

UNSTUCK

JavaScript

> 3 simple steps to
get moving again,
build momentum,
and finish your project

Scott Murray

Unstuck: JavaScript

Version 1.0 — 2021 June 6

Version 1.0.1 — 2021 September 12

By the time you read this, this book may already have been updated! Check unstuckjs.com for the current version, and sign up to be notified of new releases.

Copyright © 2021 Scott Murray

Published by CuriousT

Identifiers

978-1-7371737-0-0 — PDF ebook

978-1-7371737-1-7 — EPUB ebook

978-1-7371737-2-4 — Printed book

Context

A System of Strategies

Scope and Audience for this Book

The Strategies

Code-level

Log it out.

Trace it back.

Check the type.

Check the return type.

Check the scope.

Check for name conflicts.

Check the sequence of events.

Excerpt to isolate.

Sidestep any black boxes.

Confirm browser support.

Project-level

Clarify the problem.

Write it down.

Sketch it out.

Explain it to your dog, cat, plant, or pet rock.

Explain it to a disinterested loved one.

Reach out for help.

Let your subconscious do the work.

Process-level

Optimize for “good enough.”

Know when you’re stuck.

Embrace your humanity.

Disagree (with yourself) and commit.

Process in batches.

Flip your to-do list.

Be strategic and selective when choosing what to do next.

Do one thing at a time.

Manage your mental environment.

Closing

What I learned about moving forward while stuck in a pandemic

Circling Back

Thanks

Context

⚠️ If you are stuck right now, please skip ahead to **A System of Strategies**. Seriously, you can come back here later. I don't want you reading this as a means of procrastination when there is work to be done!

I get stuck a lot.

Stuck on a project or a task. Stuck on debugging code. On solving a new kind of problem, or an old problem I've solved ten times before. Stuck doing something I know how to do but would rather not face, or stuck facing something that feels so big and daunting I don't know where to begin.

The problem is, I don't *like* being stuck.

When I'm stuck, I feel trapped or stupid or lost or scared. I feel like I'm wasting time, and the stickiness compounds. Like the longer I *stay* stuck, the harder it will be to get moving again, and that terrifies me. What if I am stuck right here, trapped beneath this boulder, forever?

In this metaphor, the “boulder” is JavaScript.

Ouch!

We love JavaScript. We hate JavaScript.

We love JavaScript because, when our code works the way we want it to, we can create and deliver powerful and beautiful experiences. We can build websites and move data and objects around. Frankly, when it's working, it feels like magic.

We hate JavaScript because writing code often involves a lot more time staring at the screen than actually "writing" anything.

Most of my time "writing" code is actually spent scrolling through broken code that I've already written. Or pushing commits to a repo on a remote server, then hitting refresh and watching the browser just... sort of sit there. Or, worst of all, staring at an empty `.js` document with no clear starting point in mind.

Those are the moments when I hate JavaScript. But JavaScript isn't really the problem.

The problem is that, in those moments, I don't know what I should do *next*.

We all get stuck. Sometimes we stay stuck for a long time: minutes, hours, days, weeks... even longer (speaking for myself here). Consider all the time that everyone reading this book has spent in a stuck-like state this year. I don't have exact number in front of me, but it's a big number! That's many, many person-hours, -days, and -weeks of "wasted" time that could have otherwise been spent making progress.

My goal with this book is to get you *unstuck much more quickly*. Seconds instead of minutes. Minutes instead of hours. Finishing today what you would have grappled with or avoided until tomorrow.

The not-so-mysterious secret to getting unstuck is:

Knowing your immediate next step.

That may sound simple, but identifying one's next step *in the moment* — a moment of stuck-ness, no less, when your brain is tired and has already been working on this project all day and wow I just noticed I'm hungry and wait who's calling me this time —

Well, we have a lot going on, mentally and emotionally, and maintaining a constant awareness of your next step would require a level of presence and self-awareness that few possess. I aspire to be so self-aware, moment to moment, but I'm not, and I don't expect you to be, either.

That's why the rest of this book elucidates *strategies for getting unstuck*.

If you have a *strategy*, you don't *need* self-awareness.

Have you ever read Steve Krug's *Don't Make Me Think*? It's a wonderful book. The core of its argument is: a great interface gets "out of the way," cognitively speaking, by meeting users' expectations and needs. A great interface adheres so closely to its users' preexisting mental models that it intuitively "just works" and helps them accomplish whatever they are trying to do. Underlying this is an acknowledgement of the *humanity* of users; users are real people with human brains,

operating in the real world, with real competing demands for their time, energy, and — crucially — cognition. “Don’t make me think” is a mantra of reducing cognitive load, with the end result of increasing user success.

Let me ask you about one other book.

Have you read Atul Gawande’s *The Checklist Manifesto*? It’s also fantastic, and another acknowledgement of our humanity — that is, our limitations, tendencies, habits, and irritations. The core of its argument is: even highly trained experts are subject to mental lapses in attention or judgement, due in part to physical limitations in our cognitive capacity. This isn’t particularly high-stakes when I forget to take out the recycling on Monday, but when a surgeon, pilot, or other professional navigates a time-sensitive situation with life-or-death consequences, the tiniest missed step in protocol can be fatal.

The solution is to make *and use* a checklist. A checklist gets the burden of remembering out of your head and onto a piece of paper. You can forget about the next task — literally — and save those brain cells and attention for the matter at hand, right now. You can give your full attention to the current operation, confident that your next one is already written down and will be there when you’re ready for it.

My approach to getting unstuck is inspired by both Krug and Gawande. I have ups and downs throughout the day. My energy level ebbs and flows. My mood drops, then is boosted. And my tendency to get stuck fluctuates, too.

This frustrated me until I finally — *finally!* — acknowledged my own humanity. As in: Duh, Scott, you're just a person, so *of course* you can't be at your best all the time, every moment, all day long.

So:

This book is your checklist for getting unstuck.

When you are stuck, this book will be guide you to your *next step*.

To keep *moving forward* — to keep making progress — you have to know what your *next step* is. Your *next step* is your next concrete action, the very small and very manageable thing you can do right now, immediately: Make a phone call. Write a short email. Move the laundry from the washer to the dryer.

Your next step is *something you already know how to do* and — crucially — it is indivisible: it cannot be broken down into smaller steps. “Change careers” is not a next step, but “find 5 articles about the most valuable skills in [new career]” is.

When I am stuck, invariably one of these apply:

1. I don't know what the next step is.
2. I know what the next step is, but it feels too big / scary / vague / poorly defined.
3. There are many *possible* next steps, but I can't decide which one to do first.
4. The next step may be clear, but I am spending far too much time on the current step without even realizing it. I am stuck, and I don't even know that I am stuck!

If this resonates for you, let's talk about strategies.

A System of Strategies

This book is not *just* a list of strategies: It is a *system* for getting to your next step.

To code with JavaScript is to... get stuck. Here's how to get moving again.

1 When your project...	is already underway	is already underway	has not yet begun
2 and you...	know what the problem is, but not how to solve it	are unclear on the problem itself	have time to redesign your process and workflow
3 then try each of these strategies, in order.	Code-level strategies Log it out. Trace it back. Check the type. Check the return type. Check the scope. Check for name conflicts. Check the sequence of events. Excerpt to isolate. Sidestep any black boxes. Confirm browser support.	Project-level strategies Clarify the problem. Write it down. Sketch it out. Explain it to your dog, cat, plant, or pet rock. Explain it to a disinterested loved one. Reach out for help. Let your subconscious do the work.	Process-level strategies Optimize for "good enough." Know when you're stuck. Embrace your humanity. Disagree (with yourself) and commit. Process in batches. Flip your to-do list. Be strategic and selective when choosing what to do next. Do one thing at a time. Manage your mental environment.

UNSTUCK
JavaScript
Cheat Sheet

Learn how to use these strategies in the book *Unstuck: JavaScript*. Free chapters and tips at unstuckjs.com

Updated 2021 June 8
© 2020-2021 Scott Murray

Print out this cheat sheet and keep it nearby. (Or download a fresh copy from unstuckjs.com.)

When you get stuck, start in the top left corner and answer the questions — 1, 2, 3. Your answers will guide you to one of three columns:

- **Code-level strategies** for when you know what the problem is, but not how to solve it. These are the technical, nitty gritty bits.
- **Project-level strategies** for when you are unclear on the problem itself. These strategies help you clarify, conceptualize, and define the problem, so you can get back to coding a solution.
- **Process-level strategies** for when you have time to redesign your process and workflow. These are big-picture strategies that *improve*

the way you work by reducing friction and helping you build and maintain forward momentum.

Starting at the top of the column, try the first strategy on the list. Did that get you going again? Wonderful! You're off to the races.

Still stuck? Try the next strategy, and the next, and so on, until you get moving again.

Premise

Why does this work?

Well, our brains can't be trusted. It's not personal; it's just how we're wired. Our brains are *very* good at avoiding truths we don't want to face.

Using a checklist *systematizes* the process of getting unstuck; it depersonalizes the situation, thereby letting your ego save face and quietly sneak off to the side, as you plow full-steam ahead.

Get going

You've got two options:

1. Let the cheat sheet guide you to the best strategies for you *right now*.
2. Continue reading this book in normal, linear order.

If you're stuck on something right now, I recommend the former.

If you're not currently stuck: Congratulations! Get yourself in a comfortable, stress-free reading position, and we'll get started.

Scope and Audience for this Book

This is not a particularly technical book, despite the title.

This book is not *about* code, but about *working with* code.

Many of the strategies here are applicable to any kind of programming, or any kind of work. But I emphasize developing JavaScript code to run in the browser (front-end, client-side), and thus typically alongside HTML and CSS — as opposed to back-end or server-side JavaScript, such as with Node.js.

I've written this for people who are relatively new to JavaScript, but this book won't *teach* you JavaScript.

Maybe you've been tinkering with JavaScript on and off for years. Maybe you are a new web developer, right out of school. Maybe you *do* code for a living, but never had any formal training, like a computer science degree, and you kind of have a chip on your shoulder about it. (Hey, that's me, too!)

Maybe you don't even consider yourself a programmer, a developer, or a coder. You have been coding for a while, but are starting to work on more complex projects or team projects and find yourself hitting a wall, needing to take your working skills to the next level.

This book is for you new, new-ish, and intermediate JavaScript coders who need some guidance on *how to work with* JavaScript.

Given that, I assume you're developing using a current code editor, testing using a current web browser (Chrome, Firefox, Safari, Edge), and that you're already familiar with how to access the JavaScript console in your browser. (Hint: It's probably under a menu option like View > Developer > JavaScript Console.)

I also assume you have coded some web pages before, so you're familiar with the routine of:

- edit a file in your code editor, then save changes
- switch to your browser, hit refresh to observe changes

Ah, web development. Some things never change!

Whoever you are: Welcome. I am glad you're here.

The Strategies

Code-level

Strategies for when you know what the problem is, but not how to solve it. These are the technical, nitty gritty bits.

Log it out.

When in doubt, log it out.

Computers are inscrutable machines. They are entirely concrete, operating with literal precision. Yet they are completely abstract, their microscopic inner workings completely opaque and unknowable to humans.

Logging is one of our only tools for peeking inside the machine. Logging is a digital microscope, a way to directly observe what the computer is thinking and doing.

But don't just log blindly. Be strategic and focused. Here's how and what to log.

Start wherever your expectations are not being met by reality.

Let's back up. Until now, you may have been thinking of whatever coding issue you're facing as a "bug."

But bugs are just *unfulfilled expectations*. They are the spots in a program that aren't behaving the way you want them to. Framing these as "bugs" blames the computer or the code — it externalizes the problem and points the finger.

Yet, almost always, the computer is operating "as intended." Unlike [Grace Murray Hopper's 1947 bug](#), there is no literal insect stuck in the wires. The machine, and your code, are working. They are just not doing what you want them to do.

I'm sorry to tell you this, but the problem is *you*.

Well, not *you*, but *your expectations*. There are no bugs; there are only mismatches between expectations and observable reality.

Your job when “debugging” is to bring expectation and reality back into alignment.

Fortunately, you can do that by deepening your understandings of both the problem and the code behind it. This is where the microscope comes in.

Framing the question

When you first notice a problem, it helps to reframe the problem as a testable question to be answered. Consider this your hypothesis to be proven or disproven.

For example, say that something screwy is happening in your web interface — you see 8 `` bullet points where you expected only 6.

The problem is that “the list has too many items.”

You could reframe that as a hypothesis like “the list-generating function is returning an incorrect value.”

Notice that the problem statement is merely an observation, while the hypothesis is something you can test and verify. Is the function returning the expected value, or not? Is your hypothesis true or false?

By working your way toward the answer, you'll discover why your expectations aren't being met.

Where to log

Start where you first *observe* the problem.

If you're lucky, the *actual* problem will be in the same spot or nearby.

Let's pretend that you remember having a function called `listGenerator()`. This function creates the bulleted list on your page, so that's where you begin your search. In the source code, you find:

```
listGenerator(numItems);
```

Before digging into the inner workings of that function, let's validate that the value being *passed in* to the function is as expected.

How to log

That's where `console.log()` comes in. It's your easiest-to-use computer microscope. Focus your microscope on the thing that you suspect is returning the unexpected value by wrapping it in `console.log()`'s gentle parentheses. Here, we'll log out the value of `numItems` *immediately before* the `listGenerator()` is called — so we can observe whatever that value is going into the function, before any other code has a chance to change its value.

```
console.log(numItems); //Returns 8
listGenerator(numItems);
```

Now when you refresh the browser, 8 appears in the console.

This may not feel like progress, but it actually *is* progress. Now we know two things:

- the value being passed in to the `listGenerator()` is not expected
- the `listGenerator()` is working as expected — you pass in 8, and it generates 8 list items

Well, at least something around here is working!

So the value of `numItems` is unexpected. Where did it come from? Continue on to the next section, “Trace it back.”

Trace it back.

When a value is missing or unexpected, your job is to figure out how that value got that way.

So, trace it back. As in, trace that value back to its source, to figure out where it originated. Start at the end, then work backwards, one careful step at a time.

Manual, step-by-step tracing

Returning to our example, we learned that `numItems` was set to 8, but we expected it to be 6.

```
console.log(numItems); //Returns 8  
listGenerator(numItems);
```

So, where did `numItems` come from? What happened to it *just before* we observed it here?

Sometimes, you'll just know the answer off the top of your head — because you just wrote that code a moment ago. Otherwise, try searching with command-F for “numItems” will get you what you need. Usually, you'll want to search “up” or backward in the document.

Say we search back and find where, earlier in our code, `numItems` was declared and assigned a value.

```
var numItems = multiplier(2);  
  
//...  
  
console.log(numItems); //Returns 8  
listGenerator(numItems);
```

The next thing I'd do is validate what's being returned by whatever this `multiplier()` function is. These two new log statements will return the same thing, so choose whichever floats your boat.

```
var numItems = multiplier(2);  
console.log(multiplier(2)); //Returns 8  
console.log(numItems);      //Returns 8  
  
//...  
  
console.log(numItems); //Returns 8  
listGenerator(numItems);
```

Yet again, *we are actually making progress*, though it may not feel like it. We've validated that the unexpected value is first assigned to `numItems` right when it's declared. So, the problem is — cue the horns — *inside* the function!

So now we look back even earlier in our code, to find when `multiplier()` is defined.

```

function multiplier(input) {
    var output = input * 4;
    return output;
}

//...

var numItems = multiplier(2);
console.log(multiplier(2)); //Returns 8
console.log(numItems);      //Returns 8

//...

console.log(numItems); //Returns 8
listGenerator(numItems);

```

Do you see it? Can you find why we're getting back 8, and not 6?

```

var output = input * 4; // ← HINT

```

In this contrived example, I miscalibrated the multiplier! I should have written `input * 3` instead of `input * 4`. Oops.

No big deal. Literally everyone makes mistakes — even and *especially* the pros! And you, well, you just *fixed* a mistake, which earns you 10 bonus points!


```

function multiplier(input) {
    var output = input * 3; // ← FIXED
    return output;
}

//...

var numItems = multiplier(2);
console.log(multiplier(2)); //Returns 6!
console.log(numItems);      //Returns 6!

//...

console.log(numItems); //Returns 6!
listGenerator(numItems);

```

Upon re-running the code, all your log statements output the expected value. Hooray! The “bug” has been resolved; expectations and reality are back in alignment.

All that’s left is to *carefully* go back and remove or comment out those now-unneeded log statements.

Stack tracing

If your mind is reeling from trying to trace things back, step-by-step, you might be wondering “Can’t the computer do this for me?”

Great question! And yes, it can... sort of.

`console.trace()` prints a stack trace — the nested sequence of functions called to get to that point in the code where `trace()` is itself called — to the console.

For example, say you defined a `parent()` function that included its own `child()` function, like so:

```
function parent() {           // Define parent function
  function child() {         // Define child function
    console.trace();        // Trace!
  }
  child();                   // Call child() from within parent()
}
```

When you call `parent()`, it will call the `child()` function in turn, and `child()` will initiate your stack trace, with output like this:

```
> parent()
  ▼ console.trace
    child @ VM1399:3
    parent @ VM1399:5
    (anonymous) @ VM1405:1
```

See, the “stack” is just the stack of functions that get us to the moment when the trace is initiated. The stack is presented with the most recently called function first. In this case, it’s:

- child
- parent
- (anonymous)

“Anonymous” means we are no longer “inside” any specific function and, in this case, we are back to the global scope. (You could also see “anonymous” reported in a stack trace if you’re using unnamed, anonymous functions — though I don’t recommend those.)

You can also pass a value to `trace()` to log, just like `console.log()`, and it will log out the value along with the stack trace.

```
console.trace(valueInQuestion);
```

Conditional tracing with assertions

Sometimes you need that stack trace only when something goes wrong. `console.assert()` is just like `console.trace()`, but it logs the trace only when you need it to.

Here's how it works. `console.assert()` takes an assertion — a boolean expression. If the assertion is *true*, then nothing is returned. But if the assertion is *false*, then a stack trace is returned, along with any messaging you specify. For example:


```
function parent() {
  function child() {
    var bananas = 10;
    console.assert(bananas >= 10, "Not enough bananas!");
  }
  child();
}
```

Calling this `parent()` produces no output. `bananas` is set to 10, and the assertion checks that `bananas` is greater than or equal to 10. So the assertion returns `true` and that's the end of that.

But what if we didn't have enough bananas?

```
function parent() {
  function child() {
    var bananas = 9; // ← OH, NO!
    console.assert(bananas ≥ 10, "Not enough bananas!");
  }
  child();
}
```

In that case, calling `parent()` will eventually trigger the assertion, which now returns `false`, so `assert()` outputs this to the console:

```
> parent()
 ▼ Assertion failed: Not enough bananas!
  child @ VM1726:4
  parent @ VM1726:6
  (anonymous) @ VM1732:1
```

I really like how `console.assert()` combines three very useful ideas — conditional logic, logging, and stack tracing — into a single method. It's nice to have something that will warn you if a condition isn't met, but otherwise remains silent.

Fancier logging options

`console.log()` is perfectly simple. You can use it to log different types of values — numbers, strings, dates, even simple objects:

```
console.log(10);  
console.log("Text!");  
console.log(new Date);  
console.log({ name: "Scott" });
```

Note, too, that you can include emoji in your log statements, as visual cues to yourself:

```
console.log("🦄🌈😄");
```

For checking a single value, `console.log()` is perfect. But often, you need to check *multiple* values at a time, or you realize you've got several log statements scattered throughout your code. It can get a bit messy.

Let me introduce you to a few handy console logging tools that, employed sparingly, can help you in your efforts to "trace it back."

table()

Sure, you can pass more than one value to `console.log()`, separated by commas, but the output isn't pretty.

```
> console.log("one", "two", "three");  
one two three
```

Whenever you need to log out more than a couple values at once, try `console.table()`. The syntax is similar, but uses hard brackets `[]` to indicate an array.

```
> console.table([ "one", "two", "three" ]);
```

(index)	Value
0	"one"
1	"two"
2	"three"

But this method really shines when outputting an array of objects with named properties. Then the property names are used as column names and — well, wow, it's like your objects just got mapped to a spreadsheet!

```
var maple = { name: "Maple", type: "Deciduous" };  
var spruce = { name: "Spruce", type: "Evergreen" };  
var sheep = { name: "Sheep", type: "Ovine" };
```

```
console.table([ maple, spruce, sheep ]);
```

(index)	name	type
0	"Maple"	"Deciduous"
1	"Spruce"	"Evergreen"
2	"Sheep"	"Ovine"

info(), warn(), error()

These three are functionally equivalent to `log()`, only your browser may give them special visual treatments, each one getting progressively more scary and attention-grabbing.

```
> console.info("Just letting you know")
  Just letting you know

> console.warn("This doesn't look right")
⚠ This doesn't look right

> console.error("Something is really wrong")
✖ Something is really wrong
```

Also, your browser dev tools will let you filter logged output by type — say, only “info,” warnings, or errors.

When logging, it can be helpful to differentiate between something that’s just informational versus something that is a real problem.

Value substitution

String substitutions let you craft more meaningful log messages, and sprinkle in the dynamically logged values as needed. For example, in the log statement below, the %s is replaced by the string value that’s passed as a subsequent parameter.

```
var horse = { name: "Horse", type: "Equine" };
console.log("Looks like a %s", horse.name);
// Logs: Looks like a Horse
```

The most useful substitution strings are:

- %i for an integer
- %f for a floating-point value (decimal number)
- %s for a string

- %o for a JavaScript object

Take care to use the right substitution strings with the right value types, or you'll get unexpected results. For example, if I accidentally used %f to reference a string value:

```
var horse = { name: "Horse", type: "Equine" };
console.log("Looks like a %f", horse.name); // ← 00PS
// Logs: Looks like a NaN
```

These are most useful when used in conjunction with each other, to generate more meaningful log statements, such as:

```
var horse = { name: "Horse", type: "Equine", count: 5 };
console.log("I see %i %ss", horse.count, horse.name);
// Logs: I see 5 Horses
```

Okay, "I see 5 Horses" isn't super meaningful, but you get the idea.

Alternatively, you may prefer **template literals** for a simpler syntax that doesn't require you to know the value type in advance.

```
var horse = { name: "Horse", type: "Equine" };
console.log(`Looks like a ${horse.name}`);
// Logs: Looks like a Horse
```

A template literal is like a string, but enclosed in backticks (` `) instead of single quotes (' ') or double quotes (" ").

Within the template literal, `${}` indicates an expression, so whatever is between those curly braces will be executed, and then, in this

example, returned as part of the string value logged to the console. Here is a simple example:

```
console.log(`${2 + 5}`);  
// Logs: 7
```

You can use both substitution strings and template literals with `log()`, `info()`, `warn()`, `error()`, and `assert()`.

Pretty colors

If `warn()` and `error()`'s  and  aren't enough bling for you, go nuts by injecting custom CSS to style your log output.

Use `%c` to indicate that `log()` should reference the next parameter and treat it as a CSS style rule.

```
console.log("%cPurple is cool", "color: purple");  
//Logs "Purple is cool", but in purple text!
```

```
> console.log("%cPurple is cool", "color: purple");  
Purple is cool VM2729:1
```

While purple *is* cool, this feature is more commonly used to color-code log messages for easy scanning. For example, set all your thingamajig-related messages in navy blue, while doodad messages are in brown. Use multiple `%c` markers, and pass in additional CSS rules as needed.

```
console.log("%cTHINGAMAJIG:%c Loaded successfully", "color: navy",
"color: gray");
console.log("%cDOODAD:%c Configuring... ", "color: brown", "color:
gray");
```

```
> console.log("%cTHINGAMAJIG:%c Loaded successfully", "color:
navy", "color: gray");
console.log("%cDOODAD:%c Configuring...", "color: brown", "color:
gray");
THINGAMAJIG: Loaded successfully VM2737:1
DOODAD: Configuring... VM2737:2
```

You're not limited to the `color` property. Also try `background`, `border`, `font-family`, `font-weight`, `margin`, and `padding`.

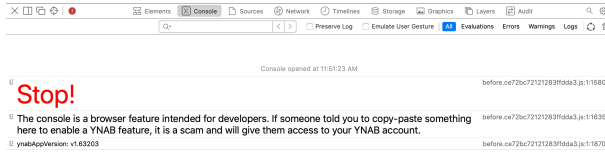
I advise against doing this, but you could really style up quite the atrocity in your own console:

```
console.log("%cPurple is cool", "color: purple; background-color:
pink; border: solid 5px navy; border-radius: 10px; padding:
10px; font-family: Helvetica; font-weight: bold; font-size:
32px;");
```

```
> console.log("%cPurple is cool", "color: purple; background-color:
pink; border: solid 5px navy; border-radius: 10px; padding: 10px;
font-family: Helvetica; font-weight: bold; font-size: 32px;");
VM2707:1
```



You may not use CSS-in-the-console much for debugging, but it can be useful to grab someone's attention, should you need to. For example, one of my favorite personal finance tools, [You Need a Budget](#), uses CSS to warn their customers against potential fraud, should some scammer instruct them to poke around in the console.



Groups

You can group related log messages together by preceding them with `console.group()` and closing the group with `console.groupEnd()`.

```
console.group("FIRST GROUP OF MESSAGES");
console.log("First message");
console.log("Second message");
console.log("Third message");
console.groupEnd();
console.group("SECOND GROUP OF MESSAGES");
console.log("Fourth message");
console.log("Fifth message");
console.log("Sixth message");
console.groupEnd();
```

▼ FIRST GROUP OF MESSAGES	VM2972:1
First message	VM2972:2
Second message	VM2972:3
Third message	VM2972:4
▼ SECOND GROUP OF MESSAGES	VM2972:6
Fourth message	VM2972:7
Fifth message	VM2972:8
Sixth message	VM2972:9

Grouping may be useful when, say, you have an overwhelming amount of tens or hundreds of log entries, so the ability to collapse them and organize them (in nested sets, even) would make the output more easily navigable.

Timers

If something is taking a reeeeeeeeeeally long time to execute, you may want to use a timer to quantify how tediously long it's taking — um, and to help identify the source of the delay.

Use `console.time()` with a string, to name your new timer and start it. Use `console.timeEnd()`, referencing the timer's name, to stop the timer and get the total elapsed time.

```
> console.time("myTimer")
undefined
> console.timeEnd("myTimer")
myTimer: 13524.662353515625 ms
```

In this case, `myTimer` ran for 13,524 milliseconds, or about 13.5 seconds.

When would you use this? Let's say you just implemented a new function, and suddenly everything feels a lot slower. You suspect something in the new function is to blame. So, inside the function definition, you start a timer as the very first step within the function, and then end with `timeEnd()`. For example:

```
function wayTooSlow() {
  console.time("stopwatch");
  //Do stuff that takes too long...
  console.timeEnd("stopwatch");
}
```

While the timer is running, you can also call `console.timeLog()` to see how much time has elapsed up to that point (but without ending the timer).

```
> console.time("myTimer")
undefined
> console.timeLog("myTimer")
myTimer: 10703.940185546875 ms
> console.timeEnd("myTimer")
myTimer: 20455.124892578125 ms
```

Note that you can have multiple timers running all at the same time, because each has a unique name.

```
> console.time("myFirstTimer")
undefined
> console.time("mySecondTimer")
undefined
> console.time("myThirdTimer")
undefined
```

Finally, you won't be able to create a new timer with the same name if an existing one is already running.

```
> console.time("myTimer")
undefined
> console.time("myTimer")
⚠ Timer 'myTimer' already exists
```

This concludes our sample...

Hey! Scott here. I hope you are enjoying the book so far.

If you're not, I hope you'll write to me and let me know why not. Email me at scott@scottmurray.org.

If you're loving it, the rest of the book, with all 26 strategies, can be found at: unstuckjs.com