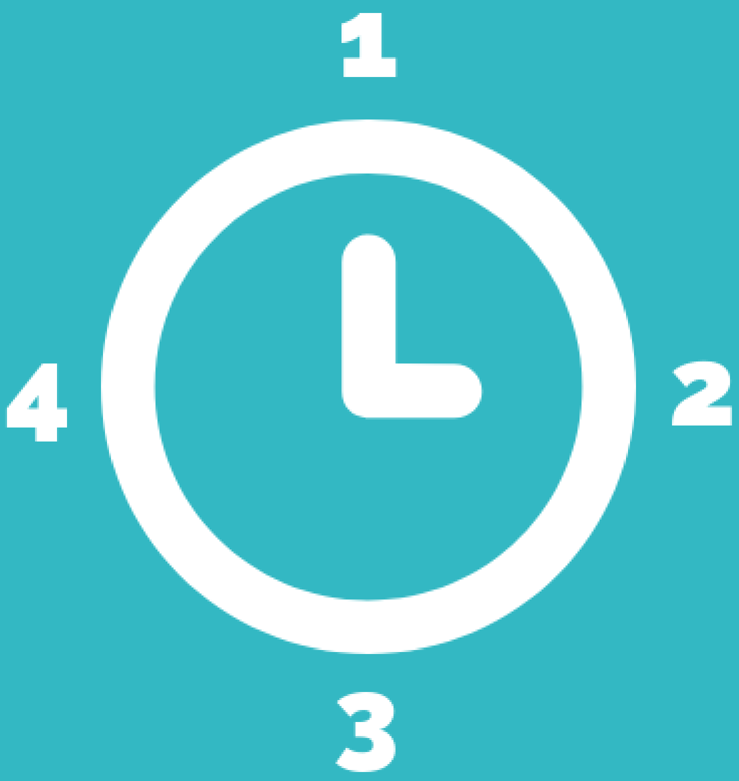
Finally understand front-end...

Front-end in four hours

Jon Hilton



Front-end in four hours

What to learn?

What should you learn when starting out with "modern" front-end development?

There's more choice than ever before and knowing where to begin is half the battle.

In many ways, the front-end ecosystem is like a thriving city....

You'll never be bored, there's always something new and exciting to explore but it can be pretty overwhelming and if you try to do everything at once you'll burn yourself out.

It pays to take it easy on yourself.

If this were a video game I'd suggest starting on easy mode.

Ultimately your mission is to get your features built, and hopefully in a way that doesn't annoy your future self (well, not too much anyway).

With that in mind, here's one thing you *should not* learn from day one...

webpack syntax

And here's one thing you *should*...

Javascript fundamentals

If you've never heard of webpack, it's a tool that's often used in front-end development to bundle, minify and package up javascript code (more on what that means later).

But, whether you're an existing back-end developer looking to go "full-stack", or just starting out on your front-end journey, you're pretty likely to wind up using one of the "big three" javascript frameworks (React.js, Vue.js or Angular) and all three will abstract webpack away so you don't need to worry about it (easy mode!)

The problem with trying to learn webpack syntax from day one is, it's exactly the kind of front-end shenanigans that will have you buried in the weeds, wasting hours trying to work out why it is (and often isn't) working.

There are many front-end rabbit holes you can find yourself down and webpack is definitely one of them.

On the other hand, even if you are using one of the frameworks, it pays dividends to grok some key javascript fundamentals early doors.

It's easy to be seduced by the frameworks and their promises, but you're going to be writing javascript, and possibly a lot of it, so knowing the fundamentals will save you a lot of time and many headaches later on.

Which brings us to this short guide.

My aim is to keep this guide as short as possible, so you can get going quickly.

You'll find it touches on all the key moving parts of the front-end ecosystem, and provides some step-by-step directions but the overall aim is to give you the big picture so you can hold you own in conversations about front-end and know where to go to undertake key tasks.

You'll discover the vital moving parts, how they interact and what to keep in mind as you build you first few front-end features.

So, take a deep breath and let's go :-)

How to use this book

This pocket guide is organised into four "hours", each one adding a new piece to the front-end puzzle.

You absolutely don't have to read it from cover-to-cover (unless you want to!)

The **first hour starts with the essential building blocks** of a web application. This is your belt and braces quick guide to the elements which make up front-end development (HTML, CSS, Javascript).

If you've done any kind of HTML, Javascript work in the past, this may be mostly be familiar to you so feel free to skim through this hour (although, there are some handy tips which you will probably still find useful!)

Hour 2 is all about today's Javascript. This is where you can start to build a picture of present day Javascript, what the browsers do (and do not) support and how you can get around browser limitations whilst still using newer Javascript language features.

Hour 3 focuses on Javascript Frameworks. What are they? Why do we need them (if we do) and what's the simplest way to get up and running with them.

And finally, **Hour 4 walks you through a process for building your front-end features**.

It's here that you'll find some really handy, repeatable steps you can take when tackling a feature in any of the popular javascript frameworks.

Each section stands by itself, so jump in as needed when you need a reminder how to do something, or can't remember how something works.

The Javascript ecosystem moves pretty quick, so if you find anything no longer works, feel free to drop me an email at jon@jonhilton.net.

With that housekeeping sorted, let's jump in to the strange and mysterious world of front-end development.

Hour 1 - The fundamentals

What is front-end

Front-end typically refers to the user interface part of an application.

You're most likely to hear "front-end" in the context of a web application (which is precisely what we're going to focus on in this short guide).

On the web, front-end means the part of our application which runs in the browser.

This can be as simple as a single html page, or as complicated as a full-blown "single page application" comprising of many javascript, html files (or templates) and style sheets (more on "SPAs" later).

Whilst it is possible to have an application which is entirely front-end, it's more common that the front-end will interact with a back-end of some sort, running on a server, which handles all the key business logic and concerns such as persisting data to a database.

What tools do you need to get started?

Throughout this book we're going to focus purely on front-end development, specifically HTML, Javascript and CSS.

To edit your HTML, CSS and Javascript code you'll need an editor of some description.

You *could* just use notepad (notepad++)...

.. but a little later in the book we'll get into some more advanced scenarios involving front-end frameworks, and a dedicated code editor makes it much easier to get those up and running.

So which editor? Well there are plenty to choose from.

Brackets

Visual Studio Code

Atom

Sublime Text

And many more besides.

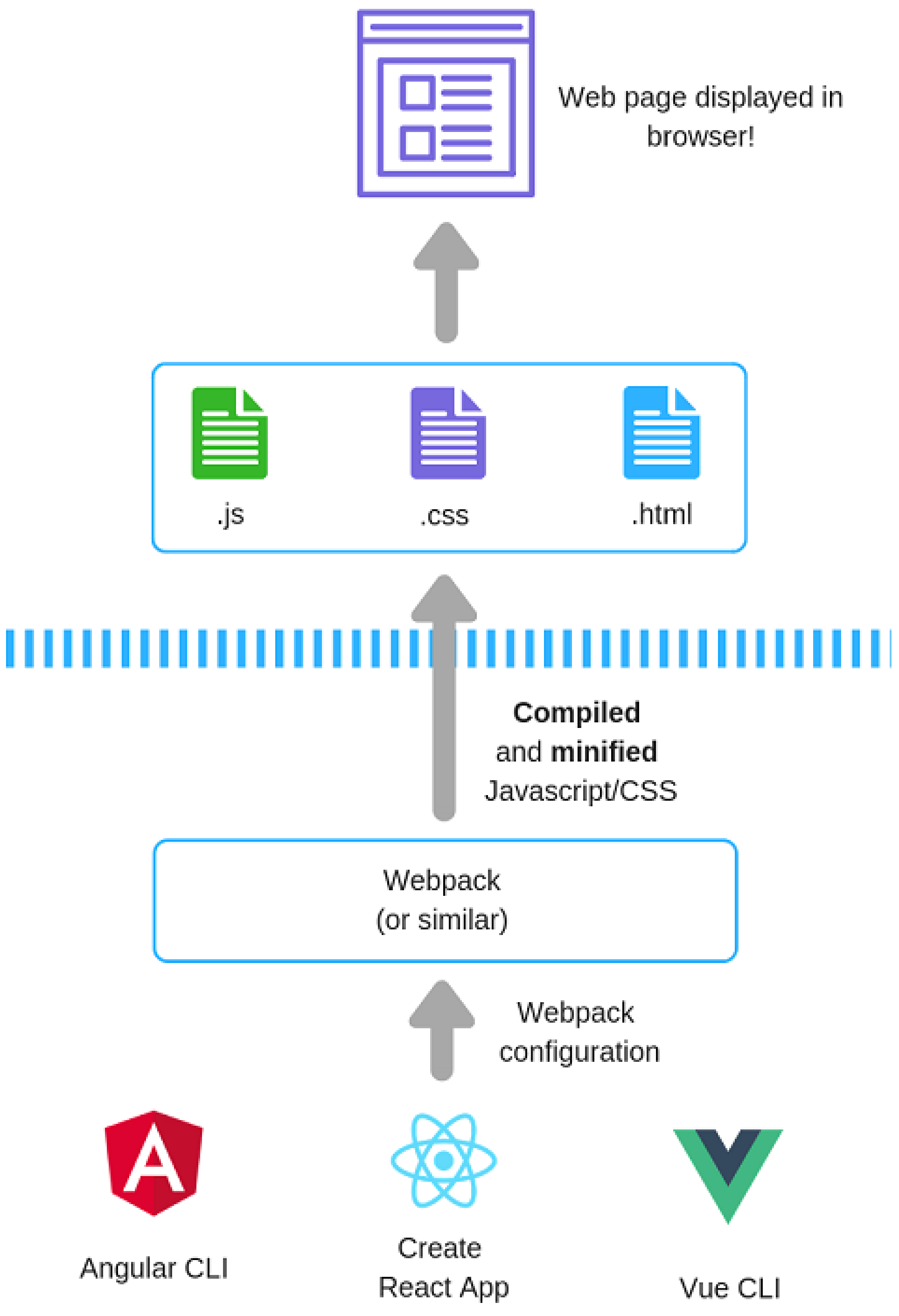
If you're just getting started, Brackets is nice and lightweight (but with enough features to keep with you as your requirements ramp up) and I've found Visual Studio Code works really well too.

I'll be using Visual Studio Code for the examples in the book.

That said, the vast majority of the steps will work in any editor and I'll make it clear when any step is "Visual Studio Code specific".

The big picture

Here's a simplified overview of web development in 2018 (and beyond, probably!)



We're going to look at each part of this particular puzzle, starting with everything above the dotted blue line...

The simplest application you can build (HTML)

1. Create a folder somewhere on your machine
2. Create a file in that folder, called index.html, and add this markup

```
<!DOCTYPE html>
<html>

<body>
    <p>Hello World</p>
</body>

</html>
```

Check it works

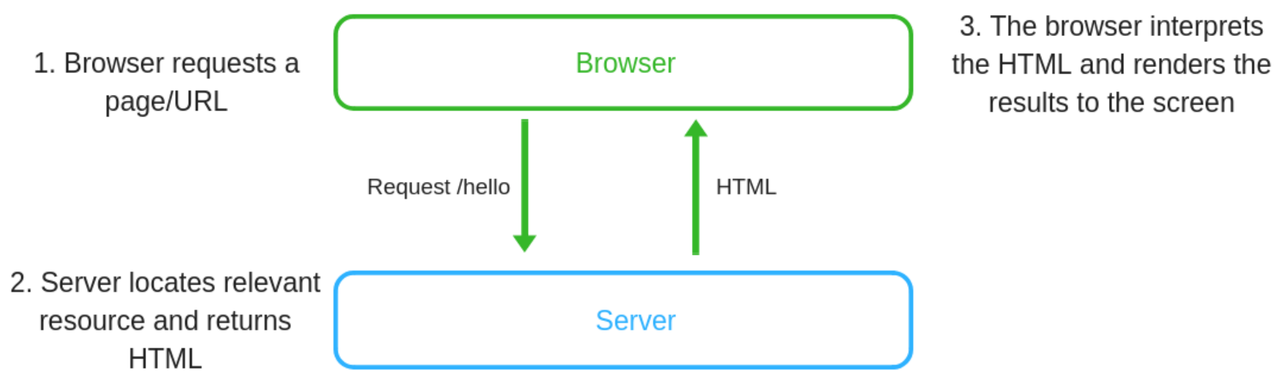
Open up a browser, then open this html file, you should see "Hello World"

Explanation

This HTML markup tells the browser what to display.

The `<html></html>` and `<body></body>` tags are key structural parts of an html page and required for a web page to be valid.

Anything within the `body` tag will be rendered by the browser. In this case, this is our `p` (paragraph) tag and the text "Hello World".


1. Browser requests a page/URL
Browser
3. The browser interprets the HTML and renders the results to the screen
Request /hello
HTML
2. Server locates relevant resource and returns HTML
Server

Jargon buster

markup

HTML is a markup language.

If you ever had a piece of work marked by a teacher, you know markup.

Your homework was "marked up" with certain codes, symbols and text (green ticks, red crosses, sarcastic comments etc.).

This mark-up gives you further feedback and instructions on how to interpret your homework.

With HTML, you (the developer) combine text and markup, which the browser then interprets when deciding how to display the text contained within the web page.

tag

Tags like `<html>`, `<p>`, `<body`> are the building blocks of HTML and indicate to the browser how to interpret the text on the page.

Text marked up with an `h1` tag will be treated as a heading, `p` as a paragraph and so on.

element

Typically you'll use two tags, an opening and a closing tag.

`<p>Hello</p>`

Together, these tags are referred to as an HTML element.

doc type

`<!DOCTYPE html>` tells the browser how to render the page. Specifically it tells the browser to render the page using "HTML 5 standards compliance mode".

In days gone by, browsers did all sorts of weird things and had their own "features" (sometimes even allowing for proprietary html tags or attributes).

This made it difficult to know how your web page would look in different browsers because they'd all render things differently.

Since HTML 5 came along, this situation has dramatically improved.

All the web browsers adhere to the standards now (for the most part) and `<!DOCTYPE html>` at the top of your html file indicates you're not using anything proprietary to any specific browser, but rather, standard html tags/elements as found in the latest official standards.

Add a little "interactivity" (Javascript)

```
<!DOCTYPE html>
<html>

<body>
    <label for="name">Your name</label>
    <input type="text" id="name" />
    <button id="sayHelloButton">Say Hello</button>
    <p id="greeting"></p>
</body>

<script>
    var button = document.getElementById('sayHelloButton');

    button.addEventListener('click', function (e) {
        document.getElementById('greeting').innerHTML = 'Hello ' +
document.getElementById('name').value;
    }, false);

</script>

</html>
```

Check it works

Open this file in the browser and you'll see an amazing looking textbox!

Pop your name in there and hit the button to be personally greeted.

Explanation

We've added a few more html tags here.

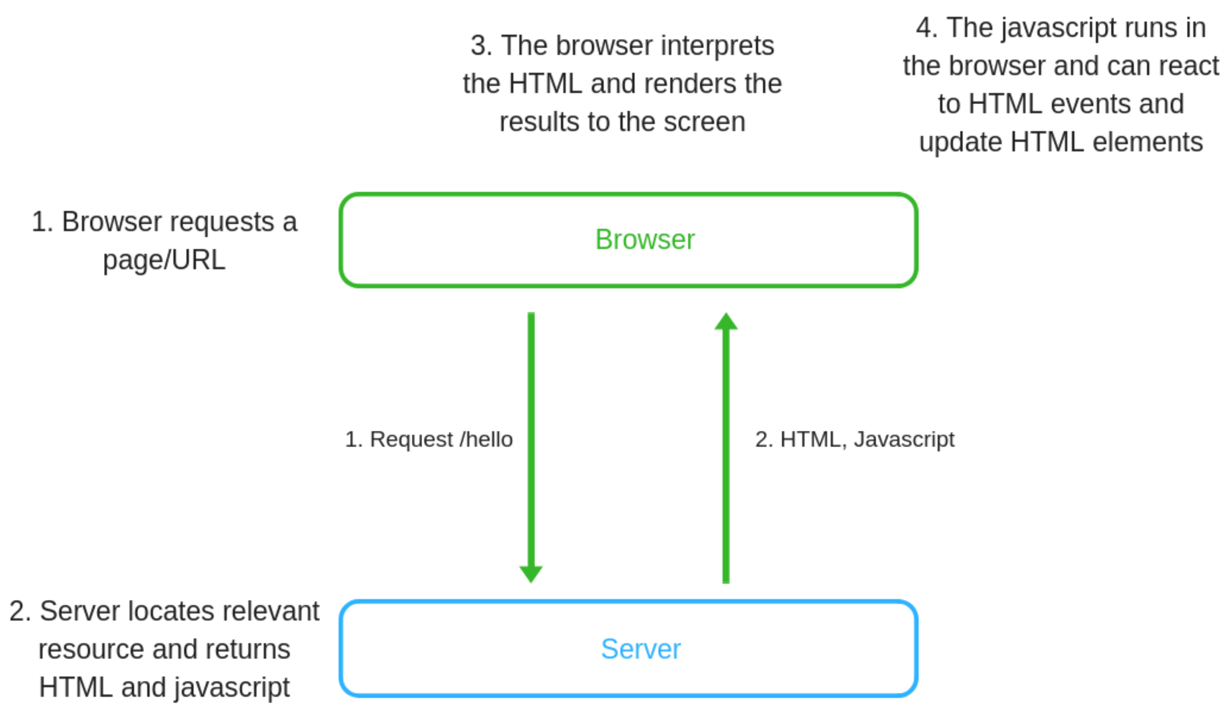
The key bit is the `button` tag.

The javascript code locates this button, and attaches an event handler to the HTML `click` event.

`click` is an HTML event, which is raised when the user clicks the button.

Everything inside `function(e){}` is our event handler code, which will run when this event fires (the user clicks the button).

In the event handler we grab the relevant element (our `greeting` paragraph) and update it to show a greeting, including the user's name (the name they entered in the text box).


3. The browser interprets the HTML and renders the results to the screen
4. The javascript runs in the browser and can react to HTML events and update HTML elements
1. Browser requests a page/URL
Browser
1. Request /hello
2. HTML, Javascript
2. Server locates relevant resource and returns HTML and javascript
Server

Jargon buster

event

Events happen to html elements and can be triggered by the browser or a user. They signify that something has happened (a user clicking a button for example).

event handler

We can write javascript code to handle these HTML events (as here, where we update the greeting when the user clicks our button).

script tag

This tells the browser that the code (contained within the `<script></script>` tags) is javascript code.

It used to be necessary to specify the script `type` here (e.g. `<script type="text/javascript"></script>` but so long as you use the HTML 5 Doc Type, you can omit the `type` attribute and the browser will assume your code is javascript.

Style it up (CSS)

```
<!DOCTYPE html>
<html>

<body>
    <label for="name">Your name</label>
    <input type="text" id="name" />
    <button id="sayHelloButton">Say Hello</button>
    <p id="greeting"></p>
</body>

<script>
    var button = document.getElementById('sayHelloButton');

    button.addEventListener('click', function (e) {
        document.getElementById('greeting').innerHTML = 'Hello ' +
document.getElementById('name').value;
    }, false);

</script>

<style>
    body {
        font-size: 1.2em;
        color: #666666;
    }

    button,
    input {
        padding: 0.6em;
        border-radius: 6px;
    }

    button {
        background-color: #866a8b;
        color: #ffffff;
    }
</style>

</html>
```

Check it works

Open this file in your browser and you should see a vaguely passable textbox, button combo!

Explanation

If you've done any front-end work it's probably etched in your soul that it's a bad idea to include your javascript and css directly in the HTML file. If so, hang tight, we're about to split this out into separate files but just before we do...

This is the third and final key element of our front-end web application.

A quick review...

HTML (markup) tells our browser what to display

Javascript lets us interact with HTML elements (and react to events)

And finally, *CSS (Cascading Style Sheets)* give us control over how the elements appear (the element's visual style).

This is a simple example of embedded styles (where the style code is included on the HTML page).

Jargon buster

Cascading Style Sheets (CSS)

A language for describing the presentation (style) of an HTML document.

Inline Styles

Style information can be included directly in HTML tags, in which case it is referred to as "inline CSS".

For example, `<h1 style="font-size: 1.3em"></h1>`

This has the disadvantage that, if all your styles are defined this way, it becomes very difficult to maintain consistency across multiple pages in your web application.

To change all headings for example you'd have to locate ever inline style (on any `h1` tags) and change every instance.

Embedded Styles

This is when the style information is included in the HTML file, within `<style></style>` tags.

Embedded styles carry the advantage that style information is loaded at the same time as the HTML and so both are available at the same time (and your users won't see a flash of unstyled content whilst the style sheet is loaded).

Has a similar disadvantage as Inline CSS, in that global changes across your entire application are difficult.

External Style Sheets

When your styles are written in separate files which are then requested by your HTML pages. This option lets you define styles in one place and re-use them from many different HTML pages, making global changes very easy.

Carries a slight disadvantage in that the style sheet has to be loaded separately to the contents of the HTML page, and this can cause a slight delay to elements appearing as they should (unstyled content).

Separate your concerns

So far we've kept all our code in one file.

This will often prove unmanageable as these HTML files grow bigger and you need to re-use some of your Javascript or CSS code in other HTML files.

You can put your styles in their own `.css` files.

index.css

```
body {
    font-size: 1.2em;
    color: #666666;
}

button,
input {
    padding: 0.6em;
    border-radius: 6px;
}

button {
    background-color: #866a8b;
    color: #ffffff;
}
```

Then reference these from your HTML.

index.html

```
<!DOCTYPE html>
<html>

<head>
    <link rel="stylesheet" type="text/css" href="index.css" />
</head>

<body>
    <label for="name">Your name</label>
    <input type="text" id="name" />
    <button id="sayHelloButton">Say Hello</button>
    <p id="greeting"></p>
</body>

<script>
    var button = document.getElementById('sayHelloButton');

    button.addEventListener('click', function (e) {
        document.getElementById('greeting').innerHTML = 'Hello ' +
document.getElementById('name').value;
    }, false);
```

```
</script>

</html>
```

Similarly, you can take the javascript code into its own file.

index.js

```
var button = document.getElementById('sayHelloButton');

button.addEventListener('click', function (e) {
    document.getElementById('greeting').innerHTML = 'Hello ' +
document.getElementById('name').value;
}, false);
```

And reference it from your HTML.

index.html

```
<!DOCTYPE html>
<html>

<head>
    <link rel="stylesheet" type="text/css" href="index.css" />
</head>

<body>
    <label for="name">Your name</label>
    <input type="text" id="name" />
    <button id="sayHelloButton">Say Hello</button>
    <p id="greeting"></p>

    <script src="index.js"></script>
</body>
</html>
```

Whilst it's generally considered good practice to split out your HTML, CSS and Javascript, precisely how you organise your code remains up to you.

Interesting, React takes a slightly different approach with it's JSX syntax, something we'll see later on.

Note that we included the reference to our index.js script just before the closing `</body>` tag.

It's equally possible to reference it in the `head` element.

```
<head>
    <script src="index.js"></script>
</head>
```

Putting it just before `</body>` though ensures everything in the HTML page (all the elements) are loaded and therefore available, prior to the script running.

If we included the script in the `head` element, it would result in an error because our `button` wouldn't have been rendered prior to the script running.

The other advantage of putting the `script` reference just before `</body>` is that the rest of the page will load before the javascript in our `index.js` file. This will result in our HTML being displayed quicker.

A brief intro to some Javascript fundamentals

Before we go much further, it's worth covering a few Javascript fundamentals...

This is potentially a very big topic so I've just included a small number of key concepts here. If you want to dig in a little deeper, I thoroughly recommend javascript.info as an excellent starting point.

Jargon buster

Variables

As with other programming languages, you can assign things to variables in javascript.

For years the standard way to do this was using `var`.

```
var name = 'Jon';
```

Nowadays though it's generally recommended to use `const` or `let`.

For the most part `let` has now replaced `var` and can be used for any variables.

`const` can be used to declare a constant (unchanging) variable.

If you try and change a `const` after declaring (and assigning it) you'll get an error.

```
const name = 'Jon';
name = 'Brian'; // error attempting to reassign a constant
```

Functions

Functions in javascript can be declared like this.

```
function sayHello(){
    alert('Hello');
}
```

Here we've created our own function `sayHello` and we're also invoking one of javascript's build-in functions (`alert`);

Another way to declare functions (as an expression) follows in this next section...

Expression

A Javascript expression is a snippet of code which results in a single value.

You can use expressions wherever Javascript expects a value.

```
console.log(2 * 9);
```

Here, `2*9` is an expression, and you can pass it to `console.log` where it will be invoked and the returned value written to the console.

As far as `console.log` is concerned, this is the same as if you had passed it the value directly.

```
console.log(18)
```

You can assign an expression to a variable like this...

```
let calculation = 2 * 9;
```

... then pass it around and use it wherever a value is expected.

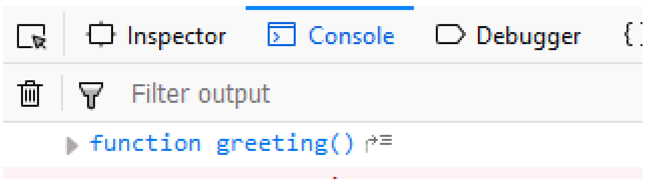
```
console.log(calculation);
```

You can also write functions as expressions and assign them to variables.

```
let greeting = function(){
    return "hello, I'm a greeting"
}
```

If you passed `greeting` to `console.log` what would you expect to see?

The answer is...



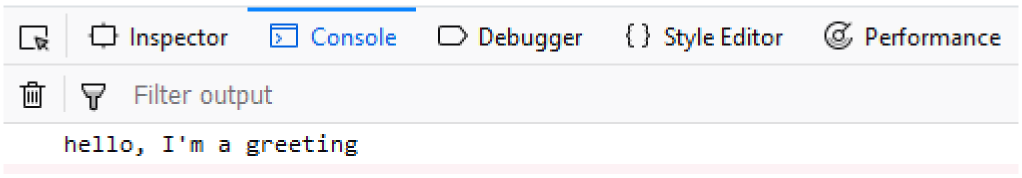
```
▸ function greeting()
```

`console.log` has logged the function itself to the console.

If you invoke the function...

```
console.log(greeting());
```

... the returned value will be written to the console.



```
hello, I'm a greeting
```

Statement

Statements are those key structures you would expect most programming languages to have.

If you're thinking of things like `if` statements or `for` loops about now, you're on the right track.

This is a statement...

```
if(a === b){

}
```

You can't pass this statement around (to invoke somewhere else), nor can you assign it to a variable.

Anonymous function

Quite literally a function without a name (identifier).

Anonymous functions can be passed around as arguments to functions, and assigned to variables.

```
let theFunction = function(){return "hello world"};
console.log(theFunction());
```

Here the anonymous function is treated as an expression and assigned to a variable.

You could write the same code without the intermediate variable...

```
console.log(function(){return "hello world"}());
```

I'll let you decide which one you'd rather see (and maintain) in your own code!

Try it for yourself

Throwing the language around and exploring it for yourself is definitely the best way to learn.

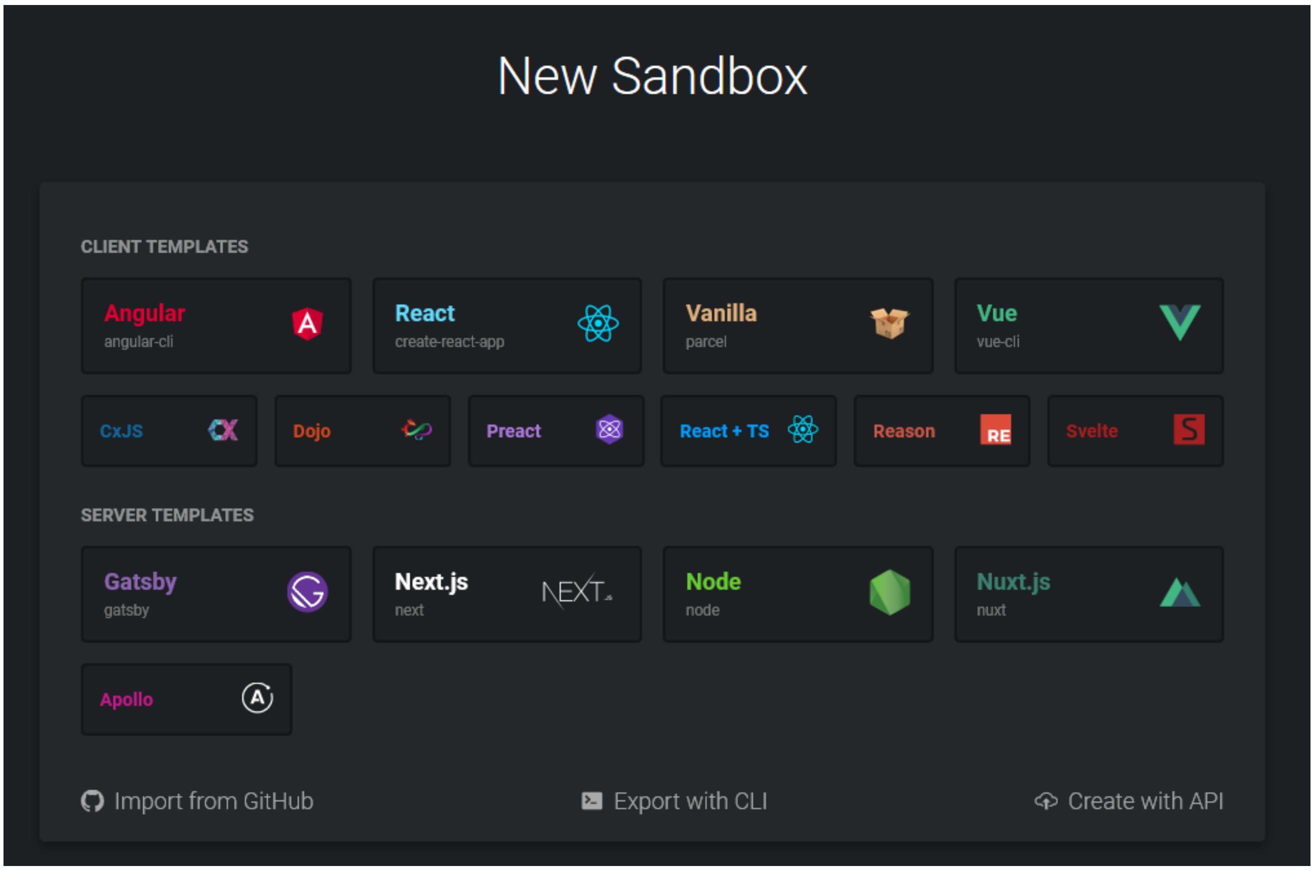
To really kick the tires on javascript, pick up a reference (like javascript.info), open up a javascript (.js) file in your editor and see how many different ways you can break stuff!

Try https://codesandbox.io

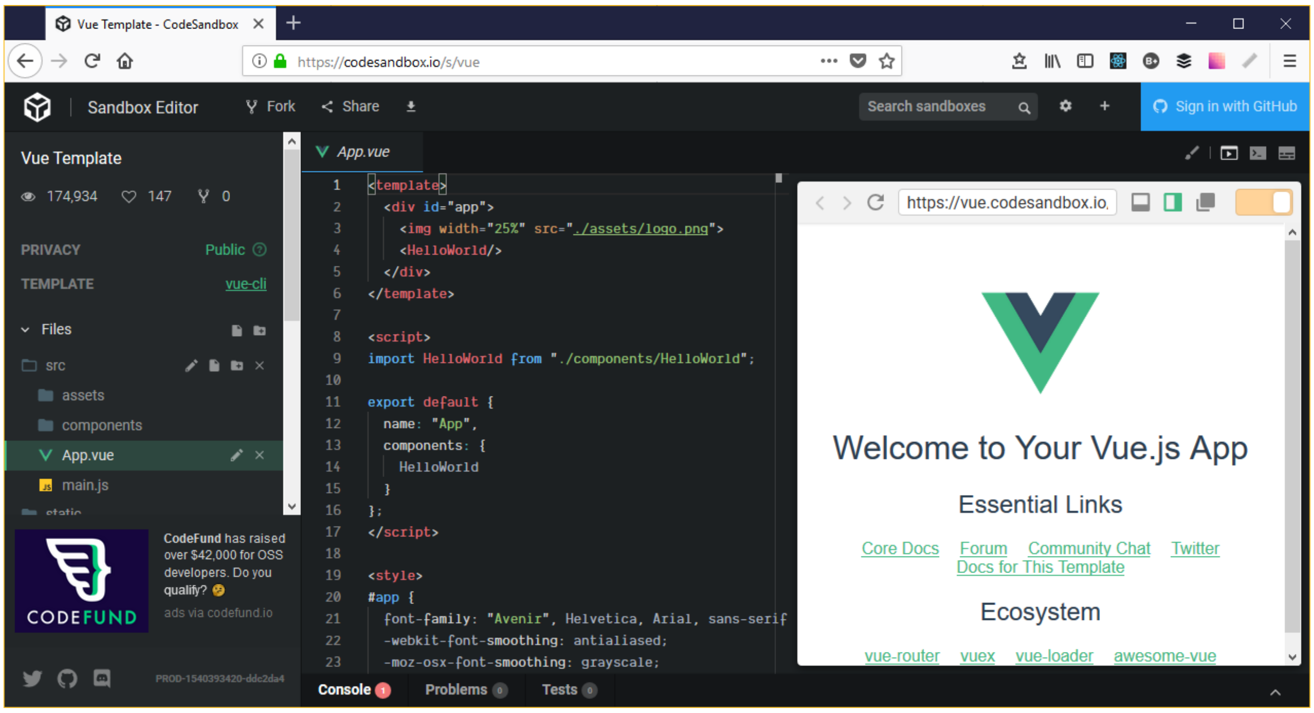
I cannot recommend Code Sandbox highly enough.

If you want to try out some javascript, or explore an entire front-end sample app, Code Sandbox has you covered.

Pro tip: Head on over to https://codesandbox.io/s/ and you'll be prompted to choose from a number of starter templates...



Choose one of the popular frameworks (e.g. Vue) and you can starting hacking with a simple project (without setting up anything on your PC).



```
<template>
  <div id="app">
    <img width="25%" src="./assets/logo.png">
    <HelloWorld/>
  </div>
</template>

<script>
import HelloWorld from "./components/HelloWorld";

export default {
  name: "App",
  components: {
    HelloWorld
  }
};
</script>

<style>
#app {
  font-family: "Avenir", Helvetica, Arial, sans-serif
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
```

What on earth is an IIFEE (and why do I need one?)

You might see something like this on your travels.

```
(function () {

    // some javascript code here

})();
```

For our `index.js` example, this would be...

```
(function () {

    let button = document.getElementById('sayHelloButton');

    button.addEventListener('click', function (e) {
        document.getElementById('greeting').innerHTML = 'Hello ' +
document.getElementById('name').value;
    }, false);

})();
```

Explanation

This is called an *Immediately-Invoked Function Expression*.

Without it, our javascript code would be added to *the global namespace*.

If you have lots of javascript used by your web page and just add it all to the global namespace,that code will break in the case that another script uses the same variable or method names.

This is true of your own code but also any third-party javascript you might include on your page (if they also add anything to the global namespace).

It's also worth noting that Javascript has a garbage collector which will attempt to free up memory for variables no longer in use. Anything added to the global namespace is not eligible for collection until the entire global namespace is "unloaded" (loses scope, typically when the page is unloaded by the browser).

If everything was added to the global namespace you would find your browser consuming much more memory than it probably needs to (if you've ever looked at how much memory your browser is using you'd be forgiven for assuming everything IS being added to the global namespace, but that's another story!).

The IIFE is an anonymous function expression...

```
function(){}
```

But you can't have a function expression sitting there by itself like that (it's not valid Javascript in this form).

If we put it somewhere where Javascript expects a value to exist (inside parenthesis)...

```
(function(){})
```

... this is legit and returns the anonymous function.

We can then invoke that function immediately (the second set of parentheses)...

```
(function(){
    // any code here will run immediately when the browser loads this script
})()
```

So in other words, as soon as this code is loaded by the browser, anything inside our anonymous function will run and (crucially) won't be added to the global namespace.

"use strict";

Worth noting this one.

If you include `"Use strict";` in your javascript files, the browser will be a little more picky about what you can get away with (in terms of syntax etc.) and will generally help you write "safer" javascript (more likely to result in console errors if you try to do something inappropriate!).

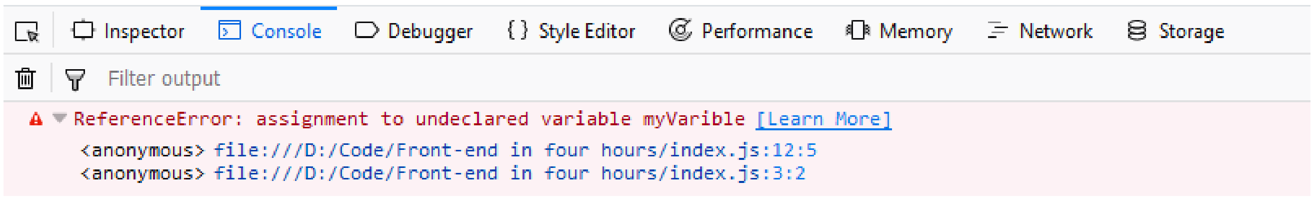
For example, in "non strict" javascript, you can reference a javascript variable which doesn't exist (perhaps because you typed the wrong name by mistake) and a new global variable will be created with that name.

```
var myVariable = 'hello Jon';

// a bit later, in a galaxy far far away...

myVarible = 'hello';
console.log(myVarible);
```

Here we "accidentally" mis-type the variable name when assigning it a new value (and log this new value to the console). Without strict mode this will create a new variable called myVarible.

So now we have two variables (with subtly different names);

With strict mode on, we'll see an error in the console.



```
ReferenceError: assignment to undeclared variable myVarible [Learn More]
    <anonymous> file:///D:/Code/Front-end in four hours/index.js:12:5
    <anonymous> file:///D:/Code/Front-end in four hours/index.js:3:2
```

Older browsers will completely ignore the "use strict"; reference but all newer browsers recognise it and switch to strict mode accordingly.

It is generally recommended to stick with `"Use strict";` unless you have a good reason not too.

Hour 2 - Javascript Today

The first ever version of Javascript was called ECMAScript 1, released in 1997.

As of 2015 the version numbering system changed and versions are now named by year. You may see the 2015 version referred to as ECMAScript2016 or ECMAScript 6, the two are interchangeable.

It's also common for the ECMAScript to be abbreviated to ES.

The most recent versions are...

- ES 2015 (aka ES 6)
- ES 2016
- ES 2017
- ES 2018

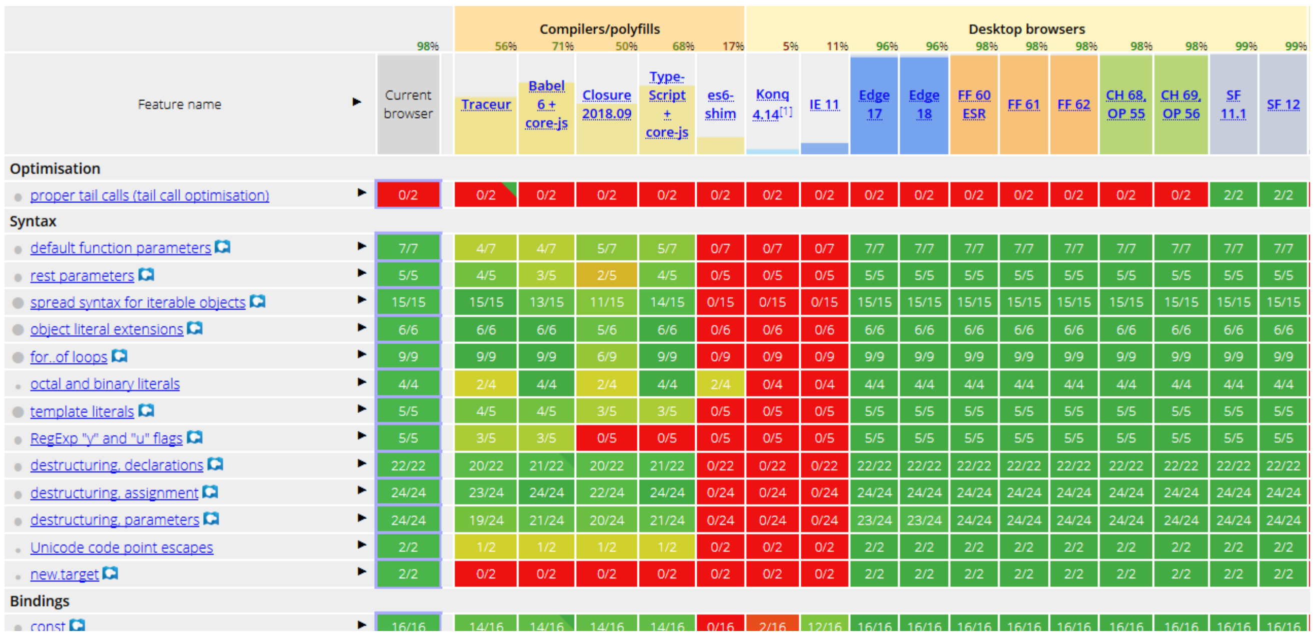
Which Javascript language version should you use?

Each version typically brings new and interesting features but, before you rush out to use the latest and greatest, there's a catch.

Browser support for the different versions varies and (generally speaking) the newer the version, the less support exists in the major browsers.

Here's a handy table showing browser compatibility for ES6.

Here the ES6 (2015) features are listed down the left and the Desktop browser compatibility can be seen under the *Desktop browsers* column.



		Compilers/polyfills					Desktop browsers										
	98%	56%	71%	50%	68%	17%	5%	11%	96%	96%	98%	98%	98%	98%	98%	99%	99%
Feature name ▸	Current browser	Traceur	Babel 6 + core-js	Closure 2018.09	Type-Script + core-js	es6-shim	Kong 4.14[1]	IE 11	Edge 17	Edge 18	FF 60 ESR	FF 61	FF 62	CH 68, OP 55	CH 69, OP 56	SF 11.1	SF 12
Optimisation																	
proper tail calls (tail call optimisation) ▸	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2
Syntax																	
default function parameters ▸	7/7	4/7	4/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7
rest parameters ▸	5/5	4/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
spread syntax for iterable objects ▸	15/15	15/15	13/15	11/15	14/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15
object literal extensions ▸	6/6	6/6	6/6	5/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6
for..of loops ▸	9/9	9/9	9/9	6/9	9/9	0/9	0/9	0/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9
octal and binary literals ▸	4/4	2/4	4/4	2/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
template literals ▸	5/5	4/5	4/5	3/5	3/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
RegExp "y" and "u" flags ▸	5/5	3/5	3/5	0/5	0/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
destructuring, declarations ▸	22/22	20/22	21/22	20/22	21/22	0/22	0/22	0/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22
destructuring, assignment ▸	24/24	23/24	24/24	22/24	24/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24
destructuring, parameters ▸	24/24	19/24	21/24	20/24	21/24	0/24	0/24	0/24	23/24	23/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24
Unicode code point escapes ▸	2/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
new.target ▸	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
Bindings																	
const ▸	16/16	14/16	14/16	14/16	14/16	0/16	2/16	12/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16

Green is good!

The chart makes it pretty clear, unless you need to support Internet Explorer 11, the latest versions of the major browsers are mostly compatible with ES2015.

Happily, all the major browsers are "evergreen" these days, meaning they automatically update to the latest available version. No more debacles like Internet Explorer 6, the browser that would not die!

For newer versions (2016+) [Check out this version](#).

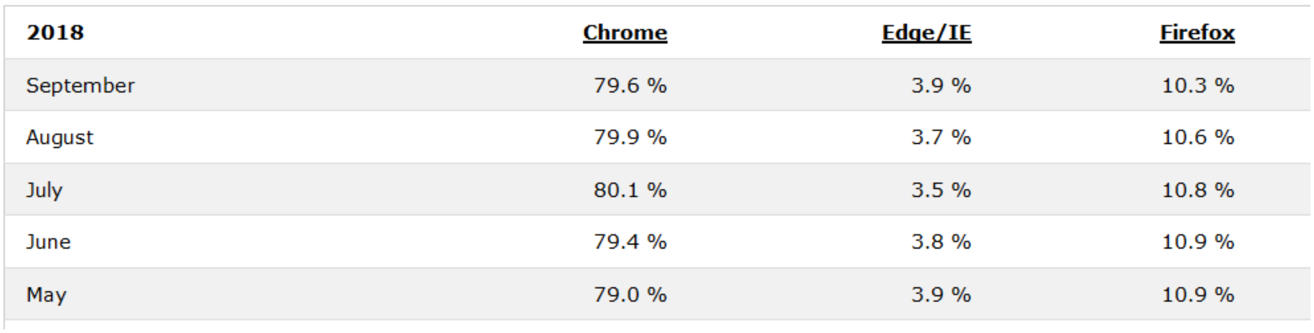
As you might expect, browser compatibility for ES 2016+ is a bit more patchy, and gets worse the closer we get to the present day.

What about Internet Explorer?

One thing's pretty clear from these charts, Internet Explorer has the least comprehensive support for "modern" javascript of all the major browsers.

One handy way to see how much of a problem this is (or indeed isn't) lies in the browser trends reports [available here](#).

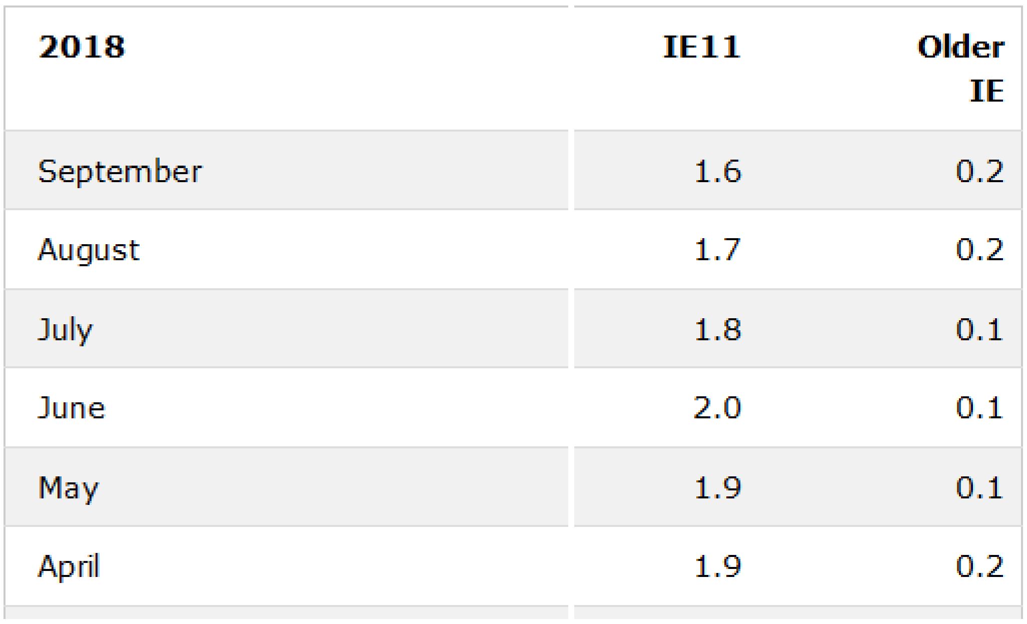
This shows an estimate of the amount of web requests handled by each browser.



2018	Chrome	Edge/IE	Firefox
September	79.6 %	3.9 %	10.3 %
August	79.9 %	3.7 %	10.6 %
July	80.1 %	3.5 %	10.8 %
June	79.4 %	3.8 %	10.9 %
May	79.0 %	3.9 %	10.9 %

This shows just how far behind Edge and Internet Explorer are.

If you drill further and look at just Internet Explorer, the numbers are even lower.



2018	IE11	Older IE
September	1.6	0.2
August	1.7	0.2
July	1.8	0.1
June	2.0	0.1
May	1.9	0.1
April	1.9	0.2

These numbers may be different if you're targeting in-house IT systems in a company dragging its heels over upgrading, but otherwise, it seems Internet Explorer has run its course...

What version should you use?

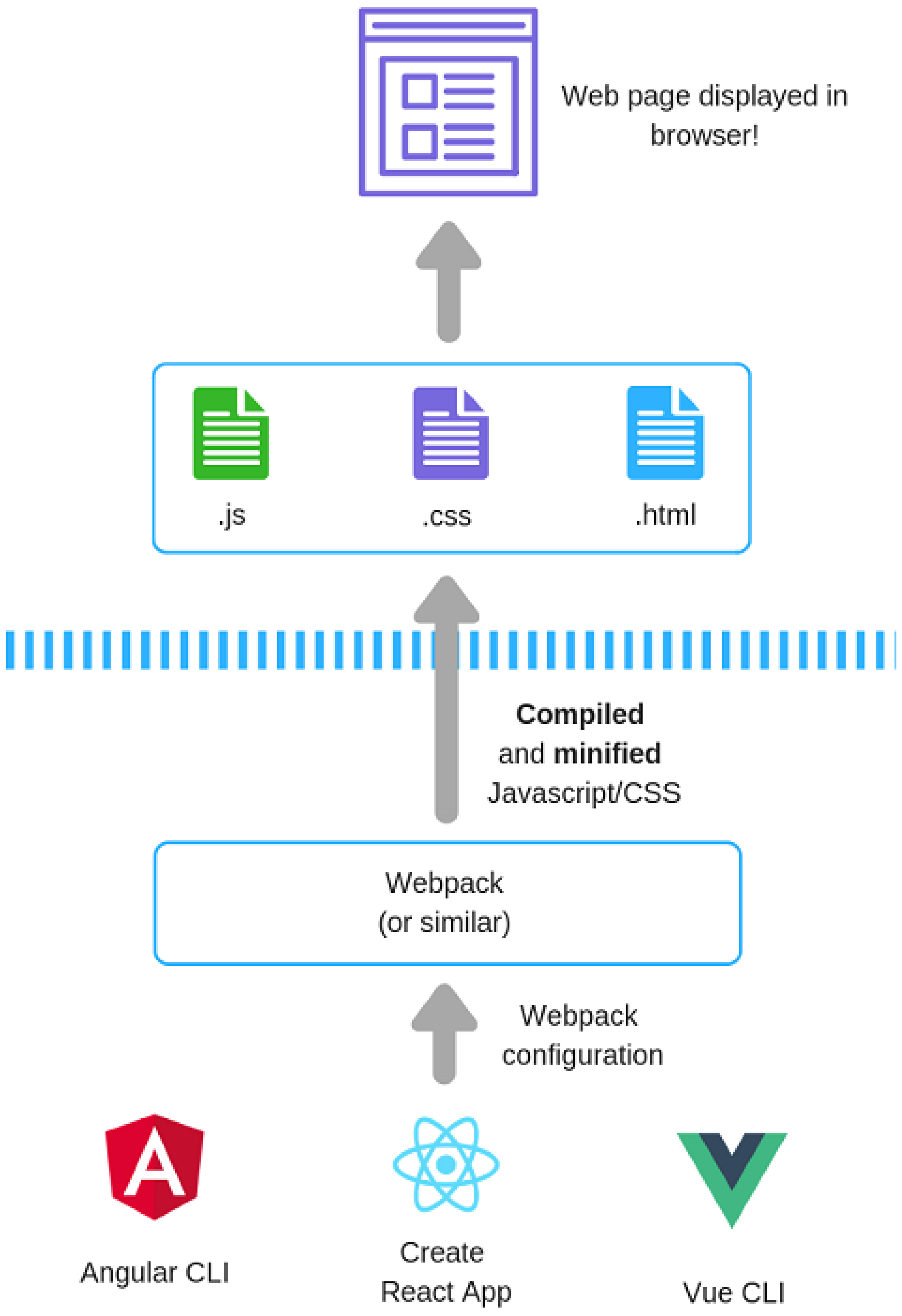
All of this presents an interesting conundrum, should you target the version with best cross-browser support (ES6 at the time of writing) or risk using language features which won't work in your user's browser?

And what if you have users who are stuck on older versions? (perhaps they or their employers have disabled automatic update for some reason).

As it turns out, there is another option, you can use a Javascript compiler, which brings us neatly back to...

The big picture

If you read (or skimmed) Hour 1, you've already seen this....



Hour 1 focused on the top half of this picture (the bit above the dotted blue line).

That's the straightforward part, where the browser takes your HTML, CSS and Javascript and uses it to render a web page in the browser.

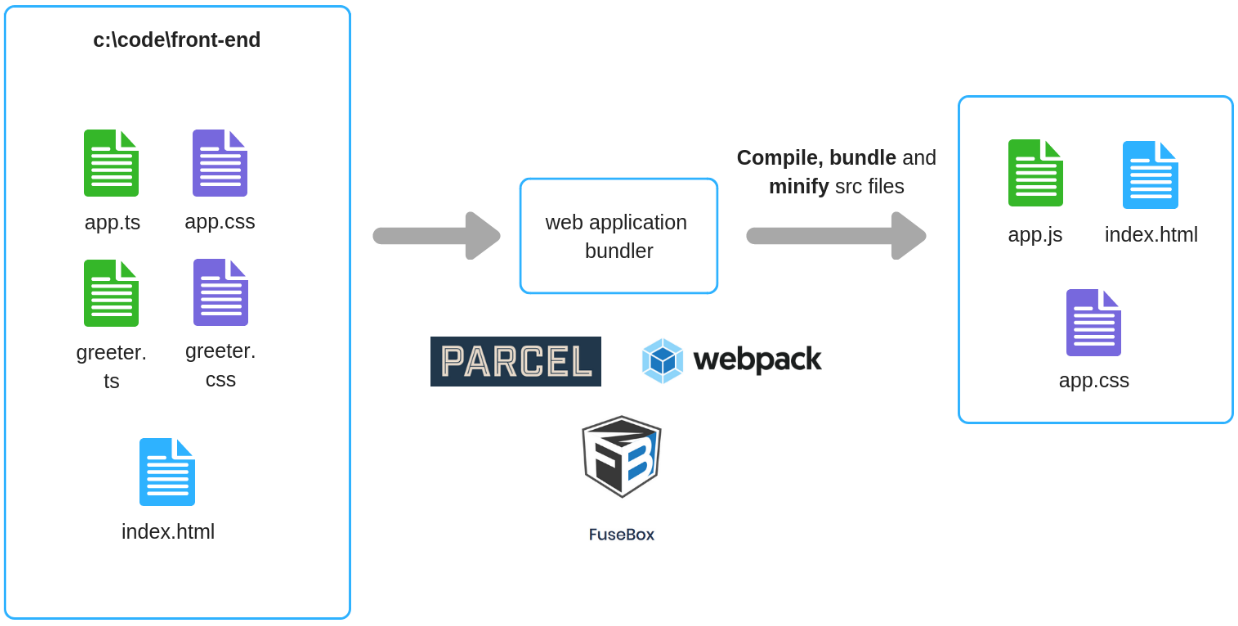
Everything above the blue dotted line is what's deployed to your web server and served to the browser when it requests a page.

The bit below the blue dotted line is where all the complexity creeps in (and is a process which happens on your development machine to turn your raw code into something that runs more efficiently in the browser).

Web Application Bundlers

The front-end ecosystem is definitely not short of options.

There are a lot of tools doing very similar things, all vying for your attention, so let's break this down a bit.



This is how your typical front-end application works today.

You work on the *source files for your application*. These include (but aren't limited to) Typescript, Javascript, CSS and HTML files.

These files may or may not be able to run directly in the browser.

Increasingly, you'll find you're writing Typescript code or Javascript code which is too modern to run on all the browsers you need to support.

So an intermediate step is needed, to take that code (which can't run in the browser) and turn it into something which can.

The tool which fits into this gap is the **web application bundler**.

It's primary job is to do three things to your source code...

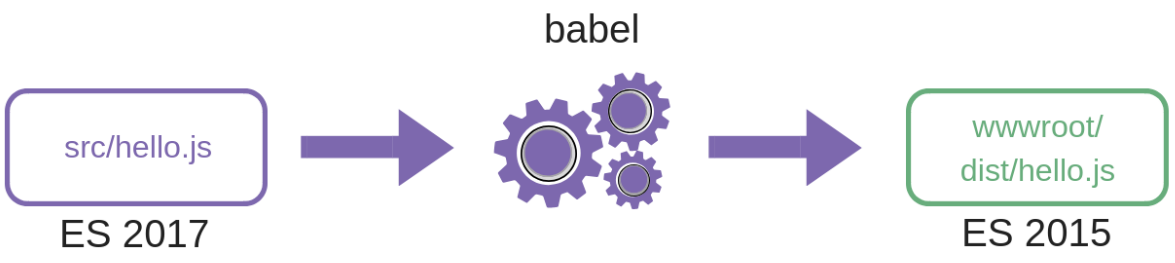
- Trigger *Javascript Compilation*
- *Minify* your javascript/css
- *Bundle* your javascript/css

Javascript Compilation

Let's say you want to use some of those shiny new features in a later version of javascript, but don't want to break your user's experience of your app (in their browser).

You can use a javascript compiler to take the code you write using a newer version and compile it to javascript which only uses an earlier versions' features.

For example, you could write your code using ES 2017 and have a compiler transform it to ES 2015.



One such compiler is called Babel.

You can try it for yourself online here (if you just want to see how it compiles your code).

Make sure es2015 is ticked in the options on the left and try typing this code into the left-hand pane.

```
[1, 2, 3].map((n) => n + 1);
```

This takes an array (with the numbers 1, 2 and 3) and executes a function against each number.

The `arrow` syntax (which was new in ES2015) is a fancy way of writing a function. In the online Babel playground, you should see the right-hand pane update with this version (which will work in ES5).

```
"use strict";

[1, 2, 3].map(function (n) {
  return n + 1;
});
```

Your chosen Web Application Bundler will work with a compiler (e.g. Babel), executing it to compile your code on demand (or when your source code changes etc.).

Minification

These days, we're building bigger and bigger front-end applications which require more and more code.

Whilst Internet connections are pretty fast (on the whole) we still don't want to require that our users download too much code, just to view our web application.

To this end, most javascript and css these days is minified.

This is the process whereby all the extraneous whitespace, new lines, and long variable/function names etc. in your javascript are either removed or shortened, to make the resulting javascript and css files as small as possible.

Bundling

Finally, multiple javascript and css files are often then bundled together.

For example, if you have `index.js`, `about.js` and `listings.js`, these would all be bundled together into one file.

This results in less files for your browser to load, which can improve the load time of your application (by reducing the number of requests the browser has to make to load it).

What Web Application Bundlers are there?

Choices, so many choices.

There are many many tools out there which will handle compilation, minification and bundling for you.

Webpack is probably the best known.

It can be configured to run your Javascript compilers, bundle and minify your code and all sorts of other build time" tasks.

But the problem with webpack is it's quite "configuration heavy" and brings its own learning curve; something you may prefer to avoid if you're just getting started.

Webpack has it's own unique syntax and it's not always intuitive.

Here's an example.

```
module.exports = {
  mode: 'development',
  entry: './foo.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'foo.bundle.js'
  }
};
```

Happily there are other options.

If you spin up a new project with any of the "big 3" javascript frameworks (Angular, Vue, React) you'll find they abstract Webpack away so you can largely ignore it. See Hour 3 for more details.

Alternatively, if you want to roll your own project, but use something less configuration-based, you can try one of the no-config/low-config tools like Parcel (more on that shortly).

Typescript - an alternative

Typescript is a superset of Javascript which brings its own features to the party!

It will check that your code meets certain standards and enforce types.

Types you say?

If you're used to working in other languages, you're most likely familiar with types.

```
public class InterestingClass {
    public void SayHello(){
        string greeting = "Hello!";
    }
}
```

This is C# code and simply indicates that `greeting` is a string.

If you try to do anything with `greeting` that isn't possible for a string, your code won't compile and you'll get an error.

Typescript brings a similar Type system to javascript.

If you've read the section on javascript versions, then you'll be pleased to know that Typescript generally sticks to to those standards, meaning you can use modern javascript features and let Typescript compile them to an older version for you.

Typescript is a superset of Javascript meaning you can mix Typescript and standard javascript code in the same file and the Typescript compiler will figure it out for you.

Try it yourself: Typescript without the headaches (using Parcel)

Pro Tip: Just before you start this bit. It's worth remembering that spinning up new projects in Angular or React.js will handle this setup for you.

But if you want to give the frameworks a miss for now, this is the quickest/simplest way to get a new Typescript project up and running so you can get building your features!

1. Open up a folder in Visual Studio Code, or your editor of choice (this will be the root folder for your test application)
2. Open up a terminal pointing to this folder (Visual Studio Code has its own; `View > Terminal`)
3. Type this command...

```
npm install -g parcel-bundler
```

4. Then this one...

```
npm init -y
```

5. Create a new file in this folder and call it `index.html`
6. Type this code in `index.html`

```
<html>
<body>
    <script src="./index.ts"></script>
</body>
</html>
```

7. Now create a file called `index.ts` and add this code

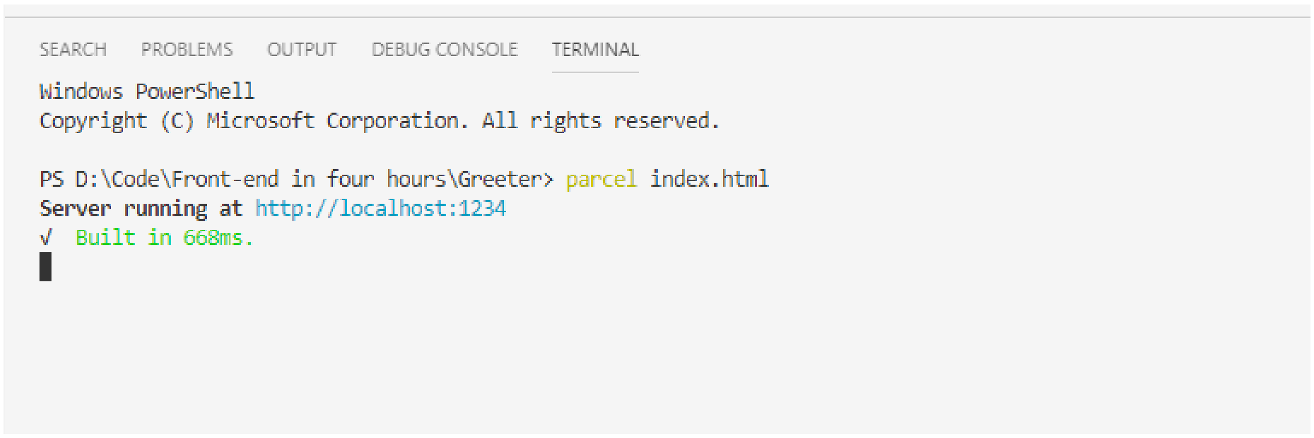
```
console.log('hello world')
```

Check it works

in the terminal, type...

```
parcel index.html
```

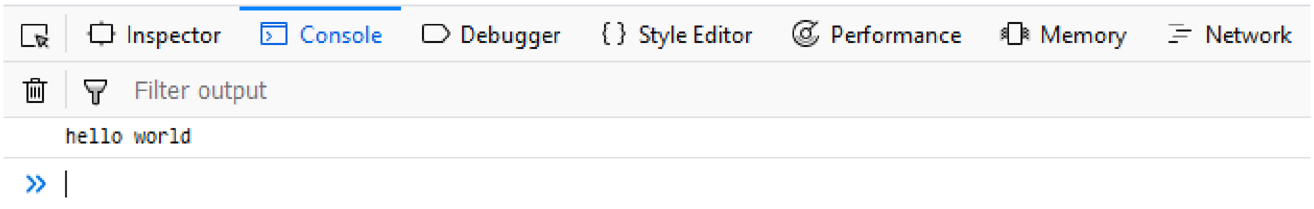
Parcel will run its own in-built development server, download/install Typescript, compile, bundle and minify your typescript files!



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS D:\Code\Front-end in four hours\Greeter> parcel index.html
Server running at http://localhost:1234
√  Built in 668ms.
```

Go to the URL it gives you (in VS Code you can hold down CTRL and click the link to navigate to it).

In the browser bring up the developer tools (usually F12) and check the console to see your app working in all its glory!



```
hello world
```

Explanation

Parcel hails itself as a no-configuration web application bundler.

We used NPM (more on this in a moment) to install it (globally so it's available from anywhere on our machine) then created the default `package.json` file using the `npm init -y` command.

The `-y` argument simply ensures npm uses the default options to create the file. If you omit the `-y` you end up having to answer a lot of questions about your project just to get the barebones package.json file added.

Then we created `index.html` to act as our home page for the application and included a script reference to `index.ts` (which we also created).

This makes `index.html` our home page and `index.ts` our entry point for the javascript part of the application.

When you run Parcel, it sits there watching your application. Every time you change your code it will kick in, compile the Typescript to regular Javascript, bundle everything up and trigger a refresh in the browser.

This is a really handy way to start working on your ideas for a Typescript application; it's low friction and you see your changes immediately.

Bringing in third party libraries (package management)

As we've just seen, you won't get far in front-end development before you run into Node and NPM.

This can be pretty confusing, especially when you're just starting out, so it pays to be clear why you need Node and what NPM is.

Node is javascript running on the server. For the most part, if you're building front-end applications using a javascript framework (more on that next) you need Node.js on your development machine.

> **Not needed in Prod**
>
> Generally speaking, you don't need Node.js on your production server.
>
> Node runs on your development machine and takes on the role of compiling/bundling your front-end assets (Javascript, HTML, CSS).
>
> It's these compiled assets which you will typically deploy to your web server, so virtually any server will do.

When developing a front-end application, you will find you're using Node "under the hood" for things like Typescript compilation and package management.

NPM

NPM is the "Node Package Manager" and provides a mechanism to download other people's javascript code, which you can then use in your own application.

This is a key part of modern front-end development.

It's common practice in front-end applications to bring in components and libraries to speed up development.

After all, why spend days building your own calendar picker if you can NPM one in and stay focused on your features?

If you've written any .NET applications you're probably familiar with NuGet (package manager for .NET), NPM is the front-end equivalent.

Once you have a recent version of node installed, you can download dependencies with commands like this.

```
npm install axios
```

This will look for a package called "axios", create a node_modules folder (in the current path) if it doesn't exist, then download the "axios" package to that folder.

This makes "axios" available within this application (folder).

Installing packages with NPM automatically installs anything else that a package depends on (its dependencies).

You can also **install packages globally**.

```
npm install -g create-react-app
```

This makes "create-react-app" available to run from anywhere on your machine.

If you start using NPM (or spin up a new project using some of the major frameworks) you'll see a **package.json** file appear in your project. This is used by npm and lists any packages which you've installed in your application (not globally).

This is a handy place to look to see what versions of any given package you're using...

```
{
  "dependencies": {
    "underscore": "1.9.1"
  },
  "devDependencies": {
    "typescript": "^3.1.3"
  }
}
```

You'll generally see it contains two sections.

`dependencies` are dependencies which your application depends on to run. So in this case, this example app uses a library called "underscore". When you build and publish your app (for deployment to a server somewhere), dependencies listed here will be included in the published javascript.

`devDependencies` are dependencies only needed during development. So we've no need to ship the typescript compiler with our application. We only need it to compile Typescript to Javascript during development (we then deploy the compiled Javascript as the browser cannot interpret Typescript).

Yarn, an NPM alternative

In recent years a new contender has burst onto the scene!

Now, generally speaking, where you can use NPM you can also use Yarn.

Yarn has some advantages over NPM in the way it caches downloaded packages and uses a special lockfile to guarantee that an install on one machine will work exactly the same way on another.

You can substitute `npm install [package]` with `yarn add [package]`.

If you choose to use Yarn you'll need to install it from here.

It uses the same **package.json** file as NPM, so generally speaking you can use Yarn instead of NPM and everything *should* just work!

Module Resolution (aka referencing javascript code)

Now that our front-end javascript code gets compiled, it raises an interesting new possibility.

Typically we're used to referencing javacsript scripts like this (somewhere in our html).

```
<script src="index.js"></script>
```

This is the standard way of referencing your own scripts plus other people's.

```html
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">
</script>
<script src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.12.1/jquery-ui.min.js">
</script>
```

In this scenario JQuery and JQuery UI will be loaded when you view `index.html` in the browser.

But now we have compilers, bundlers and minifiers, the game has changed!

The modern way to organise your javascript applications is to separate your code into modules, to be exported and then imported where needed, then let the compiler figure it all out.

If you've worked with C# and .NET, this is similar to namespaces.

In C# you might reference another namespace, so you can use the objects exposed therein.

```csharp
using Microsoft.AspNetCore.Http.Authentication;
using Microsoft.AspNetCore.Mvc;
```

We can do this with Javascript using the module system built in to versions of Javascript (from ES6 onwards).

For that to work we need to `export` our functions (or classes), for example...

greetings.ts

```ts
function sayHello(){
    console.log("Hello");
}

export default sayHello;
```

And then import that "module" where we need to use it.

```ts
import sayHello from './greetings'

sayHello();
```

When we include this `import` in a javascript file we're indicating that we want to use the `sayHello` function from `greetings.js` .

Our web application bundler (e.g. Parcel) will attempt to find any dependencies (referenced via `import` statements) as it walks through our application's javascript files.

Once it finds them it will take the code it finds and "do its thing" (typically bundle the code up with all the other dependencies it finds, into one .js file)

This approach means you can simply reference one file in your html and the web application bundler will sort resolve all the modules for you.

Try it yourself: Export/Import modules

Starting with the Typescript project we created a little earlier...

1. Create a file called `greeter.ts` and add this code...

```
export default function sayHello(){
    console.log("Hello from a module");
}
```

2. Now head over to `index.ts` and change it to this...

```
import sayHello from "./greetings";

sayHello();
```

Pro Tip: If you're using VS Code and start typing `sayHello()` (without manually adding the `import`) it will prompt you to auto-import the module. Press `tab` and it will bring the import in for you.

Check it works

Head back over to your browser and check the console to see the new "Hello from a module" message.

Explanation

Parcel has spotted the `import` reference when it started compiling/bundling `index.ts`. From here it went off to find a corresponding .ts file for the module (`greetings`) and located the `sayHello` function.

It then bundled the code for `sayHello` along with the code from `index.ts` together in a .js file in the `dist` folder (in the root of your application).

If you take a look at the `dist` folder's `index.html` you'll notice that Parcel has replaced the link to `index.ts` with a link to the bundled js file e.g.

```
<html>
<body>
    <script src="/Greeter.77de5100.js"></script>
</body>
</html>
```

And you'll find the corresponding js file in the same folder (`Greeter.77de5100.js`) in my case.

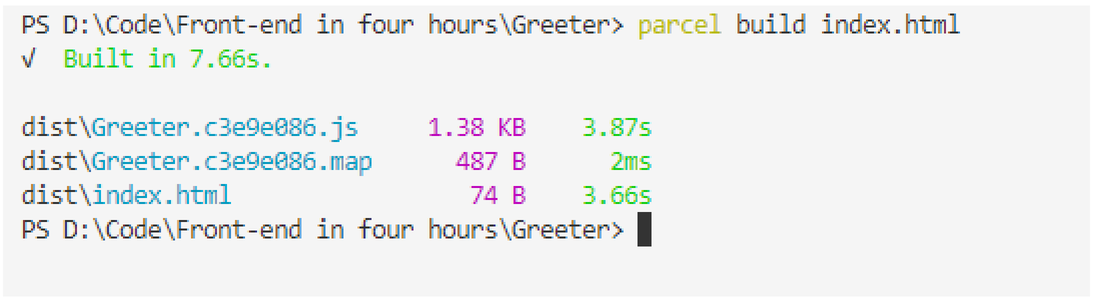
Now if you look at the contents of that .js file you may think there's quite a lot in there for such a small app.

Parcel has added it's own code to handle things like auto-refreshing the browser etc. to make life easier during development.

When you want to kick this app out to production you'll want to produce a leaner (and minified) version of the bundled files, which you can do with this command...

```
parcel build index.html
```

To deploy to production I would now deploy the three files listed here...



```
PS D:\Code\Front-end in four hours\Greeter> parcel build index.html
√  Built in 7.66s.

dist\Greeter.c3e9e086.js     1.38 KB    3.87s
dist\Greeter.c3e9e086.map      487 B      2ms
dist\index.html                 74 B    3.66s
PS D:\Code\Front-end in four hours\Greeter>
```

Jargon Buster

.map files

When you minify javascript code, it becomes largely unreadable...

```
{"./greetings":"TpkG"}]},{},["7QCb"], null)
```

This is because, in order to make the code as lean as possible, the minifier has renamed all your function/variable names etc. to something much shorter.

The problem comes if you attempt to debug an issue in production. If you've ever found yourself staring at minified code in the browser you know how hard it is to decipher.

.map files solve this problem. They are essentially a dictionary for your minified code, which tells the browser what the real (not minified) names are.

This means you can see pretty (and readable) code in the browser's developer tools, even though the application is using the lean, fast, minified version of your code.

Hour 3 - The three big frameworks (a quick start)

What are javascript frameworks?

There are many many javascript frameworks to choose from these days. Common choices being Angular, React.js and Vue.js, but what exactly are they?

Put simply, they are collections of javascript code which you can use when building your own web applications.

They are designed to make common tasks a little easier and all bring their own paradigms and ethos.

Strip all that away though, and they share some common fundamentals.

For example, rather then locating HTML elements and changing their text like this...

```
function sayHello() {
    document.getElementById('greeting').innerHTML = 'Hello ' +
document.getElementById('name').value;
}
```

... all of the frameworks provide a way to "bind" your HTML elements to data, so that if the data changes, the element automatically changes to reflect the new value.

Data binding in Angular

```
<h1>{{title}}</h1>
```

Here, this heading `h1` will contain the current value of `title`.

Where is title? in a corresponding Javascript file.

```
export class AppComponent {
  title = 'Tour of Heroes';
}
```

Whenever our Javascript code updates `title`, the new value will be rendered.

React.js and Vue.js have similar(ish) functionality.

Data binding in React.js

```
<h1>{this.state.title}</h1>
```

Data binding in Vue.js

```
<h1>{{title}</h1>
```

When is a framework not a framework

Pedantic note warning (feel free to skip this but if I didn't mention it...)

Just in case this comes up when you're talking to other developers about React.js ...

Technically, React isn't considered a framework but rather a UI library.

What's the difference?

This largely depends who you talk to, but as far as I can tell the consensus is this...

React is fairly un-opinionated about how you should structure you app, in terms of architecture (how you should arrange your code etc.) and leaves you free to "plug in" different functionality from other javascript libraries (more on that later).

On the other hand, frameworks like Angular tend to come with more "out of the box" and are more opinionated when it comes to how you should architect your application.

So if someone should pull you up on referring to React as a framework, just mumble something about architecture and move on!

Why would you use a javascript framework?

You don't have to use a framework at all (if you don't want to), but here are a few advantages of adopting one.

Features and tooling

All of the major frameworks are designed to make certain tasks that bit easier (for you, the developer).

This includes (but isn't limited to) things like...

- client-side routing
- data-binding
- separation of concerns (of your javascript/html templates)
- consolidating all the front-end "moving parts"

That last one is interesting. The latest versions of each of the major frameworks have made it much easier to get up and running.

You can set up Typescript compilation, bundling and minification etc. for yourself, or you can let the frameworks sort this out for you. Either way, you probably want to go with whichever option gets you to the "building features" part quicker (and helps you stay there).

Job prospects

The major frameworks have the advantage of being common-place across the industry.

If you’re looking to eventually get a front-end (or full stack) job building web applications it can pay to do a little research into the jobs available to you before picking a framework to learn.

One way is to hit one of the online jobs sites (Indeed, Monster etc.) and see what kinds of relevant jobs are listed in your local area.

This will, at the very least, give you a sense of what direction the wind’s blowing.

Why wouldn't you use a javascript framework?

The fact is, these frameworks can bring a lot with them.

If you're just trying to add some basic "interactivity" to your web page(s), you may not need a full-blown framework.

If you suspect this is the case, try "vanilla" javascript (or Typescript) first and see how you get on.

Should you learn javascript first?

Basically, it's a really good idea and needn't take too long.

I thoroughly recommend javascript.info.

Their interactive tutorial will get you up and running in no time.

This is a bit like learning ASP.NET MVC without a firm understanding of C#. Sooner or later you're going to end up writing over-complicated or verbose code (or take a lot longer to achieve certain tasks) if you don't have a firm grasp of the underlying language.

Whilst you'll get quite a long way following tutorials etc. you're ultimately going to be writing a lot of Javascript and it pays dividends to know the basics!

How quickly can you spin up a new project?

This is one area where things are much better than they were even a few months ago.

When Angular 2 first came out, there was a lot of ceremony involved to get anything up and running.

If you didn't understand how front-end build systems worked (using tools like WebPack), you wouldn't get very far.

Now, all three of the major frameworks; Vue.js, React.js and Angular, have simple options, tools or starter projects to abstract some of the complexities away and get you going as quickly a possible.

What is Angular?

Angular has been around for quite a while now. Initially in the form of scripts which you could easily add to your existing pages (commonly known as Angular 1, not retrospectively rebranded AngularJS).

More recently (since version 2) as a framework written in Typescript, which requires you to have a front-end build system up and running to compile, minify and bundle your application (see the big picture chapter earlier in this guide).

To make it easier to spin up new projects (and add features etc.) Angular has it's own Command Line Interface.

So when you create a new project using Angular, the WebPack configuration (for bundling etc) will be sorted out for you, meaning you don't have to learn WebPack's syntax to get started.

Angular includes a lot of things "out of the box", including Dependency Injection, Data Binding, Validation, Templates, Model View Controller etc.

This is both good in that you don't need to go looking elsewhere for core functionality and bad in that you get a lot of potential complexity from day one. Even if you don't need some aspects of the framework, you're likely to be exposed to them anyway.

What is Vue.js?

Vue is newer than Angular (and React.js) and deliberately takes a stance that you don't need all the complexities of a front-end build system just to get started.

Vue.js has its own Command Line Interface for setting up a Vue project and bringing in dependencies etc. but the team behind Vue.js recommend you avoid the CLI when first starting out.

This is no doubt because all of the modern javascript frameworks (including Vue) can get pretty complicated pretty fast when you start bringing in node packages, running NPM commands etc.

Vue.js advocates a simpler approach when you're just getting started: go "old school" and reference the vue.js scripts in your app.

It's considered to have a relatively gentle learning curve, is performant (in terms of how it handles re-rendering parts of the user interface) and lightweight (doesn't come with loads of features/architecture out of the box).

If you out-grow the "add a script reference" approach and want to bring bunding/minification and compilation to your Vue projects, the Vue CLI abstracts the details of this away and gives you some simple commands to get started.

What is React.js

The last of the "big three". React.js is a little like Vue.js in the sense that "core React.js" is pretty lightweight and doesn't require you to adopt any specific architecture to get started.

Like Vue.js, you can just start by adding a reference to the React scripts. Saying that, for a new project you can use "Create React App" which spins up a new application with one command and this is probably the easiest way to get started.

"Create React App" isn't a CLI as such, but does a similar job of abstracting away the chewy details (WebPack) so you can focus on your features.

Where does Typescript fit in?

Angular is written in Typescript and consequently most people using Angular will be writing their applications in Typescript too. This works "out of the box".

Vue.js and React.js don't enforce Typescript. It is possible to use it, but you don't have to.

If you want to use Typescript with React, check out the Typescript version of "Create React App" here.

For Vue.js, the CLI has out-of-the-box support for Typescript.

Let's run through spinning up a new project in each of the "big three".

Try it yourself: Spin up a new Vue.js application (via a script reference)

1. Create an `index.html` file in a folder on your machine
2. Reference the Vue.js dependency in `<head></head>`

```
<!-- development version, includes helpful console warnings -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

1. Then add some html to the `<body></body>`

```
<div id="app">
    Hello {{ name }}
</div>
```

Open `index.html` in a browser at this point and you'll see "Hello {{ name }}".

So how can we change `{{ name }}` to an actual name? We need a little javascript!

1. Add a script element underneath the `div`, like this

```
<body>
    <div id="app">
        Hello {{name}}
    </div>

    <script>
        var app = new Vue({
            el: '#app',
            data: {
                name: 'Jon'
            }
        })
    </script>

</body>
```

Now run it and you should see "Hello Jon" (or whichever name you used in the script!).

This javascript code has created a Vue instance, the `app` variable.

We've then told Vue to link this Vue instance to our HTML div element.

When we set up the Vue instance we can declare "data". Anything specified in `data` is added to something called the Vue "reactivity system".

This means that any changes to the data (`name` in our case) will trigger a re-render of the associated html.

Try it yourself: Spin up a new Angular application

Unlike Vue.js, there's no easy way to just add Angular to your existing application, you really need to start with a brand new Angular project from the get go.

Angular has its own Command Line Interface (CLI) to handle the process of creating new projects, adding new features etc.

To use it, you'll need to install it via NPM.

1. Make sure you have a recent version of Node.js installed
2. Open up a command prompt/terminal
3. Install the Angular CLI

    ```
    npm install -g @angular/cli
    ```

This installs the Angular Command Line Interface on your machine.

`-g` means install this package globally, for use from anywhere on your machine.

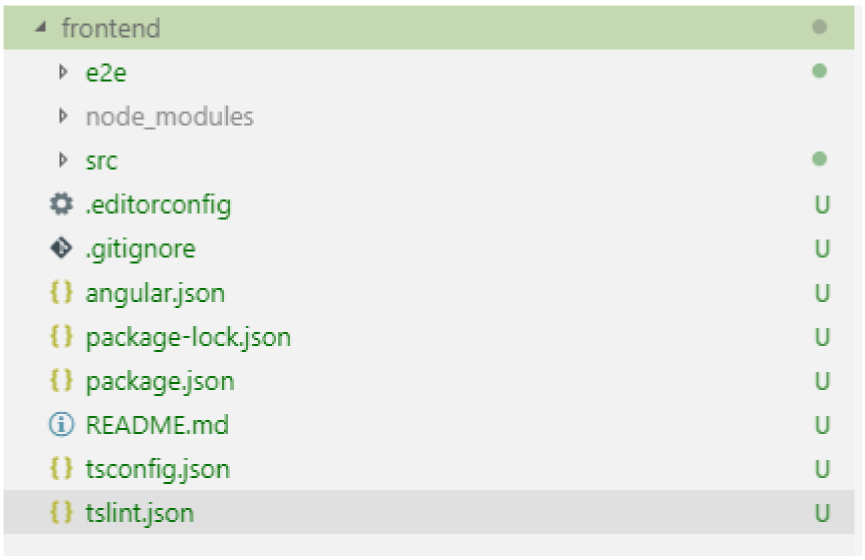
Without `-g`, the package would be downloaded and added to a `node_modules` folder in the current path, making it available only in this folder/application.

1. Now you can use `ng` to perform Angular tasks, start by creating a new project (called frontend)

```
ng new frontend
```

Angular will now run off and grab the relevant packages, create your project files etc. This can take a while!

When it's finished you should have something a bit like this...



```
frontend
  e2e
  node_modules
  src
  .editorconfig        U
  .gitignore           U
  angular.json         U
  package-lock.json    U
  package.json         U
  README.md            U
  tsconfig.json        U
  tslint.json          U
```

1. Launch the app

```
cd frontend
ng serve --open
```

The application will open up in your default browser.

As you make changes to the application, it will automatically rebuild and refresh in the browser.

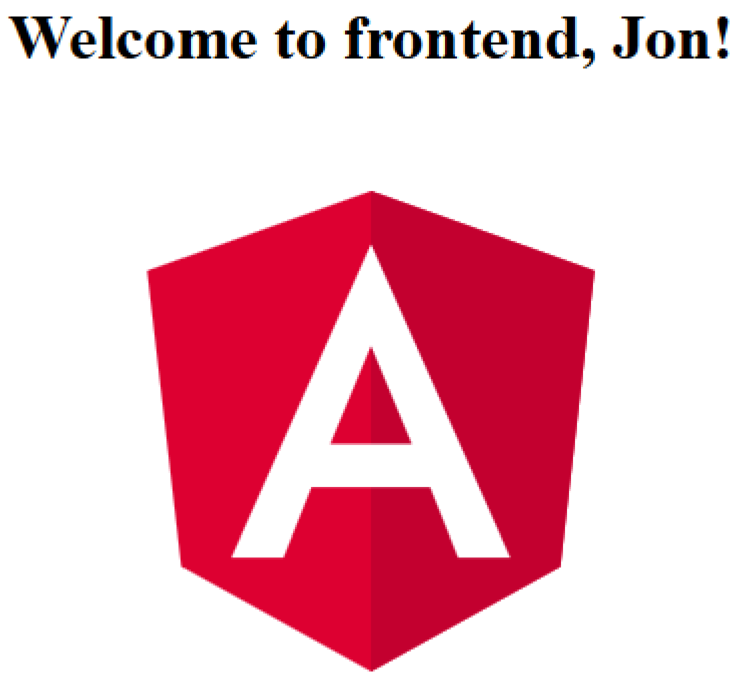
Test this out for yourself...

1. Locate the **src\app** folder, open the file called `app.component.html`
2. Make a change to the html, for example...

```
<h1>
  Welcome to {{ title }}, Jon!
</h1>
```

Here I simply added the hardcoded text `, Jon` inside the `<h1>` element.

3. The application, running in your browser, should automatically change to reflect your changes.



Welcome to frontend, Jon!

A quick tour of the default Angular app

So what have we got here?

package.json

This is where you specify what packages your application needs, so that NPM (the Node Package Manager) can download them.

For more, check out the Package Management chapter.

tsconfig.json

Angular apps are written in Typescript by default, so this is here to configure Typescript.

We saw a much simpler version of this in the chapter on Typescript.

If you want to dig in to what all these options mean, check out the official docs. On the other hand, when you're just getting started you can pretty much leave this file as is and get on with building your features!

angular.json

This is where you can configure the Angular CLI. You can configure things like which files are included when you build your project.

You may also see this referred to as the Angular Workspace file.

Again, you can leave this alone when you're just getting started.

src

This is the crux of your new Angular application.

It's in here that you'll write your own code (create components etc.)

Try it yourself: Spin up a new React.js application

Unlike Angular, React doesn't have it's own CLI.

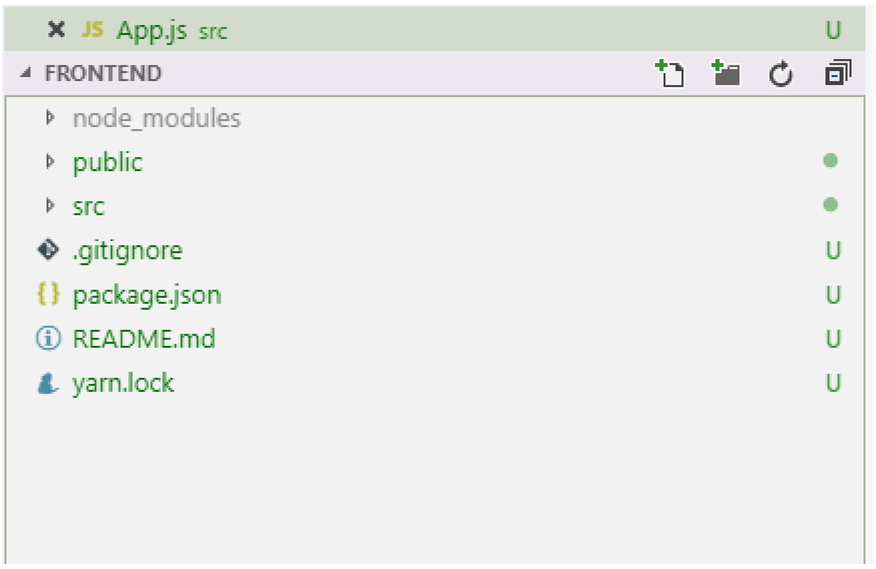
It does have something called "Create React App" which is a simple way to spin up a new React project.

1. As long as you have a recent version of Node.js installed, spinning up a new React App is as simple as typing this command...

```
npx create-react-app frontend
```

(where `frontend` is the name of the application you want to create).

You should end up with an application which looks like this...



1. Now you can open up the `frontend` folder and start the application with these commands.

```
cd frontend
npm start
```

This will compile and launch the app in your default browser.

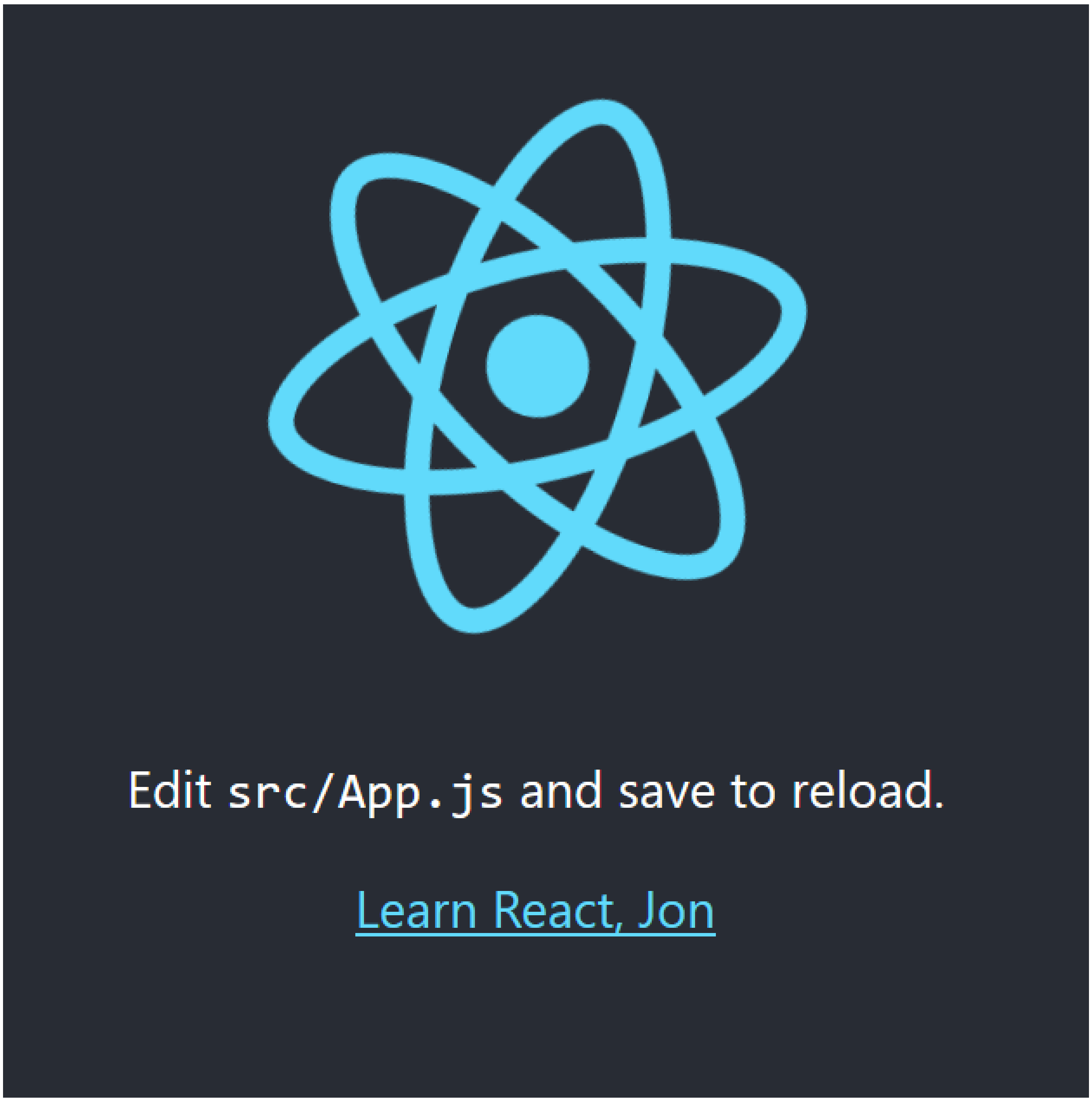
As you make changes to the app it will automatically rebuild and refresh in the browser.

1. Try changing `src\App.js`, for example...

```
<a
className="App-link"
href="https://reactjs.org"
target="_blank"
rel="noopener noreferrer"
>
    Learn React, Jon
</a>
```

Here I added the text `, Jon` after the default `Learn React`.

Save `app.js` and the browser will update to reflect your changes.


Edit `src/App.js` and save to reload.

Learn React, Jon

Jargon buster

npx

Not a typo, npx is a convenient way to launch executables found in Node packages.

When you invoke `npx create-react-app`, Node will look for the `create-react-app` command in the local `node_modules` folders, then it will look in the global directory (on your machine, in case the relevant package has been installed globally) before finally downloading the relevant package if it still can't find the command.

This is a handy way to download and use the `create-react-app` command without having to manually download or install anything.

npm start

As we've already seen, NPM is the Node Package Manager.

As well as using it to install dependencies, you can also use it to run scripts.

If you look at `package.json` for the React app, you'll see a few scripts referenced.

```
"scripts": {
   "start": "react-scripts start",
   "build": "react-scripts build",
   "test": "react-scripts test",
   "eject": "react-scripts eject"
 },
```

These invoke "react-scripts" and pass certain arguments (like `start`, to compile and launch the app).

You could just type `react-scripts start` to start the app, but `npm start` will do the same thing and is slightly less verbose.

A quick tour of the default React.js app

package.json

This is where you specify what packages your application needs, so that NPM (the Node Package Manager) can download them.

For more, check out the Package Management chapter.

README.md

This is a pretty comprehensive guide to "Create React App" and will tell you how to do all manner of useful things.

public

This is where the starting `index.html` page for your React app goes, along with any other standard things like `favicon.ico` and anything you just want to be available via the browser (you could put static html pages, images, css files etc. here).

src

This is where your React app source code goes.

You'll see you already start with an index.js, which is the first thing that will be executed when this application runs.

This in turn renders the component found in `App.js` (via this line of code).

```
ReactDOM.render(<App />, document.getElementById('root'));
```

Next Steps

So what now?

Well the next (and final) section explores how we might create our front-end features (generally, and with specific examples for each of the frameworks).

But beyond that, this is a perfect time to start throwing code around (in whichever framework takes your fancy) and start working out what you don't know, so you know what questions to throw at Google!

Hour 4 - Build your features

Now we know about the basic building blocks (HTML, Javascript and CSS), a little about Node and package managers and we've seen the easiest way to spin up a new application in each of the three major front-end frameworks.

What's left?

Well you need to build your features presumably :-)

And this is where it can feel like the training wheels just fell off; you need to build something and really aren't sure how to do it, or where to begin.

This might be the first time you've tried to use Angular (or Vue.js, or React.js), your HTML and CSS might be sketchy at best (welcome to the club) but if you can focus in on a tiny feature, you can figure out how to build it.

It takes trial and error, patience and a healthy dose of curiosity.

Keep it simple and focused

When you think about building a front-end feature, the number one tip is to keep it small, simple and as focused as possible (especially when you're just starting out).

Nearly every time I think I've got a nice small feature, it turns out to be bigger than I thought (look at this very "pocket guide" for example).

If you think you need a big form, start with one textbox and a button to submit the data.

If you think you need a whizzy menu with links which expand when you open them, start with simple hyperlinks.

If you want to show someone's information, including avatars and links to their Twitter profile, start with some read-only basic info about them (the avatar and Twitter profile links can come next).

When you think it's small enough, make it smaller.

Iterate quickly with Mockups

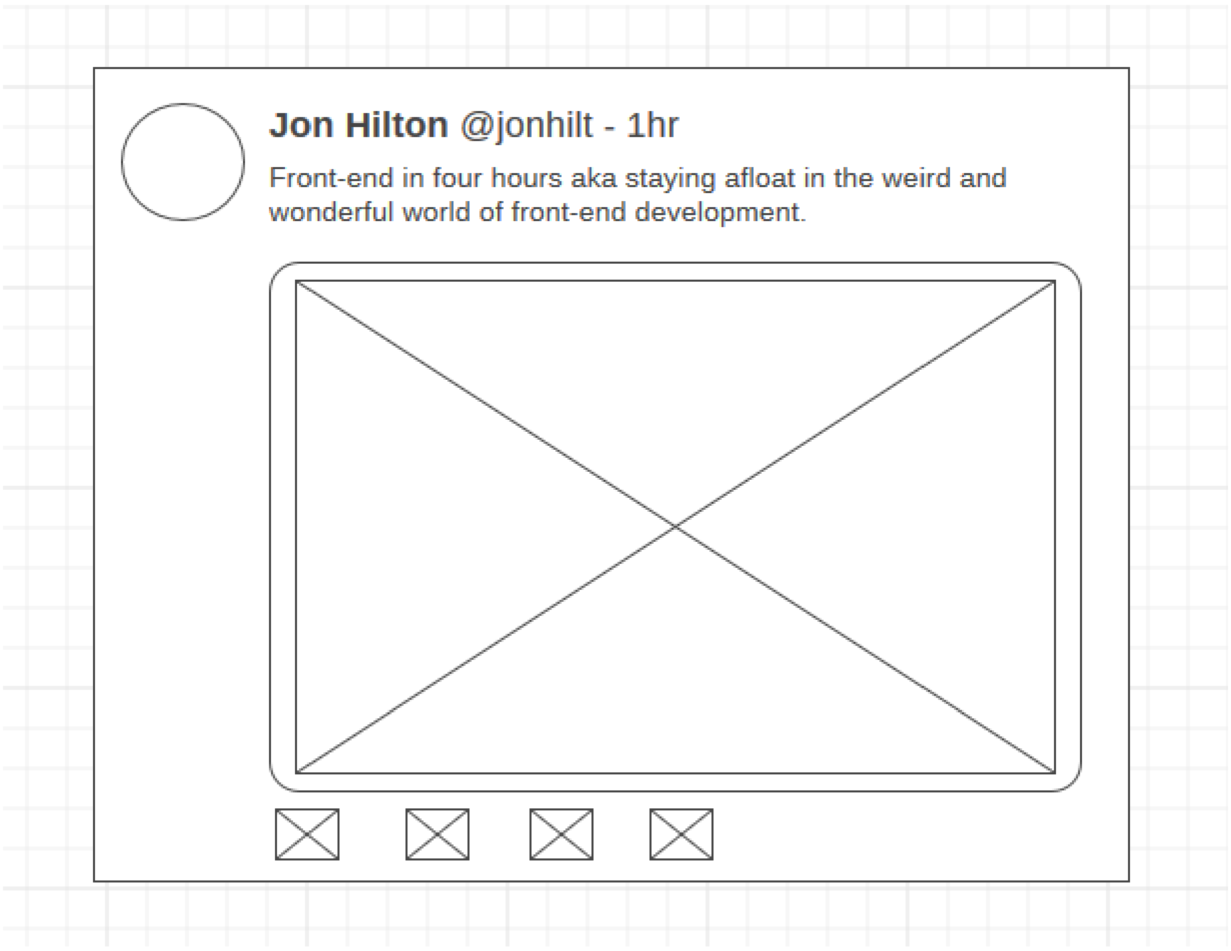
I'm a big fan of clarifying feature ideas with the help of mockups.

You can go old school (use a whiteboard, or piece of paper) or use one of the many online mocking tools.

Whichever you choose, visualising what you're about to build is a great way of getting clarity on what you do (and don't) need to build.

Let's take a really simple example, creating a Twitter style "card".

Here's a mockup quickly put together using wireframe.cc.


Jon Hilton @jonhilt - 1hr
Front-end in four hours aka staying afloat in the weird and wonderful world of front-end development.

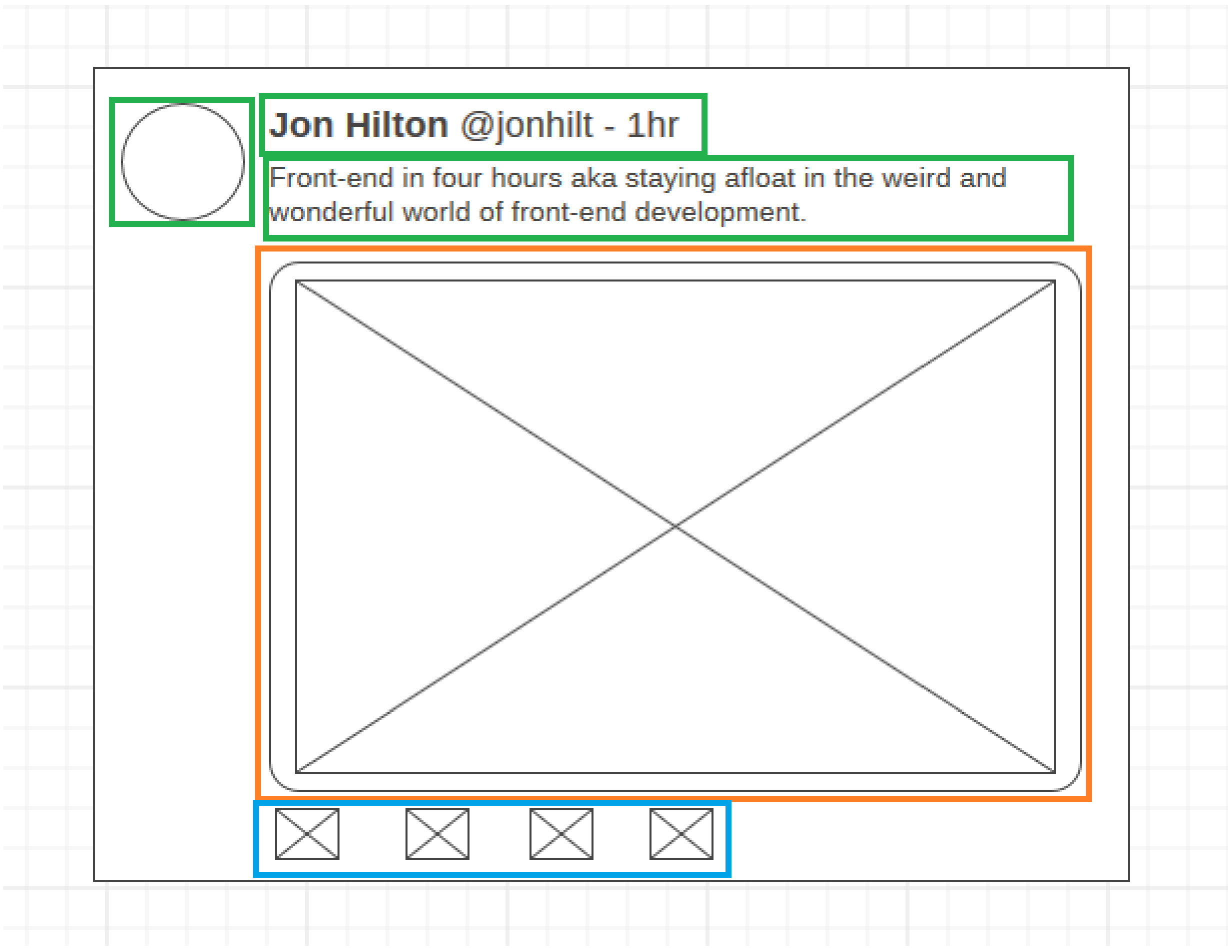
This is pretty rough and ready but in many ways that's the point.

It's much easier to throw ideas around at this stage than when this has transformed into HTML, CSS and Javascript.

Break a big feature down into components

Whichever of the frameworks you're using, it's a good idea to break your user interface down into components.

In this case, the Twitter card itself could be considered a component, but we can make other parts of it "sub components" too.


Jon Hilton @jonhilt - 1hr
Front-end in four hours aka staying afloat in the weird and wonderful world of front-end development.

How big you want your components to be is ultimately a judgement call. Even now we could go further and have each action (those boxes at the bottom, which would be retweet, like etc.) be its own component.

React is particularly well suited to build your interface in this componentised way, but Angular and Vue are equally capable.

The great part of breaking this down is that the "build a Twitter clone" requirements has just morphed into "figure out how to show someone's name and their twitter handle".

Now we have a much less daunting task to go at whilst still moving towards the bigger feature we're trying to build.

Use hard-coded data (HTML)

You might be tempted to start thinking about databases, calls to the back-end etc. to get a feature like this built.

However, it's much easier to start with hard-coded data.

And hard-coded can even mean static HTML.

A really simple version of the "user's name and twitter handle" component could just be...

```
<div>
    <strong>Jon Hilton</strong> <span>@jonhilt</span>
</div>
```

OK, so this is pretty uninspiring, but we have the first basic version of our (part of) our Twitter card and all it took was some HTML.

From here we can make this into a *React.js* component.

```
class Handle extends Component {
  render() {
    return (
      <div>
        <strong>Jon Hilton</strong> <span>@jonhilt</span>
      </div>
    );
  }
}
```

> Incidentally, this shows a key aspect of React.
>
> In React, a component includes the javascript (logic) and "HTML" in the same file.
>
> I say "HTML" because technically this is actually something called **JSX**. It looks like HTML but is really Javascript in disguise. It will render the HTML you expect it to, but can be interwoven with Javascript expressions too.

An equivalent *Angular* component might look like this...

handle.html

```
<div>
  <strong>Jon Hilton</strong> <span>@jonhilt</span>
</div>
```

handle.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'handle',
  templateUrl: './handle.component.html'
})
export class HandleComponent {
}
```

And finally, a simple *Vue.js* implementation...

```
Vue.component('handle', {
template: ` <div>
    <strong>Jon Hilton</strong> <span>@jonhilt</span>
</div>`
})
```

> **Funny looking apostrophe?!**
>
> That "not quite apostrophe" is actually a "backtick" and represents a "template literal" in Javascript.
>
> You can think of it as an alternative way of representing strings in Javascript.
>
> It brings two key advantages.
>
> One, you can split a string inside backticks onto multiple lines (to make it more legible, as in this example).
>
> Secondly, you can interpolate Javascript expressions inside the string
>
> For example...
>
> ```
> let name = 'jon';
> `Hello ${name}`
> ```
>
> Which results in the string "Hello jon".

Use hard-coded data in your Javascript

Now, unless we want every Twitter card to be from Jon Hilton (unlikely), we need to drive it from some data.

But before we go running off to the server we can simply move the hard-coded data down one level (from HTML, to Javascript).

Each of the frameworks has some form of "data binding" for this very purpose.

React.js

```
class Handle extends Component {

  state = { name: 'Jon Hilton', handle: '@jonhilt'}

  render() {
    return (
      <div>
        <strong>{this.state.name}</strong> <span>{this.state.handle}</span>
      </div>
    );
  }
}
```

React leans on "state".

Every component can have state which you can then reference in your template. If the state changes, the component automatically re-renders accordingly.

Angular

handle.component.html

```
<div>
    <strong>{{user.name}}</strong> <span>{{user.handle}}</span>
</div>
```

handle.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'handle',
  templateUrl: './handle.component.html'
})
export class HandleComponent {
    user = { name: 'Jon Hilton', handle: '@jonhilt' }
}
```

Angular lets you bind to any property in your component. As with React, a change to the `user` property here will be updated in the rendered HTML.

Vue.js

```
Vue.component('handle', {
    data: function () {
        return { name: 'Jon Hilton', handle: '@jonhilt' }
    },
    template: ` <div>
<strong>{{name}}</strong> <span>{{handle}}</span>
</div>`
})
```

And finally, Vue similarly has this mechanism for binding HTML templates to data.

Iterate while it's easy

The best time to "thrash" with your ideas is when you can easily change a mockup. It's quick and cheap before a line of code is committed to source!

After that, it's still quite easy to play around with plain old HTML.

Once Javascript kicks in, things get a little trickier but not too bad if you can break your feature down into components.

In fact, this is a good reason to break your features down into component. They are easier to tweak than big features, with all the code in one file!

By definition components are generally smaller, with less code and less easy to break by mistake!

Making AJAX Requests

A key part of building any front-end application is when the rubber hits the road and you have to interact with a back-end application.

Most browsers now support something called `Fetch` (to a greater or lesser degree).

So a simple call to a server might look like this.

```
fetch('https://localhost:44348/api/user')
```

This will default to an HTTP Get.

To handle the resulting response, you would need to use something called a "Promise".

Javascript employs promises to handle situations where you're waiting for something to happen.

So in this case we need to wait for the server to response.

A Javascript promise means we can pass a function to be invoked when the response comes back.

```
fetch('https://localhost:44348/api/user').then(response => {
    // handle the response
});
```

Less code with async

The problem with promises, is that callback chaining code can get very messy very quickly.

Recent versions of Javascript (and Typescript) introduced `async` to make this a little easier.

Now, instead of the `.then` syntax, you can do this.

```
const result = await fetch('https://localhost:44348/api/user');
```

This is super handy because now, when we run this code, we'll pause at the await line, and when the response comes back from the server, assign the result to `result`.

For this to work, the containing function needs to be declared `async`.

```
public async componentDidMount() {
    const result = await fetch('https://localhost:44348/api/user');
    const users = await result.json();
    this.setState({ users });
}
```

This is an example using React but it applies more generally. Without that `async` keyword in the `componentDidMount` declaration, we wouldn't be able to use `await`.

When it comes to making POST requests, the Fetch api works thus.

```
fetch('https://localhost:44348/api/user', {
    method: 'POST',
    headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({name: 'Jon'})
})
```

A little verbose (an alternative follows) but here we indicate the method is `POST`. We must also indicate what the posted content type is (so JSON here) and then turn the data we're posting into a JSON string using `JSON.strinfigy`.

Fetch alternative (Axios)

Now whilst the browsers generally support `Fetch` (check compatibility here), there are alternative libraries which offer more features/cleaner syntax.

Axios is one of these.

You can bring axios in via NPM (no surprise!)...

```
npm install axios
```

The axios equivalent of our `get` function is...

```
import Axios from "axios";

Axios.get('https://localhost:44348/api/user').then(response=>{
    console.log(response);
});
```

And the async version...

```
import Axios from "axios";

async function doSomething(){
    const response = await Axios.get('https://localhost:44348/api/user');
}
```

The POST with Axios is much cleaner than the `fetch` equivalent.

```
Axios.post('https://localhost:44348/api/user', { name: 'Jon' });
```

Ah that's better!

Axios assumes you're posting JSON by default (which is pretty likely) and doesn't require any extra configuration in this scenario.

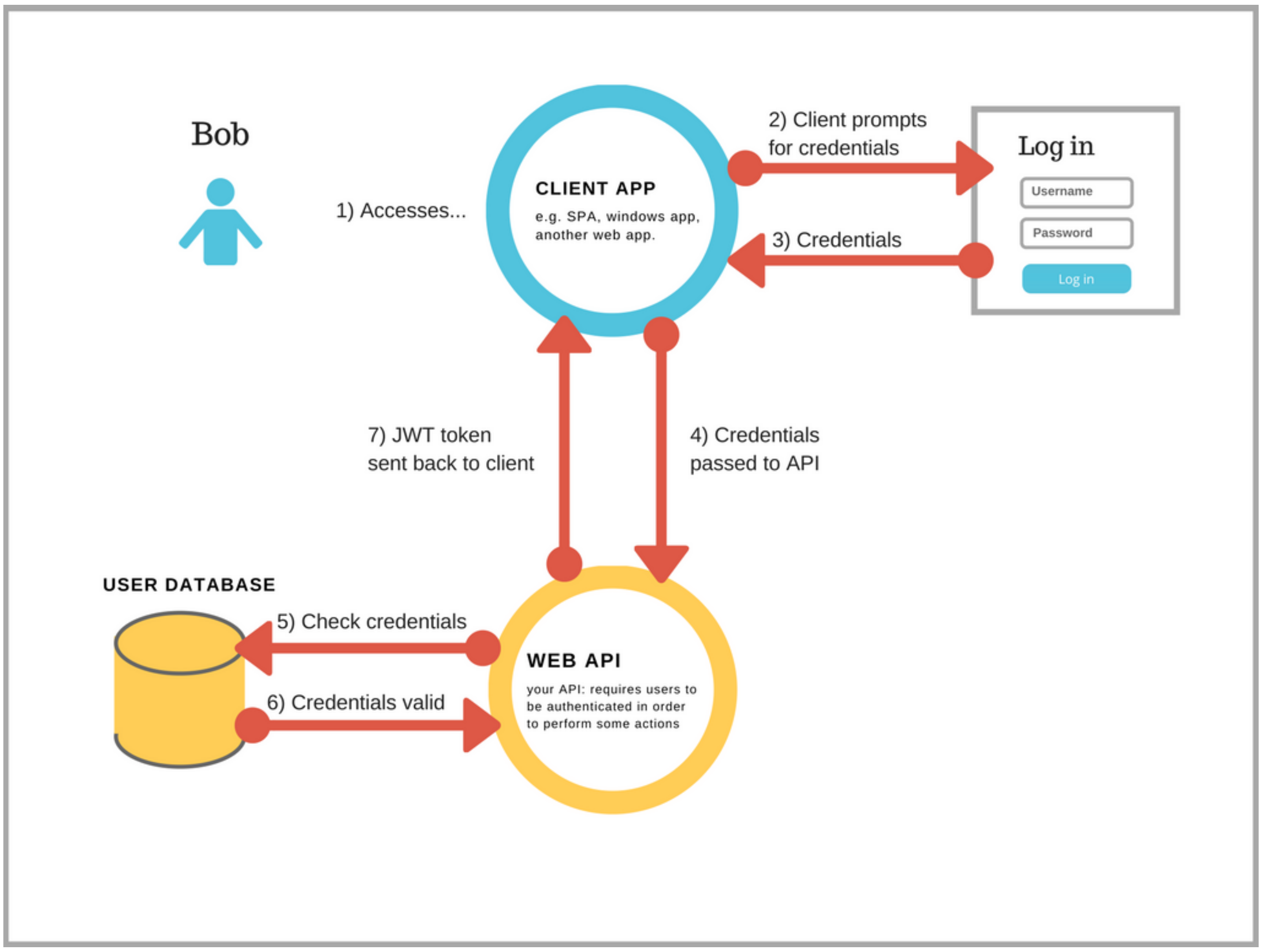
A word on authentication

As subjects go, this is a biggy and beyond the scope of this pocket guide.

But one thing is worth noting.

The "typical" setup for auth between front-end and back-end is using something called JWTs (JSON Web Tokens).

Here's an example.


Bob
1) Accesses...
CLIENT APP
e.g. SPA, windows app, another web app.
2) Client prompts for credentials
Log in
Username
Password
Log in
3) Credentials
7) JWT token sent back to client
4) Credentials passed to API
USER DATABASE
5) Check credentials
6) Credentials valid
WEB API
your API: requires users to be authenticated in order to perform some actions

Assuming you have set up your web-api to issue tokens ([here's a handy series on that]) then you'll need a way to pass that token in every request from your front-end application to the back-end.

This token becomes the all important key.

If a client (e.g. your application) makes a request and includes the token, then the back-end server can validate the token and return the requested data.

If no token is sent (or the token fails validation) the back-end server can refuse the request (401 Unauthorized).

To pass the token via fetch you'll need to set the relevant header like this...

```
fetch(`${baseUrl}/${url}`, {
    body: JSON.stringify(data),
    headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': `bearer ${Auth.getToken()}`
    },
    method: 'post'
});
```

That `Authorization` header is the key and must be in the form `bearer <token-goes-here>`.

With Axios we have a similar solution.

```
Axios.post('https://localhost:44348/api/user', { name: 'Jon' },
    { headers: { 'Authorization': `bearer ${Auth.getToken()}` } }
);
```

Go forth and build your features

Nnow you have the "big picture" of front-end development (and some handy tips on getting going with each of the frameworks, or just Parcel to keep things simple), what next?

Well if you haven't already, this would be a really good time to try and build your first few front-end components.

Here's a process to help you get going.

- If you're not sure which framework you want to try, pick the one which most appeals to you for now (you can change your mind later, I won't tell anyone!)
- Come up with an idea for a feature
- Strip away as much complexity as possible (use mock-ups to iterate on your ideas) until you have a tiny feature you can build (displaying some simple data is a good starting point)
- Spin up a new project using your chosen framework (see earlier chapters for the quickest option)
- Have a go at building that first component

If you get stuck, the official docs for all of the frameworks are pretty good these days.

If you hit a term you're not sure about, first check if it's in this guide.

If not, and you can't figure it out, feel free to drop me an email and I'll do my best to help you find answers!

Until then, happy developing :-)