



THE MODERN JAVASCRIPT COLLECTION



A FOUR VOLUME SET

The Modern JavaScript Collection

Copyright © 2018 SitePoint Pty. Ltd.

Cover Design: Alex Walker

Ebook ISBN: 978-0-6483315-5-1

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.



Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.



Preface

There's no doubt that the JavaScript ecosystem changes fast. Not only are new tools and frameworks introduced and developed at a rapid rate, the language itself has undergone big changes with the introduction of ES2015 (aka ES6) and further revisions. Understandably, many articles have been written complaining about how difficult it is to learn modern JavaScript development these days. We're aiming to minimize that confusion with this set of books on modern JavaScript.

This collection contains:

- *Practical ES6* is a collection of articles introducing many of the powerful new JavaScript language features that were introduced in ECMAScript 2015, as well as features introduced in ECMAScript 2016 and 2017. It also takes a look at the features planned for ECMAScript 2018 in this rapidly evolving language.
- *JavaScript: Best Practice* presents articles discussing modern JavaScript best practice, enabling you to write more powerful code that is clean, performant, maintainable, and reusable.
- *6 JavaScript Projects* presents six complete JavaScript projects; each taking advantage of modern JavaScript and its ecosystem. You'll learn to build several different apps, and along the way you'll pick up a ton of useful advice, tips, and techniques.
- *Modern JavaScript Tools & Skills* contains a collection of articles outlining essential tools and skills that every modern JavaScript developer should know.

Who Should Read This Collection?

This book is for all front-end developers who wish to improve their JavaScript skills. You'll need to be familiar with HTML and CSS and have a reasonable level of understanding of JavaScript in order to follow the discussion.

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```

Where existing code is required for context, rather than repeat all of it, `:` will be displayed:

```
function animate() {
  :
new_variable = "Hello";
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An \Rightarrow indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-
design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand.

Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

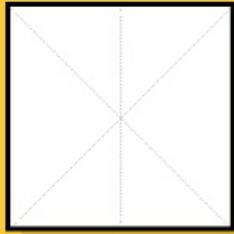
Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

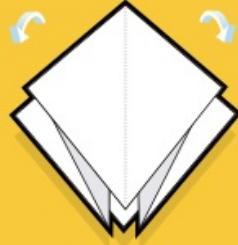
Book 1: Practical ES6



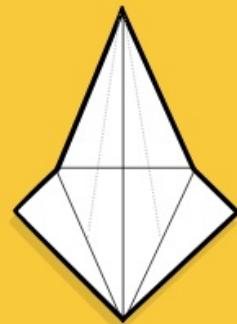
PRACTICAL ES6



1.



2.



3.

LEVEL UP YOUR JAVASCRIPT

Chapter 1: New Keywords: let and const

by Aurelio de Rosa

In this tutorial, I'll introduce `let` and `const`, two new keywords added to JavaScript with the arrival of ES6. They enhance JavaScript by providing a way to define block-scope variables and constants.

let

Up to ES5, JavaScript had only two types of scope, function scope and global scope. This caused a lot of frustration and unexpected behaviors for developers coming from other languages such as C, C++ or Java. JavaScript lacked block scope, meaning that a variable is only accessible within the block in which it's defined. A block is everything inside an opening and closing curly bracket. Let's take a look at the following example:

```
function foo() {  
  var par = 1;  
  if (par >= 0) {  
    var bar = 2;  
    console.log(par); // prints 1  
    console.log(bar); // prints 2  
  }  
  console.log(par); // prints 1  
  console.log(bar); // prints 2  
}  
foo();
```

After running this code, you'll see the following output in the console:

```
1  
2  
1  
2
```

What most developers coming from the languages mentioned above would expect, is that outside the `if` block you can't access the `bar` variable. For

example, running the equivalent code in C results in the error 'bar' undeclared at line ... which refers to the use of bar outside the if.

This situation changed in ES6 with the availability of block scope. The ECMA organization members knew that they could not change the behavior of the keyword var, as that would break backward compatibility. So they decided to introduce a new keyword called let. The latter can be used to define variables limiting their scope to the block in which they are declared. In addition, unlike var, variables declared using let aren't [hoisted](#). If you reference a variable in a block before the let declaration for that variable is encountered, this results in a ReferenceError. But what does this mean in practice? Is it only good for newbies? Not at all!

To explain you why you'll love let, consider the following code taken from my article [5 More JavaScript Interview Exercises](#):

```
var nodes = document.getElementsByTagName('button');
for (var i = 0; i < nodes.length; i++) {
  nodes[i].addEventListener('click', function() {
    console.log('You clicked element #' + i);
  });
}
```

Here you can recognize a well-known issue that comes from variable declaration, their scope, and event handlers. If you don't know what I'm talking about, go check the article I mentioned and than come back.

Thanks to ES6, we can easily solve this issue by declaring the i variable in the for loop using let:

```
var nodes = document.getElementsByTagName('button');
for (let i = 0; i < nodes.length; i++) {
  nodes[i].addEventListener('click', function() {
    console.log('You clicked element #' + i);
  });
}
```

The let statement is supported in Node and all modern browsers. There are, however, a couple of gotchas in Internet Explorer 11 which you can read about in the [ES6 compatibility table](#).

A live demo that shows the difference between var and let is [available at JSBin](#).

const

`const` addresses the common need of developers to associate a mnemonic name with a given value such that the value can't be changed (or in simpler terms, define a constant). For example, if you're working with math formulas, you may need to create a `Math` object. Inside this object you want to associate the values of `π` and `e` with a mnemonic name. `const` allows you to achieve this goal. Using it you can create a constant that can be global or local to the function in which it is declared.

Constants defined with `const` follow the same scope rules as variables, but they can't be redeclared. Constants also share a feature with variables declared using `let` in that they are block-scoped instead of function-scoped (and thus they're not hoisted). In case you try to access a constant before it's declared, you'll receive a `ReferenceError`. If you try to assign a different value to a variable declared with `const`, you'll receive a `TypeError`.

Please note, however, that `const` is *not* about immutability. As Mathias Bynens states in his blog post [ES2015 const is not about immutability](#), `const` creates an immutable binding, but does not indicate that a value is immutable, as the following code demonstrates:

```
const foo = {};
foo.bar = 42;
console.log(foo.bar);
// → 42
```

If you want to make an object's values truly immutable, use [`Object.freeze\(\)`](#).

Browser support for `const` is equally good as for `let`. The statement `const` is supported in Node and all modern browsers. But here, too, there are some gotchas in Internet Explorer 11, which you can read about in the [ES6 compatibility table](#).

An example usage of `const` is shown below:

```
'use strict';

function foo() {
  const con1 = 3.141;
  if (con1 > 3) {
```

```
    const con2 = 1.414;
    console.log(con1); // prints 3.141
    console.log(con2); // prints 1.414
}
console.log(con1); // prints 3.141
try {
  console.log(con2);
} catch(ex) {
  console.log('Cannot access con2 outside its block');
}
}
foo();
```

A live demo of the previous code is [available at JSBin](#).

Conclusion

In this tutorial, I've introduced you to `let` and `const`, two new methods for declaring variables that were introduced to the language with ES6. While `var` isn't going away any time soon, I'd encourage you to use `const` and `let` whenever possible to reduce your code's susceptibility to errors. By way of further reading, you might also like our quick tip [How to Declare Variables in JavaScript](#), which delves further into the mechanics of variable declaration.

Chapter 2: Using Map, Set, WeakMap, WeakSet

by Kyle Pennell

This chapter examines four new ES6 collections and the benefits they provide.

Most major programming languages have several types of data collections. Python has lists, tuples, and dictionaries. Java has lists, sets, maps, queues. Ruby has hashes and arrays. JavaScript, up until now, had only arrays. Objects and arrays were the workhorses of JavaScript. ES6 introduces four new data structures that will add power and expressiveness to the language: Map, Set, WeakSet, and WeakMap.

Searching for the JavaScript HashMap

HashMaps, dictionaries, and hashes are several ways that various programming languages store key/value pairs, and these data structures are optimized for fast retrieval.

In ES5, JavaScript objects — which are just arbitrary collections of properties with keys and values — can simulate hashes, but there are [several downsides to using objects as hashes](#).

Downside #1: Keys must be strings in ES5

JavaScript object property keys must be strings, which limits their ability to serve as a collection of key/value pairs of varying data types. You can, of course, coerce/stringify other data types into strings, but this adds extra work.

Downside #2: Objects are not inherently iterable

Objects weren't designed to be used as collections, and as a result there's no

efficient way to determine how many properties an object has. (See, for example, [Object.keys is slow](#)). When you loop over an object's properties, you also get its prototype properties. You could add the `iterable` property to all objects, but not all objects are meant to be used as collections. You could use the `for ... in` loop and the `hasOwnProperty()` method, but this is just a workaround. When you loop over an object's properties, the properties won't necessarily be retrieved in the same order they were inserted.

Downside #3: Challenges with built-in method collisions

Objects have built-in methods like `constructor`, `toString`, and `valueOf`. If one of these was added as a property, it could cause collisions. You could use `Object.create(null)` to create a bare object (which doesn't inherit from `object.prototype`), but, again, this is just a workaround.

ES6 includes new collection data types, so there's no longer a need to use objects and live with their drawbacks.

Using ES6 Map Collections

Map is the first data structure/collection we'll examine. Maps are collections of keys and values of any type. It's easy to create new Maps, add/remove values, loop over keys/values and efficiently determine their size. Here are the crucial methods:

Creating a map and using common methods

```
const map = new Map(); // Create a new Map
map.set('hobby', 'cycling'); // Sets a key value pair

const foods = { dinner: 'Curry', lunch: 'Sandwich', breakfast: 'Eggs'}
const normalfoods = {} // New Object

map.set(normalfoods, foods); // Sets two objects as key value pair

for (const [key, value] of map) {
  console.log(` ${key} = ${value}`); // hobby = cycling [object Object]
}

map.forEach((value, key) => {
  console.log(` ${key} = ${value}`);
```

```
}, map); // hobby = cycling [object Object] = [object Object]
map.clear(); // Clears key value pairs
console.log(map.size === 0); // True
```

[Run this example on JSBin](#)

Using the Set Collection

Sets are ordered lists of values that contain no duplicates. Instead of being indexed like arrays are, sets are accessed using keys. Sets already exist in [Java](#), [Ruby](#), [Python](#), and many other languages. One difference between ES6 Sets and those in other languages is that the order matters in ES6 (not so in many other languages). Here are the crucial Set methods:

```
const planetsOrderFromSun = new Set();
planetsOrderFromSun.add('Mercury');
planetsOrderFromSun.add('Venus').add('Earth').add('Mars'); // Chaining
console.log(planetsOrderFromSun.has('Earth')); // True

planetsOrderFromSun.delete('Mars');
console.log(planetsOrderFromSun.has('Mars')); // False

for (const x of planetsOrderFromSun) {
  console.log(x); // Same order in as out - Mercury Venus Earth
}
console.log(planetsOrderFromSun.size); // 3

planetsOrderFromSun.add('Venus'); // Trying to add a duplicate
console.log(planetsOrderFromSun.size); // Still 3, Did not add the d

planetsOrderFromSun.clear();
console.log(planetsOrderFromSun.size); // 0
```

[Run this example on JSBin](#)

Weak Collections, Memory, and Garbage Collections

JavaScript Garbage Collection is a form of memory management whereby objects that are no longer referenced are automatically deleted and their resources are reclaimed.

Map and Set's references to objects are strongly held and will not allow for garbage collection. This can get expensive if maps/sets reference large objects that are no longer needed, such as DOM elements that have already been removed from the DOM.

To remedy this, ES6 also introduces two new weak collections called `WeakMap` and `weakSet`. These ES6 collections are 'weak' because they allow for objects which are no longer needed to be cleared from memory.

WeakMap

WeakMap is the third of the new ES6 collections we're covering. WeakMaps are similar to normal Maps, albeit with fewer methods and the aforementioned difference with regards to garbage collection.

```
const aboutAuthor = new WeakMap(); // Create New WeakMap
const currentAge = {}; // key must be an object
const currentCity = {}; // keys must be an object

aboutAuthor.set(currentAge, 30); // Set Key Values
aboutAuthor.set(currentCity, 'Denver'); // Key Values can be of diff

console.log(aboutAuthor.has(currentCity)); // Test if WeakMap has a
aboutAuthor.delete(currentAge); // Delete a key
```

[Run this example on JSBin](#)

Use cases

WeakMaps [have several popular use cases](#). They can be used to keep an object's private data private, and they can also be used to keep track of DOM nodes/objects.

Private data use case

The following example is from [JavaScript expert Nicholas C. Zakas](#):

```
var Person = (function() {
  var privateData = new WeakMap();
```

```

function Person(name) {
  privateData.set(this, { name: name });
}

Person.prototype.getName = function() {
  return privateData.get(this).name;
};

return Person;
}();

```

Using a `WeakMap` here simplifies the process of keeping an object's data private. It's possible to reference the `Person` object, but access to the `privateData``weakMap` is disallowed without the specific `Person` instance.

DOM nodes use case

The [Google Polymer project](#) uses `WeakMaps` in a piece of code called `PositionWalker`.

`PositionWalker` keeps track of a position within a DOM subtree, as a current node and an offset within that node.

[WeakMap is used](#) to keep track of DOM node edits, removals, and changes:

```

_makeClone() {
  this._containerClone = this.container.cloneNode(true);
  this._cloneToNodes = new WeakMap();
  this._nodesToClones = new WeakMap();

  ...

let n = this.container;
let c = this._containerClone;

// find the currentNode's clone
while (n !== null) {
  if (n === this.currentNode) {
    this._currentNodeClone = c;
  }
  this._cloneToNodes.set(c, n);
  this._nodesToClones.set(n, c);

  n = iterator.nextNode();
  c = cloneIterator.nextNode();
}

```

}

WeakSet

WeakSets are Set Collections whose elements can be garbage collected when objects they're referencing are no longer needed. WeakSets don't allow for iteration. [Their use cases are rather limited](#) (for now, at least). Most early adopters say that WeakSets can [be used to tag objects without mutating them](#). [ES6-Features.org](#) has an [example of adding and deleting elements from a WeakSet](#) in order to keep track of whether or not the objects have been marked:

```
let isMarked      = new WeakSet()
let attachedData = new WeakMap()

export class Node {
    constructor (id) { this.id = id }
    mark() { isMarked.add(this) }
    unmark() { isMarked.delete(this) }
    marked() { return isMarked.has(this) }
    set data (data) { attachedData.set(this, data) }
    get data () { return attachedData.get(this) }
}

let foo = new Node("foo")

JSON.stringify(foo) === '{"id":"foo"}'
foo.mark()
foo.data = "bar"
foo.data === "bar"
JSON.stringify(foo) === '{"id":"foo"}'

isMarked.has(foo) === true
attachedData.has(foo) === true
foo = null /* remove only reference to foo */
attachedData.has(foo) === false
isMarked.has(foo) === false
```

Map All Things? Records vs ES6 Collections

Maps and Sets are nifty new ES6 collections of key/value pairs. That said, JavaScript objects still can be used as collections in many situations. No need to switch to the new ES6 collections unless the situation calls for it.

[MDN has a nice list of questions](#) to determine when to use an object or a

keyed collection:

- Are keys usually unknown until run time, and do you need to look them up dynamically?
- Do all values have the same type, and can be used interchangeably?
- Do you need keys that aren't strings?
- Are key-value pairs often added or removed?
- Do you have an arbitrary (easily changing) amount of key-value pairs?
- Is the collection iterated?

New ES6 Collections Yield a More Usable JavaScript

JavaScript collections have previously been quite limited, but this has been remedied with ES6. These new ES6 collections will add power and flexibility to the language, as well as simplify the task of JavaScript developers who adopt them.

Chapter 3: New Array.* and Array.prototype.* Methods

by Aurelio De Rosa

In this chapter we'll discuss most of the new methods available in ES6 that work with the `Array` type, using `Array.*` and `Array.prototype.*`.

When discussing them, I'll write `Array.method()` when I describe a "class" method and `Array.prototype.method()` when I outline an "instance" method.

We'll also see some example uses and mention several polyfills for them. If you need a polyfill-them-all library, you can use [es6-shim](#) by [Paul Miller](#).

Array.from()

The first method I want to mention is `Array.from()`. It creates a new `Array` instance from an array-like or an iterable object. This method can be used to solve an old problem with array-like objects that most developers solve using this code:

```
// typically arrayLike is arguments
var arr = [].slice.call(arrayLike);
```

The syntax of `Array.from()` is shown below:

```
Array.from(arrayLike[, mapFn[, thisArg]])
```

The meaning of its parameters are:

- `arrayLike`: an array-like or an iterable object
- `mapFn`: a function to call on every element contained
- `thisArg`: a value to use as the context (`this`) of the `mapFn` function.

Now that we know its syntax and its parameters, let's see this method in action. In the code below we're going to create a function that accepts a variable

number of arguments, and returns an array containing these elements doubled:

```
function double(arr) {
  return Array.from(arguments, function(elem) {
    return elem * 2;
  });
}

const result = double(1, 2, 3, 4);

// prints [2, 4, 6, 8]
console.log(result);
```

A live demo of the previous code is [available at JSBin](#).

This method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, there are a couple of polyfills to choose from: one is available on the [method's page on MDN](#), while the other has been written by Mathias Bynens and is called [Array.from](#).

Array.prototype.find()

Another of the methods introduced is `Array.prototype.find()`. The syntax of this method is:

```
Array.prototype.find(callback[, thisArg])
```

As you can see, it accepts a callback function used to test the elements of the array and an optional argument to set the context (`this` value) of the callback function. The callback function receives three parameters:

- `element`: the current element
- `index`: the index of the current element
- `array`: the array you used to invoke the method.

This method returns a value in the array if it satisfies the provided callback function, or `undefined` otherwise. The callback is executed once for each element in the array until it finds one where a truthy value is returned. If there's more than one element in the array, that will return a truthy value, and only the first is returned.

An example usage is shown below:

```
const arr = [1, 2, 3, 4];
const result = arr.find(function(elem) { return elem > 2; });

// prints "3" because it's the first
// element greater than 2
console.log(result);
```

A live demo of the previous code is [available at JSBin](#).

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need a polyfill, one is provided on the [method's page on MDN](#).

Array.prototype.findIndex()

A method that is very similar to the previous one is `Array.prototype.findIndex()`. It accepts the same arguments but instead of returning the first element that satisfies the callback function, it returns its index. If none of the elements return a truthy value, -1 is returned. An example usage of this method is shown below:

```
const arr = [1, 2, 3, 4];
const result = arr.findIndex(function(elem) {return elem > 2;});

// prints "2" because is the index of the
// first element greater than 2
console.log(result);
```

A live demo of the previous code is [available at JSBin](#).

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need a polyfill, one can be found on the [method's page on MDN](#).

Array.prototype.keys()

Yet another method introduced in this new version of JavaScript is `Array.prototype.keys()`. This method returns a new `Array Iterator` (not an array) containing the keys of the array's values. If you want to learn more about array iterators, you can refer to the [specifications](#) or the [MDN page](#).

The syntax of `Array.prototype.keys()` is shown below:

```
Array.prototype.keys()
```

An example of use is the following:

```
const arr = [1, 2, 3, 4];
const iterator = arr.keys();

// prints "0, 1, 2, 3", one at a time, because the
// array contains four elements and these are their indexes
let index = iterator.next();
while(!index.done) {
  console.log(index.value);
  index = iterator.next();
}
```

A live demo is [available at JSBin](#).

`Array.prototype.keys()` in Node and all modern browsers, with the exception of Internet Explorer.

Array.prototype.values()

In the same way we can retrieve the keys of an array, we can retrieve its values using `Array.prototype.values()`. This method is similar to `Array.prototype.keys()` but the difference is that it returns an `Array Iterator` containing the values of the array.

The syntax of this method is shown below:

```
Array.prototype.values()
```

An example use is shown below:

```
const arr = [1, 2, 3, 4];
const iterator = arr.values();

// prints "1, 2, 3, 4", one at a time, because the
// array contains these four elements
let index = iterator.next();
while(!index.done) {
  console.log(index.value);
  index = iterator.next();
```

```
}
```

A live demo of the previous code is [available at JSBin](#).

The `Array.prototype.values()` is currently [not implemented in most browsers](#). In order for you to use it you need to transpile it via Babel.

Array.prototype.fill()

If you've worked in the PHP world (like me), you'll recall a function named `array_fill()` that was missing in JavaScript. In ES6 this method is no longer missing. `Array.prototype.fill()` fills an array with a specified value optionally from a start index to an end index (not included).

The syntax of this method is the following:

```
Array.prototype.fill(value[, start[, end]])
```

The default values for `start` and `end` are respectively `0` and the `length` of the array. These parameters can also be negative. If `start` or `end` are negative, the positions are calculated starting from the end of the array.

An example of use of this method is shown below:

```
const arr = new Array(6);
// This statement fills positions from 0 to 2
arr.fill(1, 0, 3);
// This statement fills positions from 3 up to the end of the array
arr.fill(2, 3);

// prints [1, 1, 1, 2, 2, 2]
console.log(arr);
```

A live demo of the previous code is [available at JSBin](#).

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. As polyfills you can employ the one on the [method's page on MDN](#), or [the polyfill developed by Addy Osmani](#).

Conclusion

In this chapter we've discussed several of the new methods introduced in ES6 that work with arrays. With the exception of `Array.prototype.values()`, they enjoy good browser support and can be used today!

Chapter 4: New String Methods — String.prototype.*

by Aurelio de Rosa

In my previous chapter on [ES6 array methods](#), I introduced the new methods available in ECMAScript 6 that work with the `Array` type. In this tutorial, you'll learn about new ES6 methods that work with strings: `String.prototype.*`

We'll develop several examples, and mention the polyfills available for them. Remember that if you want to polyfill them all using a single library, you can employ [es6-shim](#) by [Paul Miller](#).

String.prototype.startsWith()

One of the most-used functions in every modern programming language is the one to verify if a string starts with a given substring. Before ES6, JavaScript had no such function, meaning you had to write it yourself. The following code shows how developers usually polyfilled it:

```
if (typeof String.prototype.startsWith !== 'function') {  
  String.prototype.startsWith = function (str){  
    return this.indexOf(str) === 0;  
  };  
}
```

Or, alternatively:

```
if (typeof String.prototype.startsWith !== 'function') {  
  String.prototype.startsWith = function (str){  
    return this.substring(0, str.length) === str;  
  };  
}
```

These snippets are still valid, but they don't reproduce exactly what the newly available `String.prototype.startsWith()` method does. The new method has the following syntax:

```
String.prototype.startsWith(searchString[, position]);
```

You can see that, in addition to a substring, it accepts a second argument. The searchString parameter specifies the substring you want to verify is the start of the string. position indicates the position at which to start the search. The default value of position is 0. The method returns true if the string starts with the provided substring, and false otherwise. Remember that the method is case sensitive, so “Hello” is different from “hello”.

An example use of this method is shown below:

```
const str = 'hello!';
let result = str.startsWith('he');

// prints "true"
console.log(result);

// verify starting from the third character
result = str.startsWith('ll', 2);

// prints "true"
console.log(result);
```

A live demo of the previous code is [available at JSBin](#).

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, a polyfill for this method can be found in the [method's page on MDN](#). [Another polyfill](#) has also been developed by Mathias Bynens.

String.prototype.endsWith()

In addition to `String.prototype.startsWith()`, ECMAScript 6 introduces the `String.prototype.endsWith()` method. It verifies that a string terminates with a given substring. The syntax of this method, shown below, is very similar to `String.prototype.startsWith()`:

```
String.prototype.endsWith(searchString[, position]);
```

As you can see, this method accepts the same parameters as `String.prototype.startsWith()`, and also returns the same type of values.

A difference is that the position parameter lets you search within the string as if the string were only this long. In other words, if we have the string house and we call the method with 'house'.endsWith('us', 4), we obtain true, because it's like we actually had the string hous (note the missing "e").

An example use of this method is shown below:

```
const str = 'hello!';
const result = str.endsWith('lo!');

// prints "true"
console.log(result);

// verify as if the string was "hell"
result = str.endsWith('lo!', 5);

// prints "false"
console.log(result);
```

A live demo of the previous snippet is [available at JSBin](#).

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, a polyfill for this method can be found in the [method's page on MDN](#). [Another polyfill](#) has been developed by Mathias Bynens.

String.prototype.includes()

While we're talking about verifying if one string is contained in another, let me introduce you to the `String.prototype.includes()` method. It returns `true` if a string is contained in another, no matter where, and `false` otherwise.

Its syntax is shown below:

```
String.prototype.includes(searchString[, position]);
```

The meaning of the parameters is the same as for `String.prototype.startsWith()`, so I won't repeat them. An example use of this method is shown below:

```
const str = 'Hello everybody, my name is Aurelio De Rosa.';
let result = str.includes('Aurelio');
```

```
// prints "true"
console.log(result);

result = str.includes('Hello', 10);

// prints "false"
console.log(result);
```

You can find a live demo [at JSBin](#).

`String.prototype.includes()` is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, as with the other methods discussed in this tutorial, you can find [a polyfill provided by Mathias Bynens](#) (this guy knows how to do his job!) and [another on the Mozilla Developer Network](#).

String.prototype.repeat()

Let's now move on to another type of method. `String.prototype.repeat()` is a method that returns a new string containing the same string it was called upon but repeated a specified number of times. The syntax of this method is the following:

```
String.prototype.repeat(times);
```

The `times` parameter indicates the number of times the string must be repeated. If you pass zero you'll obtain an empty string, while if you pass a negative number or infinity you'll obtain a `RangeError`.

An example use of this method is shown below:

```
const str = 'hello';
let result = str.repeat(3);

// prints "hellohellohello"
console.log(result);

result = str.repeat(0);

// prints ""
console.log(result);
```

A live demo of the previous code is [available at JSBin](#).

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, two polyfills are available for this method: [the one developed by Mathias Bynens](#) and [another on the Mozilla Developer Network](#).

String.raw()

The last method I want to cover in this tutorial is `String.raw()`. It's defined as "a tag function of template strings". It's interesting, because it's kind of a replacement for templating libraries, although I'm not 100% sure it can scale enough to actually replace those libraries. However, the idea is basically the same as we'll see shortly. What it does is to compile a string and replace every placeholder with a provided value.

Its syntax is the following (note the backticks):

```
String.raw`templateString`
```

The `templateString` parameter represents the string containing the template to process.

To better understand this concept, let's see a concrete example:

```
const name = 'Aurelio De Rosa';
const result = String.raw`Hello, my name is ${name}`;

// prints "Hello, my name is Aurelio De Rosa" because ${name}
// has been replaced with the value of the name variable
console.log(result);
```

A live demo of the previous code is [available at JSBin](#).

The method is supported in Node and all modern browsers, with the exception of Opera and Internet Explorer. If you need to support older browsers, you can employ a polyfill, such as [this one available on npm](#).

Conclusion

In this tutorial, you've learned about several new methods introduced in ECMAScript 6 that work with strings. Other methods that we haven't covered are [`String.fromCodePoint\(\)`](#), [`String.prototype.codePointAt\(\)`](#), and [`String.prototype.normalize\(\)`](#). I hope you enjoyed the chapter and that you'll continue to follow our channel to learn more about ECMAScript 6.

Chapter 5: New Number Methods

by Aurelio de Rosa

This chapter covers new and improved number methods in ES6 (ECMAScript 6).

I'll introduce you to the new methods and constants added to the `Number` data type. Some of the number methods covered, as we'll see, aren't new at all, but they've been improved and/or moved under the right object (for example, `isNaN()`). As always, we'll also put the new knowledge acquired into action with some examples. So, without further ado, let's get started.

Number.isInteger()

The first method I want to cover is `Number.isInteger()`. It's a new addition to JavaScript, and this is something you may have defined and used by yourself in the past. It determines whether the value passed to the function is an integer or not. This method returns `true` if the passed value is an integer, and `false` otherwise. The implementation of this method was pretty easy, but it's still good to have it natively. One of the possible solutions to recreate this function is:

```
Number.isInteger = Number.isInteger || function (number) {  
    return typeof number === 'number' && number % 1 === 0;  
};
```

Just for fun, I tried to recreate this function and I ended up with a different approach:

```
Number.isInteger = Number.isInteger || function (number) {  
    return typeof number === 'number' && Math.floor(number) === number  
};
```

Both these functions are good and useful but they don't respect the ECMAScript 6 specifications. So, if you want to polyfill this method, you need something a little bit more complex, as we'll see shortly. For the moment, let's start by discovering the syntax of `Number.isInteger()`:

`Number.isInteger(number)`

The `number` argument represents the value you want to test.

Some examples of the use of this method are shown below:

```
// prints 'true'  
console.log(Number.isInteger(19));  
  
// prints 'false'  
console.log(Number.isInteger(3.5));  
  
// prints 'false'  
console.log(Number.isInteger([1, 2, 3]));
```

A live demo of the previous code is [available at JSBin](#).

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, you can employ a polyfill, such as the one available on the Mozilla Developer Network on the [methods page](#). This is also reproduced below for your convenience:

```
if (!Number.isInteger) {  
    Number.isInteger = function isInteger (nVal) {  
        return typeof nVal === 'number' &&  
            isFinite(nVal) &&  
            nVal > -9007199254740992 &&  
            nVal < 9007199254740992 &&  
            Math.floor(nVal) === nVal;  
    };  
}
```

`Number.isNaN()`

If you've written any JavaScript code in the past, this method shouldn't be new to you. For a while now, JavaScript has had a method called `isNaN()` that's exposed through the `window` object. This method tests if a value is equal to `NaN`, in which case it returns `true`, or otherwise `false`. The problem with `window.isnan()` is that it has an issue in that it also returns `true` for values that *converted* to a number will be `NaN`. To give you a concrete idea of this issue, all the following statements return `true`:

```
// prints 'true'
```

```
console.log(window.isNaN(0/0));  
// prints 'true'  
console.log(window.isNaN('test'));  
// prints 'true'  
console.log(window.isNaN(undefined));  
// prints 'true'  
console.log(window.isNaN({prop: 'value'}));
```

What you might need is a method that returns `true` only if the `NaN` value is passed. That's why ECMAScript 6 has introduced the `Number.isNaN()` method. Its syntax is pretty much what you'd expect:

```
Number.isNaN(value)
```

Here, `value` is the value you want to test. Some example uses of this method are shown below:

```
// prints 'true'  
console.log(Number.isNaN(0/0));  
  
// prints 'true'  
console.log(Number.isNaN(NaN));  
  
// prints 'false'  
console.log(Number.isNaN(undefined));  
  
// prints 'false'  
console.log(Number.isNaN({prop: 'value'}));
```

As you can see, testing the same values we obtain different results.

A live demo of the previous snippet is [available at JSBin](#).

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you want to support other browsers, a very simple polyfill for this method is the following:

```
Number.isNaN = Number.isNaN || function (value) {  
    return value !== value;  
};
```

The reason this works is because `NaN` is the only non-reflexive value in

JavaScript, which means that it's the only value that isn't equal to itself.

Number.isFinite()

This method shares the same story as the previous one. In JavaScript there's a method called `window.isFinite()` that tests if a value passed is a finite number or not. Unfortunately, it also returns `true` for values that *converted* to a number will be a finite number. Examples of this issue are demonstrated below:

```
// prints 'true'  
console.log(window.isFinite(10));  
  
// prints 'true'  
console.log(window.isFinite(Number.MAX_VALUE));  
  
// prints 'true'  
console.log(window.isFinite(null));  
  
// prints 'true'  
console.log(window.isFinite([]));
```

For this reason, in ECMAScript 6 there's a method called `isFinite()` that belongs to `Number`. Its syntax is the following:

```
Number.isFinite(value)
```

Here, `value` is the value you want to test. If you test the same values from the previous snippet, you can see that the results are different:

```
// prints 'true'  
console.log(Number.isFinite(10));  
  
// prints 'true'  
console.log(Number.isFinite(Number.MAX_VALUE));  
  
// prints 'false'  
console.log(Number.isFinite(null));  
  
// prints 'false'  
console.log(Number.isFinite([]));
```

A live demo of the previous snippet is [available at JSBin](#).

The method is supported in Node and all modern browsers, with the exception of

Internet Explorer. You can find a polyfill for it on the [method's page on MDN](#).

Number.isSafeInteger()

The `Number.isSafeInteger()` method is a completely new addition to ES6. It tests whether the value passed is a number that is a safe integer, in which case it returns `true`. A safe integer is defined as an integer that satisfies both the following two conditions:

- the number can be exactly represented as an IEEE-754 double precision number
- the IEEE-754 representation of the number can't be the result of rounding any other integer to fit the IEEE-754 representation.

Based on this definition, safe integers are all the integers from $-(2^{53} - 1)$ inclusive to $2^{53} - 1$ inclusive. These values are important and we'll discuss them a little more at the end of this section.

The syntax of this method is:

```
Number.isSafeInteger(value)
```

Here, `value` is the value you want to test. A few example uses of this method are shown below:

```
// prints 'true'
console.log(Number.isSafeInteger(5));

// prints 'false'
console.log(Number.isSafeInteger('19'));

// prints 'false'
console.log(Number.isSafeInteger(Math.pow(2, 53)));

// prints 'true'
console.log(Number.isSafeInteger(Math.pow(2, 53) - 1));
```

A live demo of this code is [available at JSBin](#).

The `Number.isSafeInteger()` is supported in Node and all modern browsers, with the exception of Internet Explorer. A polyfill for this method, extracted from [es6-shim](#) by [Paul Miller](#), is:

```
Number.isSafeInteger = Number.isSafeInteger || function (value) {
  return Number.isInteger(value) && Math.abs(value) <= Number.MAX_SAFE_INTEGER;
};
```

Note that this polyfill relies on the `Number.isInteger()` method discussed before, so you need to polyfill the latter as well to use this one.

ECMAScript 6 also introduced two related constant values:

`Number.MAX_SAFE_INTEGER` and `Number.MIN_SAFE_INTEGER`. The former represents the maximum safe integer in JavaScript — that is, $2^{53} - 1$ — while the latter the minimum safe integer, which is $-(2^{53} - 1)$. As you might note, these are the same values I cited earlier.

Number.parseInt() and Number.parseFloat()

The `Number.parseInt()` and `Number.parseFloat()` methods are covered in the same section because, unlike other similar methods mentioned in this chapter, they already existed in a previous version of ECMAScript, but aren't different from their old global version. So, you can use them in the same way you've done so far and you can expect the same results. The purpose of adding these methods to `Number` is the modularization of globals.

For the sake of completeness, I'm reporting their syntax:

```
// Signature of Number.parseInt
Number.parseInt(string, radix)

// Signature of Number.parseFloat
Number.parseFloat(string)
```

Here, `string` represents the value you want to parse and `radix` is the radix you want to use to convert `string`.

The following snippet shows a few example uses:

```
// Prints '-3'
console.log(Number.parseInt('-3'));

// Prints '4'
console.log(Number.parseInt('100', 2));

// Prints 'NaN'
```

```
console.log(Number.parseInt('test'));  
  
// Prints 'NaN'  
console.log(Number.parseInt({}));  
  
// Prints '42.1'  
console.log(Number.parseFloat('42.1'));  
  
// Prints 'NaN'  
console.log(Number.parseFloat('test'));  
  
// Prints 'NaN'  
console.log(Number.parseFloat({}));
```

A live demo of this code is [available at JSBin](#).

These methods are supported in Node and all modern browsers, with the exception of Internet Explorer. In case you want to polyfill them, you can simply call their related global method as listed below:

```
// Polyfill Number.parseInt  
Number.parseInt = Number.parseInt || function () {  
    return window.parseInt.apply(window, arguments);  
};  
  
// Polyfill Number.parseFloat  
Number.parseFloat = Number.parseFloat || function () {  
    return window.parseFloat.apply(window, arguments);  
};
```

ES6 Number Methods: Wrapping Up

In this tutorial we've covered the new methods and constants added in ECMAScript 6 that work with the `Number` data type. It's worth noting that ES6 has also added another constant that I haven't mentioned so far. This is `Number.EPSILON` and "represents the difference between one and the smallest value greater than one that can be represented as a `Number`." With this last note, we've concluded our journey for the `Number` data type.

Chapter 6: ES6 Arrow Functions: Fat and Concise Syntax in JavaScript

by Kyle Pennell

Arrow functions were introduced with ES6 as a new syntax for writing JavaScript functions. They save developers time and simplify function scope. The good news is that many major modern browsers [support the use of arrow functions](#).

This chapter will cover the details of arrow functions — how to use them, common syntaxes, common use cases, and gotchas/pitfalls.

What Are Arrow Functions?

Arrow functions – also called “fat arrow” functions, from CoffeeScript ([a transcompiled language](#)) — are a more concise syntax for writing function expressions. They utilize a new token, `=>`, that looks like a fat arrow. Arrow functions are anonymous and change the way `this` binds in functions.

Arrow functions make our code more concise, and simplify function scoping and the `this` keyword. They are one-line mini functions which work much like [Lambdas in other languages like C# or Python](#). (See also [Lambdas in JavaScript](#)). By using arrow functions, we avoid having to type the `function` keyword, `return` keyword (it’s implicit in arrow functions), and curly brackets.

Using Arrow Functions

There are a variety of syntaxes available in arrow functions, of which [MDN has a thorough list](#). We’ll cover the common ones here to get you started. Let’s compare how ES5 code with function expressions can now be written in ES6 using arrow functions.

Basic Syntax with Multiple Parameters ([from MDN](#))

```
// (param1, param2, paramN) => expression  
  
// ES5  
var multiplyES5 = function(x, y) {  
  return x * y;  
};  
  
// ES6  
const multiplyES6 = (x, y) => { return x * y };
```

[Code Example at JSBin.](#)

The arrow function example above allows a developer to accomplish the same result with fewer lines of code and approximately half the typing.

Curly brackets aren't required if only one expression is present. The preceding example could also be written as:

```
const multiplyES6 = (x, y) => x * y;
```

Basic Syntax with One Parameter

Parentheses are optional when only one parameter is present

```
//ES5  
var phraseSplitterES5 = function phraseSplitter(phrase) {  
  return phrase.split(' ');  
};  
  
//ES6  
const phraseSplitterEs6 = phrase => phrase.split(" ");  
  
console.log(phraseSplitterEs6("ES6 Awesomeness")); // ["ES6", "Awes
```

[Code Example at JSBin.](#)

No Parameters

Parentheses are required when no parameters are present.

```
//ES5  
var docLogEs5 = function docLog() {
```

```
    console.log(document);
};

//ES6
var docLogEs6 = () => { console.log(document); };
docLogEs6(); // #document... <html> ...
```

[Code Example at JSBin.](#)

Object Literal Syntax

Arrow functions, like function expressions, can be used to return an object literal expression. The only caveat is that the body needs to be wrapped in parentheses, in order to distinguish between a block and an object (both of which use curly brackets).

```
//ES5
var setNameIdsEs5 = function setNameIds(id, name) {
  return {
    id: id,
    name: name
  };
};

// ES6
var setNameIdsEs6 = (id, name) => ({ id: id, name: name });

console.log(setNameIdsEs6 (4, "Kyle")); // Object {id: 4, name: "K
```

[Code Example at JSBin.](#)

Use Cases for Arrow Functions

Now that we've covered the basic syntaxes, let's get into how arrow functions are used.

One common use case for arrow functions is array manipulation and the like. It's common that you'll need to map or reduce an array. Take this simple array of objects:

```
const smartPhones = [
  { name:'iphone', price:649 },
  { name:'Galaxy S6', price:576 },
```

```
{ name:'Galaxy Note 5', price:489 }  
];
```

We could create an array of objects with just the names or prices by doing this in ES5:

```
// ES5  
var prices = smartPhones.map(function(smartPhone) {  
    return smartPhone.price;  
});  
  
console.log(prices); // [649, 576, 489]
```

An arrow function is more concise and easier to read:

```
// ES6  
const prices = smartPhones.map(smartPhone => smartPhone.price);  
console.log(prices); // [649, 576, 489]
```

[Code Example at JSBin.](#)

Here's another example using the [array filter method](#):

```
const array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];  
  
// ES5  
var divisibleByThrreeES5 = array.filter(function (v){  
    return v % 3 === 0;  
});  
  
// ES6  
const divisibleByThrreeES6 = array.filter(v => v % 3 === 0);  
  
console.log(divisibleByThrreeES6); // [3, 6, 9, 12, 15]
```

[Code Example at JSBin.](#)

Promises and Callbacks

Code that makes use of asynchronous callbacks or promises often contains a great deal of `function` and `return` keywords. When using promises, these function expressions will be used for chaining. Here's a simple example of [chaining promises from the MSDN docs](#):

```
// ES5
```

```
aAsync().then(function() {
  returnbAsync();
}).then(function() {
  returncAsync();
}).done(function() {
  finish();
});
```

This code is simplified, and arguably easier to read using arrow functions:

```
// ES6
aAsync().then(() => bAsync()).then(() => cAsync()).done(() => finish)
```

Arrow functions should similarly simplify callback-laden NodeJS code.

What's the meaning of this?!

The other benefit of using arrow functions with promises/callbacks is that it reduces the confusion surrounding the `this` keyword. In code with multiple nested functions, it can be difficult to keep track of and remember to bind the correct `this` context. In ES5, you can use workarounds like the `.bind` method ([which is slow](#)) or creating a closure using `var self = this;`.

Because arrow functions allow you to retain the scope of the caller inside the function, you don't need to create `self = this` closures or use `bind`.

Developer [Jack Franklin](#) provides an excellent [practical example of using the arrow function lexical this to simplify a promise](#):

Without Arrow functions, the promise code needs to be written something like this:

```
// ES5
API.prototype.get = function(resource) {
  var self = this;
  return new Promise(function(resolve, reject) {
    http.get(self.uri + resource, function(data) {
      resolve(data);
    });
  });
};
```

Using an arrow function, the same result can be achieved more concisely and clearly:

```
// ES6
API.prototype.get = function(resource) {
  return new Promise((resolve, reject) => {
    http.get(this.uri + resource, function(data) {
      resolve(data);
    });
  });
};
```

You can use function expressions if you need a dynamic `this` and arrow functions for a lexical `this`.

Gotchas and Pitfalls of Arrow Functions

The new arrow functions bring a helpful function syntax to ECMAScript, but as with any new feature, they come with their own pitfalls and gotchas.

Kyle Simpson, a JavaScript developer and writer, felt there were enough pitfalls with Arrow Functions to [warrant this flow chart when deciding to use them](#). He argues there are too many confusing rules/syntaxes with arrow functions. Others have suggested that using arrow functions saves typing but ultimately makes code more difficult to read. All those `function` and `return` statements might make it easier to read multiple nested functions or just function expressions in general.

Developer opinions vary on just about everything, including arrow functions. For the sake of brevity, here are a couple things you need to watch out for when using arrow functions.

More about `this`

As was mentioned previously, the `this` keyword works differently in arrow functions. The methods [`call\(\)`, `apply\(\)`, and `bind\(\)`](#) will not change the value of `this` in arrow functions. (In fact, the value of `this` inside a function simply can't be changed; it will be the same value as when the function was called.) If you need to bind to a different value, you'll need to use a function expression.

Constructors

Arrow functions can't be used as [`constructors`](#) as other functions can. Don't use

them to create similar objects as you would with other functions. If you attempt to use `new` with an arrow function, it will throw an error. Arrow functions, like [built-in functions](#) (aka methods), don't have a `prototype` property or other internal methods. Because constructors are generally used to create class-like objects in JavaScript, you should use the new [ES6 classes](#) instead.

Generators

Arrow functions are designed to be lightweight and can't be used as [generators](#). Using the `yield` keyword in ES6 will throw an error. Use [ES6 generators](#) instead.

Arguments object

Arrow functions don't have the local variable `arguments` as do other functions. The `arguments` object is an array-like object that allows developers to dynamically discover and access a function's arguments. This is helpful because JavaScript functions can take an unlimited number of arguments. Arrow functions do not have this object.

How Much Use Is There for Arrow Functions?

Arrow functions have been [called one of the quickest wins](#) with ES6. Developer [Lars Schöning](#) lays out how his team decided [where to use arrow functions](#):

- Use `function` in the global scope and for `Object.prototype` properties.
- Use `class` for object constructors.
- Use `=>` everywhere else.

Arrow functions, like [let and const](#), will likely become the default functions unless function expressions or declarations are necessary. To get a sense for how much arrow functions can be used, [Kevin Smith](#), counted [function expressions in various popular libraries/frameworks](#) and found that roughly [55% of function expressions](#) would be candidates for arrow functions.

Arrow functions are here: they're powerful, concise, and developers love them. Perhaps it's time for you to start using them!

Chapter 7: Symbols and Their Uses

by Nilson Jacques

While ES2015 has introduced many language features that have been on developers' wish lists for some time, there are some new features that are less well known and understood, and the benefits of which are much less clear — such as symbols.

The symbol is a new primitive type, a unique token that's guaranteed never to clash with another symbol. In this sense, you could think of symbols as a kind of [UUID](#) (universally unique identifier). Let's look at how symbols work, and what we can do with them.

Creating New Symbols

Creating new symbols is very straightforward and is simply a case of calling the [Symbol](#) function. Note that this is a just a standard function and not an object constructor. Trying to call it with the new operator will result in a `TypeError`. Every time you call the `Symbol` function, you'll get a new and completely unique value.

```
const foo = Symbol();
const bar = Symbol();

foo === bar
// <-- false
```

Symbols can also be created with a label, by passing a string as the first argument. The label doesn't affect the value of the symbol, but is useful for debugging, and is shown if the symbol's `toString()` method is called. It's possible to create multiple symbols with the same label, but there's no advantage to doing so and this would probably just lead to confusion.

```
let foo = Symbol('baz');
let bar = Symbol('baz');
```

```
foo === bar
// <-- false
console.log(foo);
// <-- Symbol(baz)
```

What Can I Do With Symbols?

Symbols could be a good replacement for strings or integers as class/module constants:

```
class Application {
  constructor(mode) {
    switch (mode) {
      case Application.DEV:
        // Set up app for development environment
        break;
      case Application.PROD:
        // Set up app for production environment
        break;
      case default:
        throw new Error('Invalid application mode: ' + mode);
    }
  }
}

Application.DEV = Symbol('dev');
Application.PROD = Symbol('prod');

// Example use
const app = new Application(Application.DEV);
```

String and integers are not unique values; values such as the number 2 or the string development, for example, could also be in use elsewhere in the program for different purposes. Using symbols means we can be more confident about the value being supplied.

Another interesting use of symbols is as object property keys. If you've ever used a JavaScript object as a [hashmap](#) (an associative array in PHP terms, or dictionary in Python) you'll be familiar with getting/setting properties using the bracket notation:

```
const data = [];
data['name'] = 'Ted Mosby';
```

```
data['nickname'] = 'Teddy Westside';
data['city'] = 'New York';
```

Using the bracket notation, we can also use a symbol as a property key. There are a couple of advantages to doing so. First, you can be sure that symbol-based keys will never clash, unlike string keys, which might conflict with keys for existing properties or methods of an object. Second, they won't be enumerated in `for ... in` loops, and are ignored by functions such as `Object.keys()`, `Object.getOwnPropertyNames()` and `JSON.stringify()`. This makes them ideal for properties that you don't want to be included when serializing an object.

```
const user = {};
const email = Symbol();

user.name = 'Fred';
user.age = 30;
user[email] = 'fred@example.com';

Object.keys(user);
// <-- Array [ "name", "age" ]

Object.getOwnPropertyNames(user);
// <-- Array [ "name", "age" ]

JSON.stringify(user);
// <-- '{"name": "Fred", "age": 30}'
```

It's worth noting, however, that using symbols as keys doesn't guarantee privacy. There are some new tools provided to allow you to access symbol-based property keys. `Object.getOwnPropertySymbols()` returns an array of any symbol-based keys, while `Reflect.ownKeys()` will return an array of all keys, including symbols.

```
Object.getOwnPropertySymbols(user);
// <-- Array [ Symbol() ]

Reflect.ownKeys(user)
// <-- Array [ "name", "age", Symbol() ]
```

Well-known Symbols

Because symbol-keyed properties are effectively invisible to pre-ES6 code,

they're ideal for adding new functionality to JavaScript's existing types without breaking backwards compatibility. The so-called "well-known" symbols are predefined properties of the `Symbol` function that are used to customize the behavior of certain language features, and are used to implement new functionality such as iterators.

`Symbol.iterator` is a well-known symbol that's used to assign a special method to objects, which allows them to be iterated over:

```
const band = ['Freddy', 'Brian', 'John', 'Roger'];
const iterator = band[Symbol.iterator]();

iterator.next().value;
// <-- { value: "Freddy", done: false }
iterator.next().value;
// <-- { value: "Brian", done: false }
iterator.next().value;
// <-- { value: "John", done: false }
iterator.next().value;
// <-- { value: "Roger", done: false }
iterator.next().value;
// <-- { value: undefined, done: true }
```

The built-in types `String`, `Array`, `TypedArray`, `Map` and `Set` all have a default `Symbol.iterator` method which is called when an instance of one of these types is used in a `for ... of` loop, or with the spread operator. Browsers are also starting to use the `Symbol.iterator` key to allow DOM structures such as `NodeList` and `HTMLCollection` to be iterated over in the same way.

The Global Registry

The specification also defines a runtime-wide symbol registry, which means that you can store and retrieve symbols across different execution contexts, such as between a document and an embedded iframe or service worker.

`Symbol.for(key)` retrieves the symbol for a given key from the registry. If a symbol doesn't exist for the key, a new one is returned. As you might expect, subsequent calls for the same key will return the same symbol.

`Symbol.keyFor(symbol)` allows you to retrieve the key for a given symbol. Calling the method with a symbol that doesn't exist in the registry returns `undefined`:

```
const debbie = Symbol.for('user');
const mike   = Symbol.for('user');

debbie === mike
// <-- true

Symbol.keyFor(debbie);
// <-- "user"
```

Use Cases

There are a couple of use cases where using symbols provides an advantage. One, which I touched on earlier in the chapter, is when you want to add “hidden” properties to objects that won’t be included when the object is serialized.

Library authors could also use symbols to safely augment client objects with properties or methods without having to worry about overwriting existing keys (or having their keys overwritten by other code). For example, widget components (such as date pickers) are often initialized with various options and state that needs to be stored somewhere. Assigning the widget instance to a property of the DOM element object is not ideal, because that property could potentially clash with another key. Using a symbol-based key neatly side steps this issue and ensures that your widget instance won’t be overwritten. See the Mozilla Hacks blog post [ES6 in Depth: Symbols](#) for a more detailed exploration of this idea.

Browser Support

If you want to experiment with symbols, mainstream browser support [is quite good](#). As you can see, the current versions of Chrome, Firefox, Microsoft Edge and Opera support the Symbol type natively, along with Android 5.1 and iOS 9 on mobile devices. There are also [polyfills available](#) if you need to support older browsers.

Conclusion

Although the primary reason for the introduction of symbols seems to have been to facilitate adding new functionality to the language without breaking existing code, they do have some interesting uses. It’s worthwhile for all developers to

have at least a basic knowledge of them, and be familiar with the most commonly used, well-known symbols and their purpose.

Chapter 8: How to Use Proxies

by Craig Buckler

In computing terms, proxies sit between you and the things you're communicating with. The term is most often applied to a proxy server — a device between the web browser (Chrome, Firefox, Safari, Edge etc.) and the web server (Apache, Nginx, IIS etc.) where a page is located. The proxy server can modify requests and responses. For example, it can increase efficiency by caching regularly accessed assets and serving them to multiple users.

ES6 proxies sit between your code and an object. A proxy allows you to perform meta-programming operations such as intercepting a call to inspect or change an object's property.

The following terminology is used in relation to ES6 proxies:

target The original object the proxy will virtualize. This could be a JavaScript object such as the jQuery library or native objects such as arrays or even another proxies.

handler An object which implements the proxy's behavior using...

traps Functions defined in the handler which provide access to the target when specific properties or methods are called.

It's best explained with a simple example. We'll create a target object named `target` which has three properties:

```
const target = {
  a: 1,
  b: 2,
  c: 3
};
```

We'll now create a handler object which intercepts all get operations. This returns the target's property when it's available or 42 otherwise:

```
const handler = {
  get: function(target, name) {
    return (
      name in target ? target[name] : 42
    );
  }
};
```

We now create a new Proxy by passing the target and handler objects. Our code can interact with the proxy rather than accessing the target object directly:

```
const proxy = new Proxy(target, handler);

console.log(proxy.a); // 1
console.log(proxy.b); // 2
console.log(proxy.c); // 3
console.log(proxy.meaningOfLife); // 42
```

Let's expand the proxy handler further so it only permits single-character properties from a to z to be set:

```
const handler = {
  get: function(target, name) {
    return (name in target ? target[name] : 42);
  },

  set: function(target, prop, value) {
    if (prop.length == 1 && prop >= 'a' && prop <= 'z') {
      target[prop] = value;
      return true;
    }
    else {
      throw new ReferenceError(prop + ' cannot be set');
      return false;
    }
  }
};

const proxy = new Proxy(target, handler);

proxy.a = 10;
proxy.b = 20;
proxy.ABC = 30;
// Exception: ReferenceError: ABC cannot be set
```

Proxy Trap Types

We've seen the get and set in action which are likely to be the most useful traps. However, there are several other trap types you can use to supplement proxy handler code:

- **construct(target, argList)** Traps the creation of a new object with the new operator.
- **get(target, property)** Traps Object.get() and must return the property's value.
- **set(target, property, value)** Traps Object.set() and must set the property value. Return true if successful. In strict mode, returning false will throw a TypeError exception.
- **deleteProperty(target, property)** Traps a delete operation on an object's property. Must return either true or false.
- **apply(target, thisArg, argList)** Traps object function calls.
- **has(target, property)** Traps in operators and must return either true or false.
- **ownKeys(target)** Traps Object.getOwnPropertyNames() and must return an enumerable object.
- **getPrototypeOf(target)** Traps Object.getPrototypeOf() and must return the prototype's object or null.
- **setPrototypeOf(target, prototype)** Traps Object.setPrototypeOf() to set the prototype object. No value is returned.
- **isExtensible(target)** Traps Object.isExtensible(), which determines whether an object can have new properties added. Must return either true or false.
- **preventExtensions(target)** Traps Object.preventExtensions(), which prevents new properties from being added to an object. Must return either true or false.
- **getOwnPropertyDescriptor(target, property)** Traps Object.getOwnPropertyDescriptor(), which returns undefined or a property descriptor object with attributes for value, writable, get, set, configurable and enumerable.
- **defineProperty(target, property, descriptor)** Traps Object.defineProperty() which defines or modifies an object property. Must return true if the target property was successfully defined or false if not.

Proxy Example 1: Profiling

Proxies allow you to create generic wrappers for any object without having to change the code within the target objects themselves.

In this example, we'll create a profiling proxy which counts the number of times a property is accessed. First, we require a `makeProfiler` factory function which returns the `Proxy` object and retains the count state:

```
// create a profiling Proxy
function makeProfiler(target) {

  const
    count = {},
    handler = {

      get: function(target, name) {
        if (name in target) {
          count[name] = (count[name] || 0) + 1;
          return target[name];
        }
      }
    };

  return {
    proxy: new Proxy(target, handler),
    count: count
  }
};
```

We can now apply this proxy wrapper to any object or another proxy. For example:

```
const myObject = {
  h: 'Hello',
  w: 'World'
};

// create a myObject proxy
const pObj = makeProfiler(myObject);

// access properties
console.log(pObj.proxy.h); // Hello
console.log(pObj.proxy.h); // Hello
console.log(pObj.proxy.w); // World
console.log(pObj.count.h); // 2
console.log(pObj.count.w); // 1
```

While this is a trivial example, imagine the effort involved if you had to perform

property access counts in several different objects without using proxies.

Proxy Example 2: Two-way Data Binding

Data binding synchronizes objects. It's typically used in JavaScript MVC libraries to update an internal object when the DOM changes and vice versa.

Presume we have an input field with an ID of `inputname`:

```
<input type="text" id="inputname" value="" />
```

We also have a JavaScript object named `myUser` with an `id` property which references this input:

```
// internal state for #inputname field
const myUser = {
  id: 'inputname',
  name: ''
};
```

Our first objective is to update `myUser.name` when a user changes the input value. This can be achieved with an `onchange` event handler on the field:

```
inputChange(myUser);

// bind input to object
function inputChange(myObject) {
  if (!myObject || !myObject.id) return;

  const input = document.getElementById(myObject.id);
  input.addEventListener('onchange', function(e) {
    myObject.name = input.value;
  });
}
```

Our next objective is to update the input field when we modify `myUser.name` within JavaScript code. This is not as simple, but proxies offer a solution:

```
// proxy handler
const inputHandler = {
  set: function(target, prop, newValue) {
    if (prop == 'name' && target.id) {
      // update object property
      target[prop] = newValue;
    }
  }
};
```

```
// update input field value
document.getElementById(target.id).value = newValue;
return true;
}
else return false;
}

// create proxy
const myUserProxy = new Proxy(myUser, inputHandler);

// set a new name
myUserProxy.name = 'Craig';
console.log(myUserProxy.name); // Craig
console.log(document.getElementById('inputname').value); // Craig
```

This may not be the most efficient data-binding option, but proxies allow you to alter the behavior of many existing objects without changing their code.

Further Examples

Hemanth.HM's article [Negative Array Index in JavaScript](#) suggests using proxies to implement negative array indexes. For example, `arr[-1]` returns the last element, `arr[-2]` returns the second-to-last element, and so on.

Nicholas C. Zakas' article [Creating type-safe properties with ECMAScript 6 proxies](#) illustrates how proxies can be used to implement type safety by validating new values. In the example above, we could verify `myUserProxy.name` was always set to a string and throw an error otherwise.

Proxy Support

The power of proxies may not be immediately obvious, but they offer powerful meta-programming opportunities. Brendan Eich, the creator of JavaScript, thinks [Proxies are Awesome!](#)

Currently, proxy support is implemented in Node and all current browsers, with the exception of Internet Explorer 11. However, please note that not all browsers support all traps. You can get a better idea of what's supported by consulting this [browser compatibility table](#) on the MDN Proxy page.

Unfortunately, it's not possible to polyfill or transpile ES6 proxy code using tools such as [Babel](#), because proxies are powerful and have no ES5 equivalent.

Chapter 9: Destructuring Assignment

by Craig Buckler

Destructuring assignment sounds complex. It reminds me of object-oriented terms such as *encapsulation* and *polymorphism*. I'm convinced they were chosen to make simple concepts appear more sophisticated!

In essence, ECMAScript 6 (ES2015) destructuring assignment allows you to extract individual items from arrays or objects and place them into variables using a shorthand syntax. Those coming from PHP may have encountered the [list\(\)](#) function, which extracts arrays into variables in one operation. ES6 takes it to another level.

Presume we have an array:

```
var myArray = ['a', 'b', 'c'];
```

We can extract these values by index in ES5:

```
var
  one  = myArray[0],
  two  = myArray[1],
  three = myArray[2];

// one = 'a', two = 'b', three = 'c'
```

ES6 destructuring permits a simpler and less error-prone alternative:

```
const [one, two, three] = myArray;

// one = 'a', two = 'b', three = 'c'
```

You can ignore certain values, e.g.

```
const [one, , three] = myArray;

// one = 'a', three = 'c'
```

or use the rest operator (...) to extract remaining elements:

```
const [one, ...two] = myArray;  
// one = 'a', two = ['b', 'c']
```

Destructuring also works on objects, e.g.

```
var myObject = {  
  one: 'a',  
  two: 'b',  
  three: 'c'  
};  
  
// ES5 example  
var  
  one = myObject.one,  
  two = myObject.two,  
  three = myObject.three;  
  
// one = 'a', two = 'b', three = 'c'  
  
// ES6 destructuring example  
const {one, two, three} = myObject;  
  
// one = 'a', two = 'b', three = 'c'
```

In this example, the variable names `one`, `two` and `three` matched the object property names. We can also assign properties to variables with any name, e.g.

```
const myObject = {  
  one: 'a',  
  two: 'b',  
  three: 'c'  
};  
  
// ES6 destructuring example  
const {one: first, two: second, three: third} = myObject;  
  
// first = 'a', second = 'b', third = 'c'
```

More complex nested objects can also be referenced, e.g.

```
const meta = {  
  title: 'Destructuring Assignment',  
  authors: [  
    {
```

```

        firstname: 'Craig',
        lastname: 'Buckler'
    }
],
publisher: {
    name: 'SitePoint',
    url: 'http://www.sitepoint.com/'
}
};

const {
    title: doc,
    authors: [{ firstname: name }],
    publisher: { url: web }
} = meta;

/*
doc    = 'Destructuring Assignment'
name   = 'Craig'
web    = 'http://www.sitepoint.com/'
*/

```

This appears a little complicated but remember that in all destructuring assignments:

- the left-hand side of the assignment is the **destructuring target** — the pattern which defines the variables being assigned
- the right-hand side of the assignment is the **destructuring source** — the array or object which holds the data being extracted.

There are a number of other caveats. First, you can't start a statement with a curly brace, because it looks like a code block, e.g.

```
// THIS FAILS
{ a, b, c } = myObject;
```

You must either declare the variables, e.g.

```
// THIS WORKS
const { a, b, c } = myObject;
```

or use parentheses if variables are already declared, e.g.

```
// THIS WORKS
({ a, b, c } = myObject);
```

You should also be wary of mixing declared and undeclared variables, e.g.

```
// THIS FAILS
let a;
let { a, b, c } = myObject;

// THIS WORKS
let a, b, c;
({ a, b, c } = myObject);
```

That's the basics of destructuring. So when would it be useful? I'm glad you asked ...

Easier Declaration

Variables can be declared without explicitly defining each value, e.g.

```
// ES5
var a = 'one', b = 'two', c = 'three';

// ES6
const [a, b, c] = ['one', 'two', 'three'];
```

Admittedly, the destructured version is longer. It's a little easier to read, although that may not be the case with more items.

Variable Value Swapping

Swapping values in ES5 requires a temporary third variable, but it's far simpler with destructuring:

```
var a = 1, b = 2;

// ES5 swap
var temp = a;
a = b;
b = temp;

// a = 2, b = 1

// ES6 swap back
[a, b] = [b, a];

// a = 1, b = 2
```

You're not limited to two variables; any number of items can be rearranged, e.g.

```
// rotate left  
[b, c, d, e, a] = [a, b, c, d, e];
```

Default Function Parameters

Presume we had a function to output our `meta` object:

```
var meta = {  
    title: 'Destructuring Assignment',  
    authors: [  
        {  
            firstname: 'Craig',  
            lastname: 'Buckler'  
        }  
    ],  
    publisher: {  
        name: 'SitePoint',  
        url: 'http://www.sitepoint.com/'  
    }  
};  
  
prettyPrint(meta);
```

In ES5, it's necessary to parse this object to ensure appropriate defaults are available, e.g.

```
// ES5 default values  
function prettyPrint(param) {  
    param = param || {};  
    var  
        pubTitle = param.title || 'No title',  
        pubName = (param.publisher && param.publisher.name) || '';  
  
    return pubTitle + ', ' + pubName;  
}
```

In ES6 we can assign a default value to any parameter, e.g.

```
// ES6 default value  
function prettyPrint(param = {}) {
```

but we can then use destructuring to extract values and assign defaults where necessary:

```
// ES6 destructured default value
function prettyPrint(
  {
    title: pubTitle = 'No title',
    publisher: { name: pubName = 'No publisher' }
  } = {}
) {
  return pubTitle + ', ' + pubName;
}
```

I'm not convinced this is easier to read, but it's significantly shorter.

Returning Multiple Values from a Function

Functions can only return one value, but that can be a complex object or multi-dimensional array. Destructuring assignment makes this more practical, e.g.

```
function f() {
  return [1, 2, 3];
}

const [a, b, c] = f();

// a = 1, b = 2, c = 3
```

For-of Iteration

Consider an array of book information:

```
const books = [
  {
    title: 'Full Stack JavaScript',
    author: 'Colin Ihrig and Adam Bretz',
    url: 'http://www.sitepoint.com/store/full-stack-javascript-devel
  },
  {
    title: 'JavaScript: Novice to Ninja',
    author: 'Darren Jones',
    url: 'http://www.sitepoint.com/store/learn-javascript-novice-to
  },
  {
    title: 'Jump Start CSS',
    author: 'Louis Lazaris',
    url: 'http://www.sitepoint.com/store/jump-start-css/'
},
```

```
];
```

The ES6 [for-of](#) is similar to `for-in`, except that it extracts each value rather than the index/key, e.g.

```
for (const b of books) {  
  console.log(b.title + ' by ' + b.author + ': ' + b.url);  
}
```

Destructuring assignment provides further enhancements, e.g.

```
for (const {title, author, url} of books) {  
  console.log(title + ' by ' + author + ': ' + url);  
}
```

Regular Expression Handling

Regular expressions functions such as [match](#) return an array of matched items, which can form the source of a destructuring assignment:

```
const [a, b, c, d] = 'one two three'.match(/\w+/g);  
  
// a = 'one', b = 'two', c = 'three', d = undefined
```

Destructuring Assignment Support

Destructuring assignment may not revolutionize your development life, but it could save some considerable typing effort!

Currently, [support for destructuring assignment](#) is good. It's available in Node and all major browsers, with the exception of Internet Explorer. If you need to support older browsers, it's advisable to use a compiler such as [Babel](#) or [Traceur](#), which will translate ES6 destructuring assignments to an ES5 equivalent.

Chapter 10: ES6 Generators and Iterators: a Developer's Guide

by **Byron Houwens**

ES6 brought a number of new features to the JavaScript language. Two of these features, generators and iterators, have substantially changed how we write specific functions in more complex front-end code.

While they do play nicely with each other, what they actually do can be a little confusing, so let's check them out.

Iterators

Iteration is a common practice in programming and is usually used to loop over a set of values, either transforming each value, or using or saving it in some way for later.

In JavaScript, we've always had `for` loops that look like this:

```
for (var i = 0; i < foo.length; i++){
  // do something with i
}
```

But ES6 gives us an alternative:

```
for (const i of foo) {
  // do something with i
}
```

This is arguably way cleaner and easier to work with, and reminds me of languages like Python and Ruby. But there's something else that's pretty important to note about this new kind of iteration: it allows you to interact with elements of a data set directly.

Imagine that we want to find out if each number in an array is prime or not. We could do this by coming up with a function that does exactly that. It might look

like this:

```
function isPrime(number) {  
    if (number < 2) {  
        return false;  
    } else if (number === 2) {  
        return true;  
    }  
  
    for (var i = 2; i < number; i++) {  
        if (number % i === 0) {  
            return false;  
            break;  
        }  
    }  
  
    return true;  
}
```

Not the best in the world, but it works. The next step would be to loop over our list of numbers and check whether each one is prime with our shiny new function. It's pretty straightforward:

```
var possiblePrimes = [73, 6, 90, 19, 15];  
var confirmedPrimes = [];  
  
for (var i = 0; i < possiblePrimes.length; i++) {  
    if (isPrime(possiblePrimes[i])) {  
        confirmedPrimes.push(possiblePrimes[i]);  
    }  
}  
  
// confirmedPrimes is now [73, 19]
```

Again, it works, but it's clunky and that clunkiness is largely down to the way JavaScript handles `for` loops. With ES6, though, we're given an almost Pythonic option in the new iterator. So the previous `for` loop could be written like this:

```
const possiblePrimes = [73, 6, 90, 19, 15];  
const confirmedPrimes = [];  
  
for (const i of possiblePrimes){  
    if ( isPrime(i) ){  
        confirmedPrimes.push(i);  
    }  
}
```

```
// confirmedPrimes is now [73, 19]
```

This is far cleaner, but the most striking bit of this is the `for` loop. The variable `i` now represents the actual item in the array called `possiblePrimes`. So, we don't have to call it by index anymore. This means that instead of calling `possiblePrimes[i]` in the loop, we can just call `i`.

Behind the scenes, this kind of iteration is making use of ES6's bright and shiny [Symbol.iterator\(\)](#) method. This bad boy is in charge of describing the iteration and, when called, returns a JavaScript object containing the next value in the loop and a `done` key that is either `true` or `false` depending on whether or not the loop is finished.

In case you're interested in this sort of detail, you can read more about it on this fantastic blog post titled [Iterators gonna iterate](#) by Jake Archibald. It'll also give you a good idea of what's going on under the hood when we dive into the other side of this chapter: generators.

Generators

Generators, also called "iterator factories", are a new type of JavaScript function that creates specific iterations. They give you special, self-defined ways to loop over stuff.

Okay, so what does all that mean? Let's look at an example. Let's say that we want a function that will give us the next prime number every time we call it. Again, we'll use our `isPrime` function from before to check if a number is prime:

```
function* getNextPrime() {
  let nextNumber = 2;

  while (true) {
    if (isPrime(nextNumber)) {
      yield nextNumber;
    }
    nextNumber++;
  }
}
```

If you're used to JavaScript, some of this stuff will look a bit like voodoo, but

it's actually not too bad. We have that strange asterisk after the keyword `function`, but all this does is to tell JavaScript that we're defining a generator.

The other funky bit would be the `yield` keyword. This is actually what a generator spits out when you call it. It's roughly equivalent to `return`, but it keeps the state of the function instead of rerunning everything whenever you call it. It "remembers" its place while running, so the next time you call it, it carries on where it left off.

This means that we can do this:

```
const nextPrime = getNextPrime();
```

And then call `nextPrime` whenever we want to obtain — you guessed it — the next prime:

```
console.log(nextPrime.next().value); // 2
console.log(nextPrime.next().value); // 3
console.log(nextPrime.next().value); // 5
console.log(nextPrime.next().value); // 7
```

You can also just call `nextPrime.next()`, which is useful in situations where your generator isn't infinite, because it returns an object like this:

```
console.log(nextPrime.next());
// {value: 2, done: false}
```

Here, that `done` key tells you whether or not the function has completed its task. In our case, our function will never finish, and could theoretically give us all prime numbers up to infinity (if we had that much computer memory, of course).

Cool, so Can I Use Generators and Iterators Now?

Although ECMAScript 2015 has been finalized and has been in the wild for some years, browser support for its features — particularly generators — is far from complete. If you really want to use these and other modern features, you can check out transpilers like [Babel](#) and [Traceur](#), which will convert your ECMAScript 2015 code into its equivalent (where possible) ECMAScript 5 code.

There are also many online editors with support for ECMAScript 2015, or that specifically focus on it, particularly Facebook's [Regenerator](#) and [JS Bin](#). If you're just looking to play around and get a feel for how JavaScript can now be written , those are worth a look.

Conclusions

IGenerators and iterators give us quite a lot of new flexibility in our approach to JavaScript problems. Iterators allow us a more Pythonic way of writing for loops, which means our code will look cleaner and be easier to read.

Generator functions give us the ability to write functions that remember where they were when you last saw them, and can pick up where they left off. They can also be infinite in terms of how much they actually remember, which can come in really handy in certain situations.

Support for these generators and iterators is good. They're supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, your best bet is to use a transpiler such as [Babel](#).

Chapter 11: Object-oriented JavaScript: A Deep Dive into ES6 Classes

by Jeff Mott

Often we need to represent an idea or concept in our programs — maybe a car engine, a computer file, a router, or a temperature reading. Representing these concepts directly in code comes in two parts: data to represent the state, and functions to represent the behavior. ES6 classes give us a convenient syntax for defining the state and behavior of objects that will represent our concepts.

ES6 classes make our code safer by guaranteeing that an initialization function will be called, and they make it easier to define a fixed set of functions that operate on that data and maintain valid state. If you can think of something as a separate entity, it's likely you should define a class to represent that "thing" in your program.

Consider this non-class code. How many errors can you find? How would you fix them?

```
// set today to December 24

const today = {

  month: 24,

  day: 12,

};
```

```
const tomorrow = {  
  
    year: today.year,  
  
    month: today.month,  
  
    day: today.day + 1,  
  
};  
  
const dayAfterTomorrow = {  
  
    year: tomorrow.year,  
  
    month: tomorrow.month,  
  
    day: tomorrow.day + 1 <= 31 ? tomorrow.day + 1 : 1,  
  
};
```

The date today isn't valid: there's no month 24. Also, today isn't fully initialized: it's missing the year. It would be better if we had an initialization

function that couldn't be forgotten. Notice also that, when adding a day, we checked in one place if we went beyond 31 but missed that check in another place. It would be better if we interacted with the data only through a small and fixed set of functions that each maintain valid state.

Here's the corrected version that uses classes.

```
class SimpleDate {  
  
    constructor(year, month, day) {  
  
        // Check that (year, month, day) is a valid date  
  
        // ...  
  
        // If it is, use it to initialize "this" date  
  
        this._year = year;  
  
        this._month = month;  
  
        this._day = day;  
  
    }  
}
```

```
addDays(nDays) {  
  
    // Increase "this" date by n days  
  
    // ...  
  
}  
  
getDay() {  
  
    return this._day;  
  
}  
  
}  
  
// "today" is guaranteed to be valid and fully initialized  
  
const today = new SimpleDate(2000, 2, 28);
```

```
// Manipulating data only through a fixed set of functions ensures we can't accidentally break it.  
  
today.addDays(1);
```

Jargon

- When a function is associated with a class or object, we call it a **method**.
- When an object is created from a class, that object is said to be an **instance** of the class.

Constructors

The constructor method is special, and it solves the first problem. Its job is to initialize an instance to a valid state, and it will be called automatically so we can't forget to initialize our objects.

Keep Data Private

We try to design our classes so that their state is guaranteed to be valid. We provide a constructor that creates only valid values, and we design methods that also always leave behind only valid values. But as long as we leave the data of our classes accessible to everyone, someone *will* mess it up. We protect against this by keeping the data inaccessible except through the functions we supply.

Jargon

Keeping data private to protect it is called **encapsulation**.

Privacy with Conventions

Unfortunately, private object properties don't exist in JavaScript. We have to fake them. The most common way to do that is to adhere to a simple convention: if a property name is prefixed with an underscore (or, less commonly, suffixed with an underscore), then it should be treated as non-public. We used this approach in the earlier code example. Generally this simple convention works, but the data is technically still accessible to everyone, so we have to rely on our own discipline to do the right thing.

Privacy with Privileged Methods

The next most common way to fake private object properties is to use ordinary variables in the constructor, and capture them in closures. This trick gives us truly private data that's inaccessible to the outside. But to make it work, our class's methods would themselves need to be defined in the constructor and attached to the instance:

```
class SimpleDate {  
  
    constructor(year, month, day) {  
  
        // Check that (year, month, day) is a valid date  
  
        // ...  
    }  
}
```

```
// If it is, use it to initialize "this" date's ordinary variables

let _year = year;

let _month = month;

let _day = day;

// Methods defined in the constructor capture variables in a closure

this.addDays = function(nDays) {

    // Increase "this" date by n days

    // ...

}

this.getDay = function() {

    return _day;
```

```
    }

}

}
```

Privacy with Symbols

Symbols are a new feature of JavaScript as of ES6, and they give us another way to fake private object properties. Instead of underscore property names, we could use unique symbol object keys, and our class can capture those keys in a closure. But there's a leak. Another new feature of JavaScript is `Object.getOwnPropertySymbols`, and it allows the outside to access the symbol keys we tried to keep private:

```
const SimpleDate = (function() {

    const _yearKey = Symbol();

    const _monthKey = Symbol();

    const _dayKey = Symbol();

    class SimpleDate {

        constructor(year, month, day) {
```

```
// Check that (year, month, day) is a valid date

// ...

// If it is, use it to initialize "this" date

this[_yearKey] = year;

this[_monthKey] = month;

this[_dayKey] = day;

}

addDays(nDays) {

    // Increase "this" date by n days

    // ...

}
```

```
getDay() {  
  
    return this[_dayKey];  
  
}  
  
}  
  
return SimpleDate;  
}());
```

Privacy with Weak Maps

[Weak maps](#) are also a new feature of JavaScript. We can store private object properties in key/value pairs using our instance as the key, and our class can capture those key/value maps in a closure:

```
const SimpleDate = (function() {  
  
    const _years = new WeakMap();  
  
    const _months = new WeakMap();  
})()
```

```
const _days = new WeakMap();

class SimpleDate {

    constructor(year, month, day) {

        // Check that (year, month, day) is a valid date

        // ...

        // If it is, use it to initialize "this" date

        _years.set(this, year);

        _months.set(this, month);

        _days.set(this, day);

    }
}
```

```
addDays(nDays) {  
  
    // Increase "this" date by n days  
  
    // ...  
  
}  
  
getDay() {  
  
    return _days.get(this);  
  
}  
  
}  
  
return SimpleDate;  
  
}());
```

Other Access Modifiers

There are other levels of visibility besides “private” that you’ll find in other

languages, such as “protected”, “internal”, “package private”, or “friend”. JavaScript still doesn’t give us a way to enforce those other levels of visibility. If you need them, you’ll have to rely on conventions and self discipline.

Referring to the Current Object

Look again at `getDay()`. It doesn’t specify any parameters, so how does it know the object for which it was called? When a function is called as a method using the `object.function` notation, there’s an implicit argument that it uses to identify the object, and that implicit argument is assigned to an implicit parameter named `this`. To illustrate, here’s how we would send the object argument explicitly rather than implicitly:

```
// Get a reference to the "getDay" function

const getDay = SimpleDate.prototype.getDay;

getDay.call(today); // "this" will be "today"

getDay.call(tomorrow); // "this" will be "tomorrow"

tomorrow.getDay(); // same as last line, but "tomorrow" is passed im
```

Static Properties and Methods

We have the option to define data and functions that are part of the class but not part of any instance of that class. We call these static properties and static methods, respectively. There will only be one copy of a static property rather than a new copy per instance:

```
class SimpleDate {  
  
    static setDateDefault(year, month, day) {  
  
        // A static property can be referred to without mentioning an instance  
  
        // Instead, it's defined on the class  
  
        SimpleDate._defaultDate = new SimpleDate(year, month, day);  
  
    }  
  
    constructor(year, month, day) {  
  
        // If constructing without arguments,  
  
        // then initialize "this" date by copying the static default date  
  
        if (arguments.length === 0) {  
  
            this._year = SimpleDate._defaultDate._year;  
        }  
    }  
}
```

```
this._month = SimpleDate._defaultDate._month;

this._day = SimpleDate._defaultDate._day;

return;

}

// Check that (year, month, day) is a valid date

// ...

// If it is, use it to initialize "this" date

this._year = year;

this._month = month;

this._day = day;
```

```
}

addDays(nDays) {

    // Increase "this" date by n days

    // ...

}

getDay() {

    return this._day;

}

}

SimpleDate.setDefaultDate(1970, 1, 1);
```

```
const defaultDate = new SimpleDate();
```

Subclasses

Often we find commonality between our classes — repeated code that we'd like to consolidate. Subclasses let us incorporate another class's state and behavior into our own. This process is often called **inheritance**, and our subclass is said to "inherit" from a parent class, also called a **superclass**. Inheritance can avoid duplication and simplify the implementation of a class that needs the same data and functions as another class. Inheritance also allows us to substitute subclasses, relying only on the [interface](#) provided by a common superclass.

Inherit to Avoid Duplication

Consider this non-inheritance code:

```
class Employee {  
  
    constructor(firstName, familyName) {  
  
        this._firstName = firstName;  
  
        this._familyName = familyName;  
  
    }  
}
```

```
getFullName() {  
  
    return `${this._firstName} ${this._familyName}`;  
  
}  
  
}  
  
  
  
  
class Manager {  
  
    constructor(firstName, familyName) {  
  
        this._firstName = firstName;  
  
        this._familyName = familyName;  
  
        this._managedEmployees = [];  
  
    }  
  
  
  
  
    getFullName() {
```

```
    return `${this._firstName} ${this._familyName}`;

}

addEmployee(employee) {

    this._managedEmployees.push(employee);

}

}
```

The data properties `_firstName` and `_familyName`, and the method `getFullName`, are repeated between our classes. We could eliminate that repetition by having our `Manager` class inherit from the `Employee` class. When we do, the state and behavior of the `Employee` class — its data and functions — will be incorporated into our `Manager` class.

Here's a version that uses inheritance. Notice the use of [super](#):

```
// Manager still works same as before but without repeated code

class Manager extends Employee {

    constructor(firstName, familyName) {

        super(firstName, familyName);
```

```
    this._managedEmployees = [];

}

addEmployee(employee) {

    this._managedEmployees.push(employee);

}

}
```

IS-A and WORKS-LIKE-A

There are design principles to help you decide when inheritance is appropriate. Inheritance should always model an IS-A and WORKS-LIKE-A relationship. That is, a manager “is a” and “works like a” specific kind of employee, such that anywhere we operate on a superclass instance, we should be able to substitute in a subclass instance, and everything should still just work. The difference between violating and adhering to this principle can sometimes be subtle. A classic example of a subtle violation is a Rectangle superclass and a Square subclass:

```
class Rectangle {

    set width(w) {
```

```
    this._width = w;  
}  
  
get width() {  
  
    return this._width;  
}  
  
set height(h) {  
  
    this._height = h;  
}  
  
get height() {  
  
    return this._height;
```

```
}

}

// A function that operates on an instance of Rectangle

function f(rectangle) {

    rectangle.width = 5;

    rectangle.height = 4;

    // Verify expected result

    if (rectangle.width * rectangle.height !== 20) {

        throw new Error("Expected the rectangle's area (width * height)

    }

}
```

```
// A square IS-A rectangle... right?
```

```
class Square extends Rectangle {
```

```
    set width(w) {
```

```
        super.width = w;
```

```
        // Maintain square-ness
```

```
        super.height = w;
```

```
}
```

```
    set height(h) {
```

```
        super.height = h;
```

```
// Maintain square-ness

super.width = h;

}

}

// But can a rectangle be substituted by a square?

f(new Square()); // error
```

A square may be a rectangle *mathematically*, but a square doesn't *work like a rectangle* behaviorally.

This rule that any use of a superclass instance should be substitutable by a subclass instance is called the [Liskov Substitution Principle](#), and it's an important part of object-oriented class design.

Beware Overuse

It's easy to find commonality everywhere, and the prospect of having a class that offers complete functionality can be alluring, even for experienced developers. But there are disadvantages to inheritance too. Recall that we ensure valid state by manipulating data only through a small and fixed set of functions. But when we inherit, we increase the list of functions that can directly manipulate the data, and those additional functions are then also responsible for maintaining valid state. If too many functions can directly manipulate the data, that data becomes

nearly as bad as global variables. Too much inheritance creates monolithic classes that dilute encapsulation, are harder to make correct, and harder to reuse. Instead, prefer to design minimal classes that embody just one concept.

Let's revisit the code duplication problem. Could we solve it without inheritance? An alternative approach is to connect objects through references to represent a part–whole relationship. We call this **composition**.

Here's a version of the manager–employee relationship using composition rather than inheritance:

```
class Employee {  
  
    constructor(firstName, familyName) {  
  
        this._firstName = firstName;  
  
        this._familyName = familyName;  
  
    }  
  
    getFullName() {  
  
        return `${this._firstName} ${this._familyName}`;  
  
    }  
  
}
```

```
class Group {  
  
    constructor(manager /* : Employee */ ) {  
  
        this._manager = manager;  
  
        this._managedEmployees = [];  
  
    }  
  
    addEmployee(employee) {  
  
        this._managedEmployees.push(employee);  
  
    }  
  
}
```

Here, a manager isn't a separate class. Instead, a manager is an ordinary Employee instance that a Group instance holds a reference to. If inheritance models the IS-A relationship, then composition models the HAS-A relationship. That is, a group "has a" manager.

If either inheritance or composition can reasonably express our program concepts and relationships, then prefer composition.

Inherit to Substitute Subclasses

Inheritance also allows different subclasses to be used interchangeably through the interface provided by a common superclass. A function that expects a superclass instance as an argument can also be passed a subclass instance without the function having to know about any of the subclasses. Substituting classes that have a common superclass is often called **polymorphism**:

```
// This will be our common superclass

class Cache {

    get(key, defaultValue) {

        const value = this._doGet(key);

        if (value === undefined || value === null) {

            return defaultValue;

        }

        return value;
    }
}
```

```
}

set(key, value) {

    if (key === undefined || key === null) {

        throw new Error('Invalid argument');

    }

    this._doSet(key, value);

}

// Must be overridden

// _doGet()

// _doSet()
```

```
}

// Subclasses define no new public methods

// The public interface is defined entirely in the superclass

class ArrayCache extends Cache {

    _doGet() {

        // ...

    }

    _doSet() {

        // ...

    }

}
```

```
class LocalStorageCache extends Cache {  
  
    _doGet() {  
  
        // ...  
  
    }  
  
    _doSet() {  
  
        // ...  
  
    }  
  
}  
  
// Functions can polymorphically operate on any cache by interacting  
  
function compute(cache) {
```

```
const cached = cache.get('result');

if (!cached) {

    const result = // ...

    cache.set('result', result);

}

// ...

}

compute(new ArrayCache()); // use array cache through superclass int

compute(new LocalStorageCache()); // use local storage cache through
```

More than Sugar

JavaScript's class syntax is often said to be syntactic sugar, and in a lot of ways it is, but there are also real differences — things we can do with ES6 classes that

we couldn't do in ES5.

Static Properties Are Inherited

ES5 didn't let us create true inheritance between constructor functions. `Object.create` could create an ordinary object but not a function object. We faked inheritance of static properties by manually copying them. Now with ES6 classes, we get a real prototype link between a subclass constructor function and the superclass constructor:

```
// ES5

function B() {}

B.f = function () {};

function D() {}

D.prototype = Object.create(B.prototype);

D.f(); // error
```

```
// ES6
```

```
class B {  
  
    static f() {}  
  
}  
  
  
  
class D extends B {}  
  
  
  
D.f(); // ok
```

Built-in Constructors Can Be Subclassed

Some objects are “exotic” and don’t behave like ordinary objects. Arrays, for example, adjust their `length` property to be greater than the largest integer index. In ES5, when we tried to subclass `Array`, the `new` operator would allocate an ordinary object for our subclass, not the exotic object of our superclass:

```
// ES5  
  
function D() {  
  
    Array.apply(this, arguments);  
  
}
```

```
D.prototype = Object.create(Array.prototype);

var d = new D();

d[0] = 42;

d.length; // 0 - bad, no array exotic behavior
```

ES6 classes fixed this by changing when and by whom objects are allocated. In ES5, objects were allocated before invoking the subclass constructor, and the subclass would pass that object to the superclass constructor. Now with ES6 classes, objects are allocated before invoking the *superclass* constructor, and the superclass makes that object available to the subclass constructor. This lets Array allocate an exotic object even when we invoke new on our subclass.

```
// ES6

class D extends Array {}

let d = new D();

d[0] = 42;
```

```
d.length; // 1 - good, array exotic behavior
```

Miscellaneous

There's a small assortment of other, probably less significant differences. Class constructors can't be function-called. This protects against forgetting to invoke constructors with `new`. Also, a class constructor's `prototype` property can't be reassigned. This may help JavaScript engines optimize class objects. And finally, class methods don't have a `prototype` property. This may save memory by eliminating unnecessary objects.

Using New Features in Imaginative Ways

Many of the features described here and in other SitePoint articles are new to JavaScript, and the community is experimenting right now to use those features in new and imaginative ways.

Multiple Inheritance with Proxies

One such experiment uses [proxies](#), a new feature to JavaScript for implementing multiple inheritance. JavaScript's prototype chain allows only single inheritance. Objects can delegate to only one other object. Proxies give us a way to delegate property accesses to multiple other objects:

```
const transmitter = {  
  // ...  
  transmit() {}  
}
```

```
};

const receiver = {

    receive() {}

};

// Create a proxy object that intercepts property accesses and forwa

// returning the first defined value it finds

const inheritsFromMultiple = new Proxy([transmitter, receiver], {

    get: function(proxyTarget, propertyKey) {

        const foundParent = proxyTarget.find(parent => parent[propertyKe

        return foundParent && foundParent[propertyKey];

    }

});
```

```
});

inheritsFromMultiple.transmit() // works

inheritsFromMultiple.receive() // works
```

Can we expand this to work with ES6 classes? A class's prototype could be a proxy that forwards property access to multiple other prototypes. The JavaScript community is working on this right now. Can you figure it out? Join the discussion and share your ideas.

Multiple Inheritance with Class Factories

Another approach the JavaScript community has been experimenting with is generating classes on demand that extend a variable superclass. Each class still has only a single parent, but we can chain those parents in interesting ways:

```
function makeTransmitterClass(Superclass = Object) {

    return class Transmitter extends Superclass {

        transmit() {}

    };
}
```

```
}

function makeReceiverClass(Superclass = Object) {

    return class Receiver extends Superclass

        receive() {}

    };

}

class InheritsFromMultiple extends makeTransmitterClass(makeReceiver

const inheritsFromMultiple = new InheritsFromMultiple();

inheritsFromMultiple.transmit(); // works
```

```
inheritsFromMultiple.receive(); // works
```

Are there other imaginative ways to use these features? Now's the time to leave your footprint in the JavaScript world.

Conclusion

Support for classes is pretty good. Hopefully this chapter has given you an insight into how classes work in ES6 and has demystified some of the jargon surrounding them.

Chapter 12: Understanding ES6 Modules

by Craig Buckler

This chapter explores ES6 modules, showing how they can be used today with the help of a transpiler.

Almost every language has a concept of *modules* — a way to include functionality declared in one file within another. Typically, a developer creates an encapsulated library of code responsible for handling related tasks. That library can be referenced by applications or other modules.

The benefits:

1. Code can be split into smaller files of self-contained functionality.
2. The same modules can be shared across any number of applications.
3. Ideally, modules need never be examined by another developer, because they've been proven to work.
4. Code referencing a module understands it's a dependency. If the module file is changed or moved, the problem is immediately obvious.
5. Module code (usually) helps eradicate naming conflicts. Function `x()` in `module1` cannot clash with function `x()` in `module2`. Options such as namespacing are employed so calls become `module1.x()` and `module2.x()`.

Where are Modules in JavaScript?

Anyone starting web development a few years ago would have been shocked to discover there was no concept of modules in JavaScript. It was impossible to directly reference or include one JavaScript file in another. Developers therefore resorted to alternative options.

Multiple HTML `<script>` Tags

HTML can load any number JavaScript files using multiple `<script>` tags:

```
<script src="lib1.js"></script>
<script src="lib2.js"></script>
<script src="core.js"></script>
<script>
console.log('inline code');
</script>
```

The [average web page in 2018 uses 25 separate scripts](#), yet it's not a practical solution:

- Each script initiates a new HTTP request, which affects page performance. [HTTP/2](#) alleviates the issue to some extent, but it doesn't help scripts referenced on other domains such as a CDN.
- Every script halts further processing while it's run.
- Dependency management is a manual process. In the code above, if `lib1.js` referenced code in `lib2.js`, the code would fail because it had not been loaded. That could break further JavaScript processing.
- Functions can override others unless appropriate [module patterns](#) are used. Early JavaScript libraries were notorious for using global function names or overriding native methods.

Script Concatenation

One solution to problems of multiple `<script>` tags is to concatenate all JavaScript files into a single, large file. This solves some performance and dependency management issues, but it could incur a manual build and testing step.

Module Loaders

Systems such as [RequireJS](#) and [SystemJS](#) provide a library for loading and namespacing other JavaScript libraries at runtime. Modules are loaded using Ajax methods when required. The systems help, but could become complicated for larger code bases or sites adding standard `<script>` tags into the mix.

Module Bundlers, Preprocessors and Transpilers

Bundlers introduce a compile step so JavaScript code is generated at build time. Code is processed to include dependencies and produce a single ES5 cross-browser compatible concatenated file. Popular options include [Babel](#),

[Browserify](#), [webpack](#) and more general task runners such as [Grunt](#) and [Gulp](#).

A JavaScript build process requires some effort, but there are benefits:

- Processing is automated so there's less chance of human error.
- Further processing can lint code, remove debugging commands, minify the resulting file, etc.
- Transpiling allows you to use alternative syntaxes such as [TypeScript](#) or [CoffeeScript](#).

ES6 Modules

The options above introduced a variety of competing module definition formats. Widely-adopted syntaxes included:

- CommonJS — the `module.exports` and `require` syntax used in Node.js
- Asynchronous Module Definition (AMD)
- Universal Module Definition (UMD).

A single, native module standard was therefore proposed in ES6 (ES2015).

Everything inside an ES6 module is private by default, and runs in strict mode (there's no need for `'use strict'`). Public variables, functions and classes are exposed using `export`. For example:

```
// lib.js
export const PI = 3.1415926;

export function sum(...args) {
  log('sum', args);
  return args.reduce((num, tot) => tot + num);
}

export function mult(...args) {
  log('mult', args);
  return args.reduce((num, tot) => tot * num);
}

// private function
function log(...msg) {
  console.log(...msg);
}
```

Alternatively, a single `export` statement can be used. For example:

```
// lib.js
const PI = 3.1415926;

function sum(...args) {
  log('sum', args);
  return args.reduce((num, tot) => tot + num);
}

function mult(...args) {
  log('mult', args);
  return args.reduce((num, tot) => tot * num);
}

// private function
function log(...msg) {
  console.log(...msg);
}

export { PI, sum, mult };
```

`import` is then used to pull items from a module into another script or module:

```
// main.js
import { sum } from './lib.js';

console.log( sum(1,2,3,4) ); // 10
```

In this case, `lib.js` is in the same folder as `main.js`. Absolute file references (starting with `/`), relative file references (starting `./` or `../`) or full URLs can be used.

Multiple items can be imported at one time:

```
import { sum, mult } from './lib.js';

console.log( sum(1,2,3,4) ); // 10
console.log( mult(1,2,3,4) ); // 24
```

and imports can be aliased to resolve naming collisions:

```
import { sum as addAll, mult as multiplyAll } from './lib.js';

console.log( addAll(1,2,3,4) ); // 10
console.log( multiplyAll(1,2,3,4) ); // 24
```

Finally, all public items can be imported by providing a namespace:

```
import * as lib from './lib.js';

console.log( lib.PI );           // 3.1415926
console.log( lib.add(1,2,3,4) ); // 10
console.log( lib.mult(1,2,3,4) ); // 24
```

Using ES6 Modules in Browsers

At the time of writing, [ES6 modules are supported](#) in Chromium-based browsers (v63+), Safari 11+, and Edge 16+. Firefox support will arrive in version 60 (it's behind an about:config flag in v58+).

Scripts which use modules must be loaded by setting a type="module" attribute in the <script> tag. For example:

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

or inline:

```
<script type="module">
  import { something } from './somewhere.js';
  ...
</script>
```

Modules are parsed once, regardless of how many times they're referenced in the page or other modules.

Server Considerations

Modules must be served with the MIME type application/javascript. Most servers will do this automatically, but be wary of dynamically generated scripts or .mjs files ([see the Node.js section below](#)).

Regular <script> tags can fetch scripts on other domains but modules are fetched using cross-origin resource sharing (CORS). Modules on different domains must therefore set an appropriate HTTP header, such as Access-Control-Allow-Origin: *.

Finally, modules won't send cookies or other header credentials unless a

`crossorigin="use-credentials"` attribute is added to the `<script>` tag and the response contains the header `Access-Control-Allow-Credentials: true`.

Module Execution is Deferred

The `<script defer>` attribute delays script execution until the document has loaded and parsed. Modules — *including inline scripts* — defer by default. Example:

```
<!-- runs SECOND -->
<script type="module">
  // do something...
</script>

<!-- runs THIRD -->
<script defer src="c.js"></script>

<!-- runs FIRST -->
<script src="a.js"></script>

<!-- runs FOURTH -->
<script type="module" src="b.js"></script>
```

Module Fallbacks

Browsers without module support won't run `type="module"` scripts. A fallback script can be provided with a `nomodule` attribute which module-compatible browsers ignore. For example:

```
<script type="module" src="runs-if-module-supported.js"></script>
<script nomodule src="runs-if-module-not-supported.js"></script>
```

Should You Use Modules in the Browser?

Browser support is growing, but it's possibly a little premature to switch to ES6 modules. For the moment, it's probably better to use a module bundler to create a script that works everywhere.

Using ES6 Modules in Node.js

When Node.js was released in 2009, it would have been inconceivable for any

runtime not to provide modules. CommonJS was adopted, which meant the Node package manager, npm, could be developed. Usage grew exponentially from that point.

A CommonJS module can be coded in a similar way to an ES2015 module. `module.exports` is used rather than `export`:

```
// lib.js
const PI = 3.1415926;

function sum(...args) {
  log('sum', args);
  return args.reduce((num, tot) => tot + num);
}

function mult(...args) {
  log('mult', args);
  return args.reduce((num, tot) => tot * num);
}

// private function
function log(...msg) {
  console.log(...msg);
}

module.exports = { PI, sum, mult };
```

`require` (rather than `import`) is used to pull this module into another script or module:

```
const { sum, mult } = require('./lib.js');

console.log( sum(1,2,3,4) ); // 10
console.log( mult(1,2,3,4) ); // 24
```

`require` can also import all items:

```
const lib = require('./lib.js');

console.log( lib.PI ); // 3.1415926
console.log( lib.add(1,2,3,4) ); // 10
console.log( lib.mult(1,2,3,4) ); // 24
```

So ES6 modules were easy to implement in Node.js, right? *Er, no.*

ES6 modules are [behind a flag in Node.js 9.8.0+](#) and will not be fully

implemented until at least version 10. While CommonJS and ES6 modules share similar syntax, they work in fundamentally different ways:

- ES6 modules are pre-parsed in order to resolve further imports before code is executed.
- CommonJS modules load dependencies on demand while executing the code.

It would make no difference in the example above, but consider the following ES2015 module code:

```
// ES2015 modules

// -----
// one.js
console.log('running one.js');
import { hello } from './two.js';
console.log(hello);

// -----
// two.js
console.log('running two.js');
export const hello = 'Hello from two.js';
```

The output for ES2015:

```
running two.js
running one.js
hello from two.js
```

Similar code written using CommonJS:

```
// CommonJS modules

// -----
// one.js
console.log('running one.js');
const hello = require('./two.js');
console.log(hello);

// -----
// two.js
console.log('running two.js');
module.exports = 'Hello from two.js';
```

The output for CommonJS:

```
running one.js
running two.js
hello from two.js
```

Execution order could be critical in some applications, and what would happen if ES2015 and CommonJS modules were mixed in the same file? To resolve this problem, Node.js will only permit ES6 modules in files with the extension `.mjs`. Files with a `.js` extension will default to CommonJS. It's a simple option which removes much of the complexity and should aid code editors and linters.

Should You Use ES6 Modules in Node.js?

ES6 modules are only practical from Node.js v10 onwards (released in April 2018). Converting an existing project is unlikely to result in any benefit, and would render an application incompatible with earlier versions of Node.js.

For new projects, ES6 modules provide an alternative to CommonJS. The syntax is identical to client-side coding, and may offer an easier route to isomorphic JavaScript, which can run in either the browser or on a server.

Module Melee

A standardized JavaScript module system took many years to arrive, and even longer to implement, but the problems have been rectified. All mainstream browsers and Node.js from mid 2018 support ES6 modules, although a switch-over lag should be expected while everyone upgrades.

Learn ES6 modules today to benefit your JavaScript development tomorrow.

Chapter 13: An Overview of JavaScript Promises

by Sandeep Panda

This tutorial covers the basics of JavaScript promises, showing how you can leverage them in your JavaScript development.

The concept of promises is not new to web development. Many of us have already used promises in the form of libraries such as Q, when.js, RSVP.js, etc. Even jQuery has something called a [Deferred object](#), which is similar to a promise. But now we have native support for promises in JavaScript, which is really exciting.

Overview

A Promise object represents a value that may not be available yet, but will be resolved at some point in the future. It allows you to write asynchronous code in a more synchronous fashion. For example, if you use the promise API to make an asynchronous call to a remote web service, you will create a Promise object which represents the data that will be returned by the web service in future. The caveat is that the actual data isn't available yet. It will become available when the request completes and a response comes back from the web service. In the meantime, the Promise object acts like a proxy to the actual data. Furthermore, you can attach callbacks to the Promise object, which will be called once the actual data is available.

The API

To get started, let's examine the following code, which creates a new Promise object:

```
const promise = new Promise((resolve, reject) => {
  //asynchronous code goes here
});
```

We start by instantiating a new `Promise` object and passing it a callback function. The callback takes two arguments, `resolve` and `reject`, which are both functions. All your asynchronous code goes inside that callback. If everything is successful, the promise is fulfilled by calling `resolve()`. In case of an error, `reject()` is called with an `Error` object. This indicates that the promise is rejected.

Now let's build something simple which shows how promises are used. The following code makes an asynchronous request to a web service that returns a random joke in JSON format. Let's examine how promises are used here:

```
const promise = new Promise((resolve, reject) => {
  const request = new XMLHttpRequest();

  request.open('GET', 'https://api.icndb.com/jokes/random');
  request.onload = () => {
    if (request.status === 200) {
      resolve(request.response); // we got data here, so resolve the
    } else {
      reject(Error(request.statusText)); // status is not 200 OK, so
    }
  };

  request.onerror = () => {
    reject(Error('Error fetching data.')); // error occurred, reject
  };

  request.send(); // send the request
});

console.log('Asynchronous request made.');

promise.then((data) => {
  console.log('Got data! Promise fulfilled.');
  document.body.textContent = JSON.parse(data).value.joke;
}, (error) => {
  console.log('Promise rejected.');
  console.log(error.message);
});
```

In the previous code, the `Promise` constructor callback contains the asynchronous code used to get data from remote service. Here, we just create an Ajax request to <https://api.icndb.com/jokes/random>, which returns a random joke. When a JSON response is received from the remote server, it's passed to `resolve()`. In case of any error, `reject()` is called with an `Error` object.

When we instantiate a `Promise` object, we get a proxy to the data that will be available in future. In our case, we're expecting some data to be returned from the remote service at some point in future. So, how do we know when the data becomes available? This is where the `Promise.then()` function is used. This function takes two arguments: a success callback and a failure callback. These callbacks are called when the `Promise` is settled (i.e. either fulfilled or rejected). If the promise was fulfilled, the success callback will be fired with the actual data you passed to `resolve()`. If the promise was rejected, the failure callback will be called. Whatever you passed to `reject()` will be passed as an argument to this callback.

Try this [CodePen](#) example. To view a new random joke, hit the *RERUN* button in the bottom right-hand corner of the embed. Also, open up your browser console so that you can see the order in which the different parts of the code are executed.

Note that a promise can have three states:

- pending (not fulfilled or rejected)
- fulfilled
- rejected

The `Promise.status` property, which is code-inaccessible and private, gives information about these states. Once a promise is rejected or fulfilled, this status gets permanently associated with it. This means a promise can succeed or fail only once. If the promise has already been fulfilled and later you attach a `then()` to it with two callbacks, the success callback will be correctly called. So, in the world of promises, we're not interested in knowing when the promise is settled. We're only concerned with the final outcome of the promise.

Chaining Promises

It's sometimes desirable to chain promises together. For instance, you might have multiple asynchronous operations to be performed. When one operation gives you data, you'll start doing some other operation on that piece of data and so on. Promises can be chained together, as demonstrated in the following example:

```
function getPromise(url) {
```

```

    // return a Promise here
    // send an async request to the url as a part of promise
    // after getting the result, resolve the promise with it
}

const promise = getPromise('some url here');

promise.then((result) => {
  //we have our result here
  return getPromise(result); //return a promise here again
}).then((result) => {
  //handle the final result
});

```

The tricky part is that, when you return a simple value inside `then()`, the next `then()` is called with that return value. But if you return a promise inside `then()`, the next `then()` waits on it and gets called when that promise is settled.

Handling Errors

You already know the `then()` function takes two callbacks as arguments. The second one will be called if the promise was rejected. But we also have a `catch()` function, which can be used to handle promise rejection. Have a look at the following code:

```

promise.then((result) => {
  console.log('Got data!', result);
}).catch((error) => {
  console.log('Error occurred!', error);
});

```

This is equivalent to:

```

promise.then((result) => {
  console.log('Got data!', result);
}).then(undefined, (error) => {
  console.log('Error occurred!', error);
});

```

Note that if the promise was rejected and `then()` doesn't have a failure callback, the control will move forward to the next `then()` with a failure callback or the next `catch()`. Apart from explicit promise rejection, `catch()` is also called when any exception is thrown from the `Promise` constructor callback. So you can also use `catch()` for logging purposes. Note that we could use `try...catch` to

handle errors, but that's not necessary with promises, as any asynchronous or synchronous error is always caught by `catch()`.

Conclusion

This was just a brief introduction to JavaScript's new Promises API. Clearly it lets us write asynchronous code very easily. We can proceed as usual without knowing what value is going to be returned from the asynchronous code in the future. There's more to the API that has not been covered here. To learn more about Promises, check out my follow-up article [A Deeper Dive Into JavaScript Promises](#), as well as these great resources:

- [HTML5Rocks](#)
- [Mozilla Developer Network](#)

Chapter 14: JavaScript Decorators: What They Are and When to Use Them

by Graham Cox

With the introduction of ES2015+, and as transpilation has become commonplace, many of you will have come across newer language features, either in real code or tutorials. One of these features that often has people scratching their heads when they first come across them are JavaScript decorators.

Decorators have become popular thanks to their use in Angular 2+. In Angular, decorators are available thanks to TypeScript, but in JavaScript they're currently a [stage 2 proposal](#), meaning they should be part of a future update to the language. Let's take a look at what decorators are, and how they can be used to make your code cleaner and more easily understandable.

What is a Decorator?

In its simplest form, a decorator is simply a way of wrapping one piece of code with another — literally “decorating” it. This is a concept you might well have heard of previously as **functional composition**, or **higher-order functions**.

This is already possible in standard JavaScript for many use cases, simply by calling on one function to wrap another:

```
function doSomething(name) {
  console.log('Hello, ' + name);
}

function loggingDecorator(wrapped) {
  return function() {
    console.log('Starting');
    const result = wrapped.apply(this, arguments);
    console.log('Finished');
```

```
        return result;
    }
}

const wrapped = loggingDecorator(doSomething);
```

This example produces a new function — in the variable `wrapped` — that can be called exactly the same way as the `doSomething` function, and will do exactly the same thing. The difference is that it will do some logging before and after the wrapped function is called:

```
doSomething('Graham');
// Hello, Graham

wrapped('Graham');
// Starting
// Hello, Graham
// Finished
```

How to Use JavaScript Decorators

Decorators use a special syntax in JavaScript, whereby they are prefixed with an `@` symbol and placed immediately before the code being decorated.

Note: at the time of writing, the decorators are currently in “[Stage 2 Draft](#)” form, meaning that they are mostly finished but still subject to changes.

It’s possible to use as many decorators on the same piece of code as you desire, and they’ll be applied in the order that you declare them.

For example:

```
@log()
@immutable()
class Example {
  @time('demo')
  doSomething() {
    //
  }
}
```

This defines a class and applies three decorators — two to the class itself, and one to a property of the class:

- `@log` could log all access to the class
- `@immutable` could make the class immutable — maybe it calls `Object.freeze` on new instances
- `@time` will record how long a method takes to execute and log this out with a unique tag.

At present, using decorators requires transpiler support, since no current browser or Node release has support for them yet. If you're using Babel, this is enabled simply by using the [transform-decorators-legacy plugin](#).

Note: the use of the word “legacy” in this plugin is because it supports the Babel 5 way of handling decorators, which might well be different from the final form when they’re standardized.

Why Use Decorators?

Whilst functional composition is already possible in JavaScript, it's significantly more difficult — or even impossible — to apply the same techniques to other pieces of code (e.g. classes and class properties).

The decorator proposal adds support for class and property decorators that can be used to resolve these issues, and future JavaScript versions will probably add decorator support for other troublesome areas of code.

Decorators also allow for a cleaner syntax for applying these wrappers around your code, resulting in something that detracts less from the actual intention of what you're writing.

Different Types of Decorator

At present, the only types of decorator that are supported are on classes and members of classes. This includes properties, methods, getters, and setters.

Decorators are actually nothing more than functions that return another function, and that are called with the appropriate details of the item being decorated. These decorator functions are evaluated once when the program first runs, and the decorated code is replaced with the return value.

Class member decorators

Property decorators are applied to a single member in a class — whether they are properties, methods, getters, or setters. This decorator function is called with three parameters:

- target: the class that the member is on.
- name: the name of the member in the class.
- descriptor: the member descriptor. This is essentially the object that would have been passed to [Object.defineProperty](#).

The classic example used here is `@readonly`. This is implemented as simply as:

```
function readonly(target, name, descriptor) {
  descriptor.writable = false;
  return descriptor;
}
```

Literally updating the property descriptor to set the “writable” flag to false.

This is then used on a class property as follows:

```
class Example {
  a() {}
  @readonly
  b() {}
}

const e = new Example();
e.a = 1;
e.b = 2;
// TypeError: Cannot assign to read only property 'b' of object '#<E'
```

But we can do better than this. We can actually replace the decorated function with different behavior. For example, let’s log all of the inputs and outputs:

```
function log(target, name, descriptor) {
  const original = descriptor.value;
  if (typeof original === 'function') {
    descriptor.value = function(...args) {
      console.log(`Arguments: ${args}`);
      try {
        const result = original.apply(this, args);
        console.log(`Result: ${result}`);
        return result;
      }
    }
  }
}
```

```

        } catch (e) {
            console.log(`Error: ${e}`);
            throw e;
        }
    }
    return descriptor;
}

```

This replaces the entire method with a new one that logs the arguments, calls the original method and then logs the output.

Note that we've used the [spread operator](#) here to automatically build an array from all of the arguments provided, which is the more modern alternative to the old `arguments` value.

We can see this in use as follows:

```

class Example {
    @log
    sum(a, b) {
        return a + b;
    }
}

const e = new Example();
e.sum(1, 2);
// Arguments: 1,2
// Result: 3

```

You'll notice that we had to use a slightly funny syntax to execute the decorated method. This could cover an entire article of its own, but in brief, the `apply` function allows you to call the function, specifying the `this` value and the arguments to call it with.

Taking it up a notch, we can arrange for our decorator to take some arguments. For example, let's re-write our `log` decorator as follows:

```

function log(name) {
    return function decorator(t, n, descriptor) {
        const original = descriptor.value;
        if (typeof original === 'function') {
            descriptor.value = function(...args) {
                console.log(`Arguments for ${name}: ${args}`);
                try {

```

```

        const result = original.apply(this, args);
        console.log(`Result from ${name}: ${result}`);
        return result;
    } catch (e) {
        console.log(`Error from ${name}: ${e}`);
        throw e;
    }
}
return descriptor;
};

}

```

This is getting more complex now, but when we break it down we have this:

- A function, `log`, that takes a single parameter: `name`.
- This function then returns a function that *is itself a decorator*.

This is identical to the earlier `log` decorator, except that it makes use of the `name` parameter from the outer function.

This is then used as follows:

```

class Example {
    @log('some tag')
    sum(a, b) {
        return a + b;
    }
}

const e = new Example();
e.sum(1, 2);
// Arguments for some tag: 1,2
// Result from some tag: 3

```

Straight away we can see that this allows us to distinguish between different log lines using a tag that we've supplied ourselves.

This works because the `log('some tag')` function call is evaluated by the JavaScript runtime straight away, and then the response from that is used as the decorator for the `sum` method.

Class decorators

Class decorators are applied to the entire class definition all in one go. The decorator function is called with a single parameter which is the constructor function being decorated.

Note that this is applied to the constructor function and not to each instance of the class that is created. This means that if you want to manipulate the instances you need to do so yourself by returning a wrapped version of the constructor.

In general, these are less useful than class member decorators, because everything you can do here you can do with a simple function call in exactly the same way. Anything you do with these needs to end up returning a new constructor function to replace the class constructor.

Going back to our logging example, let's write one that logs the constructor parameters:

```
function log(Class) {
  return (...args) => {
    console.log(args);
    return new Class(...args);
  };
}
```

Here we are accepting a class as our argument, and returning a new function that will act as the constructor. This simply logs the arguments and returns a new instance of the class constructed with those arguments.

For example:

```
@log
class Example {
  constructor(name, age) {
  }
}

const e = new Example('Graham', 34);
// [ 'Graham', 34 ]
console.log(e);
// Example {}
```

We can see that constructing our Example class will log out the arguments provided and that the constructed value is indeed an instance of Example. Exactly what we wanted.

Passing parameters into class decorators works exactly the same as for class members:

```
function log(name) {
  return function decorator(Class) {
    return (...args) => {
      console.log(`Arguments for ${name}: args`);
      return new Class(...args);
    };
  }
}

@log('Demo')
class Example {
  constructor(name, age) {}
}

const e = new Example('Graham', 34);
// Arguments for Demo: args
console.log(e);
// Example {}
```

Real World Examples

Core decorators

There's a fantastic library called [Core Decorators](#) that provides some very useful common decorators that are ready to use right now. These generally allow for very useful common functionality (e.g. timing of method calls, deprecation warnings, ensuring that a value is read-only) but utilizing the much cleaner decorator syntax.

React

The [React](#) library makes very good use of the concept of Higher-Order Components. These are simply React components that are written as a function, and that wrap around another component.

These are an ideal candidate for using as a decorator, because there's very little you need to change to do so. For example, the [react-redux library](#) has a function, connect, that's used to connect a React component to a Redux store.

In general, this would be used as follows:

```
class MyReactComponent extends React.Component {}  
export default connect(mapStateToProps, mapDispatchToProps)(MyReactC
```

However, because of how the decorator syntax works, this can be replaced with the following code to achieve the exact same functionality:

```
@connect(mapStateToProps, mapDispatchToProps)  
export default class MyReactComponent extends React.Component {}
```

MobX

The [MobX](#) library makes extensive use of decorators, allowing you to easily mark fields as Observable or Computed, and marking classes as Observers.

Summary

Class member decorators provide a very good way of wrapping code inside a class in a very similar way to how you can already do so for freestanding functions. This provides a good way of writing some simple helper code that can be applied to a lot of places in a very clean and easy-to-understand manner.

The only limit to using such a facility is your imagination!

Chapter 15: Enhanced Object Literals

by Craig Buckler

This chapter looks at what's possible with object literals in JavaScript, especially in the light of recent ECMAScript updates.

The ability to create JavaScript objects using literal notation is powerful. New features introduced from ES2015 (ES6) make object handling even easier in all modern browsers (not IE) and Node.js.

Creating objects in some languages can be expensive in terms of development time and processing power when a class must be declared before anything can be achieved. In JavaScript, it's easy to create objects on the fly. For example:

```
// ES5-compatible code
var myObject = {
  prop1: 'hello',
  prop2: 'world',
  output: function() {
    console.log(this.prop1 + ' ' + this.prop2);
  }
};

myObject.output(); // hello world
```

Single-use objects are used extensively. Examples include configuration settings, module definitions, method parameters, return values from functions, etc. ES2015 (ES6) added a range of features to enhance object literals.

Object Initialization From Variables

Objects' properties are often created from variables with the same name. For example:

```
// ES5 code
var
```

```
a = 1, b = 2, c = 3;
obj = {
  a: a,
  b: b,
  c: c
};

// obj.a = 1, obj.b = 2, obj.c = 3
```

There's no need for nasty repetition in ES6!...

```
// ES6 code
const
  a = 1, b = 2, c = 3;
  obj = {
    a
    b
    c
  };

// obj.a = 1, obj.b = 2, obj.c = 3
```

This could be useful for returned objects when using a [revealing module pattern](#), which (effectively) namespaces code in order to avoid naming conflicts. For example:

```
// ES6 code
const lib = (() => {

  function sum(a, b) { return a + b; }
  function mult(a, b) { return a * b; }

  return {
    sum,
    mult
  };
}());

console.log( lib.sum(2, 3) ); // 5
console.log( lib.mult(2, 3) ); // 6
```

You've possibly seen it used in ES6 modules:

```
// lib.js
function sum(a, b) { return a + b; }
function mult(a, b) { return a * b; }
```

```
export { sum, mult };
```

Object Method Definition Shorthand

Object methods in ES5 require the `function` statement. For example:

```
// ES5 code
var lib = {
  sum: function(a, b) { return a + b; },
  mult: function(a, b) { return a * b; }
};

console.log( lib.sum(2, 3) ); // 5
console.log( lib.mult(2, 3) ); // 6
```

This is no longer necessary in ES6; it permits the following shorthand syntax:

```
// ES6 code
const lib = {
  sum(a, b) { return a + b; },
  mult(a, b) { return a * b; }
};

console.log( lib.sum(2, 3) ); // 5
console.log( lib.mult(2, 3) ); // 6
```

It's not possible to use ES6 fat arrow `=>` function syntax here, because the method requires a name. That said, you can use arrow functions if you name each method directly (like ES5). For example:

```
// ES6 code
const lib = {
  sum: (a, b) => a + b,
  mult: (a, b) => a * b
};

console.log( lib.sum(2, 3) ); // 5
console.log( lib.mult(2, 3) ); // 6
```

Dynamic Property Keys

In ES5, it wasn't possible to use a variable for a key name, although it could be added *after* the object had been created. For example:

```
// ES5 code
var
  key1 = 'one',
  obj = {
    two: 2,
    three: 3
  };
obj[key1] = 1;
// obj.one = 1, obj.two = 2, obj.three = 3
```

Object keys can be dynamically assigned in ES6 by placing an expression in [square brackets]. For example:

```
// ES6 code
const
  key1 = 'one',
  obj = {
    [key1]: 1,
    two: 2,
    three: 3
  };
// obj.one = 1, obj.two = 2, obj.three = 3
```

Any expression can be used to create a key. For example:

```
// ES6 code
const
  i = 1,
  obj = {
    ['i' + i]: i
  };
console.log(obj.i1); // 1
```

A dynamic key can be used for methods as well as properties. For example:

```
// ES6 code
const
  i = 2,
  obj = {
    ['mult' + i]: x => x * i
  };
console.log( obj.mult2(5) ); // 10
```

Whether you *should* create dynamic properties and methods is another matter. The code can be difficult to read, and it may be preferable to create object factories or classes.

Destructuring (Variables From Object Properties)

It's often necessary to extract a property value from an object into another variable. This had to be explicitly declared in ES5. For example:

```
// ES5 code
var myObject = {
  one: 'a',
  two: 'b',
  three: 'c'
};

var
  one = myObject.one, // 'a'
  two = myObject.two, // 'b'
  three = myObject.three; // 'c'
```

ES6 supports destructuring: you can create a variable with the same name as an equivalent object property. For example:

```
// ES6 code
const myObject = {
  one: 'a',
  two: 'b',
  three: 'c'
};

const { one, two, three } = myObject;
// one = 'a', two = 'b', three = 'c'
```

It's also possible to assign properties to variables with any name using the notation `{ propertyName: newVariable }`. For example:

```
// ES6 code
const myObject = {
  one: 'a',
  two: 'b',
  three: 'c'
};
```

```
const { one: first, two: second, three: third } = myObject;
// first = 'a', second = 'b', third = 'c'
```

More complex objects with nested arrays and sub-objects can also be referenced in destructuring assignments. For example:

```
// ES6 code
const meta = {
  title: 'Enhanced Object Literals',
  pageinfo: {
    url: 'https://www.sitepoint.com/',
    description: 'How to use object literals in ES2015 (ES6).',
    keywords: 'javascript, object, literal'
  }
};

const {
  title: doc,
  pageinfo: { keywords: topic }
} = meta;

/*
  doc = 'Enhanced Object Literals'
  topic = 'javascript, object, literal'
*/
```

This initially appears complicated, but remember that in all destructuring assignments:

- the left-hand side of the assignment is the **destructuring source** — the array or object which holds the data being extracted
- the right-hand side of the assignment is the **destructuring target** — the pattern which defines the variable being assigned.

There are a number of caveats. You can't start a statement with a curly brace, because it looks like a code block. For example:

```
{ a, b, c } = myObject; // FAILS
```

You must either declare the variables — for example:

```
const { a, b, c } = myObject; // WORKS
```

or use parentheses if variables have already been declared — for example:

```
let a, b, c;
({ a, b, c } = myObject); // WORKS
```

You should therefore be careful not to mix declared and undeclared variables.

There are a number of situations where object destructuring is useful.

Default Function Parameters

It's often easier to pass a single object to a function than use a long list of arguments. For example:

```
prettyPrint( {
  title: 'Enhanced Object Literals',
  publisher: {
    name: 'SitePoint',
    url: 'https://www.sitepoint.com/'
  }
});
```

In ES5, it's necessary to parse the object to ensure appropriate defaults are set. For example:

```
// ES5 assign defaults
function prettyPrint(param) {

  param = param || {};
  var
    pubTitle = param.title || 'No title',
    pubName = (param.publisher && param.publisher.name) || 'No publi
  return pubTitle + ', ' + pubName;
}
```

In ES6, we can assign a default value to any parameter. For example:

```
// ES6 default value
function prettyPrint(param = {}) { ... }
```

We can then use destructuring to extract values and assign defaults where necessary:

```
// ES6 destructured default value
function prettyPrint(
```

```
{  
  title: pubTitle = 'No title',  
  publisher: { name: pubName = 'No publisher' }  
} = {}  
) {  
  
  return `${pubTitle}, ${pubName}`;  
  
}
```

Whether you find this code easier to read is another matter!

Parsing Returned Objects

Functions can only return one value, but that could be an object with hundreds of properties and/or methods. In ES5, it's necessary to obtain the returned object, then extract values accordingly. For example:

```
// ES5 code  
var  
  obj = getObject(),  
  one = obj.one,  
  two = obj.two,  
  three = obj.three;
```

ES6 destructuring makes this process simpler, and there's no need to retain the object as a variable:

```
// ES6 code  
const { one, two, three } = getObject();
```

You may have seen similar assignments in Node.js code. For example, if you only required the File System (`fs`) methods `readFile` and `writeFile`, you could reference them directly. For example:

```
// ES6 Node.js  
const { readFile, writeFile } = require('fs');  
  
readFile('file.txt', (err, data) => {  
  console.log(err || data);  
});  
  
writeFile('new.txt', 'new content', err => {  
  console.log(err || 'file written');  
});
```

ES2018 (ES9) Rest/Spread Properties

In ES2015, rest parameter and spread operator three-dot (...) notation applied to arrays only. ES2018 enables similar rest/spread functionality for objects. A basic example:

```
const myObject = {  
  a: 1,  
  b: 2,  
  c: 3  
};  
  
const { a, ...x } = myObject;  
// a = 1  
// x = { b: 2, c: 3 }
```

You can use the technique to pass values to a function:

```
restParam({  
  a: 1,  
  b: 2,  
  c: 3  
});  
  
function restParam({ a, ...x }) {  
  // a = 1  
  // x = { b: 2, c: 3 }  
}
```

You can only use a single rest property at the end of the declaration. In addition, it only works on the top level of each object and not sub-objects.

The spread operator can be used within other objects. For example:

```
const  
  obj1 = { a: 1, b: 2, c: 3 },  
  obj2 = { ...obj1, z: 26 };  
  
// obj2 is { a: 1, b: 2, c: 3, z: 26 }
```

You could use the spread operator to clone objects (`obj2 = { ...obj1 };`), but be aware you only get shallow copies. If a property holds another object, the clone will refer to the same object.

ES2018 (ES9) rest/spread property support is patchy, but it's available in Chrome, Firefox and Node.js 8.6+.

Object literals have always been useful. The new features introduced from ES2015 did not fundamentally change how JavaScript works, but they save typing effort and lead to clearer, more concise code.

Chapter 16: Introduction to the Fetch API

by Ludovico Fischer

In this chapter, we'll learn what the new Fetch API looks like, what problems it solves, and the most practical way to retrieve remote data inside your web page using the `fetch()` function.

For years, [XMLHttpRequest](#) has been web developers' trusted sidekick. Whether directly or under the hood, XMLHttpRequest has enabled Ajax and a whole new type of interactive experience, from Gmail to Facebook.

However, XMLHttpRequest is slowly being superseded by the [Fetch API](#). Both can be used to make network requests, but the Fetch API is Promise-based, which enables a cleaner, more concise syntax and helps keep you out of [callback hell](#).

The Fetch API

The Fetch API provides a `fetch()` method defined on the `window` object, which you can use to perform requests. This method returns a [Promise](#) that you can use to retrieve the response of the request.

The `fetch` method only has one mandatory argument, which is the URL of the resource you wish to fetch. A very basic example would look something like the following. This fetches the top five posts from [r/javascript on Reddit](#):

```
fetch('https://www.reddit.com/r/javascript/top/.json?limit=5')
  .then(res => console.log(res));
```

If you inspect the response in your browser's console, you should see a Response object with several properties:

```
{
  body: ReadableStream
```

```
bodyUsed: false
headers: Headers {}
ok: true
redirected: false
status: 200
statusText: ""
type: "cors"
url: "https://www.reddit.com/top/.json?count=5"
}
```

It seems that the request was successful, but where are our top five posts? Let's find out.

Loading JSON

We can't block the user interface waiting until the request finishes. That's why `fetch()` returns a `Promise`, an object which represents a future result. In the above example, we're using the `then` method to wait for the server's response and log it to the console.

Now let's see how we can extract the JSON payload from that response once the request completes:

```
fetch('https://www.reddit.com/r/javascript/top/.json?limit=5')
  .then(res => res.json())
  .then(json => console.log(json));
```

We start the request by calling `fetch()`. When the promise is fulfilled, it returns a `Response` object, which exposes a `json` method. Within the first `then()` we can call this `json` method to return the response body as JSON.

However, the `json` method also returns a promise, which means we need to chain on another `then()`, before the JSON response is logged to the console.

And why does `json()` return a promise? Because HTTP allows you to stream content to the client chunk by chunk, so even if the browser receives a response from the server, the content body might not all be there yet!

Async ... await

The `.then()` syntax is nice, but a more concise way to process promises in 2018

is using `async ... await` — a new syntax introduced by ES2017. Using `async` Ú `await` means that we can mark a function as `async`, then wait for the promise to complete with the `await` keyword, and access the result as a normal object. Async functions are supported in all modern browsers (not IE or Opera Mini) and Node.js 7.6+.

Here's what the above example would look like (slightly expanded) using `async ... await`:

```
async function fetchTopFive(sub) {
  const URL = `https://www.reddit.com/r/${sub}/top/.json?limit=5`;
  const fetchResult = fetch(URL)
  const response = await fetchResult;
  const jsonData = await response.json();
  console.log(jsonData);
}

fetchTopFive('javascript');
```

Not much has changed. Apart from the fact that we've created an `async` function, to which we're passing the name of the subreddit, we're now awaiting the result of calling `fetch()`, then using `await` again to retrieve the JSON from the response.

That's the basic workflow, but things involving remote services doesn't always go smoothly.

Handling Errors

Imagine we ask the server for a non-existing resource or a resource that requires authorization. With `fetch()`, you must handle application-level errors, like 404 responses, inside the normal flow. As we saw previously, `fetch()` returns a `Response` object with an `ok` property. If `response.ok` is `true`, the response status code lies within the 200 range:

```
async function fetchTopFive(sub) {
  const URL = `http://httpstat.us/404`; // Will return a 404
  const fetchResult = fetch(URL)
  const response = await fetchResult;
  if (response.ok) {
    const jsonData = await response.json();
    console.log(jsonData);
```

```
    } else {
      throw Error(response.statusText);
    }
}

fetchTopFive('javascript');
```

The meaning of a response code from the server varies from API to API, and oftentimes checking `response.ok` might not be enough. For example, some APIs will return a 200 response even if your API key is invalid. Always read the API documentation!

To handle a network failure, use a `try ... catch` block:

```
async function fetchTopFive(sub) {
  const URL = `https://www.reddit.com/r/${sub}/top/.json?limit=5`;
  try {
    const fetchResult = fetch(URL)
    const response = await fetchResult;
    const jsonData = await response.json();
    console.log(jsonData);
  } catch(e){
    throw Error(e);
  }
}

fetchTopFive('javvascript'); // Notice the incorrect spelling
```

The code inside the `catch` block will run only when a network error occurs.

You've learned the basics of making requests and reading responses. Now let's customize the request further.

Change the Request Method and Headers

Looking at the example above, you might be wondering why you couldn't just use one of the [existing XMLHttpRequest wrappers](#). The reason is that there's more the `fetch` API offers you than just the `fetch()` method.

While you must use the same instance of `XMLHttpRequest` to perform the request and retrieve the response, the `fetch` API lets you configure request objects explicitly.

For example, if you need to change how `fetch()` makes a request (e.g. to configure the request method) you can pass a `Request` object to the `fetch()` function. The first argument to the `Request` constructor is the request URL, and the second argument is an option object that configures the request:

```
async function fetchTopFive(sub) {
  const URL = `https://www.reddit.com/r/${sub}/top/.json?limit=5`;
  try {
    const fetchResult = fetch(
      new Request(URL, { method: 'GET', cache: 'reload' })
    );
    const response = await fetchResult;
    const jsonData = await response.json();
    console.log(jsonData);
  } catch(e){
    throw Error(e);
  }
}

fetchTopFive('javascript');
```

Here, we specified the request method and asked it never to cache the response.

You can change the request headers by assigning a `Headers` object to the `headers` field. Here's how to ask for JSON content only with the 'Accept' header:

```
const headers = new Headers();
headers.append('Accept', 'application/json');
const request = new Request(URL, { method: 'GET', cache: 'reload', h
```

You can create a new request from an old one to tweak it for a different use case. For example, you can create a POST request from a GET request to the same resource. Here's an example:

```
const postReq = new Request(request, { method: 'POST' });
```

You also can access the response headers, but keep in mind that they're read-only values.

```
fetch(request).then(response => console.log(response.headers));
```

`Request` and `Response` follow the HTTP specification closely; you should recognize them if you've ever used a server-side language. If you're interested in

finding out more, you can read all about them on the [Fetch API page on MDN](#).

Bringing it all Together

To round off the chapter, [here's a runnable example](#) demonstrating how to fetch the top five posts from a particular subreddit and display their details in a list.

Try entering a few subreddits (e.g. 'javascript', 'node', 'linux', 'lolcats') as well as a couple of non-existent ones.

Where to Go from Here

In this chapter, you've seen what the new Fetch API looks like and what problems it solves. I've demonstrated how to retrieve remote data with the `fetch()` method, how to handle errors and to create Request objects to control the request method and headers.

Support for `fetch()` is good. If you need to support older browsers a [polyfill is available](#).

So next time you reach for a library like jQuery to make Ajax requests, take a moment to think if you could use native browser methods instead.

Chapter 17: ES6 (ES2015) and Beyond: Understanding JavaScript Versioning

by James Wright

As programming languages go, JavaScript's development has been positively frantic in the last few years. With each year now seeing a new release of the ECMAScript specification, it's easy to get confused about JavaScript versioning, which version supports what, and how you can future-proof your code.

To better understand the how and why behind this seemingly constant stream of new features, let's take a brief look at the history of the JavaScript and JavaScript versioning, and find out why the standardization process is so important.

The Early History of JavaScript Versioning

The prototype of JavaScript was written in just ten days in May 1995 by Brendan Eich. He was initially recruited to implement a Scheme runtime for Netscape Navigator, but the management team pushed for a C-style language that would complement the then recently released Java.

JavaScript made its debut in version 2 of Netscape Navigator in December 1995. The following year, Microsoft reverse-engineered JavaScript to create their own version, calling it JScript. JScript shipped with version 3 of the Internet Explorer browser, and was almost identical to JavaScript — even including all the same bugs and quirks — but it did have some extra Internet Explorer-only features.

The Birth of ECMAScript

The necessity of ensuring that JScript (and any other variants) remained

compatible with JavaScript motivated Netscape and Sun Microsystems to standardize the language. They did this with the help of the [European Computer Manufacturers Association](#), who would host the standard. The standardized language was called ECMAScript to avoid infringing on Sun's Java trademark — a move that caused a fair deal of confusion. Eventually ECMAScript was used to refer to the specification, and JavaScript was (and still is) used to refer to the language itself.

The working group in charge of JavaScript versioning and maintaining ECMAScript is known as [Technical Committee 39](#), or TC39. It's made up of representatives from all the major browser vendors such as Apple, Google, Microsoft and Mozilla, as well as invited experts and delegates from other companies with an interest in the development of the Web. They have regular meetings to decide on how the language will develop.

When JavaScript was standardized by TC39 in 1997, the specification was known as ECMAScript version 1. Subsequent versions of ECMAScript were initially released on an annual basis, but ultimately became sporadic due to the lack of consensus and the unmanageably large feature set surrounding ECMAScript 4. This version was thus terminated and downsized into 3.1, but wasn't finalized under that moniker, instead eventually evolving into ECMAScript 5. This was released in December 2009, 10 years after ECMAScript 3, and introduced a [JSON serialization API](#), [Function.prototype.bind](#), and [strict mode](#), amongst other capabilities. A maintenance release to clarify some of the ambiguity of the latest iteration, 5.1, was released two years later.

ECMAScript 2015 and the Resurgence of Yearly Releases

With the resolution of TC39's disagreement resulting from ECMAScript 4, Brendan Eich stressed the need for nearer-term, smaller releases. The first of these new specifications was *ES2015* (originally named ECMAScript 6, or ES6). This edition was a large but necessary foundation to support the future, annual JavaScript versioning. It includes many features that are well-loved by many developers today, such as:

- [Classes](#)

- [Promises](#)
- [Arrow functions](#)
- [ES Modules](#)
- [Generators and Iterators](#)

ES2015 was the first offering to follow the *TC39 process*, a proposal-based model for discussing and adopting elements.

The TC39 Process

There are five stages through which a proposal must pass before it can be accepted into an upcoming version of ECMAScript.

Stage 0: Strawman

This is a convenience step to permit the submission of ideas to the specification. Features can be suggested by anyone — namely, TC39 members and non-members who have registered as a contributor.

Stage 1: Proposal

The first stage at which a proposal is formalized. It's necessary that:

- any existing problems rectified by the solution are described
- an API outline is provided, alongside high-level implementation details, as well as polyfills and/or demos
- potential impediments are discussed upfront.

A *champion* must be selected to adopt and advance the proposal. This individual must be a TC39 member.

Stage 2: Draft

This is the milestone at which a feature is likely to be included in a future version of ECMAScript. Here, the proposal's syntax and semantics are detailed using the [formal language](#) described by the specification. An experimental implementation should be available at this point.

Stage 3: Candidate

Here, the majority of the proposal and the backing technology have been developed, but it requires further feedback from users and implementers (such as browser vendors). Once this is available and acted upon, the outline and specification details are finalized and signed off by designated reviewers and the appointed editor. As a compliant implementation is required at this stage, only critical changes are henceforth embraced.

Stage 4: Finished

The proposal has been accepted and can be added to ECMAScript. It's thus inherent that:

- acceptance tests, which are part of the [Test262 suite](#) and are crafted with JavaScript, have been written to prove the conformity and behavior of the feature
- at least two compliant implementations are available, and have shipped, all of which demonstrate robustness and developer usability
- a pull request has been submitted to the [official ECMA-262 repo](#), which has been signed off by the specification editor.

The above repository's [contribution document](#) further details the use of GitHub issues and pull requests for managing additions to the language.

Moving Forward

Following the completion of ES2015 and the establishment of the TC39 process of JavaScript versioning and updating, subsequent releases have occurred each June, with the inclusion of proposals being timeboxed to one year. At the time of writing, there have been three new specifications.

ES2016

Also known as ES7, this was the first smaller, incremental version of ECMAScript. Aside from bug fixes, it added just two features.

[Array.prototype.includes](#)

This instance method simplifies searching for values in an Array:

```
// pre-ES2016:  
const hasBob = names.indexOf('bob') > -1;  
  
// ES2016:  
const hasBob = names.includes('bob');
```

Exponent Operator

Prior to ES2016, one could perform exponentiation with `Math.pow(base, exponent)`. This version [introduces an operator \(**\)](#) that has its own precedence:

```
// pre-ES2016  
Math.pow(5, 3); // => 125  
  
// ES2016  
5 ** 3; // => 125
```

ES2017

A slightly larger release, ES2017 (aka ES8) contains a handful of useful methods and syntactical constructs.

Asynchronous Functions

Promises have saved us from callback hell, but their API nonetheless demonstrates verbosity. [Asynchronous functions](#) abstract them with a syntax that closely resembles synchronous code:

```
// promises  
const getProfile = name => {  
  return fetch(`https://some-api/people/${name}`)  
    .then(res => res.json())  
    .then(({ profile }) => profile); // destructuring `profile` from  
};  
  
// async/await  
const getProfile = async name => {  
  const res = await fetch(`https://some-api/people/${name}`);  
  const { profile } = await res.json();  
  return profile;  
};
```

String Padding Methods

[`String.prototype.padStart\(length, padder\)`](#) and [`padEnd\(length, padder\)`](#) will respectively prepend and append padder (this is optional, defaulting to a space) to a string repeatedly until it reaches length characters:

```
'foo'.padStart(6);           // => '    foo';
'foo'.padEnd(6);            // => 'foo    ';
'foo'.padStart(10, 'bar');   // => 'barbarbfoo';
'foo'.padEnd(10, 'bar');    // => 'foobarbarb';
```

Other features include [trailing commas](#), [shared memory and atomics](#), and static Object methods ([`Object.entries\(\)`](#), [`Object.values\(\)`](#), and [`Object.getOwnPropertyDescriptors\(\)`](#).)

If you'd like to read more about the complete feature set of ES2017, please see the next chapter.

ES2018

This latest iteration, at the time of writing, introduces a small set of powerful additions.

Asynchronous Iterators

While `Promise.all()` allows you to await the resolution of multiple promises, there are cases in which you may need to sequentially iterate over asynchronously-retrieved values. It's now possible to await [async iterators](#) along with arrays of promises:

```
(async () => {
  const personRequests = ['bob', 'sarah', 'laura'].map(
    n => fetch(`https://api/people/${n}`)
  );

  for await (const response of personRequests) {
    console.log(await response.json());
  }
})();
```

Object Spread and Rest Properties

Ostensibly, these two syntactical improvements are already popular amongst JavaScript developers thanks to the availability of compilers such as [Babel](#). [Object spread and rest properties](#) are similar to array spread and rest properties, and permit the shallow copying and grouped destructuring of object properties:

```
const react = {  
    name: 'React',  
    vendor: 'Facebook',  
    description: 'A JavaScript library for building user interfaces',  
    npm: true,  
    cdn: true,  
};  
  
/* Use spread syntax inside an object literal to create  
 * a shallow copy, while overriding certain properties.  
 */  
const vue = {  
    ...react,  
    vendor: 'Evan You',  
    description: 'A JavaScript framework for building UIs',  
};  
  
/* Use rest within destructuring syntax to create a  
 * label for accessing additional object properties.  
 */  
const { name, vendor, ...rest } = vue;  
console.log(rest.description); // => 'A JavaScript framework for bui
```

Other accepted proposals are [Promise.prototype.finally\(\)](#), as well as enhancements to [regular expressions](#) and [template literals](#).

If you'd like to read more about the complete feature set of ES2018, check out the final chapter of this book.

A Final Word

JavaScript has evolved greatly over a short space of time. While this is attributable to the ECMAScript standard and the brilliant work of TC39, it was initially an arduous journey due to the previous lack of stability and cohesion in JavaScript versioning and development.

Thanks to the relatively mature proposals process, the language can only improve in a pragmatic and manageable manner. It's a great time to be a web

developer!

Chapter 18: What's New in ES2017: Async Functions, Improved Objects, and More

by Craig Buckler

Let's take a look at the most important JavaScript updates that came with ES2017, and also briefly cover how this updating process actually takes place.

The Update Process

JavaScript (ECMAScript) is an ever-evolving standard implemented by many vendors across multiple platforms. ES6 (ECMAScript 2015) was a large release which took six years to finalize. A new annual release process was formulated to streamline the process and rapidly add new features.

The modestly named Technical Committee 39 (TC39) consists of parties including browser vendors who meet to push JavaScript proposals along a strict progression path:

Stage 0: strawman - An initial submission of ideas for new or improved ECMAScript features.

Stage 1: proposal - A formal proposal document championed by at least one member of TC39, which includes API examples, language semantics, algorithms, potential obstacles, polyfills and demonstrations.

Stage 2: draft - An initial version of the feature specification. Two experimental implementations of the feature are required, although one can be in a transpiler such as Babel.

Stage 3: candidate - The proposal specification is reviewed and feedback is gathered from vendors.

Stage 4: finished - The proposal is ready for inclusion in ECMAScript. A feature should only be considered a standard once it reaches this stage. However, it can take longer to ship in browsers and runtimes such as Node.js.

If ES2015 was too large, ES2016 was purposely tiny to prove the standardization process. Two new features were added:

1. The array `.includes()` method which returns true or false when a value is contained in an array, and
2. The `a ** b` exponentiation operator, which is identical to `Math.pow(a, b)`.

What's New in ES2017

The feature set for ES2017 (or ES8 in old money) is considered to be the first proper amendment to the ECMAScript specification. It delivers the following goods ...

Async functions

Unlike most languages, JavaScript is asynchronous by default. Commands which can take any amount of time do not halt execution. That includes operations such as requesting a URL, reading a file, or updating a database. A callback function must be passed, which executes when the result of that operation is known.

This can lead to callback hell when a series of nested asynchronous functions must be executed in order. For example:

```
function doSomething() {  
  doSomething1((response1) => {  
    doSomething2(response1, (response2) => {  
      doSomething3(response2, (response3) => {  
        // etc...  
      };  
    });  
  });  
}
```

ES2015 (ES6) introduced Promises, which provided a cleaner way to express the same functionality. Once your functions were Promisified, they could be

executed using:

```
function doSomething() {  
  doSomething1()  
  .then(doSomething2)  
  .then(doSomething3)  
}
```

ES2017 Async functions expand on Promises to make asynchronous calls even clearer:

```
async function doSomething() {  
  const  
    response1 = await doSomething1(),  
    response2 = await doSomething2(response1),  
    response3 = await doSomething3(response2);  
}
```

`await` effectively makes each call appear as though it's synchronous while not holding up JavaScript's single processing thread.

Async functions are supported in all modern browsers (not IE or Opera Mini) and Node.js 7.6+. They'll change the way you write JavaScript, and a whole article could be dedicated to callbacks, Promises and Async functions. Fortunately, we have one! Refer to [Flow Control in Modern JavaScript](#).

Object.values()

`Object.values()` is a quick and more declarative way to extract an array of values from name-value pairs within an object. For example:

```
const myObject = {  
  a: 1,  
  b: 'Two',  
  c: [3,3,3]  
}  
  
const values = Object.values(myObject);  
// [ 1, 'Two', [3,3,3] ]
```

You need never write a `for ... of` loop again! `Object.values` is natively supported in all modern browsers (not IE or Opera Mini) and Node.js 7.0+.

Object.entries()

`Object.entries()` returns an array from an object containing name–value pairs. Each value in the returned array is a sub-array containing the name (index 0) and value (index 1). For example:

```
const myObject = {
  a: 1,
  b: 'Two',
  c: [3,3,3]
}

const entries = Object.entries(myObject);
/*
[
  [ 'a', 1 ],
  [ 'b', 'Two' ],
  [ 'c', [3,3,3] ]
]*/

```

This provides another way to iterate over the properties of an object. It can also be used to define a [Map](#):

```
const map = new Map(Object.entries({
  a: 1,
  b: 2,
  c: 3
}));
```

`Object.values` is natively supported in most modern browsers (but not IE, Opera Mini and iOS Safari) and Node.js 7.0+.

Object.getOwnPropertyDescriptors()

The `Object.getOwnPropertyDescriptors()` method returns another object containing all property descriptors (`.value`, `.writable`, `.get`, `.set`, `.configurable`, `.enumerable`).

The properties are directly present on an object and not in the object's prototype chain. It's similar to [Object.getOwnPropertyDescriptor\(object, property\)](#) — except all properties are returned, rather than just one. For example:

```
const myObject = {  
  prop1: 'hello',  
  prop2: 'world'  
};  
  
const descriptors = Object.getOwnPropertyDescriptors(myObject);  
  
console.log(descriptors.prop1.writable); // true  
console.log(descriptors.prop2.value); // 'world'
```

padStart() and padEnd() String Padding

String padding has been controversial in JavaScript. The popular [left-pad](#) library was pulled from npm after it attracted the attention of lawyers representing an instant messaging app with the same name. Unfortunately, it had been used as a dependency in thousands of projects and the internet broke. npm subsequently changed operating procedures and left-pad was un-unpublished.

Native string padding has been added to ES2017, so there's no need to use a third-party module. `.padStart()` and `.padEnd()` add characters to the start or end of a string respectively, until they reach a desired length. Both accept a minimum length and an optional 'fill' string (space is the default) as parameters. Examples:

```
'abc'.padStart(5);          // ' abc'  
'abc'.padStart(5, '-');    // '--abc'  
'abc'.padStart(10, '123'); // '1231231abc'  
'abc'.padStart(1);         // 'abc'  
  
'abc'.padEnd(5);          // 'abc  '  
'abc'.padEnd(5, '-');    // 'abc--'  
'abc'.padEnd(10, '123'); // 'abc1231231'  
'abc'.padEnd(1);         // 'abc'
```

`.padStart()` and `.padEnd()` are supported in all modern browsers (not IE) and Node.js 8.0+.

Trailing Commas are Permitted

A small ES2017 update: trailing commas no longer raise a syntax error in object definitions, array declarations, function parameter lists, and so on:

```
// ES2017 is happy!
const a = [1, 2, 3,];
const b = {
  a: 1,
  b: 2,
  c: 3,
};
function c(one,two,three,) {};
```

Trailing commas are enabled in all browsers and Node.js. However, trailing commas in function parameters are only supported in Chrome 58+ and Firefox 52+ at the time of writing.

SharedArrayBuffer and Atomics

The [SharedArrayBuffer](#) object is used to represent a fixed-length raw binary data buffer that can be shared between web workers. The [Atomics](#) object provided a predictable way to read from and write to memory locations defined by SharedArrayBuffer.

While both objects were implemented in Chrome and Firefox, it was disabled in January 2018 in response to the [Spectre vulnerability](#).

The full ECMAScript 2017 Language Specification is available at the [ECMA International website](#). Are you hungry for more? The new features in ES2018 have been announced, and we'll discuss them next.

Chapter 19: What's New in ES2018

by Craig Buckler

In this chapter, I'll cover the new features of JavaScript introduced via ES2018 (ES9), with examples of what they're for and how to use them.

JavaScript (ECMAScript) is an ever-evolving standard implemented by many vendors across multiple platforms. ES6 (ECMAScript 2015) was a large release which took six years to finalize. A new annual release process has been formulated to streamline the process and add features quicker. ES9 (ES2018) is the latest iteration at the time of writing.

Technical Committee 39 (TC39) consists of parties including browser vendors who meet to push JavaScript proposals along a strict progression path:

Stage 0: strawman - The initial submission of ideas.

Stage 1: proposal - A formal proposal document championed by at least one member of TC39 which includes API examples.

Stage 2: draft - An initial version of the feature specification with two experimental implementations.

Stage 3: candidate - The proposal specification is reviewed and feedback is gathered from vendors.

Stage 4: finished - The proposal is ready for inclusion in ECMAScript but may take longer to ship in browsers and Node.js.

ES2016

ES2016 proved the standardization process by adding just two small features:

1. The array [includes\(\)](#) method, which returns true or false when a value is

contained in an array, and

2. The `a ** b` [exponentiation](#) operator, which is identical to `Math.pow(a, b)`.

ES2017

ES2017 provided a larger range of new features:

- Async functions for a clearer Promise syntax
- `Object.values()` to extract an array of values from an object containing name-value pairs
- `Object.entries()`, which returns an array of sub-arrays containing the names and values in an object
- `Object.getOwnPropertyDescriptors()` to return an object defining property descriptors for own properties of another object (`.value`, `.writable`, `.get`, `.set`, `.configurable`, `.enumerable`)
- `padStart()` and `padEnd()`, both elements of string padding
- trailing commas on object definitions, array declarations and function parameter lists
- `SharedArrayBuffer` and `Atomics` for reading from and writing to shared memory locations (disabled in response to the [Spectre vulnerability](#)).

Refer to [What's New in ES2017](#) for more information.

ES2018

ECMAScript 2018 (or ES9 if you prefer the old notation) is now available. The following features have reached stage 4, although working implementations will be patchy across browsers and runtimes at the time of writing.

Asynchronous Iteration

At some point in your `async/await` journey, you'll attempt to call an asynchronous function *inside* a synchronous loop. For example:

```
async function process(array) {  
    for (let i of array) {  
        await doSomething(i);  
    }  
}
```

It won't work. Neither will this:

```
async function process(array) {  
  array.forEach(async i => {  
    await doSomething(i);  
  });  
}
```

The loops themselves remain synchronous and will always complete before their inner asynchronous operations.

ES2018 introduces asynchronous iterators, which are just like regular iterators except the `next()` method returns a Promise. Therefore, the `await` keyword can be used with `for ... of` loops to run asynchronous operations in series. For example:

```
async function process(array) {  
  for await (let i of array) {  
    doSomething(i);  
  }  
}
```

Promise.`finally()`

A Promise chain can either succeed and reach the final `.then()` or fail and trigger a `.catch()` block. In some cases, you want to run the same code regardless of the outcome — for example, to clean up, remove a dialog, close a database connection etc.

The `.finally()` prototype allows you to specify final logic in one place rather than duplicating it within the last `.then()` and `.catch()`:

```
function doSomething() {  
  doSomething1()  
  .then(doSomething2)  
  .then(doSomething3)  
  .catch(err => {  
    console.log(err);  
  })  
  .finally(() => {  
    // finish here!  
  });  
}
```

Rest/Spread Properties

ES2015 introduced the rest parameters and spread operators. The three-dot (...) notation applied to array operations only. Rest parameters convert the last arguments passed to a function into an array:

```
restParam(1, 2, 3, 4, 5);

function restParam(p1, p2, ...p3) {
  // p1 = 1
  // p2 = 2
  // p3 = [3, 4, 5]
}
```

The spread operator works in the opposite way, and turns an array into separate arguments which can be passed to a function. For example, `Math.max()` returns the highest value, given any number of arguments:

```
const values = [99, 100, -1, 48, 16];
console.log( Math.max(...values) ); // 100
```

ES2018 enables similar rest/spread functionality for object destructuring as well as arrays. A basic example:

```
const myObject = {
  a: 1,
  b: 2,
  c: 3
};

const { a, ...x } = myObject;
// a = 1
// x = { b: 2, c: 3 }
```

Or you can use it to pass values to a function:

```
restParam({
  a: 1,
  b: 2,
  c: 3
});

function restParam({ a, ...x }) {
  // a = 1
  // x = { b: 2, c: 3 }
```

```
}
```

Like arrays, you can only use a single rest parameter at the end of the declaration. In addition, it only works on the top level of each object and not sub-objects.

The spread operator can be used within other objects. For example:

```
const obj1 = { a: 1, b: 2, c: 3 };
const obj2 = { ...obj1, z: 26 };
// obj2 is { a: 1, b: 2, c: 3, z: 26 }
```

You could use the spread operator to clone objects (`obj2 = { ...obj1 };`), but be aware you only get shallow copies. If a property holds another object, the clone will refer to the same object.

Regular Expression Named Capture Groups

JavaScript regular expressions can return a match object — an array-like value containing matched strings. For example, to parse a date in YYYY-MM-DD format:

```
const
  reDate = /([0-9]{4})-([0-9]{2})-([0-9]{2})/,
  match = reDate.exec('2018-04-30'),
  year = match[1], // 2018
  month = match[2], // 04
  day = match[3]; // 30
```

It's difficult to read, and changing the regular expression is also likely to change the match object indexes.

ES2018 permits groups to be named using the notation `?<name>` immediately after the opening capture bracket `(`. For example:

```
const
  reDate = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/,
  match = reDate.exec('2018-04-30'),
  year = match.groups.year, // 2018
  month = match.groups.month, // 04
  day = match.groups.day; // 30
```

Any named group that fails to match has its property set to `undefined`.

Named captures can also be used in `replace()` methods. For example, convert a date to US MM-DD-YYYY format:

```
const
  reDate = /(?(?<year>[0-9]{4})-(?(?<month>[0-9]{2})-(?(?<day>[0-9]{2})/,
  d      = '2018-04-30',
  usDate = d.replace(reDate, '$<month>-$<day>-$<year>');
```

Regular Expression lookbehind Assertions

JavaScript currently supports *lookahead* assertions inside a regular expression. This means a match must occur but nothing is captured, and the assertion isn't included in the overall matched string. For example, to capture the currency symbol from any price:

```
const
  reLookahead = /\D(?=\\d+)/,
  match       = reLookahead.exec('$123.89');

console.log( match[0] ); // $
```

ES2018 introduces *lookbehind* assertions that work in the same way, but for preceding matches. We can therefore capture the price number and ignore the currency character:

```
const
  reLookbehind = /(?<=\\D)\\d+/,
  match       = reLookbehind.exec('$123.89');

console.log( match[0] ); // 123.89
```

This is a positive lookbehind assertion; a non-digit `\D` must exist. There's also a negative lookbehind assertion, which sets that a value must not exist. For example:

```
const
  reLookbehindNeg = /(?<!\\D)\\d+/,
  match          = reLookbehindNeg.exec('$123.89');

console.log( match[0] ); // null
```

Regular Expression s (dotAll) Flag

A regular expression dot . matches any single character *except* carriage returns. The s flag changes this behavior so line terminators are permitted. For example:

```
/hello.world/s.test('hello\nworld'); // true
```

Regular Expression Unicode Property Escapes

Until now, it hasn't been possible to access Unicode character properties natively in regular expressions. ES2018 adds Unicode property escapes — in the form \p{...} and \P{...} — in regular expressions that have the u (unicode) flag set. For example:

```
const reGreekSymbol = /\p{Script=Greek}/u;
reGreekSymbol.test('π'); // true
```

Template Literals Tweak

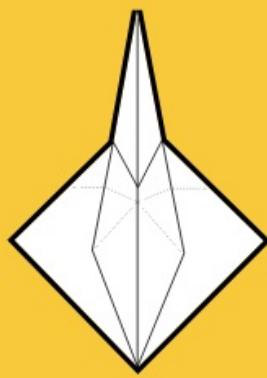
Finally, all syntactic restrictions related to escape sequences in template literals have been removed.

Previously, a \u started a unicode escape, an \x started a hex escape, and \ followed by a digit started an octal escape. This made it impossible to create certain strings such as a Windows file path c:\uuu\xxx\111. For more details, refer to the [MDN template literals documentation](#).

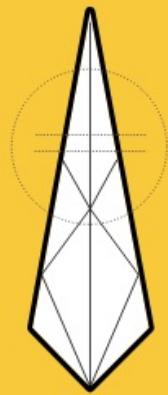
Book 2: JavaScript: Best Practice



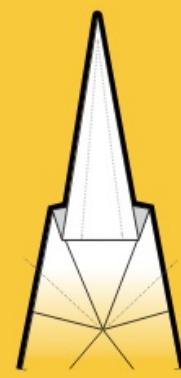
JAVASCRIPT: BEST PRACTICE



4.



5.



6.

CLEAN, MAINTAINABLE, PERFORMANT CODE

Chapter 1: The Anatomy of a Modern JavaScript Application

by James Kolce

There's no doubt that the JavaScript ecosystem changes fast. Not only are new tools and frameworks introduced and developed at a rapid rate, the language itself has undergone big changes with the introduction of ES2015 (aka ES6). Understandably, many articles have been written complaining about how difficult it is to learn modern JavaScript development these days.

In this article, I'll introduce you to modern JavaScript. We'll take a look at recent developments in the language and get an overview of the tools and techniques currently used to write front-end web applications. If you're just starting out with learning the language, or you've not touched it for a few years and are wondering what happened to the JavaScript you used to know, this article is for you.

A Note About Node.js

Node.js is a runtime that allows server-side programs to be written in JavaScript. It's possible to have full-stack JavaScript applications, where both the front and back end of the app is written in the same language. Although this article is focused on client-side development, Node.js still plays an important role.

The arrival of Node.js had a significant impact on the JavaScript ecosystem, introducing the npm package manager and popularizing the CommonJS module format. Developers started to build more innovative tools and develop new approaches to blur the line between the browser, the server, and native applications.

JavaScript ES2015+

In 2015, the sixth version of [ECMAScript](#) — the specification that defines the

JavaScript language — was released under the name of [ES2015](#) (still often referred to as ES6). This new version included substantial additions to the language, making it easier and more feasible to build ambitious web applications. But improvements don't stop with ES2015; each year, a new version is released.

Declaring variables

JavaScript now has two additional ways to declare variables: [let](#) and [const](#).

`let` is the successor to `var`. Although `var` is still available, `let` limits the scope of variables to the block (rather than the function) they're declared within, which reduces the room for error:

```
// ES5
for (var i = 1; i < 5; i++) {
  console.log(i);
}
// <-- logs the numbers 1 to 4
console.log(i);
// <-- 5 (variable i still exists outside the loop)

// ES2015
for (let j = 1; j < 5; j++) {
  console.log(j);
}
console.log(j);
// <-- 'Uncaught ReferenceError: j is not defined'
```

Using `const` allows you to define variables that cannot be rebound to new values. For primitive values such as strings and numbers, this results in something similar to a constant, as you cannot change the value once it has been declared:

```
const name = 'Bill';
name = 'Steve';
// <-- 'Uncaught TypeError: Assignment to constant variable.'

// Gotcha
const person = { name: 'Bill' };
person.name = 'Steve';
// person.name is now Steve.
// As we're not changing the object that person is bound to, JavaScr
```

Arrow functions

[Arrow functions](#) provide a cleaner syntax for declaring anonymous functions (lambdas), dropping the `function` keyword and the `return` keyword when the body function only has one expression. This can allow you to write functional style code in a nicer way:

```
// ES5
var add = function(a, b) {
  return a + b;
}

// ES2015
const add = (a, b) => a + b;
```

The other important feature of arrow functions is that they inherit the value of `this` from the context in which they are defined:

```
function Person(){
  this.age = 0;

  // ES5
  setInterval(function() {
    this.age++; // |this| refers to the global object
  }, 1000);

  // ES2015
  setInterval(() => {
    this.age++; // |this| properly refers to the person object
  }, 1000);
}

var p = new Person();
```

Improved Class syntax

If you're a fan of object-oriented programming, you might like the [addition of classes to the language](#) on top of the existent mechanism based on prototypes. While it's mostly just syntactic sugar, it provides a cleaner syntax for developers trying to emulate classical object-orientation with prototypes.

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

```
greet() {
  console.log(`Hello, my name is ${this.name}`);
}
}
```

Promises / Async functions

The asynchronous nature of JavaScript has long represented a challenge; any non-trivial application ran the risk of falling into a callback hell when dealing with things like Ajax requests.

Fortunately, ES2015 added native support for [promises](#). Promises represent values that don't exist at the moment of the computation but that may be available later, making the management of asynchronous function calls more manageable without getting into deeply nested callbacks.

ES2017 introduced [async functions](#) (sometimes referred to as `async/await`) that make improvements in this area, allowing you to treat asynchronous code as if it were synchronous:

```
async function doAsyncOp () {
  var val = await asynchronousOperation();
  console.log(val);
  return val;
};
```

Modules

Another prominent feature added in ES2015 is a native module format, making the definition and usage of modules a part of the language. Loading modules was previously only available in the form of third-party libraries. We'll look at modules in more depth in the next section.

There are other features we won't talk about here, but we've covered at some of the major differences you're likely to notice when looking at modern JavaScript. You can check a complete list with examples on the [Learn ES2015](#) page on the [Babel site](#), which you might find useful to get up to date with the language. Some of those features include template strings, block-scoped variables and constants, iterators, generators, new data structures such as Map and Set, and more.

Code linting

Linters are tools that parse your code and compare it against a set of rules, checking for syntax errors, formatting, and good practices. Although the use of a linter is recommended to everyone, it's especially useful if you're getting started. When configured correctly for your code editor/IDE, you can get instant feedback to ensure you don't get stuck with syntax errors as you're learning new language features.

You can [check out ESLint](#), which is one of the most popular and supports ES2015+.

Modular Code

Modern web applications can have thousands (even hundred of thousands) of lines of code. Working at that size becomes almost impossible without a mechanism to organize everything in smaller components, writing specialized and isolated pieces of code that can be reused as necessary in a controlled way. This is the job of modules.

CommonJS modules

A handful of module formats have emerged over the years, the most popular of which is [CommonJS](#). It's the default module format in Node.js, and can be used in client-side code with the help of module bundlers, which we'll talk about shortly.

It makes use of a `module` object to export functionality from a JavaScript file and a `require()` function to import that functionality where you need it.

```
// lib/math.js
function sum(x, y) {
  return x + y;
}

const pi = 3.141593

module.exports = {
  sum: sum,
  pi: pi
};
```

```
// app.js
const math = require("lib/math");

console.log("2π = " + math.sum(math.pi, math.pi));
```

ES2015 modules

ES2015 introduces a way to define and consume components right into the language, which was previously possible only with third-party libraries. You can have separate files with the functionality you want, and export just certain parts to make them available to your application.

Native Browser Support

At the time of writing, native browser support for ES2015 modules is still under development, so you currently need some additional tools to be able to use them.

Here's an example:

```
// lib/math.js

export function sum(x, y) {
  return x + y;
}
export const pi = 3.141593;
```

Here we have a module that *exports* a function and a variable. We can include that file in another one and use those exported functions:

```
// app.js

import * as math from "lib/math";

console.log("2π = " + math.sum(math.pi, math.pi));
```

Or we can also be specific and import only what we need:

```
// otherApp.js

import {sum, pi} from "lib/math";

console.log("2π = " + sum(pi, pi));
```

These examples have been extracted from the [Babel website](#). For an in-depth look, check out [Understanding ES6 Modules](#).

Package Management

Other languages have long had their own package repositories and managers to make it easier to find and install third-party libraries and components. Node.js comes with its own package manager and repository, [npm](#). Although there are other package managers available, npm has become the de facto JavaScript package manager and is said to be the largest package registry in the world.

In the [npm repository](#) you can find third-party modules that you can easily download and use in your projects with a single `npm install <package>` command. The packages are downloaded into a local `node_modules` directory, which contains all the packages and their dependencies.

The packages that you download can be registered as dependencies of your project in a [package.json](#) file, along with information about your project or module (which can itself be published as a package on npm).

You can define separate dependencies for both development and production. While the production dependencies are needed for the package to work, the development dependencies are only necessary for the developers of the package.

Example package.json file

```
{  
  "name": "demo",  
  "version": "1.0.0",  
  "description": "Demo package.json",  
  "main": "main.js",  
  "dependencies": {  
    "mkdirp": "^0.5.1",  
    "underscore": "^1.8.3"  
  },  
  "devDependencies": {},  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "Sitepoint",  
  "license": "ISC"  
}
```

Build Tools

The code that we write when developing modern JavaScript web applications almost never is the same code that will go to production. We write code in a modern version of JavaScript that may not be supported by the browser, we make heavy use of third-party packages that are in a `node_modules` folder along with their own dependencies, we can have processes like static analysis tools or minifiers, etc. Build tooling exists to help transform all this into something that can be deployed efficiently and that's understood by most web browsers.

Module bundling

When writing clean, reusable code with ES2015/CommonJS modules, we need some way to load these modules (at least until browsers support ES2015 module loading natively). Including a bunch of script tags in your HTML isn't really a viable option, as it would quickly become unwieldy for any serious application, and all those separate HTTP requests would hurt performance.

We can include all the modules where we need them using the `import` statement from ES2015 (or `require`, for CommonJS) and use a module bundler to combine everything together into one or more files (bundles). It's this bundled file that we're going to upload to our server and include in our HTML. It will include all your imported modules and their necessary dependencies.

There are currently a couple of popular options for this, the most popular ones being [Webpack](#), [Browserify](#) and [Rollup.js](#). You can choose one or another depending on your needs.

Further Reading on Module Bundling

If you want to learn more about module bundling and how it fits into the bigger picture of app development, I recommend reading [Understanding JavaScript Modules: Bundling & Transpiling](#).

Transpilation

While [support for modern JavaScript is pretty good among newer browsers](#), your target audience may include legacy browsers and devices with partial or no

support.

In order to make our modern JavaScript work, we need to translate the code we write to its equivalent in an earlier version (usually ES5). The standard tool for this task is [Babel](#) — a compiler that translates your code into compatible code for most browsers. In this way, you don't have to wait for vendors to implement everything; you can just use all the modern JS features.

There are a couple of features that need more than a syntax translation. Babel includes a [Polyfill](#) that emulates some of the machinery required for some complex features such as promises.

Build systems & task runners

Module bundling and transpilation are just two of the build processes that we may need in our projects. Others include code minification (to reduce file sizes), tools for analysis, and perhaps tasks that don't have anything to do with JavaScript, like image optimization or CSS/HTML pre-processing.

The management of tasks can become a laborious thing to do, and we need a way to handle it in an automated way, being able to execute everything with simpler commands. The two most popular tools for this are [Grunt.js](#) and [Gulp.js](#), which provide a way to organize your tasks into groups in an ordered way.

For example, you can have a command like `gulp build` which may run a code linter, the transpilation process with Babel, and module bundling with Browserify. Instead of having to remember three commands and their associated arguments in order, we just execute one that will handle the whole process automatically.

Wherever you find yourself manually organizing processing steps for your project, think if it can be automatized with a task runner.

Further Reading on Gulp.js

Further reading: [An Introduction to Gulp.js](#).

Application Architecture

Web applications have different requirements from websites. For example, while page reloads may be acceptable for a blog, that's certainly not the case for an application like Google Docs. Your application should behave as closely as possible to a desktop one. Otherwise, the usability will be compromised.

Old-style web applications were usually done by sending multiple pages from a web server, and when a lot of dynamism was needed, content was loaded via Ajax by replacing chunks of HTML according to user actions. Although it was a big step forward to a more dynamic web, it certainly had its complications. Sending HTML fragments or even whole pages on each user action represented a waste of resources — especially of time, from the user's perspective. The usability still didn't match the responsiveness of desktop applications.

Looking to improve things, we created two new methods to build web applications — from the way we present them to the user, to the way we communicate between the client and the server. Although the amount of JavaScript required for an application also increased drastically, the result is now applications that behave very closely to native ones, without page reloading or extensive waiting periods each time we click a button.

Single Page Applications (SPAs)

The most common, high-level architecture for web applications is called [SPA](#), which stands for *Single Page Application*. SPAs are big blobs of JavaScript that contain everything the application needs to work properly. The UI is rendered entirely client-side, so no reloading is required. The only thing that changes is the data inside the application, which is usually handled with a remote API via [Ajax](#) or another asynchronous method of communication.

One downside to this approach is that the application takes longer to load for the first time. Once it has been loaded, however, transitions between views (pages) are generally a lot quicker, since it's only pure data being sent between client and server.

Universal / Isomorphic Applications

Although SPAs provide a great user experience, depending on your needs, they might not be the optimal solution — especially if you need quicker initial response times or optimal indexing by search engines.

There's a fairly recent approach to solving these problems, called [Isomorphic](#) (or Universal) JavaScript applications. In this type of architecture, most of the code can be executed both on the server and the client. You can choose what you want to render on the server for a faster initial page load, and after that, the client takes over the rendering while the user is interacting with the app. Because pages are initially rendered on the server, search engines can index them properly.

Deployment

With modern JavaScript applications, the code you write is not the same as the code that you deploy for production: you only deploy the result of your build process. The workflow to accomplish this can vary depending on the size of your project, the number of developers working on it, and sometimes the tools/libraries you're using.

For example, if you're working alone on a simple project, each time you're ready for deployment you can just run the build process and upload the resulting files to a web server. Keep in mind that you only need to upload the resulting files from the build process (transpilation, module bundling, minification, etc.), which can be just one `.js` file containing your entire application and dependencies.

You can have a directory structure like this:



You thus have all of your application files in a `src` directory, written in ES2015+, importing packages installed with npm and your own modules from a `lib` directory.

Then you can run Gulp, which will execute the instructions from a `gulpfile.js` to build your project — bundling all modules into one file (including the ones installed with npm), transpiling ES2015+ to ES5, minifying the resulted file, etc. Then you can configure it to output the result in a convenient `dist` directory.

Files That Don't Need Processing

If you have files that don't need any processing, you can just copy them from `src` to the `dist` directory. You can configure a task for that in your build system.

Now you can just upload the files from the `dist` directory to a web server, without having to worry about the rest of the files, which are only useful for development.

Team development

If you're working with other developers, it's likely you're also using a shared code repository, like GitHub, to store the project. In this case, you can run the build process right before making commits and store the result with the other files in the Git repository, to later be downloaded onto a production server.

However, storing built files in the repository is prone to errors if several developers are working together, and you might want to keep everything clean from build artifacts. Fortunately, there's a better way to deal with that problem: you can put a service like [Jenkins](#), [Travis CI](#), [CircleCI](#), etc. in the middle of the process, so it can automatically build your project after each commit is pushed to the repository. Developers only have to worry about pushing code changes without building the project first each time. The repository is also kept clean of automatically generated files, and at the end, you still have the built files available for deployment.

Conclusion

The transition from simple web pages to modern JavaScript applications can seem daunting if you've been away from web development in recent years, but I hope this article was useful as a starting point. I've linked to more in-depth articles on each topic where possible so you can explore further.

And remember that if at some point, after looking all the options available, everything seems overwhelming and messy, just keep in mind the [KISS principle](#), and use only what you think you need and not everything you have available. At the end of the day, solving problems is what matters, not using the latest of everything.

Chapter 2: Clean Code with ES6 Default Parameters & Property Shorthands

by Moritz Kröger

Creating a method also means writing an API — whether it's for yourself, another developer on your team, or other developers using your project. Depending on the size, complexity, and purpose of your function, you have to think of default settings and the API of your input/output.

[Default function parameters](#) and [property shorthands](#) are two handy features of ES6 that can help you write your API.

ES6 Default Parameters

Let's freshen up our knowledge quickly and take a look at the syntax again. Default parameters allow us to initialize functions with default values. A default is used when an argument is either omitted or `undefined` — meaning `null` is a valid value. A default parameter can be anything from a number to another function.

```
// Basic syntax
function multiply (a, b = 2) {
  return a * b;
}
multiply(5); // 10

// Default parameters are also available to later default parameters
function foo (num = 1, multi = multiply(num)) {
  return [num, multi];
}
foo(); // [1, 2]
foo(6); // [6, 12]
```

A real-world example

Let's take a basic function and demonstrate how default parameters can speed up your development and make the code better organized.

Our example method is called `createElement()`. It takes a few configuration arguments, and returns an HTML element. The API looks like this:

```
// We want a <p> element, with some text content and two classes att
// Returns <p class="very-special-text super-big">Such unique text</
createElement('p', {
  content: 'Such unique text',
  classNames: ['very-special-text', 'super-big']
});

// To make this method even more useful, it should always return a d
// element when any argument is left out or none are passed at all.
createElement(); // <div class="module-text default">Very default</d
```

The implementation of this won't have much logic, but can become quite large due to its default coverage.

```
// Without default parameters it looks quite bloated and unnecessary
function createElement (tag, config) {
  tag = tag || 'div';
  config = config || {};

  const element = document.createElement(tag);
  const content = config.content || 'Very default';
  const text = document.createTextNode(content);
  let classNames = config.classNames;

  if (classNames === undefined) {
    classNames = ['module-text', 'default'];
  }

  element.classList.add(...classNames);
  element.appendChild(text);

  return element;
}
```

So far, so good. What's happening here? We're doing the following:

1. setting default values for both our parameters `tag` and `config`, in case they aren't passed (*note that some linters [don't like parameter reassigning](#)*)
2. creating constants with the actual content (and default values)
3. checking if `classNames` is defined, and assigning a default array if not

4. creating and modifying the element before we return it.

Now let's take this function and optimize it to be cleaner, faster to write, and so that it's more obvious what its purpose is:

```
// Default all the things
function createElement (tag = 'div', {
  content = 'Very default',
  classNames = ['module-text', 'special']
} = {}) {
  const element = document.createElement(tag);
  const text = document.createTextNode(content);

  element.classList.add(...classNames);
  element.appendChild(text);

  return element;
}
```

We didn't touch the function's logic, but removed all default handling from the function body. The [function signature](#) now contains all defaults.

Let me further explain one part, which might be slightly confusing:

```
// What exactly happens here?
function createElement ({
  content = 'Very default',
  classNames = ['module-text', 'special']
} = {}) {
  // Function body
}
```

We not only declare a default object parameter, but also default object *properties*. This makes it more obvious what the default configuration is supposed to look like, rather than only declaring a default object (e.g. `config = {}`) and later setting default properties. It might take some additional time to get used to it, but in the end it improves your workflow.

Of course, we could still argue with larger configurations that it might create more overhead and it'd be simpler to just keep the default handling inside of the function body.

ES6 Property Shorthands

If a method accepts large configuration objects as an argument, your code can become quite large. It's common to prepare some variables and add them to said object. Property shorthands are [syntactic sugar](#) to make this step shorter and more readable:

```
const a = 'foo', b = 42, c = function () {};  
  
// Previously we would use these constants like this.  
const alphabet = {  
  a: a,  
  b: b,  
  c: c  
};  
  
// But with the new shorthand we can actually do this now,  
// which is equivalent to the above.  
const alphabet = { a, b, c };
```

Shorten Your API

Okay, back to another, more common example. The following function takes some data, mutates it and calls another method:

```
function updateSomething (data = {}) {  
  const target = data.target;  
  const veryLongProperty = data.veryLongProperty;  
  let willChange = data.willChange;  
  
  if (willChange === 'unwantedValue') {  
    willChange = 'wayBetter';  
  }  
  
  // Do more.  
  
  useDataSomewhereElse({  
    target: target,  
    property: veryLongProperty,  
    willChange: willChange,  
    // .. more  
  });  
}
```

It often happens that we name variables and object property names the same. Using the property shorthand, combined with [destructuring](#), we actually can shorten our code quite a bit:

```
function updateSomething (data = {}) {
  // Here we use destructuring to store the constants from the data
  const { target, veryLongProperty: property } = data;
  let { willChange } = data;

  if (willChange === 'unwantedValue') {
    willChange = 'wayBetter';
  }

  // Do more.

  useDataSomewhereElse({ target, property, willChange });
}
```

Again, this might take a while to get used to. In the end, it's one of those new features in JavaScript which helped me write code faster and work with cleaner function bodies.

But wait, there's more! Property shorthands can also be applied to method definitions inside an object:

```
// Instead of writing the function keyword everytime,
const module = {
  foo: 42,
  bar: function (value) {
    // do something
  }
};

// we can just omit it and have shorter declarations
const module = {
  foo: 42,
  bar (value) {
    // do something
  }
};
```

Conclusion

Default parameters and property shorthands are a great way to make your methods more organized, and in some cases even shorter. Overall, default function parameters helped me to focus more on the actual purpose of the method without the distraction of lots of default preparations and if statements.

Property shorthands are indeed more of a cosmetic feature, but I found myself

being more productive and spending less time writing all the variables, configuration objects, and function keywords.

Chapter 3: JavaScript Performance Optimization Tips: An Overview

by Ivan Čurić

In this post, there's lots of stuff to cover across a wide and wildly changing landscape. It's also a topic that covers everyone's favorite: The JS Framework of the Month™.

We'll try to stick to the "Tools, not rules" mantra and keep the JS buzzwords to a minimum. Since we won't be able to cover everything related to JS performance in a 2000 word article, make sure you read the references and do your own research afterwards.

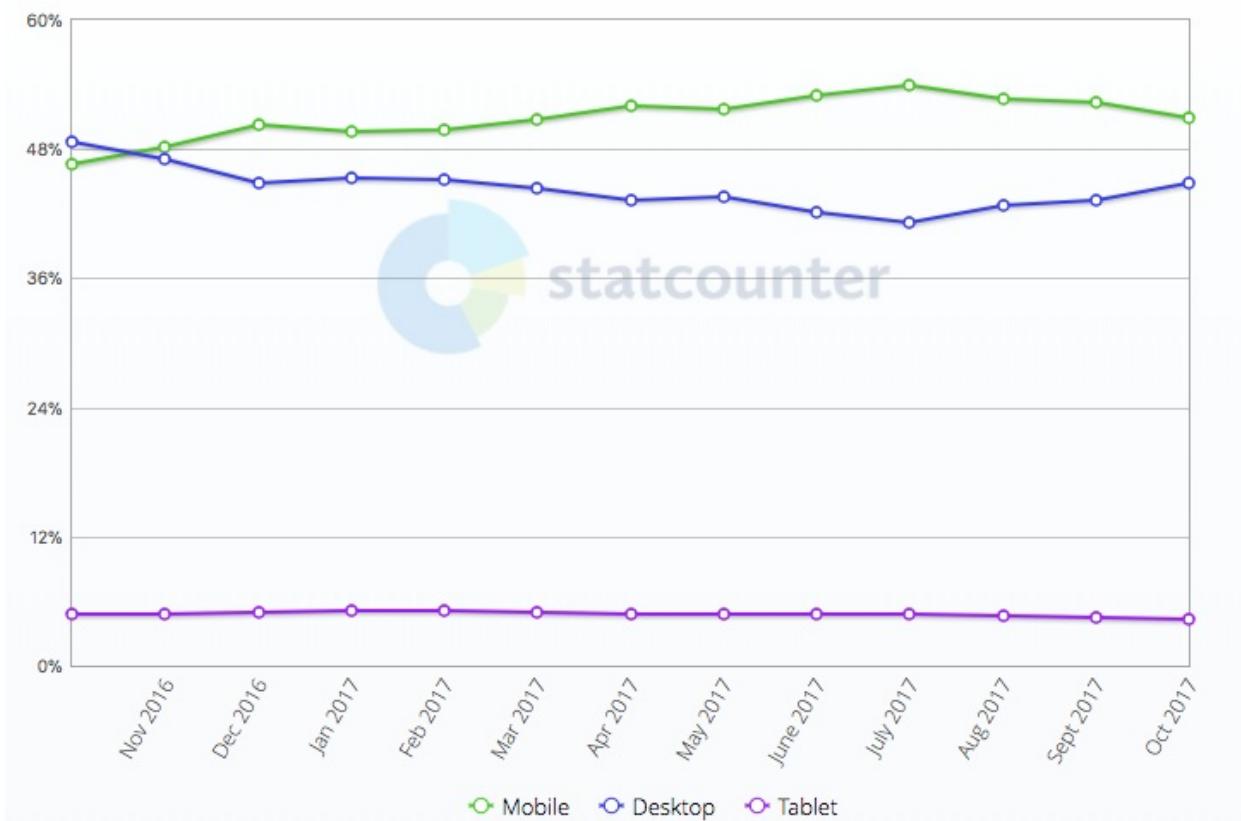
But before we dive into specifics, let's get a broader understanding of the issue by answering the following: what is considered as performant JavaScript, and how does it fit into the broader scope of web performance metrics?

Setting the Stage

First of all, let's get the following out of the way: if you're testing exclusively on your desktop device, you're excluding [more than 50%](#) of your users.

Desktop vs Mobile vs Tablet Market Share Worldwide

Oct 2016 - Oct 2017

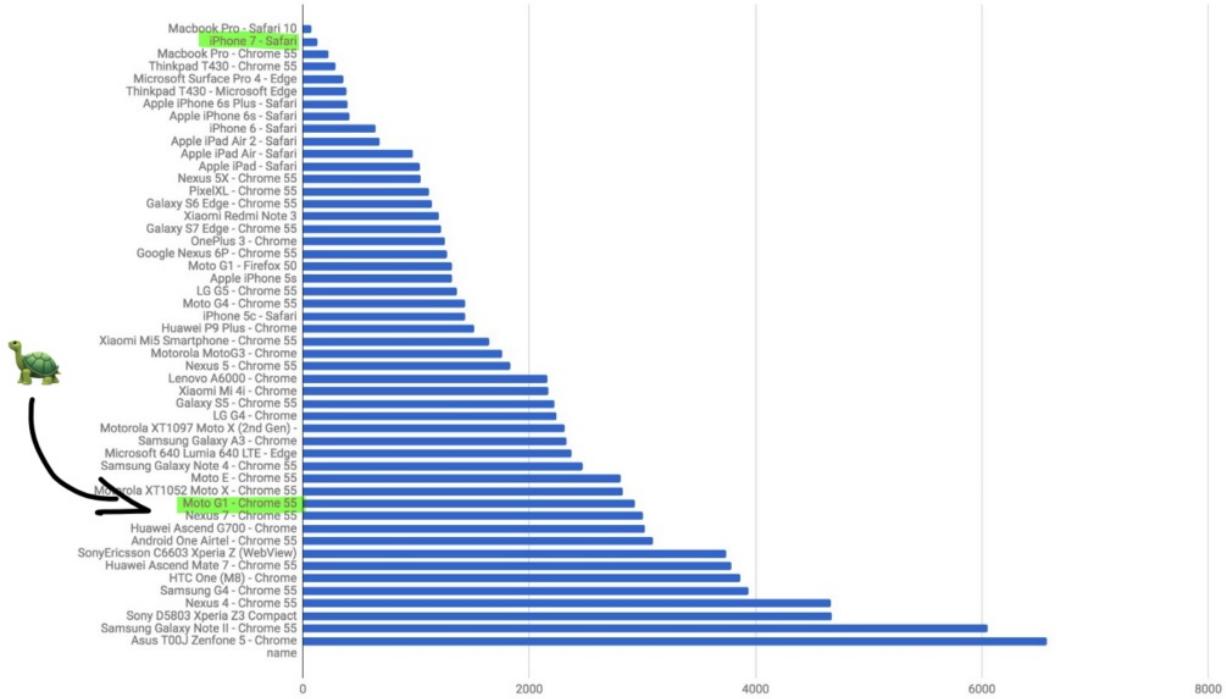


This trend will only continue to grow, as the emerging market's preferred gateway to the web is a sub-\$100 Android device. The era of the desktop as the main device to access the Internet is over, and the next billion internet users will visit your sites primarily through a mobile device.

Testing in Chrome DevTools' device mode isn't a valid substitute to testing on a real device. Using CPU and network throttling helps, but it's a fundamentally different beast. Test on real devices.

Even if you *are* testing on real mobile devices, you're probably doing so on your brand spanking new \$600 flagship phone. The thing is, that's not the device your users have. The median device is something along the lines of a Moto G1 --- a device with under 1GB of RAM, and a very weak CPU and GPU.

Let's see how it stacks up when [parsing an average JS bundle](#).



Addy Osmani: [Time spent in JS parse & eval for average JS](#).

Ouch. While this image only covers the parse and compile time of the JS (more on that later) and not general performance, it's strongly correlated and can be treated as an indicator of general JS performance.

To quote Bruce Lawson, “[it’s the World-Wide Web, not the Wealthy Western Web](#)”. So, your target for web performance is a device that’s **~25x slower** than your MacBook or iPhone. Let that sink in for a bit. But it gets worse. Let’s see what we’re actually aiming for.

What Exactly is Performant JS Code?

Now that we know what our target platform is, we can answer the next question: what *is* performant JS code?

While there’s no absolute classification of what defines performant code, we do have a user-centric performance model we can use as a reference: [The RAIL model](#).

- Respond: 100ms
- Animate: 8ms
- Idle work: 50ms chunks
- Load: 1000ms to interactive



Sam Saccone: [Planning for Performance: PRPL](#)

Respond

If your app responds to a user action in under 100ms, the user perceives the response as immediate. This applies to tappable elements, but not when scrolling or dragging.

Animate

On a 60Hz monitor, we want to target a constant 60 frames per second when animating and scrolling. That results in around 16ms per frame. Out of that 16ms budget, you realistically have 8–10ms to do all the work, the rest taken up by the browser internals and other variances.

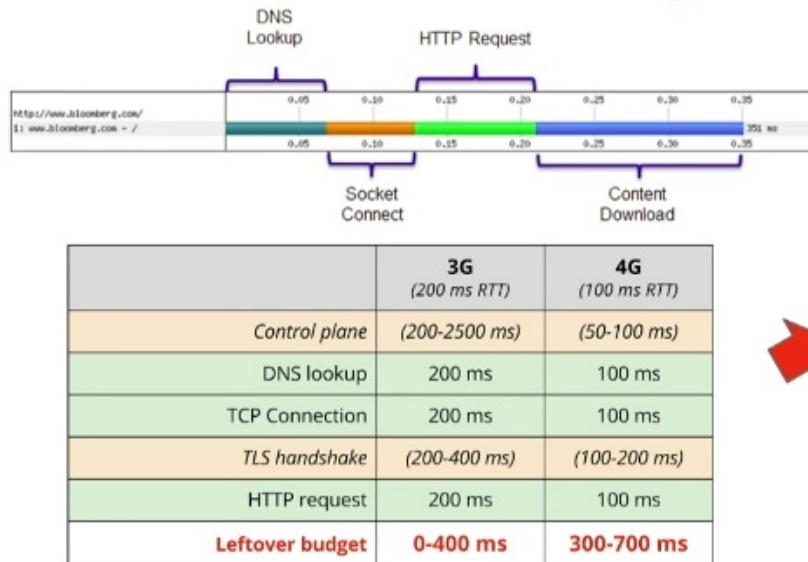
Idle work

If you have an expensive, continuously running task, make sure to slice it into smaller chunks to allow the main thread to react to user inputs. You shouldn't have a task that delays user input for more than 50ms.

Load

You should target a page load in under 1000ms. Anything over, and your users start getting twitchy. This is a pretty difficult goal to reach on mobile devices as it relates to the page being interactive, not just having it painted on screen and scrollable. In practice, it's even less:

The (short) life of our **1000 ms budget**



Network overhead of one HTTP request!



@igrigorik

[Fast By Default: Modern Loading Best Practices \(Chrome Dev Summit 2017\)](#)

In practice, aim for the 5s time-to-interactive mark. It's what Chrome uses in their [Lighthouse audit](#).

Now that we know the metrics, [let's have a look at some of the statistics](#):

- 53% of visits are abandoned if a mobile site takes more than three seconds to load
- 1 out of 2 people expect a page to load in less than 2 seconds
- 77% of mobile sites take longer than 10 seconds to load on 3G networks
- 19 seconds is the average load time for mobile sites on 3G networks.

[And a bit more, courtesy of Addy Osmani](#):

- apps became interactive in 8 seconds on desktop (using cable) and 16 seconds on mobile (Moto G4 over 3G)
- at the median, developers shipped 410KB of gzipped JS for their pages.

Feeling sufficiently frustrated? Good. Let's get to work and fix the web.

Context is Everything

You might have noticed that the main bottleneck is the time it takes to load up your website. Specifically, the JavaScript download, parse, compile and execution time. There's no way around it but to load less JavaScript and load smarter.

But what about the actual work that your code does aside from just booting up the website? There has to be some performance gains there, right?

Before you dive into optimizing your code, consider what you're building. Are you building a framework or a VDOM library? Does your code need to do thousands of operations per second? Are you doing a time-critical library for handling user input and/or animations? If not, you may want to shift your time and energy somewhere more impactful.

It's not that writing performant code doesn't matter, but it usually makes little to no impact in the grand scheme of things, especially when talking about microoptimizations. So, before you get into a Stack Overflow argument about `.map` vs `.forEach` vs `for` loops by comparing results from JSperf.com, make sure to see the forest and not just the trees. 50k ops/s might sound 50× better than 1k ops/s on paper, but it won't make a difference in most cases.

Parsing, Compiling and Executing

Fundamentally, the problem of most non-performant JS is not running the code itself, but all the steps that have to be taken *before* the code even starts executing.

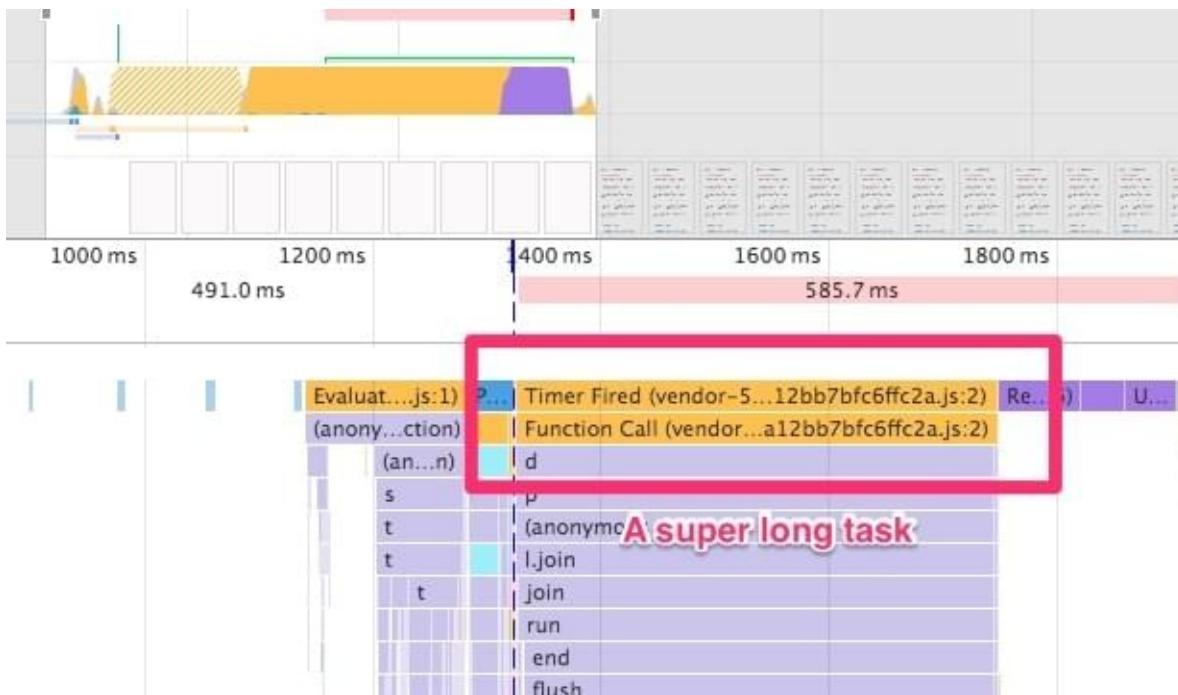
We're talking about levels of abstraction here. The CPU in your computer runs machine code. Most of the code you're running on your computer is in the compiled binary format. (I said *code* rather than *programs*, considering all the Electron apps these days.) Meaning, all the OS-level abstractions aside, it runs natively on your hardware, no prep-work needed.

JavaScript is not pre-compiled. It arrives (via a relatively slow network) as

readable code in your browser which is, for all intents and purposes, the "OS" for your JS program.

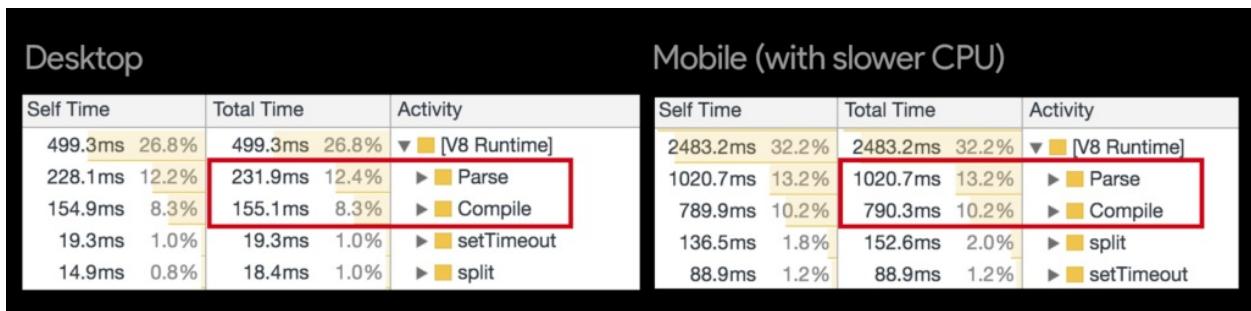
That code first needs to be parsed --- that is, read and turned into an computer-indexable structure that can be used for compiling. It then gets compiled into bytecode and finally machine code, before it can be executed by your device/browser.

Another *very* important thing to mention is that JavaScript is single-threaded, and runs on the browser's main thread. This means that only one process can run at a time. If your DevTools performance timeline is filled with yellow peaks, running your CPU at 100%, you'll have long/dropped frames, janky scrolling and all other kind of nasty stuff.



Paul Lewis: [When everything's important, nothing is!](#).

So there's all this work that needs to be done before your JS starts working. Parsing and compiling takes up to 50% of the total time of JS execution in Chrome's V8 engine.



Addy Osmani: [JavaScript Start-up Performance](#).

There are two things you should take away from this section:

1. While not necessarily linearly, JS parse time scales with the bundle size. The less JS you ship, the better.
2. Every JS framework you use (React, Vue, Angular, Preact...) is another level of abstraction (unless it's a precompiled one, like [Svelte](#)). Not only will it increase your bundle size, but also slow down your code since you're not talking directly to the browser.

There are ways to mitigate this, such as using service workers to do jobs in the background and on another thread, using asm.js to write code that's more easily compiled to machine instructions, but that's a whole 'nother topic.

What you can do, however, is avoid using JS animation frameworks for everything and [read up on what triggers paints and layouts](#). Use the libraries only when there's absolutely no way to implement the animation using regular CSS transitions and animations.

Even though they may be using CSS transitions, composited properties and `requestAnimationFrame()`, they're still running in JS, on the main thread. They're basically just hammering your DOM with inline styles every 16ms, since there's not much else they can do. You need to make sure all your JS will be done executing in under 8ms per frame in order to keep the animations smooth.

CSS animations and transitions, on the other hand, are running off the main thread --- on the GPU, if implemented performantly, without causing relayouts/reflows.

Considering that most animations are running either during loading or user

interaction, this can give your web apps the much-needed room to breathe.

The [Web Animations API](#) is an upcoming feature set that will allow you to do performant JS animations off the main thread, but for now, stick to CSS transitions and techniques like [FLIP](#).

Bundle Sizes are Everything

Today it's all about bundles. Gone are the times of Bower and dozens of `<script>` tags before the closing `</body>` tag.

Now it's all about `npm install`-ing whatever shiny new toy you find on NPM, bundling them together with Webpack in a huge single 1MB JS file and hammering your users' browser to a crawl while capping off their data plans.

Try shipping less JS. You might not need [the entire Lodash library](#) for your project. Do you absolutely *need* to use a JS framework? If yes, have you considered using something other than React, such as [Preact](#) or [HyperHTML](#), which are less than 1/20 the size of React? Do you need [TweenMax](#) for that scroll-to-top animation? The convenience of npm and isolated components in frameworks comes with a downside: the first response of developers to a problem has become to throw more JS at it. When all you have is a hammer, everything looks like a nail.

When you're done pruning the weeds and shipping less JS, try shipping it *smarter*. Ship what you need, when you need it.

Webpack 3 has *amazing* features called [code splitting](#) and [dynamic imports](#). Instead of bundling all your JS modules into a monolithic `app.js` bundle, it can automatically split the code using the `import()` syntax and load it asynchronously.

You don't need to use frameworks, components and client-side routing to gain the benefit of it, either. Let's say you have a complex piece of code that powers your `.mega-widget`, which can be on any number of pages. You can simply write the following in your main JS file:

```
if (document.querySelector('.mega-widget')) {  
    import('./mega-widget');  
}
```

If your app finds the widget on the page, it will dynamically load the required supporting code. Otherwise, all's good.

Also, Webpack needs its own runtime to work, and it injects it into all the .js files it generates. If you use the `commonChunks` plugin, you can use the following to [extract the runtime into its own chunk](#):

```
new webpack.optimize.CommonsChunkPlugin({  
    name: 'runtime',  
})
```

It will strip out the runtime from all your other chunks into its own file, in this case named `runtime.js`. Just make sure to load it before your main JS bundle. For example:

```
<script src="runtime.js">  
<script src="main-bundle.js">
```

Then there's the topic of transpiled code and polyfills. If you're writing modern (ES6+) JavaScript, you're probably using Babel to transpile it into ES5 compatible code. Transpiling not only increases file size due to all the verbosity, but also complexity, and it often has [performance regressions](#) compared to native ES6+ code.

Along with that, you're probably using the `babel-polyfill` package and `whatwg-fetch` to patch up missing features in older browsers. Then, if you're writing code using `async/await`, you also transpile it using generators needed to include the `regenerator-runtime`...

The point is, you add almost 100 kilobytes to your JS bundle, which has not only a huge file size, but also a huge parsing and executing cost, in order to support older browsers.

There's no point in punishing people who are using modern browsers, though. An approach I use, and which Philip Walton covered in [this article](#), is to create two separate bundles and load them conditionally. Babel makes this easy with `babel-preset-env`. For instance, you have one bundle for supporting IE 11, and the other without polyfills for the latest versions of modern browsers.

A dirty but efficient way is to place the following in an inline script:

```
(function() {
  try {
    new Function('async () => {}')();
  } catch (error) {
    // create script tag pointing to legacy-bundle.js;
    return;
  }
  // create script tag pointing to modern-bundle.js;;
})());
```

If the browser isn't able to evaluate an async function, we assume that it's an old browser and just ship the polyfilled bundle. Otherwise, the user gets the neat and modern variant.

Conclusion

What we would like you to gain from this article is that JS is expensive and should be used sparingly.

Make sure you test your website's performance on low-end devices, under real network conditions. Your site should load fast and be interactive as soon as possible. This means shipping less JS, and shipping faster by any means necessary. Your code should always be minified, split into smaller, manageable bundles and loaded asynchronously whenever possible. On the server side, make sure it has HTTP/2 enabled for faster parallel transfers and gzip/Brotli compression to drastically reduce the transfer sizes of your JS.

Chapter 4: JavaScript Design Patterns: The Singleton

by Samier Saeed

In this chapter, we'll dig into the best way to implement a singleton in JavaScript, looking at how this has evolved with the rise of ES6.

Among languages used in widespread production, JavaScript is by far the most quickly evolving, looking less like its earliest iterations and more like Python, with every new spec put forth by ECMA International. While the changes have their fair share of detractors, the new JavaScript does succeed in making code easier to read and reason about, easier to write in a way that adheres to software engineering best practices (particularly the concepts of modularity and SOLID principles), and easier to assemble into canonical software design patterns.

Explaining ES6

ES6 (aka ES2015) was the first major update to the language since ES5 was standardized in 2009. [Almost all modern browsers support ES6](#). However, if you need to accommodate older browsers, ES6 code can easily be transpiled into ES5 using a tool such as Babel. ES6 gives JavaScript a ton of new features, including [a superior syntax for classes](#), and [new keywords for variable declarations](#). You can learn more about it by perusing [SitePoint articles on the subject](#).

What Is a Singleton

In case you're unfamiliar with the [singleton pattern](#), it is, at its core, a design pattern that restricts the instantiation of a class to one object. Usually, the goal is to manage global application state. Some examples I've seen or written myself include using a singleton as the source of config settings for a web app, on the client side for anything initiated with an API key (you usually don't want to risk sending multiple analytics tracking calls, for example), and to store data in

memory in a client-side web application (e.g. stores in Flux).

A singleton should be immutable by the consuming code, and there should be no danger of instantiating more than one of them.

Are Singletons Bad?

There are scenarios when singletons might be bad, and arguments that they are, in fact, always bad. For that discussion, you can check out [this helpful article](#) on the subject.

The Old Way of Creating a Singleton in JavaScript

The old way of writing a singleton in JavaScript involves leveraging [closures and immediately invoked function expressions](#). Here's how we might write a (very simple) store for a hypothetical Flux implementation the old way:

```
var UserStore = (function(){
  var _data = [];

  function add(item){
    _data.push(item);
  }

  function get(id){
    return _data.find((d) => {
      return d.id === id;
    });
  }

  return {
    add: add,
    get: get
  };
}());
```

When that code is interpreted, `UserStore` will be set to the result of that immediately invoked function — an object that exposes two functions, but that does not grant direct access to the collection of data.

However, this code is more verbose than it needs to be, and also doesn't give us

the immutability we desire when making use of singletons. Code executed later could modify either one of the exposed functions, or even redefine `UserStore` altogether. Moreover, the modifying/offending code could be anywhere! If we got bugs as a result of unexpected modification of `usersStore`, tracking them down in a larger project could prove very frustrating.

There are more advanced moves you could pull to mitigate some of these downsides, as specified in [this article](#) by Ben Cherry. (His goal is to create modules, which just happen to be singletons, but the pattern is the same.) But those add unneeded complexity to the code, while still failing to get us exactly what we want.

The New Way(s)

By leveraging ES6 features, mainly modules and the new `const` variable declaration, we can write singletons in ways that are not only more concise, but which better meet our requirements.

Let's start with the most basic implementation. Here's (a cleaner and more powerful) modern interpretation of the above example:

```
const _data = [];

const UserStore = {
  add: item => _data.push(item),
  get: id => _data.find(d => d.id === id)
}

Object.freeze(UserStore);
export default UserStore;
```

As you can see, this way offers an improvement in readability. But where it really shines is in the constraint imposed upon code that consumes our little singleton module here: the consuming code cannot reassign `UserStore` because of the `const` keyword. And as a result of our use of [Object.freeze](#), its methods cannot be changed, nor can new methods or properties be added to it. Furthermore, because we're taking advantage of ES6 modules, we know exactly where `UserStore` is used.

Now, here we've made `UserStore` an object literal. Most of the time, going with an object literal is the most readable and concise option. However, there are

times when you might want to exploit the benefits of going with a traditional class. For example, stores in Flux will all have a lot of the same base functionality. Leveraging traditional object-oriented inheritance is one way to get that repetitive functionality while keeping your code DRY.

Here's how the implementation would look if we wanted to utilize ES6 classes:

```
class UserStore {
  constructor(){
    this._data = [];
  }

  add(item){
    this._data.push(item);
  }

  get(id){
    return this._data.find(d => d.id === id);
  }
}

const instance = new UserStore();
Object.freeze(instance);

export default instance;
```

This way is slightly more verbose than using an object literal, and our example is so simple that we don't really see any benefits from using a class (though it will come in handy in the final example).

One benefit to the class route that might not be obvious is that, if this is your front-end code, and your back end is written in C# or Java, you can employ a lot of the same design patterns in your client-side application as you do on the back end, and increase your team's efficiency (if you're small and people are working full-stack). Sounds soft and hard to measure, but I've experienced it firsthand working on a C# application with a React front end, and the benefit is real.

It should be noted that, technically, the immutability and non-overridability of the singleton using both of these patterns can be subverted by the motivated provocateur. An object literal can be copied, even if it itself is `const`, by using [Object.assign](#). And when we export an instance of a class, though we aren't directly exposing the class itself to the consuming code, the constructor of any instance is available in JavaScript and can be invoked to create new instances.

Obviously, though, that all takes at least a little bit of effort, and hopefully your fellow devs aren't so insistent on violating the singleton pattern.

But let's say you wanted to be extra sure that nobody messed with the singleness of your singleton, and you also wanted it to match the implementation of singletons in the object-oriented world even more closely. Here's something you could do:

```
class UserStore {  
  constructor(){  
    if(! UserStore.instance){  
      this._data = [];  
      UserStore.instance = this;  
    }  
  
    return UserStore.instance;  
  }  
  
  //rest is the same code as preceding example  
}  
  
const instance = new UserStore();  
Object.freeze(instance);  
  
export default instance;
```

By adding the extra step of holding a reference to the instance, we can check whether or not we've already instantiated a `UserStore`, and if we have, we won't create a new one. As you can see, this also makes good use of the fact that we've made `UserStore` a class.

Conclusion

There are no doubt plenty of developers who have been using the old singleton/module pattern in JavaScript for a number of years, and who find it works quite well for them. Nevertheless, because finding better ways to do things is so central to the ethos of being a developer, hopefully we see cleaner and easier-to-reason-about patterns like this one gaining more and more traction. Especially once it becomes easier and more commonplace to utilize ES6+ features.

Chapter 5: JavaScript Object Creation: Patterns and Best Practices

by Jeff Mott

In this chapter, I'm going to take you on a tour of the various styles of JavaScript object creation, and how each builds on the others in incremental steps.

JavaScript has a multitude of styles for creating objects, and newcomers and veterans alike can feel overwhelmed by the choices and unsure which they should use. But despite the variety and how different the syntax for each may look, they're more similar than you probably realize.

Object Literals

The first stop on our tour is the absolute simplest method of JavaScript object creation — the object literal. JavaScript touts that objects can be created “ex nihilo”, out of nothing — no class, no template, no prototype — just *poof!*, an object with methods and data:

```
var o = {  
  x: 42,  
  y: 3.14,  
  f: function() {},  
  g: function() {}  
};
```

But there's a drawback. If we need to create the same type of object in other places, then we'll end up copy-pasting the object's methods, data, and initialization. We need a way to create not just the one object, but a family of objects.

Factory Functions

The next stop on our JavaScript object creation tour is the factory function. This is the absolute simplest way to create a family of objects that share the same structure, interface, and implementation. Rather than creating an object literal directly, instead we return an object literal from a function. This way, if we need to create the same type of object multiple times or in multiple places, we only need to invoke a function:

```
function thing() {
  return {
    x: 42,
    y: 3.14,
    f: function() {},
    g: function() {}
  };
}

var o = thing();
```

But there's a drawback. This approach of JavaScript object creation can cause memory bloat, because each object contains its own unique copy of each function. Ideally, we want every object to *share* just one copy of its functions.

Prototype Chains

JavaScript gives us a built-in mechanism to share data across objects, called the **prototype chain**. When we access a property on an object, it can fulfill that request by delegating to some other object. We can use that and change our factory function so that each object it creates contains only the data unique to that particular object, and delegate all other property requests to a single, shared object:

```
var thingPrototype = {
  f: function() {},
  g: function() {}
};

function thing() {
  var o = Object.create(thingPrototype);

  o.x = 42;
  o.y = 3.14;

  return o;
}
```

```
var o = thing();
```

In fact, this is such a common pattern that the language has built-in support for it. We don't need to create our own shared object (the prototype object). Instead, a prototype object is created for us automatically alongside every function, and we can put our shared data there:

```
thing.prototype.f = function() {};
thing.prototype.g = function() {};

function thing() {
  var o = Object.create(thing.prototype);

  o.x = 42;
  o.y = 3.14;

  return o;
}

var o = thing();
```

But there's a drawback. This is going to result in some repetition. The first and last lines of the `thing` function are going to be repeated almost verbatim in every such delegating-to-prototype factory function.

ES5 Classes

We can isolate the repetitive lines by moving them into their own function. This function would create an object that delegates to some other arbitrary function's prototype, then invoke that function with the newly created object as an argument, and finally return the object:

```
function create(fn) {
  var o = Object.create(fn.prototype);

  fn.call(o);

  return o;
}

// ...

Thing.prototype.f = function() {};
Thing.prototype.g = function() {};
```

```
function Thing() {
  this.x = 42;
  this.y = 3.14;
}

var o = create(Thing);
```

In fact, this too is such a common pattern that the language has some built-in support for it. The `create` function we defined is actually a rudimentary version of the `new` keyword, and we can drop-in replace `create` with `new`:

```
Thing.prototype.f = function() {};
Thing.prototype.g = function() {};

function Thing() {
  this.x = 42;
  this.y = 3.14;
}

var o = new Thing();
```

We've now arrived at what we commonly call "ES5 classes". They're object creation functions that delegate shared data to a prototype object and rely on the `new` keyword to handle repetitive logic.

But there's a drawback. It's verbose and ugly, and implementing inheritance is even more verbose and ugly.

ES6 Classes

A relatively recent addition to JavaScript is ES6 classes, which offer a significantly cleaner syntax for doing the same thing:

```
class Thing {
  constructor() {
    this.x = 42;
    this.y = 3.14;
  }

  f() {}
  g() {}
}

const o = new Thing();
```

Comparison

Over the years, we JavaScripters have had an on-and-off relationship with the prototype chain, and today the two most common styles you’re likely to encounter are the class syntax, which relies heavily on the prototype chain, and the factory function syntax, which typically doesn’t rely on the prototype chain at all. The two styles differ — but only slightly — in performance and features.

Performance

JavaScript engines are so heavily optimized today that it’s nearly impossible to look at our code and reason about what will be faster. Measurement is crucial. Yet sometimes even measurement can fail us. Typically, an updated JavaScript engine is released every six weeks, sometimes with significant changes in performance, and any measurements we had previously taken, and any decisions we made based on those measurements, go right out the window. So, my rule of thumb has been to favor the most official and most widely used syntax, under the presumption that it will receive the most scrutiny and be the most performant *most of the time*. Right now, that’s the class syntax, and as I write this, the class syntax is roughly 3x faster than a factory function returning a literal.

Features

What few feature differences there were between classes and factory functions evaporated with ES6. Today, both factory functions and classes can enforce truly private data—factory functions with closures and classes with weak maps. Both can achieve multiple-inheritance factory functions by mixing other properties into their own object, and classes also by mixing other properties into their prototype, or with class factories, or with proxies. Both factory functions and classes can return any arbitrary object if need be. And both offer a simple syntax.

Conclusion

All things considered, my preference for JavaScript object creation is to use the class syntax. It’s standard, it’s simple and clean, it’s fast, and it provides every feature that once upon a time only factories could deliver.

Chapter 6: Best Practices for Using Modern JavaScript Syntax

by M. David Green

Modern JavaScript is evolving quickly to meet the changing needs of new frameworks and environments. Understanding how to take advantage of those changes can save you time, improve your skill set, and mark the difference between good code and great code.

Knowing what modern JavaScript is trying to do can help you decide when to use the new syntax to your best advantage, and when it still makes sense to use traditional techniques.

Something Solid to Cling To

I don't know anybody who isn't confused at the state of JavaScript these days, whether you're new to JavaScript, or you've been coding with it for a while. So many new frameworks, so many changes to the language, and so many contexts to consider. It's a wonder that anybody gets any work done, with all of the new things that we have to learn every month.

I believe that the secret to success with any programming language, no matter how complex the application, is getting back to the basics. If you want to understand Rails, start by working on your Ruby skills, and if you want to use [immutables](#) and [unidirectional data flow](#) in [isomorphic React](#) with [webpack](#) (or whatever the cool nerds are doing these days) start by knowing your core JavaScript.

Understanding how the language itself works is much more practical than familiarizing yourself with the latest frameworks and environments. Those change faster than the weather. And with JavaScript, we have a long history of thoughtful information online about how JavaScript was created and how to use it effectively.

The problem is that some of the new techniques that have come around with the latest versions of JavaScript make some of the old rules obsolete. But not all of them! Sometimes a new syntax may replace a clunkier one to accomplish the same task. Other times the new approach may seem like a simpler drop-in replacement for the way we used to do things, but there are subtle differences, and it's important to be aware of what those are.

A Spoonful of Syntactic Sugar

A lot of the changes in JavaScript in recent years have been described as syntactic sugar for existing syntax. In many cases, the syntactic sugar can help the medicine go down for Java programmers learning how to work with JavaScript, or for the rest of us we just want a cleaner, simpler way to accomplish something we already knew how to do. Other changes seem to introduce magical new capabilities.

But if you try to use modern syntax to recreate a familiar old technique, or stick it in without understanding how it actually behaves, you run the risk of:

- having to debug code that worked perfectly before
- introducing subtle mistakes that may catch you at runtime
- creating code that fails silently when you least expect it.

In fact, several of the changes that appear to be drop-in replacements for existing techniques actually behave differently from the code that they supposedly replace. In many cases, it can make more sense to use the original, older style to accomplish what you're trying to do. Recognizing when that's happening, and knowing how to make the choice, is critical to writing effective modern JavaScript.

When Your `const` Isn't Consistent

Modern JavaScript introduced two new keywords, `let` and `const`, which effectively replace the need for `var` when declaring variables in most cases. But they don't behave exactly the same way that `var` does.

In traditional JavaScript, it was always a clean coding practice to declare your variables with the `var` keyword before using them. Failure to do that meant that

the variables you declared could be accessed in the global scope by any scripts that happened to run in the same context. And because traditional JavaScript was frequently run on webpages where multiple scripts might be loaded simultaneously, that meant that it was possible for variables declared in one script to leak into another.

The cleanest drop-in replacement for `var` in modern JavaScript is `let`. But `let` has a few idiosyncrasies that distinguish it from `var`. Variable declarations with `var` were always hoisted to the top of their containing [scope](#) by default, regardless of where they were placed inside of that scope. That meant that even a deeply nested variable could be considered declared and available right from the beginning of its containing scope. The same is not true of `let` or `const`.

```
console.log(usingVar); // undefined
var usingVar = "defined";
console.log(usingVar); // "defined"

console.log(usingLet); // error
let usingLet = "defined"; // never gets executed
console.log(usingLet); // never gets executed
```

When you declare a variable using `let` or `const`, the scope for that variable is limited to the local block where it's declared. A block in JavaScript is distinguished by a set of curly braces `{}`, such as the body of a function or the executable code within a loop.

This is a great convenience for block-scoped uses of variables such as iterators and loops. Previously, variables declared within loops would be available to the containing scope, leading to potential confusion when multiple counters might use the same variable name. However `let` can catch you by surprise if you expect your variable declared somewhere inside of one block of your script to be available elsewhere.

```
for (var count = 0; count < 5; count++) {
  console.log(count);
} // outputs the numbers 0 - 4 to the console
console.log(count); // 5

for (let otherCount = 0; otherCount < 5; otherCount++) {
  console.log(otherCount);
} // outputs the numbers 0 - 4 to the console
console.log(otherCount); // error, otherCount is undefined
```

The other alternative declaration is `const`, which is supposed to represent a constant. But it's not completely constant.

A `const` can't be declared without a value, unlike a `var` or `let` variable.

```
var x; // valid
let y; // valid
const z; // error
```

A `const` will also throw an error if you try to set it to a new value after it's been declared:

```
const z = 3; // valid
z = 4; // error
```

But if you expect your `const` to be immutable in all cases, you may be in for a surprise when an object or an array declared as a `const` lets you alter its content.

```
const z = []; // valid
z.push(1); // valid, and z is now [1]
z = [2] // error
```

For this reason, I remain skeptical when people recommend using `const` constantly in place of `var` for all variable declarations, even when you have every intention of never altering them after they've been declared.

While it's a good practice to treat your variables as immutable, JavaScript won't enforce that for the contents of a reference variables like arrays and objects declared with `const` without some help from external scripts. So the `const` keyword may make casual readers and newcomers to JavaScript expect more protection than it actually provides.

I'm inclined to use `const` for simple number or string variables I want to initialize and never alter, or for named functions and classes that I expect to define once and then leave closed for modification. Otherwise, I use `let` for most variable declarations — especially those I want to be bounded by the scope in which they were defined. I haven't found the need to use `var` lately, but if I wanted a declaration to break scope and get hoisted to the top of my script, that's how I would do it.

Limiting the Scope of the Function

Traditional functions, defined using the `function` keyword, can be called to execute a series of statements defined within a block on any parameters passed in, and optionally return a value:

```
function doSomething(param) {  
    return(`Did it: ${param}`);  
}  
console.log(doSomething("Hello")); // "Did it: Hello"
```

They can also be used with the `new` keyword to construct objects with prototypal inheritance, and that definition can be placed anywhere in the scope where they might be called:

```
function Animal(name) {  
    this.name = name;  
}  
let cat = new Animal("Fluffy");  
console.log(`My cat's name is ${cat.name}.`); // "My cat's name is Fl
```

Functions used in either of these ways can be defined before or after they're called. It doesn't matter to JavaScript.

```
console.log(doSomething("Hello")); // "Did it: Hello"  
  
let cat = new Animal("Fluffy");  
console.log(`My cat's name is ${cat.name}.`); // "My cat's name is Fl  
  
function doSomething(param) {  
    return(`Did it: ${param}`);  
}  
function Animal(name) {  
    this.name = name;  
}
```

A traditional function also creates its own context, defining a value for `this` that exists only within the scope of the statement body. Any statements or sub-functions defined within it are executing, and optionally allowing us to bind a value for `this` when calling the function.

That's a lot for the keyword to do, and it's usually more than a programmer needs in any one place. So modern JavaScript split out the behavior of the traditional function into arrow functions and classes.

Getting to Class on Time

One part of the traditional function has been taken over by the `class` keyword. This allows programmers to choose whether they would prefer to follow a more functional programming paradigm with callable arrow functions, or use a more object-oriented approach with classes to substitute for the prototypal inheritance of traditional functions.

Classes in JavaScript look and act a lot like simple classes in other object-oriented languages, and may be an easy stepping stone for Java and C++ developers looking to expand into JavaScript as JavaScript expands out to the server.

One difference between functions and classes when doing object-oriented programming in JavaScript is that classes in JavaScript require forward declaration, the way they do in C++ (although not in Java). That is, a `class` needs to be declared in the script before it is instantiated with a `new` keyword. Prototypal inheritance using the `function` keyword works in JavaScript even if it's defined later in the script, since a `function` declaration is automatically hoisted to the top, unlike a `class`.

```
// Using a function to declare and instantiate an object (hoisted)
let aProto = new Proto("Myra");
aProto.greet(); // "Hi Myra"

function Proto(name) {
  this.name = name;
  this.greet = function() {
    console.log(`Hi ${this.name}`);
  };
}

// Using a class to declare and instantiate an object (not hoisted)
class Classy {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hi ${this.name}`);
  }
}

let aClassy = new Classy("Sonja");
aClassy.greet(); // "Hi Sonja"
```

Pointed Differences with Arrow Functions

The other aspect of traditional functions can now be accessed using arrow functions, a new syntax that allows you to write a callable function more concisely, to fit more neatly inside a callback. In fact, the simplest syntax for an arrow function is a single line that leaves off the curly braces entirely, and automatically returns the result of the statement executed:

```
const traditional = function(data) {
  return (`${data} from a traditional function`);
}
const arrow = data => `${data} from an arrow function`;

console.log(traditional("Hi")); // "Hi from a traditional function"
console.log(arrow("Hi")); // "Hi from an arrow function"
```

Arrow functions encapsulate several qualities that can make calling them more convenient, and leave out other behavior that isn't as useful when calling a function. They are not drop-in replacements for the more versatile traditional function keyword.

For example, an arrow function inherits both `this` and arguments from the contexts in which it's called. That's great for situations like event handling or `setTimeout` when a programmer frequently wants the behavior being called to apply to the context in which it was requested. Traditional functions have forced programmers to write convoluted code that binds a function to an existing `this` by using `.bind(this)`. None of that is necessary with arrow functions.

```
class GreeterTraditional {
  constructor() {
    this.name = "Joe";
  }
  greet() {
    setTimeout(function () {
      console.log(`Hello ${this.name}`);
    }, 1000); // inner function has its own this with no name
  }
}
let greeterTraditional = new GreeterTraditional();
greeterTraditional.greet(); // "Hello "

class GreeterBound {
  constructor() {
    this.name = "Steven";
  }
  greet() {
    setTimeout(function () {
```

```
        console.log(`Hello ${this.name}`);
    }.bind(this), 1000); // passing this from the outside context
}
}
let greeterBound = new GreeterBound(); // "Hello Steven"
greeterBound.greet();

class GreeterArrow {
  constructor() {
    this.name = "Ravi";
  }
  greet() {
    setTimeout(() => {
      console.log(`Hello ${this.name}`);
    }, 1000); // arrow function inherits this by default
  }
}
let greeterArrow = new GreeterArrow();
greeterArrow.greet(); // "Hello Ravi"
```

Understand What You're Getting

It's not all just syntactic sugar. A lot of the new changes in JavaScript have been introduced because new functionality was needed. But that doesn't mean that the old reasons for JavaScript's traditional syntax have gone away. Often it makes sense to continue using the traditional JavaScript syntax, and sometimes using the new syntax can make your code much faster to write and easier to understand.

Check out those online tutorials you're following. If the writer is using `var` to initialize all of the variables, ignoring classes in favor of prototypal inheritance, or relying on `function` statements in callbacks, you can expect the rest of the syntax to be based on older, traditional JavaScript. And that's fine. There's still a lot that we can learn and apply today from the traditional ways the JavaScript has always been taught and used. But if you see `let` and `const` in the initializations, arrow functions in callbacks, and classes as the basis for object-oriented patterns, you'll probably also see other modern JavaScript code in the examples.

The best practice in modern JavaScript is paying attention to what the language is actually doing. Depending on what you're used to, it may not always be obvious. But think about what the code you're writing is trying to accomplish, where you're going to need to deploy it, and who will be modifying it next. Then

decide for yourself what the best approach would be.

Chapter 7: Flow Control in Modern JS: Callbacks to Promises to Async/Await

by Craig Buckler

JavaScript is regularly claimed to be *asynchronous*. What does that mean? How does it affect development? How has the approach changed in recent years?

Consider the following code:

```
result1 = doSomething1();
result2 = doSomething2(result1);
```

Most languages process each line *synchronously*. The first line runs and returns a result. The second line runs once the first has finished *regardless of how long it takes*.

Single-thread Processing

JavaScript runs on a single processing thread. When executing in a browser tab, everything else stops. This is necessary because changes to the page DOM can't occur on parallel threads; it would be dangerous to have one thread redirecting to a different URL while another attempts to append child nodes.

This is rarely evident to the user, because processing occurs quickly in small chunks. For example, JavaScript detects a button click, runs a calculation, and updates the DOM. Once complete, the browser is free to process the next item on the queue.

Other Languages

Other languages such as PHP also use a single thread but may be managed by

a multi-threaded server such as Apache. Two requests to the same PHP page at the same time can initiate two threads running isolated instances of the PHP runtime.

Going Asynchronous with Callbacks

Single threads raise a problem. What happens when JavaScript calls a “slow” process such as an Ajax request in the browser or a database operation on the server? That operation could take several seconds — *even minutes*. A browser would become locked while it waited for a response. On the server, a Node.js application would not be able to process further user requests.

The solution is asynchronous processing. Rather than wait for completion, a process is told to call another function when the result is ready. This is known as a *callback*, and it’s passed as an argument to any asynchronous function. For example:

```
doSomethingAsync(callback1);
console.log('finished');

// call when doSomethingAsync completes
function callback1(error) {
  if (!error) console.log('doSomethingAsync complete');
}
```

`doSomethingAsync()` accepts a callback function as a parameter (only a reference to that function is passed so there’s little overhead). It doesn’t matter how long `doSomethingAsync()` takes; all we know is that `callback1()` will be executed at some point in the future. The console will show:

```
finished
doSomethingAsync complete
```

Callback Hell

Often, a callback is only ever called by one asynchronous function. It’s therefore possible to use concise, anonymous inline functions:

```
doSomethingAsync(error => {
  if (!error) console.log('doSomethingAsync complete');
});
```

A series of two or more asynchronous calls can be completed in series by nesting callback functions. For example:

```
async1((err, res) => {
  if (!err) async2(res, (err, res) => {
    if (!err) async3(res, (err, res) => {
      console.log('async1, async2, async3 complete.');
    });
  });
});
```

Unfortunately, this introduces **callback hell** — a notorious concept that even [has its own web page](#)! The code is difficult to read, and will become worse when error-handling logic is added.

Callback hell is relatively rare in client-side coding. It can go two or three levels deep if you're making an Ajax call, updating the DOM and waiting for an animation to complete, but it normally remains manageable.

The situation is different on OS or server processes. A Node.js API call could receive file uploads, update multiple database tables, write to logs, and make further API calls before a response can be sent.

Promises

[ES2015 \(ES6\) introduced Promises](#). Callbacks are still used below the surface, but Promises provide a clearer syntax that *chains* asynchronous commands so they run in series (more about that in the [next section](#)).

To enable Promise-based execution, asynchronous callback-based functions must be changed so they immediately return a Promise object. That object *promises* to run one of two functions (passed as arguments) at some point in the future:

- **resolve**: a callback function run when processing successfully completes, and
- **reject**: an optional callback function run when a failure occurs.

In the example below, a database API provides a `connect()` method which accepts a callback function. The outer `asyncDBconnect()` function immediately returns a new Promise and runs either `resolve()` or `reject()` once a connection

is established or fails:

```
const db = require('database');

// connect to database
function asyncDBconnect(param) {

    return new Promise((resolve, reject) => {

        db.connect(param, (err, connection) => {
            if (err) reject(err);
            else resolve(connection);
        });

    });
}
```

Node.js 8.0+ provides a [util.promisify\(\)](#) utility to convert a callback-based function into a Promise-based alternative. There are a couple of conditions:

1. the callback must be passed as the last parameter to an asynchronous function, and
2. the callback function must expect an error followed by a value parameter.

Example:

```
// Node.js: promisify fs.readFile
const
  util = require('util'),
  fs = require('fs'),
  readFileAsync = util.promisify(fs.readFile);

readFileAsync('file.txt');
```

Various client-side libraries also provide promisify options, but you can create one yourself in a few lines:

```
// promisify a callback function passed as the last parameter
// the callback function must accept (err, data) parameters
function promisify(fn) {
  return function() {
    return new Promise(
      (resolve, reject) => fn(
        ...Array.from(arguments),
        (err, data) => err ? reject(err) : resolve(data)
    );
  };
}
```

```

        )
    );
}

// example
function wait(time, callback) {
    setTimeout(() => { callback(null, 'done'); }, time);
}

const asyncWait = promisify(wait);

asyncWait(1000);

```

Asynchronous Chaining

Anything that returns a Promise can start a series of asynchronous function calls defined in `.then()` methods. Each is passed the result from the previous `resolve`:

```

asyncDBconnect('http://localhost:1234')
    .then(asyncGetSession)          // passed result of asyncDBconnect
    .then(async GetUser)           // passed result of asyncGetSession
    .then(asyncLogAccess)          // passed result of async GetUser
    .then(result => {              // non-asynchronous function
        console.log('complete');   // (passed result of asyncLogAccess)
        return result;             // (result passed to next .then())
    })
    .catch(err => {               // called on any reject
        console.log('error', err);
    });

```

Synchronous functions can also be executed in `.then()` blocks. The returned value is passed to the next `.then()` (if any).

The `.catch()` method defines a function that's called when any previous `reject` is fired. At that point, no further `.then()` methods will be run. You can have multiple `.catch()` methods throughout the chain to capture different errors.

ES2018 introduces a `.finally()` method, which runs any final logic regardless of the outcome — for example, to clean up, close a database connection etc. It's currently supported in Chrome and Firefox only, but Technical Committee 39 has released a [.finally\(\) polyfill](#).

```
function doSomething() {
```

```
doSomething1()
  .then(doSomething2)
  .then(doSomething3)
  .catch(err => {
    console.log(err);
  })
  .finally(() => {
    // tidy-up here!
  });
}
```

Multiple Asynchronous Calls with Promise.all()

Promise .then() methods run asynchronous functions one after the other. If the order doesn't matter — for example, initialising unrelated components — it's faster to launch all asynchronous functions at the same time and finish when the last (slowest) function runs resolve.

This can be achieved with Promise.all(). It accepts an array of functions and returns another Promise. For example:

```
Promise.all([ async1, async2, async3 ])
  .then(values => {           // array of resolved values
    console.log(values);      // (in same order as function array)
    return values;
  })
  .catch(err => {             // called on any reject
    console.log('error', err);
  });
}
```

Promise.all() terminates immediately if any one of the asynchronous functions calls reject.

Multiple Asynchronous Calls with Promise.race()

Promise.race() is similar to Promise.all(), except that it will resolve or reject as soon as the *first* Promise resolves or rejects. Only the fastest Promise-based asynchronous function will ever complete:

```
Promise.race([ async1, async2, async3 ])
  .then(value => {           // single value
    console.log(value);
    return value;
  });
}
```

```
.catch(err => {           // called on any reject
  console.log('error', err);
});
```

A Promising Future?

Promises reduce callback hell but introduce their own problems.

Tutorials often fail to mention that *the whole Promise chain is asynchronous*. Any function using a series of promises should either return its own Promise or run callback functions in the final `.then()`, `.catch()` or `.finally()` methods.

I also have a confession: *Promises confused me for a long time*. The syntax often seems more complicated than callbacks, there's a lot to get wrong, and debugging can be problematic. However, it's essential to learn the basics.

Further Promise resources:

- [MDN Promise documentation](#)
- [JavaScript Promises: an Introduction](#)
- [JavaScript Promises ... In Wicked Detail](#)
- [Promises for asynchronous programming](#)

Async/Await

Promises can be daunting, so [ES2017](#) introduced `async` and `await`. While it may only be syntactical sugar, it makes Promises far sweeter, and you can avoid `.then()` chains altogether. Consider the Promise-based example below:

```
function connect() {

  return new Promise((resolve, reject) => {

    asyncDBconnect('http://localhost:1234')
      .then(asyncGetSession)
      .then(async GetUser)
      .then(asyncLogAccess)
      .then(result => resolve(result))
      .catch(err => reject(err))

  });
}
```

```
// run connect (self-executing function)
(() => {
  connect()
    .then(result => console.log(result))
    .catch(err => console.log(err))
})();
```

To rewrite this using `async/await`:

1. the outer function must be preceded by an `async` statement, and
2. calls to asynchronous Promise-based functions must be preceded by `await` to ensure processing completes before the next command executes.

```
async function connect() {

  try {
    const
      connection = await asyncDBconnect('http://localhost:1234'),
      session = await asyncGetSession(connection),
      user = await async GetUser(session),
      log = await asyncLogAccess(user);

    return log;
  }
  catch (e) {
    console.log('error', err);
    return null;
  }
}

// run connect (self-executing async function)
(async () => { await connect(); })();
```

`await` effectively makes each call appear as though it's synchronous, while not holding up JavaScript's single processing thread. In addition, `async` functions always return a Promise so they, in turn, can be called by other `async` functions.

`async/await` code may not be shorter, but there are considerable benefits:

1. The syntax is cleaner. There are fewer brackets and less to get wrong.
2. Debugging is easier. Breakpoints can be set on any `await` statement.
3. Error handling is better. `try/catch` blocks can be used in the same way as synchronous code.

- Support is good. It's implemented in all browsers (except IE and Opera Mini) and Node 7.6+.

That said, not all is perfect ...

Promises, Promises

`async/await` still relies on Promises, which ultimately rely on callbacks. You'll need to understand how Promises work, and there's no direct equivalent of `Promise.all()` and `Promise.race()`. It's easy to forget about `Promise.all()`, which is more efficient than using a series of unrelated `await` commands.

Asynchronous Awaits in Synchronous Loops

At some point you'll try calling an asynchronous function *inside* a synchronous loop. For example:

```
async function process(array) {
  for (let i of array) {
    await doSomething(i);
  }
}
```

It won't work. Neither will this:

```
async function process(array) {
  array.forEach(async i => {
    await doSomething(i);
  });
}
```

The loops themselves remain synchronous and will always complete before their inner asynchronous operations.

ES2018 introduces asynchronous iterators, which are just like regular iterators except the `next()` method returns a Promise. Therefore, the `await` keyword can be used with `for ... of` loops to run asynchronous operations in series. for example:

```
async function process(array) {
  for await (let i of array) {
    doSomething(i);
  }
}
```

```
    }  
}
```

However, until asynchronous iterators are implemented, it's possibly best to map array items to an `async` function and run them with `Promise.all()`. For example:

```
const  
  todo = ['a', 'b', 'c'],  
  alltodo = todo.map(async (v, i) => {  
    console.log('iteration', i);  
    await processSomething(v);  
});  
  
await Promise.all(alltodo);
```

This has the benefit of running tasks in parallel, but it's not possible to pass the result of one iteration to another, and mapping large arrays could be computationally expensive.

try/catch Ugliness

`async` functions will silently exit if you omit a `try/catch` around any `await` which fails. If you have a long set of asynchronous `await` commands, you may need multiple `try/catch` blocks.

One alternative is a higher-order function, which catches errors so `try/catch` blocks become unnecessary (thanks to [@wesbos](#) for the suggestion):

```
async function connect() {  
  
  const  
    connection = await asyncDBconnect('http://localhost:1234'),  
    session = await asyncGetSession(connection),  
    user = await async GetUser(session),  
    log = await asyncLogAccess(user);  
  
  return true;  
}  
  
// higher-order function to catch errors  
function catchErrors(fn) {  
  return function (...args) {  
    return fn(...args).catch(err => {  
      console.log('ERROR', err);  
    });  
  };  
}
```

```
        });
    }
}

(async () => {
  await catchErrors(connect)();
})();
```

However, this option may not be practical in situations where an application must react to some errors in a different way from others.

Despite some pitfalls, `async/await` is an elegant addition to JavaScript. Further resources:

- MDN [async](#) and [await](#)
- [Async functions - making promises friendly](#)
- [TC39 Async Functions specification](#)
- [Simplifying Asynchronous Coding with Async Functions](#)

JavaScript Journey

Asynchronous programming is a challenge that's impossible to avoid in JavaScript. Callbacks are essential in most applications, but it's easy to become entangled in deeply nested functions.

Promises abstract callbacks, but there are many syntactical traps. Converting existing functions can be a chore and `.then()` chains still look messy.

Fortunately, `async/await` delivers clarity. Code looks synchronous, but it can't monopolize the single processing thread. It will change the way you write JavaScript and could even make you appreciate Promises — if you didn't before!

Chapter 8: JavaScript's New Private Class Fields, and How to Use Them

by Craig Buckler

ES6 introduced classes to JavaScript, but they're too simplistic for complex applications. Class fields (also referred to as *class properties*) aim to deliver simpler constructors with private and static members. The proposal is currently at [TC39 stage 3: candidate](#) and could appear in ES2019 (ES10).

A quick recap of ES6 classes is useful before we examine class fields in more detail.

ES6 Class Basics

JavaScript's prototypal inheritance model can appear confusing to developers with an understanding of the classical inheritance used in languages such as C++, C#, Java and PHP. JavaScript classes are primarily syntactical sugar, but they offer more familiar object-oriented programming concepts.

A class is a *template* which defines how objects of that type behave. The following `Animal` class defines generic animals (classes are normally denoted with an initial capital to distinguish them from objects and other types):

```
class Animal {  
  
    constructor(name = 'anonymous', legs = 4, noise = 'nothing') {
```

```
this.type = 'animal';

this.name = name;

this.legs = legs;

this.noise = noise;

}

speak() {

    console.log(`"${this.name}" says "${this.noise}"`);

}

walk() {
```

```
        console.log(`\$ {this.name} walks on \$ {this.legs} legs`);  
    }  
  
}  
}
```

Class declarations execute in strict mode; there's no need to add 'use strict'.

A constructor method is run when an object of this type is created, and it typically defines initial properties. `speak()` and `walk()` are methods which add further functionality.

An object can now be created from this class with the `new` keyword:

```
const rex = new Animal('Rex', 4, 'woof');  
  
rex.speak();           // Rex says "woof"  
  
rex.noise = 'growl';  
  
rex.speak();           // Rex says "growl"
```

Getters and Setters

Setters are special methods used to define values only. Similarly, *Getters* are special methods used to return a value only. For example:

```
class Animal {  
  
    constructor(name = 'anonymous', legs = 4, noise = 'nothing') {  
  
        this.type = 'animal';  
  
        this.name = name;  
  
        this.legs = legs;  
  
        this.noise = noise;  
  
    }  
  
    speak() {  
  
        console.log(`"${this.name}" says "${this.noise}"`);  
  
    }  
}
```

```
}

walk() {

    console.log(` ${this.name} walks on ${this.legs} legs`);

}

// setter

set eats(food) {

    this.food = food;

}

// getter

get dinner() {
```

```
    return `${this.name} eats ${this.food || 'nothing'} for dinner.`

}

}

const rex = new Animal('Rex', 4, 'woof');

rex.eats = 'anything';

console.log( rex.dinner ); // Rex eats anything for dinner.
```

Child or Sub-Classes

It's often practical to use one class as the base for another. If we're mostly creating dog objects, `Animal` is too generic, and we must specify the same 4-leg and "woof" noise defaults every time.

A `Dog` class can inherit all the properties and methods from the `Animal` class using `extends`. Dog-specific properties and methods can be added or removed as necessary:

```
class Dog extends Animal {
```

```
constructor(name) {  
  
    // call the Animal constructor  
  
    super(name, 4, 'woof');  
  
    this.type = 'dog';  
  
}  
  
  
// override Animal.speak  
  
speak(to) {  
  
    super.speak();
```

```
if (to) console.log(`to ${to}`);  
  
}  
  
}  
  
}
```

`super` refers to the parent class and is usually called in the constructor. In this example, the `Dog speak()` method overrides the one defined in `Animal`.

Object instances of `Dog` can now be created:

```
const rex = new Dog('Rex');  
  
rex.speak('everyone');      // Rex says "woof" to everyone  
  
rex.eats = 'anything';  
  
console.log( rex.dinner ); // Rex eats anything for dinner.
```

Static Methods and Properties

Defining a method with the `static` keyword allows it to be called on a class without creating an object instance. JavaScript doesn't support static properties in the same way as other languages, but it is possible to add properties to the class definition (a class is a JavaScript object in itself!).

The `Dog` class can be adapted to retain a count of how many dog objects have been created:

```
class Dog extends Animal {  
  
    constructor(name) {  
  
        // call the Animal constructor  
  
        super(name, 4, 'woof');  
  
        this.type = 'dog';  
  
        // update count of Dog objects  
  
        Dog.count++;  
    }  
}  
  
Dog.prototype.speak = function() {  
    return this.type + ': ' + this.legs + ' legs, ' + this.bark;  
};
```

```
}

// override Animal.speak

speak(to) {

    super.speak();

    if (to) console.log(`to ${to}`);

}

// return number of dog objects

static get COUNT() {

    return Dog.count;
```

```
}

}

// static property (added after class is defined)

Dog.count = 0;
```

The class's static COUNT getter returns the number of dogs created:

```
console.log(`Dogs defined: ${Dog.COUNT}`); // Dogs defined: 0

const don = new Dog('Don');

console.log(`Dogs defined: ${Dog.COUNT}`); // Dogs defined: 1
```

```
const kim = new Dog('Kim');

console.log(`Dogs defined: ${Dog.COUNT}`); // Dogs defined: 2
```

For more information, refer to [Object-oriented JavaScript: A Deep Dive into ES6 Classes](#).

ESnext Class Fields

The class fields proposal allows properties to be initialized at the top of a class:

```
class MyClass {  
  
    a = 1;  
  
    b = 2;  
  
    c = 3;  
  
}
```

This is equivalent to:

```
class MyClass {  
  
    constructor() {  
  
        this.a = 1;  
  
        this.b = 2;  
  
        this.c = 3;  
  
    }  
  
}
```

Initializers are executed before any constructor runs (*presuming a constructor is still necessary*).

Static Class Fields

Class fields permit static properties to be declared within the class. For example:

```
class MyClass {  
  
    x = 1;  
  
    y = 2;  
  
    static z = 3;  
  
}  
  
console.log( MyClass.z ); // 3
```

The inelegant ES6 equivalent:

```
class MyClass {  
  
    constructor() {  
  
        this.x = 1;
```

```
this.y = 2;  
}  
  
}  
  
MyClass.z = 3;  
  
console.log( MyClass.z ); // 3
```

Private Class Fields

All properties in ES6 classes are public by default and can be examined or modified outside the class. In the `Animal` example above, there's nothing to prevent the `food` property being changed without calling the `eats` setter:

```
class Animal {  
  
}  
  
constructor(name = 'anonymous', legs = 4, noise = 'nothing') {
```

```
this.type = 'animal';

this.name = name;

this.legs = legs;

this.noise = noise;

}
```

```
set eats(food) {

    this.food = food;

}
```

```
get dinner() {
```

```
    return `${this.name} eats ${this.food || 'nothing'} for dinner.`

}

}

const rex = new Animal('Rex', 4, 'woof');

rex.eats = 'anything';      // standard setter

rex.food = 'tofu';          // bypass the eats setter altogether

console.log( rex.dinner ); // Rex eats tofu for dinner.
```

Other languages permit private properties to be declared. That's not possible in ES6, although developers can work around it using an underscore convention (`_propertyName`), [closures, symbols, or WeakMaps](#).

In ESnext, private class fields are defined using a hash # prefix:

```
class MyClass {
```

```
a = 1;           // .a is public

#b = 2;           // .#b is private

static #c = 3;   // .#c is private and static

incB() {

    this.#b++;

}

const m = new MyClass();

m.incB(); // runs OK
```

```
m.#b = 0; // error - private property cannot be modified outside class
```

Note that there's no way to define private methods, getters and setters, although a [TC39 stage 2: draft proposal](#) suggests using a hash # prefix on names. For example:

```
class MyClass {  
  
    // private property  
  
    #x = 0;  
  
    // private method (can only be called within the class)  
  
    #incX() {  
  
        this.#x++;  
  
    }  
  
    // private setter (can only be called within the class)
```

```
set #setX(x) {  
  
    this.#x = x;  
  
}  
  
  
  
  
  
  
// private getter (can only be called within the class)  
  
get #getX() {  
  
    return this.$x;  
  
}  
  
  
}
```

Immediate Benefit: Cleaner React Code!

React components often have methods tied to DOM events. To ensure this resolves to the component, it's necessary to bind every method accordingly. For example:

```
class App extends Component {
```

```
constructor() {  
  
    super();  
  
    state = { count: 0 };  
  
    // bind all methods  
  
    this.incCount = this.incCount.bind(this);  
  
}  
  
incCount() {  
  
    this.setState(ps => ({ count: ps.count + 1 }));
```

```
}

render() {

  return (
    <div>
      <p>{ this.state.count }</p>
      <button onClick={this.incCount}>add one</button>
    </div>
  );
}

}
```

If `incCount` is defined as a class field, it can be set to a function using an ES6 => fat arrow, which automatically binds it to the defining object. The state can also be declared as a class field so no constructor is required:

```
class App extends Component {  
  
  state = { count: 0 };  
  
  incCount = () => {  
  
    this.setState(ps => ({ count: ps.count + 1 }));  
  
  };  
  
  render() {  
  
    return (  
  
      <div>
```

```
<p>{ this.state.count }</p>

<button onClick={this.incCount}>add one</button>

</div>

);

}

}
```

Using Class Fields Today

Class fields are not currently supported in browsers or Node.js. However, it's possible to [transpile the syntax using Babel](#), which is enabled by default when using [Create React App](#). Alternatively, Babel can be installed and configured using the following terminal commands:

```
mkdir class-properties

cd class-properties

npm init -y
```

```
npm install --save-dev babel-cli babel-plugin-transform-class-proper  
  
echo '{ "plugins": ["transform-class-properties"] }' > .babelrc
```

Add a build command to the scripts section of package.json:

```
"scripts": {  
  
  "build": "babel in.js -o out.js"  
  
},
```

Then run with `npm run build` to transpile the ESnext file `in.js` to a cross-browser-compatible `out.js`.

Babel support for [private methods, getters and setters has been proposed](#).

Class Fields: an Improvement?

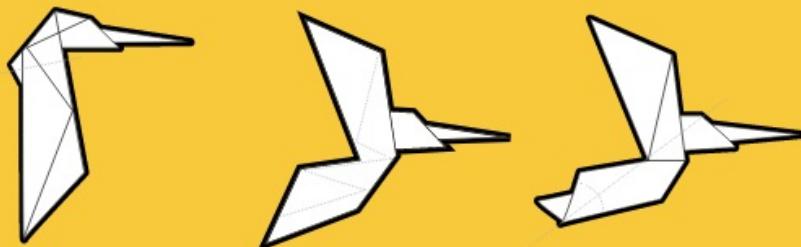
ES6 class definitions were simplistic. Class fields should aid readability and enable some interesting options. I don't particularly like using a hash # to denote private members, but unexpected behaviors and performance issues would be incurred without it (refer to [JavaScript's new #private class fields](#) for a detailed explanation).

Perhaps it goes without saying, but I'm going to say it anyway: *the concepts discussed in this article are subject to change and may never be implemented!* That said, JavaScript class fields have practical benefits and interest is rising. It's a safe bet.

Book 3: 6 JavaScript Projects



6 JAVASCRIPT PROJECTS



7.

8.

9.

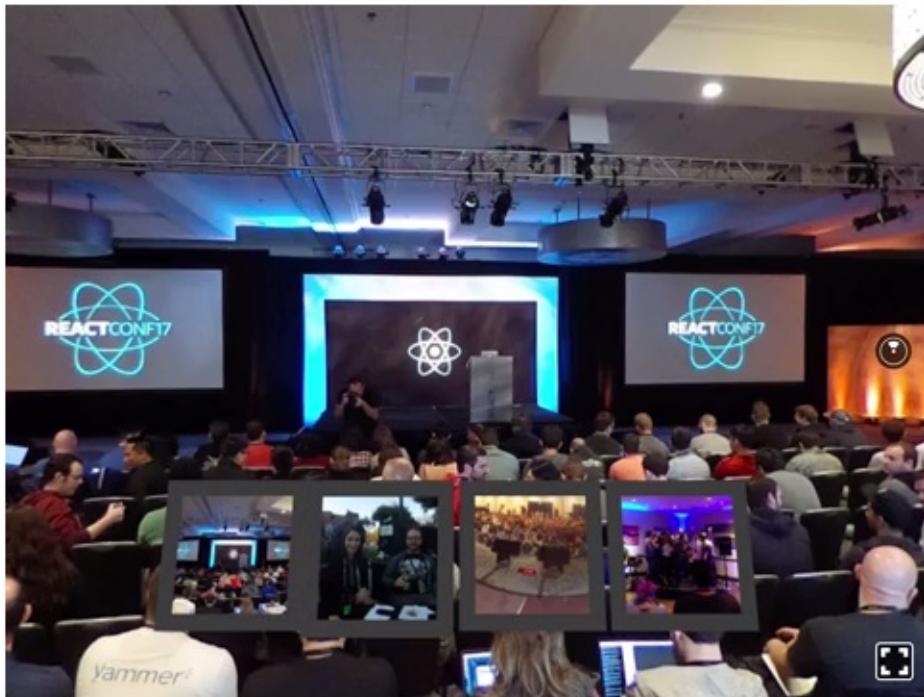
COMPLETE PROJECT TUTORIALS

Chapter 1: Build a Full-Sphere 3D Image Gallery with React VR

by Michaela Lehr

[React VR](#) is a JavaScript library by Facebook that reduces the effort of creating a [WebVR](#) application. You may compare React VR with [A-Frame by Mozilla](#), but instead of writing HTML, with React VR we're using JavaScript to create a WebVR scene.

React VR is built on the WebGL library [three.js](#) and the React Native framework. This means that we're able to use JSX tags, React Native components, like `<View>` or `<Text>`, or React Native concepts, like the flexbox layout. To simplify the process of creating a WebVR scene, React VR has built-in support for 3D meshes, lights, videos, 3D shapes, or spherical images.



In this chapter, we want to use React VR to build a viewer for spherical images. For this, we'll use four [equirectangular](#) photos, which I shot at [React Conf 2017](#) with my [Theta S camera](#). The gallery will have four buttons to swap the images, which will work with the mouse and/or VR headset. You can download the equirectangular images as well as the button graphics [here](#). Last but not least, we'll take a look at how animations work with React VR by adding a simple button transition.

For development, we're using a browser like Chrome on the desktop. To check if the stereoscopic rendering for VR devices works, we're using a Samsung phone with Gear VR. In theory, any mobile browser capable of WebVR should be able to render our app in a stereoscopic way for the usage with GearVR, Google Cardboard, or even Google Daydream. But the library, as well as the API, are still under development, so the support may not be reliable. [Here's a good summary](#) of browsers currently supporting WebVR features.

Development Setup and Project Structure

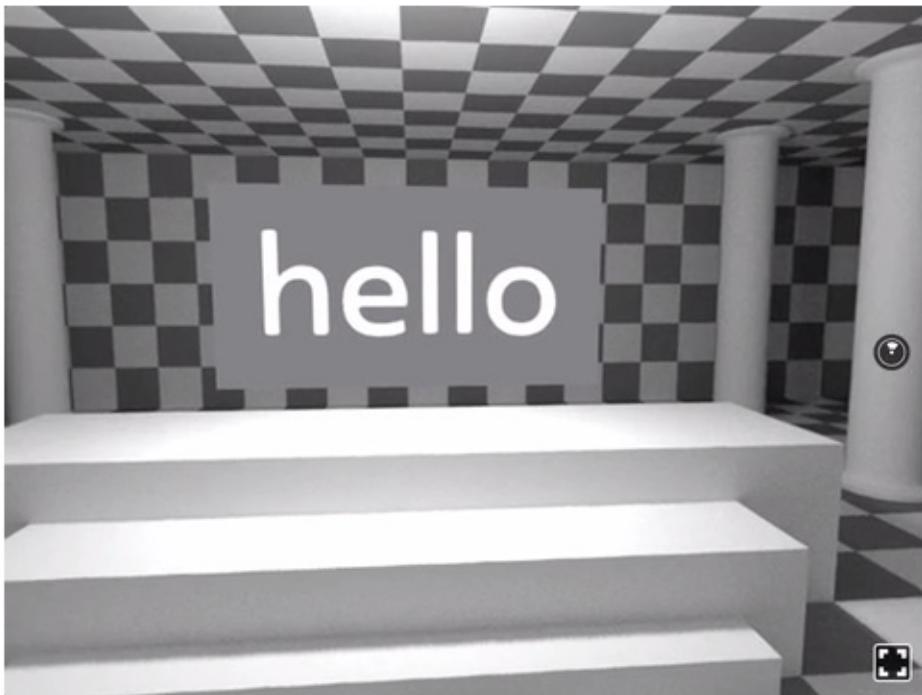
Let's start by installing the React VR CLI tool. Then create a new React VR project with all its dependencies in a new folder called GDVR.REACTVR_SITEPOINT_GALLERY:

```
npm install -g react-vr-cli  
  
react-vr init GDVR.REACTVR_SITEPOINT_GALLERY  
  
cd GDVR.REACTVR_SITEPOINT_GALLERY
```

To start a local development server, we'll run an npm script and browse to <http://localhost:8081/vr/> in Chrome.

```
npm start
```

If you see a black and white room with stairs, pillars, and a “hello” text plane, everything's correct.



The most important files and folders scaffolded by the React VR CLI are:

- `index.vr.js`. This is the entry point of the application. Currently, the file contains the whole source code of React VR's default scene, as we already saw in the browser.
- `static_assets`. This folder should contain all assets used in the application. We'll put the equirectangular images and the button graphics in this folder.

We want our project to have three components:

- a **Canvas** component, which holds the code for the full-sphere images
- a **Button** component, which creates a VR button to swap the images
- a **UI** component, which builds a UI out of four Button components.

The three components will each have their own file, so let's create a components folder to contain these files. Then, before we start creating the Canvas component, let's remove the scaffolded example code from the `index.vr.js` file so it looks like this:

```
/* index.vr.js */

import React from 'react';

import {
  AppRegistry,
  View,
} from 'react-vr';

export default class GDVR.REACTVR_SITEPOINT_GALLERY extends React.Component {

  render() {

    return (
      <View>

        </View>
    );
  }
}
```

```
);

}

};

AppRegistry.registerComponent('GDVR.REACTVR_SITEPOINT_GALLERY', () =>
```

Adding a Spherical Image to the Scene

To add a spherical image to the scene, we'll create a new file `Canvas.js` in the `components` folder:

```
/* Canvas.js */

import React from 'react';

import {
  asset,
  Pano,
} from 'react-vr';
```

```
class Canvas extends React.Component {  
  
  constructor(props) {  
  
    super(props);  
  
    this.state = {  
  
      src: this.props.src,  
  
    }  
  
  }  
  
  render() {  
  
    return (  
      <img alt="Placeholder image" src={this.state.src} />  
    );  
  }  
}
```

```
<Pano source={asset(this.state.src)}/>

);

}

};

export default Canvas;
```

In the first six lines of code, we import the dependencies. Then we declare our Canvas component and define how it renders by using the JSX syntax.

More on JSX

If you want to learn more about JSX, I recommend you check out ["Getting Started with React and JSX"](#).

A look at the JSX code reveals that the Canvas component returns only one component, the React VR `<Pano>` component. It has a parameter, the `source` prop, that uses an `asset` function to load the image from the `static_assets` folder. The argument refers to a state, which we initialized in the constructor function.

In our case, we don't want to define the path in the Canvas component itself, but use the `index.vr.js` file to define all image paths. This is why the `state.src` object refers to the component's `props` object.

More on State and Props

Check out the ReactJS documentation for [React.Component](#) if you would like to know more about state and props.

Let's continue by modifying the `index.vr.js` file to use the Canvas component and render it to the scene:

```
/* index.vr.js */

import React from 'react';

import {

  AppRegistry,
  View,

} from 'react-vr';

import Canvas from './components/Canvas';

export default class GDVR.REACTVR_SITEPOINT_GALLERY extends React.Component {

  constructor() {
    super();
  }

  render() {
    return (
      <View>
        <Canvas>
          <Image source={require('./img/placeholder.png')} />
        </Canvas>
      </View>
    );
  }
}
```

```
        this.state = {  
  
          src: 'reactconf_00.jpg',  
  
        };  
  
      }  
  
      render() {  
  
        return (  
  
          <View>  
  
            <Canvas  
  
              src={this.state.src}  
  
            />  
  
          </View>  
        );  
      }  
    );  
  );  
};
```

```
);

}

};

AppRegistry.registerComponent('GDVR.REACTVR_SITEPOINT_GALLERY', () =>
```

Besides the already used React VR dependencies, we need to import our custom Canvas component. Next, we declare the application class in line six:

```
/* index.vr.js */

import Canvas from './components/Canvas';
```

Then, we add the `<Canvas>` component as a child component of the `<View>` component. We're using `src` as the component's prop because we're referring to it in the Canvas component. A look in the browser should now show the panoramic image, and we should already be able to interact with it.



Create a UI Component to Hold Four Buttons

What we want to do now is to create four buttons that a user can trigger to swap the images. So we'll add two new components: a UI component, and its child component, a Button component. Let's start with the Button component:

```
/* Button.js */

import React from 'react';

import {
```

```
Image,  
  
View,  
  
VrButton,  
  
} from 'react-vr';  
  
  
  
class Button extends React.Component {  
  
  
  
  
  
onButtonClick = () => {  
  
    this.props.onClick();  
  
}  
  
  
  
  
  
render () {  
  
    return (  
        <VrButton  
            style={...}  
            onButtonClick={this.onButtonClick}/>  
    );  
}  
};
```

```
<View

    style={{

        alignItems: 'center',

        flexDirection: 'row',

        margin: 0.0125,

        width: 0.7,


    }}

>

<VrButton

    onClick={this.onButtonClick}

>

<Image

    style={{
```

```
        width: 0.7,  
  
        height: 0.7,  
  
    }  
  
    source={asset(this.props.src)}  
  
>  
  
</Image>  
  
</VrButton>  
  
</View>  
);  
  
};  
  
};  
  
export default Button;
```

To build the button, we're using React VR's `<VrButton>` component, which we import in line six. Also, we're using an image component to add our asset images to each button, since the `<VrButton>` component itself has no appearance. Like before, we're using a prop to define the image source. Another feature we're using twice in this component is the `style` prop, to add layout values to each button and its image. The `<VrButton>` also makes use of an event listener, `onClick`.

To add four Button components to our scene, we'll use the UI parent component, which we'll add as a child in `index.vr.js` afterward. Before writing the UI component, let's create a config object defining the relation between the equirectangular images, the button images, and the buttons themselves. To do this, we declare a constant right after the import statements in the `index.vr.js` file:

```
/* index.vr.js */

const Config = [
  {
    key: 0,
    imageSrc: 'reactconf_00.jpg',
    buttonImageSrc: 'button-00.png',
  },
  {

```

```
        key: 1,  
  
        imageSrc: 'reactconf_01.jpg',  
  
        buttonImageSrc: 'button-01.png',  
  
    },  
  
{  
  
    key: 2,  
  
    imageSrc: 'reactconf_02.jpg',  
  
    buttonImageSrc: 'button-02.png',  
  
},  
  
{  
  
    key: 3,  
  
    imageSrc: 'reactconf_03.jpg',  
  
    buttonImageSrc: 'button-03.png',
```

```
}
```

```
];
```

The UI component will use the values defined in the config to handle the gaze and click events:

```
/* UI.js */  
  
import React from 'react';  
  
import {  
  
    View,  
  
} from 'react-vr';  
  
import Button from './Button';  
  
class UI extends React.Component {  
  
    constructor(props) {
```

```
super(props);

this.buttons = this.props.buttonConfig;

}

render () {

const buttons = this.buttons.map((button) =>

<Button

key={button.key}

onClick={()=>{

this.props.onClick(button.key);

}};

src={button.buttonImageSrc}
```

```
 />

);

return (
<View
style={{
  flexDirection: 'row',
  flexWrap: 'wrap',
  transform: [
    {rotateX: -12},
    {translate: [-1.5, 0, -3]},
  ],
  width: 3,
```

```
        }

      >

      {buttons}

    </View>

  );

}

};

export default UI;
```

To set the source of an image, we're using the config values we already added to the `index.vr.js` file. We're also using the prop `onClick` to handle the click event, which we'll also add in a few moments to the `index.vr.js` file. Then we create as many buttons as defined in the button config object, to add them later in the JSX code that will be rendered to the scene:

```
/* UI.js */

const buttons = this.buttons.map((button) =>
```

```
<Button

    key={button.key}

    onClick={()=>{

        this.props.onClick(button.key);

    }}

    src={button.buttonImageSrc}

/>

);
```

Now, all we have to do is add the UI component to the scene defined in the `index.vr.js` file. So we import the UI component right after importing the Canvas component:

```
/* index.vr.js */

import UI from './components/UI';
```

Next, we add the `<Canvas>` component to the scene:

```
/* index.vr.js */
```

```
<View>

  <Canvas

    src={this.state.src}

  />

  <UI

    buttonConfig={Config}

    onClick={(key)=>{

      this.setState({src: Config[key].imageSrc});

    }}

  />

</View>
```

When checking this code in the browser, you'll notice that the click doesn't trigger an image source swap at the moment. To listen for updated props, we'll have to add another function to the Canvas component right after the constructor function.

Component Lifecycle

If you're interested in the lifecycle of a React component, you might want to read about [React.Component in the React docs](#).

```
/* Canvas.js */\n\ncomponentWillReceiveProps(nextProps) {\n\n    this.setState({src: nextProps.src});\n\n}
```

A test in the browser should now be successful, and a click on a button image should change the spherical image.



Add Animations for Button State Transitions

To make the buttons more responsive to user interactions, we want to add some hover states and transitions between the default idle and the hover state. To do this, we'll use the [Animated library](#) and [Easing functions](#), and then write two functions for each transition: `animateIn` and `animateOut`:

```
/* Button.js */\n\nimport React from 'react';\n\nimport {\n  Animate,\n  AnimatedValue,\n  Easing,\n  useSharedValue,\n  useValue,\n  withTiming,\n} from 'react-native-reanimated';\n\nconst Button = () => {\n  const [value, setValue] = useSharedValue(0);\n\n  const handlePress = () => {\n    Animated.timing(value, {\n      duration: 500,\n      easing: Easing.inOut,\n      to: 1,\n    }).start();\n  }\n\n  const handleRelease = () => {\n    Animated.timing(value, {\n      duration: 500,\n      easing: Easing.out,\n      to: 0,\n    }).start();\n  }\n\n  return (\n    <View>\n      <Text>{value}</Text>\n      <Text>{value}</Text>\n      <Text>{value}</Text>\n      <Text>{value}</Text>\n    </View>\n  );\n};\n\nexport default Button;
```

```
Animated,  
  
asset,  
  
Image,  
  
View,  
  
VrButton,  
  
} from 'react-vr';  
  
const Easing = require('Easing');  
  
class Button extends React.Component {  
  
  constructor(props) {  
  
    super();  
  }  
}
```

```
this.state = {  
  
    animatedTranslation: new Animated.Value(0),  
  
};  
  
}  
  
  
animateIn = () => {  
  
    Animated.timing(  
  
        this.state.animatedTranslation,  
  
        {  
  
            toValue: 0.125,  
  
            duration: 100,  
  
            easing: Easing.in,
```

```
        }

    ).start();

}

animateOut = () => {

    Animated.timing(
        this.state.animatedTranslation,
        {
            toValue: 0,
            duration: 100,
            easing: Easing.in,
        }
    ).start();
}
```

```
}

onButtonClick = () => {

    this.props.onClick();

}

render () {

    return (

        <Animated.View

            style={{

                alignItems: 'center',

                flexDirection: 'row',

                margin: 0.0125,
    
```

```
    transform: [  
      {translateZ: this.state.animatedTranslation},  
    ],  
  
    width: 0.7,  
  
  }}  
  
>  
  
<VrButton  
  
  onClick={this.onButtonClick}  
  
  onEnter={this.animateIn}  
  
  onExit={this.animateOut}  
  
>  
  
<Image  
  
  style={{
```

```
        width: 0.7,  
  
        height: 0.7,  
  
    }]  
  
    source={asset(this.props.src)}  
  
>  
  
</Image>  
  
</VrButton>  
  
</Animated.View>  
  
);  
  
}  
  
};  
  
export default Button;
```

After adding the dependencies, we define a new state to hold the translation value we want to animate:

```
/* Button.js */

constructor(props) {

  super();

  this.state = {

    animatedTranslation: new Animated.Value(0),

  };

}


```

Next, we define two animations, each in a separate function, that describe the animation playing when the cursor enters the button, and when the cursor exits the button:

```
/* Button.js */

animateIn = () => {

  Animated.timing(
```

```
    this.state.animatedTranslation,  
  
    {  
  
      toValue: 0.125,  
  
      duration: 100,  
  
      easing: Easing.in,  
  
    }  
  
.start();  
  
}  
  
animateOut = () => {  
  
  Animated.timing(  
  
    this.state.animatedTranslation,  
  
    {
```

```
        toValue: 0,  
  
        duration: 100,  
  
        easing: Easing.in,  
  
    }  
  
.start();  
  
}  
}
```

To use the `state.animatedTranslation` value in the JSX code, we have to make the `<View>` component animatable, by adding `<Animated.View>`:

```
/* Button.js */  
  
<Animated.View  
  
    style={{  
  
        alignItems: 'center',  
  
        flexDirection: 'row',  
  
        margin: 0.0125,
```

```
    transform: [  
  
      {translateZ: this.state.animatedTranslation},  
  
    ],  
  
    width: 0.7,  
  
  }}  
  
>
```

We'll call the function when the event listeners `onButtonEnter` and `onButtonExit` are triggered:

```
/* Button.js */  
  
<VrButton  
  
  onClick={this.onButtonClick}  
  
  onEnter={this.animateIn}  
  
  onExit={this.animateOut}  
  
>
```

A test of our code in the browser should show transitions between the position on the z-axis of each button:



Building and Testing the Application

Open your app in a browser that supports WebVR and navigate to your development server, by using not `http://localhost:8081/vr/index.html`, but your IP address, for example, `http://192.168.1.100:8081/vr/index.html`. Then, tap on the `View in VR` button, which will open a full-screen view and start the stereoscopic rendering.



To upload your app to a server, you can run the npm script `npm run bundle`, which will create a new `build` folder within the `vr` directory with the compiled files. On your web server you should have the following directory structure:

Web Server

```
├ static_assets/  
|  
└ index.html  
  
└ index.bundle.js  
  
└ client.bundle.js
```

Further Resources

Full Project Code

This is all we had to do create a small WebVR application with React VR. You can find the [entire project code on GitHub](#).

This is all we had to do create a small WebVR application with React VR. React VR has a few more components we didn't discuss in this tutorial:

- There's a `Text` component for rendering text.
- Four different light components can be used to add light to a scene: `AmbientLight`, `DirectionalLight`, `PointLight`, and `Spotlight`.
- A `Sound` component adds spatial sound to a location in the 3D scene.
- To add videos, the `Video` component or the `VideoPano` component can be used. A special `videoControl` component adds controls for video playback and volume.
- With the `Model` component we can add 3D models in the `obj` format to the application.
- A `cylindricalPanel` component can be used to align child elements to the inner surface of a cylinder — for example, to align user interface elements.
- There are three components to create 3D primitives: a `sphere` component, a `plane` component and a `box` component.

Also, React VR is still under development, which is also the reason for it running only in the Carmel Developer Preview browser. If you're interested in learning more about React VR, here are a few interesting resources:

- [React VR Docs](#)
- [React VR on GitHub](#)

- [Awesome React VR](#), a collection of React VR resources.

And if you'd like to dig deeper in WebVR in general, these articles might be right for you:

- “[A-Frame: The Easiest Way to Bring VR to the Web Today](#)”
- “[Embedding Virtual Reality Across the Web with VR Views](#)”

Chapter 2: Build a WebRTC Video Chat Application with SimpleWebRTC

by Michael Wanyoike

With the advent of WebRTC and the increasing capacity of browsers to handle peer-to-peer communications in real time, it's easier than ever to build real-time applications. In this tutorial, we'll take a look at [SimpleWebRTC](#) and how it can make our lives easier when implementing WebRTC. Throughout the chapter, we'll be building a WebRTC video chat app with messaging features.

If you need a bit of a background regarding WebRTC and peer-to-peer communication, I recommend reading [The Dawn of WebRTC](#) and [Introduction to the getUserMedia API](#).

What is SimpleWebRTC

Before we move on, it's important that we understand the main tool that we'll be using. [SimpleWebRTC](#) is a JavaScript library that simplifies WebRTC peer-to-peer data, video, and audio calls.

SimpleWebRTC acts as a wrapper around the browser's WebRTC implementation. As you might already know, browser vendors don't exactly agree on a single way of implementing different features, which means that for every browser there's a different implementation for WebRTC. As the developer, you'd have to write different code for every browser you plan to support. SimpleWebRT acts as the wrapper for that code. The API that it exposes is easy to use and understand, which makes it a really great candidate for implementing cross-browser WebRTC.

Building the WebRTC Video Chat App

Now it's time to get our hands dirty by building the app. We'll build a single page application that runs on top of an Express server.

Running the Examples

Please note that you can download the code for this tutorial from our [GitHub repo](#). To run it, or to follow along at home, you'll need to have Node and npm installed. If you're not familiar with these, or would like some help getting them installed, check out our previous tutorials:

- [Install Multiple Versions of Node.js using nvm](#)
- [A Beginner's Guide to npm — the Node Package Manager](#)

You also need a PC or laptop that has a webcam. If not, you'll need to get yourself a USB webcam that you can attach to the top of your monitor. You'll probably need a friend or a second device to test remote connections.

Dependencies

We'll be using the following dependencies to build our project:

- [SimpleWebRTC](#) — the WebRTC library
- [Semantic UI CSS](#) — an elegant CSS framework
- [jQuery](#) — used for selecting elements on the page and event handling.
- [Handlebars](#) — a JavaScript templating library, which we'll use to generate HTML for the messages
- [Express](#) — NodeJS server.

Project Setup

Go to your workspace and create a folder `simplewebrtc-messenger`. Open the folder in VSCode or your favorite editor and create the following files and folder structure:

```
simplewebrtc-messenger
├── public
│   ├── images
│   │   └── image.png
│   └── index.html
```

```
└── js
    └── app.js
── README.md
└── server.js
```

Or, if you prefer, do the same via the command line:

```
mkdir -p simplewebrtc-messenger/public/{images,js}
cd simplewebrtc-messenger
touch public/js/app.js public/index.html .gitignore README.md server
```

Open README.md and copy the following content:

```
# Simple WebRTC Messenger

A tutorial on building a WebRTC video chat app using SimpleWebRTC.
```

Add the line node_modules to the .gitignore file if you plan to use a git repository. Generate the package.json file using the following command:

```
npm init -y
```

You should get the following output:

```
{
  "name": "simplewebrtc-messenger",
  "version": "1.0.0",
  "description": "A tutorial on building a WebRTC video chat app using SimpleWebRTC",
  "main": "server.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1",
    "start": "node server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Now let's install our dependencies:

```
npm install express handlebars jquery semantic-ui-css simplewebrtc
```

As the installation proceeds, copy this code to server.js:

```
const express = require('express');
```

```

const app = express();
const port = 3000;

// Set public folder as root
app.use(express.static('public'));

// Provide access to node_modules folder from the client-side
app.use('/scripts', express.static(`${__dirname}/node_modules/`));

// Redirect all traffic to index.html
app.use((req, res) => res.sendFile(`${__dirname}/public/index.html`))

app.listen(port, () => {
  console.info('listening on %d', port);
});

```

The server code is pretty standard. Just read the comments to understand what's going on.

Next, let's set up our public/index.html file:

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1"
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="scripts/semantic-ui-css/semantic.min.css"/>
  <title>SimpleWebRTC Demo</title>
  <style>
    html { margin-top: 20px; }
    #chat-content { height: 180px; overflow-y: scroll; }
  </style>
</head>
<body>
  <!-- Main Content -->
  <div class="ui container">
    <h1 class="ui header">Simple WebRTC Messenger</h1>
    <hr>
  </div>

  <!-- Scripts -->
  <script src="scripts/jquery/dist/jquery.min.js"></script>
  <script src="scripts/semantic-ui-css/semantic.min.js"></script>
  <script src="scripts/handlebars/dist/handlebars.min.js "></script>
  <script src="scripts/simplewebrtc/out/simplewebrtc-with-adapter.bundle.js"></script>
  <script src="js/app.js"></script>
</body>

```

```
</html>
```

Next, let's set up our base client-side JavaScript code. Copy this code to `public/js/app.js`:

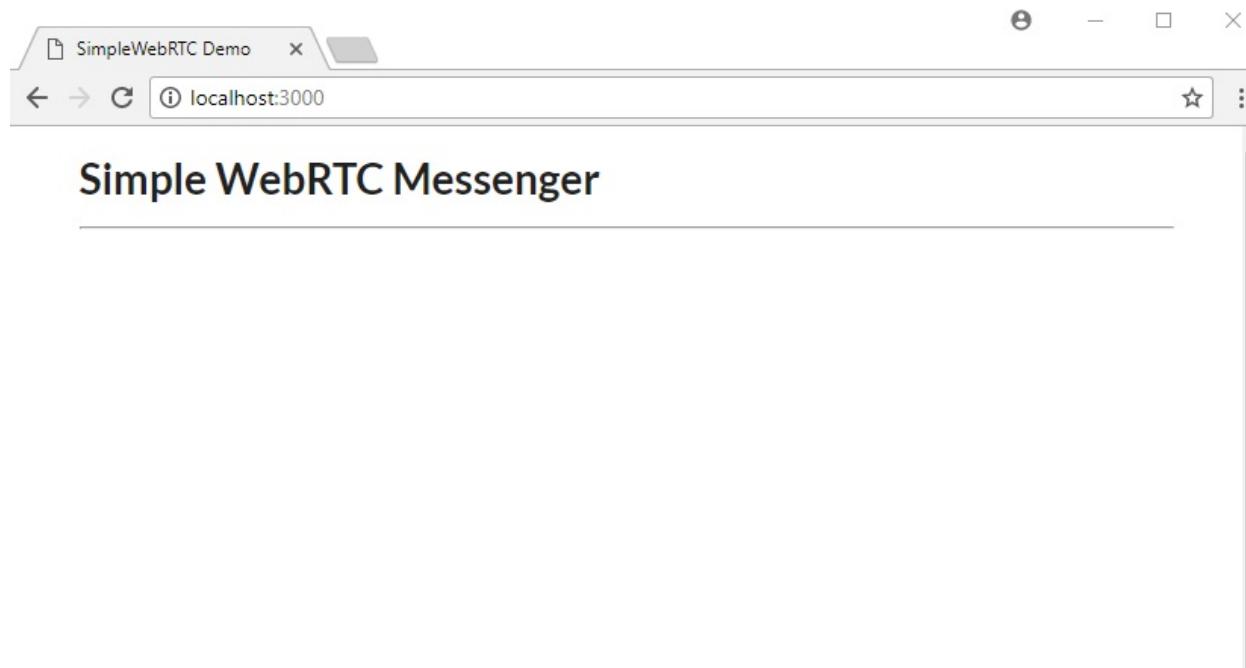
```
window.addEventListener('load', () => {
  // Put all client-side code here
});
```

Finally, download this [image](#) from our GitHub repository and save it inside the `public/images` folder.

Now we can run our app:

```
npm start
```

Open the URL localhost:3000 in your browser and you should see the following:



Markup

Let's now work on `public/index.html`. For the sake of simplicity (especially if you're already familiar with Handlebars) you can copy the entire markup code from our [GitHub repository](#). Otherwise, let's go through things step by step. First off, copy this code and place it after the `<hr>` tag within the `ui container` div:

```
<div class="ui two column stackable grid">

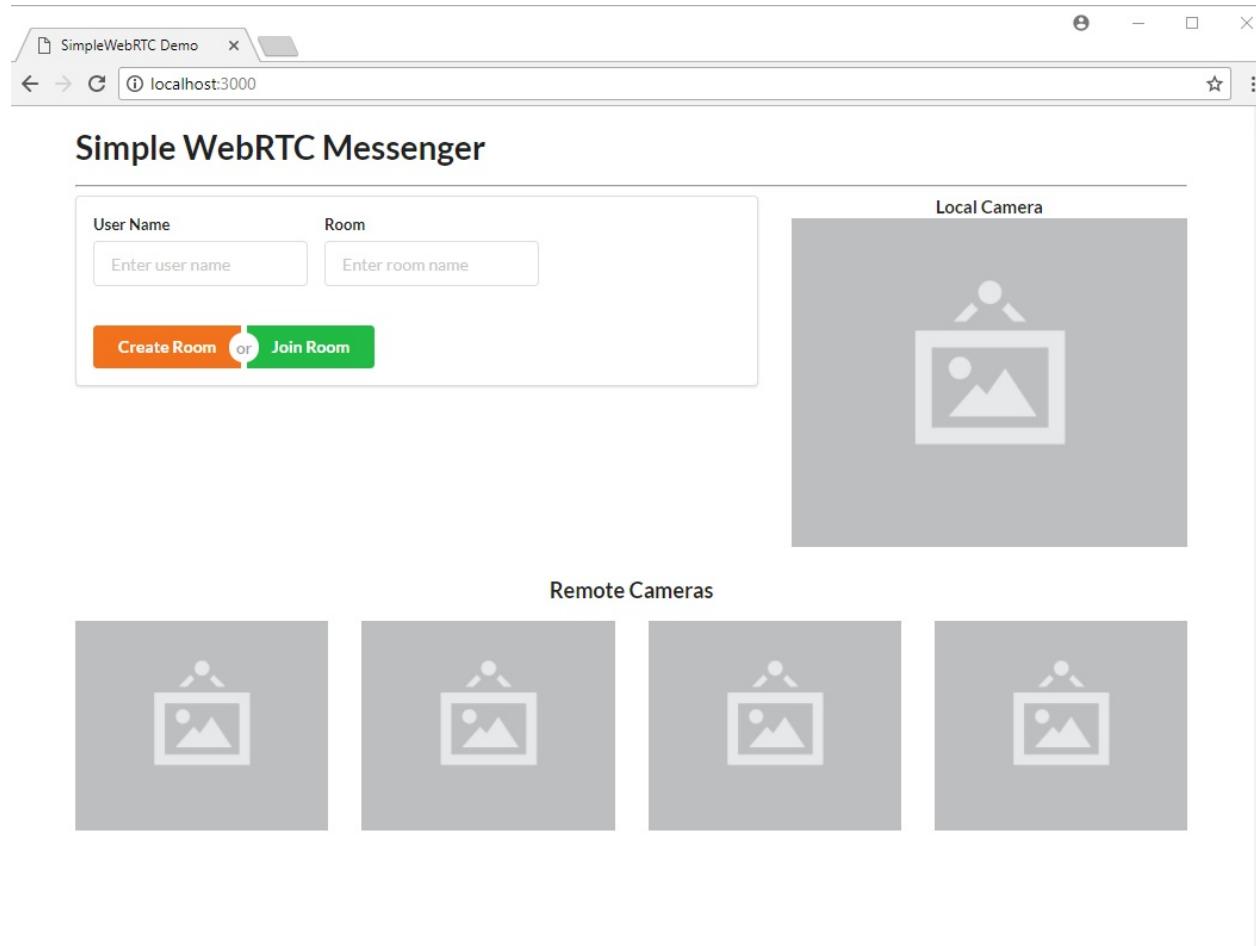
    <!-- Chat Section -->
    <div class="ui ten wide column">
        <div class="ui segment">
            <!-- Chat Room Form -->
            <div class="ui form">
                <div class="fields">
                    <div class="field">
                        <label>User Name</label>
                        <input type="text" placeholder="Enter user name" id="user-name">
                    </div>
                    <div class="field">
                        <label>Room</label>
                        <input type="text" placeholder="Enter room name" id="room-name">
                    </div>
                </div>
                <br>
                <div class="ui buttons">
                    <div id="create-btn" class="ui submit orange button">Create Room</div>
                    <div class="or">Or</div>
                    <div id="join-btn" class="ui submit green button">Join Room</div>
                </div>
            <!-- Chat Room Messages -->
            <div id="chat"></div>
        </div>
    </div>
    <!-- End of Chat Section -->

    <!-- Local Camera -->
    <div class="ui six wide column">
        <h4 class="ui center aligned header" style="margin:0;">
            Local Camera
        </h4>
        
        <video id="local-video" class="ui large image hidden" autoplay></video>
    </div>
</div>

    <!-- Remote Cameras -->
    <h3 class="ui center aligned header">Remote Cameras</h3>
    <div id="remote-videos" class="ui stackable grid">
        <div class="four wide column">
            
        </div>
        <div class="four wide column">
            
        </div>
    </div>
```

```
<div class="four wide column">
  
</div>
<div class="four wide column">
  
</div>
</div>
```

Go through the markup code and read the comments to understand what each section is for. Also check out the [Semantic UI](#) documentation if you're unfamiliar with the CSS library. Refresh your browser. You should have the following view:



We're using a blank image as a placeholder to indicate where the camera location will stream to on the web page. Take note that this app will be able to support multiple remote connections, provided your internet bandwidth can handle it.

Templates

Now let's add the three Handlebar templates that will make our web page interactive.

Place the following markup right after the `ui container` div (although the location doesn't really matter). We'll start off with the chat container, which simply is made up of:

- Room ID
- empty chat messages container (to be populated later via JavaScript)
- input for posting messages.

```
<!-- Chat Template -->
<script id="chat-template" type="text/x-handlebars-template">
  <h3 class="ui orange header">Room ID -> <strong>{{ room }}</strong>
  <hr>
  <div id="chat-content" class="ui feed"> </div>
  <hr>
  <div class="ui form">
    <div class="ui field">
      <label>Post Message</label>
      <textarea id="post-message" name="post-message" rows="1"></text
    </div>
    <div id="post-btn" class="ui primary submit button">Send</div>
  </div>
</script>
```

Next, add the following template, which will be used to display user chat messages:

```
<!-- Chat Content Template -->
<script id="chat-content-template" type="text/x-handlebars-template">
  {{#each messages}}
    <div class="event">
      <div class="label">
        <i class="icon blue user"></i>
      </div>
      <div class="content">
        <div class="summary">
          <a href="#"> {{ username }}</a> posted on
          <div class="date">
            {{ postedOn }}
          </div>
        </div>
      </div>
    </div>
  {{/each}}
</script>
```

```

<div class="extra text">
    {{ message }}
</div>
</div>
{{/each}}
</script>

```

Finally, add the following template, which will be used to display streams from a remote camera:

```

<!-- Remote Video Template -->
<script id="remote-video-template" type="text/x-handlebars-template">
    <div id="{{ id }}" class="four wide column"></div>
</script>

```

The markup code is hopefully pretty self-explanatory, so let's move on to writing the client-side JavaScript code for our application.

Main App Script

Open the file `public/js/app.js` and add this code:

```

// Chat platform
const chatTemplate = Handlebars.compile($('#chat-template').html());
const chatContentTemplate = Handlebars.compile($('#chat-content-template'));
const chatEl = $('#chat');
const formEl = $('.form');
const messages = [];
let username;

// Local Video
const localImageEl = $('#local-image');
const localVideoEl = $('#local-video');

// Remote Videos
const remoteVideoTemplate = Handlebars.compile($('#remote-video-template'));
const remoteVideosEl = $('#remote-videos');
let remoteVideosCount = 0;

// Add validation rules to Create/Join Room Form
formEl.form({
    fields: {
        roomName: 'empty',
        username: 'empty',
    },
});

```

```
});
```

Here we're initializing several elements that we plan to manipulate. We've also added validation rules to the form so that a user can't leave either of the fields blank.

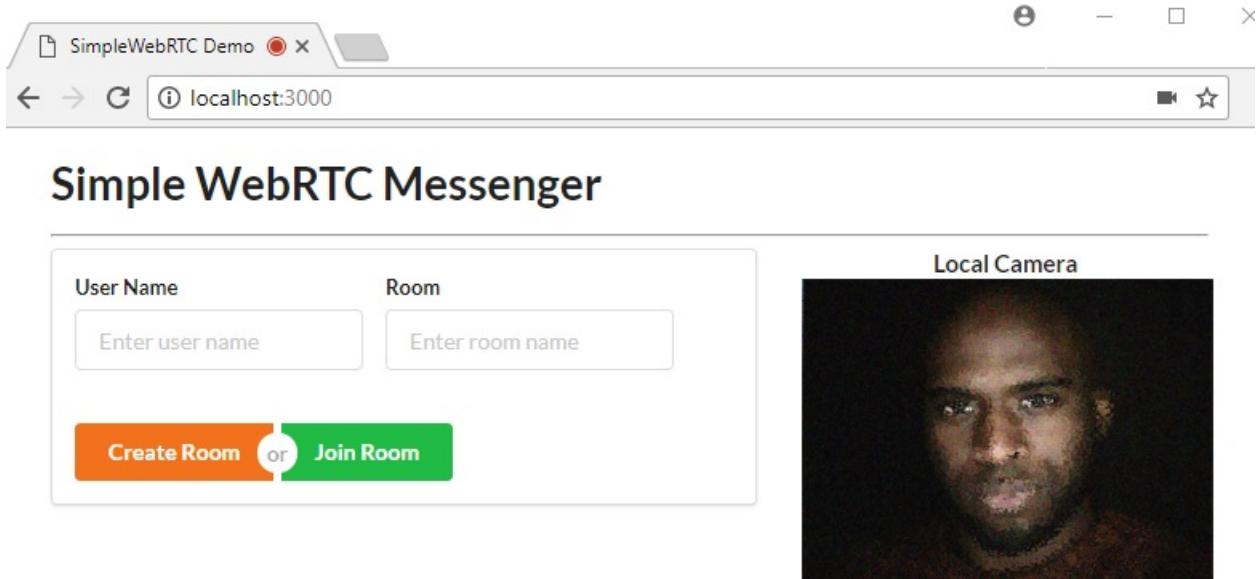
Next, let's initialize our WebRTC code:

```
// create our WebRTC connection
const webrtc = new SimpleWebRTC({
  // the id/element dom element that will hold "our" video
  localVideoEl: 'local-video',
  // the id/element dom element that will hold remote videos
  remoteVideosEl: 'remote-videos',
  // immediately ask for camera access
  autoRequestMedia: true,
});

// We got access to local camera
webrtc.on('localStream', () => {
  localImageEl.hide();
  localVideoEl.show();
});
```

Now you know why it's called SimpleWebRTC. That's all we need to do to initialize our WebRTC code. Noticed we haven't even specified any ICE servers or STUN servers. It just works. However, you can use other TURN services such as [Xirsys](#). You'll need to set up a local [SignalMaster](#) server for handling WebRTC signaling.

Let's do a quick refresh of the web page to confirm the new code is working:



The page should request access to your camera and microphone. Just click *Accept* and you should get the above view.

Chat Room Script

Now let's make the form functional. We need to write logic for creating and joining a room. In addition, we need to write additional logic for displaying the chat room. We'll use the `chat-room-template` for this. Let's start by attaching click handlers to the form's buttons:

```
$('.submit').on('click', (event) => {
  if (!formEl.form('is valid')) {
    return false;
  }
  username = $('#username').val();
  const roomId = $('#roomName').val().toLowerCase();
  if (event.target.id === 'create-btn') {
    createRoom(roomId);
  }
});
```

```

    } else {
      joinRoom(roomName);
    }
    return false;
});

```

Next, we need to declare the `createRoom` and `joinRoom` functions. Place the following code before the click handler code:

```

// Register new Chat Room
const createRoom = (roomName) => {
  console.info(`Creating new room: ${roomName}`);
  webrtc.createRoom(roomName, (err, name) => {
    showChatRoom(name);
    postMessage(`${username} created chatroom`);
  });
};

// Join existing Chat Room
const joinRoom = (roomName) => {
  console.log(`Joining Room: ${roomName}`);
  webrtc.joinRoom(roomName);
  showChatRoom(roomName);
  postMessage(`${username} joined chatroom`);
};

```

Creating or joining a room is as simple as that: just use [SimpleWebRTC's `createRoom` and `joinRoom` methods](#).

You may also have noticed that we have `showChatroom` and `postMessage` functions that we haven't defined yet. Let's do that now by inserting the following code before the calling code:

```

// Post Local Message
const postMessage = (message) => {
  const chatMessage = {
    username,
    message,
    postedOn: new Date().toLocaleString('en-GB'),
  };
  // Send to all peers
  webrtc.sendToAll('chat', chatMessage);
  // Update messages locally
  messages.push(chatMessage);
  $('#post-message').val('');
  updateChatMessages();
};

```

```
// Display Chat Interface
const showChatRoom = (room) => {
  // Hide form
  formEl.hide();
  const html = chatTemplate({ room });
  chatEl.html(html);
  const postForm = $('form');
  // Post Message Validation Rules
  postForm.form({
    message: 'empty',
  });
  $('#post-btn').on('click', () => {
    const message = $('#post-message').val();
    postMessage(message);
  });
  $('#post-message').on('keyup', (event) => {
    if (event.keyCode === 13) {
      const message = $('#post-message').val();
      postMessage(message);
    }
  });
};
```

Take some time to go through the code to understand the logic. You'll soon come across another function we haven't declared, `updateChatMessages`. Let's add it now:

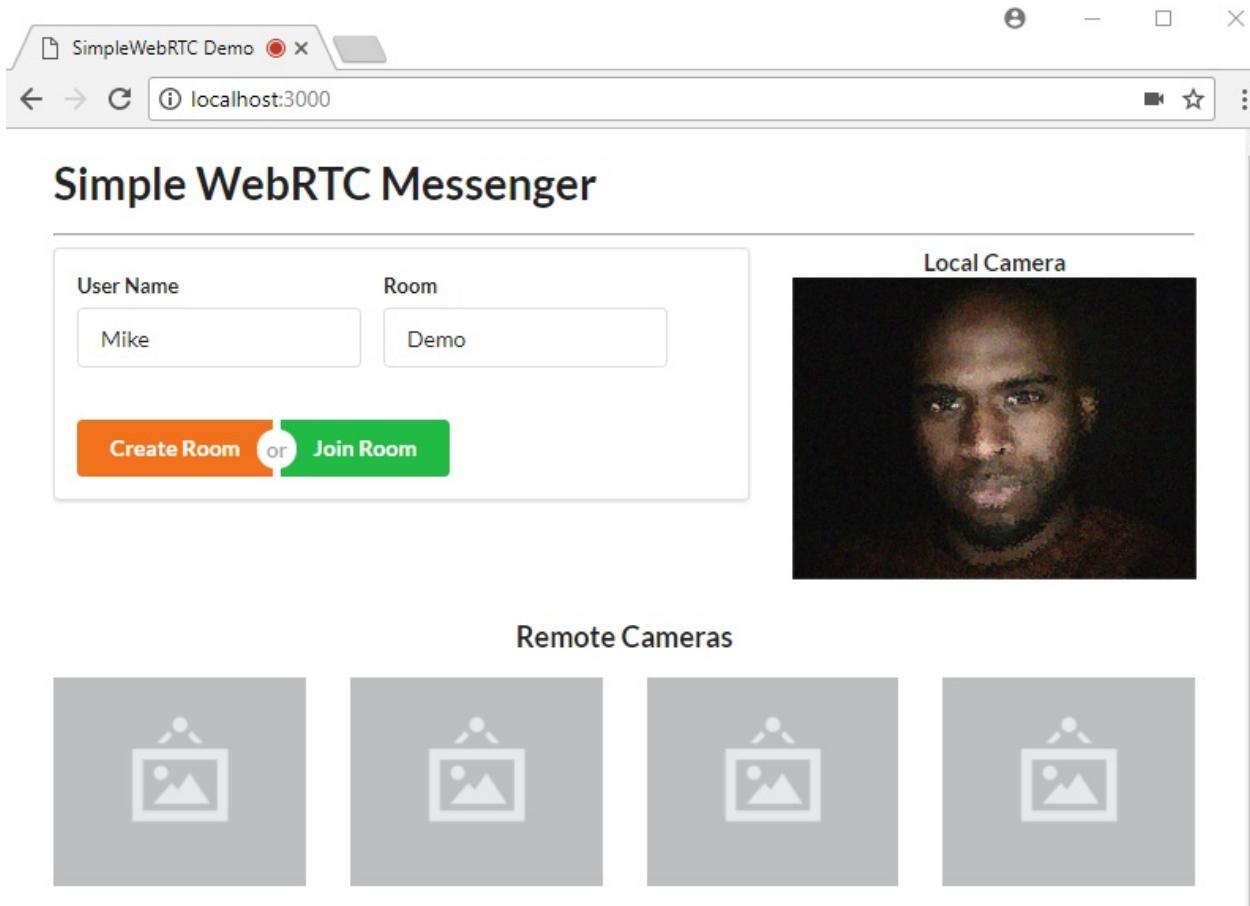
```
// Update Chat Messages
const updateChatMessages = () => {
  const html = chatContentTemplate({ messages });
  const chatContentEl = $('#chat-content');
  chatContentEl.html(html);
  // automatically scroll downwards
  const scrollHeight = chatContentEl.prop('scrollHeight');
  chatContentEl.animate({ scrollTop: scrollHeight }, 'slow');
};
```

The purpose of this function is simply to update the Chat UI with new messages. We need one more function that accepts messages from remote users. Add the following function to `app.js`:

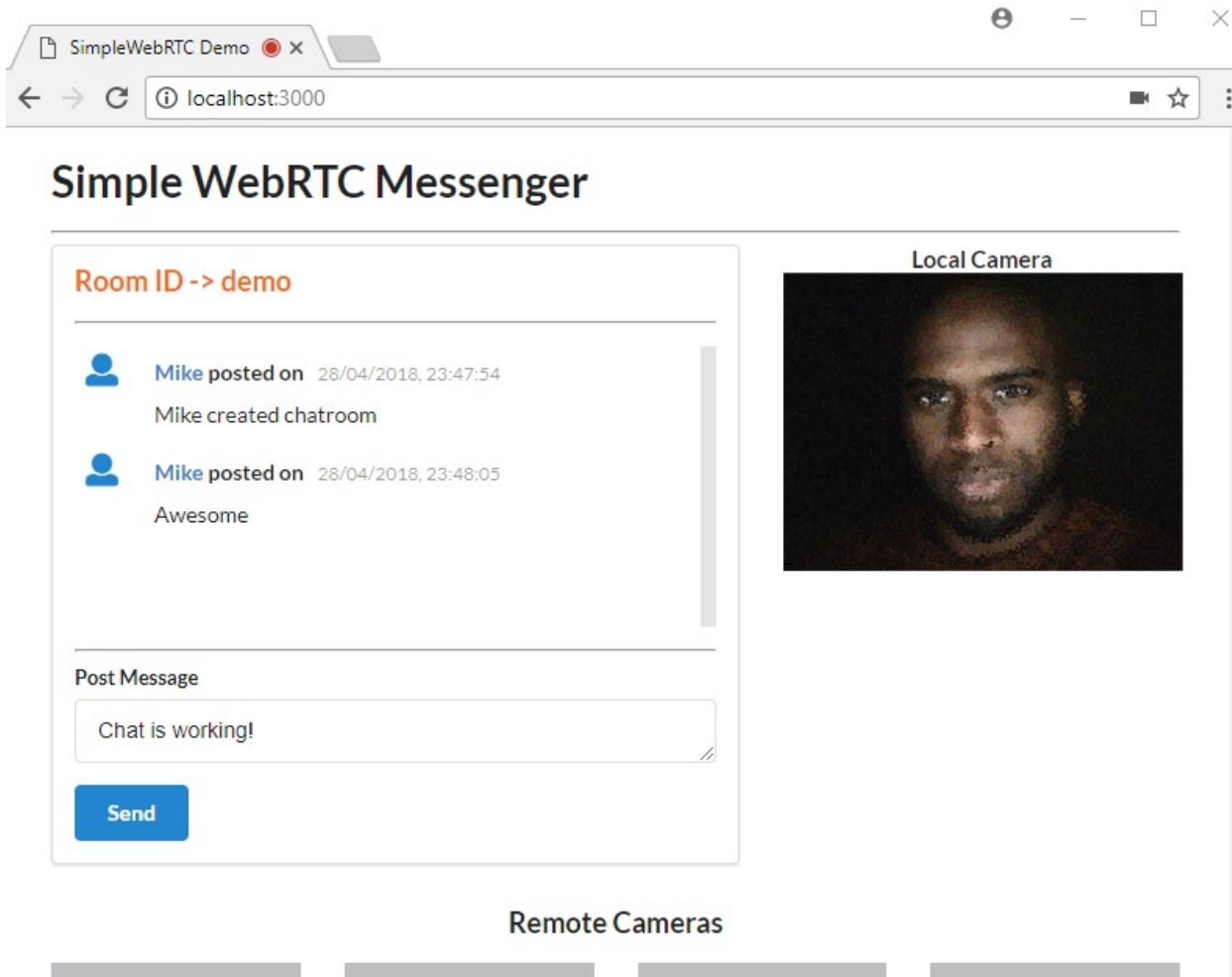
```
// Receive message from remote user
webrtc.connection.on('message', (data) => {
  if (data.type === 'chat') {
    const message = data.payload;
    messages.push(message);
```

```
        updateChatMessages();
    }
});
```

That's all the logic we need to have the chat room work. Refresh the page and log in:



Hit the *Create Room* button. You'll be taken to this view. Post some messages to confirm the chat room is working.



Once you've confirmed it's working, move on to the next task.

Remote Video Camera

As mentioned earlier, SimpleWebRTC supports multiple peers. Here's the code for adding remote video streams when a new user joins a room:

```
// Remote video was added
webrtc.on('videoAdded', (video, peer) => {
  const id = webrtc.getDomId(peer);
  const html = remoteVideoTemplate({ id });
  if (remoteVideosCount === 0) {
    remoteVideosEl.html(html);
  } else {
    remoteVideosEl.append(html);
  }
  `#${id}`).html(video);
```

```
$(`#${id} video`).addClass('ui image medium'); // Make video element
remoteVideosCount += 1;
});
```

That's it. I'm sorry if you were expecting something more complicated. What we did is simply add an event listener for `videoAdded`, the callback of which receives a `video` element that can be directly add to the DOM. It also receives a `peer` object that contains useful information about our peer connection, but in this case, we're only interested in the DOM element's ID.

Unfortunately, testing this bit of code isn't possible without running it on an HTTPS server. Theoretically, you can generate a self-signed certificate for your Express server in order to run the app within your internal network. But the bad news is that browsers won't allow you to access the webcam if the certificate isn't from a trusted authority.

The simplest solution to testing the above code is to deploy it to a public server that supports the HTTPS protocol.

Deployment

This method that we're about to perform is one of the easiest ways to deploy a NodeJS app. All we have to do is first register an account with [now.sh](#).

Simply choose the free plan. You'll need to provide your email address. You'll also need to verify your email address for your account to activate. Next, install now CLI tool on your system:

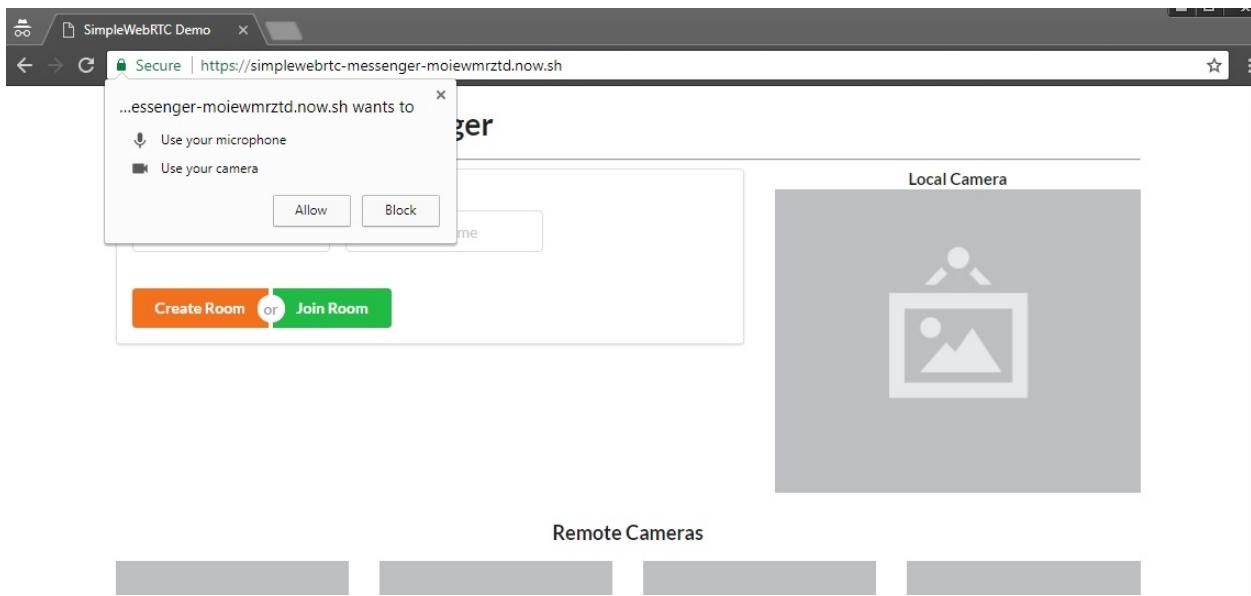
```
npm install -g now
```

After installation is complete, you can deploy the application. Simply execute the following command at the root of your project folder:

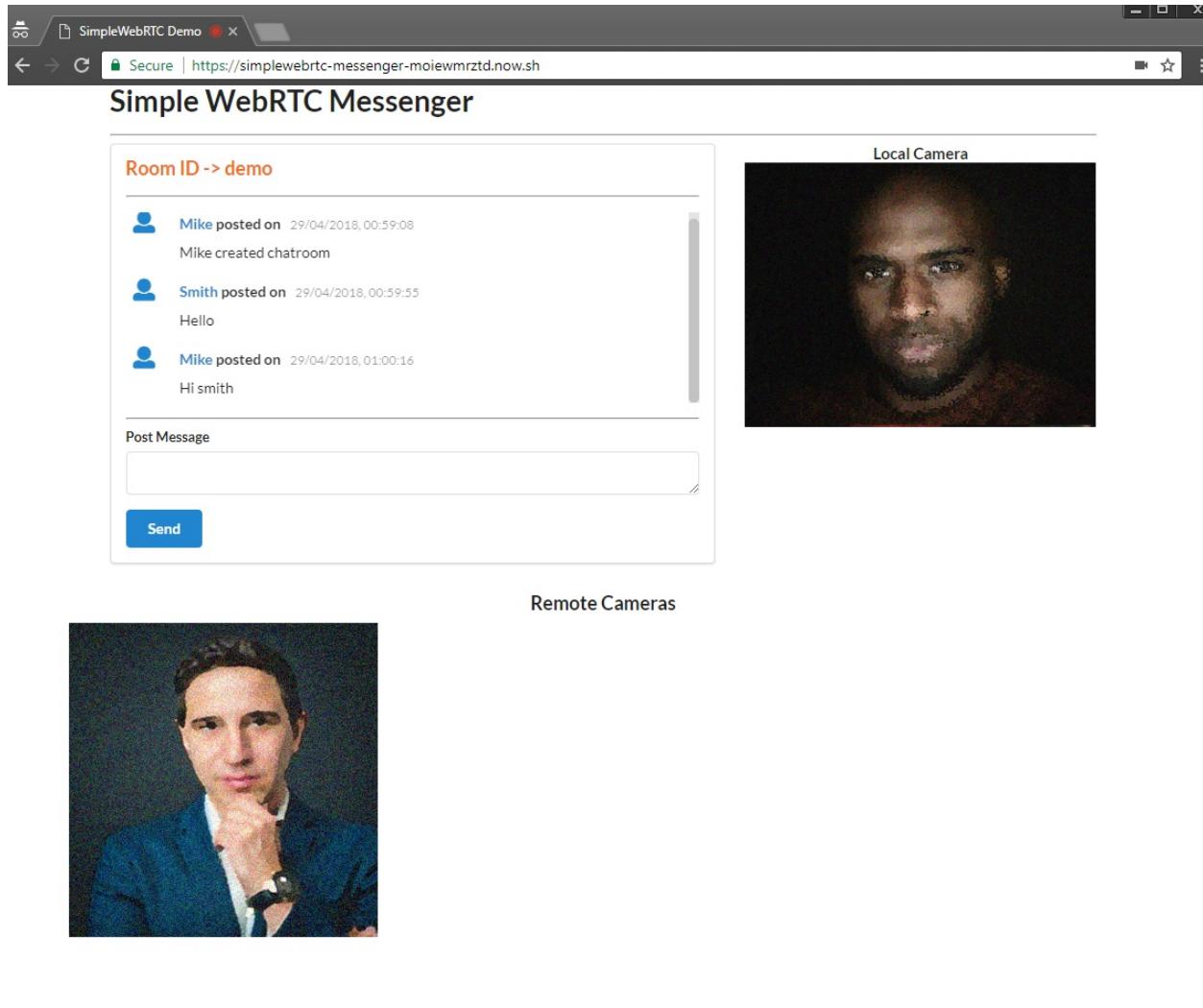
```
now --public
```

If this is the first time you're running the command, you'll be asked to enter your email address. You'll then receive an email that you'll need in order to verify your login. After verification has been done, you'll need to execute the command `now --public` again. After a few seconds, your app will be up and running at a specified URL that will be printed out on the terminal.

If you're using the VSCode integrated terminal, simply press ALT and click to open the URL in your browser.



You'll need to allow the page to access your camera and microphone. Next create a room just like before. After you've signed in, you need to access another device — such as another laptop or smartphone with a front-facing camera. You could also ask a friend with an internet connection to help you with this. Simply access the same URL, and enter a new username and the same room name. The remote user will have to hit the *Join Room* button. Within a few seconds, both devices should be connected to the chat room. If a device doesn't have a camera, that's okay, as the chat functionality will still work.



Conclusion

In this tutorial, you've learned about SimpleWebRTC and how you can use it to create real-time apps. Specifically we've created a messaging application that allows the user to send text and make a video call to a remote peer. SimpleWebRTC is a really great cross-browser library for painlessly implementing WebRTC in web applications.

Full Code

Don't forget that the code used in this tutorial is available [on GitHub](#). Clone it, make something cool, and have fun!

Chapter 3: Build a JavaScript Single Page App Without a Framework

by Michael Wanyoike

Front-end frameworks are great. They abstract away much of the complexity of building a single-page application (SPA) and help you organize your code in an intelligible manner as your project grows.

However, there's a flip side: these frameworks come with a degree of overhead and can introduce complexity of their own.

That's why, in this tutorial, we're going to learn how to build an SPA from scratch, without using a client-side JavaScript framework. This will help you evaluate what these frameworks actually do for you and at what point it makes sense to use one. It will also give you an understanding of the pieces that make up a typical SPA and how they're wired together.

Let's get started ...

Prerequisites

For this tutorial, you'll need a fundamental knowledge of [modern JavaScript](#) and [jQuery](#). Some experience using [Handlebars](#), [Express](#) and [Axios](#) will come handy, though it's not strictly necessary. You'll also need to have the following setup in your environment:

- [Node.js](#)
- [Git](#) or [Git Bash](#) for Window users.

Example Code

You can find the completed project on our [GitHub repository](#).

Building the Project

We're going to build a simple currency application that will provide the following features:

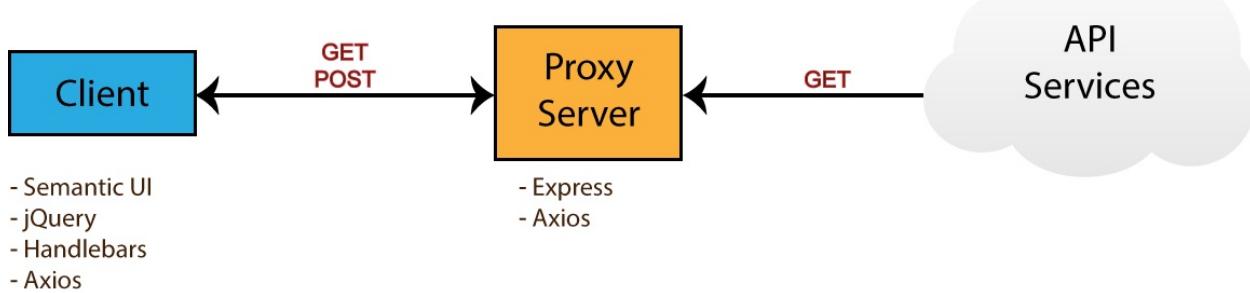
- display the latest currency rates
- convert from one currency to another
- display past currency rates based on a specified date.

We'll make use of the following free online REST APIs to implement these features:

- [fixer.io API](#)
- [Free Currency Converter API](#).

Fixer is a well-built API that provides a foreign exchange and currency conversion JSON API. Unfortunately, it's a commercial service and the free plan doesn't allow currency conversion. So we'll also need to use the Free Currency Converter API. The conversion API has a few limitations, which luckily won't affect the functionality of our application. It can be accessed directly without requiring an API key. However, Fixer requires an API key to perform any request. Simply [sign up on their website to get an access key for the free plan](#).

Ideally, we should be able to build the entire single-page application on the client side. However, since we'll be dealing with sensitive information (our API key) it won't be possible to store this in our client code. Doing so will leave our app vulnerable and open to any junior hacker to bypass the app and access data directly from our API endpoints. To protect such sensitive information, we need to put it in server code. So, we'll set up an [Express](#) server to act as a proxy between the client code and the cloud services. By using a proxy, we can safely access this key, since server code is never exposed to the browser. Below is a diagram illustrating how our completed project will work.



Take note of the npm packages that will be used by each environment — i.e. browser (client) and server. Now that you know what we'll be building, head over to the next section to start creating the project.

Project Directories and Dependencies

Head over to your workspace directory and create the folder `single-page-application`. Open the folder in VSCode or your favorite editor and create the following files and folders using the terminal:

```
touch .env .gitignore README.md server.js
mkdir public lib
mkdir public/js
touch public/index.html
touch public/js/app.js
```

Open `.gitignore` and add these lines:

```
node_modules
.env
```

Open `README.md` and add these lines:

```
# Single Page Application

This is a project demo that uses Vanilla JS to build a Single Page A
```

Next, create the package.json file by executing the following command inside the terminal:

```
npm init -y
```

You should get the following content generated for you:

```
{  
  "name": "single-page-application",  
  "version": "1.0.0",  
  "description": "This is a project demo that uses Vanilla JS to bui  
  "main": "server.js",  
  "directories": {  
    "lib": "lib"  
  },  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1",  
    "start": "node server.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC"  
}
```

See how convenient the npm command is? The content has been generated based on the project structure. Let's now install the core dependencies needed by our project. Execute the following command in your terminal:

```
npm install jquery semantic-ui-css handlebars vanilla-router express
```

After the packages have finished installing, head over to the next section to start building the base of the application.

Application Base

Before we start writing our front-end code, we need to implement a server-client base to work from. That means a basic HTML view being served from an Express server. For performance and reliability reasons, we'll inject front-end dependencies straight from the node_modules folder. We'll have to set up our Express server in a special way to make this work. Open `server.js` and add the following:

```
require('dotenv').config(); // read .env files  
const express = require('express');  
  
const app = express();  
const port = process.env.PORT || 3000;  
  
// Set public folder as root
```

```

app.use(express.static('public'));

// Allow front-end access to node_modules folder
app.use('/scripts', express.static(`$__dirname__/node_modules/`));

// Listen for HTTP requests on port 3000
app.listen(port, () => {
  console.log('listening on %d', port);
});

```

This gives us a basic Express server. I've commented the code, so hopefully this gives you a fairly good idea of what's going on. Next, open public/index.html and enter:

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="scripts/semantic-ui-css/semantic.min.
  <title>SPA Demo</title>
</head>
<body>
  <div class="ui container">
    <!-- Navigation Menu -->
    <div class="ui four item inverted orange menu">
      <div class="header item">
        <i class="money bill alternate outline icon"></i>
        Single Page App
      </div>
      <a class="item" href="/">
        Currency Rates
      </a>
      <a class="item" href="/exchange">
        Exchange Rates
      </a>
      <a class="item" href="/historical">
        Historical Rates
      </a>
    </div>

    <!-- Application Root -->
    <div id="app"></div>
  </div>

  <!-- JS Library Dependencies -->
  <script src="scripts/jquery/dist/jquery.min.js"></script>

```

```
<script src="scripts/semantic-ui-css/semantic.min.js"></script>
<script src="scripts/axios/dist/axios.min.js"></script>
<script src="scripts/handlebars/dist/handlebars.min.js"></script>
<script src="scripts/vanilla-router/dist/vanilla-router.min.js"></script>
<script src="js/app.js"></script>
</body>
</html>
```

We're using Semantic UI for styling. Please refer to the [Semantic UI Menu](#) documentation to understand the code used for our navigation bar. Go to your terminal and start the server:

```
npm start
```

Open localhost:3000 in your browser. You should have a blank page with only the navigation bar showing:



Let's now write some view templates for our app.

Front-end Skeleton Templates

We'll use [Handlebars](#) to write our templates. JavaScript will be used to render the templates based on the current URL. The first template we'll create will be for displaying error messages such as 404 or server errors. Place this code in `public/index.html` right after the the navigation section:

```
<!-- Error Template -->
<script id="error-template" type="text/x-handlebars-template">
  <div class="ui {{color}} inverted segment" style="height:250px;">
    <br>
    <h2 class="ui center aligned icon header">
      <i class="exclamation triangle icon"></i>
      <div class="content">
        {{title}}
        <div class="sub header">{{message}}</div>
      </div>
    </h2>
  </div>
</script>
```

Next, add the following templates that will represent a view for each URL path

we specified in the navigation bar:

```
<!-- Currency Rates Template -->
<script id="rates-template" type="text/x-handlebars-template">
  <h1 class="ui header">Currency Rates</h1>
  <hr>
</script>

<!-- Exchange Conversion Template -->
<script id="exchange-template" type="text/x-handlebars-template">
  <h1 class="ui header">Exchange Conversion</h1>
  <hr>
</script>

<!-- Historical Rates Template -->
<script id="historical-template" type="text/x-handlebars-template">
  <h1 class="ui header">Historical Rates</h1>
  <hr>
</script>
```

Next, let's compile all thesees templates in `public/js/app.js`. After compilation, we'll render the `rates-template` and see what it looks like:

```
window.addEventListener('load', () => {
  const el = $('#app');

  // Compile Handlebar Templates
  const errorTemplate = Handlebars.compile($('#error-template').html)
  const ratesTemplate = Handlebars.compile($('#rates-template').html)
  const exchangeTemplate = Handlebars.compile($('#exchange-template'))
  const historicalTemplate = Handlebars.compile($('#historical-template'))

  const html = ratesTemplate();
  el.html(html);
});
```

Take note that we're wrapping all JavaScript client code inside a `load` event. This is just to make sure that all dependencies have been loaded and that the DOM has completed loading. Refresh the page and see what we have:



We're making progress. Now, if you click the other links, except *Currency Rates*, the browser will try to fetch a new page and end up with a message like this: Cannot GET /exchange.

We're building a single page application, which means all the action should happen in one page. We need a way to tell the browser to stop fetching new pages whenever the URL changes.

Client-side Routing

To control routing within the browser environment, we need to implement client-side routing. There are many client-side routing libraries that can help out with this. For our project, we'll use [vanilla router](#), which is a very easy-to-use routing package.

If you recall, we had earlier included all the JavaScript libraries we need in `index.html`. Hence we can call the `Router` class right away. Remove the last two statements you added to `app.js` and replace them with this code:

```
// Router Declaration
const router = new Router({
  mode: 'history',
  page404: (path) => {
    const html = errorTemplate({
      color: 'yellow',
      title: 'Error 404 - Page NOT Found!',
      message: `The path '/${path}' does not exist on this site`,
    });
    el.html(html);
  },
});

router.add('/', () => {
  let html = ratesTemplate();
  el.html(html);
});

router.add('/exchange', () => {
  let html = exchangeTemplate();
  el.html(html);
});

router.add('/historical', () => {
  let html = historicalTemplate();
```

```

    el.html(html);
});

// Navigate app to current url
router.navigateTo(window.location.pathname);

// Highlight Active Menu on Refresh/Page Reload
const link = $(`a[href=${window.location.pathname}]`);
link.addClass('active');

$('a').on('click', (event) => {
  // Block browser page load
  event.preventDefault();

  // Highlight Active Menu on Click
  const target = $(event.target);
  $('.item').removeClass('active');
  target.addClass('active');

  // Navigate to clicked url
  const href = target.attr('href');
  const path = href.substr(href.lastIndexOf('/'));
  router.navigateTo(path);
});

```

Take some time to go through the code. I've added comments in various sections to explain what's happening. You'll notice that, in the router's declaration, we've specified the `page404` property to use the error template. Let's now test the links:



The links should now work. But we have a problem. Click either the `/exchange` or `historical` link, then refresh the browser. We get the same error as before — `Cannot GET /exchange`. To fix this, head over to `server.js` and add this statement right before the `listen` code:

```
// Redirect all traffic to index.html
app.use((req, res) => res.sendFile(`__dirname}/public/index.html`)
```

You'll have to restart the the server using `Ctrl + c` and executing `npm start`. Go back to the browser and try to refresh. You should now see the page render correctly. Now, let's try entering a non-existent path in the URL like

/exchanges. The app should display a 404 error message:

Single Page App Currency Rates Exchange Conversion Historical Rates



Error 404 - Page NOT Found!

The path '/exchanges' does not exist on this site

We've now implemented the necessary code to create our single-page-app skeleton. Let's now start working on listing the latest currency rates.

Latest Currency Rates

For this task, we'll make use of the [Fixer Latest Rates Endpoint](#). Open the `.env` file and add your API key. We'll also specify the timeout period and the symbols we'll list on our page. Feel free to increase the timeout value if you have a slower internet connection:

```
API_KEY=<paste key here>
PORT=3000
TIMEOUT=5000
SYMBOLS=EUR, USD, GBP, AUD, BTC, KES, JPY, CNY
```

Next create the file `lib/fixer-service.js`. This is where we'll write helper code for our Express server to easily request information from Fixer. Copy the following code:

```
require('dotenv').config();
const axios = require('axios');

const symbols = process.env.SYMBOLS || 'EUR,USD,GBP';

// Axios Client declaration
const api = axios.create({
  baseURL: 'http://data.fixer.io/api',
```

```

params: {
  access_key: process.env.API_KEY,
},
timeout: process.env.TIMEOUT || 5000,
});

// Generic GET request function
const get = async (url) => {
  const response = await api.get(url);
  const { data } = response;
  if (data.success) {
    return data;
  }
  throw new Error(data.error.type);
};

module.exports = {
  getRates: () => get(`/latest&symbols=${symbols}&base=EUR`),
};

```

Again, take some time to go through the code to understand what's happening. If you're unsure, you can also check out the documentation for [dotenv](#), [axios](#) and read up on [module exports](#). Let's now do a quick test to confirm the `getRates()` function is working.

Open `server.js` and add this code:

```

const { getRates } = require('./lib/fixer-service');

...
// Place this block at the bottom
const test = async() => {
  const data = await getRates();
  console.log(data);
}

test();

```

Run `npm start` or `node server`. After a few seconds, you should get the following output:

```
{
  success: true,
  timestamp: 1523871848,
  base: 'EUR',
  date: '2018-04-16',
  rates: {

```

```

    EUR: 1,
    USD: 1.23732,
    GBP: 0.865158,
    AUD: 1.59169,
    BTC: 0.000153,
    KES: 124.226892,
    JPY: 132.608498,
    CNY: 7.775567
}
}

```

If you get something similar to the above, it means the code is working. The values will of course be different, since the rates change every day. Now comment out the test block and insert this code right before the statement that redirects all traffic to `index.html`:

```

// Express Error handler
const errorHandler = (err, req, res) => {
  if (err.response) {
    // The request was made and the server responded with a status code
    // that falls out of the range of 2xx
    res.status(403).send({ title: 'Server responded with an error',
  } else if (err.request) {
    // The request was made but no response was received
    res.status(503).send({ title: 'Unable to communicate with server',
  } else {
    // Something happened in setting up the request that triggered an error
    res.status(500).send({ title: 'An unexpected error occurred', message: err.message
  }
};

// Fetch Latest Currency Rates
app.get('/api/rates', async (req, res) => {
  try {
    const data = await getRates();
    res.setHeader('Content-Type', 'application/json');
    res.send(data);
  } catch (error) {
    errorHandler(error, req, res);
  }
});

```

As we can see, there's a custom error handler function that's designed to handle different error scenarios, which can occur during execution of server code. When an error occurs, an error message is constructed and sent back to the client.

Let's confirm this bit of code is working. Restart the Express server and navigate

your browser to this URL: localhost:3000/api/rates. You should see the same JSON result that was displayed in the console. We can now implement a view that will display this information in a neat, elegant table.

Open public/index.html and replace the rates-template with this code:

```
<!-- Currency Rates Template -->
<script id="rates-template" type="text/x-handlebars-template">
  <h1 class="ui header">Currency Rates</h1>
  <hr>
  <div class="ui loading basic segment">
    <div class="ui horizontal list">
      <div class="item">
        <i class="calendar alternate outline icon"></i>
        <div class="content">
          <div class="ui sub header">Date</div>
          <span>{{date}}</span>
        </div>
      </div>
      <div class="item">
        <i class="money bill alternate outline icon"></i>
        <div class="content">
          <div class="ui sub header">Base</div>
          <span>{{base}}</span>
        </div>
      </div>
    </div>
  </div>

  <table class="ui celled striped selectable inverted table">
    <thead>
      <tr>
        <th>Code</th>
        <th>Rate</th>
      </tr>
    </thead>
    <tbody>
      {{#each rates}}
        <tr>
          <td>{{@key}}</td>
          <td>{{this}}</td>
        </tr>
      {{/each}}
    </tbody>
  </table>
</div>
</script>
```

Remember we're using Semantic UI to provide us with styling. I'd like you to

pay close attention to the [Segment loading](#) component. This will be an indication to let users know that something is happening as the app fetches the data. We're also using the [Table UI](#) to display the rates. Please go through the linked documentation if you're new to Semantic.

Now let's update our code in `public/js/app.js` to make use of this new template. Replace the first `route.add('/')` function with this code:

```
// Instantiate api handler
const api = axios.create({
  baseURL: 'http://localhost:3000/api',
  timeout: 5000,
});

// Display Error Banner
const showError = (error) => {
  const { title, message } = error.response.data;
  const html = errorTemplate({ color: 'red', title, message });
  el.html(html);
};

// Display Latest Currency Rates
router.add('/', async () => {
  // Display loader first
  let html = ratesTemplate();
  el.html(html);
  try {
    // Load Currency Rates
    const response = await api.get('/rates');
    const { base, date, rates } = response.data;
    // Display Rates Table
    html = ratesTemplate({ base, date, rates });
    el.html(html);
  } catch (error) {
    showError(error);
  } finally {
    // Remove loader status
    $('.loading').removeClass('loading');
  }
});
```

The first code block instantiates an API client for communicating with our proxy server. The second block is a global function for handling errors. Its work is simply to display an error banner in case something goes wrong on the server side. The third block is where we get rates data from the `localhost:3000/api/rates` endpoint and pass it to the `rates-template` to

display the information.

Simply refresh the browser. You should now have the following view:

The screenshot shows a user interface for a currency rates application. At the top, there is a navigation bar with four items: "Single Page App" (highlighted in orange), "Currency Rates", "Exchange Conversion", and "Historical Rates". Below the navigation bar, the title "Currency Rates" is displayed. Underneath the title, there are two small status indicators: "DATE 2018-04-16" and "BASE EUR". The main content is a table with two columns: "Code" and "Rate". The table lists the following data:

Code	Rate
EUR	1
USD	1.236554
GBP	0.864524
AUD	1.590827
BTC	0.000153
KES	124.149993
JPY	132.603023
CNY	7.769511

Next we'll build an interface for converting currencies.

Exchange Conversion

For the currency conversion, we'll use two endpoints:

- [Fixer's Symbols Endpoint](#)
- [Free Currency Converter Endpoint.](#)

We need the symbols endpoint to get a list of supported currency codes. We'll use this data to populate the dropdowns that the users will use to select which currencies to convert. Open `lib/fixer-service.js` and add this line right after the `getRates()` function:

```
getSymbols: () => get('/symbols'),
```

Create another helper file, `lib/free-currency-service.js`, and add the following code:

```
require('dotenv').config();
```

```

const axios = require('axios');

const api = axios.create({
  baseURL: 'https://free.currencyconverterapi.com/api/v5',
  timeout: process.env.TIMEOUT || 5000,
});

module.exports = {
  convertCurrency: async (from, to) => {
    const response = await api.get(`convert?q=${from}_${to}&compact`);
    const key = Object.keys(response.data)[0];
    const { val } = response.data[key];
    return { rate: val };
  },
};

```

This will help us get the conversion rate from one currency to another for free. In the client code, we'll have to calculate the conversion amount by multiplying amount by rate. Now let's add these two service methods to our Express server code. Open `server.js` and update accordingly:

```

const { getRates, getSymbols, } = require('./lib/fixer-service');
const { convertCurrency } = require('./lib/free-currency-service');
...
// Insert right after get '/api/rates', just before the redirect sta

// Fetch Symbols
app.get('/api/symbols', async (req, res) => {
  try {
    const data = await getSymbols();
    res.setHeader('Content-Type', 'application/json');
    res.send(data);
  } catch (error) {
    errorHandler(error, req, res);
  }
});

// Convert Currency
app.post('/api/convert', async (req, res) => {
  try {
    const { from, to } = req.body;
    const data = await convertCurrency(from, to);
    res.setHeader('Content-Type', 'application/json');
    res.send(data);
  } catch (error) {
    errorHandler(error, req, res);
  }
});

```

Now our proxy server should be able to get symbols and conversion rates. Take note that /api/convert is a POST method. We'll use a form on the client side to build the currency conversion UI. Feel free to use the test function to confirm both endpoints are working. Here's an example:

```
// Test Symbols Endpoint
const test = async() => {
  const data = await getSymbols();
  console.log(data);
}

// Test Currency Conversion Endpoint
const test = async() => {
  const data = await convertCurrency('USD', 'KES');
  console.log(data);
}
```

You'll have to restart the server for each test. Remember to comment out the tests once you've confirmed the code is working so far. Let's now work on our currency conversion UI. Open public/index.html and update the exchange-template by replacing the existing code with this:

```
<script id="exchange-template" type="text/x-handlebars-template">
  <h1 class="ui header">Exchange Rate</h1>
  <hr>
  <div class="ui basic loading segment">
    <form class="ui form">
      <div class="three fields">
        <div class="field">
          <label>From</label>
          <select class="ui dropdown" name="from" id="from">
            <option value="">Select Currency</option>
            {{#each symbols}}
              <option value="{{@key}}>{{this}}</option>
            {{/each}}
          </select>
        </div>
        <div class="field">
          <label>To</label>
          <select class="ui dropdown" name="to" id="to">
            <option value="">Select Currency</option>
            {{#each symbols}}
              <option value="{{@key}}>{{this}}</option>
            {{/each}}
          </select>
        </div>
        <div class="field">
```

```

        <label>Amount</label>
        <input type="number" name="amount" id="amount" placeholder="0.00">
    </div>
    </div>
    <div class="ui primary submit button">Convert</div>
    <div class="ui error message"></div>
</form>
<br>
<div id="result-segment" class="ui center aligned segment">
    <h2 id="result" class="ui header">
        0.00
    </h2>
</div>
</div>
</script>

```

Take your time to go through the script and understand what's happening. We're using [Semantic UI Form](#) to build the interface. We're also using Handlebars notation to populate the dropdown boxes. Below is the JSON format used by Fixer's Symbols endpoint:

```
{
  "success": true,
  "symbols": {
    "AED": "United Arab Emirates Dirham",
    "AFN": "Afghan Afghani",
    "ALL": "Albanian Lek",
    "AMD": "Armenian Dram",
  }
}
```

Take note that the symbols data is in map format. That means the information is stored as key {{@key}} and value {{this}} pairs. Let's now update public/js/app.js and make it work with the new template. Open the file and replace the existing route code for /exchange with the following:

```

// Perform POST request, calculate and display conversion results
const getConversionResults = async () => {
  // Extract form data
  const from = $('#from').val();
  const to = $('#to').val();
  const amount = $('#amount').val();
  // Send post data to Express(proxy) server
  try {
    const response = await api.post('/convert', { from, to });
    const { rate } = response.data;
    const result = rate * amount;
  }
}

```

```

        $('#result').html(`#${to} ${result}`);
    } catch (error) {
        showError(error);
    } finally {
        $('#result-segment').removeClass('loading');
    }
};

// Handle Convert Button Click Event
const convertRatesHandler = () => {
    if ($('#ui.form').form('is valid')) {
        // hide error message
        $('.ui.error.message').hide();
        // Post to Express server
        $('#result-segment').addClass('loading');
        getConversionResults();
        // Prevent page from submitting to server
        return false;
    }
    return true;
};

router.add('/exchange', async () => {
    // Display loader first
    let html = exchangeTemplate();
    el.html(html);
    try {
        // Load Symbols
        const response = await api.get('/symbols');
        const { symbols } = response.data;
        html = exchangeTemplate({ symbols });
        el.html(html);
        $('.loading').removeClass('loading');
        // Validate Form Inputs
        $('#ui.form').form({
            fields: {
                from: 'empty',
                to: 'empty',
                amount: 'decimal',
            },
        });
        // Specify Submit Handler
        $('#submit').click(convertRatesHandler);
    } catch (error) {
        showError(error);
    }
});

```

Refresh the page. You should now have the following view:

From: United States Dollar

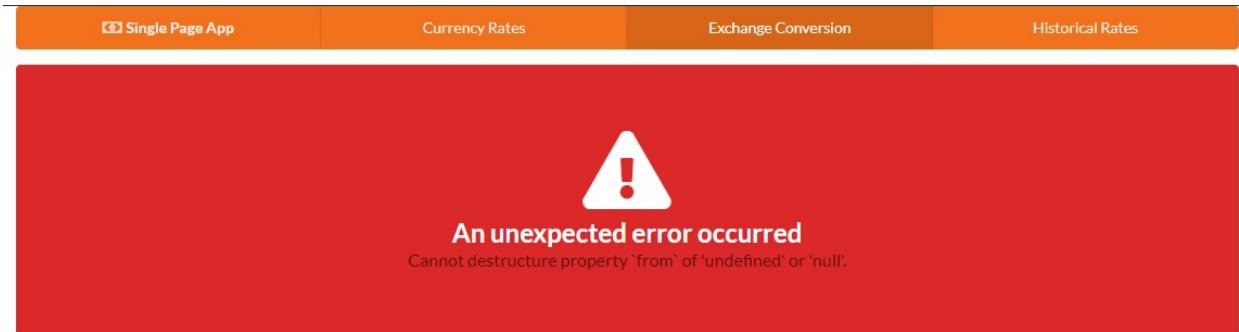
To: Kenyan Shilling

Amount: 2500

Convert

0.00

Select some currencies of your choosing and enter an amount. Then hit the *Convert* button:



Oops! We just hit an error scenario. At least we know our error handling code is working. To figure out why the error is occurring, go back to the server code and look at the `/api/convert` function. Specifically, look at the line that says `const { from, to } = req.body;`.

It seems Express is unable to read properties from the `request` object. To fix this, we need to install middleware that can help out with this:

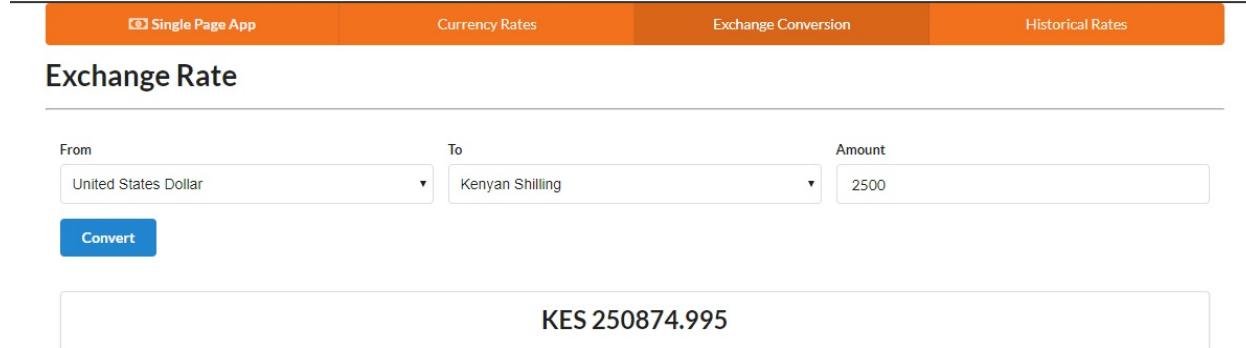
```
npm install body-parser
```

Next, update the server code as follows:

```
const bodyParser = require('body-parser');
...
/** Place this code right before the error handler function **/
// Parse POST data as URL encoded data
app.use(bodyParser.urlencoded({
```

```
extended: true,  
}));  
  
// Parse POST data as JSON  
app.use(bodyParser.json());
```

Start the server again and refresh the browser. Try doing another conversion. It should now work.



The screenshot shows a user interface for currency conversion. At the top, there are four tabs: "Single Page App" (which is active), "Currency Rates", "Exchange Conversion" (the current view), and "Historical Rates". Below the tabs, the title "Exchange Rate" is displayed. The conversion form has three dropdown menus: "From" (set to "United States Dollar"), "To" (set to "Kenyan Shilling"), and "Amount" (set to "2500"). A blue "Convert" button is located below the dropdowns. To the right of the "Convert" button, the result "KES 250874.995" is displayed in a large, bold font.

Let's now focus on the final bit — historical currency rates. Let's start with the views.

Historical Currency Rates

Implementing this feature will be like combining the tasks from the first and second pages. We're going to build a tiny form where the user will be expected to input a date. When the user clicks submit, the currency rates for the specified date will be displayed in table format. We'll use the [Historical Rates Endpoint](#) from Fixer API to achieve this. The API request looks like this:

```
https://data.fixer.io/api/2013-12-24  
? access_key = API_KEY  
& base = GBP  
& symbols = USD, CAD, EUR
```

And the response will look like this:

```
{  
  "success": true,  
  "historical": true,  
  "date": "2013-12-24",
```

```
"timestamp": 1387929599,  
"base": "GBP",  
"rates": {  
    "USD": 1.636492,  
    "EUR": 1.196476,  
    "CAD": 1.739516  
}  
}
```

Open `lib/fixer-service.js` and the Historical Rates Endpoint like this:

```
...  
/** Place right after getSymbols **/  
getHistoricalRate: date => get(`/${date}&symbols=${symbols}&base=E  
...  
...
```

Open `server.js` and add this code:

```
...  
const { getRates, getSymbols, getHistoricalRate } = require('./lib/f  
...  
/** Place this after '/api/convert' post function **/  
  
// Fetch Currency Rates by date  
app.post('/api/historical', async (req, res) => {  
    try {  
        const { date } = req.body;  
        const data = await getHistoricalRate(date);  
        res.setHeader('Content-Type', 'application/json');  
        res.send(data);  
    } catch (error) {  
        errorHandler(error, req, res);  
    }  
});  
...  
...
```

If you're in any doubt as to how the code is arranged, please refer to the complete `server.js` file on [GitHub](#). Feel free to write a quick test to confirm the historical endpoint is working:

```
const test = async() => {  
    const data = await getHistoricalRate('2012-07-14');  
    console.log(data);  
}  
  
test();
```

Do remember to comment out the test block once you confirm everything's working. Now let's now work on the client code.

Open `index.html`. Delete the existing `historical-template` we used as a placeholder, and replace it with the following:

```
<script id="historical-template" type="text/x-handlebars-template">
  <h1 class="ui header">Historical Rates</h1>
  <hr>
  <form class="ui form">
    <div class="field">
      <label>Pick Date</label>
      <div class="ui calendar" id="calendar">
        <div class="ui input left icon">
          <i class="calendar icon"></i>
          <input type="text" placeholder="Date" id="date">
        </div>
      </div>
    </div>
    <div class="ui primary submit button">Fetch Rates</div>
    <div class="ui error message"></div>
  </form>

  <div class="ui basic segment">
    <div id="historical-table"></div>
  </div>
</script>
```

Take a look at the form first. One thing I'd like to point out is that Semantic UI doesn't officially have a date input. However, thanks to [Michael de Hoog](#)'s contribution, we have the [Semantic-UI-Calendar](#) module available to us. Simply install it using npm:

```
npm install semantic-ui-calendar
```

Go back to `public/index.html` and include it within the scripts section:

```
...
<script src="scripts/semantic-ui-css/semantic.min.js"></script>
<script src="scripts/semantic-ui-calendar/dist/calendar.min.js"></script>
...</pre>
```

To display the historical rates, we'll simply reuse the `rates-template`. Next open `public/js/app.js` and update the existing route code for `/historical`:

```
const getHistoricalRates = async () => {
```

```

const date = $('#date').val();
try {
  const response = await api.post('/historical', { date });
  const { base, rates } = response.data;
  const html = ratesTemplate({ base, date, rates });
  $('#historical-table').html(html);
} catch (error) {
  showError(error);
} finally {
  $('.segment').removeClass('loading');
}
};

const historicalRatesHandler = () => {
  if ($.ui.form().form('is valid')) {
    // hide error message
    $('.ui.error.message').hide();
    // Indicate loading status
    $('.segment').addClass('loading');
    getHistoricalRates();
    // Prevent page from submitting to server
    return false;
  }
  return true;
};

router.add('/historical', () => {
  // Display form
  const html = historicalTemplate();
  el.html(html);
  // Activate Date Picker
  $('#calendar').calendar({
    type: 'date',
    formatter: { //format date to yyyy-mm-dd
      date: date => new Date(date).toISOString().split('T')[0],
    },
  });
  // Validate Date input
  $('.ui.form').form({
    fields: {
      date: 'empty',
    },
  });
  $('.submit').click(historicalRatesHandler);
});

```

Once again, take time to read the comments and understand the code and what it's doing. Then restart the server, refresh the browser and navigate to the /historical path. Pick any date before the year 1999 then click *Fetch Rates*.

You should have something like this:

Single Page App Currency Rates Exchange Conversion Historical Rates

Historical Rates

Pick Date
2015-02-09

Fetch Rates

Currency Rates

DATE 2015-02-09 BASE EUR

Code	Rate
EUR	1
USD	1.132543
GBP	0.744364
AUD	1.452675
BTC	0.005127
KES	103.663641
JPY	134.428049
CNY	7.054123

If you pick a date before the year 1999 or a date in the future, an error banner will be displayed when you submit the form.

Single Page App Currency Rates Exchange Conversion Historical Rates



An unexpected error occurred
no_rates_available

Summary

Now that we've come to the end of the tutorial, you should see that it's not that difficult to build a single-page application powered by REST APIs without using a framework. But there are a few things we should be concerned with:

- **DOM Performance.** In our client-side code, we're directly manipulating the DOM. This can soon get out of hand as the project grows, causing the UI to become sluggish.
- **Browser Performance.** There are quite a number of front-end libraries that we've loaded as scripts in `index.html`, which is okay for development purposes. For production deployment, we need a system for bundling all scripts such that the browsers use a single request for loading the necessary JavaScript resources.
- **Monolithic Code.** For the server code, it's easier to break down code into modular parts since it runs within a Node environment. However, for client-side code, it's not easy to organize in modules unless you use a bundler like [webpack](#).
- **Testing.** So far we've been doing manual testing. For a production-ready application, we need to set up a testing framework like Jasmine, Mocha or Chai to automate this work. This will help prevent recurring errors.

These are just a few of the many issues you'll face when you approach project development without using a framework. Using something such as Angular, React or Vue will help you alleviate a lot of these concerns. I hope this tutorial has been helpful and that it will aid you in your journey to becoming a professional JavaScript developer.

Chapter 4: Build a To-do List with Hyperapp, the 1KB JS Micro-framework

by Darren Jones

In this tutorial, we'll be using Hyperapp to build a to-do list app. If you want to learn functional programming principles, but not get bogged down in details, read on.

Hyperapp is hot right now. It recently surpassed 11,000 stars on GitHub and made the 5th place in the Front-end Framework section of the [2017 JavaScript Rising Stars](#). It was also featured [on SitePoint](#) recently, when it hit version 1.0.

The reason for Hyperapp's popularity can be attributed to its pragmatism and ultralight size (1.4 kB), while at the same time achieving results similar to React and Redux out of the box.

So, What Is HyperApp?

Hyperapp allows you to build dynamic, single-page web apps by taking advantage of a virtual DOM to update the elements on a web page quickly and efficiently in a similar way to React. It also uses a single object that's responsible for keeping track of the application's state, just like Redux. This makes it easier to manage the state of the app and make sure that different elements don't get out of sync with each other. The main influence behind Hyperapp was the [Elm architecture](#).

At its core, Hyperapp has three main parts:

- **State.** This is a single object tree that stores all of the information about the application.
- **Actions.** These are methods that are used to change and update the values in the state object.

- **View.** This is a function that returns virtual node objects that compile to HTML code. It can use JSX or a similar templating language and has access to the state and actions objects.

These three parts interact with each other to produce a dynamic application. Actions are triggered by events on the page. The action then updates the state, which then triggers an update to the view. These changes are made to the Virtual DOM, which Hyperapp uses to update the actual DOM on the web page.

Getting Started

To get started as quickly as possible, we're going to use CodePen to develop our app. You need to make sure that the JavaScript preprocessor is set to *Babel* and the Hyperapp package is loaded as an external resource using the following link:

```
https://unpkg.com/hyperapp
```

To use Hyperapp, we need to import the `app` function as well as the `h` method, which Hyperapp uses to create VDOM nodes. Add the following code to the JavaScript pane in CodePen:

```
const { h, app } = hyperapp;
```

We'll be using JSX for the view code. To make sure Hyperapp knows this, we need to add the following comment to the code:

```
/** @jsx h */
```

The `app()` method is used to initialize the application:

```
const main = app(state, actions, view, document.body);
```

This takes the `state` and `actions` objects as its first two parameters, the `view()` function as its third parameter, and the last parameter is the HTML element where the application is to be inserted into your markup. By convention, this is usually the `<body>` tag, represented by `document.body`.

To make it easy to get started, I've created a boilerplate Hyperapp code template on CodePen that contains all the elements mentioned above. It can be forked by [clicking on this link](#).

Hello Hyperapp!

Let's have a play around with Hyperapp and see how it all works. The `view()` function accepts the `state` and `actions` objects as arguments and returns a Virtual DOM object. We're going to use JSX, which means we can write code that looks a lot more like HTML. Here's an example that will return a heading:

```
const view = (state, actions) => (
  <h1>Hello Hyperapp!</h1>
);
```

This will actually return the following VDOM object:

```
{
  name: "h1",
  props: {},
  children: "Hello Hyperapp!"
}
```

The `view()` function is called every time the `state` object changes. Hyperapp will then build a new Virtual DOM tree based on any changes that have occurred. Hyperapp will then take care of updating the actual web page in the most efficient way by comparing the differences in the new Virtual DOM with the old one stored in memory.

Components

Components are pure functions that return virtual nodes. They can be used to create reusable blocks of code that can then be inserted into the view. They can accept parameters in the usual way that any function can, but they don't have access to the `state` and `actions` objects in the same way that the `view` does.

In the example below, we create a component called `Hello()` that accepts an object as a parameter. We extract the `name` value from this object using destructuring, before returning a heading containing this value:

```
const Hello = ({name}) => <h1>Hello {name}</h1>;
```

We can now refer to this component in the view as if it were an HTML element entitled `<Hello />`. We can pass data to this element in the same way that we can pass `props` to a React component:

```
const view = (state, actions) => (
  <Hello name="Hyperapp" />
);
```

Note that, as we're using JSX, component names must start with capital letters or contain a period.

State

The state is a plain old JavaScript object that contains information about the application. It's the “single source of truth” for the application and can only be changed using actions.

Let's create the state object for our application and set a property called `name`:

```
const state = {
  name: "Hyperapp"
};
```

The view function now has access to this property. Update the code to the following:

```
const view = (state, actions) => (
  <Hello name={state.name} />
);
```

Since the view can access the state object, we can use its `name` property as an attribute of the `<Hello />` component.

Actions

Actions are functions used to update the state object. They're written in a particular form that returns another, curried function that accepts the current state and returns an updated, partial state object. This is partly stylistic, but also ensures that the state object remains immutable. A completely new state object is created by merging the results of an action with the previous state. This will then result in the view function being called and the HTML being updated.

The example below shows how to create an action called `changeName()`. This function accepts an argument called `name` and returns a curried function that's used to update the `name` property in the state object with this new name.

```
const actions = {
  changeName: name => state => ({name: name})
};
```

To see this action, we can create a button in the view and use an `onclick` event handler to call the action, with an argument of “Batman”. To do this, update the `view` function to the following:

```
const view = (state, actions) => (
  <div>
    <Hello name={state.name} />
    <button onclick={() => actions.changeName('Batman')}>I'm Batman</button>
  </div>
);
```

Now try clicking on the button and watch the name change!

You can see a [live example here](#).

Hyperlist

Now it’s time to build something more substantial. We’re going to build a simple to-do list app that allows you to create a list, add new items, mark them as complete and delete items.

First of all, we’ll need to start a new pen on CodePen. Add the following code, or simply fork my HyperBoiler pen:

```
const { h, app } = hyperapp;
/** @jsx h */

const state = {

};

const actions = {

};

const view = (state, actions) => (

);

const main = app(state, actions, view, document.body);
```

You should also add the following in the CSS section and set it to SCSS:

```
// fonts
@import url("https://fonts.googleapis.com/css?family=Racing+Sans+One");
$base-fonts: Helvetica Neue, sans-serif;
$heading-font: Racing Sans One, sans-serif;

// colors
$primary-color: #00caff;
$secondary-color: hotpink;
$bg-color: #222;

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  padding-top: 50px;
  background: $bg-color;
  color: $primary-color;
  display: flex;
  height: 100vh;
  justify-content: center;
  font-family: $base-fonts;
}

h1 {
  color: $secondary-color;
  & strong{ color: $primary-color; }
  font-family: $heading-font;
  font-weight: 100;
  font-size: 4.2em;
  text-align: center;
}

a{
  color: $primary-color;
}

.flex{

  display: flex;
  align-items: top;
  margin: 20px 0;

  input {
```

```
border: 1px solid $primary-color;
background-color: $primary-color;

font-size: 1.5em;
font-weight: 200;

width: 50vw;
height: 62px;

padding: 15px 20px;
margin: 0;
outline: 0;

&::placeholder {
  color: $bg-color;
}

&::moz-placeholder {
  color: $bg-color;
}

&::ms-placeholder {
  color: $bg-color;
}

&:hover, &:focus, &:active {
  background: $primary-color;
}
}

button {
  height: 62px;
  font-size: 1.8em;
  padding: 5px 15px;
  margin: 0 3px;
}
}

ul#list {
  display: flex;
  flex-direction: column;
  padding: 0;
  margin: 1.2em;
  width: 50vw;
  li {
    font-size: 1.8em;
    vertical-align: bottom;
    &.completed{
      color: $secondary-color;
      text-decoration: line-through;
    }
  }
}
```

```
button{
  color: $primary-color;
}
}
button {
  visibility: hidden;
  background: none;
  border: none;
  color: $secondary-color;
  outline: none;
  font-size: 0.8em;
  font-weight: 50;
  padding-top: 0.3em;
  margin-left: 5px;
}
&:hover{
  button{
    visibility: visible;
  }
}
}
}

button {
  background: $bg-color;
  border-radius: 0px;
  border: 1px solid $primary-color;
  color: $primary-color;

  font-weight: 100;

  outline: none;

  padding: 5px;

  margin: 0;

  &:hover, &:disabled {
    background: $primary-color;
    color: #111;
  }
  &:active {
    outline: 2px solid $primary-color;
  }
  &:focus {
    border: 1px solid $primary-color;
  }
}
```

These just add a bit of style and Hyperapp branding to the application.

Now let's get on and start building the actual application!

Initial State and View

To start with, we're going to set up the initial state object and a simple view.

When creating the initial state object, it's useful to think about what data and information your application will want to keep track of throughout its lifecycle. In the case of our list, we'll need an array to store the to-dos, as well as a string that represents whatever is written in the input field where the actual to-dos are entered. This will look like the following:

```
const state = {
  items: [],
  input: '',
};
```

Next, we'll create the `view()` function. To start with, we'll focus on the code required to add an item. Add the following code:

```
const view = (state, actions) => (
  <div>
    <h1><strong>Hyper</strong>List</h1>
    <AddItem add={actions.add} input={actions.input} value={state.in}>
  </div>
);
```

This will display a title, as well as an element called `<AddItem />`. This isn't a new HTML element, but a component that we'll need to create. Let's do that now:

```
const AddItem = ({ add, input, value }) => (
  <div class='flex'>
    <input type="text" value={value}
      onkeyup={e => (e.keyCode === 13 ? add() : null)}
      oninput={e => input({ value: e.target.value })}>
    />
    <button onclick={add}></button>
  </div>
);
```

This returns an `<input>` element that will be used to enter our to-dos, as well as a `<button>` element that will be used to add them to the list. The component accepts an object as an argument, from which we extract three properties: `add`, `input` and `value`.

As you might expect, the `add()` function will be used to add an item to our list of to-dos. This function is called, either if the `Enter` key is pressed (it has a `KeyCode` of 13) or if the button is clicked. The `input()` function is used to update the value of the current item in state and is called whenever the text field receives user input. Finally, the `value` property is whatever the user has typed into the input field.

Note that the `input()` and `add()` functions are actions, passed as props to the `<AddItem />` component:

```
<AddItem add={actions.add} input={actions.input} value={state.input}>
```

You can also see that the `value` prop is taken from the state's `input` property. So the text that's displayed in the input field is actually stored in the state and updated every time a key is pressed.

To glue everything together, we need to add the `input` action:

```
const actions = {
  input: ({ value }) => ({ input: value })
}
```

Now if you start typing inside the input field, you should see that it displays what you're typing. This demonstrates the Hyperapp loop:

1. the `oninput` event is triggered as the user types text into the input field
2. the `input()` action is called
3. the action updates the `input` property in the state
4. a change in state causes the `view()` function to be called and the VDOM is updated
5. the changes in the VDOM are then made to the actual DOM and the page is re-rendered to display the key that was pressed.

Have a go and you should see what's typed appear in the input field. Unfortunately, pressing `Enter` or clicking on the “+” button doesn't do anything at the moment. That's because we need to create an action that adds items to our

list.

Adding a Task

Before we look at creating a list item, we need to think how they're going to be represented. JavaScript's object notation is perfect, as it lets us store information as key-value pairs. We need to think about what properties a list item might have. For example, it needs a value that describes what needs to be done. It also needs a property that states if the item has been completed or not. An example might be:

```
{  
  value: 'Buy milk',  
  completed: false,  
  id: 123456  
}
```

Notice that the object also contains a property called `id`. This is because VDOM nodes in Hyperapp require a unique key to identify them. We'll [use a timestamp for this](#).

Now we can have a go at creating an action for adding items. Our first job is to reset the input field to be empty. This is done by resetting the `input` property to an empty string. We then need to add a new object to the `items` array. This is done using the `Array.concat` method. This acts in a similar way to the `Array.push()` method, but it returns a new array, rather than mutating the array it's acting on. Remember, we want to create a new state object and then merge it with the current state rather than simply mutating the current state directly. The `value` property is set to the value contained in `state.input`, which represents what's been entered in the input field:

```
add: () => state => ({  
  input: '',  
  items: state.items.concat({  
    value: state.input,  
    completed: false,  
    id: Date.now()  
  })  
})
```

Note that this action contains two different states. There's the *current* state that's represented by the argument supplied to the second function. There's also the

new state that's the return value of the second function.

To demonstrate this in action, let's imagine the app has just started with an empty list of items and a user has entered the text "Buy milk" into the input field and pressed Enter, triggering the `add()` action.

Before the action, the state looks like this:

```
state = {  
  input: 'Buy milk',  
  items: []  
}
```

This object is passed as an argument to the `add()` action, which will return the following state object:

```
state = {  
  input: '',  
  items: [{  
    value: 'Buy milk',  
    completed: false,  
    id: 1521630421067  
  }]  
}
```

Now we can add items to the `items` array in the state, but we can't see them! To fix this, we need to update our view. First of all we need to create a component for displaying the items in the list:

```
const ListItem = ({ value, id }) => <li id={id} key={id}>{value}</li>
```

This uses a `` element to display a value, which is provided as an argument. Note also that the `id` and `key` attributes both have the same value, which is the unique ID of the item. The `key` attribute is used internally by Hyperapp, so isn't displayed in the rendered HTML, so it's useful to also display the same information using the `id` attribute, especially since this attribute shares the same condition of uniqueness.

Now that we have a component for our list items, we need to actually display them. JSX makes this quite straightforward, as it will loop over an array of values and display each one in turn. The problem is that the `state.items` doesn't include JSX code, so we need to use `Array.map` to change each item object in the array into JSX code, like so:

```
state.items.map(item => ( <ListItem id={item.id} value={item.value}>
```

This will iterate over each object in the `state.items` array and create a new array that contains `ListItem` components instead. Now we just need to add this to the view. Update the `view()` function to the code below:

```
const view = (state, actions) => (
  <div>
    <h1><strong>Hyper</strong>List</h1>
    <AddItem add={actions.add} input={actions.input} value={state.in
    <ul id='list'>
      { state.items.map(item => ( <ListItem id={item.id} value={item
    </ul>
  </div>
);
```

This simply places the new array of `ListItem` components inside a pair of `` tags so they're displayed as an unordered list.

Now if you try adding items, you should see them appear in a list below the input field!

Mark a Task as Completed

Our next job is to be able to toggle the `completed` property of a to-do. Clicking on an uncompleted task should update its `completed` property to `true`, and clicking on a completed task should toggle its `completed` property back to `false`.

This can be done by using the following action:

```
toggle: id => state => ({
  items: state.items.map(item => (
    id === item.id ? Object.assign({}, item, { completed: !item.comp
  )))
})
```

There's quite a bit going on in this action. First of all, it accepts a parameter called `id`, which refers to the unique ID of each task. The action then iterates over *all* of the items in the array, checking if the ID value provided matches the `id` property of each list item object. If it does, it changes the `completed` property to the opposite of what it currently is using the negation operator `!`. This change

is made using the `Object.assign()` method, which creates a new object and performs a shallow merge with the old `item` object and the updated properties. Remember, we *never* update objects in the state directly. Instead, we create a new version of the state that overwrites the current state.

Now we need to wire this action up to the view. We do this by updating the `ListItem` component so that it has an `onclick` event handler that will call the `toggle` action we just created. Update the `ListItem` component code so that it looks like the following:

```
const ListItem = ({ value, id, completed, toggle, destroy }) => (
  <li class={completed && "completed"} id={id} key={id} onclick={e =>
    e.preventDefault();
    toggle(id);
  }>
```

The eagle-eyed among you will have spotted that the component has gained some extra parameters and there's also some extra code in the `` attributes list:

```
class={completed && "completed"}
```

This is a common pattern in Hyperapp that's used to insert extra fragments of code when certain conditions are true. It uses *short-circuit* or *lazy* evaluation to set the class as `completed` if the `completed` argument is true. This is because when using `&&`, the return value of the operation will be the second operand if both operands are true. Since the string `"completed"` is always true, this will be returned if the first operand — the `completed` argument — is true. This means that, if the task has been completed, it will have class of `"completed"` and can be styled accordingly.

Our last job is to update the code in the `view()` function to add the extra argument to the `<ListItem />` component:

```
const view = (state, actions) => (
  <div>
    <h1><strong>Hyper</strong>List</h1>
    <AddItem add={actions.add} input={actions.input} value={state.input}>
      <ul id='list'>
        {
          state.items.map(item => (
            <ListItem id={item.id} value={item.value} completed={item.completed}>
            </ListItem>
          ))
        }
      </ul>
    </AddItem>
  </div>
)
```

```
</div>
);
```

Now if you add some items and try clicking on them, you should see that they get marked as complete, with a line appearing through them. Click again and they revert back to incomplete.

Delete a Task

Our list app is running quite nicely at the moment, but it would be good if we could delete any items that we no longer need in the list.

Our first job is to add a `destroy()` action that will remove an item from the `items` array in state. We can't do this using the `Array.slice()` method, as this is a destructive method that acts on the original array. Instead, we use the `filter()` method, which returns a new array that contains all the item objects that pass a specified condition. This condition is that the `id` property doesn't equal the ID that was passed as an argument to the `destroy()` action. In other words, it returns a new array that doesn't include the item we want to get rid of. This new list will then replace the old one when the state is updated.

Add the following code to the `actions` object:

```
destroy: id => state => ({ items: state.items.filter(item => item.id
```

Now we again have to update the `ListItem` component to add a mechanism for triggering this action. We'll do this by adding a button with an `onclick` event handler:

```
const ListItem = ({ value, id, completed, toggle, destroy }) => (
  <li class={completed && "completed"} id={id} key={id} onclick={e =>
    {value}
    <button onclick={() => destroy(id)}>x</button>
  </li>
);
```

Note that we also need to add another parameter called `destroy` that represents the action we want to use when the button is clicked. This is because components don't have direct access to the `actions` object in the same way as the view does, so the view needs to pass any actions explicitly.

Last of all, we need to update the view to pass `actions.destroy` as an argument to the `<ListItem />` component:

```
const view = (state, actions) => (
  <div>
    <h1><strong>Hyper</strong>List</h1>
    <AddItem add={actions.add} input={actions.input} value={state.input}>
    <ul id='list'>
      {state.items.map(item => (
        <ListItem
          id={item.id}
          value={item.value}
          completed={item.completed}
          toggle={actions.toggle}
          destroy={actions.destroy}
        />
      ))}
    </ul>
  </div>
);
```

Now if you add some items to your list, you should notice the “x” button when you mouse over them. Click on this and they should disappear into the ether!

Delete All Completed Tasks

The last feature we’ll add to our list app is the ability to remove all completed tasks at once. This uses the same `filter()` method that we used earlier — returning an array that only contains item objects with a `completed` property value of `false`. Add the following code to the `actions` object:

```
clearAllCompleted: ({items}) => ({ items: items.filter(item => !item.comPLETED) })
```

To implement this, we simply have to add a button with an `onclick` event handler to call this action to the bottom of the view:

```
const view = (state, actions) => (
  <div>
    <h1><strong>Hyper</strong>List</h1>
    <AddItem add={actions.add} input={actions.input} value={state.input}>
    <ul id='list'>
      {state.items.map(item => (
        <ListItem
          id={item.id}
          value={item.value}
```

```
        completed={item.completed}
        toggle={actions.toggle}
        destroy={actions.destroy}
    />
  ))
)
</ul>
<button onclick={() => actions.clearAllCompleted({ items: state.
  Clear completed items
</button>
</div>
);
```

Now have a go at adding some items, mark a few of them as complete, then press the button to clear them all away. Awesome!

Complete Example

See the Pen [Hyperlist](#).

That's All, Folks

That brings us to the end of this tutorial. We've put together a simple to-do list app that does most things you'd expect such an app to do. If you're looking for inspiration and want to add to the functionality, you could look at adding priorities, changing the order of the items using drag and drop, or adding the ability to have more than one list.

I hope this tutorial has helped you to gain an understanding about how Hyperapp works. If you'd like to dig a bit deeper into Hyperapp, I'd recommend [reading the docs](#) and also having a peek at the [source code](#). It's not very long, and will give you a useful insight into how everything works in the background. You can also ask more questions on the [Hyperapp Slack group](#). It's one of the friendliest groups I've used, and I've been given a lot of help by the knowledgeable members. You'll also find that [Jorge Bucaran](#), the creator of Hyperapp, frequently hangs out on there and offers help and advice.

Using CodePen makes developing Hyperapp applications really quick and easy, but you'll eventually want to build your own applications locally and also deploy them online. For tips on how to do that, check out the next chapter on bundling a Hyperapp app and deploying it to GitHub Pages!

Chapter 5: Use Parcel to Bundle a Hyperapp App & Deploy to GitHub Pages

by Darren Jones

In the last chapter we met Hyperapp, a tiny library that can be used to build dynamic, single-page web apps in a similar way to React or Vue.

In this chapter we're going to turn things up a notch. We're going to create the app locally (we were working on CodePen previously), learn how to bundle it using [Parcel](#) (a module bundler similar to webpack or Rollup) and deploy it to the web using [GitHub Pages](#).

Don't worry if you didn't complete the project from the first post. All the code is provided here (although I won't go into detail explaining what it does) and the principles outlined can be applied to most other JavaScript projects.

If you'd like to see what we'll be ending up with, you can [view the finished project here](#), or download the code from [our GitHub repo](#).

Basic Setup

In order to follow along, you'll need to have both [Node.js and npm](#) installed (they come packaged together). I'd recommend using a version manager such as [nvm](#) to manage your Node installation ([here's how](#)), and if you'd like some help getting to grips with npm, then check out our [beginner-friendly npm tutorial](#).

We'll be using the terminal commands to create files and folders, but feel free to do it by just pointing and clicking instead if that's your thing.

To get started, create a new folder called `hyperlist`:

```
mkdir hyperlist
```

Now change to that directory and initialize a new project using npm:

```
cd hyperlist/  
npm init
```

This will prompt you to answer some questions about the app. It's fine to just press enter to accept the default for any of these, but feel free to add in your name as the author and to add a description of the app.

This should create a file called `package.json` inside the `hyperlist` directory that looks similar to the following:

```
{  
  "name": "hyperlist",  
  "version": "1.0.0",  
  "description": "A To-do List made with Hyperapp",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "DAZ",  
  "license": "MIT"  
}
```

Now we need to install the Hyperapp library. This is done using npm along with the `--save` flag, which means that the `package.json` file will be updated to include it as a dependency:

```
npm install --save hyperapp
```

This might give some warnings about not having a `repository` field. Don't worry about this, as we'll be fixing it later. It should update the `package.json` file to include the following entry (there might be a slight difference in version number):

```
"dependencies": {  
  "hyperapp": "^1.2.5"  
}
```

It will also create a directory called `node_modules` where all the Hyperapp files are stored, as well as a file called `package-lock.json`. This is used to keep track of the dependency tree for all the packages that have been installed using npm.

Now we're ready to start creating the app!

Folder Structure

It's a common convention to put all of your source code into a folder called `src`. Within this folder, we're going to put all of our JavaScript files into a directory called `js`. Let's create both of those now:

```
mkdir -p src/js
```

In the [previous post](#) we learned that apps are built in Hyperapp using three main parts: state, actions and view. In the interests of code organization, we're going to place the code for each part in a separate file, so we need to create these files inside the `js` directory:

```
cd src/js
touch state.js actions.js view.js
```

Don't worry that they're all empty. We'll add the code soon!

Last of all, we'll go back into the `src` directory and create our "entry point" files. These are the files that will link to all the others. The first is `index.html`, which will contain some basic HTML, and the other is `index.js`, which will link to all our other JavaScript files and also our SCSS files:

```
cd ..
touch index.html index.js
```

Now that our folder structure is all in place, we can go ahead and start adding some code and wiring all the files together. Onward!

Some Basic HTML

We'll start by adding some basic HTML code to the `index.html` file. Hyperapp takes care of creating the HTML and can render it directly into the `<body>` tag. This means that we only have to set up the meta information contained in the `<head>` tag. Except for the `<title>` tag's value, you can get away with using the same `index.html` file for every project. Open up `index.html` in your favorite text editor and add the following code:

```
<!doctype html>
<html lang='en'>
```

```
<head>
  <meta charset='utf-8'>
  <meta name='viewport' content='width=device-width, initial-scale=1'>
  <title>HyperList</title>
</head>
<body>
  <script src='index.js'></script>
</body>
</html>
```

Now it's time to add some JavaScript code!

ES6 Modules

[Native JavaScript modules](#) were introduced in ES6 (aka ES2015). Unfortunately, browsers have been slow to adopt the use of ES6 modules natively, although [things are now starting to improve](#). Luckily, we can still use them to organize our code, and Parcel will sort out piecing them all together.

Let's start by adding the code for the initial state inside the `state.js` file:

```
const state = {
  items: [],
  input: '',
  placeholder: 'Make a list..'
};

export default state;
```

This is the same as the object we used in [the previous chapter](#), but with the `export` declaration at the end. This will make the object available to any other file that imports it. By making it the default export, we don't have to explicitly name it when we import it later.

Next we'll add the actions to `actions.js`:

```
const actions = {
  add: () => state => ({
    input: '',
    items: state.items.concat({
      value: state.input,
      completed: false,
      id: Date.now()
    })
})
```

```

}),
input: ({ value }) => ({ input: value }),
toggle: id => state => ({
  items: state.items.map(item => (
    id === item.id ? Object.assign({}, item, { completed: !item.completed })
  ))
}),
destroy: id => state => ({
  items: state.items.filter(item => item.id !== id)
}),
clearAllCompleted: ({ items }) => ({
  items: items.filter(item => !item.completed)
})
};

export default actions;

```

Again, this is the same as the object we used in the previous chapter, with the addition of the `export` declaration at the end.

Last of all we'll add the view code to `view.js`:

```

import { h } from 'hyperapp'

const AddItem = ({ add, input, value, placeholder }) => (
  <div class='flex'>
    <input
      type="text"
      onkeyup={e => (e.keyCode === 13 ? add() : null)}
      oninput={e => input({ value: e.target.value })}
      value={value}
      placeholder={placeholder}
    />
    <button onclick={add}></button>
  </div>
);

const ListItem = ({ value, id, completed, toggle, destroy }) => (
  <li class={completed && "completed"} id={id} key={id} onclick={e => {
    value} <button onclick={ () => destroy(id) }>x</button>
  </li>
);

const view = (state, actions) => (
  <div>
    <h1><strong>Hyper</strong>List</h1>
    <AddItem
      add={actions.add}

```

```

        input={actions.input}
        value={state.input}
        placeholder={state.placeholder}
    />
    <ul id='list'>
      {state.items.map(item => (
        <ListItem
          id={item.id}
          value={item.value}
          completed={item.completed}
          toggle={actions.toggle}
          destroy={actions.destroy}
        />
      ))}
    </ul>
    <button onclick={() => actions.clearAllCompleted({ items: state.items })}>
      Clear completed items
    </button>
  </div>s
);
export default view;

```

First of all, this file uses the `import` declaration to import the `h` module from the Hyperapp library that we installed using npm earlier. This is the function that Hyperapp uses to create the Virtual DOM nodes that make up the view.

This file contains two components: `AddItem` and `ListItem`. These are just functions that return JSX code and are used to abstract different parts of the view into separate building blocks. If you find that you're using a large number of components, it might be worth moving them into a separate `components.js` file and then importing them into the `view.js` file.

Notice that only the `view` function is exported at the end of the file. This means that only this function can be imported by other files, rather than the separate components.

Now we've added all our JavaScript code, we just need to piece it all together in the `index.js` file. This is done using the `import` directive. Add the following code to `index.js`:

```

import { app } from 'hyperapp'

import state from './js/state.js'
import actions from './js/actions.js'

```

```
import view from './js/view.js'  
const main = app(state, actions, view, document.body);
```

This imports the `app` function from the Hyperapp library, then imports the three JavaScript files that we just created. The object or function that was exported from each of these files is assigned to the variables `state`, `actions` and `view` respectively, so they can be referenced in this file.

The last line of code calls the `app` function, which starts the app running. It uses each of the variables created from our imported files as the first three arguments. The last argument is the HTML element where the app will be rendered — which, by convention, is `document.body`.

Add Some Style

Before we go on to build our app, we should give it some style. Let's go to the `src` directory and create a folder for our SCSS:

```
mkdir src/scss
```

Now we'll create the two files that will contain the SCSS code that we used in part 1:

```
cd src/scss  
touch index.scss _settings.scss
```

We're using a file called `_settings.scss` to store all the Sass variables for the different fonts and colors our app will use. This makes them easier to find if you decide to update any of these values in the future. Open up the `_settings.scss` file and add the following code:

```
// fonts  
@import url("https://fonts.googleapis.com/css?family=Racing+Sans+One");  
$base-fonts: Helvetica Neue, sans-serif;  
$heading-font: Racing Sans One, sans-serif;  
  
// colors  
$primary-color: #00caff;  
$secondary-color: hotpink;  
$bg-color: #222;
```

The app-specific CSS goes in `index.scss`, but we need to make sure we import the `_settings.scss` file at the start, as the variables it contains are referenced later on in the file. Open up `index.scss` and add the following code:

```
@import 'settings';

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  padding-top: 50px;
  background: $bg-color;
  color: $primary-color;
  display: flex;
  height: 100vh;
  justify-content: center;
  font-family: $base-fonts;
}

h1 {
  color: $secondary-color;
  & strong{ color: $primary-color; }
  font-family: $heading-font;
  font-weight: 100;
  font-size: 4.2em;
  text-align: center;
}

a{ color: $primary-color; }

.flex{
  display: flex;
  align-items: top;
  margin: 20px 0;

  input {
    border: 1px solid $primary-color;
    background-color: $primary-color;
    font-size: 1.5em;
    font-weight: 200;
    width: 50vw;
    height: 62px;
    padding: 15px 20px;
    margin: 0;
    outline: 0;
  }
}
```

```
&::placeholder { color: $bg-color; }
&::moz-placeholder { color: $bg-color; }
&::ms-input-placeholder { color: $bg-color; }
&:hover, &:focus, &:active { background: $primary-color; }
}

button {
  height: 62px;
  font-size: 1.8em;
  padding: 5px 15px;
  margin: 0 3px;
}
}

ul#list {
  display: flex;
  flex-direction: column;
  padding: 0;
  margin: 1.2em;
  width: 50vw;
  li {
    font-size: 1.8em;
    vertical-align: bottom;
    &.completed{
      color: $secondary-color;
      text-decoration: line-through;
      button{
        color: $primary-color;
      }
    }
    button {
      background: none;
      border: none;
      color: $secondary-color;
      outline: none;
      font-size: 0.8em;
      font-weight: 50;
      padding-top: 0.3em;
      margin-left: 5px;
    }
  }
}

button {
  background: $bg-color;
  border-radius: 0px;
  border: 1px solid $primary-color;
  color: $primary-color;
  font-weight: 100;
  outline: none;
```

```
padding: 5px;  
margin: 0;  
  
  &:hover, &:disabled {  
    background: $primary-color;  
    color: #111;  
  }  
  
  &:active { outline: 2px solid $primary-color; }  
  &:focus { border: 1px solid $primary-color; }  
}
```

If your SCSS starts to get more complicated, you can break it up into separate files and then import them all into `index.scss`.

Now we need to link these files to our app. We don't actually place the link in our HTML file, as you usually do with CSS. Instead, we place it in the `index.js` file. This is because we're using SCSS and it needs to be pre-processed into CSS. Parcel will do this for us and also sort out linking the HTML file to the standard CSS file that it creates.

To import the SCSS files, we just need to update our `index.js` file to include the following line:

```
import './scss/index.scss'
```

Now that all our code's complete, it's time to start work on the build process!

Babel

Babel will transpile the modern JavaScript code into code that most browsers can consume. It will also take care of rewriting the JSX code into pure JavaScript.

In order to be able to use Babel with JSX transforms, we need to install it along with the JSX plugin:

```
npm install --save babel-plugin-transform-react-jsx babel-preset-env
```

We also need to create a `.babel.rc` file that's used to tell Babel to use the `h` function from Hyperapp when processing the JSX. The following code will create the file with the relevant information:

```
echo '{ "plugins": [ ["transform-react-jsx", { "pragma": "h" }]] }' >
```

Note that this is a *hidden file*, so you might not be able to see it after it's been created!

Parcel

Unfortunately, our code won't currently work in all browsers as it stands. We need to use a build process to transpile our ES6+ code to ES5 and merge all our JS files into a single file. Let's use [Parcel](#) to do that.

Parcel is a module bundler, similar to webpack or Rollup, that promises zero-configuration and is blazingly fast. It allows us to write modern JavaScript in separate files, and then bundles them together into a single, minified JavaScript file that most browsers will be able to consume. It also supports multiple CSS, SCSS and PostCSS files out of the box.

First of all, let's install Parcel:

```
npm install --save parcel-bundler
```

Parcel comes with its own built-in server. This means that you can continue to develop and make changes to the app and Parcel will build it in the background, so any changes are shown instantly!

To start the server running, enter the following command:

```
./node_modules/.bin/parcel src/index.html --out-dir docs
```

This specifies that the entry point is the `index.html` file. This is all Parcel needs to know about, as it will follow the link to `index.js` that's in this file and then follow the `import` directives in that file.

It also specifies that a folder called `docs` be used to output all of the static files to. By default, this is usually called `dist` — but, as you'll see later, we need it to be called `docs` so that we can integrate it with GitHub Pages.

You should also see a message that the app is being built in the terminal window. You might even notice that Parcel installs the npm module `node-sass` for you as it automatically notices that we've been using SCSS files, but also that we don't

have node-sass installed. How cool is that?!

After a few seconds, you should see a message similar to the following:

```
Server running at http://localhost:1234
Built in 3.15s.
```

The server is now running, and if you open up your browser and go to <http://localhost:1234>, you'll be able to see the app running. This will update on the fly, so any changes you make in your code will be reflected on the page straight away (or after a brief pause to rebuild the code). It also hotloads modules, so it will automatically install any npm modules that are required as they're needed, like it did with "node-sass". Awesome!

Once you're happy with how the site looks, it's time to build the static site. First of all, stop the server running by holding down **ctrl** and **c** together. Then run the following command in the terminal:

```
./node_modules/.bin/parcel build src/index.html --out-dir docs --pub
```

This will build the static files and place them inside the `docs` folder.

If you take a peak inside the `docs` folder, you should find a file called `index.html`. Open this in your browser and you should see the site running, using only the static files in the `docs` folder. Parcel has bundled all the relevant code together and used Babel to transpile our modern JavaScript into a single JavaScript file and used node-sass to pre-process our SCSS files into a single CSS file. Open them up and you can see that the code has also been minimized!

npm Scripts

npm has a useful feature called *scripts* that allows you to run specific pieces of code with a single command. We can use this to create a couple of scripts that will speed up our use of Parcel.

Add the following to the "scripts" section of the `package.json` file:

```
"start": "parcel src/index.html --out-dir docs",
"build": "parcel build src/index.html --out-dir docs --public-url ./
```

Now we can simply run the following commands to start the server:

```
npm start
```

And the following command will run the build process:

```
npm run build
```

If you've never used npm scripts, or would like a refresher, you might like to check out our [beginner-friendly tutorial on the subject](#).

Deploying to GitHub Pages

GitHub is a great place for hosting your code, and it also has a great feature called [GitHub Pages](#) that allows you to host static sites on GitHub. To get started, you'll need to make sure you have a GitHub account and you have [git](#) installed on your local machine.

To make sure we don't commit unnecessary files, let's add a `gitignore` file to the `hyperlist` directory:

```
touch .gitignore
```

As the name suggests, this file tells git which files (or patterns) it should ignore. It's usually used to avoid committing files that aren't useful to other collaborators (such as the temporary files IDEs create, etc.).

I'd recommend adding the following items to make sure they're not tracked by git (remember that `gitignore` is a hidden file!):

```
# Logs
logs
*.log
npm-debug.log*

# Runtime data
pids
*.pid
*.seed

# Dependency directory
node_modules

# Optional npm cache directory
.npm
```

```
# Optional REPL history  
.node_repl_history  
  
# Cache for Parcel  
.cache  
  
# Apple stuff  
.DS_Store
```

Now we're ready to initialize git in the `hyperlist` directory:

```
git init
```

Next, we add all the files we've created so far:

```
git add .
```

Then we commit those files to version control:

```
git commit -m 'Initial Commit'
```

Now that our important files are being tracked by git, we need to create a remote repository on GitHub. Just log into your account and click on the *New Repository* button and follow the instructions. If you get stuck, you can consult GitHub's documentation here: [Create A Repo](#).

After you've done this, you'll need to add the URL of your remote GitHub repository on your local machine:

```
git remote add origin https://github.com/<username>/<repo-name>.git
```

Be sure to replace `<username>` and `<repo-name>` with the correct values. If you'd like to check that you've done everything correctly, you can use `git remote -v`.

And, finally, we need to push our code to GitHub:

```
git push origin master
```

This will push all of your code to your GitHub repository, including the static files in the `docs` directory. GitHub Pages can now be configured to use the files in this directory. To do this, log in to the repository on GitHub and go to the

Settings section of the repository and scroll down to the *GitHub Pages* section. Then under *Source*, select the option that says “master branch /docs folder”, as can be seen in the screenshot below:



This should mean that you can now access the app at the following address:
<https://username.github.io/repo-name>.

For example, you can see ours at sitepoint-editors.github.io/hyperlist/.

Workflow

From now on, if you make any changes to your app, you can adhere to the following workflow:

1. start the development server: `npm start`
2. make any changes
3. check that the changes work on the development server
4. shut the server down by holding down `ctrl + c`
5. rebuild the app: `npm run build`
6. stage the changes for commit: `git add .`
7. commit all the changes to git: `git commit -m 'latest update'`
8. push the changes to GitHub: `git push origin master`.

We can speed this process up by creating an npm script to take care of the last three steps in one go. Add the following to the “scripts” entry in `package.json`:

```
"deploy": "npm run build && git add . && git commit -a -m 'latest bu
```

Now all you need to do if you want to deploy your code after making any changes is to run the following command:

```
npm run deploy
```

That's All, Folks!

And that brings us to the end of this tutorial. I used the app we created in part 1 of this tutorial, but the principles remain the same for most JavaScript projects. Hopefully I've demonstrated how easy it is to use Parcel to build a static JS site and automatically deploy it to GitHub Pages with just a single command!

Chapter 6: Interactive Data Visualization with Modern JavaScript and D3

by Adam Janes

In this chapter, I want to take you through an example project that I built recently — a *totally original* type of visualization using the D3 library, which showcases how each of these components add up to make D3 a great library to learn.

D3 stands for Data Driven Documents. It's a JavaScript library that can be used to make all sorts of wonderful data visualizations and charts.

If you've ever seen any of the [fabulous interactive stories](#) from the New York Times, you'll already have seen D3 in action. You can also see some cool examples of great projects that have been built with D3 [here](#).

The learning curve is pretty steep for getting started with the library, since D3 has [a few special quirks](#) that you probably won't have seen before. However, if you can get past the first phase of learning enough D3 to be dangerous, then you'll soon be able to build some really cool stuff for yourself.

There are three main factors that really make D3 stand out from any other libraries out there:

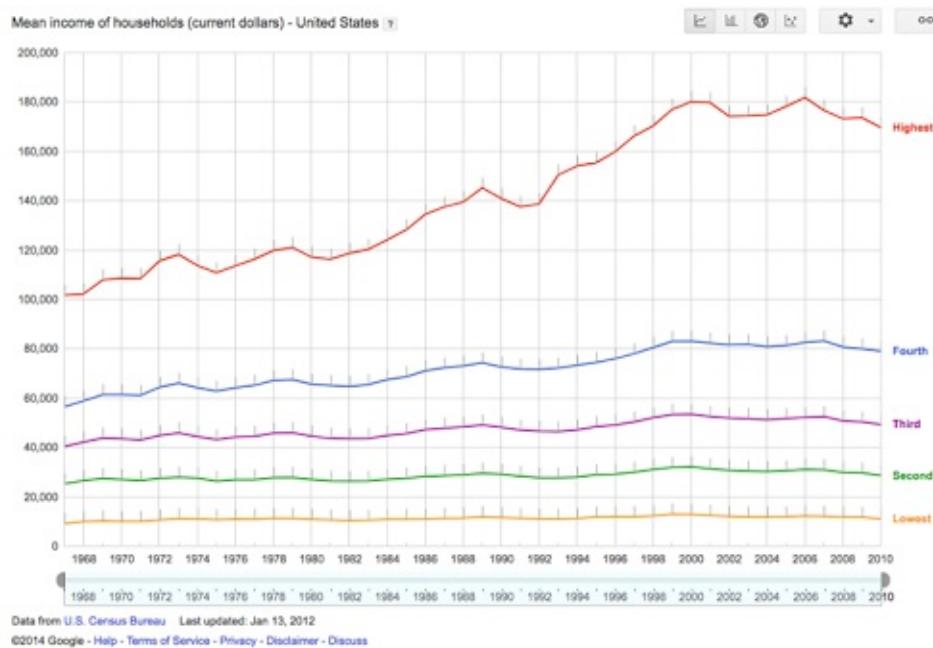
1. **Flexibility.** D3 lets you take any kind of data, and directly associate it with shapes in the browser window. This data can be *absolutely anything*, allowing for a huge array of interesting use cases to create completely original visualizations.
2. **Elegance.** It's easy to add interactive elements with *smooth transitions* between updates. The library is *written beautifully*, and once you get the hang of the syntax, it's easy to keep your code clean and tidy.
3. **Community.** There's a vast ecosystem of fantastic developers using D3 already, who readily share their code online. You can use sites like

blocks.org and blockbuilder.org to quickly find pre-written code by others, and copy these snippets directly into your own projects.

The Project

As an economics major in college, I had always been interested in income inequality. I took a few classes on the subject, and it struck me as something that wasn't fully understood to the degree that it should be.

I started exploring income inequality using [Google's Public Data Explorer](https://publicdata.google.com/explore/#/income) ...



When you adjust for inflation, household income has *stayed pretty much constant* for the bottom 40% of society, although per-worker productivity has been skyrocketing. It's only really been *the top 20%* that have reaped more of the benefits (and within that bracket, the difference is even more shocking if you look at the top 5%).

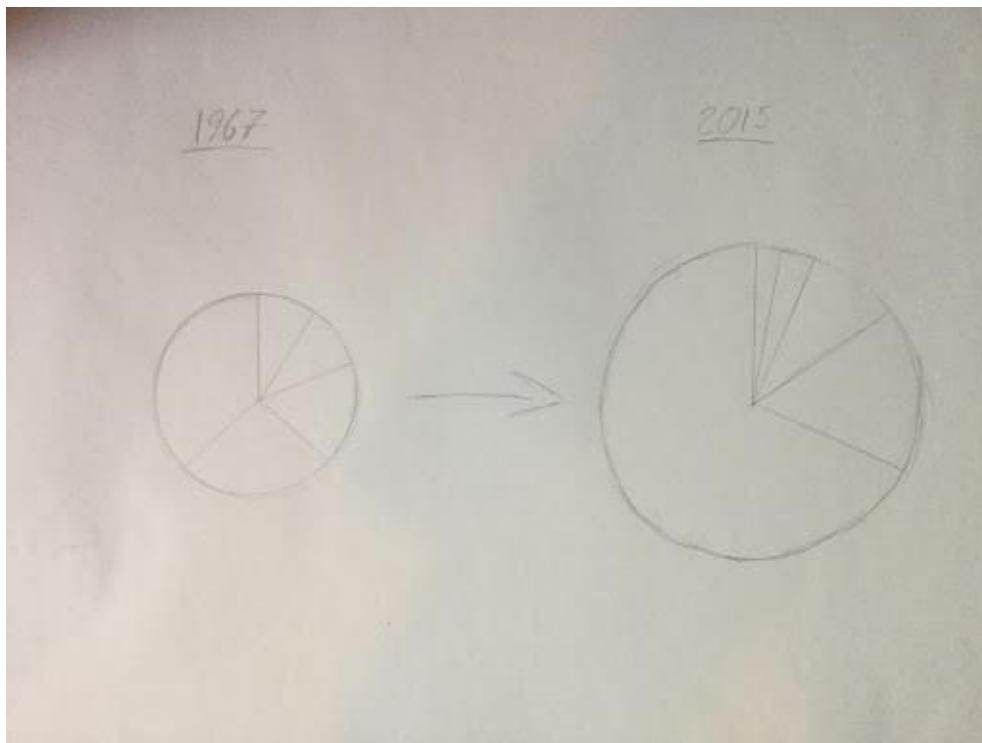
Here was a message that I wanted to get across in a convincing way, which provided a perfect opportunity to use some D3.js, so I started sketching up a few ideas.

Sketching

Because we're working with D3, I could more or less just start sketching out *absolutely anything* that I could think of. Making a simple line graph, bar chart, or bubble chart would have been easy enough, but I wanted to make something different.

I find that the most common analogy that people tended to use as a counterargument to concerns about inequality is that "if the pie gets bigger, then there's more to go around". The intuition is that, if the total share of GDP manages to increase by a large extent, then even if some people are getting a *thinner slice* of pie, then they'll still be *better off*. However, as we can see, it's totally possible for the pie to get bigger *and* for people to be getting less of it overall.

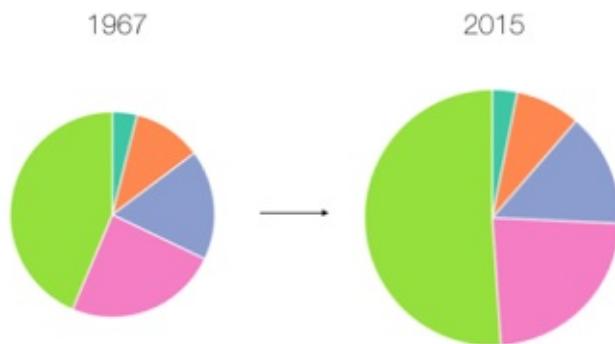
My first idea for visualizing this data looked something like this:



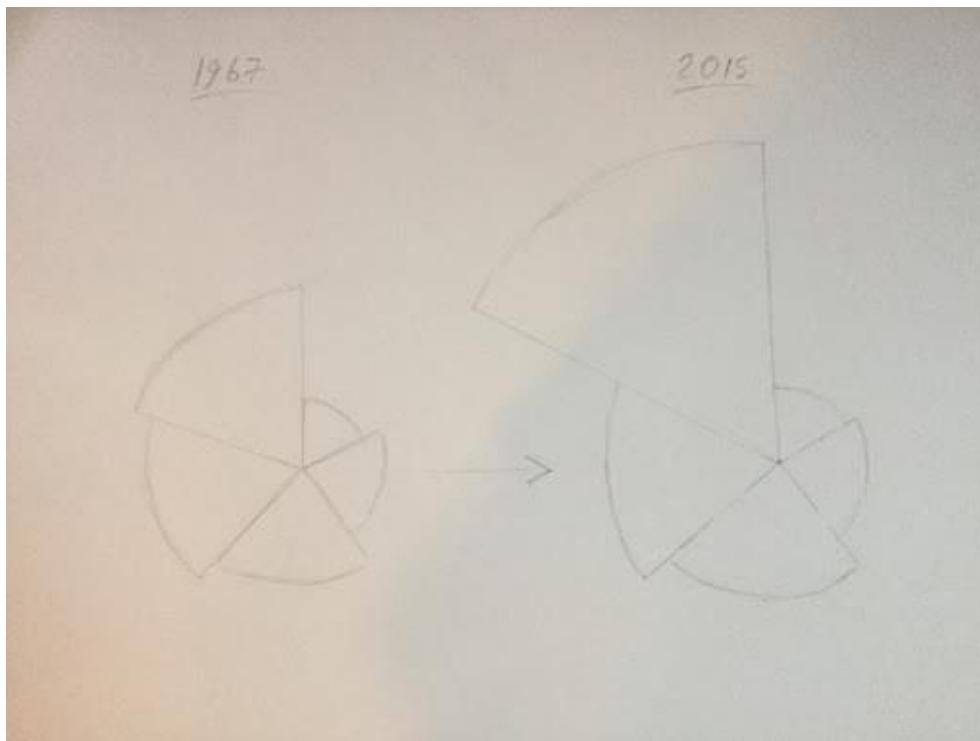
The idea would be that we'd have this pulsating pie chart, with each slice representing a fifth of the US income distribution. The area of each pie slice would relate to how much income that segment of the population is taking in, and the total area of the chart would represent its total GDP.

However, I soon came across a bit of a problem. It turns out that the human brain is *exceptionally poor at distinguishing between the size of different areas*. When

I mapped this out more concretely, the message wasn't anywhere near as obvious as it should have been:



Here, it actually looks like the poorest Americans are getting *richer* over time, which confirms what seems to be intuitively true. I thought about this problem some more, and my solution involved keeping the angle of each arc constant, with the radius of each arc changing dynamically.



Here's how this ended up looking in practice:

1967

2015



I want to point out that this image still tends to underestimate the effect here. The effect would have been more obvious if we used a simple bar chart:



However, I was committed to making a unique visualization, and I wanted to hammer home this message that the *pie* can get *bigger*, whilst a *share* of it can get *smaller*. Now that I had my idea, it was time to build it with D3.

Borrowing Code

So, now that I know what I'm going to build, it's time to get into the real meat of this project, and start *writing some code*.

You might think that I'd start by writing my first few lines of code from scratch, but you'd be wrong. This is D3, and since we're working with D3, we can always find some pre-written code from the community to start us off.

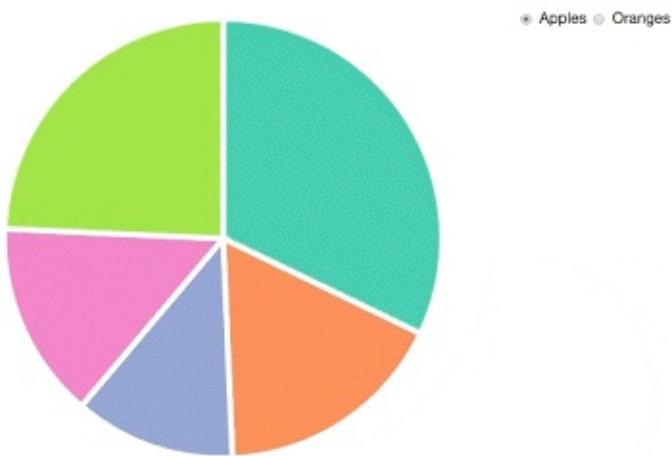
We're creating something completely new, but it has a lot in common with a regular pie chart, so I took a quick look on bl.ocks.org, and I decided to go with

this [classic implementation](#) by Mike Bostock, one of the creators of D3. This file has probably been copied thousands of times already, and the guy who wrote it is a real wizard with JavaScript, so we can be sure that we're starting with a nice block of code already.

This file is written in D3 V3, which is now two versions out of date, since version 5 was finally released last month. A big change in D3 V4 was that the library switched to using a flat namespace, so that scale functions like `d3.scale.ordinal()` are written like `d3.scaleOrdinal()` instead. In version 5, the biggest change was that data loading functions are now structured as [Promises](#), which makes it easier to handle multiple datasets at once.

To avoid confusion, I've already gone through the trouble of creating an updated V5 version of this code, which I've saved on [blockbuilder.org](#). I've also converted the syntax to fit with ES6 conventions, such as switching ES5 anonymous functions to arrow functions.

Here's what we're starting off with already:



I then copied these files into my working directory, and made sure that I could replicate everything on my own machine. If you want to follow along with this tutorial yourself, then you can [clone this project from our GitHub repo](#). You can start with the code in the file `starter.html`. Please note that you will need a server (such as [this one](#)) to run this code, as under the hood it relies on the [Fetch API](#) to retrieve the data.

Let me give you a quick rundown of how this code is working.

Walking Through Our Code

First off, we're declaring a few constants at the top of our file, which we'll be using to define the size of our pie chart:

```
const width = 540;
const height = 540;
const radius = Math.min(width, height) / 2;
```

This makes our code super reusable, since if we ever want to make it bigger or smaller, then we only need to worry about changing these values right here.

Next, we're appending an SVG canvas to the screen. If you don't know much about SVGs, then you can think about the canvas as the space on the page that we can draw shapes on. If we try to draw an SVG outside of this area, then it simply won't show up on the screen:

```
const svg = d3.select("#chart-area")
  .append("svg")
    .attr("width", width)
    .attr("height", height)
  .append("g")
    .attr("transform", `translate(${width / 2}, ${height / 2})`);
```

We're grabbing hold of an empty div with the ID of `chart-area` with a call to `d3.select()`. We're also attaching an SVG canvas with the `d3.append()` method, and we're setting some dimensions for its width and height using the `d3.attr()` method.

We're also attaching an SVG group element to this canvas, which is a special type of element that we can use to structure elements together. This allows us to shift our entire visualization into the center of the screen, using the group element's `transform` attribute.

After that, we're setting up a default scale that we'll be using to assign a new color for every slice of our pie:

```
const color = d3.scaleOrdinal(["#66c2a5", "#fc8d62", "#8da0cb", "#e78
```

Next, we have a few lines that set up D3's pie layout:

```
const pie = d3.pie()
```

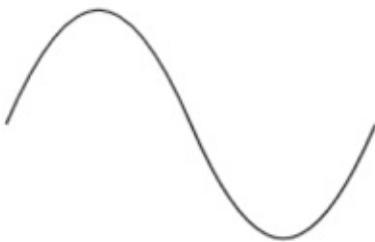
```
.value(d => d.count)  
.sort(null);
```

In D3, **layouts** are special functions that we can call on a set of data. A layout function takes in an array of data in a particular format, and spits out a *transformed array* with some automatically generated values, which we can then do something with.

We then need to define a [path generator](#) that we can use to draw our arcs. Path generators allow us to draw path SVGs in a web browser. All that D3 really does is to associate pieces of data with shapes on the screen, but in this case, we want to define a more complicated shape than just a simple circle or square. Path SVGs work by defining a route for a line to be drawn between, which we can define with its d attribute.

Here's what this might look like:

```
<svg width="190" height="160">  
  <path d="M10 80 C 40 10, 65 10, 95 80 S 150 150, 180 80" stroke="b  
</svg>
```



The d attribute contains a special encoding that lets the browser draw the path that we want. If you really want to know what this string means, you can find out about it in [MDN's SVG documentation](#). For programming in D3, we don't really need to know anything about this special encoding, since we have generators that will spit out our d attributes for us, which we just need to initialize with some simple parameters.

For an arc, we need to give our path generator an `innerRadius` and an `outerRadius` value in pixels, and the generator will sort out the complex maths that goes into calculating each of the angles for us:

```
const arc = d3.arc()  
  .innerRadius(0)  
  .outerRadius(radius);
```

For our chart, we're using a value of zero for our `innerRadius`, which gives us a standard pie chart. However, if we wanted to draw a **donut chart** instead, then all we would need to do is plug in a value that's smaller than our `outerRadius` value.

After a couple of function declarations, we're loading in our data with the `d3.json()` function:

```
d3.json("data.json", type).then(data => {  
  // Do something with our data  
});
```

In D3 version 5.x, a call to `d3.json()` returns a [Promise](#), meaning that D3 will fetch the contents of the JSON file that it finds at the relative path that we give it, and execute the function that we're calling in the `then()` method once it's been loaded in. We then have access to the object that we're looking at in the `data` argument of our callback.

We're also passing in a function reference here — `type` — which is going to convert all of the values that we're loading in into numbers, which we can work with later:

```
function type(d) {  
  d.apples = Number(d.apples);  
  d.oranges = Number(d.oranges);  
  return d;  
}
```

If we add a `console.log(data);` statement to the top our `d3.json` callback, we can take a look at the data that we're now working with:

```
{apples: Array(5), oranges: Array(5)}  
  apples: Array(5)  
    0: {region: "North", count: "53245"}  
    1: {region: "South", count: "28479"}  
    2: {region: "East", count: "19697"}  
    3: {region: "West", count: "24037"}  
    4: {region: "Central", count: "40245"}  
  oranges: Array(5)  
    0: {region: "North", count: "200"}
```

```
1: {region: "South", count: "200"}  
2: {region: "East", count: "200"}  
3: {region: "West", count: "200"}  
4: {region: "Central", count: "200"}
```

Our data is split into two different arrays here, representing our data for **apples** and **oranges**, respectively.

With this line, we're going to switch the data that we're looking at whenever one of our radio buttons gets clicked:

```
d3.selectAll("input")  
.on("change", update);
```

We'll also need to call the `update()` function on the first run of our visualization, passing in an initial value (with our “apples” array).

```
update("apples");
```

Let's take a look at what our `update()` function is doing. If you're new to D3, this might cause some confusion, since it's one of the most difficult parts of D3 to understand ...

```
function update(value = this.value) {  
  // Join new data  
  const path = svg.selectAll("path")  
    .data(pie(data[value]));  
  
  // Update existing arcs  
  path.transition().duration(200).attrTween("d", arcTween);  
  
  // Enter new arcs  
  path.enter().append("path")  
    .attr("fill", (d, i) => color(i))  
    .attr("d", arc)  
    .attr("stroke", "white")  
    .attr("stroke-width", "6px")  
    .each(function(d) { this._current = d; });  
}
```

Firstly, we're using a default function parameter for `value`. If we're passing in an argument to our `update()` function (when we're running it for the first time), we'll use that string, or otherwise we'll get the value that we want from the click event of our radio inputs.

We're then using the [General Update Pattern](#) in D3 to handle the behavior of our arcs. This usually involves performing a data join, exiting old elements, updating existing elements on the screen, and adding in new elements that were added to our data. In this example, we don't need to worry about exiting elements, since we always have the same number of pie slices on the screen.

First off, there's our data join:

```
// JOIN
const path = svg.selectAll("path")
  .data(pie(data[val]));
```

Every time our visualization updates, this associates a new array of data with our SVGs on the screen. We're passing our data (either the array for "apples" or "oranges") into our `pie()` layout function, which is computing some start and end angles, which can be used to draw our arcs. This `path` variable now contains a special *virtual selection* of all of the arcs on the screen.

Next, we're updating all of the SVGs on the screen that still exist in our data array. We're adding in a [transition](#) here — a fantastic feature of the D3 library — to spread these updates over 200 milliseconds:

```
// UPDATE
path.transition().duration(200)
  .attrTween("d", arcTween);
```

We're using the `attrTween()` method on the `d3.transition()` call to define a custom transition that D3 should use to update the positions of each of its arcs (transitioning with the `d` attribute). We don't need to do this if we're trying to add a transition to most of our attributes, but we need to do this for transitioning between different paths. D3 can't really figure out how to transition between custom paths, so we're using the `arcTween()` function to let D3 know how each of our paths should be drawn at every moment in time.

Here's what this function looks like:

```
function arcTween(a) {
  const i = d3.interpolate(this._current, a);
  this._current = i(1);
  return t => arc(i(t));
}
```

We're using `d3.interpolate()` here to create what's called an **interpolator**. When we call the function that we're storing in the `i` variable with a value between 0 and 1, we'll get back a value that's somewhere between `this._current` and `a`. In this case, `this._current` is an object that contains the start and end angle of the pie slice that we're looking at, and `a` represents the new datapoint that we're updating to.

Once we have the interpolator set up, we're updating the `this._current` value to contain the value that we'll have at the end (`i(a)`), and then we're returning a function that will calculate the path that our arc should contain, based on this `t` value. Our transition will run this function on every tick of its clock (passing in an argument between 0 and 1), and this code will mean that our transition will know where our arcs should be drawn at any point in time.

Finally, our `update()` function needs to add in new elements that weren't in the previous array of data:

```
// ENTER
path.enter().append("path")
  .attr("fill", (d, i) => color(i))
  .attr("d", arc)
  .attr("stroke", "white")
  .attr("stroke-width", "6px")
  .each(function(d) { this._current = d; });
```

This block of code will set the initial positions of each of our arcs, the first time that this update function is run. The `enter()` method here gives us all the elements in our data that need to be added to the screen, and then we can loop over each of these elements with the `attr()` methods, to set the fill and position of each of our arcs. We're also giving each of our arcs a white border, which makes our chart look a little neater. Finally, we're setting the `this._current` property of each of these arcs as the initial value of the item in our data, which we're using in the `arcTween()` function.

Don't worry if you can't follow exactly how this is working, as it's a fairly advanced topic in D3. The great thing about this library is that you don't need to know all of its inner workings to create some powerful stuff with it. As long as you can understand the bits that you need to change, then it's fine to abstract some of the details that aren't completely essential.

That brings us to the next step in the process ...

Adapting Code

Now that we have some code in our local environment, and we understand what it's doing, I'm going to switch out the data that we're looking at, so that it works with the data that we're interested in.

I've included the data that we'll be working with in the `data/` folder of our project. Since this new `incomes.csv` file is in a CSV format this time (it's the kind of file that you can open with Microsoft Excel), I'm going to use the `d3.csv()` function, instead of the `d3.json()` function:

```
d3.csv("data/incomes.csv").then(data => {  
  ...  
});
```

This function does basically the same thing as `d3.json()` — converting our data into a format that we can use. I'm also removing the `type()` initializer function as the second argument here, since that was specific to our old data.

If you add a `console.log(data)` statement to the top of the `d3.csv` callback, you'll be able to see the shape of the data we're working with:

```
(50) [ {...}, {...}, {...}, {...}, {...}, {...}, {...} ... columns: Array(9)]  
  0:  
    1: "12457"  
    2: "32631"  
    3: "56832"  
    4: "92031"  
    5: "202366"  
    average: "79263"  
    top: "350870"  
    total: "396317"  
    year: "2015"  
  1: {1: "11690", 2: "31123", 3: "54104", 4: "87935", 5: "194277", y  
  2: {1: "11797", 2: "31353", 3: "54683", 4: "87989", 5: "196742", y  
  ...
```

We have an array of 50 items, with each item representing a year in our data. For each year, we then have an object, with data for each of the five income groups, as well as a few other fields. We could create a pie chart here for one of these years, but first we'll need to shuffle around our data a little, so that it's in the right format. When we want to write a data join with D3, we need to pass in an array, where each item will be tied to an SVG.

Recall that, in our last example, we had an array with an item for every pie slice that we wanted to display on the screen. Compare this to what we have at the moment, which is an object with the keys of 1 to 5 representing each pie slice that we want to draw.

To fix this, I'm going to add a new function called `prepareData()` to replace the `type()` function that we had previously, which will iterate over every item of our data as it's loaded:

```
function prepareData(d){  
  return {  
    name: d.year,  
    average: parseInt(d.average),  
    values: [  
      {  
        name: "first",  
        value: parseInt(d["1"])  
      },  
      {  
        name: "second",  
        value: parseInt(d["2"])  
      },  
      {  
        name: "third",  
        value: parseInt(d["3"])  
      },  
      {  
        name: "fourth",  
        value: parseInt(d["4"])  
      },  
      {  
        name: "fifth",  
        value: parseInt(d["5"])  
      }  
    ]  
  }  
}  
  
d3.csv("data/incomes.csv", prepareData).then(data => {  
  ...  
});
```

For every year, this function will return an object with a `values` array, which we'll pass into our data join. We're labelling each of these values with a `name` field, and we're giving them a numerical value based on the income values that we had already. We're also keeping track of the average income in each year for

comparison.

At this point, we have our data in a format that we can work with:

```
(50) [ { ... }, { ... }, { ... }, { ... }, { ... }, { ... }, { ... } ... columns: Array(9)]
  0:
    average: 79263
    name: "2015"
    values: Array(5)
      0: {name: "first", value: 12457}
      1: {name: "second", value: 32631}
      2: {name: "third", value: 56832}
      3: {name: "fourth", value: 92031}
      4: {name: "fifth", value: 202366}
    1: {name: "2014", average: 75826, values: Array(5)}
    2: {name: "2013", average: 76513, values: Array(5)}
    ...

```

I'll start off by generating a chart for the first year in our data, and then I'll worry about updating it for the rest of the years.

At the moment, our data starts in the year 2015 and ends in the year 1967, so we'll need to reverse this array before we do anything else:

```
d3.csv("data/incomes.csv", prepareData).then(data => {
  data = data.reverse();
  ...
});
```

Unlike a normal pie chart, for our graph, we want to fix the angles of each of our arcs, and just have the radius change as our visualization updates. To do this, we'll change the `value()` method on our pie layout, so that each pie slice always gets the same angles:

```
const pie = d3.pie()
  .value(1)
  .sort(null);
```

Next, we'll need to update our radius every time our visualization updates. To do this, we'll need to come up with a [scale](#) that we can use. A **scale** is a function in D3 that takes an *input* between two values, which we pass in as the **domain**, and then spits out an *output* between two different values, which we pass in as the **range**. Here's the scale that we'll be using:

```
d3.csv("data/incomes.csv", prepareData).then(data => {
  data = data.reverse();
  const radiusScale = d3.scaleSqrt()
    .domain([0, data[49].values[4].value])
    .range([0, Math.min(width, height) / 2]);
  ...
});
```

We're adding this scale as soon as we have access to our data and we're saying that our input should range between 0 and the largest value in our dataset, which is the income from the richest group in the last year in our data (`data[49].values[4].value`). For the domain, we're setting the interval that our output value should range between.

This means that an input of zero should give us a pixel value of zero, and an input of the largest value in our data should give us a value of half the value of our width or height — whichever is smaller.

Notice that we're also using a *square root scale* here. The reason we're doing this is that we want the area of our pie slices to be proportional to the income of each of our groups, rather than the radius. Since $\text{area} = \pi r^2$, we need to use a square root scale to account for this.

We can then use this scale to update the `outerRadius` value of our arc generator inside our `update()` function:

```
function update(value = this.value) {
  arc.outerRadius(d => radiusScale(d.data.value));
  ...
};
```

Whenever our data changes, this will edit the radius value that we want to use for each of our arcs.

We should also remove our call to `outerRadius` when we initially set up our arc generator, so that we just have this at the top of our file:

```
const arc = d3.arc()
  .innerRadius(0);
```

Finally, we need to make a few edits to this `update()` function, so that everything matches up with our new data:

```

function update(data) {
  arc.outerRadius(d => radiusScale(d.data.value));

  // JOIN
  const path = svg.selectAll("path")
    .data(pie(data.values));

  // UPDATE
  path.transition().duration(200).attrTween("d", arcTween);

  // ENTER
  path.enter().append("path")
    .attr("fill", (d, i) => color(i))
    .attr("d", arc)
    .attr("stroke", "white")
    .attr("stroke-width", "2px")
    .each(function(d) { this._current = d; });
}

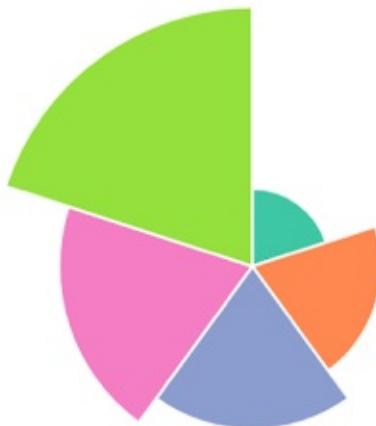
}

```

Since we're not going to be using our radio buttons anymore, I'm just passing in the year-object that we want to use by calling:

```
// Render the first year in our data
update(data[0]);
```

Finally, I'm going to remove the event listener that we set for our form inputs. If all has gone to plan, we should have a beautiful-looking chart for the first year in our data:



Making it Dynamic

The next step is to have our visualization cycle between different years, showing how incomes have been changing over time. We'll do this by adding in call to JavaScript's `setInterval()` function, which we can use to execute some code repeatedly:

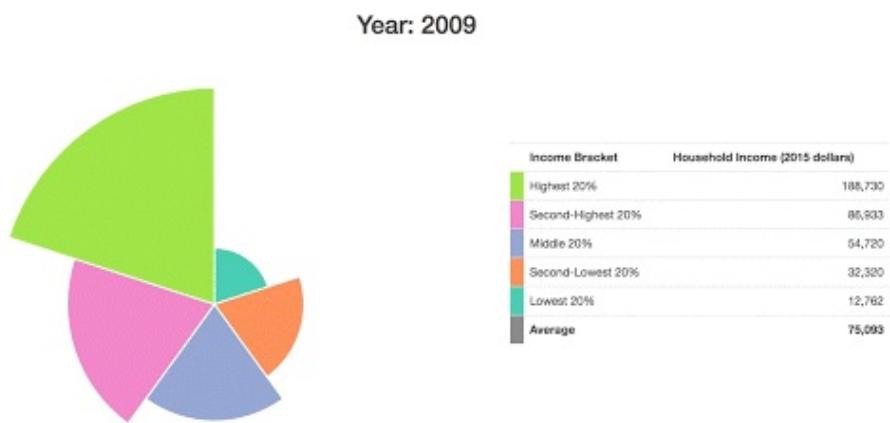
```
d3.csv("data/incomes.csv", prepareData).then(data => {
  ...
  function update(data) {
    ...
  }

  let time = 0;
  let interval = setInterval(step, 200);

  function step() {
    update(data[time]);
    time = (time == 49) ? 0 : time + 1;
  }

  update(data[0]);
});
```

We're setting up a timer in this `time` variable, and every 200ms, this code will run the `step()` function, which will update our chart to the next year's data, and increment the timer by 1. If the timer is at a value of 49 (the last year in our data), it will reset itself. This now gives us a nice loop that will run continuously:



To make things a little more useful. I'll also add in some labels that give us the raw figures. I'll replace all of the HTML code in the body of our file with this:

```

<h2>Year: <span id="year"></span></h2>

<div class="container" id="page-main">
  <div class="row">
    <div class="col-md-7">
      <div id="chart-area"></div>
    </div>

    <div class="col-md-5">
      <table class="table">
        <tbody>
          <tr>
            <th></th>
            <th>Income Bracket</th>
            <th>Household Income (2015 dollars)</th>
          </tr>
          <tr>
            <td id="leg5"></td>
            <td>Highest 20%</td>
            <td class="money-cell"><span id="fig5"></span></td>
          </tr>
          <tr>
            <td id="leg4"></td>
            <td>Second-Highest 20%</td>
            <td class="money-cell"><span id="fig4"></span></td>
          </tr>
          <tr>
            <td id="leg3"></td>
            <td>Middle 20%</td>
            <td class="money-cell"><span id="fig3"></span></td>
          </tr>
          <tr>
            <td id="leg2"></td>
            <td>Second-Lowest 20%</td>
            <td class="money-cell"><span id="fig2"></span></td>
          </tr>
          <tr>
            <td id="leg1"></td>
            <td>Lowest 20%</td>
            <td class="money-cell"><span id="fig1"></span></td>
          </tr>
        </tbody>
        <tfoot>
          <tr>
            <td id="avLeg"></td>
            <th>Average</th>
            <th class="money-cell"><span id="avFig"></span></th>
          </tr>
        </tfoot>
      </table>
    </div>
  </div>
</div>

```

```
        </div>
    </div>
</div>
```

We're structuring our page here using [Bootstrap's grid system](#), which lets us neatly format our page elements into boxes.

I'll then update all of this with [jQuery](#) whenever our data changes:

```
function updateHTML(data) {
    // Update title
    $("#year").text(data.name);

    // Update table values
    $("#fig1").html(data.values[0].value.toLocaleString());
    $("#fig2").html(data.values[1].value.toLocaleString());
    $("#fig3").html(data.values[2].value.toLocaleString());
    $("#fig4").html(data.values[3].value.toLocaleString());
    $("#fig5").html(data.values[4].value.toLocaleString());
    $("#avFig").html(data.average.toLocaleString());
}

d3.csv("data/incomes.csv", prepareData).then(data => {
    ...
    function update(data) {
        updateHTML(data);
        ...
    }
    ...
}
```

I'll also make a few edits to the CSS at the top of our file, which will give us a legend for each of our arcs, and also center our heading:

```
<style>
#chart-area svg {
    margin:auto;
    display:inherit;
}

.money-cell { text-align: right; }
h2 { text-align: center; }

#leg1 { background-color: #66c2a5; }
#leg2 { background-color: #fc8d62; }
#leg3 { background-color: #8da0cb; }
#leg4 { background-color: #e78ac3; }
#leg5 { background-color: #a6d854; }
```

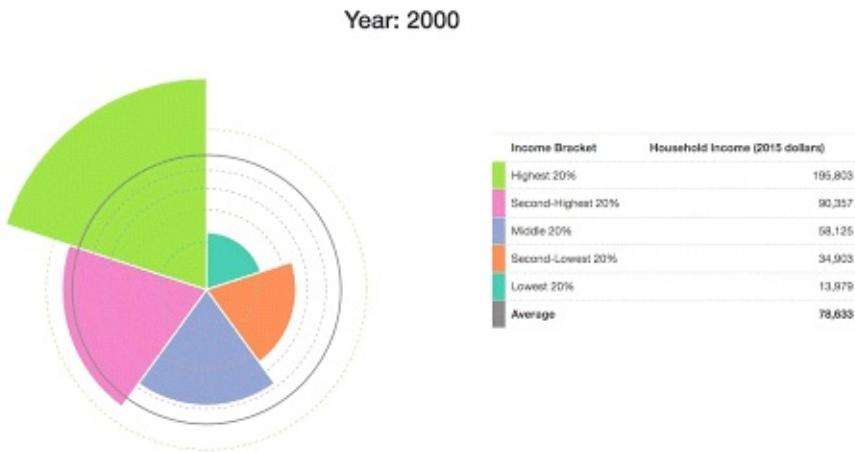
```

#avLeg { background-color: grey; }

@media screen and (min-width: 768px) {
    table { margin-top: 100px; }
}

```

What we end up with is something rather presentable:



Since it's pretty tough to see how these arcs have changed over time here, I want to add in some grid lines to show what the income distribution looked like in the first year of our data:

```

d3.csv("data/incomes.csv", prepareData).then(data => {
    ...
    update(data[0]);

    data[0].values.forEach((d, i) => {
        svg.append("circle")
            .attr("fill", "none")
            .attr("cx", 0)
            .attr("cy", 0)
            .attr("r", radiusScale(d.value))
            .attr("stroke", color(i))
            .attr("stroke-dasharray", "4,4");
    });
});

```

I'm using the `Array.forEach()` method to accomplish this, although I could have also gone with D3's usual *General Update Pattern* again

(JOIN/EXIT/UPDATE/ENTER).

I also want to add in a line to show the average income in the US, which I'll update every year. First, I'll add the average line for the first time:

```
d3.csv("data/incomes.csv", prepareData).then(data => {
  ...
  data[0].values.forEach((d, i) => {
    svg.append("circle")
      .attr("fill", "none")
      .attr("cx", 0)
      .attr("cy", 0)
      .attr("r", radiusScale(d.value))
      .attr("stroke", color(i))
      .attr("stroke-dasharray", "4,4");
  });

  svg.append("circle")
    .attr("class", "averageLine")
    .attr("fill", "none")
    .attr("cx", 0)
    .attr("cy", 0)
    .attr("stroke", "grey")
    .attr("stroke-width", "2px");
});
```

Then I'll update this at the end of our `update()` function whenever the year changes:

```
function update(data) {
  ...
  svg.select(".averageLine").transition().duration(200)
    .attr("r", radiusScale(data.average));
}
```

I should note that it's important for us to add each of these circles *after* our first call to `update()`, because otherwise they'll end up being rendered *behind* each of our arc paths (SVG layers are determined by the order in which they're added to the screen, rather than by their z-index).

At this point, we have something that conveys the data that we're working with a bit more clearly:

Year: 2000



Making it Interactive

As a last step, I want us to add in some controls to let the user dig down into a particular year. I want to add in a *Play/Pause* button, as well as a year slider, allowing the user to pick a particular date to look at.

Here's the HTML that I'll use to add these elements onto the screen:

```
<div class="container" id="page-main">
  <div id="controls" class="row">
    <div class="col-md-12">
      <button id="play-button" class="btn btn-primary">Play</button>
      <div id="slider-div">
        <label>Year: <span id="year-label"></span></label>
        <div id="date-slider"></div>
      </div>
    </div>
  </div>
  ...
</div>
```

We'll need to add some event listeners to both of these elements, to engineer the behavior that we're looking for.

First off, I want to define the behavior of our *Play/Pause* button. We'll need to replace the code that we wrote for our interval earlier to allow us to stop and start the timer with the button. I'll assume that the visualization starts in a "Paused" state, and that we need to press this button to kick things off.

```

function update(data) {
  ...

  let time = 0;
  let interval;

  function step() {
    update(data[time]);
    time = (time == 49) ? 0 : time + 1;
  }

  $("#play-button").on("click", function() {
    const button = $(this);
    if (button.text() === "Play"){
      button.text("Pause");
      interval = setInterval(step, 200);
    } else {
      button.text("Play");
      clearInterval(interval);
    }
  });
  ...
}

```

Whenever our button gets clicked, our `if/else` block here is going to define a different behavior, depending on whether our button is a “Play” button or a “Pause” button. If the button that we’re clicking says “Play”, we’ll change the button to a “Pause” button, and start our interval loop going. Alternatively, if the button is a “Pause” button, we’ll change its text to “Play”, and we’ll use the `clearInterval()` function to stop the loop from running.

For our slider, I want to use the slider that comes with the [jQuery UI library](#). I’m including this in our HTML, and I’m going to write a few lines to add this to the screen:

```

function update(data) {
  ...
  $("#date-slider").slider({
    max: 49,
    min: 0,
    step: 1,
    slide: (event, ui) => {
      time = ui.value;
      update(data[time]);
    }
  });
}

```

```
    update(data[0]);
    ...
}
```

Here, we're using the `slide` option to attach an event listener to the slider. Whenever our slider gets moved to another value, we're updating our timer to this new value, and we're running our `update()` function at that year in our data.

We can add this line at the end of our `update()` function so that our slider moves along to the right year when our loop is running:

```
function update(data) {
    ...

    // Update slider position
    $("#date-slider").slider("value", time);
}
```

I'll also add in a line to our `updateHTML()` function (which runs whenever our visualization changes), which can adjust the value of the label based on the current year in the data:

```
function updateHTML(data) {
    // Update title
    $("#year").text(data.name);

    // Update slider label
    $("#year-label").text(data.name);

    // Update table values
    $("#fig1").html(data.values[0].value.toLocaleString());
    ...
}
```

I'll throw in a few more lines to our CSS to make everything look a little neater:

```
<style>
    ...
    @media screen and (min-width: 768px) {
        table { margin-top: 100px; }
    }

    #page-main { margin-top: 10px; }
    #controls { margin-bottom: 20px; }

    #play-button {
```

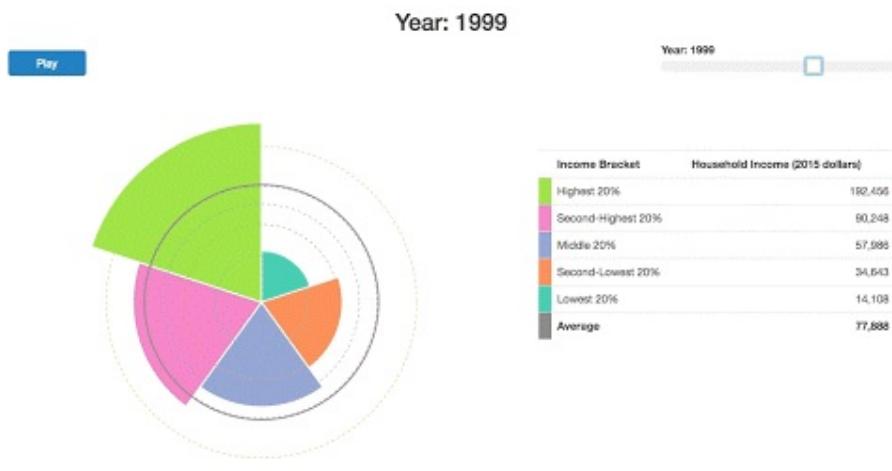
```

        margin-top: 10px;
        width: 100px;
    }

    #slider-div {
        width:300px;
        float:right;
    }

```

And there we have it — our finished product — a fully functioning interactive data visualization, with everything working as expected.



Hopefully, this tutorial demonstrated the real power of D3, letting you create absolutely anything you can imagine.

Getting started with D3 from scratch is always a tough process, but the rewards are worth it. If you want to learn how to create custom visualizations of your own, here are a few online resources that you might find helpful:

- An overview of [SitePoint's D3.js content](#).
- The [introduction to the library](#) on D3's homepage. This runs through some of the most basic commands, showing you how to make your first few steps in D3.
- “[Let's Make a Bar Chart](#)” by Mike Bostock — the creator of D3 — showing beginners how to make one of the simplest graphs in the library.
- [D3.js in Action by Elijah Meeks](#) (\$35), which is a solid introductory textbook that goes into a lot of detail.
- [D3's Slack channel](#) is very welcoming to newcomers to D3. It also has a

“learning materials” section with a collection of great resources.

- [This online Udemy course](#) (\$20), which covers everything in the library in a series of video lectures. This is aimed at JavaScript developers, and includes four cool projects.
- The multitude of example visualizations that are available at bl.ocks.org and blockbuilder.org.
- The [D3 API Reference](#), which gives a thorough technical explanation of everything that D3 has to offer.

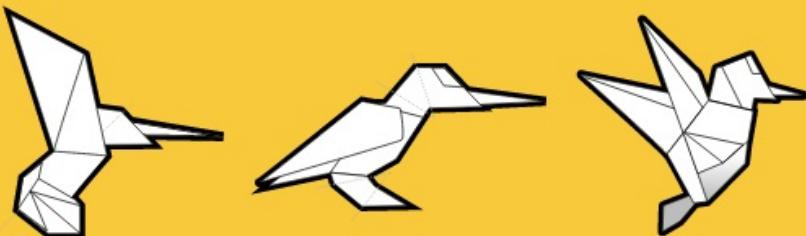
Finished Code

Don’t forget, if you want to see the finished version of the code that I was using in the chapter, then you can [find it on our GitHub repo](#).

Book 4: Modern JavaScript Tools & Skills



MODERN JAVASCRIPT TOOLS & SKILLS



I0.

II.

I2.

A COMPLETE ECOSYSTEM

Chapter 1: A Beginner's Guide to Babel

by James Kolce

This article introduces [Babel](#), a JavaScript compiler that allows developers to use next-generation JavaScript today.

It can be frustrating to write JavaScript when building web applications. We have to think about the features available in the browsers we're targeting and what happens when a feature isn't implemented. Some people would recommend simply not using it, which is a painful experience most of the time if we're building something complicated.

Thankfully, some tools allow us to stop worrying about what's supported and just write the best code we can. They're called transpilers. A **transpiler** is a tool that takes source code as input and produces new source code as output, with a different syntax but semantically as close as possible — or ideally equivalent — to the original.

Babel is pretty much the standard transpiler to translate modern JavaScript (ES2015+) into compatible implementations that run in old browsers. It's the perfect solution if you just want to concentrate on writing JavaScript.

And although the main goal of Babel is to translate the latest standards of ECMAScript (ES) for old — or sometimes current — browsers, it can do more. There's an ecosystem of presets and plugins that make possible the addition of non-standard features as well. Each plugin makes a new feature/transformation available for your code, and presets are just a collection of plugins.

Getting Started

There are different ways to set up Babel depending on your project and the tools you use. In this article, we're going to explain how to set up Babel using the CLI, although if you're using a build system or framework, you can check out

specific instructions on the [official site](#). Most of the time the CLI is the fastest and easiest way to get started, so if you're a first-time user, feel free to continue.

The first step to set up Babel in a project is to install the package using npm and add it as a dev dependency. Assuming you have a working Node.js environment already in place, it's just a matter of running the following in your terminal:

```
mkdir babel-test
cd babel-test
npm init -y
npm install --save-dev babel-cli
```

This will create a directory (`babel-test`) change into the directory, initialize an npm project (thus creating a `package.json` file) and then install the `babel-cli` as a dev dependency.

If you need any help with the above, please consult our tutorials on [installing Node](#) and [working with npm](#).

Next, we can open `package.json` and add a `build` command to our npm scripts:

```
"scripts": {
  "build": "babel src -d dist"
}
```

This will take the source files from the `src` directory and output the result in a `dist` directory. Then we can execute it as:

```
npm run build
```

But wait! Before running Babel we must install and set up the plugins that will transform our code. The easiest and quickest way to do this is to add the [Env preset](#), which selects the appropriate plugins depending on the target browsers that you indicate. It can be installed using:

```
npm install babel-preset-env --save-dev
```

Then create a `.babelrc` file in the root of your project and add the preset:

```
{
  "presets": ["env"]
}
```

The `.babelrc` file is the place where you put all your settings for Babel. You'll be using this primarily for setting up presets and plugins, but a lot more options are available. You can check the complete list in the [Babel API page](#).

Please note that, depending on your operating system, files beginning with a dot will be hidden by default. If this is problematic for you (or if you just prefer fewer files), you can put your Babel settings in the package.json file, under a `babel` key, like so:

```
{  
  "name": "babel-test",  
  "version": "1.0.0",  
  "babel": {  
    // config  
  }  
}
```

Finally, let's create the directories and files Babel is expecting to find:

```
mkdir src dist
```

And give it something to transform:

```
let a = 1;  
let b = 2;  
[a, b] = [b, a];  
console.log(a);  
console.log(b);
```

This example uses [destructuring assignment](#) to swap the values of two variables.

Running Babel

Now that you have a ready-to-use Babel installation, you can execute the `build` command to run the compilation process:

```
npm run build
```

This will take the code from `src/main.js`, transform it to ES5 code and output the transformed code to `dist/main.js`.

Here's what it produced:

```
"use strict";

var a = 1;
var b = 2;
var _ref = [b, a];
a = _ref[0];
b = _ref[1];

console.log(a);
console.log(b);
```

As you can see, `let` has been replaced by `var` and Babel has introduced a temporary variable (denoted by the underscore) to facilitate the swap.

And that's it. The code that you write in the `src` directory will be translated to previous versions of the language. By default, if you don't add any options to the preset, it will load all the transformations. You can also indicate the target browsers as follows:

```
{
  "presets": [
    ["env", {
      "targets": {
        "browsers": ["last 2 versions", "safari >= 7"]
      }
    }]
  ]
}
```

This will load the required transformations to support the latest two versions of each browser and Safari greater or equal to version 7. You can find the available options for the target browsers in the [Browserlist repository](#).

Babel Ecosystem: A Quick Overview

As you noticed in the previous section, Babel won't do anything by itself when you install it. We have to install a set of plugins to obtain the desired behavior, or we can use presets, which are predefined sets of plugins.

Usually, each feature that you want to include will be in the form of a plugin. Some examples for ES2015 include:

- [constants](#)

- [arrow functions](#)
- [block-scoped functions](#)
- [classes](#)
- [for-of](#)
- [spread](#)
- [template literals](#)

See the [Plugins page](#) in the Babel Docs for a complete list.

But sometimes you don't want to include all the plugins one by one. So there are prebuilt presets that will facilitate the process of installing each plugin.

The three official presets currently available are:

- [Env](#)
- [React](#)
- [Flow](#)

Env is the most frequently used and the one we've used here. It automatically loads all the necessary transformations to make your code compatible depending on the targeted browsers.

The **React** preset transforms code typically found in React projects, mainly adding compatibility with [Flow](#) annotations and [JSX](#).

And finally, the **Flow** preset is used to clean up the code from [Flow](#) type annotations (although it doesn't check whether the types are valid or not.)

Babel Polyfill

There are JavaScript features that can't be transformed syntactically, usually because there's no equivalent functionality implemented — for example, Promises and generator functions.

Those kinds of features have to be implemented in the browser by a library to be used in your code, and that's the work of a polyfill.

The Babel polyfill is composed by [core-js](#) and the [Regenerator](#) runtime. Together, they cover all the features in ES2015+.

Advanced Use

As mentioned, Babel can also be used to transform features that haven't yet been implemented in the language. A good example of this is the class fields proposal (currently at [TC39 stage 3: candidate](#)). This is particularly popular among React devs, as it removes the necessity to explicitly bind methods to a particular component, and also means that a component's state can be declared as a class field (potentially eliminating the need for a constructor).

For those of you wanting to use class fields today, you would need to add the [babel-plugin-transform-class-properties](#) as a dev dependency:

```
npm install --save-dev babel-plugin-transform-class-properties
```

You'd also update your `.babelrc` file as follows:

```
{  
  "presets": ["env"],  
  "plugins": ["transform-class-properties"]  
}
```

Now you can write:

```
class App extends Component {  
  state = { count: 0 };  
  
  incCount = () => {  
    this.setState(ps => ({ count: ps.count + 1 }));  
  };  
  
  render() {  
    return (  
      <div>  
        <p>{ this.state.count }</p>  
        <button onClick={this.incCount}>add one</button>  
      </div>  
    );  
  }  
}
```

And it doesn't stop there. You can also use Babel to add new features of your own to the language, as our tutorial [Understanding ASTs by Building Your Own Babel Plugin](#) demonstrates.

Alternatives

Writing modern web applications sometimes requires more than the features available in JavaScript. Other languages can also be translated to compatible JavaScript but also implement other useful features.

The most popular option is [TypeScript](#), which is regular JavaScript that implements modern ES features but also adds others, especially regarding type safety.

On the other extreme, there are entirely different languages across different categories, from the functional ones like PureScript to the object-oriented like Dart.

For a deeper overview of alternative languages, take a look at [10 Languages that Compile to JavaScript](#).

Conclusion

Babel is a great option for writing modern applications while still serving up JavaScript that can be understood by all developers and the wide range of browsers the code needs to run in.

Babel is not only useful for transforming ES2015+ to previous versions of the language — both in the browser and on platforms such as Node.js — but also for adding new features that aren't part of the standard. To see what I mean, just take a look at the npm directory to find all the available Babel [plugins](#) or [presets](#).

As JavaScript is evolving at such a rapid pace, it's obvious that browser manufacturers will need a while to implement the latest features. Giving Babel a place in your toolkit means that you can write cutting-edge JavaScript today, safe in the knowledge that you're not abandoning any of your users. What's not to love?

Chapter 2: A Beginner's Guide to Webpack 4 and Module Bundling

by **Mark Brown**

The [Webpack 4 docs](#) state that:

Webpack is a module bundler. Its main purpose is to bundle JavaScript files for usage in a browser, yet it is also capable of transforming, bundling, or packaging just about any resource or asset.

[Webpack](#) has become one of the most important tools for modern web development. It's primarily a module bundler for your JavaScript, but it can be taught to transform all of your front-end assets like HTML, CSS, even images. It can give you more control over the number of HTTP requests your app is making and allows you to use other flavors of those assets (Pug, Sass, and ES8, for example). Webpack also allows you to easily consume packages from npm.

This article is aimed at those who are new to Webpack, and will cover initial setup and configuration, modules, loaders, plugins, code splitting and hot module replacement. If you find video tutorials helpful, I can highly recommend Glen Maddern's [Webpack from First Principles](#) as a starting point to understand what it is that makes Webpack special. It's a little old now, but the principles are still the same, and it's a great introduction.

Example Code

To follow along at home, you'll need to have [Node.js installed](#). You can also [download the demo app from GitHub](#).

Setup

Let's initialize a new project with npm and install webpack and webpack-cli:

```
mkdir webpack-demo && cd webpack-demo
```

```
npm init -y
npm install --save-dev webpack webpack-cli
```

Next we'll create the following directory structure and contents:

```
webpack-demo
|- package.json
+ |- webpack.config.js
+ |- /src
+   |- index.js
+ |- /dist
+   |- index.html
```

dist/index.html

```
<!doctype html>
<html>
  <head>
    <title>Hello Webpack</title>
  </head>
  <body>
    <script src="bundle.js"></script>
  </body>
</html>
```

src/index.js

```
const root = document.createElement("div")
root.innerHTML = `<p>Hello Webpack.</p>`
document.body.appendChild(root)
```

webpack.config.js

```
const path = require('path')

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
}
```

This tells Webpack to compile the code in our entry point `src/index.js` and output a bundle in `/dist/bundle.js`. Let's add an [npm script](#) for running Webpack.

package.json

```
{  
  ...  
  "scripts": {  
    -   "test": "echo \"Error: no test specified\" && exit 1"  
    +   "develop": "webpack --mode development --watch",  
    +   "build": "webpack --mode production"  
  },  
  ...  
}
```

Using the `npm run develop` command, we can create our first bundle!

Asset	Size	Chunks	Chunk Names
bundle.js	2.92 KiB	main [emitted]	main

You should now be able to load `dist/index.html` in your browser and be greeted with “Hello Webpack”.

Open up `dist/bundle.js` to see what Webpack has done. At the top is Webpack’s module bootstrapping code, and right at the bottom is our module. You may not be colored impressed just yet, but if you’ve come this far you can now start using ES Modules, and Webpack will be able to produce a bundle for production that will work in all browsers.

Restart the build with `ctrl + c` and run `npm run build` to compile our bundle in *production mode*.

Asset	Size	Chunks	Chunk Names
bundle.js	647 bytes	main [emitted]	main

Notice that the bundle size has come down from **2.92 KiB** to **647 bytes**.

Take another look at `dist/bundle.js` and you’ll see an ugly mess of code. Our bundle has been minified with UglifyJS: the code will run exactly the same, but it’s done with the smallest file size possible.

- `--mode development` optimizes for build speed and debugging
- `--mode production` optimizes for execution speed at runtime and output file size.

Modules

Using ES Modules, you can split up your large programs into many small, self-contained programs.

Out of the box, Webpack knows how to consume ES Modules using `import` and `export` statements. As an example, let's try this out now by installing [lodash-es](#) and adding a second module:

```
npm install --save-dev lodash-es
```

src/index.js

```
import { groupBy } from "lodash-es"
import people from "./people"

const managerGroups = groupBy(people, "manager")

const root = document.createElement("div")
root.innerHTML = `<pre>${JSON.stringify(managerGroups, null, 2)}</pre>
document.body.appendChild(root)
```

src/people.js

```
const people = [
  {
    manager: "Jen",
    name: "Bob"
  },
  {
    manager: "Jen",
    name: "Sue"
  },
  {
    manager: "Bob",
    name: "Shirley"
  }
]

export default people
```

Run `npm run develop` to start Webpack and refresh `index.html`. You should see an array of people grouped by manager printed to the screen.

Imports Without a Relative Patch

Imports without a relative path like '`es-lodash`' are modules from npm installed to `/node_modules`. Your own modules will always need a relative path like '`./people`', as this is how you can tell them apart.

Notice in the console that our bundle size has increased to **1.41 MiB!** This is worth keeping an eye on, though in this case there's no cause for concern. Using `npm run build` to compile in *production* mode, all of the unused lodash modules from `lodash-es` are removed from bundle. This process of removing unused imports is known as **tree-shaking**, and is something you get for free with Webpack.

```
> npm run develop

Asset      Size      Chunks          Chunk Names
bundle.js  1.41 MiB  main  [emitted]  [big]  main
```

```
> npm run build

Asset      Size      Chunks          Chunk Names
bundle.js  16.7 KiB  0   [emitted]  main
```

Loaders

Loaders let you run preprocessors on files as they're imported. This allows you to bundle static resources *beyond JavaScript*, but let's look at what can be done when loading `.js` modules first.

Let's keep our code modern by running all `.js` files through the next-generation JavaScript transpiler [Babel](#):

```
npm install --save-dev "babel-loader@^8.0.0-beta" @babel/core @babel
```

webpack.config.js

```
const path = require('path')

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
```

```
+   },
+   module: {
+     rules: [
+       {
+         test: /\.js$/,
+         exclude: /(node_modules|bower_components)/,
+         use: [
+           {
+             loader: 'babel-loader',
+           }
+         ]
+       }
+     ]
+   }
}
```

.babelrc

```
{
  "presets": [
    ["@babel/env", {
      "modules": false
    }]
  ],
  "plugins": ["syntax-dynamic-import"]
}
```

This config prevents Babel from transpiling `import` and `export` statements into ES5, and enables dynamic imports — which we'll look at later in the section on Code Splitting.

We're now free to use modern language features, and they'll be compiled down to ES5 that runs in all browsers.

Sass

Loaders can be chained together into a series of transforms. A good way to demonstrate how this works is by importing Sass from our JavaScript:

```
npm install --save-dev style-loader css-loader sass-loader node-sass
```

webpack.config.js

```
module.exports = {
  ...
  module: {
    rules: [

```

```
...
{
  test: /\.scss$/,
  use: [
    {
      loader: 'style-loader'
    },
    {
      loader: 'css-loader'
    },
    {
      loader: 'sass-loader'
    }
  ]
}
```

These loaders are processed in reverse order:

- `sass-loader` transforms Sass into CSS.
- `css-loader` parses the CSS into JavaScript and resolves any dependencies.
- `style-loader` outputs our CSS into a `<style>` tag in the document.

You can think of these as function calls. The output of one loader feeds as input into the next:

```
styleLoader(cssLoader(sassLoader("source")))
```

Let's add a Sass source file and import it as a module.

src/style.scss

```
$bluegrey: #2b3a42;

pre {
  padding: 8px 16px;
  background: $bluegrey;
  color: #e1e6e9;
  font-family: Menlo, Courier, monospace;
  font-size: 13px;
  line-height: 1.5;
  text-shadow: 0 1px 0 rgba(23, 31, 35, 0.5);
  border-radius: 3px;
}
```

src/index.js

```
import { groupBy } from 'lodash-es'
```

```
import people from './people'  
+ import './style.scss'  
...
```

Restart the build with `ctrl + c` and `npm run develop`. Refresh `index.html` in the browser and you should see some styling.

CSS in JS

We just imported a Sass file from our JavaScript, as a module.

Open up `dist/bundle.js` and search for “`pre {`”. Indeed, our Sass has been compiled to a string of CSS and saved as a module within our bundle. When we import this module into our JavaScript, `style-loader` outputs that string into an embedded `<style>` tag.

Why would you do such a thing?

I won’t delve too far into this topic here, but here are a few reasons to consider:

- A JavaScript component you may want to include in your project may *depend* on other assets to function properly (HTML, CSS, Images, SVG). If these can all be bundled together, it’s far easier to import and use.
- Dead code elimination: When a JS component is no longer imported by your code, the CSS will no longer be imported either. The bundle produced will only ever contain code that does something.
- CSS Modules: The global namespace of CSS makes it very difficult to be confident that a change to your CSS will not have any side effects. [CSS modules](#) change this by making CSS local by default and exposing unique class names that you can reference in your JavaScript.
- Bring down the number of HTTP requests by bundling/splitting code in clever ways.

Images

The last example of loaders we’ll look at is the handling of images with `file-loader`.

In a standard HTML document, images are fetched when the browser encounters an `img` tag or an element with a `background-image` property. With Webpack, you can optimize this in the case of small images by storing the source of the images as strings inside your JavaScript. By doing this, you preload them and the browser won't have to fetch them with separate requests later:

```
npm install --save-dev file-loader
```

webpack.config.js

```
module.exports = {
  ...
  module: {
    rules: [
      ...
      {
        test: /\.(png|svg|jpg|gif)$/,
        use: [
          {
            loader: 'file-loader'
          }
        ]
      }
    ]
  }
}
```

Download a [test image](#) with this command:

```
curl https://raw.githubusercontent.com/sitepoint-editors/webpack-demo/10.0.0/test-image.png
```

Restart the build with `ctrl + c` and `npm run develop` and you'll now be able to import images as modules!

src/index.js

```
import { groupBy } from 'lodash-es'
import people from './people'

import './style.scss'
+ import './image-example'

...
```

src/image-example.js

```
import codeURL from "./code.png"

const img = document.createElement("img")
img.src = codeURL
img.style = "background: #2B3A42; padding: 20px"
img.width = 32
document.body.appendChild(img)
```

This will include an image where the `src` attribute contains a [data URI](#) of the image itself:

```
 {
  import(/* webpackChunkName: "chat" */ "./chat").then(chat => {
    chat.init()
  })
}
```

src/chat.js

```
import people from "./people"

export function init() {
  const root = document.createElement("div")
  root.innerHTML = `<p>There are ${people.length} people in the room</p>`
  document.body.appendChild(root)
}
```

src/app.scss

```
button {
  padding: 10px;
  background: #24b47e;
  border: 1px solid rgba(#000, .1);
  border-width: 1px 1px 3px;
  border-radius: 3px;
  font: inherit;
```

```
color: #fff;  
cursor: pointer;  
text-shadow: 0 1px 0 rgba(#000, .3), 0 1px 1px rgba(#000, .2);  
}
```

Not Webpack Specific Syntax

Note: Despite the `/* webpackChunkName */` comment for giving the bundle a name, this syntax is *not* Webpack specific. It's the [proposed syntax for dynamic imports](#) intended to be supported directly in the browser.

Let's run `npm run build` and see what this generates:

Asset	Size	Chunks	Chunk Names
chat.bundle.js	377 bytes	0 [emitted]	chat
app.bundle.js	7.65 KiB	1 [emitted]	app

As our entry bundle has changed, we'll need to update our path to it as well.

dist/index.html

```
<!doctype html>  
<html>  
  <head>  
    <title>Hello Webpack</title>  
  </head>  
  <body>  
-    <script src="bundle.js"></script>  
+    <script src="app.bundle.js"></script>  
  </body>  
</html>
```

Let's start up a server from the dist directory to see this in action:

```
cd dist  
npx serve
```

Open `http://localhost:5000` in the browser and see what happens. Only `bundle.js` is fetched initially. When the button is clicked, the `chat` module is imported and initialized.

[Open chat](#)

There are 3 people in the room.

St...	M...	File	S...	0 ms		
● 200	GET	app.bundle.js	7.65 KB	1 → 28 ms		
● 200	GET	chat.bundle.js	377 B			

With very little effort, we've added dynamic code splitting and lazy loading of modules to our app. This is a great starting point for building a highly performant web app.

Plugins

While *loaders* operate transforms on single files, *plugins* operate across larger chunks of code.

Now that we're bundling our code, external modules *and* static assets, our bundle will grow — *quickly*. Plugins are here to help us split our code in clever ways and optimize things for production.

Without knowing it, we've actually already used many [default Webpack plugins with “mode”](#)

development

- Provides `process.env.NODE_ENV` with value “development”
- `NamedModulesPlugin`

production

- Provides `process.env.NODE_ENV` with value “production”
- `UglifyJsPlugin`
- `ModuleConcatenationPlugin`
- `NoEmitOnErrorsPlugin`

Production

Before adding additional plugins, we'll first split our config up so that we can apply plugins specific to each environment.

Rename `webpack.config.js` to `webpack.common.js` and add a config file for development and production.

```
- | - webpack.config.js
+ | - webpack.common.js
+ | - webpack.dev.js
+ | - webpack.prod.js
```

We'll use `webpack-merge` to combine our common config with the environment-specific config:

```
npm install --save-dev webpack-merge
```

webpack.dev.js

```
const merge = require('webpack-merge')
const common = require('./webpack.common.js')

module.exports = merge(common, {
  mode: 'development'
})
```

webpack.prod.js

```
const merge = require('webpack-merge')
const common = require('./webpack.common.js')

module.exports = merge(common, {
  mode: 'production'
})
```

package.json

```
"scripts": {
-   "develop": "webpack --watch --mode development",
-   "build": "webpack --mode production"
+   "develop": "webpack --watch --config webpack.dev.js",
+   "build": "webpack --config webpack.prod.js"
},
```

Now we can add plugins specific to development into `webpack.dev.js` and plugins specific to production in `webpack.prod.js`.

Split CSS

It's considered best practice to split your CSS from your JavaScript when bundling for production using [ExtractTextWebpackPlugin](#).

The current .scss loaders are perfect for development, so we'll move those from webpack.common.js into webpack.dev.js and add ExtractTextWebpackPlugin to webpack.prod.js only.

```
npm install --save-dev extract-text-webpack-plugin@4.0.0-beta.0
```

webpack.common.js

```
...
module.exports = {
  ...
  module: {
    rules: [
      ...
      {
        test: /\.scss$/,
        use: [
          {
            loader: 'style-loader'
          },
          {
            loader: 'css-loader'
          },
          {
            loader: 'sass-loader'
          }
        ]
      },
      ...
    ]
  }
}
```

webpack.dev.js

```
const merge = require('webpack-merge')
const common = require('./webpack.common.js')

module.exports = merge(common, {
  mode: 'development',
+  module: {
+    rules: [
+      {

```

```

+         test: /\.scss$/,
+         use: [
+           {
+             loader: 'style-loader'
+           },
+             loader: 'css-loader'
+           },
+             loader: 'sass-loader'
+           }
+         ]
+       ]
+     }
+   )
)

```

webpack.prod.js

```

const merge = require('webpack-merge')
+ const ExtractTextPlugin = require('extract-text-webpack-plugin')
const common = require('./webpack.common.js')

module.exports = merge(common, {
  mode: 'production',
+  module: {
+    rules: [
+      {
+        test: /\.scss$/,
+        use: ExtractTextPlugin.extract({
+          fallback: 'style-loader',
+          use: ['css-loader', 'sass-loader']
+        })
+      }
+    ],
+    plugins: [
+      new ExtractTextPlugin('style.css')
+    ]
+  }
})

```

Let's compare the output of our two build scripts:

```

> npm run develop

Asset            Size      Chunks      Chunk Names
app.bundle.js  28.5 KiB  app      [emitted]  app
chat.bundle.js  1.4 KiB  chat      [emitted]  chat

```

```
> npm run build
```

Asset	Size	Chunks	Chunk	Names
chat.bundle.js	375 bytes	0	[emitted]	chat
app.bundle.js	1.82 KiB	1	[emitted]	app
style.css	424 bytes	1	[emitted]	app

Now that our CSS is extracted from our JavaScript bundle for production, we need to <link> to it from our HTML.

dist/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Code Splitting</title>
+    <link href="style.css" rel="stylesheet">
  </head>
  <body>
    <script type="text/javascript" src="app.bundle.js"></script>
  </body>
</html>
```

This allows for parallel download of the CSS and JavaScript in the browser, so will be faster-loading than a single bundle. It also allows the styles to be displayed before the JavaScript finishes downloading.

Generating HTML

Whenever our outputs have changed, we've had to keep updating `index.html` to reference the new file paths. This is precisely what `html-webpack-plugin` was created to do for us automatically.

We may as well add `clean-webpack-plugin` at the same time to clear out our `/dist` directory before each build.

```
npm install --save-dev html-webpack-plugin clean-webpack-plugin
```

webpack.common.js

```
const path = require('path')
+ const CleanWebpackPlugin = require('clean-webpack-plugin');
+ const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
```

```
...
+   plugins: [
+     new CleanWebpackPlugin(['dist']),
+     new HtmlWebpackPlugin({
+       title: 'My killer app'
+     })
+   ]
}
```

Now every time we build, dist will be cleared out. We'll now see `index.html` output too, with the correct paths to our entry bundles.

Running `npm run develop` produces this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>My killer app</title>
  </head>
  <body>
    <script type="text/javascript" src="app.bundle.js"></script>
  </body>
</html>
```

And `npm run build` produces this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>My killer app</title>
    <link href="style.css" rel="stylesheet">
  </head>
  <body>
    <script type="text/javascript" src="app.bundle.js"></script>
  </body>
</html>
```

Development

The `webpack-dev-server` provides you with a simple web server and gives you *live reloading*, so you don't need to manually refresh the page to see changes.

```
npm install --save-dev webpack-dev-server
```

package.json

```
{  
  ...  
  "scripts": {  
    - "develop": "webpack --watch --config webpack.dev.js",  
    + "develop": "webpack-dev-server --config webpack.dev.js",  
    }  
  ...  
}
```

```
> npm run develop  
i wds: Project is running at http://localhost:8080/  
i wds: webpack output is served from /
```

Open up <http://localhost:8080/> in the browser and make a change to one of the JavaScript or CSS files. You should see it build and refresh automatically.

HotModuleReplacement

The `HotModuleReplacement` plugin goes one step further than Live Reloading and swaps out modules at runtime *without the refresh*. When configured correctly, this saves a huge amount of time during development of single page apps. Where you have a lot of state in the page, you can make incremental changes to components, and only the changed modules are replaced and updated.

webpack.dev.js

```
+ const webpack = require('webpack')  
const merge = require('webpack-merge')  
const common = require('./webpack.common.js')  
  
module.exports = merge(common, {  
  mode: 'development',  
+  devServer: {  
+    hot: true  
+  },  
+  plugins: [  
+    new webpack.HotModuleReplacementPlugin()  
+  ],  
  ...  
})
```

Now we need to *accept* changed modules from our code to re-initialize things.

src/app.js

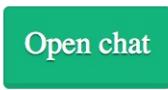
```
+ if (module.hot) {  
+   module.hot.accept()  
+ }  
  
...
```

Hot Module Replacement

When Hot Module Replacement is enabled, `module.hot` is set to `true` for development and `false` for production, so these are stripped out of the bundle.

Restart the build and see what happens when we do the following:

- Click *Open chat*
- Add a new person to the `people.js` module
- Click *Open chat* again.

 Open chat

There are 3 people in the room.

 Open chat

There are 4 people in the room.

Here's what's happening:

1. When *Open chat* is clicked, the `chat.js` module is fetched and initialized
2. HMR detects when `people.js` is modified
3. `module.hot.accept()` in `index.js` causes all modules loaded by this entry chunk to be replaced
4. When *Open chat* is clicked again, `chat.init()` is run with the code from the updated module.

CSS Replacement

Let's change the button color to red and see what happens:

src/app.scss

```
button {  
  ...  
-  background: #24b47e;  
+  background: red;  
  ...  
}
```

[Open chat](#)

There are 3 people in the room.

[Open chat](#)

There are 4 people in the room.

Now we get to see instant updates to our styles without losing any state. This is a much-improved developer experience! And it feels like the future.

HTTP/2

One of the primary benefits of using a module bundler like Webpack is that it can help you improve performance by giving you control over how the assets are *built* and then *fetched* on the client. It has been considered [best practice](#) for years to concatenate files to reduce the number of requests that need to be made on the client. This is still valid, but [HTTP/2 now allows multiple files to be delivered in a single request](#), so concatenation isn't a silver bullet anymore. Your app may actually benefit from having many small files individually cached. The client could then fetch a single changed module rather than having to fetch an entire bundle again with *mostly* the same contents.

The creator of Webpack, [Tobias Koppers](#), has written an informative post explaining why bundling is still important, even in the HTTP/2 era.

Read more about this over at [Webpack & HTTP/2](#).

Over to You

I hope you've found this introduction to Webpack helpful and are able to start using it to great effect. It can take a little time to wrap your head around Webpack's configuration, loaders and plugins, but learning how this tool works will pay off.

The documentation for Webpack 4 is currently being worked on, but is really well put together. I highly recommend reading through the [Concepts](#) and [Guides](#) for more information. Here's a few other topics you may be interested in:

- [Source Maps for development](#)
- [Source Maps for production](#)
- [Cache busting with hashed filenames](#)
- [Splitting a vendor bundle](#)

Chapter 3: An Introduction to Gulp.js

by Craig Buckler

Developers spend precious little time coding. Even if we ignore irritating meetings, much of the job involves basic tasks which can sap your working day:

- generating HTML from templates and content files
- compressing new and modified images
- compiling Sass to CSS code
- removing console and debugger statements from scripts
- transpiling ES6 to cross-browser-compatible ES5 code
- code linting and validation
- concatenating and minifying CSS and JavaScript files
- deploying files to development, staging and production servers.

Tasks must be repeated every time you make a change. You may start with good intentions, but the most infallible developer will forget to compress an image or two. Over time, pre-production tasks become increasingly arduous and time-consuming; you'll dread the inevitable content and template changes. It's mind-numbing, repetitive work. Wouldn't it be better to spend your time on more profitable jobs?

If so, you need a *task runner* or *build process*.

That Sounds Scarily Complicated!

Creating a build process will take time. It's more complex than performing each task manually, but over the long term, you'll save hours of effort, reduce human error and save your sanity. Adopt a pragmatic approach:

- Automate the most frustrating tasks first.
- Try not to over-complicate your build process. An hour or two is more than

enough for the initial setup.

- Choose task runner software and stick with it for a while. Don't switch to another option on a whim.

Some of the tools and concepts may be new to you, but take a deep breath and concentrate on one thing at a time.

Task Runners: the Options

Build tools such as [GNU Make](#) have been available for decades, but web-specific task runners are a relatively new phenomenon. The first to achieve critical mass was [Grunt](#) — a Node.js task runner which used plugins controlled (originally) by a JSON configuration file. Grunt was hugely successful, but there were a number of issues:

1. Grunt required plugins for basic functionality such as file watching.
2. Grunt plugins often performed multiple tasks, which made customisation more awkward.
3. JSON configuration could become unwieldy for all but the most basic tasks.
4. Tasks could run slowly because Grunt saved files between every processing step.

Many issues were addressed in later editions, but [Gulp](#) had already arrived and offered a number of improvements:

1. Features such as file watching were built in.
2. Gulp plugins were (*mostly*) designed to do a single job.
3. Gulp used JavaScript configuration code that was less verbose, easier to read, simpler to modify, and provided better flexibility.
4. Gulp was faster because it uses [Node.js streams](#) to pass data through a series of piped plugins. Files were only written at the end of the task.

Of course, Gulp itself isn't perfect, and new task runners such as [Broccoli.js](#), [Brunch](#) and [webpack](#) have also been competing for developer attention. More recently, [npm itself has been touted as a simpler option](#). All have their pros and cons, but [Gulp remains the favorite and is currently used by more than 40% of web developers](#).

Gulp requires Node.js, but while some JavaScript knowledge is beneficial,

developers from all web programming faiths will find it useful.

What About Gulp 4?

This tutorial describes how to use Gulp 3 — the most recent release version at the time of writing. Gulp 4 has been in development for some time but remains a beta product. It's possible to [use or switch to Gulp 4](#), but I recommend sticking with version 3 until the final release.

Step 1: Install Node.js

Node.js can be downloaded for Windows, macOS and Linux from [nodejs.org/download/](#). There are various options for installing from binaries, package managers and docker images, and full instructions are available.

Windows Users

Node.js and Gulp run on Windows, but some plugins may not install or run if they depend on native Linux binaries such as image compression libraries. One option for Windows 10 users is the new [bash command-line](#), which solves many issues.

Once installed, open a command prompt and enter:

```
node -v
```

This reveals the version number. You're about to make heavy use of npm — the Node.js package manager which is used to install modules. Examine its version number:

```
npm -v
```

For Linux Users

Node.js modules can be installed globally so they're available throughout your system. However, most users will not have permission to write to the global directories unless npm commands are prefixed with sudo. There are a number of [options to fix npm permissions](#) and tools such as [nvm can help](#), but I often

change the default directory. For example, on Ubuntu/Debian-based platforms:

```
cd ~  
mkdir .node_modules_global  
npm config set prefix=$HOME/.node_modules_global  
npm install npm -g
```

Then add the following line to the end of `~/.bashrc`:

```
export PATH="$HOME/.node_modules_global/bin:$PATH"
```

Finally, update with this:

```
source ~/.bashrc
```

Step 2: Install Gulp Globally

Install Gulp command-line interface globally so the `gulp` command can be run from any project folder:

```
npm install gulp-cli -g
```

Verify Gulp has installed with this:

```
gulp -v
```

Step 3: Configure Your Project

For Node.js Projects

You can skip this step if you already have a `package.json` configuration file.

Presume you have a new or pre-existing project in the folder `project1`. Navigate to this folder and initialize it with `npm`:

```
cd project1  
npm init
```

You'll be asked a series of questions. Enter a value or hit **Return** to accept

defaults. A package.json file will be created on completion which stores your npm configuration settings.

For Git Users

Node.js installs modules to a node_modules folder. You should add this to your .gitignore file to ensure they're not committed to your repository. When deploying the project to another PC, you can run npm install to restore them.

For the remainder of this article, we'll presume your project folder contains the following sub-folders:

src folder: preprocessed source files

This contains further sub-folders:

- html - HTML source files and templates
- images — the original uncompressed images
- js — multiple preprocessed script files
- scss — multiple preprocessed Sass .scss files

build folder: compiled/processed files

Gulp will create files and create sub-folders as necessary:

- html — compiled static HTML files
- images — compressed images
- js — a single concatenated and minified JavaScript file
- css — a single compiled and minified CSS file

Your project will almost certainly be different but this structure is used for the examples below.

Following Along on Unix

If you're on a Unix-based system and you just want to follow along with the tutorial, you can recreate the folder structure with the following command:

```
mkdir -p src/{html,images,js,scss} build/{html,images,js,css}
```

Step 4: Install Gulp Locally

You can now install Gulp in your project folder using the command:

```
npm install gulp --save-dev
```

This installs Gulp as a development dependency and the "devDependencies" section of package.json is updated accordingly. We'll presume Gulp and all plugins are development dependencies for the remainder of this tutorial.

Alternative Deployment Options

Development dependencies are not installed when the NODE_ENV environment variable is set to production on your operating system. You would normally do this on your live server with the Mac/Linux command:

```
export NODE_ENV=production
```

Or on Windows:

```
set NODE_ENV=production
```

This tutorial presumes your assets will be compiled to the build folder and committed to your Git repository or uploaded directly to the server. However, it may be preferable to build assets on the live server if you want to change the way they are created. For example, HTML, CSS and JavaScript files are minified on production but not development environments. In that case, use the --save option for Gulp and all plugins, i.e.

```
npm install gulp --save
```

This sets Gulp as an application dependency in the "dependencies" section of package.json. It will be installed when you enter npm install and can be run wherever the project is deployed. You can remove the build folder from your repository since the files can be created on any platform when required.

Step 4: Create a Gulp Configuration File

Create a new gulpfile.js configuration file in the root of your project folder.

Add some basic code to get started:

```
// Gulp.js configuration
var
  // modules
  gulp = require('gulp'),
  // development mode?
  devBuild = (process.env.NODE_ENV !== 'production'),
  // folders
  folder = {
    src: 'src/',
    build: 'build/'
  }
;
```

This references the Gulp module, sets a `devBuild` variable to `true` when running in development (or non-production mode) and defines the source and build folder locations.

ES6

ES6 note: ES5-compatible JavaScript code is provided in this tutorial. This will work for all versions of Gulp and Node.js with or without the `--harmony` flag. [Most ES6 features are supported in Node 6 and above](#) so feel free to use arrow functions, `let`, `const`, etc. if you're using a recent version.

`gulpfile.js` won't do anything yet because you need to ...

Step 5: Create Gulp Tasks

On its own, Gulp does nothing. You must:

1. install Gulp plugins, and
2. write tasks which utilize those plugins to do something useful.

It's possible to write your own plugins but, since almost 3,000 are available, it's unlikely you'll ever need to. You can search using Gulp's own directory at gulpjs.com/plugins/, on npmjs.com, or search "`gulp something`" to harness the mighty power of Google.

Gulp provides three primary task methods:

- `gulp.task` — defines a new task with a name, optional array of dependencies and a function.
- `gulp.src` — sets the folder where source files are located.
- `gulp.dest` — sets the destination folder where build files will be placed.

Any number of plugin calls are set with `pipe` between the `.src` and `.dest`.

Image Task

This is best demonstrated with an example, so let's create a basic task which compresses images and copies them to the appropriate build folder. Since this process could take time, we'll only compress new and modified files. Two plugins can help us: [gulp-newer](#) and [gulp-imagemin](#). Install them from the command-line:

```
npm install gulp-newer gulp-imagemin --save-dev
```

We can now reference both modules the top of `gulpfile.js`:

```
// Gulp.js configuration

var
  // modules
  gulp = require('gulp'),
  newer = require('gulp-newer'),
  imagemin = require('gulp-imagemin'),
```

We can now define the image processing task itself as a function at the end of `gulpfile.js`:

```
// image processing
gulp.task('images', function() {
  var out = folder.build + 'images/';
  return gulp.src(folder.src + 'images/**/*')
    .pipe(newer(out))
    .pipe(imagemin({ optimizationLevel: 5 }))
    .pipe(gulp.dest(out));
});
```

All tasks are syntactically similar. This code:

1. Creates a new task named `images`.
2. Defines a function with a return value which ...
3. Defines an `out` folder where build files will be located.
4. Sets the Gulp `src` source folder. The `/**/*` ensures that images in sub-folders are also processed.
5. Pipes all files to the `gulp-newer` module. Source files that are newer than corresponding destination files are passed through. Everything else is removed.
6. The remaining new or changed files are piped through `gulp-imagemin` which sets an optional `optimizationLevel` argument.
7. The compressed images are output to the Gulp `dest` folder set by `out`.

Save `gulpfile.js` and place a few images in your project's `src/images` folder before running the task from the command line:

```
gulp images
```

All images are compressed accordingly and you'll see output such as:

```
Using file gulpfile.js
Running 'imagemin'...
Finished 'imagemin' in 5.71 ms
gulp-imagemin: image1.png (saved 48.7 kB)
gulp-imagemin: image2.jpg (saved 36.2 kB)
gulp-imagemin: image3.svg (saved 12.8 kB)
```

Try running `gulp images` again and nothing should happen because no newer images exist.

HTML Task

We can now create a similar task which copies files from the source HTML folder. We can safely minify our HTML code to remove unnecessary whitespace and attributes using the [gulp-htmlclean](#) plugin:

```
npm install gulp-htmlclean --save-dev
```

This is then referenced at the top of `gulpfile.js`:

```
var
  // modules
  gulp = require('gulp'),
```

```
newer = require('gulp-newer'),
imagemin = require('gulp-imagemin'),
htmlclean = require('gulp-htmlclean'),
```

We can now create an `html` task at the end of `gulpfile.js`:

```
// HTML processing
gulp.task('html', ['images'], function() {
  var
    out = folder.build + 'html/',
    page = gulp.src(folder.src + 'html/**/*')
      .pipe(newer(out));

  // minify production code
  if (!devBuild) {
    page = page.pipe(htmlclean());
  }

  return page.pipe(gulp.dest(out));
});
```

This reuses `gulp-newer` and introduces a couple of concepts:

1. The `[images]` argument states that our `images` task must be run before processing the HTML (the HTML is likely to reference images). Any number of dependent tasks can be listed in this array and all will complete before the task function runs.
2. We only pipe the HTML through `gulp-htmlclean` if `NODE_ENV` is set to `production`. Therefore, the HTML remains uncompressed during development which may be useful for debugging.

Save `gulpfile.js` and run `gulp html` from the command line. Both the `html` and `images` tasks will run.

JavaScript Task

Too easy for you? Let's process all our JavaScript files by building a basic module bundler. It will:

1. Ensure dependencies are loaded first using the [gulp-deorder](#) plugin. This analyses comments at the top of each script to ensure correct ordering. For example, `// requires: defaults.js lib.js`.
2. Concatenate all script files into a single `main.js` file using [gulp-concat](#).

3. Remove all console and debugging statements with [gulp-strip-debug](#) and minimize code with [gulp-uglify](#). This step will only occur when running in production mode.

Install the plugins:

```
npm install gulp-deporder gulp-concat gulp-strip-debug gulp-uglify -
```

Reference them at the top of `gulpfile.js`:

```
var  
  ...  
  concat = require('gulp-concat'),  
  deporder = require('gulp-deporder'),  
  stripdebug = require('gulp-strip-debug'),  
  uglify = require('gulp-uglify'),
```

Then add a new js task:

```
// JavaScript processing  
gulp.task('js', function() {  
  
  var jsbuild = gulp.src(folder.src + 'js/**/*')  
    .pipe(deporder())  
    .pipe(concat('main.js'));  
  
  if (!devBuild) {  
    jsbuild = jsbuild  
      .pipe(stripdebug())  
      .pipe(uglify());  
  }  
  
  return jsbuild.pipe(gulp.dest(folder.build + 'js/'));  
});
```

Save then run `gulp js` to watch the magic happen!

CSS Task

Finally, let's create a CSS task which compiles Sass `.scss` files to a single `.css` file using [gulp-sass](#). This is a Gulp plugin for [node-sass](#) which binds to the superfast [LibSass C/C++ port of the Sass engine](#) (*you won't need to install Ruby*). We'll presume your primary Sass file `scss/main.scss` is responsible for

loading all partials.

Our task will also utilize the fabulous [PostCSS](#) via the [gulp-postcss](#) plugin. PostCSS requires its own set of plugins and we'll install these:

- [postcss-assets](#) to manage assets. This allows us to use properties such as `background: resolve('image.png');` to resolve file paths or `background: inline('image.png');` to inline data-encoded images.
- [autoprefixer](#) to automatically add vendor prefixes to CSS properties.
- [css-mqpacker](#) to pack multiple references to the same CSS media query into a single rule.
- [cssnano](#) to minify the CSS code when running in production mode.

First, install all the modules:

```
npm install gulp-sass gulp-postcss postcss-assets autoprefixer css-n
```

Then reference them at the top of `gulpfile.js`:

```
var
  ...
sass = require('gulp-sass'),
postcss = require('gulp-postcss'),
assets = require('postcss-assets'),
autoprefixer = require('autoprefixer'),
mqpacker = require('css-mqpacker'),
cssnano = require('cssnano'),
```

We can now create a new css task at the end of `gulpfile.js`. Note the `images` task is set as a dependency because the `postcss-assets` plugin can reference images during the build process. In addition, most plugins can be passed arguments (refer to their documentation for more information):

```
// CSS processing
gulp.task('css', ['images'], function() {

  var postCssOpts = [
    assets({ loadPaths: ['images/] }),
    autoprefixer({ browsers: ['last 2 versions', '> 2%'] }),
    mqpacker
  ];

  if (!devBuild) {
    postCssOpts.push(cssnano);
  }
})
```

```
}

return gulp.src(folder.src + 'scss/main.scss')
  .pipe(sass({
    outputStyle: 'nested',
    imagePath: 'images/',
    precision: 3,
    errLogToConsole: true
  }))
  .pipe(postcss(postCssOpts))
  .pipe(gulp.dest(folder.build + 'css/')));

});
```

Save the file and run the task from the command line:

```
gulp css
```

Step 6: Automate Tasks

We've been running one task at a time. We can run them all in one command by adding a new `run` task to `gulpfile.js`:

```
// run all tasks
gulp.task('run', ['html', 'css', 'js']);
```

Save and enter `gulp run` at the command line to execute all tasks. Note that I omitted the `images` task because it's already set as a dependency for the `html` and `css` tasks.

Is this still too much hard work? Gulp offers another method — `gulp.watch` — which can monitor your source files and run an appropriate task whenever a file is changed. The method is passed a folder and a list of tasks to execute when a change occurs. Let's create a new `watch` task at the end of `gulpfile.js`:

```
// watch for changes
gulp.task('watch', function() {

  // image changes
  gulp.watch(folder.src + 'images/**/*', ['images']);

  // html changes
  gulp.watch(folder.src + 'html/**/*', ['html']);

  // javascript changes
  gulp.watch(folder.src + 'js/**/*', ['js']);
});
```

```
gulp.watch(folder.src + 'js/**/*', ['js']);

// css changes
gulp.watch(folder.src + 'scss/**/*', ['css']);

});
```

Rather than running `gulp watch` immediately, let's add a default task:

```
// default task
gulp.task('default', ['run', 'watch']);
```

Save `gulpfile.js` and enter `gulp` at the command line. Your images, HTML, CSS and JavaScript will all be processed, then Gulp will remain active watching for updates and re-running tasks as necessary. Hit `Ctrl/Cmd + C` to abort monitoring and return to the command line.

Step 7: Profit!

Other plugins you may find useful:

- [gulp-load-plugins](#) ± load all Gulp plugin modules without `require` declarations
- [gulp-preprocess](#) — a simple HTML and JavaScript [preprocessor](#)
- [gulp-less](#) — the [Less CSS preprocessor](#) plugin
- [gulp-stylus](#) — the [Stylus CSS preprocessor](#) plugin
- [gulp-sequence](#) — run a series of gulp tasks in a specific order
- [gulp-plumber](#) — error handling which prevents Gulp stopping on failures
- [gulp-size](#) — displays file sizes and savings
- [gulp-nodemon](#) — uses [nodemon](#) to automatically restart Node.js applications when changes occur.
- [gulp-util](#) — utility functions including logging and color coding.

One useful method in `gulp-util` is `.noop()` which passes data straight through without performing any action. This could be used for cleaner development/production processing code. For example:

```
var gutil = require('gulp-util');

// HTML processing
gulp.task('html', ['images'], function() {
  var out = folder.src + 'html/**/*';
```

```
    return gulp.src(folder.src + 'html/**/*')
      .pipe(newer(out))
      .pipe(devBuild ? gutil.noop() : htmlclean())
      .pipe(gulp.dest(out));
  });
}
```

Gulp can also call other Node.js modules, and they don't necessarily need to be plugins. For example:

- [browser-sync](#) — automatically reload assets or refresh your browser when changes occur
- [del](#) — delete files and folders (perhaps clean your build folder at the start of every run).

Invest a little time and Gulp could save many hours of development frustration. The advantages:

- [plugins are plentiful](#)
- configuration using pipes is readable and easy to follow
- `gulpfile.js` can be adapted and reused in other projects
- your total page weight can be reduced to improve performance
- you can simplify your deployment.

Useful links:

- [Gulp home page](#)
- [Gulp plugins](#)
- [npm home page](#)

Applying the processes above to a simple website reduced the total weight by more than 50%. You can test your own results using [page weight analysis tools](#) or a service such as [New Relic](#), which provides a range of sophisticated application performance monitoring tools.

Gulp can revolutionize your workflow. I hope you found this tutorial useful and consider Gulp for your production process.

Chapter 4: 10 Languages That Compile to JavaScript

by James Kolce

This chapter includes a list of ten interesting languages that can compile to JavaScript to be executed in the browser or on a platform like Node.js.

Modern applications have [different requirements](#) from simple websites. But the browser is a platform with a (mostly) fixed set of technologies available, and JavaScript remains as the core language for web applications. Any application that needs to run in the browser has to be implemented in that language.

We all know that JavaScript isn't the best language for every task, and when it comes to complex applications, it might fall short. To avoid this problem, several new languages and transpilers of existing ones have been created, all of them producing code that can work in the browser without any lines of JavaScript having to be written, and without you having to think about the limitations of the language.

Dart

Dart is a classical, object-oriented language where everything is an object and any object is an instance of a class (objects can act as functions too.) It's specially made to build applications for browsers, servers, and mobile devices. It's maintained by Google and is [the language that powers the next generation AdWords UI](#), the most important product of Google regarding revenue, which is in itself a proof of its power at scale.

The language can be translated to JavaScript to be used in a browser, or be directly interpreted by the Dart VM, which allows you to build server applications too. Mobile applications can be made using the Flutter SDK.

Complex applications also require a mature set of libraries and language features specially designed for the task, and Dart includes all of this. An example of a

popular library is [AngularDart](#), a version of Angular for Dart.

It allows you to write type-safe code without being too intrusive. You can write types, but you aren't required to do so,* since they can be inferred. This allows for rapid prototyping without having to overthink the details, but once you have something working, you can add types to make it more robust.

Regarding concurrent programming in the VM, instead of shared-memory threads (Dart is single-threaded), Dart uses what they call **Isolates**, with their own memory heap, where communication is achieved using messages. In the browser, the story is a little different: instead of creating new isolates, you create new **Workers**.

```
// Example extracted from dartlang.org

import 'dart:async';
import 'dart:math' show Random;

main() async {
    print('Compute π using the Monte Carlo method.');
    await for (var estimate in computePi()) {
        print('π ≈ $estimate');
    }
}

/// Generates a stream of increasingly accurate estimates of π.
Stream<double> computePi({int batch: 1000000}) async* {
    var total = 0;
    var count = 0;
    while (true) {
        var points = generateRandom().take(batch);
        var inside = points.where((p) => p.isInsideUnitCircle);
        total += batch;
        count += inside.length;
        var ratio = count / total;
        // Area of a circle is A = π·r2, therefore π = A/r2.
        // So, when given random points with x ∈ <0,1>,
        // y ∈ <0,1>, the ratio of those inside a unit circle
        // should approach π / 4. Therefore, the value of π
        // should be:
        yield ratio * 4;
    }
}

Iterable<Point> generateRandom([int seed]) sync* {
    final random = new Random(seed);
```

```
while (true) {
    yield new Point(random.nextDouble(), random.nextDouble());
}
}

class Point {
    final double x, y;
    const Point(this.x, this.y);
    bool get isInsideUnitCircle => x * x + y * y <= 1;
}
```

For more reading, I recommend Dart's [Get started with Dart](#) resource.

TypeScript

[TypeScript](#) is a superset of JavaScript. A valid JavaScript program is also valid TypeScript, but with static typing added. The compiler can also work as a transpiler from ES2015+ to current implementations, so you always get the latest features.

Unlike many other languages, TypeScript keeps the spirit of JavaScript intact, only adding features to improve the soundness of the code. These are type annotations and other type-related functionality that makes writing JavaScript more pleasant, thanks to the enabling of specialized tools like static analyzers and other tools to aid in the refactoring process. Also, the addition of types improve the interfaces between the different components of your applications.

Type inference is supported, so you don't have to write all the types from the beginning. You can write quick solutions, and then add all the types to get confident about your code.

TypeScript also has support for advanced types, like intersection types, union types, type aliases, discriminated unions and type guards. You can check out all these in the [Advanced Types](#) page in the [TypeScript Documentation](#) site.

JSX is also supported by adding the React typings if you use React:

```
class Person {
    private name: string;
    private age: number;
    private salary: number;
```

```
constructor(name: string, age: number, salary: number) {  
    this.name = name;  
    this.age = age;  
    this.salary = salary;  
}  
  
toString(): string {  
    return `${this.name} (${this.age}) (${this.salary})`;  
}  
}
```

For more on typeScript, check out SitePoint's [getting started with TypeScript](#) article.

Elm

Elm is a purely functional programming language that can compile to JavaScript, HTML, and CSS. You can build a complete site with just Elm, making it a great alternative to JavaScript frameworks like React. The applications that you build with it automatically use a virtual DOM library, making it very fast. One big plus is the built-in architecture that makes you forget about data-flow and focus on data declaration and logic instead.

In Elm, all functions are pure, which means they'll always return the same output for a given input. They can't do anything else unless you specify it. For example, to access a remote API you'd create *command* functions to communicate with the external world, and *subscriptions* to listen for responses. Another point for purity is that values are immutable: when you need something, you create new values, instead of modifying them.

The adoption of Elm can be gradual. It's possible to communicate with JavaScript and other libraries using *ports*. Although Elm hasn't reached version 1 yet, it's being used for complex and large applications, making it a feasible solution for complex applications.

One of the most attractive features of Elm is the beginner-friendly compiler, which, instead of producing hard-to-read messages, generates code that helps you to fix your code. If you're learning the language, the compiler itself can be of big help.

```
module Main exposing (..)
```

```
import Html exposing (..)

-- MAIN

main : Program Never Model Msg
main =
    Html.program
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        }

-- INIT

type alias Model = String

init : ( Model, Cmd Msg )
init = ( "Hello World!", Cmd.none )

-- UPDATE

type Msg
    = DoNothing

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        DoNothing ->
            ( model, Cmd.none )

-- VIEW

view : Model -> Html Msg
view model =
    div [] [text model]
```

```
-- SUBSCRIPTIONS
```

```
subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none
```

SitePoint has a handy [getting started with Elm](#) article if you want to find out more.

PureScript

[PureScript](#) is a purely functional and strongly typed programming language, created by Phil Freeman. It aims to provide strong compatibility with available JavaScript libraries, similar to Haskell in spirit, but keeping JavaScript at its core.

A strong point for PureScript is its minimalism. It doesn't include any libraries for functionality that would be considered essential in other languages. For example, instead of including generators and promises in the compiler itself, you can use specific libraries for the task. You can choose the implementation you want for the feature you need, which allows a highly efficient and personalized experience when using PureScript, while keeping the generated code as small as possible.

Another distinctive feature of its compiler is the ability to make clean and readable code while maintaining compatibility with JavaScript, both concerning libraries and tools.

Like other languages, PureScript has its own [build tool called Pulp](#), which can be compared to Gulp, but for projects written in this language.

Regarding the type system — unlike Elm, which is the other ML-like language — PureScript has support for advanced type features like [higher-kinded types](#) and type classes, which are taken from Haskell, allowing the creation of sophisticated abstractions:

```
module Main where
```

```

import Prelude
import Data.Foldable (fold)
import TryPurescript

main =
  render $ fold
    [ h1 (text "Try PureScript!")
    , p (text "Try out the examples below, or create your own!")
    , h2 (text "Examples")
    , list (map fromExample examples)
    ]
  where
    fromExample { title, gist } =
      link ("?gist=" <> gist) (text title)

    examples =
      [ { title: "Algebraic Data Types"
        , gist: "37c3c97f47a43f20c548"
        }
      , { title: "Loops"
        , gist: "cfdbabdc085d4ac3dc46"
        }
      , { title: "Operators"
        , gist: "3044550f29a7c5d3d0d0"
        }
      ]

```

To take your next step with PureScript, check out the [getting started with PureScript](#) guide on GitHub.

CoffeeScript

CoffeeScript is a language that aims to expose the good parts of JavaScript while providing a cleaner syntax and keeping the semantics in place. Although the popularity of the language has been waning in recent years, it's changing direction and recently received a new major version providing support for ES2015+ features.

The code you write in CoffeeScript is directly translated to readable JavaScript code and maintains compatibility with existing libraries. From version 2, the compiler produces code compatible with the latest versions of ECMAScript. For example, every time you use a class, you get a class in JavaScript. Also, if you use React, there's good news: JSX is compatible with CoffeeScript.

A very distinctive feature of the compiler is the ability to process code written in the [literate style](#), where instead of making emphasis in the code and having comments as an extra, you write comments in the first place, and the code only occasionally appears. This style of programming was introduced by Donald Knuth, making a code file very similar to a technical article.

Unlike the other languages, CoffeeScript code can be interpreted directly in the browser using a library. So if you want to create a quick test, you can write your code in `<text/coffeescript>` script tags, and include the compiler, which will translate the code to JavaScript on the fly:

```
# Assignment:  
number = 42  
opposite = true  
  
# Conditions:  
number = -42 if opposite  
  
# Functions:  
square = (x) -> x * x  
  
# Arrays:  
list = [1, 2, 3, 4, 5]  
  
# Objects:  
math =  
  root: Math.sqrt  
  square: square  
  cube: (x) -> x * square x  
  
# Splats:  
race = (winner, runners...) ->  
  print winner, runners  
  
# Existence:  
alert "I knew it!" if elvis?  
  
# Array comprehensions:  
cubes = (math.cube num for num in list)
```

The CoffeeScript site has a handy [getting started with CoffeeScript 2](#) resource.

ClojureScript

ClojureScript is a compiler that translates the Clojure programming language to

JavaScript. It's a general-purpose, functional language with dynamic typing and support for immutable data structures.

It's the only one from this list that belongs to the Lisp family of programming languages and, naturally, it shares a lot of the features. For example, the code can be treated as data, and a macro system is available, making metaprogramming techniques possible. Unlike other Lisps, Clojure has support for immutable data structures, making the management of side effects easier.

The syntax can look intimidating for newcomers because of its use of parentheses, but it has profound reasons to be that way, and you'll certainly appreciate it in the long run. That minimalism in the syntax and its syntactic abstraction capabilities make Lisp a powerful tool for solving problems that require high levels of abstraction.

Although Clojure is mainly a functional language, it isn't pure like PureScript or Elm. Side effects can still happen, but other functional features are still present.

ClojureScript uses Google Closure for code optimization and also has compatibility with existing JavaScript libraries:

```
; Extracted from https://github.com/clojure/clojurescript/blob/master/test/dom/test.cljs

(ns dom.test
  (:require [clojure.browser.event :as event]
            [clojure.browser.dom    :as dom]))

(defn log [& args]
  (.log js/console (apply pr-str args)))

(defn log-obj [obj]
  (.log js/console obj))

(defn log-listener-count []
  (log "listener count: " (event/total-listener-count)))

(def source      (dom/get-element "source"))
(def destination (dom/get-element "destination"))

(dom/append source
             (dom/element "Testing me "))
             (dom/element "out!"))

(def success-count (atom 0))
```

```
(log-listener-count)

(event/listen source
  :click
  (fn [e]
    (let [i (swap! success-count inc)
          e (dom/element :li
                        {:id "testing"
                         :class "test me out please"}
                        "It worked!")]
      (log-obj e)
      (log i)
      (dom/append destination
                    e)))))

(log-obj (dom/element "Text node"))
(log-obj (dom/element :li))
(log-obj (dom/element :li {:class "foo"}))
(log-obj (dom/element :li {:class "bar"} "text node"))
(log-obj (dom/element [:ul [:li :li :li]]))
(log-obj (dom/element :ul [:li :li :li]))
(log-obj (dom/element :li {}) [:ul {} [:li :li :li]]))
(log-obj (dom/element [:li {:class "baz"} [:li {:class "quux"}]]))

(log-obj source)
(log-listener-count)
```

To learn more, head over to the ClojureScript site's [getting started with ClojureScript](#) resource.

Scala.js

Scala.js is a compiler that translates the Scala programming language to JavaScript. Scala is a language that aims to merge the ideas from object-oriented and functional programming into one language to create a powerful tool that is also easy to adopt.

As a strongly typed language, you get the benefits of a flexible type system with partial type inference. Most values can be inferred, but function parameters still require explicit type annotations.

Although many common object-oriented patterns are supported (for example, every value is an object and operations are method calls), you also get functional features like support for first-class functions and immutable data structures.

One of the special advantages of Scala.js is that you can start with a familiar, object-oriented approach and move to a more functional one as you need and at your own speed, without having to do a lot of work. Also, existing JavaScript code and libraries are compatible with your Scala code.

Beginner Scala developers will find the language not very different from JavaScript. Compare the following equivalent code:

```
// JavaScript
var xhr = new XMLHttpRequest();

xhr.open("GET",
  "https://api.twitter.com/1.1/search/" +
  "tweets.json?q=%23scalajs"
);
xhr.onload = (e) => {
  if (xhr.status === 200) {
    var r = JSON.parse(xhr.responseText);
    $("#tweets").html(parseTweets(r));
  }
};
xhr.send();
```

```
// Scala.js
val xhr = new XMLHttpRequest()

xhr.open("GET",
  "https://api.twitter.com/1.1/search/" +
  "tweets.json?q=%23scalajs"
)
xhr.onload = { (e: Event) =>
  if (xhr.status == 200) {
    val r = JSON.parse(xhr.responseText)
    $("#tweets").html(parseTweets(r))
  }
}
xhr.send()
```

Check out the Scala.js [getting started with Scala.js](#) docs for more.

Reason

Reason is a language created and maintained by Facebook, which offers a new syntax for the OCaml compiler, and the code can be translated to both JavaScript and native code.

Being part of the ML family and a functional language itself, it naturally offers a powerful but flexible type system with inference, algebraic data types and pattern matching. It also has support for immutable data types and parametric polymorphism (also known as *generics* in other languages) but, as in OCaml, support for object-oriented programming is available as well.

The use of existing JavaScript libraries is possible with [bucklescript](#) bindings. You can also mix in JavaScript alongside your Reason code. The inserted JavaScript code won't be strictly checked, but it works fine for quick fixes or prototypes.

If you're a React developer, [bindings are available](#), and the language also has support for JSX:

```
/* A type variant being pattern matched */

let possiblyNullValue1 = None;
let possiblyNullValue2 = Some "Hello@";

switch possiblyNullValue2 {
| None => print_endline "Nothing to see here."
| Some message => print_endline message
};

/* Parametrized types */

type universityStudent = {gpa: float};
type response 'studentType = {status: int, student: 'studentType};
let result: response universityStudent = fetchDataFromServer ();

/* A simple typed object */

type payload = Js.t {
  name: string,
  age: int
};
let obj1: payload = {"name": "John", "age": 30};
```

Check out the Reason site's [getting started with Reason](#) guide for more.

Haxe

Haxe is a multi-paradigm programming language, and its compiler can produce both binaries and source code in other languages.

Although Haxe provides a strict type system with support for type inference, it can also work as a dynamic language if the target language supports it. In the same way, it provides support for a variety of programming styles like object-oriented, generic, and functional.

When you write Haxe code, you can target several platforms and languages for compilation without having to make considerable changes. Target-specific code blocks are also available.

You can write both back ends and front ends in Haxe with the same code and achieve communication using [Haxe Remoting](#), for both synchronous and asynchronous connections.

As expected, Haxe code is compatible with existing libraries, but it also provides a mature standard library:

```
// Example extracted from http://code.haxe.org

extern class Database {
    function new();
    function getProperty<T>(property:Property<T>):T;
    function setProperty<T>(property:Property<T>, value:T):Void;
}

abstract Property<T>(String) {
    public inline function new(name) {
        this = name;
    }
}

class Main {
    static inline var PLAYER_NAME = new Property<String>("playerName");
    static inline var PLAYER_LEVEL = new Property<Int>("playerLevel");

    static function main() {
        var db = new Database();

        var playerName = db.getProperty(PLAYER_NAME);
        trace(playerName.toUpperCase());

        db.setProperty(PLAYER_LEVEL, 1);
    }
}
```

Check out the Haxe site's [getting started with Haxe](#) pages for more.

Nim

Nim is a statically typed, multi-paradigm programming language with minimalist and whitespace-sensitive syntax that can compile to JavaScript as well as C, C++.

The language itself is very small, but its metaprogramming capabilities make it attractive to implement features by yourself that you might find built-in to other languages. The building blocks for this are macros, templates, and generics, and with them you can implement things from simple features to different paradigms. This makes Nim an extremely versatile language that can be adapted to your needs, in the spirit of Lisp.

The syntactic abstraction features of Nim allow you to adapt the language to your problems, making true [DSLs](#) possible. If you have specialized tasks to solve, you can get a higher level of expressiveness:

```
# Reverse a string
proc reverse(s: string): string =
    result = ""
    for i in countdown(high(s), 0):
        result.add s[i]

var str1 = "Reverse This!"
echo "Reversed: ", reverse(str1)

# Using templates
template genType(name, fieldname: expr, fieldtype: typedesc) =
    type
        name = object
            fieldname: fieldtype

genType(Test, foo, int)

var x = Test(foo: 4566)
echo(x.foo) # 4566
```

The Nim site has some useful [getting started](#) docs for more information.

Conclusion

If JavaScript isn't your favorite language, you can still create web applications

without having to suffer the shortcomings of the technology. The options available to create those applications can fill a wide spectrum of taste, from purely functional languages like PureScript to object-oriented ones like Dart. And if you want something more than a one-to-one language translation, you have options like Elm that provide you with tools like a virtual DOM and a built-in architecture.

Chapter 5: 10 Must-have VS Code Extensions for JavaScript Developers

by Michael Wanyoike

In this article, I'll focus on a list of must-have VS Code extensions for JavaScript developers.

[Visual Studio Code](#) is undoubtedly the most popular lightweight code editor today. It does borrow heavily from other popular code editors, mostly Sublime Text and Atom. However, its success mainly comes from its ability to provide better performance and stability. In addition, it also provides much needed features like IntelliSense, which were only available in full-sized IDEs like Eclipse or Visual Studio 2017.

The power of VS Code no doubt comes from the [marketplace](#). Thanks to the wonderful open-source community, the editor is now capable of supporting almost every programming language, framework and development technology. Support for a library or framework comes in various ways, which mainly includes snippets, syntax highlighting, [Emmet](#) and IntelliSense features for that specific technology.

VS Code Extensions by Category

For this article, I'll focus on VS Code extensions specifically targeting JavaScript developers. Currently, there are many VS Code extensions that fit this criterion, which of course means I won't be able to mention all of them. Instead, I'll highlight VS Code extensions that have gained popularity and those that are indispensable for JavaScript developers. For simplicity, I'll group them into ten specific categories.

Snippet Extensions

When you first install VS Code, it comes with a several snippets for JavaScript and Typescript. Before you start writing modern JavaScript, you'll need some additional snippets to help you quickly write repetitive ES6/ES7 code:

- [VS Code JavaScript \(ES6\) snippets](#): currently the most popular, with over 1.2 million installs to date. This extension provides ES6 syntax for JavaScript, TypeScript, HTML, React and Vue extensions.
- [JavaScript Snippet Pack](#): a collection of useful snippets for JavaScript.
- [Atom JavaScript Snippet](#): JavaScript snippets ported from the atom/language-javascript extension.
- [JavaScript Snippets](#): a collection of ES6 snippets. This extension has snippets for Mocha, Jasmine and other BDD testing frameworks.

Syntax Extensions

VS Code comes with pretty good syntax highlighting for JavaScript code. You can change the colors by installing themes. However, you need a syntax highlighter extension if you want a certain level of readability. Here are a couple of them:

- [JavaScript Atom Grammar](#): this extension replaces the JavaScript grammar in Visual Studio Code with the JavaScript grammar from the Atom editor.
- [Babel JavaScript](#): syntax highlighting for ES201x JavaScript, React, FlowType and GraphQL code.
- [DotENV](#): syntax highlighting for .env files. Handy if you're working with Node.

Linter Extensions

Writing JavaScript code efficiently with minimum fuss requires a linter that enforces a specific standard style for all team members. ESLint is the most popular, since it supports many styles including Standard, Google and Airbnb. Here are the most popular linter plugins for Visual Studio Code:

- [ESLint](#): this extension integrates [ESLint](#) into VS Code. It's the most popular linter extension, with over 6.7 million installs to date. Rules are configured in `.eslintrc.json`.
- [JSHint](#): a code checker extension for [JSHint](#). Uses `.jshintrcfile` at the root of your project for configuration.
- [JavaScript Standard Style](#): a linter with zero configuration and rigid rules. Enforces the [StandardJS Rules](#).
- [JSLint](#): linter extension for [JSLint](#).

If you'd like an overview of available linters and their pros and cons, check out our [comparison of JavaScript linting tools](#).

Node Extensions

Every JavaScript project needs to at least one Node package, unless you're someone who likes doing things the hard way. Here are a few VS Code extensions that will help you work with Node modules more easily.

- [npm](#): uses `package.json` to validate installed packages. Ensures that the installed packages have the correct version numbers, highlights installed packages missing from `package.json`, and packages that haven't been installed.
- [Node.js Modules IntelliSense](#): autocomplete for JavaScript and TypeScript modules in import statements.
- [Path IntelliSense](#): it's not really Node related, but you definitely need IntelliSense for local files and this extension will autocomplete filenames.
- [Node exec](#): allows you to execute the current file or your selected code with Node.js.
- [View Node Package](#): quickly view a Node package source with this extension, which allows you to open a Node package repository/documentation straight from VS Code.

- [Node Readme](#): quickly open npm package documentation.
- [Search node_modules](#): this extension allows you to search the node_modules folder, which is usually excluded from standard search.
- [Import Cost](#): displays size of an imported package.

Formatting Extensions

Once in a while, you find yourself formatting code that wasn't written in a preferred style. To save time, you can install any of these VS Code extensions to quickly format and refactor existing code:

- [Beautify](#): a [jsBeautifier](#) extension that supports JavaScript, JSON, CSS and HTML. Can be customized via .jsbeautifyrc file. Most popular formatter with 2.3 million installs to date.
- [Prettier Code Formatter](#): an extension that supports the formatting of JavaScript, TypeScript and CSS using [Prettier](#) (an opinionated code formatter). Has over 1.5 million installs to date.
- [JS Refactor](#): provides a number of utilities and actions for refactoring JavaScript code, such as extracting variables/methods, converting existing code to use arrow functions or template literals, and exporting functions.
- [JavaScript Booster](#): an amazing code refactoring tool. Features several coding actions such as converting var to const or let, removing redundant else statements, and merging declaration and initialization. Largely inspired by [WebStorm](#).

Browser Extensions

Unless you're writing a console program in JavaScript, you'll most likely be executing your JavaScript code inside a browser. This means you'll need to frequently refresh the page to see the effect of each code update you make. Instead of doing this manually all the time, here are a few tools that can significantly reduce the development time of your iteration process:

- [Debugger for Chrome](#): debug your JavaScript easily in Chrome (by setting

breakpoints inside the editor.

- [Live Server](#): local development server with live reload feature for static and dynamic pages.
- [Preview on Web Server](#): provides web server and live preview features.
- [PHP Server](#): useful for testing JavaScript code that needs to run client-side only.
- [Rest Client](#): instead of using a browser or a CURL program to test your REST API endpoints, you can install this tool to interactively run HTTP requests right inside the editor.

Framework Extensions

For most projects, you'll need a suitable framework to structure your code and cut down your development time. VS Code has support for most major frameworks through extensions. However, there are still a number of established frameworks that don't have a fully developed extension yet. Here are some of the VS Code extensions that offer significant functionality.

- [Angular 6](#): snippets for Angular 6. Supports Typescript, HTML, Angular Material ngRx, RxJS & Flex Layout. Has 2.2+ million installs and 172 Angular Snippets to date.
- [Angular v5 snippets](#): provides Angular snippets for TypeScript, RxJS, HTML and Docker files. Has 2.7+ million installs to date.
- [React Native/React/Redux snippets for es6/es7](#): provides snippets in ES6/ES7 syntax for all of these frameworks.
- [React Native Tools](#): provides IntelliSense, commands and debugging features for the React Native framework.
- [Vetur](#): provides syntax highlighting, snippets, Emmet, linting, formatting, IntelliSense and debugging support for the Vue framework. It comes with proper documentation published on [GitBook](#)

- [Ember](#): provides command support and IntelliSense for Ember. After installation, all `ember cli` commands are available through Code's own command list.
- [Cordova Tools](#): support for Cordova plugins and the Ionic framework. Provides IntelliSense, debugging and other support features for Cordova-based projects.
- [jQuery Code Snippets](#): provides over 130 jQuery code snippets. Activated by the prefix `jq`.

Testing Extensions

Testing is a critical part of software development, especially for projects that are in production phase. You can get a broad overview of testing in JavaScript and read more about the different kind of tests you can run in our guide — [JavaScript Testing: Unit vs Functional vs Integration Tests](#). Here are some popular VS Code extensions for testing:

- [Mocha sidebar](#): provides support for testing using the Mocha library. This extension helps you run tests directly on the code and shows errors as decorators.
- [ES6 Mocha Snippets](#): Mocha snippets in ES6 syntax. The focus of this extension is to keep the code dry, leveraging arrow functions and omitting curlies by where possible. Can be configured to allow semicolons.
- [Jasmine Code Snippets](#): code snippets for Jasmine test framework.
- [Protractor Snippets](#): end-to-end testing snippets for the Protractor framework. Supports both JavaScript and Typescript.
- [Node TDD](#): provides support for test-driven development for Node and JavaScript projects. Can trigger an automatic test build whenever sources are updated.

Awesome Extensions

I'm just putting this next bunch of VS Code extensions into the "awesome"

category, because that best describes them!

- [Quokka.js](#): an awesome debugging tool that provides a rapid prototyping playground for JavaScript code. Comes with [excellent documentation](#).
- [Paste as JSON](#): quickly convert JSON data into JavaScript code.
- [Code Metrics](#): this is another awesome extension that calculates complexity in JavaScript and TypeScript code.

Extension Packs

Now that we've come to our final category, I would just like to let you know that the VS Code marketplace has [a category for extension packs](#). Essentially, these are collections of related VS Code extensions bundled into one package for easy installation. Here are some of the better ones:

- [Nodejs Extension Pack](#): this pack contains ESLint, npm, JavaScript (ES6) snippets, Search node_modules, NPM IntelliSense and Path IntelliSense.
- [VS Code for Node.js - Development Pack](#): this one has NPM IntelliSense, ESLint, Debugger for Chrome, Code Metrics, Docker and Import Cost.
- [Vue.js Extension Pack](#): a collection of Vue and JavaScript extensions. It currently contains about 12 VS Code extensions, some of which haven't been mentioned here such as [auto-rename-tag](#) and [auto-close-tag](#).
- [Ionic Extension Pack](#): this pack contains a number of VS Code extensions for Ionic, Angular, RxJS, Cordova and HTML development.

Summary

VS Code's huge number of quality extensions makes it a popular choice for JavaScript developers. It's never been easier to write JavaScript code this efficiently. Extensions such as ESLint help you avoid common mistakes, while others such as Debugger for Chrome help you debug your code more easily. Node.js extensions with their IntelliSense features help you import modules correctly, and the availability of tools such as Live Server and REST client reduces your reliance on external tools to complete your work. All of these tools

make your iteration process far much easier.

I hope this list has been introduced you to new VS Code extensions that can help you in your workflow.

Chapter 6: Debugging JavaScript Projects with VS Code & Chrome Debugger

by Michael Wanyoike

Debugging JavaScript isn't the most fun aspect of JavaScript programming, but it's a vital skill. This article covers two tools that will help you debug JavaScript like a pro.

Imagine for a moment that the `console.log()` function did not exist in JavaScript. I'm pretty sure the first question you'd ask yourself would be "How am I ever going to confirm my code is working correctly?"

The answer lies in using debugging tools. For a long time, most developers, including myself, have been using `console.log` to debug broken code. It's quick and easy to use. However, things can get finicky at times if you don't know where and what is causing the bug. Often you'll find yourself laying down `console.log` traps all over your code to see which one will reveal the culprit.

To remedy this, we need to change our habits and start using debugging tools. There are a number of tools available for debugging JavaScript code, such as the Chrome Dev Tools, [Node Debugger](#), Node Inspect and others. In fact, every [major browser](#) provides its own tools.

In this this article, we'll look at how to use the debugging facilities provided by Visual Studio Code. We'll also look at how to use the [Debugger for Chrome](#) extension that allows VS Code to integrate with Chrome Dev Tools. Once we're finished, you'll never want to use a `console.log()` again.

Prerequisites

For this tutorial, you only need to have a solid foundation in [modern JavaScript](#). We'll also look at how we can debug a test written using [Mocha and Chai](#). We'll

be using a broken project, [debug-example](#), to learn how to fix various bugs without using a single `console.log`. You'll need the following to follow along:

- [Node.js](#)
- [Visual Studio Code](#)
- [Chrome Browser](#)

Start by cloning the [debug-example](#) project to your workspace. Open the project in VS Code and install the dependencies via the integrated terminal:

```
# Install package dependencies
npm install

# Install global dependencies
npm install -g mocha
```

Now we're ready to learn how to debug a JavaScript project in VS Code.

Debugging JavaScript in VS Code

The first file I'd like you to look at is `src/places.js`. You'll need to open the `debug-project` folder in VS Code (*File > Open Folder*) and select the file from within the editor.

```
const places = [];

module.exports = {
  places,

  addPlace: (city, country) => {
    const id = ++places.length;
    let numType = 'odd';
    if (id % 2) {
      numType = 'even';
    }
    places.push({
      id, city, country, numType,
    });
  },
};
```

The code is pretty simple, and if you have enough experience in coding you might notice it has a couple of bugs. If you do notice them, please ignore them.

If not, perfect. Let's add a few of lines at the bottom to manually test the code:

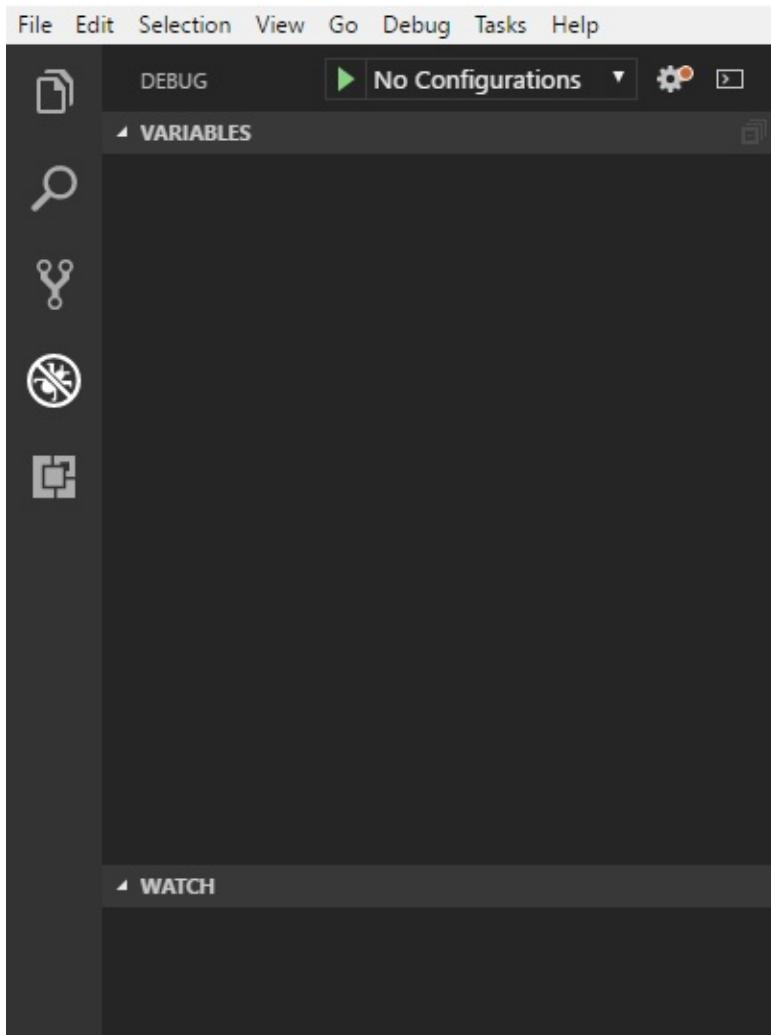
```
module.exports.addPlace('Mombasa', 'Kenya');
module.exports.addPlace('Kingston', 'Jamaica');
module.exports.addPlace('Cape Town', 'South Africa');
```

Now, I'm sure you're itching to do a `console.log` to output the value of `places`. But let's not do that. Instead, let's add **breakpoints**. Simply add them by left-clicking on the gutter — that is, the blank space next to the line numbers:



```
5
6  addPlace: (city, country) => {
7    const id = ++places.length;
8    let numType = 'odd';
9    if (id % 2) {
10      numType = 'even';
11    }
12    places.push({
13      id, city, country, numType,
14    });
15  },
16};
17
● 18 module.exports.addPlace('Mombasa', 'Kenya');
● 19 module.exports.addPlace('Kingston', 'Jamaica');
● 20 module.exports.addPlace('Cape Town', 'South Africa');
● 21 |
```

See the red dots on the side? Those are the breakpoints. A breakpoint is simply a visual indication telling the debugger tool where to pause execution. Next, on the action bar, click the debug button (the icon that says “No Bugs Allowed”).



Look at the top section. You'll notice there's a gear icon with a red dot. Simply click on it. A debug configuration file, `launch.json`, will be created for you. Update the config like this so that you can run VS Code's debugger on `places.js`:

```
"configurations": [
  {
    "type": "node",
    "request": "launch",
    "name": "Launch Places",
    "program": "${workspaceFolder}\\\\src\\\\places.js"
  }
]
```

Backslashes

Depending on your operating system, you might have to replace the double backslash (\\) with a single forward slash (/).

After you've saved the file, you'll notice that the debug panel has a new dropdown, *Launch Places*. To run it, you can:

- hit the Green Play button on the debug panel
- press F5
- click *Debug > Start Debugging* on the menu bar.

Use whatever method you like and observe the debug process in action:

```
const places = [];

module.exports = {
  places,
  addPlace: (city, country) => {
    const id = ++places.length;
    let numType = 'odd';
    if (id % 2) {
      numType = 'even';
    }
    places.push({
      id, city, country, numType,
    });
  },
};

module.exports.addPlace('Mombasa', 'Kenya');
module.exports.addPlace('Kingston', 'Jamaica');
module.exports.addPlace('Cape Town', 'South Africa');
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
C:\Program Files\nodejs\node.exe --inspect-brk=4747 src\places.js
Debugger listening on ws://127.0.0.1:4747/387dee87-c24e-4699-a1b5-2db735a51199

CALL STACK	PAUSED ON BREAKPOINT
(anonymous function)	places.js 18:8
Module._compile	module.js 649:14
Module._extensions..js	module.js 663:10
Module.load	module.js 565:32
tryModuleLoad	module.js 505:12
Module._load	module.js 497:9

A number of things happen in quick succession once you hit the debug button. First, there's a toolbar that appears at the top of the editor. It has the following controls:

- **Drag Dots anchor:** for moving the toolbar to somewhere it's not blocking anything
- **Continue:** continue the debugging session
- **Step over:** execute code line by line, skipping functions
- **Step into:** execute code line by line, going inside functions
- **Step out:** if already inside a function, this command will take you out
- **Restart:** restarts the debugging session
- **Stop:** stops the debugging session.

Right now, you'll notice that the debug session has paused on your first breakpoint. To continue the session, just hit the *Continue* button, which will cause execution to continue until it reaches the second breakpoint and pause again. Hitting *Continue* again will complete the execution and the debugging session will complete.

Let's start the debugging process again by hitting F5. Make sure the two breakpoints are still in place. When you place a breakpoint, the code pauses at the specified line. It doesn't execute that line unless you hit *Continue* (F5) or *Step Over* (F10). Before you hit anything, let's take a look at the sections that make up the debug panel:

- **Variables:** displays local and global variables within the current scope (i.e. at the point of execution)
- **Watch:** you can manually add expressions of variables you want to monitor
- **Call Stack:** displays a call stack of the highlighted code
- **Breakpoints:** displays a list of files with breakpoints, along with their line numbers.

To add an expression to the *Watch* section, simply click the + sign and add any valid JavaScript expression — such as `places.length`. When the debugger pauses, if your expression is in scope, the value will be printed out. You can also hover over variables that are currently in scope. A popup will appear displaying their values.

Currently the `places` array is empty. Press any navigation control to see how debugging works. For example, *Step over* will jump into the next line, while *Step into* will navigate to the `addPlace` function. Take a bit of time to get familiar with the controls.

As soon as you've done some stepping, hover over the `places` variable. A popup will appear. Expand the values inside until you have a similar view:

```

1  const places = [];
2
3  module.exports = {
4    places,
5
6    addPlace: (city, country) => {
7      const id = ++places.length;
8      let numType = 'odd';
9      if (id % 2) {
10        numType = 'even';
11      }
12      places.push({
13        id, city, country, numType,
14      });
15    },
16  };
17
18  module.exports.addPlace('Mombasa', 'Kenya');
19  module.exports.addPlace('Kingston', 'Jamaica');
20  module.exports.addPlace('Cape Town', 'South Africa');
21

```

You can also inspect all variables that are in scope in the *Variables* section.

```

1  const places = [];
2
3  module.exports = {
4    places,
5
6    addPlace: (city, country) => {
7      const id = ++places.length;
8      let numType = 'odd';
9      if (id % 2) {
10        numType = 'even';
11      }
12      places.push({
13        id, city, country, numType,
14      });
15    },
16  };
17
18  module.exports.addPlace('Mombasa', 'Kenya');
19  module.exports.addPlace('Kingston', 'Jamaica');
20  module.exports.addPlace('Cape Town', 'South Africa');
21

```

That's pretty awesome compared to what we normally do with `console.log`. The debugger allows us to inspect variables at a deeper level. You may have also noticed a couple of problems with the `places` array output:

1. there are multiple blanks in the array — that is, `places[0]` and `places[2]` are `undefined`
2. the `numType` property displays `even` for odd `id` values.

For now, just end the debugging session. We'll fix them in the next section.

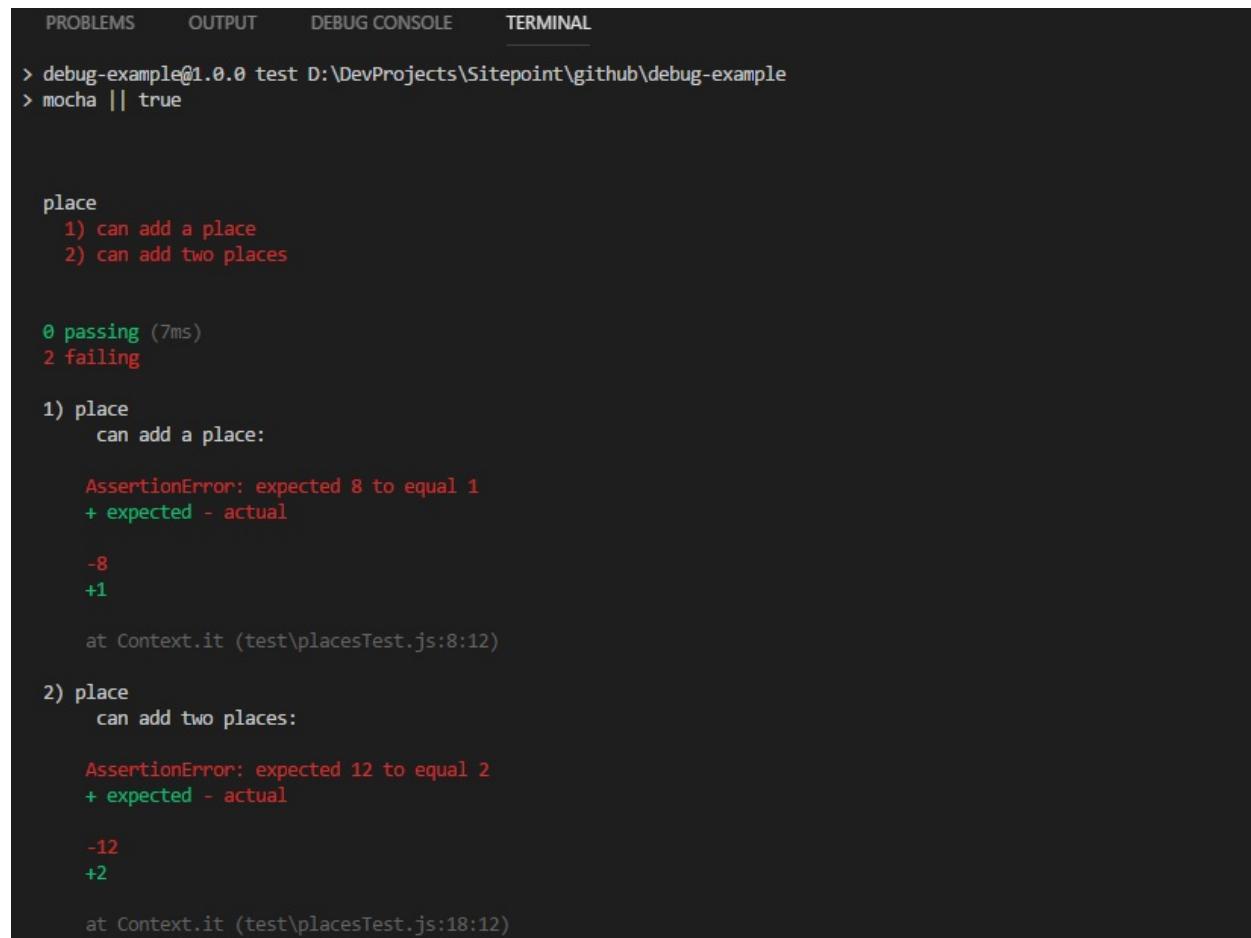
Debugging Tests with Mocha

Open test/placesTest.js and review the code that's been written to test the code in places.test. If you've never used Mocha before, you need to install it globally first in order to run the tests.

```
# Install mocha globally
npm install -g mocha

# Run mocha tests
mocha
```

You can also run `npm test` to execute the tests. You should get the following output:



The screenshot shows the VS Code interface with the 'TERMINAL' tab selected. The terminal window displays the following output from running Mocha tests:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

> debug-example@1.0.0 test D:\DevProjects\Sitepoint\github\debug-example
> mocha || true

place
  1) can add a place
  2) can add two places

  0 passing (7ms)
  2 failing

  1) place
    can add a place:
      AssertionError: expected 8 to equal 1
      + expected - actual

      -8
      +1

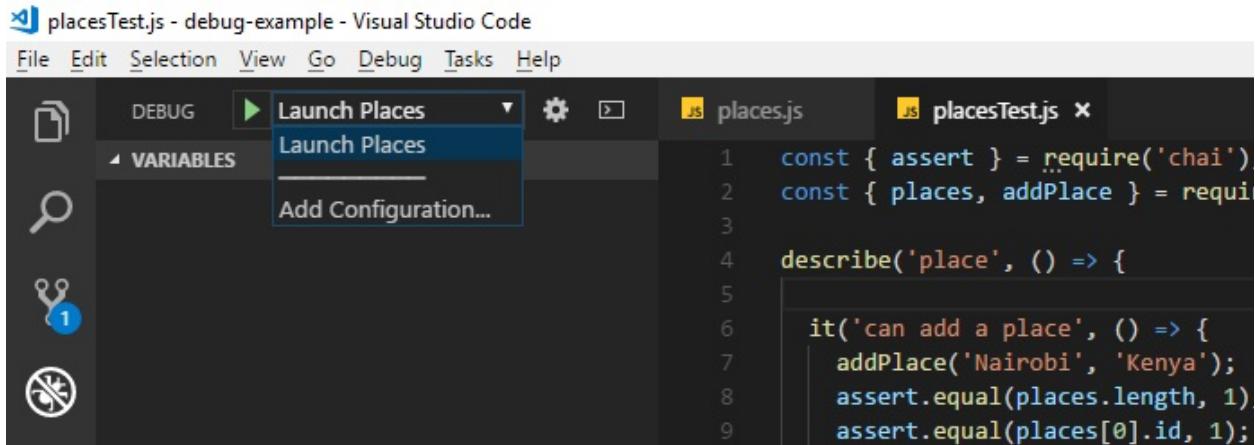
      at Context.it (test\placesTest.js:8:12)

  2) place
    can add two places:
      AssertionError: expected 12 to equal 2
      + expected - actual

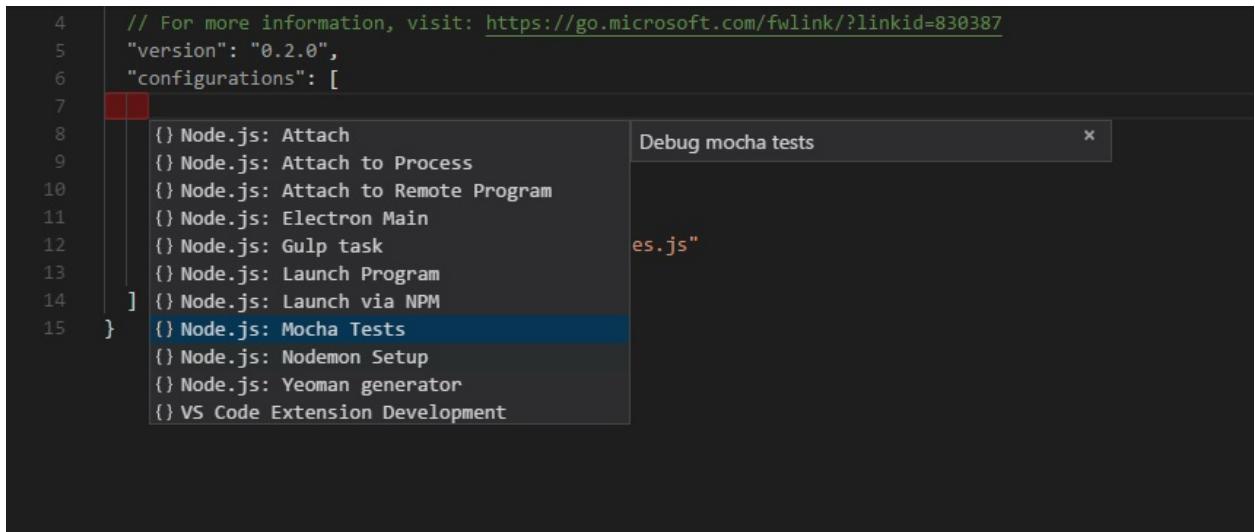
      -12
      +2

      at Context.it (test\placesTest.js:18:12)
```

All the tests are failing. To find out the problem, we're going to run the tests in debug mode. To do that, we need a new configuration. Go to the debug panel and click the dropdown to access the Add Configuration option:



The `launch.json` file will open for you with a popup listing several configurations for you to choose from.



Simply select *Mocha Tests*. The following configuration will be inserted for you:

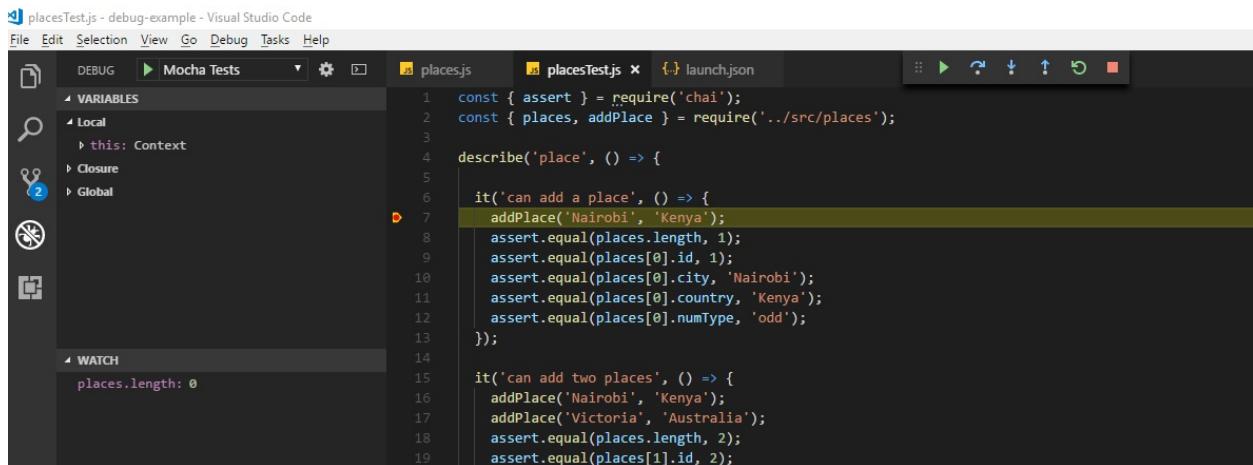
```
{
  "type": "node",
  "request": "launch",
  "name": "Mocha Tests",
  "program": "${workspaceFolder}/node_modules/mocha/bin/_mocha",
  "args": [
    "-u",
    "tdd",
    "--timeout",
    "999999",
    "--colors",
    "${workspaceFolder}/test"
  ],
  "preLaunchTask": null
}
```

```
"internalConsoleOptions": "openOnSessionStart"  
},
```

The default settings are fine. Go back to the dropdown and select *Mocha Tests*. You'll need to comment out the last three lines you added in `places.js`; otherwise the tests won't run as expected. Go back to `placesTest.js` and add a breakpoint on the line just before where the first test failure occurs. That should be line seven, where it says:

```
addPlace('Nairobi', 'Kenya');
```

Make sure to add a `places.length` expression in the watch section. Hit the *Play* button to start the debugging session.



At the start of the test, `places.length` should read zero. If you hit *Step over*, `places.length` reads 2, yet only one place has been added. How can that be?

Restart the debugging session, and this time use *Step into* to navigate to the `addPlace` function. The debugger will navigate you to `places.js`. The value of `places.length` is still zero. Click *Step over* to execute the current line.

```
const places = [];
module.exports = {
  places,
  addPlace: (city, country) => {
    const id = ++places.length;
    let numType = 'odd';
    if (id % 2) {
      numType = 'even';
    }
    places.push({
      id, city, country, numType,
    });
  },
};

// module.exports.addPlace('Mombasa', 'Kenya');
// module.exports.addPlace('Kingston', 'Jamaica');
// module.exports.addPlace('Cape Town', 'South Africa');
```

Aha! The value of `places.length` just incremented by 1, yet we haven't added anything to the array. The problem is caused by the `++` operator which is mutating the array's length. To fix this, simply replace the line with:

```
const id = places.length + 1;
```

This way, we can safely get the value of `id` without changing the value of `places.length`. While we're still in debug mode, let's try to fix another problem where the `numType` property is given the value `even` while `id` is 1. The problem seems to be the modulus expression inside the `if` statement:

```
const places = [];
module.exports = {
  places,
  addPlace: (city, country) => {
    // const id = ++places.length;
    const id = places.length + 1;
    let numType = 'odd';
    if (id % 2) {
      numType = 'even';
    }
    places.push({
      id, city, country, numType,
    });
  },
};
```

Let's do a quick experiment using the debug console. Start typing a proper expression for the `if` statement:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
C:\Program Files\nodejs\node.exe --inspect-brk=14058 node_modules\mocha\
Debugger listening on ws://127.0.0.1:14058/f540225c-d0a0-4d27-bef8-e091b

place

id
1
id % 2
1
id % 2 === 0
false

> id % 2 === 0
e Auto Attach: Off ① javascript places.js
```

The debug console is similar to the browser console. It allows you to perform experiments using variables that are currently in scope. By trying out a few ideas in the console, you can easily find the solution without ever leaving the editor. Let's now fix the failing `if` statement:

```
if (id % 2 === 0) {
  numType = 'even';
}
```

Restart the debug session and hit *Continue* to skip the current breakpoint. The first test, “can add a place”, is now passing. But the second test isn't. To fix this, we need another breakpoint. Remove the current one and place a new breakpoint on line 16, where it says:

```
addPlace('Cape Town', 'South Africa');
```

Start a new debugging session:

```

1  const { assert } = require('chai');
2  const { places, addPlace } = require('../src/places');
3
4  describe('place', () => {
5
6    it('can add a place', () => {
7      addPlace('Nairobi', 'Kenya');
8      assert.equal(places.length, 1);
9      assert.equal(places[0].id, 1);
10     assert.equal(places[0].city, 'Nairobi');
11     assert.equal(places[0].country, 'Kenya');
12     assert.equal(places[0].numType, 'odd');
13   });
14
15   it('can add two places', () => {
16     addPlace('Cape Town', 'South Africa');
17     addPlace('Victoria', 'Australia');
18     assert.equal(places.length, 2);
19     assert.equal(places[1].id, 2);
20     assert.equal(places[1].city, 'Victoria');
21     assert.equal(places[1].country, 'Australia');
22     assert.equal(places[1].numType, 'even');
23   });
24 });
25

```

There! Look at the *Variables* section. Even before the second test begins we discover that the `places` array already has existing data created by the first test. This has obviously polluted our current test. To fix this, we need to implement some kind of setup function that resets the `places` array for each test. To do this in Mocha, just add the following code before the tests:

```
beforeEach(() => {
  places.length = 0;
});
```

Restart the debugger and let it pause on the breakpoint. Now the `places` array has a clean state. This should allow our test to run unpolluted. Just click *Continue* to let the rest of the test code execute.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
C:\Program Files\nodejs\node.exe --inspect-brk=28920 node_modules\mocha\bin\_mocha -u tdd --timeout 999
Debugger listening on ws://127.0.0.1:28920/3d14da55-6333-4313-b13f-25b74d492807

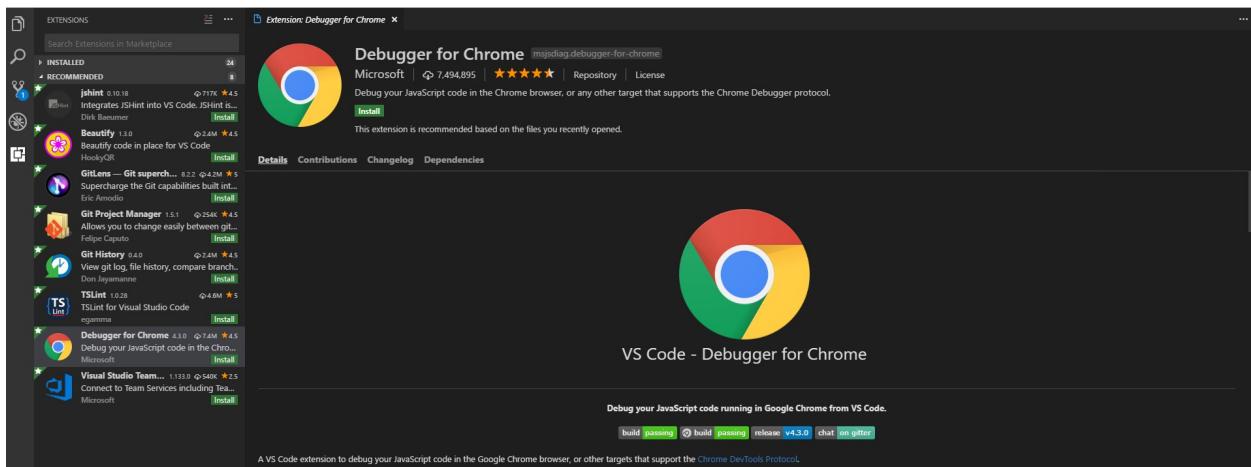
place
  ✓ can add a place
  ✓ can add two places (124762ms)
2 passing (2m)

```

All tests are now passing. You should feel pretty awesome, since you've learned how to debug code without writing a single line of `console.log`. Let's now look at how to debug client-side code using the browser.

Debugging JavaScript with Chrome Debugger

Now that you've become familiar with the basics of debugging JavaScript in VS Code, we're going to see how to debug a slightly more complex project using the [Debugger for Chrome](#) extension. Simply open the marketplace panel via the action bar. Search for the extension and install it.



After installation, hit reload to activate the extension. Let's quickly review the code that we'll be debugging. The web application is mostly a client-side JavaScript project that's launched by running an Express server:

```
const express = require('express');

const app = express();
const port = 3000;

// Set public folder as root
app.use(express.static('public'));

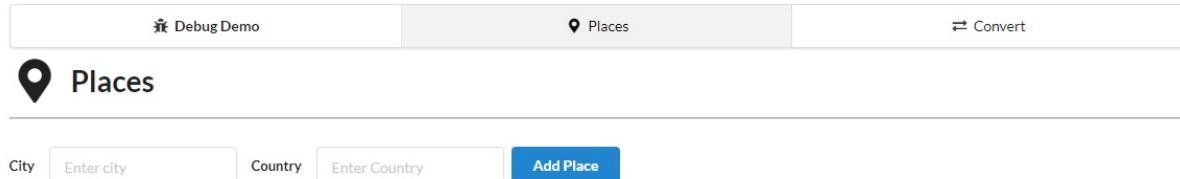
// Provide access to node_modules folder
app.use('/scripts', express.static(`__dirname/node_modules/`));

// Redirect all traffic to index.html
app.use((req, res) => res.sendFile(`__dirname/public/index.html`)

app.listen(port, () => {
  console.info('listening on %d', port);
```

```
});
```

All the client-side code is in the `public` folder. The project's dependencies include Semantic-UI-CSS, jQuery, Vanilla Router, Axios and Handlebars. This is what the project looks like when you run it with `npm start`. You'll have to open the URL localhost:3000 in your browser to view the application.



Try to add a new place. When you do, you'll see that nothing seems to be happening. Clearly something's going wrong, so it's time to look under the hood. We'll first review the code before we start our debugging session. Open `public/index.html`. Our focus currently is this section:

```
<!-- TEMPLATES -->
<!-- Places Form Template -->
<script id="places-form-template" type="text/x-handlebars-template">
  <h1 class="ui header">
    <i class="map marker alternate icon"></i>
    <div class="content"> Places</div>
  </h1>
  <hr>
  <br>
  <form class="ui form">
    <div class="fields">
      <div class="inline field">
        <label>City</label>
        <input type="text" placeholder="Enter city" id="city" name="city">
      </div>
      <div class="inline field">
        <label>Country</label>
        <input type="text" placeholder="Enter Country" name="country">
      </div>
      <div class="ui submit primary button">Add Place</div>
    </div>
  </form>
  <br>
```

```

<div id="places-table"></div>
</script>

<!-- Places Table Template -->
<script id="places-table-template" type="text/x-handlebars-template"
  <table class="ui celled striped table">
    <thead>
      <tr>
        <th>Id</th>
        <th>City</th>
        <th>Country</th>
        <th>NumType</th>
      </tr>
    </thead>
    <tbody>
      {{#each places}}
        <tr>
          <td>{{id}}</td>
          <td>{{city}}</td>
          <td>{{country}}</td>
          <td>{{numType}}</td>
        </tr>
      {{/each}}
    </tbody>
  </table>
</script>

```

If you take a quick glance, the code will appear to be correct. So the problem must be in `app.js`. Open the file and analyze the code there. Below are the sections of code you should pay attention to. Take your time to read the comments in order to understand the code.

```

// Load DOM roots
const el = $('#app');
const placesTable = $('#places-table');

// Initialize empty places array
const places = [];

// Compile Templates
const placesFormTemplate = Handlebars.compile($('#places-form-template'));
const placesTableTemplate = Handlebars.compile($('#places-table-template'));

const addPlace = (city, country) => {
  const id = places.length + 1;
  const numType = (id % 2 === 0) ? 'even' : 'odd';
  places.push({
    id, city, country, numType,
  });
}

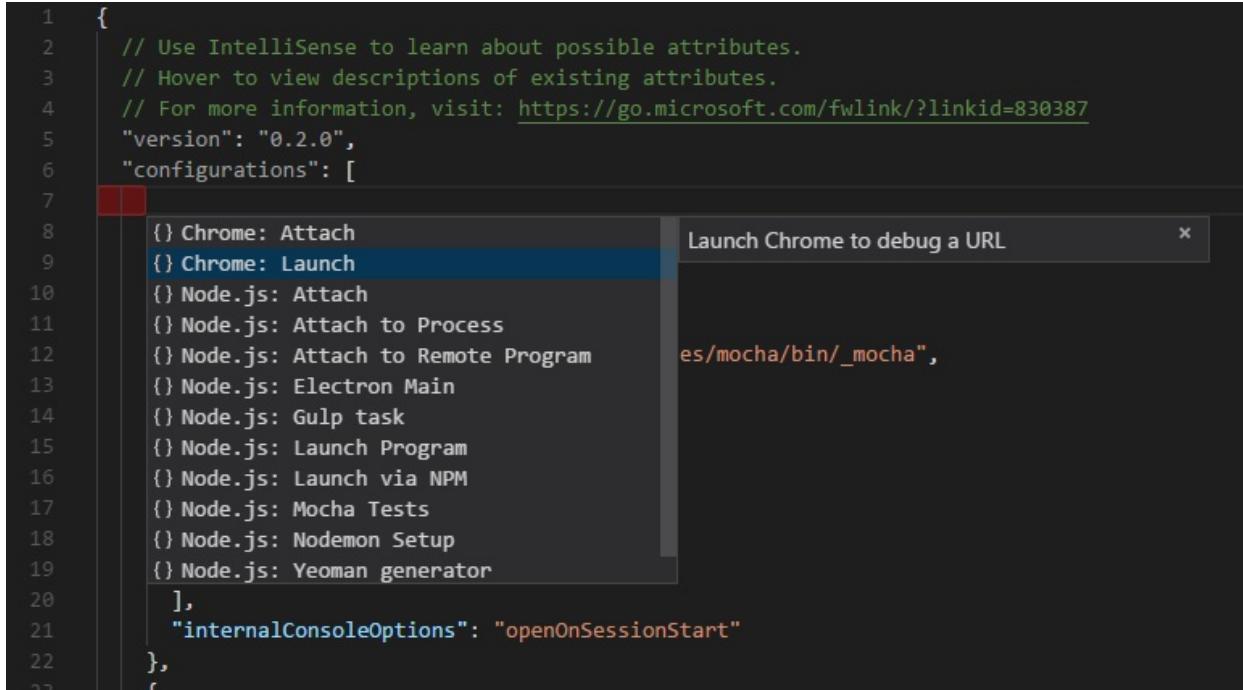
```

```
});  
};  
  
// Populate places array  
addPlace('Nairobi', 'Kenya');  
  
...  
  
// Places View - '/'  
router.add('/', () => {  
    // Display Places Form  
    const html = placesFormTemplate();  
    el.html(html);  
    // Form Validation Rules  
    $('.ui.form').form({  
        fields: {  
            city: 'empty',  
            country: 'empty',  
        },  
    });  
    // Display Places Table  
    const tableHtml = placesTableTemplate({ places });  
    placesTable.html(tableHtml);  
    $('.submit').on('click', () => {  
        const city = $('#city').val();  
        const country = $('#country').val();  
        addPlace(city, country);  
        placesTable.html(placesTableTemplate({ places }));  
        $('form').form('clear');  
        return false;  
    });  
});
```

Everything seems fine. But what could be the problem? Let's place a breakpoint on line 53 where it says:

```
placesTable.html(tableHtml);
```

Next, create a *Chrome* configuration via the debug panel. Select the highlighted option:

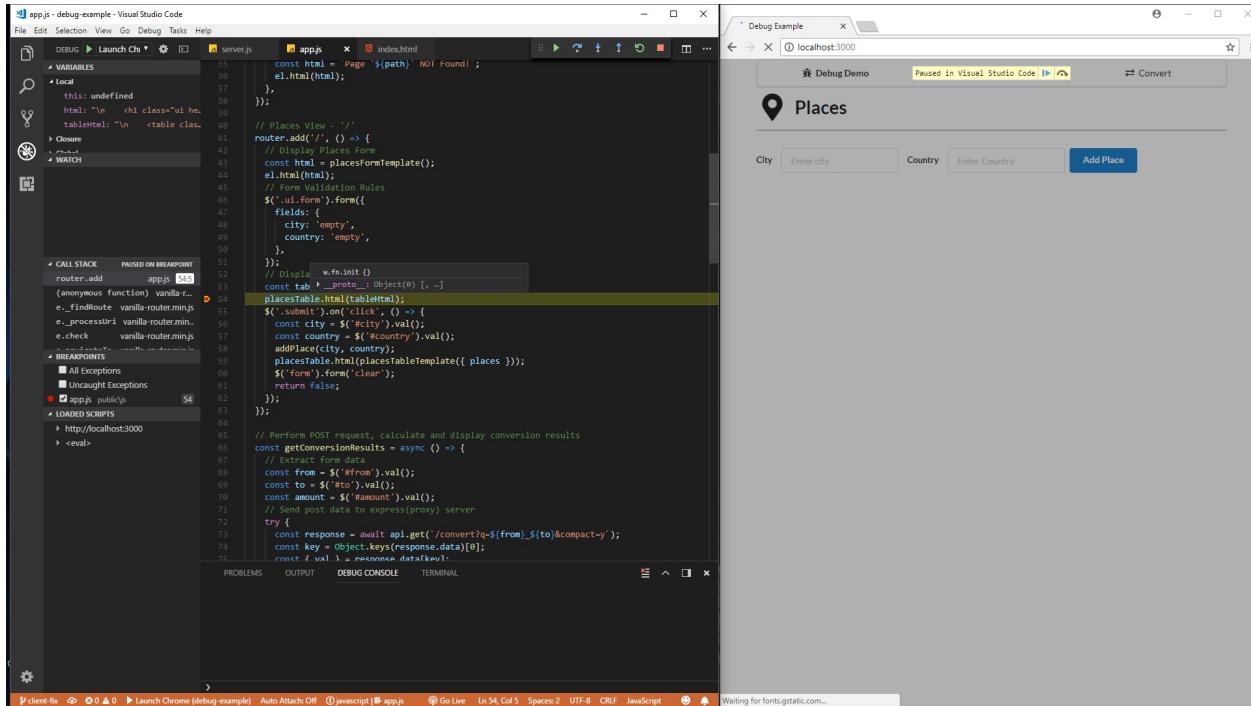


```
1  {
2    // Use IntelliSense to learn about possible attributes.
3    // Hover to view descriptions of existing attributes.
4    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
5    "version": "0.2.0",
6    "configurations": [
7      {
8        "type": "chrome",
9        "request": "launch",
10       "name": "Launch Chrome to debug a URL",
11       "url": "http://localhost:3000",
12       "webRoot": "${workspaceFolder}/public"
13     }
14   ],
15   "internalConsoleOptions": "openOnSessionStart"
16 },
17 }
```

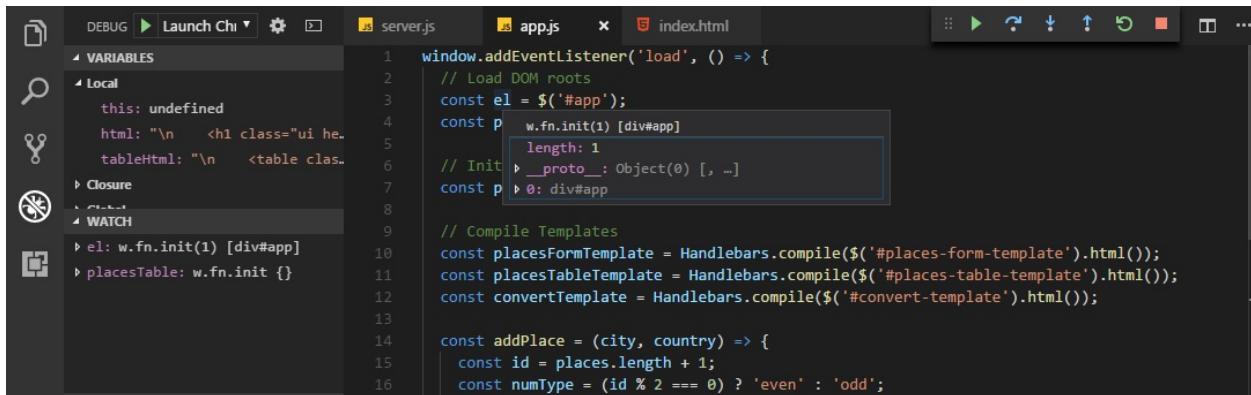
Then update the Chrome config as follows to match our environment:

```
{
  "type": "chrome",
  "request": "launch",
  "name": "Launch Chrome",
  "url": "http://localhost:3000",
  "webRoot": "${workspaceFolder}/public"
},
```

Next, start the server as normal using `npm start` or `node server`. Then select *Launch Chrome* and start the debugging session. A new instance of Chrome will be launched in debug mode and execution should pause where you set the breakpoint. Now's a good time to position Visual Studio Code and the Chrome instance side by side so you can work efficiently.



Mouse over the `placesTable` constant. A popup appears, but it seems empty. In the watch panel, add the expressions `e1` and `placesTable`. Or, alternatively, just scroll up to where the constants have been declared.



Notice that `e1` is populated but `placesTable` is empty. This means that jQuery was unable to find the element `#places-table`. Let's go back to `public/index.html` and find where this `#places-table` is located.

Aha! The table div we want is located on line 55, right inside the `places-form-template`. This means the div `#places-table` can only be found after the template, `places-form-template`, has been loaded. To fix this, just go back to `app.js` and move the code to line 52, right after the “Display Places Table”

comment:

```
const placesTable = $('#places-table');
```

Save the file, and restart the debugging session. When it reaches the breakpoint, just hit *Continue* and let the code finish executing. The table should now be visible:

Id	City	Country	NumType
1	Nairobi	Kenya	odd

You can now remove the breakpoint. Let's try adding a new place — for example, Cape Town, South Africa

The screenshot shows a browser window titled "Debug Example" at "localhost:3000". The main content is a "Places" application. At the top, there are three buttons: "Debug Demo", "Places", and "Convert". Below this is a form with two input fields: "Enter city" and "Enter Country", followed by a blue "Add Place" button. A table below the form lists two entries:

Id	City	Country	NumType
1	Nairobi	Kenya	odd
2	Cape Town		even

Hmm ... that's not right. The place is added, but the country is not being displayed. The problem obviously isn't the HTML table code, since the first row has the country cell populated, so something must be happening on the JavaScript side. Open `app.js` and add a breakpoint on line 58 where it says:

```
addPlace(city, country);
```

Restart the debug session and try to add a new place again. The execution should pause at the breakpoint you just set. Start hovering over the relevant variables. You can also add expressions to the watch panel, as seen below:

```

    DEBUG Launch Chrome server.js app.js index.html
    ▾ VARIABLES
    ▾ Local
      this: undefined
      city: "Cape Town"
      country: undefined
    ▾ Closure
    ▾ Closure
    ▾ WATCH
      $('#country'): w.fn.init {}
      $('#city'): w.fn.init(1) [input#city]
    
```

```

43 el.html(html);
44 // Form Validation Rules
45 $('.ui.form').form({
46   fields: {
47     city: 'empty',
48     country: 'empty',
49   },
50 });
51 // Display Places Table
52 const placesTable = $('#places-table');
53 const tableHtml = placesTableTemplate({ places });
54 placesTable.html(tableHtml);
55 $('.submit').on('click', () => {
56   const city [undefined].val();
57   const country = $('#country').val();
58   addPlace(city, country);
59   placesTable.html(placesTableTemplate({ places }));
60   $('form').form('clear');
61   return false;
62 });
63 });

```

As you can see, the `country` variable is `undefined`, but the `city` variable is. If you look at the jQuery selector expressions that have been set up in the watch panel, you'll notice that the `#country` selector returns nothing. This means it wasn't present in the DOM. Head over to `index.html` to verify.

Alas! If you look at line 59 where the `country` input has been defined, it's missing the `ID` attribute. You need to add one like this:

```
<input type="text" placeholder="Enter Country" name="country" id="co
```

Restart the debugging session and try to add a new place.

The browser window title is "Debug Example". The address bar shows "localhost:3000". The page header includes "Debug Demo", "Places", and "Convert" buttons. The main content area is titled "Places" with a location pin icon. It features two input fields: "City" with placeholder "Enter city" and "Country" with placeholder "Enter Country". A blue "Add Place" button is positioned next to the country field. Below these is a table with the following data:

Id	City	Country	NumType
1	Nairobi	Kenya	odd
2	Cape Town	South Africa	even

It now works! Great job fixing another bug without `console.log`. Let's now move on to our final bug.

Debugging Client-side Routing

Click the *Convert* link in the navigation bar. You should be taken to this view to perform a quick conversion:

The screenshot shows a user interface for a currency converter. At the top, there is a navigation bar with three items: 'Debug Demo', 'Places', and 'Convert'. Below the navigation bar is a title 'Convert Currency' with a double arrow icon. The main form has three input fields: 'From' (set to 'United States Dollar'), 'To' (set to 'Kenyan Shilling'), and 'Amount' (set to '2500'). A blue 'Convert' button is located below these fields. The result of the conversion, 'KES 249500.0075', is displayed in a large, bold, black font within a rounded rectangle at the bottom of the form.

That runs fine. No bugs there.

Actually there are, and they have nothing to do with the form. To spot them, refresh the page.

As soon as you hit reload, the user is navigated back to `/`, the root of the app. This is clearly a routing problem which the Vanilla Router package is suppose to handle. Head back to `app.js` and look for this line:

```
router.navigateTo(window.location.path);
```

This piece of code is supposed to route users to the correct page based on the URL provided. But why isn't it working? Let's add a breakpoint here, then navigate back to the `/convert` URL and try refreshing the page again.

As soon as you refresh, the editor pauses at the breakpoint. Hover over the

`express` `windows.location.path`. A popup appears which says the value is `undefined`. Let's go to the debug console and start typing the expression below:

The screenshot shows the VS Code interface with the debugger extension active. The code editor displays a file named `app.js` with the following content:

```
121 // eslint-disable-next-line no-console
122 console.error(error);
123 }
124 });
125
126 router.navigateTo(window.location.path);
127
128 // Navigate to clicked route
129 $('a').on('click', (event) => {
130   // Block page load
131   event.preventDefault();
132
133   // Highlight Active Menu on Click
134   const target = $(event.target);
135   $('.item').removeClass('active');
136   target.addClass('active');
137
138 // Navigate to clicked url
```

The line `126 router.navigateTo(window.location.path);` is highlighted with a yellow background, indicating it is the current line of execution.

The left sidebar shows the `CALL STACK`, `BREAKPOINTS` (with `app.js` checked), and `LOADED SCRIPTS`.

The bottom status bar shows the file name `client-fix*`, line count `0`, column count `0`, and the command `Launch Chrome (debug-example)`.

The `DEBUG CONSOLE` tab is selected, and the console output shows the following:

```
 pathname
propertyIsEnumerable
> window.location.pa|
```

Hold up! The debug console just gave us the correct expression. It's supposed to read `window.location.pathname`. Correct the line of code, remove the breakpoint and restart the debugging session.

Navigate to the `/convert` URL and refresh. The page should reload the correct path. Awesome!

That's the last bug we're going to squash, but I do recommend you keep on experimenting within the debug session. Set up new breakpoints in order to inspect other variables. For example, check out the `response` object in the `router('/convert')` function. This demonstrates how you can use a debug session to figure out the data structure returned by an API request when dealing with new REST endpoints.

The screenshot shows the Visual Studio Code interface with the title bar "app.js - debug-example - Visual Studio Code". The menu bar includes File, Edit, Selection, View, Go, Debug, Tasks, Help. The top bar has tabs for DEBUG, Launch Chrome, server.js, app.js (which is active), and index.html. On the left is a sidebar with icons for Variables, Block, this: e.Page, response: Object, results: Object, and a search icon. The main editor area shows code with line numbers 96 to 117. A red dot at line 107 indicates a breakpoint. The code uses ES6 syntax like async functions and template literals. To the right of the editor is a sidebar showing the variable "results" expanded to show objects for various currencies like AED, AFN, ALL, AMD, ANG, AOA, ARS, AUD, AWG, AZN, BAM, BBD, BDT, and BGN.

Summary

Now that we've come to the end of this tutorial, you should be proud of yourself for learning a vital skill in programming. Learning how to debug code properly will help you fix errors faster. You should be aware, however, that this article only scratches the surface of what's possible, and you should take a look at the complete [debugging documentation](#) for VS Code. Here you'll find more details about specific commands and also types of breakpoints we haven't covered, such as [Conditional Breakpoints](#).

I hope from now on you'll stop using `console.log` to debug and instead reach for VS Code to start debugging JavaScript like a pro!

Chapter 7: Introducing Axios, a Popular, Promise-based HTTP Client

by Nilson Jacques

[Axios](#) is a popular, promise-based HTTP client that sports an easy-to-use API and can be used in both the browser and Node.js.

Making HTTP requests to fetch or save data is one of the most common tasks a client-side JavaScript application will need to do. Third-party libraries — especially jQuery — have long been a popular way to interact with the more verbose browser APIs, and abstract away any cross-browser differences.

As people move away from jQuery in favor of improved native DOM APIs, or front-end UI libraries like React and Vue.js, including it purely for its `$.ajax` functionality makes less sense.

Let's take a look at how to get started using Axios in your code, and see some of the features that contribute to its popularity among JavaScript developers.

Axios vs Fetch

As you're probably aware, modern browsers ship with the newer [Fetch API](#) built in, so why not just use that? There are several differences between the two that many feel gives Axios the edge.

One such difference is in how the two libraries treat [HTTP error codes](#). When using Fetch, if the server returns a 4xx or 5xx series error, your `catch()` callback won't be triggered and it is down to the developer to check the response status code to determine if the request was successful. Axios, on the other hand, will reject the request promise if one of these status codes is returned.

Another small difference, which often trips up developers new to the API, is that

Fetch doesn't automatically send cookies back to the server when making a request. It's necessary to explicitly pass an option for them to be included. Axios has your back here.

One difference that may end up being a show-stopper for some is progress updates on uploads/downloads. As Axios is built on top of the older XHR API, you're able to register callback functions for `onUploadProgress` and `onDownloadProgress` to display the percentage complete in your app's UI. Currently, Fetch has no support for doing this.

Lastly, Axios can be used in both the browser and Node.js. This facilitates sharing JavaScript code between the browser and the back end or doing server-side rendering of your front-end apps.

Fetch API for Node

there are versions of the [Fetch API available for Node](#) but, in my opinion, the other features Axios provides give it the edge.

Installing

As you might expect, the most common way to install Axios is via the npm package manager:

```
npm i axios
```

and include it in your code where needed:

```
// ES2015 style import
import axios from 'axios';

// Node.js style require
const axios = require('axios');
```

If you're not using some kind of module bundler (e.g. webpack), then you can always pull in the library from a CDN in the traditional way:

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

Browser support

Axios works in all modern web browsers, and Internet Explorer 8+.

Making Requests

Similar to jQuery's `$.ajax` function, you can make any kind of HTTP request by passing an options object to Axios:

```
axios({
  method: 'post',
  url: '/login',
  data: {
    user: 'brunos',
    lastName: 'ilovenodejs'
  }
});
```

Here, we're telling Axios which HTTP method we'd like to use (e.g. GET/POST/DELETE etc.) and which URL the request should be made to.

We're also providing some data to be sent along with the request in the form of a simple JavaScript object of key/value pairs. By default, Axios will serialize this as JSON and send it as the request body.

Request Options

There are a whole bunch of [additional options](#) you can pass when making a request, but here are the most common ones:

- `baseURL`: if you specify a base URL, it'll be prepended to any relative URL you use.
- `headers`: an object of key/value pairs to be sent as headers.
- `params`: an object of key/value pairs that will be serialized and appended to the URL as a query string.
- `responseType`: if you're expecting a response in a format other than JSON, you can set this property to `arraybuffer`, `blob`, `document`, `text`, or `stream`.
- `auth`: passing an object with `username` and `password` fields will use these credentials for HTTP Basic auth on the request.

Convenience methods

Also like jQuery, there are shortcut methods for performing different types of request.

The get, delete, head and options methods all take two arguments: a URL, and an optional config object.

```
axios.get('/products/5');
```

The post, put, and patch methods take a data object as their second argument, and an optional config object as the third:

```
axios.post(
  '/products',
  { name: 'Waffle Iron', price: 21.50 },
  { options }
);
```

Receiving a Response

Once you make a request, Axios returns a promise that will resolve to either a response object or an error object.

```
axios.get('/product/9')
  .then(response => console.log(response))
  .catch(error => console.log(error));
```

The response object

When the request is successful, your then() callback will receive a response object with the following properties:

- data: the payload returned from the server. By default, Axios expects JSON and will parse this back into a JavaScript object for you.
- status: the HTTP code returned from the server.
- statusText: the HTTP status message returned by the server.
- headers: all the headers sent back by the server.
- config: the original request configuration.
- request: the actual XMLHttpRequest object (when running in a browser).

The error object

If there's a problem with the request, the promise will be rejected with an error object containing at least some of the following properties:

- message: the error message text.
- response: the response object (if received) as described in the previous section.
- request: the actual XMLHttpRequest object (when running in a browser).
- config: the original request configuration.

Transforms and Interceptors

Axios provides a couple of neat features inspired by [Angular's \\$http library](#). Although they appear similar, they have slightly different use cases.

Transforms

Axios allows you to provide functions to transform the outgoing or incoming data, in the form of two configuration options you can set when making a request: `transformRequest` and `transformResponse`. Both properties are arrays, allowing you to chain multiple functions that the data will be passed through.

Any functions passed to `transformRequest` are applied to PUT, POST and PATCH requests. They receive the request data, and the headers object as arguments and must return a modified data object

```
const options = {
  transformRequest: [
    (data, headers) => {
      // do something with data
      return data;
    }
  ]
}
```

Functions can be added to `transformResponse` in the same way, but are called only with the response data, and before the response is passed to any chained `then()` callbacks.

So what could we use transforms for? One potential use case is dealing with an API that expects data in a particular format — say XML or even CSV. You could

set up a pair of transforms to convert outgoing and incoming data to and from the format the API requires.

It's worth noting that Axios' default `transformRequest` and `transformResponse` functions transform data to and from JSON, and specifying your own transforms will override these.

Interceptors

While transforms let you modify outgoing and incoming data, Axios also allows you to add functions called [interceptors](#). Like transforms, these functions can be attached to fire when a request is made, or when a response is received.

```
// Add a request interceptor
axios.interceptors.request.use((config) => {
  // Do something before request is sent
  return config;
}, (error) => {
  // Do something with request error
  return Promise.reject(error);
});

// Add a response interceptor
axios.interceptors.response.use((response) => {
  // Do something with response data
  return response;
}, (error) => {
  // Do something with response error
  return Promise.reject(error);
});
```

As you might have noticed from the examples above, interceptors have some important differences from transforms. Instead of just receiving the data or headers, interceptors receive the full request config or response object.

When creating interceptors, you can also choose to provide an error handler function that allows you to catch any errors and deal with them appropriately.

Request interceptors can be used to do things such as [retrieve a token from local storage and send with all requests](#), while a response interceptor could be used to [catch 401 responses and redirect to a login page](#) for authorization.

Third-party Add-ons

Being a popular library, Axios benefits from an ecosystem of third-party libraries that extend its functionality. From interceptors to testing adaptors to loggers, there's quite a variety available. Here are a few that I think you may find useful:

- [axios-mock-adaptor](#): allows you to easily mock requests to facilitate testing your code.
- [axios-cache-plugin](#): a wrapper for selectively caching GET requests.
- [redux-axios-middleware](#): if you're a Redux user, this middleware provides a neat way to dispatch Ajax requests with plain old actions.

A list of more [Axios add-ons and extensions](#) is available on the Axios GitHub repo.

In summary, Axios has a lot to recommend it. It has a straightforward API, with helpful shortcut methods that will be familiar to anyone who's used jQuery before. Its popularity, and the availability of a variety of third-party add-ons, make it a solid choice for including in your apps, whether front end, back end, or both.