References

Table of Contents

Session 1: Introduction to JavaScript

How to use JavaScript Classes?

Source	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes
Date of Retrieval	02.03.2019

JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

Defining classes

Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

Class declarations

One way to define a class is using a **class declaration**. To declare a class, you use the classkeyword with the name of the class ("Rectangle" here).

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Hoisting

An important difference between **function declarations** and **class declarations** is that function declarations are hoisted and class declarations are not. You first need to declare your class and then access it, otherwise code like the following will throw a ReferenceError:

```
const p = new Rectangle(); // ReferenceError

class Rectangle {}
```

Class expressions

A class expression is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body. (it can be retrieved through the class's (not an instance's) name property, though)

```
// unnamed
•       let Rectangle = class {
•         constructor(height, width) {
•           this.height = height;
•           this.width = width;
•         }
•       };
•       console.log(Rectangle.name);
•       // output: "Rectangle"
•
•       // named
•       let Rectangle = class Rectangle2 {
•         constructor(height, width) {
•           this.height = height;
•           this.width = width;
•         }
•       };
•       console.log(Rectangle.name);
•       // output: "Rectangle2"
```

***Note:** Class **expressions** are subject to the same hoisting restrictions as described in the* Class declarations *section.*

Class body and method definitions

The body of a class is the part that is in curly brackets {}. This is where you define class members, such as methods or constructor.

Strict modeSection

The body of a class is executed in strict mode, i.e., code written here is subject to stricter syntax for increased performance, some otherwise silent errors will be thrown, and certain keywords are reserved for future versions of ECMAScript.

ConstructorSection

The constructor method is a special method for creating and initializing an object created with a class. There can only be one special method with the name "constructor" in a class. A SyntaxError will be thrown if the class contains more than one occurrence of a constructor method.

A constructor can use the super keyword to call the constructor of the super class.

Prototype methods

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area() {
    return this.calcArea();
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}

const square = new Rectangle(10, 10);

console.log(square.area); // 100
```

Static methods

The static keyword defines a static method for a class. Static methods are called without instantiating their class and cannot be called through a class instance. Static methods are often used to create utility functions for an application.

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;

    return Math.hypot(dx, dy);
  }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);

console.log(Point.distance(p1, p2)); // 7.0710678118654755
```

Boxing with prototype and static methods

When a static or prototype method is called without a value for this, the this value will be undefined inside the method. This behavior will be the same even if the "use strict" directive isn't present, because code within the class body's syntactic boundary is always executed in strict mode.

```
class Animal {
  speak() {
    return this;
  }
  static eat() {
    return this;
  }
}

let obj = new Animal();
obj.speak(); // Animal {}
let speak = obj.speak;
speak(); // undefined

Animal.eat() // class Animal
let eat = Animal.eat;
eat(); // undefined
```

If the above is written using traditional function-based syntax, then autoboxing in method calls will happen in non–strict mode based on the initial *this* value. If the initial value is undefined, *this* will be set to the global object.

Autoboxing will not happen in strict mode, the *this* value remains as passed.

```
function Animal() { }

Animal.prototype.speak = function() {
  return this;
}

Animal.eat = function() {
  return this;
}

let obj = new Animal();
let speak = obj.speak;
speak(); // global object

let eat = Animal.eat;
eat(); // global object
```

Instance Properties

Instance properties must be defined inside of class methods:

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Static class-side properties and prototype data properties must be defined outside of the ClassBody declaration:

```
Rectangle.staticWidth = 20;
Rectangle.prototype.prototypeWidth = 25;
```

Field Declarations

Public and private field declarations are an experimental feature (stage 3) proposed at TC39, the JavaScript standards committee. Support in browsers is limited, but the feature can be used through a build step with systems like Babel.

Public field declarations

With the JavaScript field declaration syntax, the above example can be written as:

```
class Rectangle {
  height = 0;
  width;
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

By declaring fields up-front, class definitions become more self-documenting, and the fields are always present.

As seen above, the fields can be declared with or without a default value.

Private field declarations

Using private fields, the definition can be refined as below.

```
class Rectangle {
  #height = 0;
  #width;
  constructor(height, width) {
    this.#height = height;
    this.#width = width;
  }
}
```

It's an error to reference private fields from outside of the class; they can only be read or written within the class body. By defining things which are not visible outside of the class, you ensure that your classes' users can't depend on internals, which may change version to version.

Sub classing with `extends`

The extends keyword is used in *class declarations* or *class expressions* to create a class as a child of another class.

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(this.name + ' makes a noise.');
  }
}

class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and pass in the name parameter
  }

  speak() {
    console.log(this.name + ' barks.');
  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

If there is a constructor present in the subclass, it needs to first call super() before using "this". One may also extend traditional function-based "classes":

```
function Animal (name) {
  this.name = name;
}

Animal.prototype.speak = function () {
  console.log(this.name + ' makes a noise.');
}

class Dog extends Animal {
  speak() {
    console.log(this.name + ' barks.');
  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

Note that classes cannot extend regular (non-constructible) objects. If you want to inherit from a regular object, you can instead use Object.setPrototypeOf():

```
const Animal = {
  speak() {
    console.log(this.name + ' makes a noise.');
  }
};

class Dog {
  constructor(name) {
    this.name = name;
  }
}

// If you do not do this you will get a TypeError when you invoke speak
Object.setPrototypeOf(Dog.prototype, Animal);
```

```
let d = new Dog('Mitzie');
d.speak(); // Mitzie makes a noise.
```

Species

You might want to return Array objects in your derived array class MyArray. The species pattern lets you override default constructors.For example, when using methods such as map() that returns the default constructor, you want these methods to return a parent Array object, instead of the MyArray object. The Symbol.species symbol lets you do this:

```
class MyArray extends Array {
  // Overwrite species to the parent Array constructor
  static get [Symbol.species]() { return Array; }
}

let a = new MyArray(1,2,3);
let mapped = a.map(x => x * x);

console.log(mapped instanceof MyArray); // false
console.log(mapped instanceof Array);   // true
```

Super class calls with `super`

The super keyword is used to call corresponding methods of super class. This is one advantage over prototype-based inheritance.

```
class Cat {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Lion extends Cat {
  speak() {
    super.speak();
    console.log(`${this.name} roars.`);
  }
}

let l = new Lion('Fuzzy');
l.speak();
// Fuzzy makes a noise.
// Fuzzy roars.
```

Mix-ins

Abstract subclasses or *mix-ins* are templates for classes. An ECMAScript class can only have a single superclass, so multiple inheritance from tooling classes, for example, is not possible. The functionality must be provided by the superclass.

A function with a superclass as input and a subclass extending that superclass as output can be used to implement mix-ins in ECMAScript:

```
let calculatorMixin = Base => class extends Base {
  calc() { }
};

let randomizerMixin = Base => class extends Base {
  randomize() { }
};
```

A class that uses these mix-ins can then be written like this:

```
class Foo { }
class Bar extends calculatorMixin(randomizerMixin(Foo)) { }
```

Running in Scratchpad

A class can't be redefined. If you're playing with code in Scratchpad (Firefox menu Tools > Web Developer > Scratchpad) and you 'Run' a definition of a class with the same name twice, you'll get a confusing SyntaxError: redeclaration of let <class-name>.

To re-run a definition, use Scratchpad's menu Execute > Reload and Run.

Multiple ways to create objects in JavaScript

Source	https://codeburst.io/various-ways-to-create-javascript-object-9563c6887a47
Date of Retrieval	02.03.2019

Create JavaScript Object with Object Literal

One of easiest way to create a javascript object is object literal, simply define the property and values inside curly braces as shown below

```
let bike = {name: 'SuperSport', maker:'Ducati', engine:'937cc'};
```

- **Property accessors**
  Properties of a javascript object can be accessed by dot notation or bracket notation as shown below

```
let bike = {name: 'SuperSport', maker:'Ducati', engine:'937cc'};
console.log(bike.engine);     //Output: '937cc'
console.log(bike['maker']);   //Output: 'Ducati'
```

- **Adding property to the object**
  To add property to the already created object, no need to change the existing object literal, property can be added later with dot notation as shown below

```
let bike = {name: 'SuperSport', maker:'Ducati', engine:'937cc'};
bike.wheelType = 'Alloy';

console.log(bike.wheelType);   //Output: Alloy
```

- **Object methods**
  Behavior can be added to the object as well, behaviors are nothing but functions or methods. Methods can be part of object while creation or can be added later like properties as shown below

```
let bike = {
  name: 'SuperSport',
  maker:'Ducati',
  start: function() {
     console.log('Starting the engine...');
  }
};
bike.start();   //Output: Starting the engine...
/* Adding method stop() later to the object */
bike.stop = function() {
   console.log('Applying Brake...');
}
bike.stop();   //Output: Applying Brake...
```

Create JavaScript Object with Constructor

Constructor is nothing but a function and with help of new keyword, constructor function allows to create multiple objects of same flavor as shown below

```
function Vehicle(name, maker) {

  this.name = name;

  this.maker = maker;

}

let car1 = new Vehicle(’Fiesta’, 'Ford’);

let car2 = new Vehicle(’Santa Fe’, 'Hyundai’)

console.log(car1.name);    //Output: Fiesta

console.log(car2.name);    //Output: Santa Fe
```

Create JavaScript Object with create method

Object.create() allowed to create object with more atrribute options like configurable, enumerable, writable and value as shown below

```
let car = Object.create(Object.prototype,
 {
   name:{
     value: 'Fiesta',
     configurable: true,
     writable: true,
     enumerable: false
   },
   maker:{
     value: 'Ford',
     configurable: true,
     writable: true,
     enumerable: true
   }
 });
```

prototype

- Every single object is built by constructor function.
- A constructor function makes an object linked to its own prototype.

- Prototype is an arbitrary linkage between the constructor function and object.

Create JavaScript Object using ES6 classes

ECMAScript 6 (newer version of javascript) supports class concept like any other Statically typed or object oriented language. So, object can be created out of a class in javascript as well as shown below

```
class Vehicle {
 constructor(name, maker, engine) {
   this.name = name;
   this.maker =  maker;
   this.engine = engine;
 }
}
let bike1 = new Vehicle('Hayabusa', 'Suzuki', '1340cc');
```

- **methods to the JavaScript class**
  Methods can be part of class while declaration or can be added later to the created object as shown below

```
class Vehicle {
 constructor(name, maker, engine) {
   this.name = name;
   this.maker =  maker;
   this.engine = engine;
 }
 start() {
   console.log("Starting...");
 }
}
let bike = new Vehicle('Hayabusa', 'Suzuki', '1340cc');
bike.start();    //Output: Starting...
```

Session 2: Matching Patterns in JavaScript

A guide to JavaScript Regular Expressions

Source	https://blog.bitsrc.io/a-beginners-guide-to-regular-expressions-regex-in-javascript-9c58feb27eb4
Date of Retrieval	02.03.2019

When you first encounter Regular Expressions, they may seem like a random string of gibberish. While they might look awkward (with a somewhat confusing syntax), they are also extremely useful.
The truth is properly understanding regular expressions will make you a much more effective programmer. In order to fully understand the regex world you first need to learn the basics concepts, on which you can later build.

What are Regular Expressions?

Regular expressions are a way to describe patterns in a string data. They form a small language of its own, which is a part of many programming languages like Javascript, Perl, Python, Php, and Java.
Regular expressions allow you to check a string of characters like an e-mail address or password for patterns, to see so if they match the pattern defined by that regular expression and produce actionable information.

Creating a Regular Expression

There are two ways to create a regular expression in Javascript. It can be either created with RegExp constructor, or by using forward slashes (/) to enclose the pattern.
Regular Expression Constructor:
Syntax: new RegExp(pattern[, flags])

Example:

```
var regexConst = new RegExp('abc');
```

Regular Expression Literal:

Syntax: /pattern/flags
Example:

```
var regexLiteral = /abc/;
```

Here the flags are optional, I will explain these later in this article.

There might also be cases where you want to create regular expressions dynamically, in which case regex literal won't work, so you have to use a regular expression constructor.
No matter which method you choose, the result is going to be a regex object. Both regex objects will have same methods and properties attached to them.

Since forward slashes are used to enclose patterns in the above example, you have to escape the forward slash (/) with a backslash (\ ) if you want to use it as a part of the regex.

Regular Expressions Methods

There are mainly two methods for testing regular expressions.

RegExp.prototype.test()

This method is used to test whether a match has been found or not. It accepts a string which we have to test against regular expression and returns true or false depending upon if the match is found or not.

For example:

```
var regex = /hello/;
var str = 'hello world';
var result = regex.test(str);
console.log(result);
// returns true
```

RegExp.prototype.exec()

This method returns an array containing all the matched groups. It accepts a string that we have to test against a regular expression.

For example:

```
var regex = /hello/;
var str = 'hello world';
var result = regex.exec(str);
console.log(result);
// returns [ 'hello', index: 0, input: 'hello world', groups: undefined ]
// 'hello' -> is the matched pattern.
// index: -> Is where the regular expression starts.
// input: -> Is the actual string passed.
```

We are going to use the test() method in this article.

Simple Regex Patterns

It is the most basic pattern, which simply matches the literal text with the test string. For example:

```
var regex = /hello/;
console.log(regex.test('hello world'));
```

```
// true
```

Special Characters

Up until now we've created simple regular expression patterns. Now, let's tap into the full power of regular expressions when handling more complex cases.
For example, instead of matching a specific email address let's say we'd like to match a number of email addresses. That's where special characters come into play. There are special symbols and characters that you have to memorize in order to fully understand the regular expressions.

Flags:

Regular expressions have five optional flags or modifiers. Let's discuss the two most important flags:

- g—Global search, don't return after the first match
- i—Case-insensitive search

You can also combine the flags in a single regular expression. Note that their order doesn't have any effect on the result.

Let's look at some code examples:

Regular Expression Literal — Syntax /pattern/flags

```
var regexGlobal = /abc/g;
console.log(regexGlobal.test('abc abc'));
// it will match all the occurence of 'abc', so it won't return
// after first match.
var regexInsensitive = /abc/i;
console.log(regexInsensitive.test('Abc'));
// returns true, because the case of string characters don't matter
// in case-insensitive search.
```

Regular Expression Constructor — Syntax new RegExp('pattern', 'flags')

```
var regexGlobal = new RegExp('abc','g')
console.log(regexGlobal.test('abc abc'));
// it will match all the occurence of 'abc', so it won't return // after first match.
var regexInsensitive = new RegExp('abc','i')
console.log(regexInsensitive.test('Abc'));
// returns true, because the case of string characters don't matter // in case-insensitive
search.
```

Character groups:

Character set [xyz]—A character set is a way to match different characters in a single position, it matches any single character in the string from characters present inside the brackets. For example:

```
var regex = /[bt]ear/;
```

```
console.log(regex.test('tear'));
// returns true
console.log(regex.test('bear'));
// return true
console.log(regex.test('fear'));
// return false
```

Note—All the special characters except for caret (^) (Which has entirely different meaning inside the character set) lose their special meaning inside the character set.

Negated character set [^xyz]—It matches anything that is not enclosed in the brackets. For example:

```
var regex = /[^bt]ear/;
console.log(regex.test('tear'));
// returns false
console.log(regex.test('bear'));
// return false
console.log(regex.test('fear'));
// return true
```

Ranges [a-z]—Suppose we want to match all of the letters of an alphabet in a single position, we could write all the letters inside the brackets, but there is an easier way and that is **ranges**. For example: **[a-h]** will match all the letters from a to h. Ranges can also be digits like **[0-9]** or capital letters like **[A-Z]**.

```
var regex = /[a-z]ear/;
console.log(regex.test('fear'));
// returns true
console.log(regex.test('tear'));
// returns true
```

Meta-characters—Meta-characters are characters with a special meaning. There are many meta character but I am going to cover the most important ones here.

- **\d**—Match any digit character (same as [0-9]).
- **\w**—Match any word character. A word character is any letter, digit, and underscore. (Same as [a-zA-Z0–9_]) i.e alphanumeric character.
- **\s**—Match a whitespace character (spaces, tabs etc).
- **\t**—Match a tab character only.
- **\b**—Find a match at beginning or ending of a word. Also known as word boundary.
- **.**—(period) Matches any character except for newline.
- **\D**—Match any non digit character (same as [^0–9]).
- **\W**—Match any non word character (Same as [^a-zA-Z0–9_]).
- **\S**—Match a non whitespace character.

Quantifiers:—Quantifiers are symbols that have a special meaning in a regular expression.

- **+—Matches the preceding expression 1 or more times.**

```
 var regex = /\d+/;
 console.log(regex.test('8'));
 // true
console.log(regex.test('88899'));
// true
console.log(regex.test('8888845'));
// true
```

- *** —Matches the preceding expression 0 or more times.**

```
var regex = /go*d/;
console.log(regex.test('gd'));
// true
console.log(regex.test('god'));
// true
console.log(regex.test('good'));
// true
console.log(regex.test('goood'));
// true
```

- **? — Matches the preceding expression 0 or 1 time, that is preceding pattern is optional.**

```
var regex = /goo?d/;
console.log(regex.test('god'));
// true
console.log(regex.test('good'));
// true
console.log(regex.test('goood'));
// false
```

- **^ — Matches the beginning of the string, the regular expression that follows it should be at the start of the test string. i.e the caret (^) matches the start of string.**

```
var regex = /^g/;
console.log(regex.test('good'));
// true
console.log(regex.test('bad'));
// false
console.log(regex.test('tag'));
// false
```

- **$ — Matches the end of the string, that is the regular expression that precedes it should be at the end of the test string. The dollar ($) sign matches the end of the string.**

```
var regex = /.com$/;
```

```
console.log(regex.test('test@testmail.com'));
// true
console.log(regex.test('test@testmail'));
// false
```

- **{N}—Matches exactly N occurrences of the preceding regular expression.**

```
var regex = /go{2}d/;
console.log(regex.test('good'));
// true
console.log(regex.test('god'));
// false
```

- **{N,}—Matches at least N occurrences of the preceding regular expression.**

```
var regex = /go{2,}d/;
console.log(regex.test('good'));
// true
console.log(regex.test('goood'));
// true
console.log(regex.test('gooood'));
// true
```

- **{N,M}—Matches at least N occurrences and at most M occurrences of the preceding regular expression (where M > N).**

```
var regex = /go{1,2}d/;
console.log(regex.test('god'));
// true
console.log(regex.test('good'));
// true
console.log(regex.test('goood'));
// false
```

Alternation X|Y — Matches either X or Y. For example:

```
var regex = /(green|red) apple/;
console.log(regex.test('green apple'));
// true
console.log(regex.test('red apple'));
// true
```

```
console.log(regex.test('blue apple'));
// false
```

Note—If you want to use any special character as a part of the expression, say for example you want to match literal + or ., then you have to escape them with backslash (\ ).
For example:

```
var regex = /a+b/;  // This won't work
var regex = /a\+b/; // This will work
console.log(regex.test('a+b')); // true
```

Advanced

(x)—Matches x and remembers the match. These are called capturing groups. This is also used to create sub expressions within a regular expression. For example :-

```
var regex = /(foo)bar\1/;
console.log(regex.test('foobarfoo'));
// true
console.log(regex.test('foobar'));
// false
```

\1 remembers and uses that match from first subexpression within parentheses.
(?:x)—Matches x and does not remember the match. These are called non capturing groups. Here \1 won't work, it will match the literal \1.

```
var regex = /(?:foo)bar\1/;
console.log(regex.test('foobarfoo'));
// false
console.log(regex.test('foobar'));
// false
console.log(regex.test('foobar\1'));
// true
```

x(?=y)—Matches x only if x is followed by y. Also called positive look ahead. For example:

```
var regex = /Red(?=Apple)/;
console.log(regex.test('RedApple'));
// true
```

In the above example, match will occur only if Redis followed by Apple.
Practicing Regex:
Let's practice some of the concepts that we have learned above.

- **Match any 10 digit number :**

```
var regex = /^\d{10}$/;
console.log(regex.test('9995484545'));
// true
```

Let's break that down and see what's going on up there.

1. If we want to enforce that the match must span the whole string, we can add the quantifiers ^ and $. The caret ^ matches the start of the input string, whereas the dollar sign $ matches the end. So it would not match if string contain more than 10 digits.
2. \d matches any digit character.
3. {10} matches the previous expression, in this case \d exactly 10 times. So if the test string contains less than or more than 10 digits, the result will be false.

- **Match a date with following format** DD-MM-YYYY or DD-MM-YY

```
var regex = /^(\d{1,2}-){2}\d{2}(\d{2})?$/;
console.log(regex.test('01-01-1990'));
// true
console.log(regex.test('01-01-90'));
// true
console.log(regex.test('01-01-190'));
// false
```

Let's break that down and see what's going on up there.

1. Again, we have wrapped the entire regular expression inside ^ and $, so that the match spans entire string.
2. (start of first subexpression.
3. \d{1,2} matches at least 1 digit and at most 2 digits.
4. - matches the literal hyphen character.
5.) end of first subexpression.
6. {2} match the first subexpression exactly two times.
7. \d{2} matches exactly two digits.
8. (\d{2})? matches exactly two digits. But it's optional, so either year contains 2 digits or 4 digits.

- **Matching Anything But a Newline**

The expression should match any string with a format like abc.def.ghi.jklwhere each variable a, b, c, d, e, f, g, h, i, j, k, l can be any character except new line.

```
var regex = /^(.{3}\.){3}.{3}$/;
console.log(regex.test('123.456.abc.def'));
// true
console.log(regex.test('1243.446.abc.def'));
// false
console.log(regex.test('abc.def.ghi.jkl'));
// true
```

Let's break that down and see what's going on up there.

1. We have wrapped entire regular expression inside ^ and $, so that the match spans entire string.
2. (start of first sub expression
3. .{3} matches any character except new line for exactly 3 times.
4. \. matches the literal . period
5.) end of first sub expression
6. {3} matches the first sub expression exactly 3 times.
7. .{3} matches any character except new line for exactly 3 times.

Session 3: Preprocessing JavaScript Files

Set Up CSS and JS Preprocessors

Source	https://developers.google.com/web/tools/setup/setup-preprocessors
Date of Retrieval	02.03.2019

CSS preprocessors such as Sass, as well as JS preprocessors and transpilers can greatly accelerate your development when used correctly. Learn how to set them up.

TL;DR

- Preprocessors let you use features in CSS and JavaScript that your browser doesn't support natively, for example, CSS variables.
- If you're using preprocessors, map your original source files to the rendered output using Source Maps.
- Make sure your web server can serve Source Maps.
- Use a supported preprocessor to automatically generate Source Maps.

What's a preprocessor?

A preprocessor takes an arbitrary source file and converts it into something that the browser understands.

With CSS as output, they are used to add features that otherwise wouldn't exist (yet): CSS Variables, Nesting and much more. Notable examples in this category are Sass, Less and Stylus.

With JavaScript as output, they either convert (compile) from a completely different language, or convert (transpile) a superset or new language standard down to today's standard. Notable examples in this category are CoffeeScript and ES6 (via Babel).

Debugging and editing preprocessed content

As soon as you are in the browser and use DevTools to edit your CSS or debug your JavaScript, one issue becomes very apparent: what you are looking at does not reflect your source, and doesn't really help you fix your problem.
In order to work around, most modern preprocessors support a feature called Source Maps.

What are Source Maps?

A source map is a JSON-based mapping format that creates a relationship between a minified file and its sources. When you build for production, along with minifying and combining your JavaScript files, you generate a source map that holds information about your original files.

How Source Maps work

For each CSS file it produces, a CSS preprocessor generates a source map file (.map) in addition to the compiled CSS. The source map file is a JSON file that defines a mapping between each generated CSS declaration and the corresponding line of the source file.

Each CSS file contains an annotation that specifies the URL of its source map file, embedded in a special comment on the last line of the file:

```
/*# sourceMappingURL=<url> */
```

For instance, given an Sass source file named styles.scss:

```
%$textSize: 26px;
$fontColor: red;
$bgColor: whitesmoke;
h2 {
    font-size: $textSize;
    color: $fontColor;
    background: $bgColor;
}
```

Sass generates a CSS file, styles.css, with the sourceMappingURL annotation:

```
h2 {
  font-size: 26px;
  color: red;
  background-color: whitesmoke;
}
/*# sourceMappingURL=styles.css.map */
```

Below is an example source map file:

```
{
  "version": "3",
  "mappings":"AAKA,EAAG;EACC,SAAS,EANF,IAAI;EAOX,KAAK"
  "sources": ["sass/styles.scss"],
  "file": "styles.css"
}
```

Verify web server can serve Source Maps

Some web servers, like Google App Engine for example, require explicit configuration for each file type served. In this case, your Source Maps should be served with a MIME type of application/json, but Chrome will actually accept any content-type, for example application/octet-stream.

Bonus: Source mapping via custom header

If you don't want an extra comment in your file, use an HTTP header field on the minified JavaScript file to tell DevTools where to find the source map. This requires configuration or customization of your web server and is beyond the scope of this document.

```
X-SourceMap: /path/to/file.js.map
```

Like the comment, this tells DevTools and other tools where to look for the source map associated with a JavaScript file. This header also gets around the issue of referencing Source Maps in languages that don't support single-line comments.

Session 4: Grouping Scripts and Non-Blocking Scripts

Non-Blocking JavaScript

Source	https://medium.com/js-360/non-blocking-javascript-c39628d7a8c2
Date of Retrieval	02.03.2019

Executing javascript in browser is a blocking process both UI process and Network request. When Js file size increase browser taken long time for both downloading and parsing Js. Both this operation block the browser from doing other work like rendering the UI. We will see various option available to mitigate this issue.

Deferred Scripts

From HTML-4 script tag can have an attribute called deferred.

```
<script type="text/javascript" src="file1.js" defer></script>
```

Script with deferred attribute can be placed anywhere in the document. file1.js will start to download once script tag got parsed by browser. But file1.js will not be executed until DOM parsing is completed but will finish execution before DOMContentLoaded event get fired. We can use this attribute when the scripts inside file1.js is not depends on DOM. Browsers will ignore deferred attribute when there is no src attribute in script tag.

Async Scripts

HTML-5 introduced another attribute for script tag called **async**.

```
<script type="text/javascript" src="file1.js" async></script>
```

When the browser sees a script tag with async attribute during parsing the DOM, it will not stop the parsing for downloading the script and executing it. Rather it will start downloading and executing the script asynchronously while parsing the DOM is already in progress. The script we want to load asynchronously does not really depend on anything being ready in the DOM when it runs and it also doesn't need to be run in any particular order.
Since execution order is not guaranteed following will be problem

```
<script type="text/javascript" src="lib.js" async></script>
<script type="text/javascript" src="src.js" async></script>
```

So extra care needs to be taken when you plan to use async.

Dynamically Loading Scripts

When we add our script in a page by create script tag using javascript, it will executed asynchronously (same as above) and does not block the browser.

```
var script = document.createElement('script')
script.src="file1.js";
document.head.appendChild(script);
```

Script will start download and execute after it has been added to header. If you want to execute this script synchronously set async attribute to false.

```
var script = document.createElement('script')
script.src="file1.js";
script.async=false;
document.head.appendChild(script);
```

Now script will start download and execute synchronously and block the browser.
By using above mentioned approach we can improve the time to first meaningful paint also a use experience.

Internal and External JavaScript

Source	https://www.thoughtco.com/how-to-create-and-use-external-javascript-files-4072716
Date of Retrieval	02.03.2019

Placing JavaScripts directly into the file containing the HTML for a web page is ideal for short scripts used while learning JavaScript. When you start creating scripts to provide significant functionality for your web page, however, the quantity of JavaScript can become quite large, and including these large scripts directly in the web page poses two problems:

- It may affect the ranking of your page with the various search engines if the JavaScript takes up a majority part of the page content. This lowers the frequency of use of keywords and phrases that identify what the content is about.
- It makes it harder to reuse the same JavaScript feature on multiple pages on your website. Each time you want to use it on a different page, you will need to copy it and insert it into each additional page, plus any changes the new location requires.

It is much better if we make the JavaScript independent of the web page that uses it.

Selecting JavaScript Code to Be Moved

Fortunately, the developers of HTML and JavaScript have provided a solution to this problem. We can move our JavaScripts off of the web page and still have it function exactly the same.

The first thing that we need to do to make a JavaScript external to the page that uses it is to select the actual JavaScript code itself (without the surrounding HTML script tags) and copy it into a separate file.

For example, if the following script is on our page we would select and copy the part in bold:

```
<script type="text/javascript">
var hello = 'Hello World';
document.write(hello);
</script>
```

There used to be a practice placing JavaScript in an HTML document inside of comment tags to stop older browsers from displaying the code; however, new HTML standards say that browsers should automatically treat the code inside of HTML comment tags as comments, and this results in browsers ignoring your Javascript.

If you have inherited HTML pages from someone else with JavaScript inside of comment tags, then you don't need to include the tags in the JavaScript code that you select and copy.

For example, you would only copy the bold code, leaving out the HTML comment tags <!-- and --> in the code sample below:

```
<script type="text/javascript"><!--
var hello = 'Hello World';
document.write(hello);
// --></script>
```

Saving JavaScript Code as a File

Once you have selected the JavaScript code you want to move, paste it into a new file. Give the file a name that suggests what the script does or identifies the page where the script belongs.

Give the file a .js suffix so that you know the file contains JavaScript. For example we might use hello.js as the name of the file for saving the JavaScript from the example above.

Linking to the External Script

Now that we have our JavaScript copied and saved into a separate file, all we need to do is reference the external script file on our HTML web page document.

First, delete everything between the script tags:

```
<script type="text/javascript">
</script>
```

This doesn't yet tell the page what JavaScript to run, so we next need to add an extra attribute to the script tag itself that tells the browser where to find the script.

Our example will now look like this:

```
<script type="text/javascript"
src="hello.js">
</script>
```

The src attribute tells the browser the name of the external file from where the JavaScript code for this web page should be read (which is hello.js in our example above).

You do not have to put all of your JavaScripts into the same location as your HTML web page documents. You may want to put them into a separate JavaScript folder. In this case, you just modify the value in the src attribute to include the file's location. You can specify any relative or absolute web address for the location of the JavaScript source file.

Using What You Know

You can now take any script that you have written or any script that you have obtained from a script library and move it from the HTML web page code into an externally referenced JavaScript file.

You may then access that script file from any web page simply by adding the appropriate HTML script tags that call that script file.

Session 5: Interacting with HTML Objects Using JavaScript

How to traverse the DOM in JavaScript?

Source	https://medium.com/javascript-in-plain-english/how-to-traverse-the-dom-in-javascript-d6555c335b4e
Date of Retrieval	02.03.2019

The Document Object Model, or shortly the DOM, serves as a reference for the browser when placing elements on the web page. The locations where the elements are placed in the DOM are called Nodes, and on the web page, it's not that only HTML elements get their node, but also the attributes of the HTML elements have their nodes (attribute nodes), every piece of text has its node (text nodes), and there are many other node types. The structural relation of these nodes reflects the structure of the HTML document. Because of that, we can define the relations between the elements on the page as the relations between their nodes in the DOM.

When we are manipulating the elements on the web page with a programming language, such as JavaScript, we are doing that through their DOM nodes. By accessing a DOM node of a given element we can manipulate its properties, such as position, appearance, content, behavior, etc. Often we want to perform actions on the elements that have some kind of relation between them, known as related nodes. In order to do that, we must have a way of moving from one node to the other nodes, that is, the way of traversing the DOM.

Having the access to a certain node in the DOM, there are ways to traverse through the DOM using its related nodes. We can move up or down the DOM tree, or we can move sideways staying at the same DOM level. In this article, we will take a look how can we access the related DOM nodes using the JavaScript programming language.

Relations between DOM nodes

First, let's see what relations between elements exist so that we can later better understand the techniques used to access them. HTML elements are nested within each other creating a tree like structure. There may be many levels of nesting elements and all of that reflects in the DOM tree of element nodes.

Descendant and ancestor elements

This relation is of no practical use, but it will help us with expressing some element relations clearer.

One element may have many levels of other elements nested under it, and all of those nested elements in all of the nesting levels are called the descendant elements of our starting element. For example, let's have a main element for the main content of the page with the following content:

```
<main>
        <h1>Today's articles</h1>


        <article id="article-1">
          <h2>First Contact with Alien Beings in History of Mankind</h2>
          <section>
            <p>Finally! They are here.....</p>
            <p>What do we do now!?</p>
          </section>
        </article>


        <article id="article-2">
          <h2>How to grow Bonsai</h2>
          <section>
            <p>If you are in need of stress relief ...</p>
          </section>
        </article>
      </main>
```

At the first level of nesting there are <h1> element and two <article> elements. Then, at the second nesting level we have <h2> and <section> elements, and finally in the third level there are <p> elements inside <section> elements. All of these elements are descendants of the <main> element.

Given that said, the <main> element is their ancestor element, the element that they belong to in the DOM tree.

But descendant/ancestor relationship can be seen between other elements in this example too. For instance, the <article> elements are ancestors for their nested <h2>, <section> and <p> elements, and those elements are, in turn, its descendant elements. The same relation applies to the<section> and <p> elements.

Here, we have to point out one important thing. For example, the heading “How to grow Bonsai” is not a descendant of the <article id=”article-1">, neither that article is the ancestor of the mentioned heading. The reason for that is because the “How to grow Bonsai” heading is not nested within the first article, but rather under the second article. Therefore, they do not have descendant/ancestor relationship between them. The same applies to <section> and <p> elements, they are descendants of the article element under which they are nested, and that article element is their ancestor.

Parent and Child elements

The special, and very useful, descendant/ancestor relationship is the case where the elements are direct descendants or ancestors of the given node. The ‘direct’, means that they are just one nesting level away from the given node.

The parent node (element) is the closest ancestor element to the given element. If we select heading “How to grow Bonsai”, the first ancestor element (one level up) is the <article id=”article-2"> element, and we call it a parent of the given heading. The <article> elements share the same parent, they are both nested under the <main> element which is their parent. Note that <h1> has the <main> element as its parent too. The paragraph “What do we do now!?” has as its parent the <section> element under the first article element.

Opposite to the parent is a child element, but while the element can have only one parent element, it can have many child elements under it. The child elements are all the direct descendant elements (one level down) of the given element. Children of the <main> element are <h1> and both <article> elements and no other elements. Child elements of the second article are “How to grow Bonsai” heading and the <section> nested under it. The paragraph in that section is not a child element of the second article element, it is a child of the <section> element.

Another important thing to note here is that every bit of text in HTML is presented with a text node in the DOM. Given that said, the heading in the first article has one text node as its child, a node containing the text “First Contact with Alien Beings in History of Mankind”, and the heading element is the parent of that text node, and the heading in the second article is the parent of the child text node with text “How to grow Bonsai”.

Sibling elements

Two or more elements are siblings if they have the same element as their parent. In our example, <h1> and both <article> elements are siblings because they have the same parent, the <main> element. The <p> elements in the first article are siblings because their parent is the <section> element in the first article. But the <p> element in the second article is not a sibling of the <p> elements in the first article because they do not all share the same parent even though they are at the same level of nesting.

Traversing the DOM via the related nodes

Finally, we will see how we can use the relations between the nodes to traverse through the DOM tree. A node in the DOM tree is represented with a Node object, and the Node object has properties that allow us to get to the related nodes of a given node.

We will add some id's and classes in our example HTML so we can better access elements in the DOM tree:

```
<main>
        <h1>Today's articles</h1>

        <article id="article-1">
          <h2 class="sensations">First Contact with Alien Beings in History of Mankind</h2>
          <section>
            <p>Finally! They are here.....</p>
            <p>What do we do now!?</p>
          </section>
        </article>

        <article id="article-2">
          <h2 class="horticulture">How to grow Bonsai</h2>
          <section>
            <p>If you are in need of stress relief ...</p>
          </section>
        </article>
      </main>
```

Finding the parent node of a given node

If we have a Node object that is a reference to a node in the DOM, to get its parent node we can use the parentNode property on the node. Since node is an object and parentNode is a property, we can use the 'dot' notation to access the parent element of the node like this:

const parent = node.parentNode;

Let's find the parent of the first article node in our example HTML.

```
// this is our node
                   const articleOne = document.querySelector('#article-1');
                   // this is the parent of our node, the node representing the 'main' element
                   const parent = articleOne.parentNode;
```

Now, let's find the parent of the heading "How to grow Bonsai".

```
// our node
            const bonsai = document.querySelector('.horticulture');
            // the parent node of bonsai node, the node representing the second article
            const parent = bonsai.parentNode;
```

We can use the parent node of a given node to get all the ancestors of the given node up in the DOM tree. For example:

```
// our node
            const bonsai = document.querySelector('.horticulture');
            // the parent node of bonsai node, the node representing the second article
            const parent = bonsai.parentNode;
            // the parent of the parent; the node representing <main> element
            const grandParent = parent.parentNode;
```

The grandParent node could also be obtained by chaining the parentNode property on the bonsai node:

const grandParent = bonsai.parentNode.parentNode;

The parentNode is often used to remove a given node from the DOM. Say we want to remove the first article from the document since it's too distressing. We would do it like this:

```
  // get the node we want to remove

  const articleOne = document.querySelector('#article-1');

  // get its parent node and call removeChild method to which we pass the node we want removed

  articleOne.parentNode.removeChild(articleOne);
```

The parent of a HTML element can be an Element node, a Document node or a DocumentFragment node. The parentNode property of a node can return null in cases where it is applied to the Document and DocumentFragment nodes because they can never have parent nodes. If a node is just created but it is not attached to the DOM, applying the parenNode on it will also return null.

One more thing to notice is that the parentNode property is a read-only property, meaning that it is not possible to do something like this:

```
const articleOne = document.querySelector('#article-
1');
                                                      const articleTwo =
                                                      document.querySelector('#article-2');
                                                      const bonsai =
                                                      document.querySelector('.horticulture');
                                                      // trying to assign a new parent will
```

```
not work
bonsai.parentNode = articleOne;
console.log(bonsai.parentNode); //
<article id="article-2">; still the
second article
```

Finding the child nodes of a given node

To get all child nodes of a node we can use its childNodes property. For example:

const children = node.childNodes;

The result is a node list, the list of objects that each represent one child node of our node. The node list is similar to the arrays in a way that it is possible to loop through the list using array indexes. For example, let's print the nodeName property of all the children in a node list

```
// get child nodes of a node
                                    const children = node.childNodes;
                                    // children node list has a length property, similarly to the
                                    arrays
                                    const len = children.length;
                                    // loop through the node list and log the names of nodes
                                    // we are using the same syntax as for the arrays
                                    for (let i = 0; i < len; i++) {
                                      console.log(children[i].nodeName);
```

Let's use a different HTML to explain some important implications of using childNodes property:

```
 <ul id="list">
   <li>Item 1</li>
   <li>Item 2</li>
   <li>Item 3</li>
   <li>Item 4</li>
   <li>Item 5</li>
   <li>Item 6</li>
 </ul>
```

If we apply the previous javascript code to print out the node names of child nodes from the <ul> element, we will get this result:

```
#text
    LI
    #text
    LI
    #text
    LI
    #text
    LI
    #text
    LI
    #text
    LI
```

Not quite as one could expect. You could think that there will only be printed out six li element names, but instead we get seven more text nodes.

The reason for this is that the node list also contains text nodes and comment nodes so we must take care when using this property. This is happening because in HTML a new line is treated as whitespace, i.e. the text node. So, the line breaks and white space between HTML tags add the text nodes to the node list of children.

To avoid this we could reorganize our HTML structure like this:

```
<ul id="list"><li>
                    Item 1</li><li>
                    Item 2</li><li>
                    Item 3</li><li>
                    Item 4</li><li>
                    Item 5</li><li>
                    Item 6</li></ul>
```

Now we get the expected output:

```
li
  li
  li
  li
  li
  li
```

The node list is a live list of nodes. That means that adding or removing child elements to the node will update the list, we don’t have to fetch it again. That also means that if there are changes to the number of elements in the node list, the for loop used like this will fail:

```
const list = document.querySelector('#list');
                                             const children = list.childNodes;
                                             const len = children.length;
                                             for (let i = 0; i < len; i++) {
                                               if (children[i].id === 'three') {
```

```
                                                        // remove the item with id="three"

                                                    children[i].parentNode.removeChild(children[i]);
                                                      }
                                                      console.log(children[i].nodeName);
                                                    }
```

This code will result in an error when it comes to the sixth iteration: TypeError: children[i] is undefined. It is because the childrennode list has changed with removing the third item. To avoid this, we should update the len variable in every iteration, or better, use children.length in the loop conditions instead of len as we did.

If we want to get only HTML elements as children of a node we can use the children property instead of childNodes property:

const childElements = list.children;

The 'special' children

Well, often in families there are favorites among the children, the 'special' ones. The same is with the node family, there are children that are special. What children are those? — you may ask. And you probably guessed, those are the first one and the last one. :)

This analogy is not a pointless joke. In the DOM tree, a node with child nodes has defined properties firstChild and lastChild. They are used to quickly find the first and the last child node under the given node. In our example the first child will correspond to the first list item, and the last child will correspond to the sixth list item. Here is how we can obtain those elements:

```
const list = document.querySelector('#list');
// the first list item
const first = list.firstChild;
console.log(first.textContent); // 'Item 1'
// the last list item
const last = list.lastChild;
console.log(last.textContent); // 'Item 6'
```

It is important to note that the firstChild and lastChild also treat the line breaks as text nodes and in the case of our first list (one with the line breaks) they would result in the text nodes rather than the list items.

Finding siblings of a node

When we have the access to a node, we can access its sibling nodes using the nextSibling and previousSibling properties.

Property nextSibling will get the sibling node that immediately follows the given node. The syntax is the following:

const next = node.nextSibling;

Let’s find the next sibling of the item with the id=”three”:

```
const current = document.querySelector('#three');
                                                  const next = current.nextSibling;
                                                  console.log(next.textContent); // 'Item 4'
```

We can now proceed to go through the siblings that follow, step by step:

```
const nextNext = next.nextSibling;
                                    console.log(nextNext.textContent); // 'Item 5'
```

When we get to the last sibling in the parent node, using the nextSibling will return null because there are no more siblings after the last child node:

```
const afterLast = last.nextSibling;
                                     console.log(afterLast); // null
```

Property previousSibling will get the sibling node that immediately precedes the given node. The syntax is analogous to the syntax for nextSibling:

const previous = node.previousSibling;

Let’s find the previous sibling of the item with the id=”three”:

```
const current = document.querySelector('#three');
                                                  const previous = current.previousSibling;
                                                  console.log(previous.textContent); // 'Item
                                                  2'
```

We can now proceed to go through other previous siblings, step by step:

```
  const previousPrevious = previous.previousSibling;
  console.log(previousPrevious.textContent); // 'Item 1'
```

When we get to the first sibling in the parent node, using the previousSibling will return null because there are no siblings before the first child node:

```
const beforeFirst = first.previousSibling;
                                           console.log(beforeFirst); // null
```

Final example

Now, let’s do one example that requires more complex traversal than we have shown so far in our previous examples. Let’s again use the first HTML example structure:

```
<main>
        <h1>Today's articles</h1>


        <article id="article-1">
          <h2 class="sensations">First Contact with Alien Beings in History of Mankind</h2>
          <section>
            <p>Finally! They are here.....</p>
            <p>What do we do now!?</p>
          </section>
        </article>


        <article id="article-2">
          <h2 class="horticulture">How to grow Bonsai</h2>
          <section>
            <p>If you are in need of stress relief ...</p>
          </section>
        </article>
      </main>
```

Say we have the access to the heading in the first article (<h2 class=”sensations”>) and we want to read the heading text from the next article.

```
// get the heading from the first article
                                         const firstH2 =
                                         document.querySelector('.sensations');
                                         // get its parent, the current article
                                         const currentArticle = firstH2.parentNode;
                                         console.log(currentArticle); // <article id="article-
                                         1">
                                         // get the next sibling of this parent
                                         const nextArticle = currentArticle.nextSibling;
                                         console.log(nextArticle); // <TextNode
                                         textContent="\n\n  ">
```

Oops, wrong node! Remember that the line breaks are treated as white space and that white space is presented with a text node. So, how can we avoid this?

We could use a loop that goes through the next siblings and that checks if the node is an element node rather than a text or any other node. To check if a node is an element node we can check a node property called nodeType. The nodeType property for an element node will have the value ELEMENT_NODE which is called a constant or it can have numeric value of 1, you may check for either one. To see all other node types and their values you can check out this article.

It is best to use while loop for this task. Let’s change the previous code and add a while loop and the check for the node type.

```
// get the heading from the first article
const firstH2 =
document.querySelector('.sensations');
// get its parent, the current article
const currentArticle = firstH2.parentNode;
console.log(currentArticle); // <article id="article-
1">
// get the next sibling node of this parent
let nextNode = currentArticle.nextSibling;
while (nextNode && nextNode.nodeType !== 1) {
  // while there is a next sibling node and it is not
an
  // element node, skip to the next sibling node
  nextNode = nextNode.nextSibling;
}
if (nextNode) {
  // if there is next sibling node and it is
  // the element node, then log it
  console.log(nextNode); // <article id="article-2">
}
```

Now we have found the next article. Finally, we want to get the heading under it. Since we know that the heading is the first element node under the article node we could think of using the firstChild property on the nextNode node, but the problem with the text nodes will come up again. To avoid this, we can use the loop again to check the type of the node, or we can use querySelector on nextNode to get the first h2 element, and, probably, this second approach might be the safest to use. But, for the sake of practice, let’s use the children property of the node object that we briefly mentioned earlier.

The children property will return the list child element nodes, and then we can just select the first one from the list. Here is the complete code:

```
// get the heading from the first article
const firstH2 =
document.querySelector('.sensations');
// get its parent, the current article
const currentArticle = firstH2.parentNode;
console.log(currentArticle); // <article id="article-
```

```
1">
// get the next sibling node of this parent
let nextNode = currentArticle.nextSibling;
while (nextNode && nextNode.nodeType !== 1) {
  // while there is a next sibling node and it is not
an
  // element node, skip to the next sibling node
  nextNode = nextNode.nextSibling;
}
if (nextNode) {
  // if there is next sibling node and it is
  // the element node, then log it
  console.log(nextNode); // <article id="article-2">
  // get the first element child node of the
nextArticle node
  const secondH2 = nextNode.children[0];
  console.log(secondH2.textContent); // 'How to grow
Bonsai'
```

Conclusion
Taking a closer look at the last example, one may ask why bother with all this traversing trough the related nodes when we can just do a simple query like this:

const secondH2 = document.querySelector('.horticulture');
While that is true in this simple case, remember that we explicitly wanted to get to the next article's heading, and that in a real website the page can be much larger, and the content can be nested much deeper. Also, the content could be generated automatically and we do not know what classes will be available to us. So, we would have to do a full query of the DOM for every next heading. But by using the related siblings we are able to be around the nodes that are of interest to us and not to query the whole DOM tree every time, therefore increasing performance when we have a very large web page.

To use this effectively on the large web page with many articles, we can wrap it all into a function that accepts the current node as the parameter:

```
function nextHeading(currentHeading) {

  // get its parent, the current article

  const currentArticle = currentHeading.parentNode;

  // get the next sibling node of this parent

  let nextNode = currentArticle.nextSibling;
```

```
    while (nextNode && nextNode.nodeType !== 1 || nextNode.nodeName !== 'ARTICLE') {
      // while there is a next sibling node and it is not
      // an element node or it is not an 'article' element,
      // skip to the next sibling node
      nextNode = nextNode.nextSibling;
    }
    // get the first element child node of the nextNode node
    // and return it if it is a 'h2' element
    // if there is no next 'article' element, then return the message
    return (nextNode && nextNode.children[0]) || 'There are no more articles to be found on
  this page!';
  }
  // get the heading from the first article
  const firstH2 = document.querySelector('.sensations');
  // call the function to get the next heading
  const secondH2 = nextHeading(firstH2);
  console.log(secondH2); // <h2 class="horticulture">
```

Notice that we added one more check nextNode.nodeName === 'ARTICLE' to the while loop condition, just to make sure we are getting the right type of the element, because it might happen that there are other siblings to the article elements which are not articles.

Now, we can call the function nextHeading on the second heading to try get the heading from the next article:

```
// try to get the next heading
                              const thirdH2 = nextHeading(secondH2);
                              console.log(thirdH2); // 'There are no more articles to be found
                              on this page!'
```

There are many use cases when you could use the related nodes to traverse the DOM. The method that you'll use depend on the HTML structure and on your imagination. So, get to know the node family and their relations well, and they will help you a lot with DOM related tasks.
Thank you for reading and have fun finding your way through the DOM!

Session 6: JavaScript Minification

How to Minify CSS, JS, and HTML?

Source	https://codeburst.io/how-to-minify-css-js-and-html-ddd9dbea25c6
Date of Retrieval	02.03.2019

Google is very strict when it comes to ranking websites. Not only your site should be SEO optimized, but also adhere to the strict loading times for optimal user experience. It all starts when a visitor visits your website for the first time. If the site is slow, the visitor will drop from the website in search of a better website that loads fast and offers the same or better content.

Google knows that. And, that's why they also rank websites that load faster. As a blogger or a business owner, your job is to load your site as fast as possible. There are many ways to do so, but in this article, we will be solely focusing on the how to minify CSS, JS, and HTML.

Your website is built using a lot of files where the majority of them are HTML, CSS, and JS. These files contain tons of codes that is auto-generated or written by a developer. This makes them have size. The code that is written in these files is humanly readable as they need to be maintained. However, the same is not true for machines as they can quickly read through the code. In short, it means that the computer doesn't need formatted code which we can use to save space and help improve the websites file size.

Before we get started with the real process, we will first learn what exactly minify means and how it can benefit website loading time.

What is Minify?

Minify is the process by which unnecessary space or characters are removed from the code. These spaces and characters are not necessary to run the code and hence just add size to the file. When they are removed, the files become lighter which in turn improves the page loading time. Minify is a great strategy to improve meet visitor's expectation and rank better.

What does Minify remove?

When the minification process is done, it removes the following things from your code:

1. New line characters

2. Whitespace characters

3. Block delimiters

4. Comments

All of these characters and comments add readability to the code which is solely meant for human readers. Minification helps overall data transferred when a website is requested from the server.

As a developer, it is easy to distinguish between a minified file and an un-minified file. The minified file has an extension of .min in it. For example, header.min.css

Minification and compression difference

Compression is not similar to minification. Compression is a technique where file size is reduced by using compression algorithms or schemes ushc as brotli or gzip. Both of them serve the purpose of reducing the file size but with different approaches. So, basically, you can minify the files and then compress it before sending it to the client requesting the website. The files once received from the client side will then be uncompressed and then used for rendering purposes.

This difference is explained in most of the beginner's blogging guide in performance optimization guides.

Minifying manually

Now that we have understood what minification is and its difference from compression, it is now time for us to learn how to minify CSS, JS, and HTML.

The first process is very straightforward. All you need to do is remove the unnecessary things from your code. Let's take a look at an example below.

```
<!DOCTYPE html>

<html>

<head>

 <title>Portfolio</title>

 <meta charset="utf-8">

 <meta name="viewpoint" content="width=device-width", initial-scale="1">

 <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

 <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></script>

 <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>

</head>

<body>

<nav>

 <ul>

 <li>Home</li>

 </ul>

</nav>

</body>
```

</html>

After minifications it will look like below:

```
<!DOCTYPE html><html><head><title>Portfolio</title><meta charset=”utf-8"><meta
name=”viewpoint” content=”width=device-width”, initial-scale=”1"><link rel=”stylesheet”
href=”https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"> <script
src=”https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></script> <script
src=”https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script></head><bod
y><nav><ul><li>Home</li></ul></nav></body></html>
```

Isn’t that hard to read? Maybe for humans, but not for the computer. The above format removes any unnecessary whitespaces, breaks and so on making it smaller and hence decreasing the file loading time.

Using Online Tools

The manual process can be, and you don’t want to waste time when setting up a website for the first time. That’s why we have tools that do the minification for you. So, let’s list them below.

CSS

CSSminifier.com: A simple to use tool that lets you minify CSS. All you need to do is copy and paste your code and download the minified version as a file.

phpied.com: It is a development tool that uses YUI Compressor CSS minification.

JS

Jscompress.com: It is a JavaScript-based minification tool. This tool lets you copy-paste your code and download the minified code.

yui.github.io: A development tool that can be used during development.

HTML

htmlcompress.com: An online minification tool that lets you minify HTML, CSS and JavaScript.

HTMLMinifier: You can also check out this JavaScript-based HTML compressor. It is one of the best and can be integrated into your project easily.

Conclusion

There is a lot of online competition. To succeed you need to make the most out of the available tools, techniques, and methods. That’s why we went forward and shared insight on minification. We also listed the tools that let you naturally do the minification. If you are using a CMS such as WordPress, you will find a lot of caching or even dedicated minification plugin that does it for you.

Session 7: Web Workers

Why Web Workers Make JavaScript More Than a Single Thread?

Source	https://codeburst.io/why-web-workers-make-javascript-more-than-a-single-thread-3d489ffad502
Date of Retrieval	02.03.2019

It's been drilled into our heads that JavaScript is not concurrent, that it's just a single threaded language. Though that may be technically true, the problem is that, unlike other scripting languages, JavaScript was created to interact with web browsers. So, if you want to do anything computationally demanding, you do that at the cost of lots of jank, thanks to single threading.

To those uninitiated to the ways of jank, this phenomenon is caused by code that blocks the runtime from rendering important content. It's the user having to twiddle their thumbs at a useless interface. That's not good UX(User Experience).

Then, how do websites function nowadays? Is there some Jedi magic going on that allows JavaScript developers to excuse themselves from ever running any code that computes anything at all? Well, the answer might be yes actually. You can bind C++ code to JavaScript with the help of libraries, but this is used within the Node environment. That's the environment in which JavaScript is more synonymous with Ruby and all those other back end scripting languages.

We're talking about client-side JavaScript here, and, no, there is no magic in this realm, though some may beg to differ.

What we do have is Web Workers.

Workers allow you to run background threads so that your main program can take care of simple logic. This is where the underworld of Web APIs meets JavaScript's runtime, and this is why JavaScript is technically single threaded.

See, JavaScript knows it's supposed to run one piece of code at a time, but it has no problem making shady deals under the table. These shady deals come in the form of asynchronous calls. You know, the usual setTimeout(). That's a web API and not part of JavaScript's core language.

A Worker is also an API that we can utilize to implement concurrency. If you have doubts, this is nothing new. It has nothing to do with es6. It's been around the block—so much so that it's actually widely supported. Of course, IE 10 support is as good as you're going to get.

How does it work? It's actually easy to understand. Here's the work flow of a program with a worker:

main.js:main thread | worker.js: worker thread

1.new worker created |

2.post a message -> | 3.catch the message => {do something…}

5. catch the result | 4. <- post the result

1.In our main.js file, we instantiate a instance of the Worker object like so:

main.js: let worker = new Worker('worker.js');

Note that main.js can be any JavaScript file containing the data that you want to pass on to the worker to undergo computation.

2. We then post a message.

let myObj = {name: 'Ozymandias', age:200 };

main.js: postMessage(myObj);

3. On the worker.js side of life, we add an event listener to self. You can say that your worker js file is wrapped by a Worker class. Any event listener you add is a reference to itself.

```
worker.js:

addEventListener('message', (e)=>{

  e.data // {name: 'Ozymandias', age:200}

});
```

4. After making changes to our data, we can send the results back with another postMessage()`

```
worker.js:

addEventListener('message', (e)=>{

  e.data; // {name: 'Ozymandias', age:200}

  postMessage(e.data);

});
```

5. Now we can catch it in our main.js.

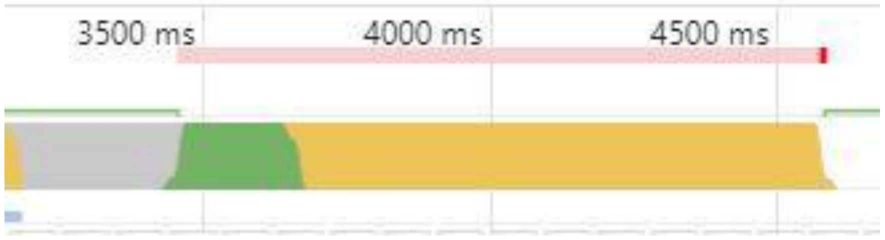
```
main.js:

worker.addEventListener('message', (e)=>{

  e.data // {name: 'Ozymandias', age:200}

});
```

Into the Jungle

Now that we have an idea of how Web Workers work, let’s actually use a semi-wild example to illustrate their overwhelming benefit. Here’s the scenario. Geo Pixer, a photophile, wants to build an image sharing site called GPix. He’s written the JavaScript code below to prototype the uploading proccess.

```
et $photoInput = document.getElementById("input");

let fileReader = new FileReader();

let image = new Image();

let $editor = document.getElementById("editor");

let $editorCtx = $editor.getContext("2d");

//This is a performance test

function opacitor(op) {

  let imgData = $editorCtx.getImageData(0, 0, $editor.width, $editor.height);

  for (let x = 0; x < image.width; x++) {

    for (let y = 0; y < image.height; y++) {

      let index = (x + y * image.width) * 4;

      imgData.data[index + 3] = op;

    }

  }

  $editorCtx.putImageData(imgData, 0, 0);

}

image.addEventListener("load", e => {

  $editorCtx.drawImage(image, 0, 0);

  opacitor(23);
});

fileReader.addEventListener("load", e => {

  let base64 = e.target.result;

  image.src = base64;

});

$photoInput.addEventListener("change", e => {

  let file = e.target.files[0];

  fileReader.readAsDataURL(file);

});
```

When he uploads a 2400x2000 image, this is how the code performs:



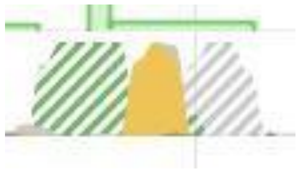
If you look at that red bar, that's Chrome warning Geo that his code is full of jank.

Sure, one culprit of the jank is the opacity function he created. But the two load() event listeners(the ugly yellow block) contribute a major portion to the jankification of Geo's code. That's because the image has to be decoded before it can load.

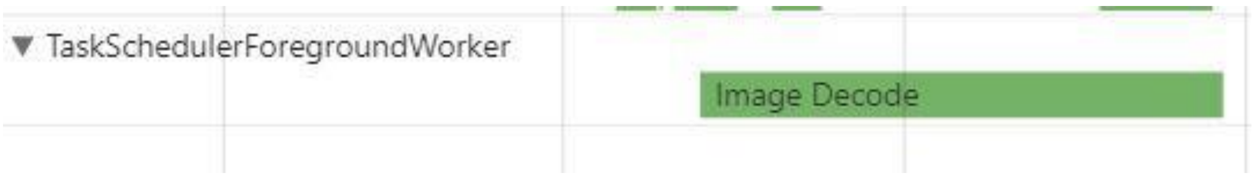
So, Geo stares at a blank screen for three seconds before the image shows up. That's a no, no. Here's a refactored version of the code.

```
let $photoInput = document.getElementById("input");
                                                   let image = new Image();
                                                   let $editor =
                                                   document.getElementById("editor");
                                                   let $editorCtx = $editor.getContext("2d");


                                                   function opacitor(op) {
                                                     let imgData = $editorCtx.getImageData(0,
0, $editor.width, $editor.height);
  for (let x = 0; x < image.width; x++) {
    for (let y = 0; y < image.height; y++)
{
      let index = (x + y * image.width) *
4;
      imgData.data[index + 3] = op;
    }
  }
  $editorCtx.putImageData(imgData, 0, 0);
}


$photoInput.addEventListener("change", e =>
{
  let file = e.target.files[0];
  createImageBitmap(file).then(bitmap => {
    $editor.width = bitmap.width;
    $editor.height = bitmap.height;
    $editorCtx.drawImage(bitmap, 0, 0);
    opacitor(257);
  });
});
```

Note:createImageBitmap() is an alternative to the fileReader() API. You can see immediately that createImageBitmap() is promise-based, which means there are background processes that decode the image. *Then*, .thenwaits for the decoding to complete to start drawing the image.
When Geo goes to test the code in the browser, he gets this:



No red bar! Those dashed lines indicate that there's a seperate thread in the background.Lo and behold:



Chrome actually schedules a worker to decode images off of the UI thread if you use createImageBitmap().

So, what's the takeaway here?

This isn't to evangelize createImageBitmap(), because it's relatively new and is unsurprisingly not supported by Internet Explorer. What you can take away from this is how fast you can feed information to your users if you offload taxing functions to a worker.

Workers are JavaScript's dirty little secret(they're not really a secret), so take advantage of them.

Bonus:

You probably noticed that the opacitor function contained a nested for loop. That's a O(n^2) level of trouble. Every pixel is traversed. With an image that's 2400x2000, that's a boat load of pixels to have to analyze.

Challenge: Why don't you try to refactor that code even further by using a worker?

A Note on Data "Transfer"

You shouldn't equate the flow of data from one file to another to the way a ball is passed from one athlete to another. The better analogy is the way a Fax machine works. Your document isn't sucked up by the machine and zipped to the recipient. Imagine how long it would take for them to receive that document. This data isn't shared, it's copied.

Can we have multiple files share a worker?

Yes, this is called a *shared worker*. The implementation differs from our *dedicated* worker example. It's not as widely accepted, but if you want to learn how to use them read Mozilla's great guide.

Session 8: JavaScript Developer Tools

Chrome DevTools

Source	https://developers.google.com/web/tools/chrome-devtools/
Date of Retrieval	02.03.2019

Chrome DevTools is a set of web developer tools built directly into the Google Chrome browser. DevTools can help you edit pages on-the-fly and diagnose problems quickly, which ultimately helps you build better websites, faster.
With DevTools you can view and change any page. Even the Google homepage, as the video demonstrates.

Open DevTools

There are many ways to open DevTools, because different users want quick access to different parts of the DevTools UI.

- When you want to work with the DOM or CSS, right-click an element on the page and select **Inspect** to jump into the **Elements** panel. Or press Command+Option+C (Mac) orControl+Shift+C (Windows, Linux, Chrome OS).
- When you want to see logged messages or run JavaScript, press Command+Option+J (Mac) or Control+Shift+J (Windows, Linux, Chrome OS) to jump straight into the **Console** panel.

See Open Chrome DevTools for more details and workflows.

DevTools for Beginners

DevTools for Beginners teaches you the **fundamentals of web development** as well as the basics of DevTools. Check out Get started below if you'd prefer tutorials that focus on DevTools.

- Get Started with HTML and the DOM
- Get Started with CSS

Get started

If you're a more experienced web developer, here are the recommended starting points for learning how DevTools can improve your productivity:

- View and Change a Page's Styles (CSS)
- Debug JavaScript
- View Messages and Run JavaScript in the Console
- Optimize Website Speed

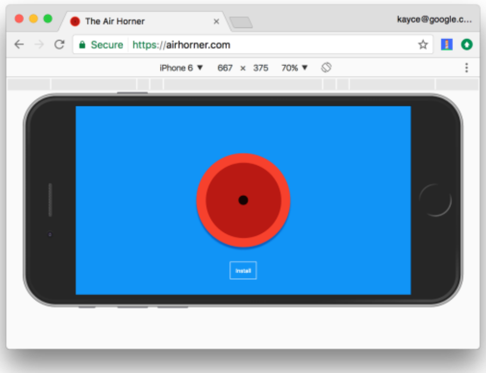
Discover DevTools

The DevTools UI can be a little overwhelming... there are so many tabs! But, if you take some time to get familiar with each tab to understand what's possible, you may discover that DevTools can seriously boost your productivity.

Device Mode

Simulate mobile devices.

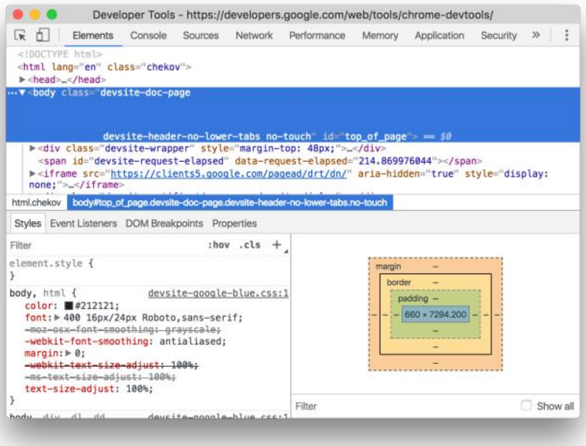
- Device Mode
- Test Responsive and Device-specific Viewports
- Emulate Sensors: Geolocation & Accelerometer



Elements panel

View and change the DOM and CSS.

- Get Started With Viewing And Changing CSS
- Inspect and Tweak Your Pages
- Edit Styles
- Edit the DOM
- Inspect Animations



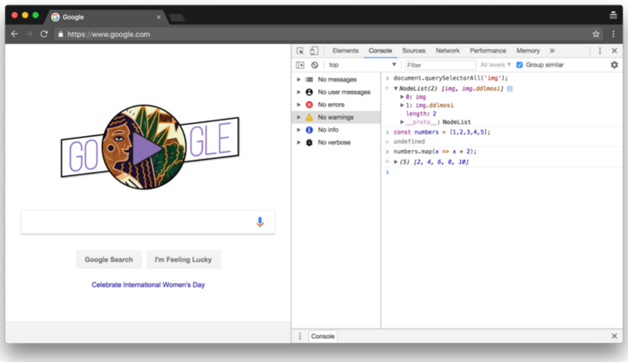
```
<!DOCTYPE html>
<html lang="en" class="chekov">
▶<head>…</head>
…▼<body class="devsite-doc-page
            devsite-header-no-lower-tabs no-touch" id="top_of_page"> == $0
  ▶<div class="devsite-wrapper" style="margin-top: 48px;">…</div>
   <span id="devsite-request-elapsed" data-request-elapsed="214.869976044"></span>
  ▶<iframe src="https://clients5.google.com/pagead/drt/dn/" aria-hidden="true" style="display:
  none;">…</iframe>
```

```
element.style {
}
body, html {                              devsite-google-blue.css:1
   color: ■#212121;
   font:▶ 400 16px/24px Roboto,sans-serif;
   ~~-moz-osx-font-smoothing: grayscale;~~
   -webkit-font-smoothing: antialiased;
   margin:▶ 0;
   ~~-webkit-text-size-adjust: 100%;~~
   ~~-ms-text-size-adjust: 100%;~~
   text-size-adjust: 100%;
}
```

Console panel

View messages and run JavaScript from the Console.

- Get Started With The Console
- Using the Console
- Interact from Command Line
- Console API Reference

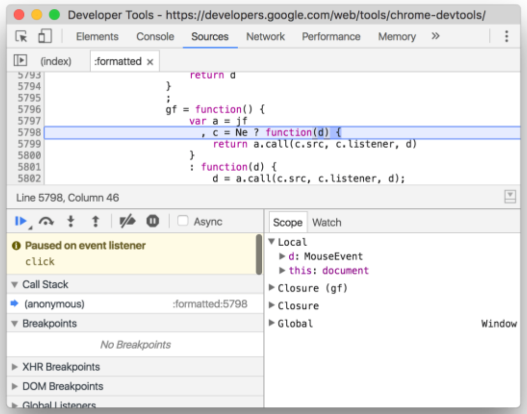


```
> document.querySelectorAll('img');
< ▼NodeList(2) [img, img.ddlmosi]
    ▶0: img
    ▶1: img.ddlmosi
     length: 2
    ▶__proto__: NodeList
> const numbers = [1,2,3,4,5];
< undefined
> numbers.map(x => x * 2);
< ▶(5) [2, 4, 6, 8, 10]
>
```

Sources panel

Debug JavaScript, persist changes made in DevTools across page reloads, save and run snippets of JavaScript, and save changes that you make in DevTools to disk.

- [Get Started With Debugging JavaScript]()
- [Pause Your Code With Breakpoints]()
- [Save Changes to Disk with Workspaces]()
- [Run Snippets Of Code From Any Page]()
- [JavaScript Debugging Reference]()
- [Persist Changes Across Page Reloads with Local Overrides]()


Developer Tools - https://developers.google.com/web/tools/chrome-devtools/
Elements Console Sources Network Performance Memory
(index) :formatted

```
                    return d
                }
                ;
                gf = function() {
                    var a = jf
                      , c = Ne ? function(d) {
                        return a.call(c.src, c.listener, d)
                    }
                    : function(d) {
                        d = a.call(c.src, c.listener, d);
```

Line 5798, Column 46

Paused on event listener
click

Call Stack
(anonymous) :formatted:5798

Breakpoints
No Breakpoints

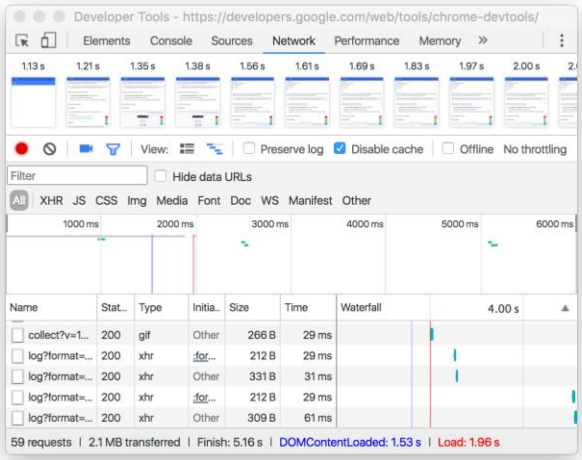
XHR Breakpoints
DOM Breakpoints
Global Listeners

Scope Watch
Local
d: MouseEvent
this: document
Closure (gf)
Closure
Global Window

Network panel

View and debug network activity.

- [Get Started]()
- [Network Issues Guide]()
- [Network Panel Reference]()


Developer Tools - https://developers.google.com/web/tools/chrome-devtools/
Elements Console Sources Network Performance Memory
1.13 s 1.21 s 1.35 s 1.38 s 1.56 s 1.61 s 1.69 s 1.83 s 1.97 s 2.00 s
View:
Preserve log
Disable cache
Offline
No throttling
Filter
Hide data URLs
All XHR JS CSS Img Media Font Doc WS Manifest Other
1000 ms 2000 ms 3000 ms 4000 ms 5000 ms 6000 ms

Name	Stat...	Type	Initia..	Size	Time	Waterfall 4.00 s
collect?v=1...	200	gif	Other	266 B	29 ms	
log?format=...	200	xhr	:for...	212 B	29 ms	
log?format=...	200	xhr	Other	331 B	31 ms	
log?format=...	200	xhr	:for...	212 B	29 ms	
log?format=...	200	xhr	Other	309 B	61 ms	

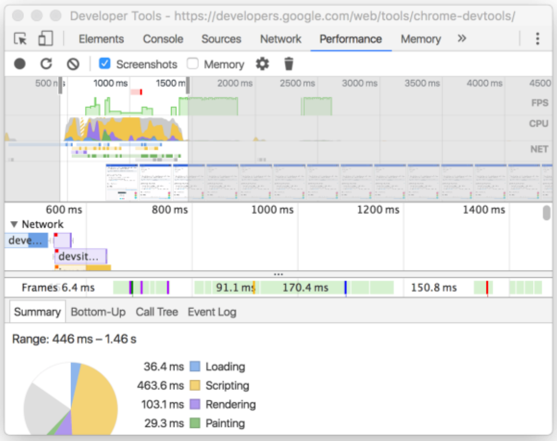
59 requests | 2.1 MB transferred | Finish: 5.16 s | DOMContentLoaded: 1.53 s | Load: 1.96 s

Performance panel

Note: *In Chrome 58 the Timeline panel was renamed to the Performance panel.*

Find ways to improve load and runtime performance.

- [Optimize Website Speed]()
- [Get Started With Analyzing Runtime Performance]()
- [Performance Analysis Reference]()
- [Analyze runtime performance]()
- [Diagnose Forced Synchronous Layouts]()


Developer Tools - https://developers.google.com/web/tools/chrome-devtools/
Elements Console Sources Network Performance Memory
Screenshots
Memory
500 ms 1000 ms 1500 ms 2000 ms 2500 ms 3000 ms 3500 ms 4000 ms 4500
FPS
CPU
NET
600 ms 800 ms 1000 ms 1200 ms 1400 ms
Network
deve...
devsit...
Frames 6.4 ms 91.1 ms 170.4 ms 150.8 ms
Summary Bottom-Up Call Tree Event Log

Range: 446 ms – 1.46 s

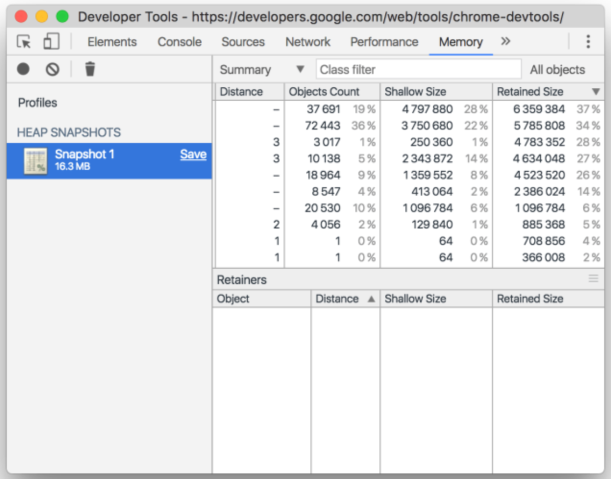
36.4 ms	Loading
463.6 ms	Scripting
103.1 ms	Rendering
29.3 ms	Painting

Memory panel

***Note:** In Chrome 58 the Profiles panel was renamed to the Memory panel.*

Profile memory usage and track down leaks.

- Fix Memory Problems
- JavaScript CPU Profiler


Developer Tools - https://developers.google.com/web/tools/chrome-devtools/
Elements Console Sources Network Performance Memory
Summary
Class filter
All objects
Profiles
HEAP SNAPSHOTS
Snapshot 1
16.3 MB
Save
Retainers

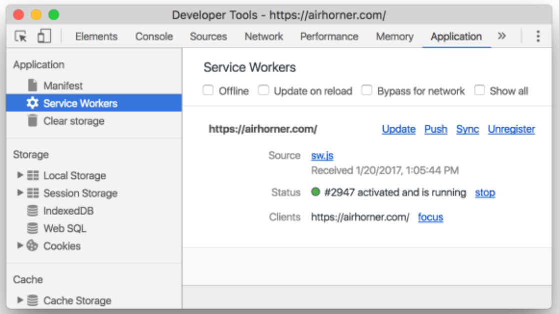
Distance	Objects Count		Shallow Size		Retained Size	
–	37 691	19 %	4 797 880	28 %	6 359 384	37 %
–	72 443	36 %	3 750 680	22 %	5 785 808	34 %
3	3 017	1 %	250 360	1 %	4 783 352	28 %
3	10 138	5 %	2 343 872	14 %	4 634 048	27 %
–	18 964	9 %	1 359 552	8 %	4 523 520	26 %
–	8 547	4 %	413 064	2 %	2 386 024	14 %
–	20 530	10 %	1 096 784	6 %	1 096 784	6 %
2	4 056	2 %	129 840	1 %	885 368	5 %
1	1	0 %	64	0 %	708 856	4 %
1	1	0 %	64	0 %	366 008	2 %

Object	Distance	Shallow Size	Retained Size

Application panel

Inspect all resources that are loaded, including IndexedDB or Web SQL databases, local and session storage, cookies, Application Cache, images, fonts, and stylesheets.

- Debug Progressive Web Apps
- Inspect and Manage Storage, Databases, and Caches
- Inspect and Delete Cookies
- Inspect Resources



https://airhorner.com/

Source sw.js
Received 1/20/2017, 1:05:44 PM

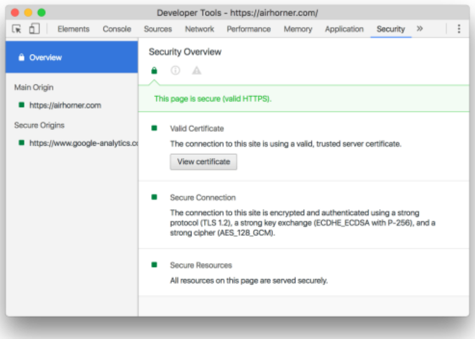
Status #2947 activated and is running stop

Clients https://airhorner.com/ focus

Security panel

Debug mixed content issues, certificate problems, and more.

- Understand Security Issues


Developer Tools - https://airhorner.com/
Elements Console Sources Network Performance Memory Application Security
Overview
Main Origin
https://airhorner.com
Secure Origins
https://www.google-analytics.co
Security Overview
View certificate

This page is secure (valid HTTPS).

Valid Certificate
The connection to this site is using a valid, trusted server certificate.

Secure Connection
The connection to this site is encrypted and authenticated using a strong protocol (TLS 1.2), a strong key exchange (ECDHE_ECDSA with P-256), and a strong cipher (AES_128_GCM).

Secure Resources
All resources on this page are served securely.

Session 9: Using Responsive APIs in JavaScript

Introduction to Web APIs

Source	https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction
Date of Retrieval	02.03.2019

What are APIs?

Application Programming Interfaces (APIs) are constructs made available in programming languages to allow developers to create complex functionality more easily. They abstract more complex code away from you, providing some easier syntax to use in its place.

As a real-world example, think about the electricity supply in your house, apartment, or other dwellings. If you want to use an appliance in your house, you simply plug it into a plug socket and it works. You don't try to wire it directly into the power supply — to do so would be really inefficient and, if you are not an electrician, difficult and dangerous to attempt.

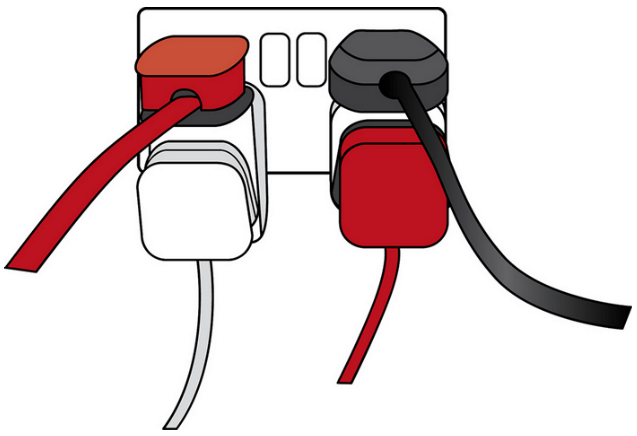


Image source: Overloaded plug socket by The Clear Communication People, on Flickr.

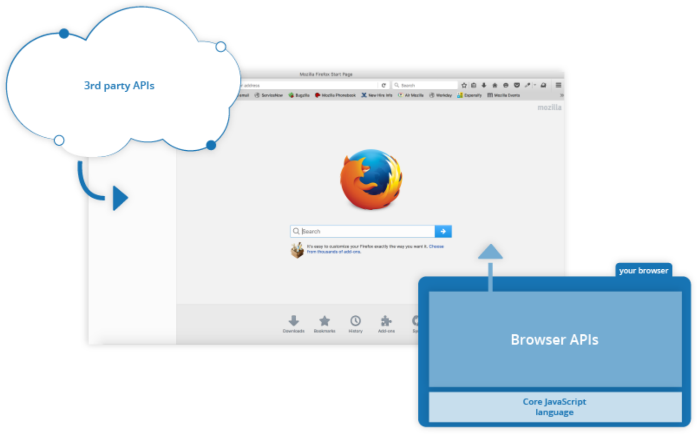
In the same way, if you want to say, program some 3D graphics, it is a lot easier to do it using an API written in a higher level language such as JavaScript or Python, rather than try to directly write low level code (say C or C++) that directly controls the computer's GPU or other graphics functions.

APIs in client-side JavaScript

Client-side JavaScript, in particular, has many APIs available to it — these are not part of the JavaScript language itself, rather they are built on top of the core JavaScript language, providing you with extra superpowers to use in your JavaScript code. They generally fall into two categories:

- **Browser APIs** are built into your web browser and are able to expose data from the browser and surrounding computer environment and do useful complex things with it. For example, the Web Audio API provides JavaScript constructs for manipulating audio in the browser — taking an audio track, altering its volume, applying effects to it, etc. In the background, the browser is actually using some complex lower-level code (e.g. C++ or Rust) to do the actual audio processing. But again, this complexity is abstracted away from you by the API.

- **Third party APIs** are not built into the browser by default, and you generally have to retrieve their code and information from somewhere on the Web. For example, the Twitter API allows you to do things like displaying your latest tweets on your website. It provides a special set of constructs you can use to query the Twitter service and return specific information.



Relationship between JavaScript, APIs, and other JavaScript tools

So above, we talked about what client-side JavaScript APIs are, and how they relate to the JavaScript language. Let's recap this to make it clearer, and also mention where other JavaScript tools fit in:

- JavaScript — A high-level scripting language built into browsers that allows you to implement functionality on web pages/apps. Note that JavaScript is also available in other programming environments, such as Node.
- Browser APIs — constructs built into the browser that sit on top of the JavaScript language and allow you to implement functionality more easily.
- Third party APIs — constructs built into third-party platforms (e.g. Twitter, Facebook) that allow you to use some of those platform's functionality in your own web pages (for example, display your latest Tweets on your web page).
- JavaScript libraries — Usually one or more JavaScript files containing custom functionsthat you can attach to your web page to speed up or enable writing common functionality. Examples include jQuery, Mootools and React.
- JavaScript frameworks — The next step up from libraries, JavaScript frameworks (e.g. Angular and Ember) tend to be packages of HTML, CSS, JavaScript, and other technologies that you install and then use to write an entire web application from scratch. The key

difference between a library and a framework is “Inversion of Control”. When calling a method from a library, the developer is in control. With a framework, the control is inverted: the framework calls the developer's code.

What can APIs do?

There are a huge number of APIs available in modern browsers that allow you to do a wide variety of things in your code. You can see this by taking a look at the MDN APIs index page.

Common browser APIs

In particular, the most common categories of browser APIs you'll use (and which we'll cover in this module in greater detail) are:

- **APIs for manipulating documents** loaded into the browser. The most obvious example is the DOM (Document Object Model) API, which allows you to manipulate HTML and CSS — creating, removing and changing HTML, dynamically applying new styles to your page, etc. Every time you see a popup window appear on a page, or some new content displayed, for example, that's the DOM in action. Find out more about these types of API in Manipulating documents.
- **APIs that fetch data from the server** to update small sections of a webpage on their own are very commonly used. This seemingly small detail has had a huge impact on the performance and behaviour of sites — if you just need to update a stock listing or list of available new stories, doing it instantly without having to reload the whole entire page from the server can make the site or app feel much more responsive and "snappy". APIs that make this possible include XMLHttpRequest and the Fetch API. You may also come across the term **Ajax**, which describes this technique. Find out more about such APIs in Fetching data from the server.
- **APIs for drawing and manipulating graphics** are now widely supported in browsers — the most popular ones are Canvas and WebGL, which allow you to programmatically update the pixel data contained in an HTML <canvas> element to create 2D and 3D scenes. For example, you might draw shapes such as rectangles or circles, import an image onto the canvas, and apply a filter to it such as sepia or grayscale using the Canvas API, or create a complex 3D scene with lighting and textures using WebGL. Such APIs are often combined with APIs for creating animation loops (such as window.requestAnimationFrame()) and others to make constantly updating scenes like cartoons and games.
- Audio and Video APIs like HTMLMediaElement, the Web Audio API, and WebRTCallow you to do really interesting things with multimedia such as creating custom UI controls for playing audio and video, displaying text tracks like captions and subtitles along with your videos, grabbing video from your web camera to be manipulated via a canvas (see above) or displayed on someone else's computer in a web conference, or adding effects to audio tracks (such as gain, distortion, panning, etc).
- **Device APIs** are basically APIs for manipulating and retrieving data from modern device hardware in a way that is useful for web apps. Examples include telling the user that a

useful update is available on a web app via system notifications (see the Notifications API) or vibration hardware (see the Vibration API).

- **Client-side storage APIs** are becoming a lot more widespread in web browsers — the ability to store data on the client-side is very useful if you want to create an app that will save its state between page loads, and perhaps even work when the device is offline. There are a number of options available, e.g. simple name/value storage with the Web Storage API, and more complex tabular data storage with the IndexedDB API.

Common third-party APIs

Third party APIs come in a large variety; some of the more popular ones that you are likely to make use of sooner or later are:

- The Twitter API, which allows you to do things like displaying your latest tweets on your website.
- Map APIs like Mapquest and the Google Maps API allows you to do all sorts of things with maps on your web pages.
- The Facebook suite of APIs enables you to use various parts of the Facebook ecosystem to benefit your app, for example by providing app login using Facebook login, accepting in-app payments, rolling out targetted ad campaigns, etc.
- The YouTube API, which allows you to embed YouTube videos on your site, search YouTube, build playlists, and more.
- The Twilio API, which provides a framework for building voice and video call functionality into your app, sending SMS/MMS from your apps, and more.

How do APIs work?

Different JavaScript APIs work in slightly different ways, but generally, they have common features and similar themes to how they work.

They are based on objects

Your code interacts with APIs using one or more JavaScript objects, which serve as containers for the data the API uses (contained in object properties), and the functionality the API makes available (contained in object methods).

***Note**: If you are not already familiar with how objects work, you should go back and work through our JavaScript objects module before continuing.*

Let's return to the example of the Web Audio API — this is a fairly complex API, which consists of a number of objects. The most obvious ones are:

- AudioContext, which represents an audio graph that can be used to manipulate audio playing inside the browser, and has a number of methods and properties available to manipulate that audio.
- MediaElementAudioSourceNode, which represents an <audio> element containing sound you want to play and manipulate inside the audio context.
- AudioDestinationNode, which represents the destination of the audio, i.e. the device on your computer that will actually output it — usually your speakers or headphones.
- So how do these objects interact? If you look at our simple web audio example (see it live also), you'll first see the following HTML:

```
•  <audio src="outfoxing.mp3"></audio>
•
•  <button class="paused">Play</button>
•  <br>
•  <input type="range" min="0" max="1" step="0.01" value="1"
   class="volume">
```

We first of all include an <audio> element with which we embed an MP3 into the page. We don't include any default browser controls. Next, we include a <button> that we'll use to play and stop the music, and an <input> element of type range, which we'll use to adjust the volume of the track while it's playing.

Next, let's look at the JavaScript for this example.
We start by creating an AudioContext instance inside which to manipulate our track:

```
const AudioContext = window.AudioContext || window.webkitAudioContext;
const audioCtx = new AudioContext();
```

Next, we create constants that store references to our <audio>, <button>, and <input>elements, and use the AudioContext.createMediaElementSource() method to create an MediaElementAudioSourceNode representing the source of our audio — the <audio>element it will be played from:

```
const audioElement = document.querySelector('audio');
const playBtn = document.querySelector('button');
const volumeSlider = document.querySelector('.volume');

const audioSource = audioCtx.createMediaElementSource(audioElement);
```

Next up we include a couple of event handlers that serve to toggle between play and pause when the button is pressed, and reset the display back to the beginning when the song has finished playing:

```
// play/pause audio
playBtn.addEventListener('click', function() {
    // check if context is in suspended state (autoplay policy)
    if (audioCtx.state === 'suspended') {
        audioCtx.resume();
    }

  // if track is stopped, play it
    if (this.getAttribute('class') === 'paused') {
        audioElement.play();
        this.setAttribute('class', 'playing');
        this.textContent = 'Pause'
    // if track is playing, stop it
} else if (this.getAttribute('class') === 'playing') {
        audioElement.pause();
        this.setAttribute('class', 'paused');
        this.textContent = 'Play';
    }
});

// if track ends
audioElement.addEventListener('ended', function() {
    playBtn.setAttribute('class', 'paused');
    this.textContent = 'Play'
});
```

Note: Some of you may notice that the play() and pause() methods being used to play and pause the track are not part of the Web Audio API; they are part of the HTMLMediaElement API, which is different but closely-related.

Next, we create a GainNode object using the AudioContext.createGain() method, which can be used to adjust the volume of audio fed through it, and create another event handler that changes the value of the audio graph's gain (volume) whenever the slider value is changed:

```
const gainNode = audioCtx.createGain();

volumeSlider.addEventListener('input', function() {
    gainNode.gain.value = this.value;
});
```

The final thing to do to get this to work is to connect the different nodes in the audio graph up, which is done using the AudioNode.connect() method available on every node type:

```
audioSource.connect(gainNode).connect(audioCtx.destination);
```

The audio starts in the source, which is then connected to the gain node so the audio's volume can be adjusted. The gain node is then connected to the destination node so the sound can be played on your computer (the AudioContext.destination property represents whatever is the default AudioDestinationNode available on your computer's hardware, e.g. your speakers).

They have recognizable entry points

When using an API, you should make sure you know where the entry point is for the API. In The Web Audio API, this is pretty simple — it is the AudioContext object, which needs to be used to do any audio manipulation whatsoever.

The Document Object Model (DOM) API also has a simple entry point — its features tend to be found hanging off the Document object, or an instance of an HTML element that you want to affect in some way, for example:

```
var em = document.createElement('em'); // create a new em element
var para = document.querySelector('p'); // reference an existing p element
em.textContent = 'Hello there!'; // give em some text content
para.appendChild(em); // embed em inside para
```

The Canvas API also relies on getting a context object to use to manipulate things, although in this case it's a graphical context rather than an audio context. Its context object is created by getting a reference to the <canvas> element you want to draw on, and then calling its HTMLCanvasElement.getContext() method:

```
var canvas = document.querySelector('canvas');
var ctx = canvas.getContext('2d');
```

Anything that we want to do to the canvas is then achieved by calling properties and methods of the content object (which is an instance of CanvasRenderingContext2D), for example:

```
Ball.prototype.draw = function() {
  ctx.beginPath();
  ctx.fillStyle = this.color;
  ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);
  ctx.fill();
};
```

Note: You can see this code in action in our bouncing balls demo (see it running live also).

They use events to handle changes in state

We already discussed events earlier on in the course in our Introduction to events article, which looks in detail at what client-side web events are and how they are used in your code. If you are not already familiar with how client-side web API events work, you should go and read this article first before continuing.
Some web APIs contain no events, but most contain at least a few. The handler properties that allow us to run functions when events fire are generally listed in our reference material in separate "Event handlers" sections.

We already saw a number of event handlers in use in our Web Audio API example above.

To provide another example, instances of the XMLHttpRequest object (each one represents an HTTP request to the server to retrieve a new resource of some kind) have a number of events available on them, for example the load event is fired when a response has been successfully returned containing the requested resource, and it is now available.
The following code provides a simple example of how this would be used:

```
var requestURL = 'https://mdn.github.io/learning-
area/javascript/oojs/json/superheroes.json';
var request = new XMLHttpRequest();
request.open('GET', requestURL);
request.responseType = 'json';
request.send();

request.onload = function() {
  var superHeroes = request.response;
  populateHeader(superHeroes);
  showHeroes(superHeroes);
}
```

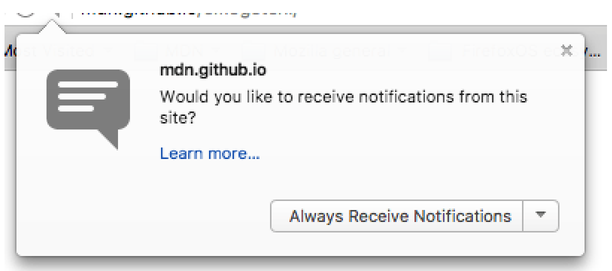
***Note**: You can see this code in action in our ajax.html example (see it live also).*

The first five lines specify the location of resource we want to fetch, create a new instance of a request object using the XMLHttpRequest() constructor, open an HTTP GET request to retrieve the specified resource, specify that the response should be sent in JSON format, then send the request. The onload handler function then specifies what we do with the response. We know the response will be successfully returned and available after the load event has required (unless an error occurred), so we save the response containing the returned JSON in the superHeroes variable, then pass it to two different functions for further processing.

They have additional security mechanisms where appropriate

WebAPI features are subject to the same security considerations as JavaScript and other web technologies (for example same-origin policy), but they sometimes have additional security mechanisms in place. For example, some of the more modern WebAPIs will only work on pages served over HTTPS due to them transmitting potentially sensitive data (examples include Service Workers and Push).

In addition, some WebAPIs request permission to be enabled from the user once calls to them are made in your code. As an example, the Notifications API asks for permission using a pop-up dialog box:


mdn.github.io

Would you like to receive notifications from this site?

Learn more...

The Web Audio and HTMLMediaElement APIs are subject to a security mechanism called autoplay policy — this basically means that you can't automatically play audio when a page loads — you've got to allow your users to initiate audio play through a control like a button. This is done because autoplaying audio is usually really annoying and we really shouldn't be subjecting our users to it.

***Note**: Depending on how strict the browser is, such security mechanisms might even stop the example from working locally, i.e. if you load the local example file in your browser instead of running it from a web server. At the time of writing, our Web Audio API example wouldn't work locally on Google Chrome — we had to upload it to GitHub before it would work.*

HTML5 Security Issues and General Guidelines

Communication API's

A Web browser provides security and privacy features to prevent documents of different domains to communicate with each other. There has to be a messaging system which will allow documents to communicate safely. Web Messaging API allows safe communication among documents of different domains in a Web browser.

Web Messaging

Web Messaging or Cross Domain Messaging is an API (Application Programming Interface). Web documents or Web pages of different origins or domains can communicate safely through a Web browser using the Web Messaging API.

Prior to HTML5, most of the Web browsers excluded the cross-site scripting technique to protect against security attacks. This particular practice denied communication between non-hostile pages as well as making interactions of any kind between documents difficult. Web messaging allows scripts to interact across these boundaries, while providing a simple level of security.

Requirements and Attributes

The Web Messaging API provides a postMessage method. Using which plain text messages can be sent from one domain to another very securely.

One basic requirement to send message from source document securely is to obtain the Window object of the receiving document. Messages can be posted to the following components of a document:

- Other frames or iframes within the sender document's window.
- A window that the sender document explicitly opens through JavaScript calls.
- The parent window of the sender document.
- The window which opened the sender document.

The messages are received using certain message attributes. The message 'event' received by the receiving document has the following attributes:

- Data - The actual content or the data of the incoming message.
- Origin - The origin of the sender document. This typically includes the scheme, hostname and port. It does not include the path or fragment identifier.
- Source - The WindowProxy of where the document came from. That is the source window.

Recommendations

Though, Web Messaging is a secure method establishing communication between documents of different domains, there are few guidelines that one should follow while using the Web Messaging feature which are as follows:

- A typical postMessage method is written as follows:
  postMessage(`‘Hello’, e.origin)`
  Here, the second parameter of the postMessage method should contain the value of the expected origin. The usage of * is invalid because messages from unknown origin is not accepted.
- Whenever a page receives a message, it should always check the ‘origin’ attribute of the sender.
- This validation verifies that the data is originating from the expected location or not.
- The second operation that must be always performed is input validation of the ‘data’ attribute of the event. An input check ensures that the received data is in the desired format.
- Extra care must be taken about the ‘data’ attribute received. As a single error made by the author of the sending page in Cross Site Scripting will allow a hacker to send messages of any given format.
- It is important that the messages exchanged between pages are interpreted as ‘Data’ only and not as ‘Code’. Interpretation as ‘Code’ leads to DOM-based XSS vulnerabilities.
- Whenever it is necessary to assign a data value to an element, it is preferable to use a safer option like element.textContent = data rather than using an insecure method like element.innerHTML = data which can lead to vulnerability.
- It is always a good practice to check the ‘origin’ attribute of the message being received properly so that it exactly matches the expected Fully Qualified Domain Name (FQDN).
- An external content or an untrusted gadget or a user-controlled script can be embedded using JavaScript rewriting framework such as ‘Google caja’.

Cross Origin Resource Sharing

Cross Origin Resource Sharing (CORS) is a mechanism that allows JavaScript on a Web page to make an XMLHttpRequests to another domain and not the domain in which the JavaScript originated actually. CORS defines a way in which the browser and the server can interact with each other to determine whether or not to allow the cross-origin request.

Whenever a URL is being passed to the XMLHttpRequest.open method, it is a good practice to validate such URLs, as current browsers allow these URLs to be cross domain and this behaviour can lead to code injection by a remote attacker. It is always preferred to pay some extra attention to absolute URLs to avoid any code injection.

In cases where a URL is responding using Access-Control-Allow-Origin: *, where the symbol * represents the wildcard characters, that allows any origin to access the resource. It is better to ensure that these URLs do not include any sensitive content or information that gets exposed, which can aid an attacker to attack. A URL that needs to be accessed cross- domain must use the Access-Control-Allow-Origin header. It is always preferable to allow only selected, trusted domains in the Access-Control-Allow-Origin header by whitelisting of domains over blacklisting or allowing any domain to access the resource of that site. Whenever a data is being requested, CORS never prevents the requested data from going to an unauthenticated location, so it is very much important for the server to perform the usual Cross-Site Request Forgery (CSRF) prevention. As the current implementations might not perform this, the RFC recommends pre-flight request with the OPTIONS verb. It is very much important that ’ordinary’ requests such as GET and POST perform any access control that is necessary. To prevent mixed content bugs, it is always advised to discard the requests received over plain HTTP with HTTPS origins. If there are some sensitive data, then an Application-level protocol should be used to protect such data than relying only on the Origin header for Access Control checks. It is better to use an Application-level protocol as a browser always sends the Origin header in CORS requests, but this header can be spoofed outside the browser exposing the sensitive data to attackers.

Web Sockets

A Web Socket is a protocol that provides a full-duplex communications channel over a single TCP connection through which messages can be sent between client and server. Its standard simplifies much of the complexity around bi-directional Web communications and connection management. It is designed to be implemented in Web browsers and Web servers, but it can be used by any client or server application.

The latest Web Socket protocol version supported in latest versions of all current browsers is RFC 6455 (supported by Firefox 11+, Chrome 16+, Safari 6, Opera 12.50 and IE10).

The TCP service tunnelling is easy using Web sockets such as File Transfer Protocol (FTP), however, this will enable a CSS attacker to get access to all the tunnelled services.
Authorization and/or authentication must be handled separately using Application-level protocols as the WebSocket protocols cannot handle these, in case any sensitive data is being transferred. Whenever a message is received by the WebSocket, it must always be processed as 'data' than as a 'code'. In cases when the response is JSON, it is not preferable to use the insecure eval() function, rather use the safer JSON.parse() method instead.

The clients are vulnerable to Spoofing outside a browser. It is the job of the WebSocket server to validate the incoming input coming from a remote site.

At the server, there should be validation of the Origin header, as in a WebSocket connection, the Origin header will have information related to the origin of the page that initiated the WebSocket connection. Even if the message is spoofed, its origin can be found out using the Origin header information.

WebSocketValidation of data coming through a WebSocket connection is mandatory as WebSocket clients are vulnerable to JavaScript calls by which the communication can be spoofed or hijacked through Cross Site Scripting.

Server–Sent Events

Server-Sent Events (SSE) is a technology where a browser gets automatic updates from a server via HTTP connection. The SSE EventSource API is standardised as part of HTML5 by the W3C. SSE is a standard describing how servers can initiate data transmission towards clients once an initial client connection has been established. They are commonly used to send message updates or continuous data streams to a browser client and designed to enhance native, cross-browser streaming through a JavaScript API called EventSource.
Here are a few guidelines to use SSE in an efficient manner:

- Whenever URLs get passed to the EventSource Constructor, it is preferable to validate these URLs, though same-origin URLs are allowed.
- Messages (event.data) must always be processed as data and never be evaluated as HTML or script code.
- Validate the origin attribute of the message (event.origin) to ensure trusted domain messages to reach client.
- Use a whitelist approach over the blacklisting approach.

Storage API's

There are several storage techniques that allow safety of the data stored.

Local Storage

Local storage is also known as Offline Storage or Web Storage. There are several reasons to avoid local storage of sensitive information. They are as follows:

- An user who has privileges over a local machine can bypass an authentication process; this might be dangerous as sensitive information stored on the local machine might become vulnerable.
- In cases where persistent storage is not needed, it is preferable to use the object sessionStorage instead of localStorage as a sessionStorage object is available till the expiry of window/tab session.
- A single Cross Site Scripting can steal all the data.
- Objects in a local storage cannot be trusted as a single Cross Site Scripting can load malicious data into these objects.
- Cautious use of 'localStorage.getItem' and 'setItem' in HTML5 page will help developers to detect whether sensitive information is being stored into the local storage.
- It is always advisable not to store session identifiers in local storage as the data is always accessible by JavaScript calls. Further, the Cookies avoid this risk by using 'httpOnly' flag.

Client-side Databases

W3C has made Web SQL Database or relational SQL database as outdated on November 2010. Nowadays, a new key-value pair based Indexed database is used. It provides rich database storage and querying techniques. Sensitive information is not recommended to be stored locally. Web Database contents must be properly validated and parameterised on the client-side as they are vulnerable to SQL injection easily. Web databases cannot be trusted as a single Cross Site Scripting can be used to load malicious data into these objects.

Geolocation

The HTML5 Geolocation API is used to get the geographical position of a user. The Geolocation RFC takes permission from the user before the location information is fetched. The browser properties decide the storage of the location parameters. The getCurrentPosition or watchPosition methods are initiated after taking the user's permission. Location information is fetched without permission only when the user turns on the ability.

Web Workers

A Web worker, as defined by the World Wide Web Consortium (W3C) and Web Hypertext Application Technology Working Group (WHATWG), is a JavaScript executed from an HTML page that runs in the background, independently of other user-interface scripts that may also have been executed from the same HTML page.

In order to perform in-domain and Cross Origin Resource Sharing requests, Web workers are allowed to use the XMLHttpRequest object as this object can be used to exchange data with a server as a background process.

It is always a good practice to not allow Web workers scripts to operate from user supplied. Malicious Web Workers can use excessive CPU for computation, leading to Denial of Service condition or abuse Cross Origin Resource Sharing for further exploitation.

Web workers should validate the data that is getting exchanged and the data should not include code as this may lead to hacking.

Sandboxed Frames

Sandbox provides options to restrict plug-ins, forms, scripts and links that are from malicious sites. The Sandbox attribute of an iframe is used when the data is from untrusted source.

When a sandbox attribute is set for an iframe, the following restrictions are activated:

- If any markup is being used, then all these markups are treated as being from a unique origin.
- If the Web page uses forms and scripts, then all these forms and scripts get disabled.
- The links available on the Web page are prevented from targeting other browsing contexts.
- Automatic triggering of features is blocked.
- Any plug-ins of the Web page is disabled.

The value of the 'sandbox' attribute is used as an additional protection layer and this allows the frame to display the untrusted data only if this feature is supported by the browser. Apart from the 'sandbox' attribute, the header X-Frame-Option can be used, which supports deny and same-origin values to prevent 'ClickJacking' attacks and unsolicited framing.

Offline Applications

Offline application is a list of URLs for HTML, CSS, JavaScript, images or any other kind of resource. In an offline application, the home page points to the list of resources which is known as a 'manifest' file. A Web browser that implements HTML5 offline applications will read this list of URLs from the manifest file, download the resources, cache them locally and automatically keep the local copies up to date as they change.
When the user tries to access the Web application without a network connection, the Web browser will automatically switch over to the local copies instead.

As the resources gets cached locally, it very much varies from one browser to another over whether the user agent requests permission to the user to store data for offline browsing and when this cache gets deleted.

Before sending any manifest file, for privacy reasons, it is always encouraged to require user inputs as Cache poisoning is an issue if the user connects through insecure networks.
It is always a good practice to only cache the resources from trusted Web sites and to clean the cache after browsing through any open or insecure networks.

HTTP Headers to Enhance Security

There are several HTTP headers to provide security to the Web pages. They are as follows:
Frame-Options
This feature of the HTTP header field indicates a policy that specifies whether the browser should render the transmitted resource within a <frame> or an <iframe>. The X-Frame- Options can be used to prevent 'ClickJacking' in modern browsers, which ensure that their content is not embedded into other pages or frames. A user can frame a content from same origin using the same-origin attribute or block the content using the deny attribute. For example, the code

`X-Frame-Options`: DENY will block the content.

SS-Protection
This feature is supported by Internet Explorer 8 and newer versions. This header allows the domain toggle on and off the 'XSS Filter' of IE8, which prevents few categories of XSS attacks. The 'XSS Filter' has been activated by default, but the servers can switch it off using the Settings option. Whenever the XSS filter is enabled, it only works for Reflected XSS. For example, `X-XSS-Protection: 1; mode=block`.

Strict Transport Security
It is a security policy that allows browsers to communicate using HTTPS type of connections. Once a supported browser receives this header, the browser will prevent any communications from being sent over HTTP to the specified domain and will instead send all communications over HTTPS. It disallows HTTPS click through prompts on browsers. The main use of Strict Transport Security is that it forces every browser request to be sent over TLS/SSL, which will prevent SSL strip attacks. Using the directive includeSubDomains, the Web application can protect its '*domain cookie*' in an adequate way. For example, the code `Strict-Transport-Security: max-age=8640000`; includes SubDomains.

Content Security Policy (CSP)
It is a security policy that prevents Cross Site Scripting (XSS) and related attacks. CSP supports a standard HTTP header that allows a browser to authenticate sources of content. Here is a list of approved sources of content:

- JavaScript
- CSS
- HTML frames
- Fonts
- Images
- Embeddable objects such as Java applets, ActiveX
- Audio and video files

A sample code is as follows:
```
X-Content-Security-Policy: allow 'self';
img-src *;
object-src media.example.com;
script-src js.example.com;
```

Origin
The Origin header is added by the user agent to describe the security contexts that initiated an HTTP request. HTTP servers can use the Origin header to weaken Cross-Site Request Forgery (CSRF) vulnerabilities. The Origin header is sent when a CORS/WebSockets requests are made.

HTML should be used carefully to develop interactive Web sites. The potential attacks on the Web are majorly due to cross-origin interactions. Let us see the types of attacks on a site created using HTML.

SQL Injection

It is a technique which passes SQL commands to a backend database through an input to a Web application. It is very much necessary to validate untrusted inputs such as messages in a text box, values passed through the URL address or messages from third party sites. A failure in the input validation process can lead to variety of attacks. Some of them can be very serious leading to execution of scripts or code that may cause serious defects such as deleting all data in a server.
Filters should be written to validate the user input. A Whitelist-based filter which allows known origin inputs barring all others is better than Blacklist-based filters that disallow known-origin bad inputs and allows all others.
Whitelist filters can be written using options as follows:

- Image files used in the img tag can be whitelisted along with all the attributes. This might cause an attacker to use the onload attribute to run a script or code.
- The scheme to use URL's should be whitelisted.
- Care must be taken while using the base element as they are vulnerable to attacks.

Now, let us see the working of a Cross Site Scripting type of attack.

For example, suppose a page used the URL's query string to determine the display parameters and then redirected the user to a page to display a message, as follows:

```
<ul>
<li><a href=”message.cgi?say=Hi”>Say Hi</a>
<li><a    href=”message.cgi?say=Are     you  there?”>Say    Are
     you
there?</a>
<li><a href=”message.cgi?say=Welcome”>Say Welcome</a>
</ul>
```

An invalidated message would allow an attacker to misuse this URL and include a malicious script as follows:
http://example.com/message.cgi?say=%3Cscript%3Ealert%28%27Oh%20no%21%27%29%3C/script%3E

Such scripts can do any type of adverse attacks; it can even make a vulnerable user to visit an unintended site and perform unintentional actions. This is known as a Cross Site Scripting attack.

Cross-site Request Forgery (CSRF)

In this type of attack, a Web site may be tricked by unauthorised commands coming from an authorised user of the site. A site may allow a user to submit forms with messages; such data coming from the user should be validated by the site before it is processed, as there are chances of another site during the user to make unintentional requests to the site unknowingly. This type of problem prevails because a HTML form can be submitted to other origins. In such cases, sites can prevent such attacks by populating forms with user- specific hidden tokens or by checking Origin headers on all requests.

ClickJacking

This type of attack forces a Web user to click a malicious link and thus, become a victim. When a Web page provides the users with an interface to perform actions, there are certain actions that the user might not wish to perform, in such cases the page should be designed in such a way that it avoids the possibility that the users can be tricked into activating an interface.
For example, a user could be tricked if a hostile site places the victim site in a small iframe and then convinces the user to click, for instance, the user play a reaction game. Once the user is playing the game, the hostile site can quickly position the iframe under the mouse cursor just as the user is about to click, thus tricking the user into clicking the victim site's interface.
One way to avoid this type of attack is that sites are expected to be used in frames, are encouraged to only enable their interface if they detect that they are not in a frame.

Common Pit alls to Avoid When Using the Scripting APIs

HTML scripts run-to-completion before further action is initiated. Alternatively, HTML parsing can pause to allow the scripts to run which may cause an event to be triggered. The methods to allow events to trigger safely are as follows:

- Use event handler content attributes
- Create the element and add the event handlers in the same script later

The second method is safe as the event is triggered after parsing the HTML code.

For example, when load event is used after the img element, load event will be generated as soon as the element has been parsed, especially if the image has already been cached which is common in most of the scenarios. Here, the onload handler is used on an img element to catch the load event. An example code is as follows:

```
<img src=”games.png” alt=”Games”
onload=”gamesLogoHasLoaded(event)”>
```

If the element is being added using a script, then so long as the event handlers are added in the same script, the event will still not be missed:

```
<script>
var img = new Image();
img.src = 'games.png';
img.alt = 'Games';
img.onload = gamesLogoHasLoaded;
</script>
```

JavaScript Security Guidelines

JavaScript Security Guidelines

JavaScript is a dynamic scripting language that can be used on both, client and server machine. On client machine, it can be used to collect information from the client or the user, change browser characteristics and interact asynchronously to name a few. On the server- side, it can be used to perform server side programming. Web pages permitting JavaScript in them are vulnerable to attacks as they allow executable content to be embedded in them. Such scripts can be malicious and can damage valuable data of a user's computer or invade into user's private information. Hence, it is very much essential that JavaScript security guidelines are followed.

Client-side Security Guidelines

A client computer may contain a lot of valuable data, as a precautionary measure, JavaScript does not support file access capabilities and hence cannot delete user's data or even host viruses in the system. Further, JavaScript do not have networking capabilities. They cannot connect to another computer and send messages on a distributed environment. All of these measures ensure safe use of JavaScript, however, there are components such as ActiveX controls or plug-ins which have the file and networking capabilities, also they can be easily controlled by JavaScript. This facility might be misused by hackers. There are other security issues such as transfer of user's Web browser information and browsing history which can be accessed by a JavaScript. Having client-side security is mandatory. Here is a list of options that can be used in JavaScript as a security measure:

- ***Use .innerText instead of .innerHtml***

  It is safe to use **innerText** over **innerHTML** as the use of **innerText** will prevent most of the XSS related problems as they have the capacity to encode the text. It is advisable to use **innerHtml** while displaying HTML.

- ***Avoid use of eval()***

  The eval() function can execute code or script, hence it should be cautiously used.

- ***Canonicalise data to consumer (read: encode before use)***

  Any data used in HTML or a scripting language should be verified for its logical meaning. Data should be encoded before it is used to prevent injection issues and retain its logical meaning.

- ***Client security logic alone is not sufficient***

  Browser components such as ActiveX and plug-ins can use JavaScript to set breakpoints, execute malicious code, change values and so on. Hence, dependence on client logic alone is not enough.

- ***Secure client business logic***

  It is necessary that any client business logic or rules should be duplicated on a server for security reasons.

- ***Avoid writing serialization code***

  Code written using serialization concept is difficult to develop and maintain, hence avoid it.

- ***Avoid building XML dynamically***

  JavaScript can introduce XML injection if XML code is built dynamically. It is an attack technique that injects unintended or malicious XML content and/or structures into an XML message that can alter the intend logic of the application.

- ***Avoid storing secret information on client***

  It is not safe to keep secure information on the client machine as it may be used by different users, hence keep valuable information on the server.

- ***Avoid encryption of client side code***

  It is a good practice to use TSL/SSL and encrypt the data on the server than encrypting the data at the client side.

- ***Do not perform security impacting logic on client side***

  Client-side authorization and authentication checks are not safe as a user can bypass the checks by disabling the JavaScript; instead, perform such checks on the server-side for security reasons.

Server-side Security Guidelines

The communication between client and server machine is handled by JavaScript Object Notation (JSON), which is a lightweight data-interchange format. JSON is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition – December 1999. JavaScript or JSON can be used with malicious intentions to access server-side data and functionality from an authenticated client browser. JavaScript hijacking is the term used when JavaScript is used with malicious intentions. It is necessary that security guidelines are followed at the server side as well. Here is a list of security guidelines against JSON or JavaScript Hijacking:

- ***Use CSRF protection***

  Cross-site request forgery (CSRF) is a type of attack which a malicious user can use to access server information from an authenticated client machine. A CSRF protection adds a CSRF token as a hidden field for forms or within the URL of the client, the server rejects the requested action if the CSRF token failsvalidation.

- ***Always return JSON with an Object on the outside***

  While accessing data using JSON, avoid arrays as the top level element. This can be avoided by placing a filter to check for JSON responses where the first non- whitespace element is a '['.

  Always have the outside primitive be an object for JSON strings.

- ***For example, an exploitable code is as follows:***

  ```
  [{"object": "inside an array"}]
  ```

- ***The code can be made non-exploitable as follows:***

  ```
  {"object": "not inside an array"}
  ```

  or

  ```
  {"result": [{"object": "inside an array"}]}
  ```

- ***Services can be called by users directly***

  A client code can call services on the server, similarly, the user using the client machine can do the same. Untrusted messages coming from the client machine cannot be trusted, hence validate all the incoming requests before processing it on the server.

- ***Avoid building XML by hand, use the framework***

  Building XML code by hand can lead to issues such as XML injections; this can be avoided by using the facilities of a framework.

- ***Avoid building JSON by hand, use an existing framework***

  On similar lines to XML, it is better to use the framework facilities to build JSON.

~~ End of References~~