces and XMLHttpRequest (XHR) objects for fetching remote data sources, to name a couple. Let’s take a look at the synchronous nature of JavaScript and the asynchronous nature of the web.

Synchronicity is easy to understand

When code is synchronous, it's easier to understand. It's easier to mentally map the instructions that we're seeing on the screen to sequential steps in our heads; do this, then do that; check this, if true, do that, and so on. This type of serial processing is easy enough to understand, because there aren't any surprises, assuming the code isn't completely horrible. Here's an example of how we might visualize a chunk of synchronous code:

![Diagram of synchronous code]

First Step

Second Step

[condition]

[else]

Third Step

Fourth Step
Concurrent programming, on the other hand, isn't so easy to grasp. This is because there's no linear logic for us to follow in our code editors. Instead, we constantly jump around, trying to map what this piece of code is doing relative to that piece of code. Time is an important factor with concurrent designs; it is something that goes against the brain's natural way of comprehending code. When we read code, we naturally execute it in our heads. This is how we figure out what it's doing. This approach falls apart when the actual execution doesn't line up with what's in our head. Normally, code reads like a book—concurrent code is like a book where the pages are numbered, but out of order. Let's take a look at some trivial pseudo JavaScript code:

```javascript
var collection = [ 'a', 'b', 'c', 'd' ];
var results = [];

for (let item of collection) {
  results.push(String.fromCharCode(item.charCodeAt(0)));
}
//  [ 'b', 'c', 'd', 'e' ]
```

In traditional multi-threading environments, a thread is something that runs asynchronously with other threads. We use threads to take advantage of multiple CPUs found on most systems today, resulting in better performance. However, this comes at a cost because it forces us to rethink how our code is executed at runtime. It's no longer the usual step by step execution. This code could be running alongside other code in another CPU, or it could be competing with other threads for time on the same CPU.

A lot of simplicity goes away when we introduce concurrency to synchronous code—it's the code equivalent of brain freeze. This is why we write concurrent code: code that makes an upfront assumption of concurrency. We'll elaborate on this concept as we progress through the book. With JavaScript, it's important to assume a concurrent design, because that's the way the web works.

**Asynchronous is inevitable**

The reason that concurrency in JavaScript is such an important idea is because the web is a concurrent place, both from a very high level and an implementation detail level. In other words, the web is concurrent because at any given point in time, there's oodles of data flowing over the miles of fiber, which encase the globe. It has to do with the applications themselves that are deployed to web browsers, and how the back-end servers handle the litany of requests for data.
Why JavaScript Concurrency?

Asynchronous browsers
Let's take a closer look at the browser and the kinds of asynchronous actions found there. When a user loads a webpage, one of the first actions that the page will perform is to download and evaluate our JavaScript code that goes with the page. This in itself is an asynchronous action, because while our code downloads, the browser will continue doing other things, such as rendering page elements.

Another asynchronous data source that arrives over the network is the application data itself. Once our page has loaded and our JavaScript code starts running, we'll need to display some data for the user. This is actually one of the first things that our code will do so that the user has something to look at right away. Again, while we're waiting on this data to arrive, the JavaScript engine will move our code right along to it's next set of instructions. Here's a request for remote data that doesn't wait for the response before continuing on with executing code:

![Diagram of asynchronous process]

After the page elements have all been rendered and populated with data, the user starts interacting with our page. This means events are dispatched—clicking an element dispatches a click event. The DOM environment, where these events are dispatched from, is a sand-boxed environment. This means that within the browser, the DOM is a subsystem, separate from the JavaScript interpreter, which runs our code. This separation makes certain JavaScript concurrency scenarios especially difficult. We'll cover these in depth in the next chapter.

With all these sources of asynchronicity, it's no wonder that our pages can become bloated with special case handling to deal with the edge cases that inevitably pop up. Thinking asynchronously isn't natural, so this type of monkey-patching is the likely result of thinking synchronously. It's best to embrace the asynchronous nature of the web. After all, a synchronous web can lead to unbearable user experiences. Now, let's dig a little further into the types of concurrency we're likely to face in our JavaScript architectures.
Types of concurrency

JavaScript is a run-to-completion language. There's no getting around it, despite any concurrency mechanisms that are thrown on top of it. In other words, our JavaScript code isn't going to yield control to another thread in the middle of an if statement. The reason this matters is so that we can pick a level of abstraction that makes sense in helping us think about JavaScript concurrency. Let's look at the two types of concurrent actions found in our JavaScript code.

Asynchronous actions

A defining characteristic of asynchronous actions is that they do not block other actions that follow. Asynchronous actions don't necessarily mean fire-and-forget. Rather, when the part of the action we're waiting on completes, we run a callback function. This callback function falls out of sync with the rest of our code; hence, the term asynchronous.

In web front-ends, this generally means fetching data from a remote service. These fetching actions are relatively slow, because they have to traverse the network connection. It makes sense for these actions to be asynchronous, just because our code is waiting on some data to return so that it can fire a callback function, doesn't mean the user should have to sit around and wait too. Furthermore, it's unlikely that any screen that the user is currently looking at depends on only one remote resource. So, serially processing multiple remote fetch requests would have a detrimental effect on the user experience.

Here's a general idea of what asynchronous code looks like:

```javascript
var request = fetch('/foo');

request.addEventListener((response) => {
  // Do something with "response" now that it has arrived.
});

// Don't wait with the response, update the DOM immediately.
updateUI();
```

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Why JavaScript Concurrency?

We’re not limited to fetching remote data, as the single source of asynchronous actions. When we make network requests, these asynchronous control flows actually leave the browser. But what about asynchronous actions that are confined within the browser? Take the `setTimeout()` function as an example. It follows the same callback pattern that's used with network fetch requests. The function is passed a callback, which is executed at a later point. However, nothing ever leaves the browser. Instead, the action is queued behind any number of other actions. This is because asynchronous actions are still just one thread of control, executed by one CPU. This means that as our applications grow in size and complexity, we're faced with a concurrency scaling issue. But then, maybe asynchronous actions weren't meant to solve the single CPU problem.

Perhaps a better way to think about asynchronous actions performed on a single CPU is to picture a juggler. The juggler's brain is the CPU, coordinating his motor actions. The balls that get tossed around is the data our actions operate on. There's only two fundamental actions we care about—toss and catch:

![Diagram of juggler actions](image)

Since the juggler only has one brain, he can't possibly devote his mental capacity to perform more than one task at a time. However, the juggler is experienced and knows he doesn't need more than a tiny fraction of attention given to the toss or catch actions. Once the ball is in the air, he's free to return his attention to catching the ball that's about to land.

To anyone observing this juggler in action, it appears as though he's paying full attention to all six balls, when in reality, he's ignoring five of them at any point in time.

**Parallel actions**

Like asynchronicity, parallelism allows control flow to continue without waiting on actions to complete. Unlike asynchronicity, parallelism depends on hardware. This is because we can't have two or more flows of control taking place in parallel on a single CPU. However, the main aspect that sets parallelism apart from asynchronicity is the rationale for using it. The two approaches to concurrency solve different problems, and both require different design principles.
At the end of the day, we want to perform actions in parallel that would otherwise be time prohibitive, if performed synchronously. Think about a user who is awaiting three expensive actions to complete. If each takes 10 seconds to complete (an eternity on a UX timescale), then this means the user will have to wait for 30 seconds. If we're able to perform these tasks in parallel, we can bring the aggregate wait time closer to 10 seconds. We get more for less, leading to a performant user interface.

None of this is free. Like asynchronous actions, parallel actions lead to callbacks as a communication mechanism. In general, designing for parallelism is hard, because in addition to communicating with worker threads, we have to worry about the task at hand, that is, what are we hoping to achieve by using worker threads? And how do we break down our problem into smaller actions? The following is a rough idea of what our code starts to look like when we introduce parallelism:

```javascript
var worker = new Worker('worker.js');
var myElement = document.getElementById('myElement');

worker.addEventListener('message', (e) => {
  myElement.textContent = 'Done working!';
});

myElement.addEventListener('click', (e) => {
  worker.postMessage('work');
});
```

Don't worry about the mechanics of what's happening with this code, as they'll all be covered in depth later on. The takeaway is that as we throw workers into the mix, we add more callbacks to an environment that's already polluted with them. This is why we have to design for parallelism in our code, which is a major focus of this book, starting in Chapter 5, Working with Workers.

Let's think about the juggler analogy from the preceding section. The toss and catch actions are performed asynchronously by the juggler; that is, he has only one brain/CPU. But suppose the environment around us is constantly changing. There's a growing audience for our juggling act and a single juggler can't possibly keep them all entertained:
Why JavaScript Concurrency?

The solution is to introduce more jugglers to the act. This way we add more computing power capable, of performing multiple toss and catch actions in the same instant. This simply isn't possible with a single juggler running asynchronously.

We're not out of the woods yet, because we can't just have the newly-added jugglers stand in one place, and perform their act the same way our single juggler did. The audience is larger, more diverse, and needs to be entertained. The jugglers need to be able to handle different items. They need to move around on the floor so that the various sections of the audience are kept happy. They might even start juggling with each other. It's up to us to produce a design that's capable of orchestrating this juggling act.

**JavaScript concurrency principles: Parallelize, Synchronize, Conserve**

Now that we've been through the basics of what concurrency is, and its role in front-end web development, let's look at some fundamental concurrency principles of JavaScript development. These principles are merely tools that inform our design choices when we write concurrent JavaScript code.

When we apply these principles, they force us to step back and ask the appropriate questions before we move forward with implementation. In particular, they're the why and how questions:

- Why are we implementing this concurrent design?
- What do we hope to get out of it that we couldn't otherwise get out of a simpler synchronous approach?
- How do we implement concurrency in a way that's unobtrusive to the features of our applications?

Here's a reference visualization of each concurrency principle, feeding on one another during the development process. And with that, we'll turn our attention to each principle for further exploration:
Parallelize

The parallelize principle means taking advantage of modern CPU capabilities to compute results in less time. This is now possible in any modern browser or NodeJS environment. In the browser, we can achieve true parallelism using web workers. In Node, we can achieve true parallelism by spawning new processes. Here’s what the CPU looks like from the browser’s perspective:

With the goal being more computations in less time, we must now ask ourselves why we want to do this? Besides the fact that raw performance is super cool in its own right, there has to be some tangible impact for the user. This principle makes us look at our parallel code and ask—what does the user get out of this? The answer is that we can compute using larger data sets as input, and have a smaller opportunity of an unresponsive user experience due to long-running JavaScript.

It’s important to scrutinize the tangible benefit of going parallel because when we do so, we add complexity to our code that wouldn’t otherwise be there. So if the user sees the same result no matter what we do, the parallelize principle probably isn’t applicable. On the other hand, if scalability is important and there’s a strong possibility of growing data set sizes, the trade off of code simplicity for parallelism is probably worthwhile. Here’s a checklist to follow when thinking about the parallelize principle:

- Does our application perform expensive computations against large data sets?
- As our data sets grow in size, is there potential for processing bottlenecks that negatively impact the user experience?
- Do our users currently experience bottlenecks in our application’s performance?
- How feasible is parallelism in our design, given other constraints? What are the trade-offs?
- Do the benefits of our concurrency implementation outweigh the overhead costs, either in terms of user-perceived latency or in terms of code maintainability?
Why JavaScript Concurrency?

Synchronize
The synchronize principle is about the mechanisms used to coordinate concurrent actions and the abstractions of those mechanisms. Callback functions are a JavaScript notion with deep roots. It's the obvious tool of choice when we need to run some code, but we don't want to run it now. We want to run it when some condition becomes true. By and large, there's nothing inherently wrong with this approach. Used in isolation, the callback pattern is probably the most succinct, readable concurrency pattern that we can use. Callbacks fall apart when there are plenty them, and lots of dependencies between them.

The Promise API
The Promise API is the core JavaScript language construct, introduced in ECMAscript 6 to address the synchronization woes faced by every application on the planet. It's a simple API that actually makes use of callbacks (yes, we're fighting callbacks with callbacks). The aim of promises isn't to eliminate callbacks, it's to remove the unnecessary callbacks. Here's what a promise that's used to synchronize two network fetch calls looks like:

What's crucial about promises is that they're a generic synchronization mechanism. This means that they're not specifically made for network requests, web workers, or DOM events. We, the programmers, have to wrap our asynchronous actions with promises and resolve them as necessary. The reason why this is a good thing is because the callers that rely on the promise interface don't care about what's going on inside the promise. As the name implies, it's a promise to resolve a value at some point. This could be in 5 seconds or immediately. The data can come from a network resource or a web worker. The caller doesn't care, because it makes an assumption of concurrency, which means we can fulfill it any in way we like without breaking the application. Here's a modified version of the preceding diagram, which will give us a taste of what promises make possible:
When we learn to treat values as values at some point in the future, concurrent code is suddenly much more approachable. Promises, and similar mechanisms, can be used to synchronize just network requests, or just web worker events. But they're real power is using them to write concurrent applications, where concurrency is the default. Here's a checklist to reference when thinking about the synchronize principle:

- Does our application heavily rely on callback functions as a synchronization mechanism?
- Do we often have to synchronize more than one asynchronous event such as network requests?
- Do our callback functions contain more synchronization boilerplate code than application code?
- What kind of assumptions does our code make about the concurrency mechanisms that drive asynchronous events?
- If we had a magic kill concurrency button, would our application still behave as expected?

**Conserve**

The conserve principle is about saving on compute and memory resources. This is done by using lazy evaluation techniques. The name lazy stems from the idea that we don't actually compute a new value until we're sure we actually need it. Imagine an application component that renders page elements. We can pass this component the exact data that it needs to render. This means that several computations take place before the component actually needs it. It also means that the data that's used needs to be allocated into memory, so that we can pass it to the component. There's nothing wrong with this approach. In fact, it's the standard way to pass data around in our JavaScript components.
Why JavaScript Concurrency?

The alternative approach uses lazy evaluation to achieve the same result. Rather than computing the values to be rendered, then allocating them in a structure to be passed, we compute one item, and then render it. Think of this as a kind of cooperative multi-tasking, where the larger action is broken down into smaller tasks that pass the focus of control back and forth.

Here's an eager approach to compute data and pass it to the component that renders UI elements:

![Diagram of eager approach]

There's two undesirable aspects to this approach. First, the transformation happens up-front, which could be a costly computation. What happens if the component is unable to render it for whatever reason—due to some constraint? Then we've performed this computation to transform data that wasn't needed. As a corollary, we've allocated a new data structure for the transformed data so that we could pass it to our component. This transient memory structure doesn't really serve any purpose, as it's garbage-collected immediately. Let's take a look at what the lazy approach might look like:

![Diagram of lazy approach]
Using the lazy approach, we're able to remove the expensive transform computation that happens up-front. Instead, we transform only one item at a time. We're also able to remove the up-front allocation of the transformed data structure. Instead, only the transformed item is passed into the component. Then, the component can ask for another item or stop. The conserve principle uses concurrency as a means to only compute what's needed and only allocate memory that's needed.

The following checklist will help us think about the conserve principle when writing concurrent code:

- Are we computing values that are never used?
- Do we only allocate data structures as a means to pass them from one component to the next?
- Do we chain-together data transformation actions?

**Summary**

In this chapter, we introduced some motivations for concurrency in JavaScript. While synchronous JavaScript is easy to maintain and understand, asynchronous JavaScript code is inevitable on the web. So it's important to make concurrency our default assumption when writing JavaScript applications.

There's two main types of concurrency we're interested in—asynchronous actions and parallel actions. Asynchronicity is about the time ordering of actions, which gives the impression that things are happening at the same time. Without this type of concurrency, the user experience would suffer greatly, because it would constantly be waiting on other actions to complete. Parallelism is another type of concurrency that solves a different type of problem, where we want to increase performance by computing results faster.

Finally, we looked at the three principles of concurrency in JavaScript programming. The parallelize principle is about leveraging the multi-core CPUs found in modern systems. The synchronize principle is about creating abstractions that enable us to write concurrent code, hiding the concurrency mechanisms from our feature code. The conserve principle uses lazy evaluation to only compute what is needed and to avoid unnecessary memory allocations.

In the next chapter, we'll turn our attention to the JavaScript execution environment. To be effective with JavaScript concurrency, we need a sound understanding of what's actually happening when our code is run.
Where to buy this book

You can buy JavaScript Concurrency from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.