NOT THE OLD MUSTY INTERNET!

BREAKTHROUGH JAVASCRIPT

MODERN & ADVANCED JAVASCRIPT

gps | webcam | speech recognition
indexed database | websockets
workers | push notifications | more

A BOOK BY W.S. TOH

BREAKTHROUGH JAVASCRIPT

Find us on the web at https://code-boxx.com

Notice of Rights

Trademarks

Notice of Liability

FOREWORD

Thank you for buying this book! Once upon a time in the Cyber World, Javascript is but a humble programming language that works silently behind web pages. Simple, robust, and easy to learn. That was Javascript in the early days.

But as the Internet grew with the rise of mobile devices, things became increasingly complicated. The demand for complex operations and functional capabilities changed. People started to brand Javascript as "slow", "useless", and "too simplistic".

Little did these people know, Javascript developers did not just sit down and watch things burn down to the ground. Even wonder why Javascript is one of the world's most popular programming languages? It is still an easy language for beginners to learn, but it is also disgustingly capable.

If you dig deep enough, Javascript can actually do a plethora of "unthinkable" things – Accessing the GPS coordinates, reading the gyroscope, creating a local database, asynchronous processing, threading, voice recognition, taking photos, real-time systems, etc...

So congratulations on getting this book, let us explore some of the "unthinkable side" of Javascript where many did not even thread on. Take a long good laugh at the ignorant folks who still think Javascript is going obsolete.

W.S. Toh

Founder, Code Boxx

SOFTWARE LICENSE

TABLE OF CONTENTS

INTRODUCTION

ASYNCHRONOUS & BACKGROUND PROCESSING

DATA STORAGE & RETRIEVAL

PUSH & PULL

MEDIA

MOBILE & SENSORS

OTHERS

- CHAPTER A -

PRELUDE & OVERVIEW

INTENDED AUDIENCE

Here we go, starting with the obligatory "boring expectation management". But this is quite important, this book is to be considered "rather advanced" and written for the more experienced developers.

If you have never heard of things like OOP, AJAX, asynchronous, parallel processing, JSON, SSL, and API – It will be a difficult struggle trying to understand this book.

WARNING – EXPERIMENTAL TECHNOLOGY

Some chapters in this book use experimental technologies. Yes, technologies that are still in their "working draft" status at the time of writing. Don't expect 100% support across all browsers and systems. Also, be prepared that the standards and procedures may change as the technology matures.

SERVER-SIDE REQUIREMENTS

- **NodeJS** – For the uninitiated, this is server-side Javascript. Download, install, and get started with Node if you have not already done so.
- **HTTP Server** – Apache, Nginx, IIS, simple HTTP server with NodeJS, or even the Chrome Web Server extension. Not going to restrict what you use, just set up your own.

CLIENT-SIDE REQUIREMENTS

- **Grade A Modern Browser** – Chrome, Firefox, Opera, or Edge (the later Webkit versions). Sorry to the Apple fans, but Safari is considered "grade B", some things may not work correctly.
- **Mobile Device** – It is best to use a smartphone or tablet for some of the mobile-related chapters. At least use an emulator.

NO BACKWARD COMPATIBILITY CHECKS & FALLBACK

To keep things simple, all examples in this book will not do any backward compatibility checks. For example, `if ("serviceWorker" in navigator) { … }`. If you have to support legacy browsers, please do your own checks, implement fallback, and even use a polyfill if necessary – A library I will recommend is Modernizr.

SECURE ORIGINS – HTTPS://

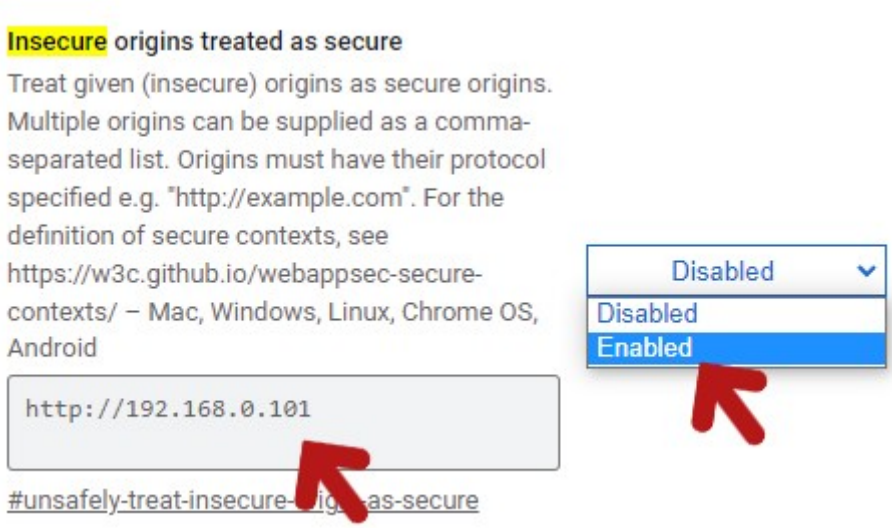
The good but irritating part regarding security. Some features such as the microphone and webcam require the page to be accessed with `https://` to work properly. We call this a "secure origin check". If you have already setup your own SSL certificate, well done, there is no need to sweat. If you have not, the options are:

- Stick with using `http://localhost` for development. This is the only exception that browsers will skip secure origins checks and allow all features for testing.
- Setup your own self-signed SSL cert. It's out of topic for this book

though, you will need to do your own research.

- Override the browser's security policy, skip `https://` checks for certain domains. See below.

OVERRIDING THE SECURITY POLICY



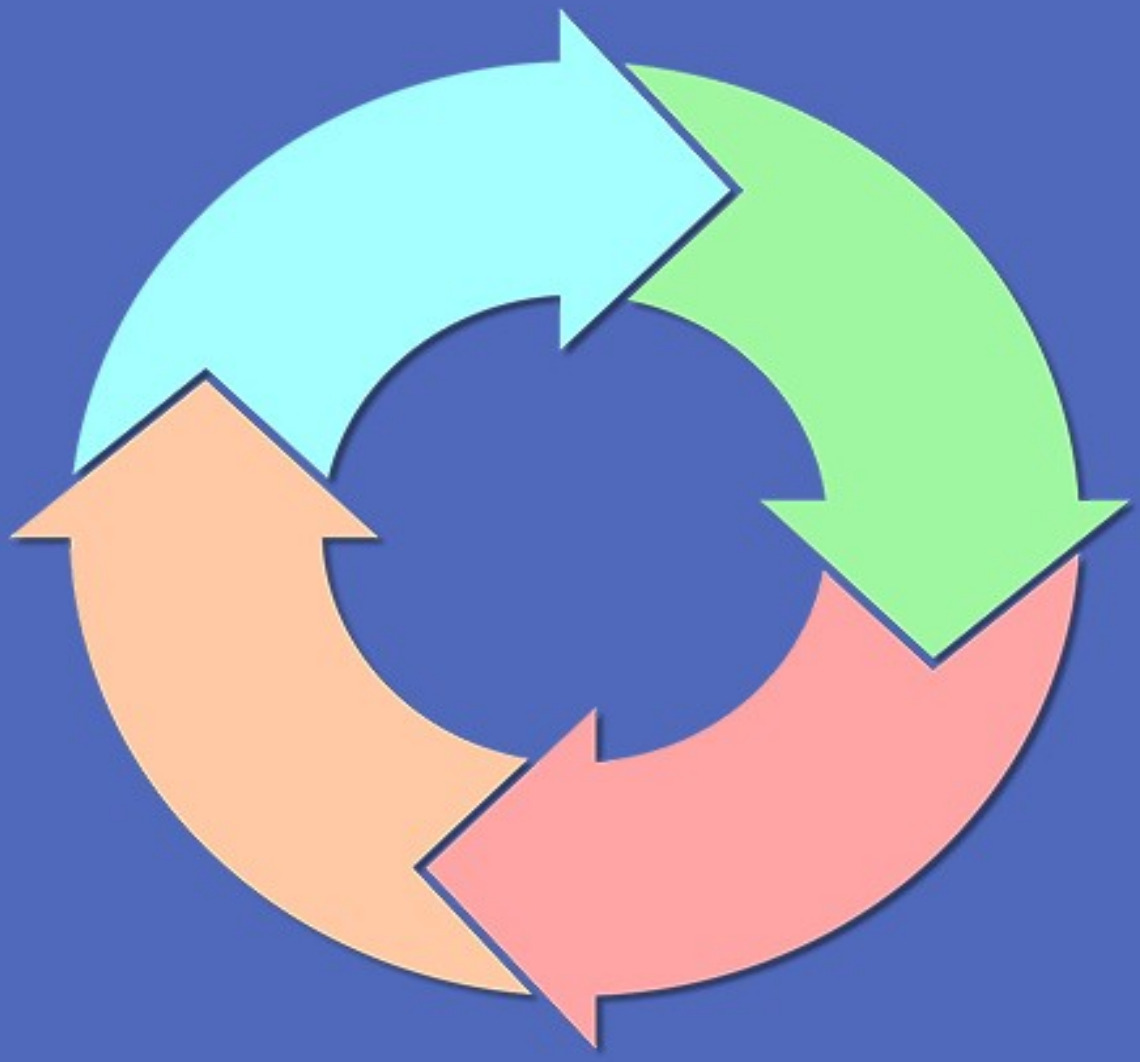
This is the lazy way to not deal with an SSL cert, but only works on Chromium-based browsers.

- Open `chrome://flags OR opera://flags OR edge://flags`.
- Search for "Insecure origins".
- Under "Insecure origins treated as secure", enter your server's address, then enable it.

Done. The browser will skip secure origin checks, and allow access to the microphone, webcam, GPS, or whatever else features.

- CHAPTER B -

ASYNC & PROMISES



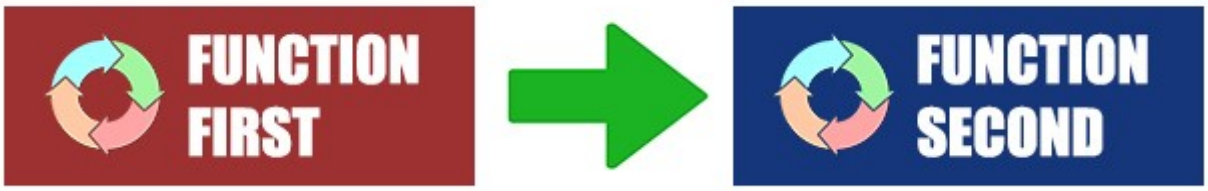
SIMPLE START

For this first "serious chapter", let us touch on asynchronous functions. This is an easy one to help get some of you guys a little more up to speed with modern Javascript. Feel free to skip this chapter if you already know.

"TRADITIONAL" FUNCTIONS ARE SYNCHRONOUS

CHAPTER-B/1-SYNC-FN.HTML

```
// (A) SYNC FUNCTIONS
function first (a, b) { return a + b; }
function second (a, b) { return a * b; }

// (B) RUN!
var foo = first(2, 3),
    bar = second(2, 3);
console.log(foo); // 5
console.log(bar); // 6
```



This is a piece of cake that everyone should already know – Javascript functions and processes are synchronous by default. That is, one process must end before the next one can start, the next function cannot start until the current one has ended.

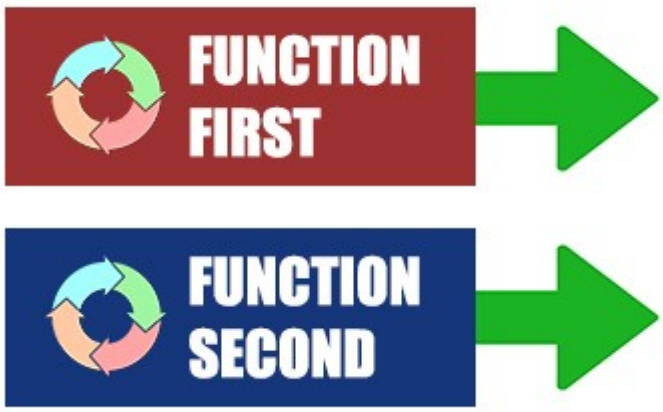
This works fine by all means, but it causes a problem known as "blocking". That is, consider this example itself – `first()` must be

completed before `second()` can run. If `first()` is a massive function, users will be staring at a "frozen screen" for a long time. This is why asynchronous functions and parallel processing are introduced in modern Javascript.

ASYNCHRONOUS FUNCTIONS & PROMISES

CHAPTER-B/2-ASYNC-FN.HTML

```
// (A) ASYNC FUNCTIONS
async function first (a, b) { return a + b; }
async function second (a, b) { return a * b; }

// (B) RUN!
var foo = first(2, 3),
    bar = second(2, 3);
console.log(foo); // PROMISE
console.log(bar); // PROMISE

// (C) GET RESULT
foo.then((result) => { console.log(result); }); // 5
bar.then((result) => { console.log(result); }); // 6
```



Defining an asynchronous function in Javascript is as simple as adding `async` in front of `function` – These functions will now run

independently instead of waiting for one another to complete. While this may sound easy enough, quite a number of people will definitely trip on the mechanism called `promise`.

- Notice how `foo = first(2, 3)` and `bar = second(2, 3)` returns a `promise`?
- That is because we don't wait for asynchronous functions to finish processing. Javascript will not return the processed results immediately, but gives a `promise` instead – "I promise to get back to you with the results when it is ready".
- To resolve the results, we use `promise.then((results) => { DO SOMETHING })`. That is, `then()` will only be triggered when the function is done processing.

ARROW FUNCTIONS

So far so good? Let us go off topic a little bit. Notice the `(results) => { DO SOMETHING }`? This is called an "arrow function", a "shorthand" way to define functions. All right, let us go back to defining a "traditional" function first:

CHAPTER-B/3-ARROW-FN.HTML

```
// (A) "TRADITIONAL WAY"
function demo (a, b) { return a + b; }
```

In modern Javascript, we can simplify that to:

CHAPTER-B/3-ARROW-FN.HTML

```
// (B) ARROW FUNCTION
```

```
demo = (a, b) => { return a + b; };
console.log(demo(1, 2)); // 3
```

If it is a "one line return function" as above, we can further simplify:

CHAPTER-B/3-ARROW-FN.HTML

```
// (C) FURTHER SIMPLIFICATION
demo = (a, b) => a + b;
console.log(demo(3, 4)); // 7
```

Very handy. But take note of legacy support, older browsers don't understand the arrow operator `=>`.

AWAIT

CHAPTER-B/4-AWAIT.HTML

```
// (A) ASYNC FUNCTIONS
async function first (a, b) { return a + b; }
async function second (a, b) { return a * b; }

// (B) RUN!
async function third () {
  var result = await first(1, 2);
  result += await second(3, 4);
  console.log(result); // 15
}
third();
```

Lastly, there just times where we have to use multiple `async function`. It will be super annoying to chain `then()`, so `await` is introduced.

What it does should be self-explanatory, wait for an `async function` to return the results. But take note though, `await` can only be used

inside another `async function`.

LINKS & REFERENCES

- Async Function – MDN
- Promise – MDN
- Await – MDN
- Arrow Function – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Async Functions
- Arrow Function

- CHAPTER C -

JAVASCRIPT WORKERS

THREADING MAGIC

For you guys who are slightly newer to computer science:

- Asynchronous is still single threaded.
- To achieve multi-threading, we have to assign tasks to what we call "workers".

We shall not go out-of-point in this book, do your own homework if you are interested in learning more about threading and processes. But to keep things simple – Use "workers" if you want "true background processing".

SIMPLE WEB WORKER

So how does this "worker" thing work? How do we run Javascript in the background? The general steps:

1. Create a worker script – `worker.js`
2. In the "main page", we create a worker object and point to the worker script – `var work = new Worker("worker.js");`
3. Send data from the "main page" to the worker.
4. The worker does processing in the background, returns the results back to the "main page" when it is done.
5. The "main page" receives the results and proceeds with whatever is required next.

Yes, this is something like an AJAX call to a Javascript. An example is worth a thousand words:

WORKER SCRIPT

CHAPTER-C/1-WORKER.JS

```
// (A) ON RECEIVING DATA FROM "MAIN PAGE"
onmessage = (evt) => {
  // (A1) DO PROCESSING
  console.log("Worker has received data");
  console.log(evt.data);
  var result = +evt.data.a + +evt.data.b;

  // (A2) RESPOND BACK TO "MAIN PAGE"
  postMessage(result);
};

// (B) OPTIONAL - HANDLE ERRORS
onmessageerror = (err) => { console.log(err); };
onerror = (err) => { console.log(err); };
```

A basic worker script is as simple as that.

- `onmessage` – On receiving data from the "main page", we use it as the cue to start (or manage) the processing. In this example, we do the very unproductive thing of adding two numbers. Finally, use `postMessage()` to send the results back to the "main page".
- `onmessageerror` – When the worker fails to parse the message from the "main page". I.E. A corrupted data exchange.
- `onerror` – Handle errors if necessary.

THE "MAIN PAGE"

CHAPTER-C/1-MAIN.HTML

```
// (A) CREATE A NEW WORKER
var work = new Worker("1-worker.js");

// (B) DO THIS ON WORKER RESPONSE
work.onmessage = (evt) => {
  console.log("Worker has responded");
  console.log(evt.data);
};

// (C) OPTIONAL - MANAGE ERRORS
work.onerror = (err) => {
  console.log("Worker script error!");
  console.log(err);
};

// (D) SEND DATA TO WORKER
work.postMessage({a: 99, b: 101});
```

On the "main page" itself, we create a `new Worker()` and send data to it for processing. The worker runs in the background, and returns the result when it is done.

WORKER RESTRICTIONS

1. Workers do not have access to the DOM. Yes, `document` does not exist in the worker. Think of it this way – The worker is simply a script that runs independently.
2. Same origin policy applies. That is, if you are thinking of hosting a worker script on other sites – `new Worker("http://other-`

`site.com/worker.js")` will not work.

3. Of course, we must access the page with `http://` for workers to run properly.

Other than that, workers pretty much work like "any other normal Javascript". We can even perform AJAX Fetch calls in workers. Use that to do meaningful processing in your own projects. For example, a background auto save feature.

SHARED WORKER

To explain what shared workers are, consider this first:

- We create `new Worker("worker.js")` on page A.
- Open a new window, and create `new Worker("worker.js")` on page B.

While it is the same `worker.js`, this will create 2 different worker objects and spawn 2 worker instances. The scope of the worker object is also limited to the page itself – Worker A only exists in page A, Worker B only exists in page B.

As a small sideline, since the worker only exists in a single page, we also commonly call it a "dedicated worker". This is definitely not the smartest and most efficient way to use system resources, so shared workers are later introduced – Where a single worker is shared among many pages, or even among workers themselves (It is possible to create a worker in a worker).

SHARED WORKER SCRIPT

CHAPTER-C/2-SHARED-WORKER.JS

```
// (A) ON "CLIENT CONNECT"
// WHEN A PAGE HAS CREATED THIS AS A SHARED WORKER
onconnect = (evt) => {
  const [port] = evt.ports;

  // (B) ON RECEIVING MESSAGE FROM PAGE
  port.onmessage = (e) => {
    console.log("Worker received " + e.data);
    var result = parseInt(e.data[0]) +
                 parseInt(e.data[1]);
    port.postMessage(result);
  };
};
```

This is pretty much the same as a dedicated worker. The difference here is `onconnect = (evt) => { const [port] = evt.ports; }`

- `onconnect` is fired when a page creates a new shared worker, and "connects" with this script.
- `evt.ports` is an array, and there's some crazy mambo jumbo behind it. But let's just simplify to less the confusion.
  - Just use `port = evt.ports[0]` or `[port] = evt.ports`.
  - `port` is something like a "pipeline" between the page and shared worker.
  - Use `port.onmessage` to listen for messages sent from the page to the shared worker.
  - Use `port.postMessage()` to send data from the worker to the page.

FIRST PAGE

CHAPTER-C/2-PAGE-A.HTML

```
<!-- (A) ADD TWO NUMBERS -->
<form onsubmit="return workDemo()">
  <input type="number" id="numA" required value="99"/>
  <input type="number" id="numB" required value="101"/>
  <input type="submit" value="Go!"/>
</form>

<script>
// (B) CREATE WORKER
var work = new SharedWorker("2-shared-worker.js");

// (C) DO THIS ON WORKER RESPONSE
work.port.onmessage = (evt) => {
  console.log("Worker has responded");
  console.log(evt.data);
};

// (D) OPTIONAL - MANAGE ERRORS
work.onerror = (err) => {
  console.log("Worker script error!");
  console.log(err);
};

// (E) SEND DATA TO WORKER
function workDemo () {
  work.port.postMessage([
    document.getElementById("numA").value,
    document.getElementById("numB").value
  ]);
  return false;
}
</script>
```

This example simply sends 2 numbers to the shared worker to be added together, and it should look very familiar.

- Instead of `new Worker()`, we use `new SharedWorker()` now.
- Same story, use `port.postMessage()` to send data to the worker.
- Use `port.onmessage` to listen for responses from the worker.

Easy? Go ahead and open in the browser, see for yourself how it works.

SECOND PAGE

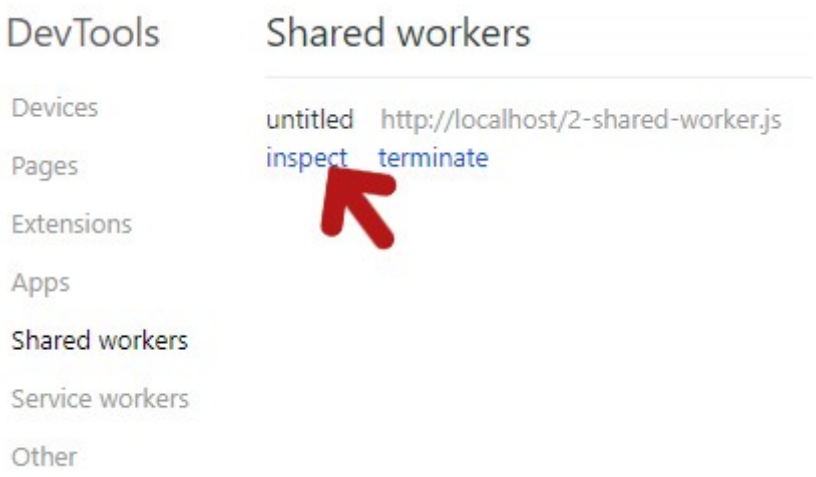
CHAPTER-C/2-PAGE-B.HTML

```
<!-- (A) DOUBLE THE NUMBER -->
<form onsubmit="return workDemo()">
  <input type="number" id="numA" required value="99"/>
  <input type="submit" value="Go!"/>
</form>

<script>
// (B) CREATE WORKER
var work = new SharedWorker("2-shared-worker.js");

// (C) DO THIS ON WORKER RESPONSE
work.port.onmessage = (evt) => {
  console.log("Worker has responded");
  console.log(evt.data);
};

// (D) OPTIONAL - MANAGE ERRORS
work.onerror = (err) => {
  console.log("Worker script error!");
  console.log(err);
```

```
};

// (E) SEND DATA TO WORKER
function workDemo () {
  work.port.postMessage([
    document.getElementById("numA").value,
    document.getElementById("numA").value
  ]);
  return false;
}
</script>
```

This second page is a variant of the first – It sends a number to the worker and doubles it. Take note, we are creating a `new SharedWorker()` with the same `2-shared-worker.js`. Opening this page in the browser will not spawn a new worker instance, but "share use" the one created on the first page.

INSPECTING SHARED WORKERS



To "prove" that only a single worker instance is shared between both pages, we can check with the developer's console. But the problem is, shared workers will not show up in the console of the page itself. I.E.

Shared workers have a global scope. To debug and see the list of shared workers, we have to open up the "hidden" debug page in browsers:

- **Chromium-based Browsers** – Open `chrome://inspect/ OR edge://inspect OR opera://inspect/`, select "shared workers" in the menu.
- **Firefox** – Open the URL `about:debugging`, "This Firefox", "Shared Workers".
- **Safari** – Shared worker? What's that? Not supported at the time of writing. (Told you Safari is Grade B)

SHARED WORKER RESTRICTIONS

A shared worker is still a worker.

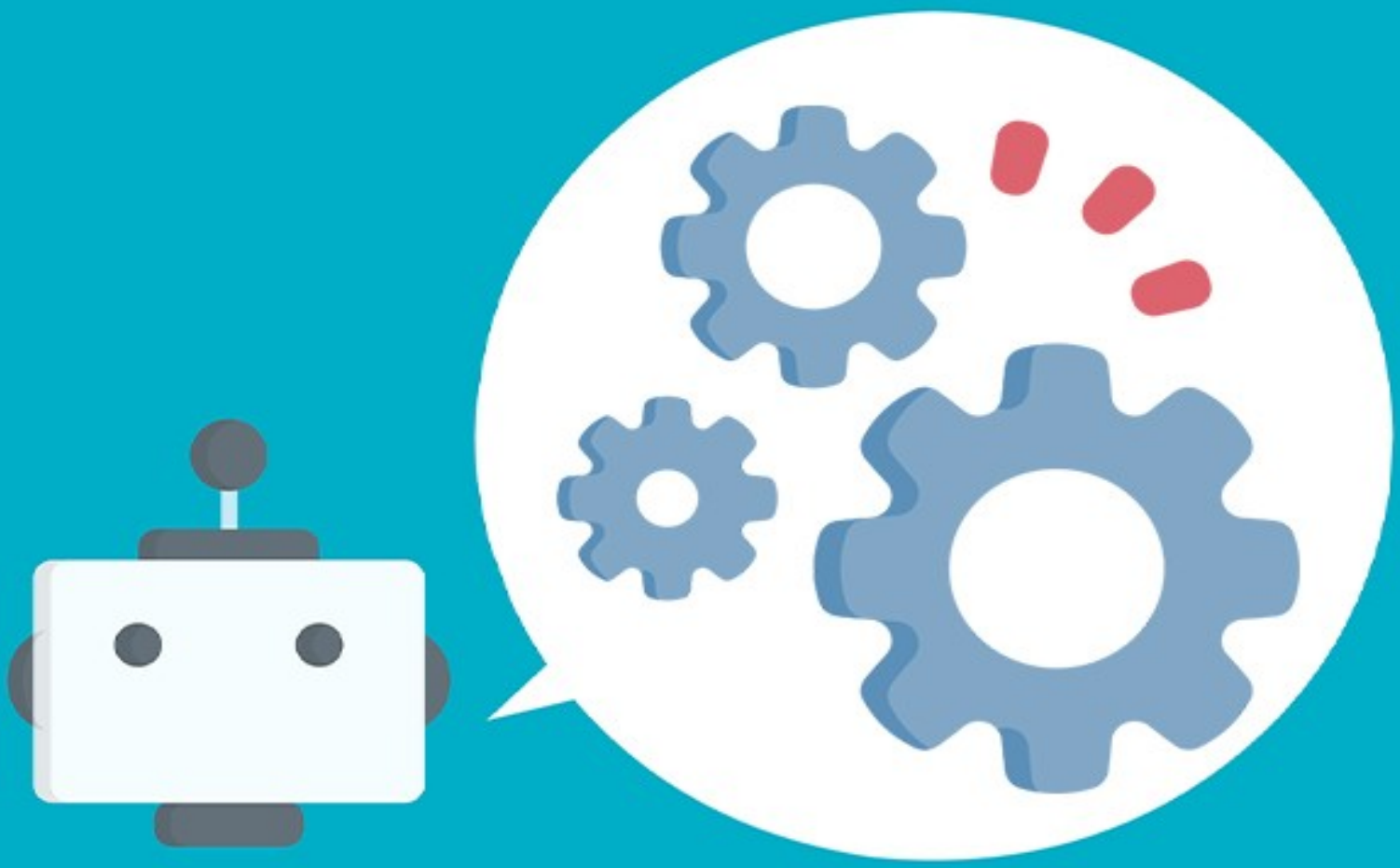
- Same origin policy applies. A shared worker can be used by many pages, but all of them must be from the same domain or IP address.
- Must use `http://` to access the page.
- Of course, no access to the DOM, being an independent script.

LINKS & REFERENCES

- Threading – Wikipedia
- Web Worker – MDN
- Shared Worker – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Web Worker
- Shared Web Workers



- CHAPTER D -

SERVICE WORKERS

SERVICE WORKER

Next, we have yet another kind of worker, the service worker. But this is totally different, as it can run even when the user is offline. Yes, you read that right. Service workers can run even after the user closes the page, even when there is no Internet connection.

Service workers are insanely powerful, and there are already a ton of possible uses – Cache site resources, speed up loading, synchronize contents, push data, offline web app, etc...

REGISTERING SERVICE WORKERS

THE MAIN PAGE

CHAPTER-D/1-BASIC-SW.HTML

```
// (A) REGISTER SERVICE WORKER
navigator.serviceWorker.register("1-basic-sw.js")

// (B) REGISTER SERVICE WORKER OK
.then((reg) => {
  if (reg.installing) {
    console.log("Installing service worker");
  } else if (reg.waiting) {
    console.log("Waiting for old instance to end");
  } else if (reg.active) {
    console.log("Service worker active");
  }
})

// (C) ERROR
.catch((err) => { console.log(err); });
```

How do we get started with service workers?

A) First, we register the service worker – `navigator.serviceWorker.register("worker.js")`

B) Then do something on successful registration. Take note that there are 3 possible registration states here:

- `installing` The worker is currently installing, this will only trigger for first time visitors.
- `waiting` Is an interesting one. When we update the worker, the new version will wait for the old one to gracefully end before taking over. This is the waiting state, to make sure that only one instance of the worker is running.
- `active` The worker ready and active.

C) Something went wrong with the service worker registration, handle the error.

SERVICE WORKER SCRIPT

CHAPTER-D/1-BASIC-SW.JS

```
// (A) WHEN WORKER IS REGISTERED FOR FIRST TIME
self.addEventListener("install", (evt) => {
  console.log("install", evt);
});

// (B) WORKER IS ACTIVE
self.addEventListener("activate", (evt) => {
  console.log("activate", evt);
});
```

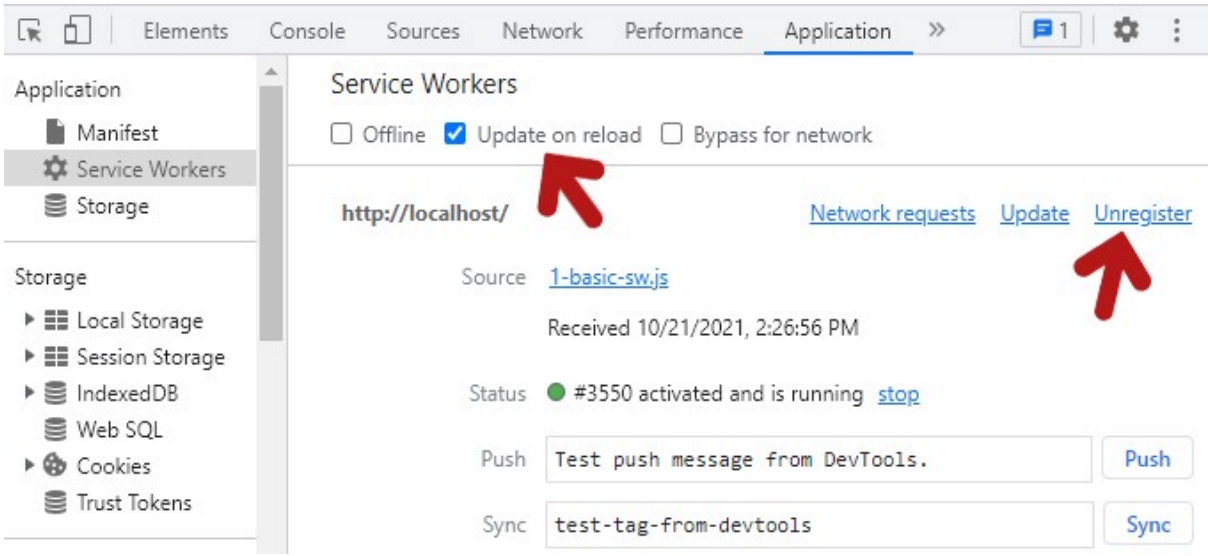
```
// (C) WHENEVER THE WEB PAGE MAKES A REQUEST
self.addEventListener("fetch", (evt) => {
  console.log("fetch", evt);
});

// (D) ON RECEIVING A PUSH REQUEST
self.addEventListener("push", (evt) => {
  console.log("push", evt);
});

// (E) ON RECEIVING A MESSAGE OR DATA
self.addEventListener("message", (evt) => {
  console.log("message", evt);
});

// (F) BACKGROUND SYNC REQUEST
self.addEventListener("sync", (evt) => {
  console.log("sync", evt);
});
```

As you can see, a service worker is very much event driven. Let's start with a quick overview of the common events first:

- `install` When the worker is being registered in the user's browser. A common use in this event is to cache some site scripts and resources, to speed up future loading. Yes, this is a separate persistent "pre-cache", not the "usual perishable" browser cache.
- `activate` Fired after `install`, the worker is ready and listening.
- `fetch` Whenever the page makes a request. A possible use is to "hijack" the request and serve cached versions instead.
- `push` On receiving a push request. To show a push notification, or update certain things.

- `message` Just like the "normal workers", on receiving a message or data.
- `sync` A background sync request. For example, sending chat messages to the server after "recovering" from a bad connection.

INSPECTING & MANAGING SERVICE WORKERS



Now that you have registered your first service worker, there are a few ways to manage and inspect the service worker.

1. Open the developer's console on the page itself – Under the application tab, service workers. A good idea for development is to check "update on reload" here, this will force reload the service worker script every time. You can also send test push messages, unregister service workers here.
2. Open the inspect or debug page.
    - On Chromium-Based browsers – `chrome://inspect OR edge://inspect OR opera://inspect`

- Firefox – `about:debugging`

3. To see the full list of registered service workers in Chromium-based browsers, open `chrome://serviceworker-internals/` OR `opera://serviceworker-internals/` OR `edge://serviceworker-internals/`

P.S. Remember to unregister the example service worker after each run in this book, so they don't clash with each other.

SERVICE WORKER SCOPE

Before we proceed with more examples, here's something very important to touch on – The scope of the service worker.

- By default, the scope of the service worker is set to the folder where it is placed in.
- For example, if the service worker is placed in `products/sw.js`, this service worker will only apply to `products/` and everything below it – `products/cameras`, `products/cameras/nikon/`, `products/lenses/sony/`, etc…
- If the service worker is at the base URL `http://site.com/sw.js`, it will apply to the entire site.

Of course, we can also manually override and specify the scope of the service worker during registration.

CHAPTER-D/2-SCOPE-SW.HTML

```
navigator.serviceWorker.register("1-basic-sw.js", {
  scope: "/"
```

```
  // scope: "/products/"
})
.then((reg) => { console.log(reg.scope); })
.catch((err) => { console.log(err); });
```

SERVICE WORKER RESTRICTIONS

- A valid `https://` is required on live websites to register service workers.
- Service workers cannot be registered in incognito or privacy mode. Firefox users, service workers are not available if the user sets the browser to "never remember history".

SERVICE WORKER CACHING

THE TEST HTML PAGE

CHAPTER-D/3A-CACHE-SW.HTML

```
<script>
navigator.serviceWorker.register
("3-SOMETHING.JS", {scope: "/"})
.then((reg) => { console.log("OK"); })
.catch((err) => { console.log(err); });
</script>

<h1>Hello World!</h1>
<img src="img-science.png"/>
<img src="img-raspberry.png"/>
<img src="nada.png"/>
```

So far so good? Let us now get into "major service worker feature number 1" – Caching. On this test page, we will register the service

worker, and show 3 different images. Will get more into that later.

SAVING FILES INTO THE CACHE

CHAPTER-D/3B-CACHE-NETWORK.JS

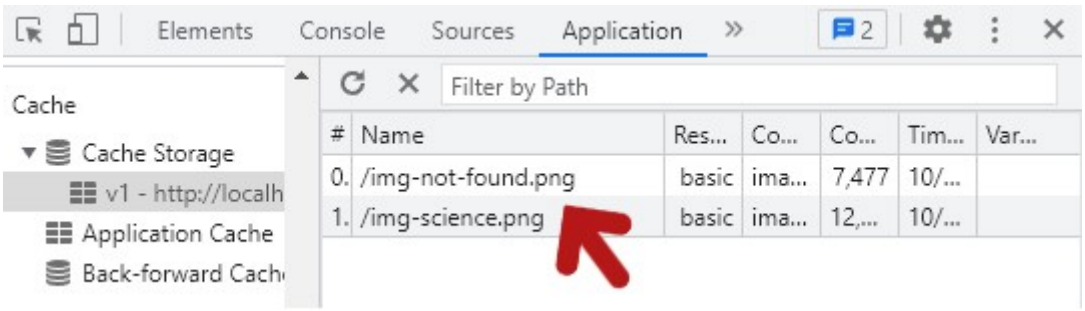
```
// (A) FILES TO CACHE
const cName = "v1",
cList = ["img-science.png", "img-not-found.png"];

// (B) ADD FILES INTO CACHE ON WORKER INSTALL
self.addEventListener("install", (evt) => {
  evt.waitUntil(
    caches.open(cName)
    .then((cache) => { return cache.addAll(cList); })
    .catch((err) => { console.log(err); })
  );
  console.log("Files cached");
});
```

To get started with this "caching thing":

- We usually target the `install` event, save things into the cache on the user's first visit to the website.
- Take note of the use of `evt.waitUntil()` here. This basically tells the browser "don't terminate the worker until this part is fully complete". This will ensure all files are fetched from the server and saved into the local cache.
- We open the cache with `caches.open("v1")`. Take note of "v1", yes, we can create different named caches and even update it later.
- Finally, use `cache.addAll(ARRAY-OF-FILES)` to save a list of

files into the cache.

INSPECTING & MANAGING THE CACHE



Now that you have cached some files, open the developer's console.

- In Chromium-based browsers, you can see and manage the cached files under application, cache.
- In Firefox, it is under storage, cache storage.

Yes, as mentioned earlier, this is a persistent cache storage that is different from the "usual browser cache".

CACHE STRATEGY OPTION 1 – LOAD FROM CACHE ONLY

CHAPTER-D/3B-CACHE-NETWORK.JS

```
// (C) "HIJACK" FETCH REQUESTS - CACHE STRATEGIES
self.addEventListener("fetch", (evt) => {
  // (C1) SERVE FROM CACHE ONLY
  evt.respondWith(caches.match(evt.request));
}
```

The next part of the service worker deals with how the files are being served. We simply "hijack" all the fetch requests and "override" them – `self.addEventListener("fetch", …)`.

In this first option, we will only serve files from the cache itself. Not recommended unless you have a very robust cache, or want to create something like a "version" thing – Do your own manual checks, prompt the user to "update" when there is a new version.

CACHE STRATEGY OPTION 2 – LOAD FROM NETWORK ONLY

CHAPTER-D/3B-CACHE-NETWORK.JS

```
// (C) "HIJACK" FETCH REQUESTS - CACHE STRATEGIES
self.addEventListener("fetch", (evt) => {
  // (C2) SERVE FROM NETWORK ONLY
  // evt.respondWith(fetch(evt.request));
}
```

This one does not make any sense. Cache the files and don't use it!?

CACHE STRATEGY OPTION 3 – NETWORK LOAD FIRST, THEN FALLBACK TO CACHE

CHAPTER-D/3B-CACHE-NETWORK.JS

```
// (C) "HIJACK" FETCH REQUESTS - CACHE STRATEGIES
self.addEventListener("fetch", (evt) => {
  // (C3) NETWORK FIRST, FALLBACK TO CACHE
  evt.respondWith(
    fetch(evt.request)
    .catch(() => { return caches.match(evt.request); })
  );
}
```

Recommended option, try to load from the network first, then fallback to use the cache if the server cannot be reached.

CACHE STRATEGY OPTION 4 – CACHE LOAD FIRST, THEN FALLBACK TO NETWORK

CHAPTER-D/3B-CACHE-NETWORK.JS

```
// (C) "HIJACK" FETCH REQUESTS - CACHE STRATEGIES
self.addEventListener("fetch", (evt) => {
  // (C4) LOAD FROM CACHE, FALLBACK TO NETWORK
  evt.respondWith(
    caches.match(evt.request)
    .then((res) => {
      if (res) { console.log("Serve cached"); }
      return res || fetch(evt.request);
    })
  );
});
```

Another recommended option. Try to load from the cache first, then load from the network if the file is not in the cache.

STORAGE CACHE TEST

The example script above is set to the above "cache first, network fallback". Go ahead – Load/reload this page and see the cache in action.

This will serve the cached copy of `img-science.png`, but since `img-raspberry.png` is not cached, it will load from the server instead.

ADDING MORE FILES INTO CACHE

CHAPTER-D/3C-CACHE-NETWORK-MORE.JS

```
// (C) "HIJACK" FETCH REQUESTS
self.addEventListener("fetch", (evt) => {
  evt.respondWith(
    caches.match(evt.request).then((res) => {
      // (B1) FILE FOUND IN CACHE - SERVE CACHED COPY
      if (res !== undefined) { return res; }

      // (B2) ELSE PROCEED WITH NETWORK LOAD
      else {
        return fetch(evt.request).then((res) => {
          // REQUESTED FILE TYPE
          let type = res.url.split(/[#?]/)[0].split('.')
                      .pop().trim().toLowerCase(),
              imgs = ["png","jpg","jpeg","webp","gif"];

          // CACHE IMAGES ONLY
          // RES CAN ONLY BE USED ONCE, THUS THE CLONE()
          if (res.status==200 && imgs.includes(type)) {
            let clone = res.clone();
            caches.open("v1").then((cache) => {
              cache.put(evt.request, clone);
            });
          }

          // NOT FOUND - SERVE STANDARD "NOT FOUND IMAGE"
          if (res.status==404 && imgs.includes(type)) {
            return caches.match("/img-not-found.png");
          }

          // RETURN WHATEVER SERVER RETURNS
          return res;
        });
      }
```

```
    })
  );
});
```

This is a "slightly improved version" of the previous example, with 2 additions:

- As you can see, it is possible to add more files into the cache. Simply open the cache `caches.open("v1")`, and push the file into the cache `cache.put(evt.request, clone)`.
- If an image file does not exist in the cache, and cannot be loaded from the server – We can serve a generic "not found" image.

Feel free to tweak this example for your future projects – We can also use the cache to store HTML, CSS, Javascript, and even JSON data to speed up loading.

But take note that this approach may not always be the best – Once cached, it will skip loading from the server, potentially missing out updated files and resources. Some form of meticulous file versioning or "expiry" need to be in place if you wish to cache things extensively.

DELETING & UPDATING THE CACHE

CHAPTER-D/3D-UPDATE-CACHE.JS

```
// (A) NEW V2 CACHE
const cName = "v2",
cList = ["img-science.png", "img-raspberry.png", "img-not-found.png"];

// (B) ADD FILES INTO CACHE ON WORKER INSTALL
self.addEventListener("install", (evt) => {
```

```
  evt.waitUntil(
    caches.open(cName)
    .then((cache) => { return cache.addAll(cList); })
    .catch((err) => { console.log(err); })
  );
  console.log("Files cached");
});

// (C) REMOVE ALL OLD CACHES
self.addEventListener("activate", (evt) => {
  var keep = ["v2"];
  evt.waitUntil(
    caches.keys().then((allCache) => {
      return Promise.all(allCache.map((key) => {
        if (keep.indexOf(key) === -1)
        { return caches.delete(key); }
      }));
    })
  );
});
```

If you need to overhaul the entire cache – Simply create a new cache on `install`, and delete all the old ones on `activate`.

SERVICE WORKER MESSAGE

SIMPLE RELAY – THE JAVASCRIPT

CHAPTER-D/4A-MESSAGE.HTML

```
<!-- (A) SERVICE WORKER & DEMO SCRIPT -->
<script>
// (A1) REGISTER SERVICE WORKER
navigator.serviceWorker.register("4b-message.js")
.then((reg) => { console.log("Ready"); });
```

```
// (A2) SEND MESSAGE TO SERVICE WORKER
function worksend () {
  var msg = document.getElementById("demoMsg").value;
  navigator.serviceWorker.controller.postMessage(msg)
  return false;
}

// (A3) ON RECEIVING MESSAGE FROM WORKER
navigator.serviceWorker.addEventListener
("message", (evt) => {
  console.log("Page received message", evt);
  document.getElementById("demoRecv")
  .innerHTML = evt.data;
});
</script>
```

Remember that we can send messages to "normal workers" and shared workers from earlier?

- We can also send messages to service workers with `navigator.serviceWorker.controller.postMessage()`.
- Listen to messages received from service workers – `navigator.serviceWorker.addEventListener("message", (evt) => { evt.data })`.

SIMPLE RELAY – THE HTML

CHAPTER-D/4A-MESSAGE.HTML

```
<!-- (B) SEND MESSAGE -->
<form id="demoSend" onsubmit="return worksend()">
  <input type="text" id="demoMsg" value="Foo Bar"
         required/>
  <input type="submit" value="Go!"/>
```

```
</form>

<!-- (C) RECEIVE MESSAGE -->
<div id="demoRecv"></div>
```

In this example, we simply send a text message to the service worker, and it will relay that message to all registered clients. Open this page in two different windows (or browsers) and test this for yourself – The message should forward itself to both windows.

SIMPLE RELAY – SERVICE WORKER

CHAPTER-D/4B-MESSAGE.JS

```
// (B) HANDLE MESSAGES
self.addEventListener("message", (evt) => {
  // (B1) MESSAGE RECEIVED
  console.log("Worker received message", evt);

  // (B2) WE CAN ALSO DETERMINE WHO IS THE SENDER
  var sender = evt.source.id;

  // (B3) FORWARD MESSAGE TO ALL CLIENTS
  self.clients.matchAll().then((all) => {
    all.forEach((client) => {
      client.postMessage(evt.data);
      // IF YOU WANT TO EXCLUDE THE SENDER
      // if (client != sender) { ... }
    });
  });
});
```

Yes, it's that easy. Just get the whole list of clients and forward the message to everyone.

SMARTER USE OF MESSAGE

A smarter way to use the message event is to probably turn it into a "custom background service API".

```
self.addEventListener("message", (evt) => {
  switch (evt.data.req) {
    case "autosave":
      // SAVE DATA INTO CACHE OR INDEXED DB OR SERVER
      break;

    case "SERVICE":
      // MORE CUSTOM BACKGROUND PROCESSING SERVICES
      break;
  }
});
```

Then on the main page – `postMessage({ req:"autosave", data:"SOMETHING" })`. But this is only one of the possibilities, feel free to do whatever creative stuff with it.

SERVICE WORKER BACKGROUND SYNC

WHAT IS BACKGROUND SYNC?

To fully understand and appreciate background sync, consider this – You are inside an elevator with no network reception. Tried to submit a form, only to get a "no Internet connection" error. That is a real bummer, and you have to go back to refill the entire form again.

But with background sync – It can hold off the network request, and automatically resend when the user has a better connection. Sounds cool? But background sync can be pretty hard to understand, so follow along closely.

STEP 1) DUMMY SERVER SCRIPT

CHAPTER-D/5A-DUMMY.PHP

```
<?php
file_put_contents("POST.txt", date("Y-m-d H:i:s"));
echo "OK";
```

First, create a simple dummy server script to handle "form submissions". Here is one in PHP, and all it does is create a `POST.txt` with the current timestamp. Feel free to create your own, using whatever language you are familiar with.

STEP 2) REGISTER SYNC REQUEST

CHAPTER-D/5B-SYNC.HTML

```
<!-- (A) TEST BUTTON -->
<input type="button" value="Go!" id="demoGo" disabled/>

<!-- (B) SERVICE WORKER -->
<script>
navigator.serviceWorker.register("5c-sync.js")
.then((reg) => { return navigator.serviceWorker.ready; })
.then((reg) => {
  var btn = document.getElementById("demoGo");
  btn.addEventListener("click", () => {
    reg.sync.register("demosend").then(() => {
      console.log("Sync registered");
    });
  });
  btn.disabled = false;
})
.catch((err) => { console.log(err); });
</script>
```

Next, open this HTML page in your browser. Take extra note, clicking on the button will not fire a fetch request to the dummy server-side script immediately, but register a sync request with the service worker.

P.S. Don't click on this button yet.

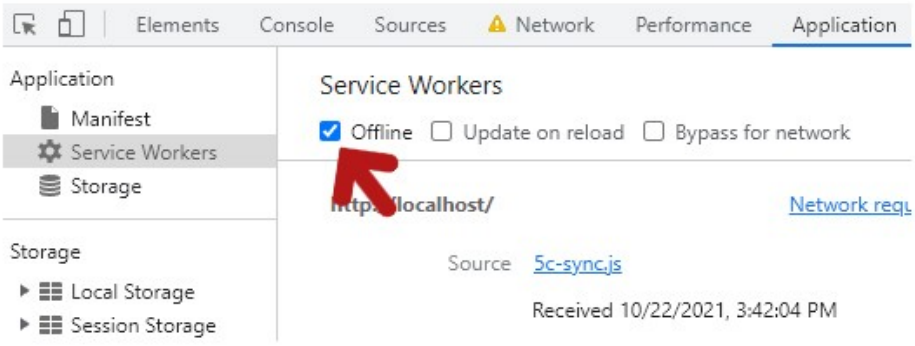
STEP 3) SERVER WORKER SYNC HANDLER

CHAPTER-D/5C-SYNC.HTML

```
self.addEventListener("sync", (evt) => {
  if (evt.tag == "demosend") {
    evt.waitUntil(
      fetch("5a-dummy.php")
      .then((res) => { return res; })
      .then((text) => { console.log("Fetch OK", text); })
      .catch((err) => { console.log(err); })
    );
  }
});
```

The fetch request to the server-side script is done in the service worker instead – This `sync` event will trigger when there is a good connection.

STEP 4) GO OFFLINE



Next, go offline under the developer's console.

STEP 5) COMPLETE TEST

Click on the button and watch the sync request register – The fetch request will not send out because you are currently offline.

Lastly, uncheck "offline" and watch the background sync fire the fetch request immediately. The server-side script should also generate the dummy `POST.txt`.

HOW ABOUT THE FORM DATA!?

Some of you sharp code ninjas should have already noticed – A fetch call is made, but no form data is sent to the server. Yes, this is the sticky situation.

- Remember that workers do not have access to the DOM? That is, we cannot directly do a `document.getElement(FORM-FIELD).value` in the worker itself.
- At the time of writing, we cannot pass data into `sync.register()` either.

So in order to feed the form data:

- On the page itself, we have to first create an indexed database.
- Save the data into the indexed database on form submit.

- Register the background sync.
- Then in the worker sync process, retrieve the data from the indexed database.
- Append the data into the fetch request.
- After a successful fetch call, clean up and remove the data in the indexed database.

Yep, this is a major pain, and too much to walk through right now. So we will cover both "how to create an indexed database" and "do a POST fetch call" in later chapters instead.

MORE SERVICE WORKER USES

Congratulations, you have covered most of the basics of service workers. But there's a lot more, really.

- We will hold off push notifications to a later chapter.
- Service workers can be used to push, pull, and synchronize data.
- Cache can also be used to deploy an offline version, you can use it to build a "save this to read later" feature.
- If you like, you can even build an entire offline web app with it.
- There is also a "periodic sync", but it is not really well documented and supported at the time of writing.

The list can go on, but I shall sign off this chapter for now... Or it will never end. If you are interested for more, follow up with the links below, check out the cookbook especially.

LINKS & REFERENCES

- Service Worker – MDN
- Service Worker Cookbook (A good collection of many service worker examples)

COMPATIBILITY CHECKS (WITH CANIUSE)

- Service Worker

- CHAPTER E -

CACHE STORAGE

PERSISTENT CACHE

If you have not skipped the previous chapter, this one should be very familiar – Yes, it's the same cache storage that we used in service workers. Captain Obvious to the rescue, they are not just a "service worker only" feature, we can also access it in "normal web pages" too. Here is a quick chapter to "officially document" the functions, and some creative uses.

CREATING CACHES

CHAPTER-E/1-CREATE.HTML

```
// (A) CHECK IF "V1" CACHE ALREADY CREATED
caches.has("v1").then((exist) => {
  // (B) "V1" ALREADY CREATED
  if (exist) { console.log("Has V1 cache"); }

  // (C) CREATE "V1" CACHE
  else {
    caches.open("v1").then((cache) => {
      cache.addAll(["img-raspberry.png"])
      .then(() => { console.log("V1 created"); });
    });
  }
});
```

- `caches.has("NAME")` – Check if the specified cache exists.
- `caches.open("NAME")` – Opens the specified cache.
- `cache.addAll(ARRAY)` – Adds the list of files into the cache.

ADDING FILES

CHAPTER-E/2-PUT.HTML

```
// (A) CREATE TEXT BLOB
var txtBlob = new Blob(
  ["Hello World!"], {type: "text/plain"}
);

// (B) BLOB OBJECT URL
var urlBlob = URL.createObjectURL(txtBlob);

// (C) FETCH BLOB & STORE
fetch(urlBlob)
.then((res) => {
  caches.open("v1").then((cache) => {
    cache.put("demo.txt", res);
    URL.revokeObjectURL(urlBlob);
  });
});
```

Right, you already know how to use `cache.put()` to add a file into the cache. But how about… We twist it to create and save our custom files? This is kind of "hackish", but it works. The cache storage is persistent too, so this file is not going to disappear.

RETRIEVE TEXT FILES

CHAPTER-E/3A-RETRIEVE-TXT.HTML

```
<!-- (A) EMPTY DIV -->
<div id="demo"></div>

<script>
// (B) MATCH FILE IN ALL CACHE STORAGE
caches.match("demo.txt")
```

```
// (C) RETURN AS TEXT
.then((res) => { return res.text(); })
.then((txt) => {
  document.getElementById("demo").innerHTML = txt;
});
</script>
```

This should be straightforward – We can "fetch" the previous text file from the cache.

RETRIEVE IMAGE FILES

CHAPTER-E/3B-RETRIEVE-IMG.HTML

```
<!-- (A) EMPTY IMAGE TAG -->
<img id="demo"/>

<script>
// (B) MATCH FILE IN ALL CACHE STORAGE
caches.match("img-raspberry.png")

// (C) RETURN AS BLOB
.then((res) => { return res.blob(); })

// (D) BLOB > BASE64 > INTO IMG TAG
.then((imgBlob) => {
  let reader = new window.FileReader();
  reader.addEventListener("load", () => {
    document.getElementById("demo").src = reader.result;
  });
  reader.readAsDataURL(imgBlob);
});
</script>
```

Image files? No problem.

DELETING CACHES

CHAPTER-E/4-DELETE.HTML

```
// (A) GET ALL CACHE NAMES
caches.keys().then((all) => {
  // (B) LOOP & DELETE
  for (let cname of all) {
    caches.delete(cname).then(() => {
      console.log(`${cname} deleted`);
    });
  }
});
```

- `caches.keys()` – Returns the whole list of cache names.
- `caches.delete(NAME)` – Delete the specified cache.

LINKS & REFERENCES

- Cache Storage – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Cache Storage

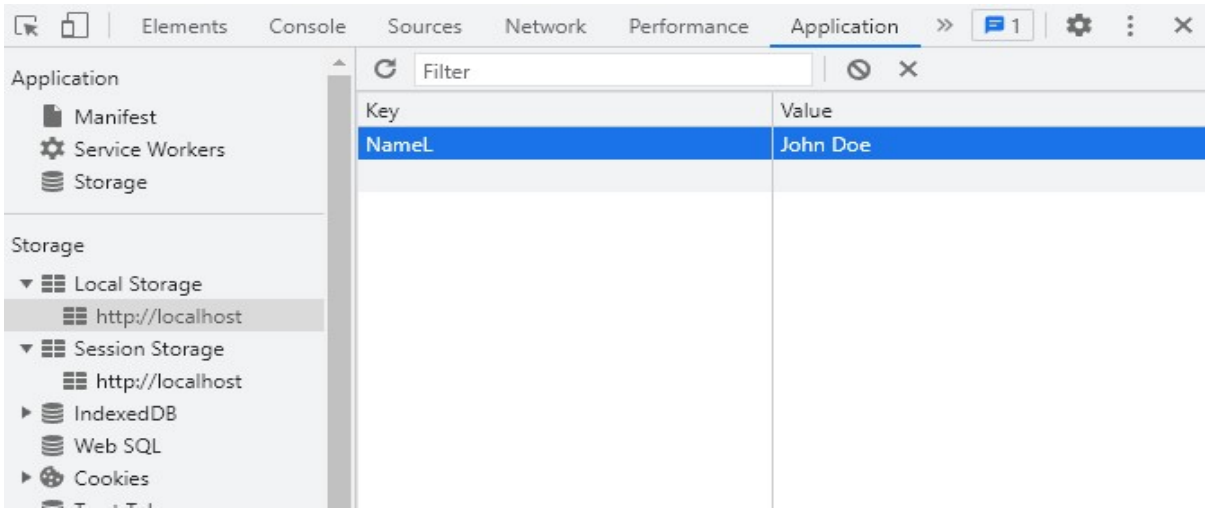
- CHAPTER F -

LOCAL STORAGE & SESSION STORAGE

LOCAL STORAGE & SESSION STORAGE

Yes, some of you guys may have already heard of it – It is possible to store and retrieve data in Javascript, the local storage/session storage is one of the most common options.

WHAT'S IN MY LOCAL & SESSION STORAGE?



If you want to check the contents of local or session storage, simply open the developer's console.

- In Chromium-Based browsers (Chrome, Opera, Edge) - Check under Application > Storage > Local Storage.
- In Firefox, it is under Storage > Local Storage.

LOCAL STORAGE BASIC USAGE

CHAPTER-F/1-LOCAL-STORAGE.HTML

```
// (A) SET ITEM INTO LOCAL STORAGE
localStorage.setItem("Name", "John Doe");

// (B) GET ITEM FROM LOCAL STORAGE
var name = localStorage.getItem("Name");
console.log(name); // John Doe

// (C) REMOVE ITEM FROM LOCAL STORAGE
localStorage.removeItem("Name");

// (D) TO CLEAR EVERYTHING AT ONCE
localStorage.clear();
```

That is as straightforward as it gets, there are only 4 functions to handle local storage like a pro:

- Just like defining `var NAME= VALUE`, we use `localStorage.setItem("NAME", "VALUE")` to save a value into the local storage.
- To retrieve a value from the local storage, we use `localStorage.getItem("NAME")`.
- To delete a stored value, we use `localStorage.removeItem("NAME")`.
- Lastly, to clear out everything in the local storage, we use `localStorage.clear()`.

SESSION STORAGE BASIC USAGE

CHAPTER-F/2-SESSION-STORAGE.HTML

```
// (A) SET ITEM INTO SESSION STORAGE
sessionStorage.setItem("Name", "Jane Doe");

// (B) GET ITEM FROM SESSION STORAGE
var name = sessionStorage.getItem("Name");
console.log(name); // Jane Doe

// (C) REMOVE ITEM FROM SESSION STORAGE
sessionStorage.removeItem("Name");

// (D) TO CLEAR EVERYTHING AT ONCE
sessionStorage.clear();
```

Look no further, session storage has the exact same 4 functions as local storage.

LOCAL STORAGE VS SESSION STORAGE

So what is the difference between these two?

- Local storage is persistent. The user can close the entire browser, and the local storage will still be there on the next visit/session.
- Session storage is temporary. The data is wiped out once the user closes the browser or ends the session.

Go ahead and verify this yourself. Access this page in the browser first:

CHAPTER-F/3A-SET-DATA.HTML

```
// (A) SET ITEM INTO LOCAL STORAGE
localStorage.setItem("NameL", "John Doe");
```

```
// (B) SET ITEM INTO LOCAL STORAGE
sessionStorage.setItem("NameS", "Jane Doe");

console.log("ALL SET! CLOSE THE BROWSER.");
```

Then, close the browser and open this page:

CHAPTER-F/3B-GET-DATA.HTML

```
// (A) GET ITEM FROM LOCAL STORAGE
var nameL = localStorage.getItem("NameL");
console.log(nameL); // John Doe

// (B) GET ITEM FROM SESSION STORAGE
var nameS = sessionStorage.getItem("NameS");
console.log(nameS); // Null
```

BUT take extra note – Once the user do a "clear browser cache", everything will still be lost regardless.

THE BITS & PIECES

CHAPTER-F/4-MORE.HTML

```
// (A) DATA ARRAY
var data = ["John", "Jane"];

// (B) LOCAL/SESSION STORAGE ONLY ACCEPTS FLAT STRINGS &
// NUMBERS. JSON ENCODE ARRAYS & OBJECTS BEFORE STORING
localStorage.setItem("Names", JSON.stringify(data));

// (C) RETRIEVE ARRAY FROM LOCAL/SESSION STORAGE
// SIMPLE JSON DECODE TO GET IT BACK
var retrieved = localStorage.getItem("Names");
console.log(retrieved); // STRING
retrieved = JSON.parse(retrieved);
console.log(retrieved); // ARRAY
```

```
// (D) NAMES ARE CASE SENSITIVE
var retrieved = localStorage.getItem("names");
console.log(retrieved); // NULL
```

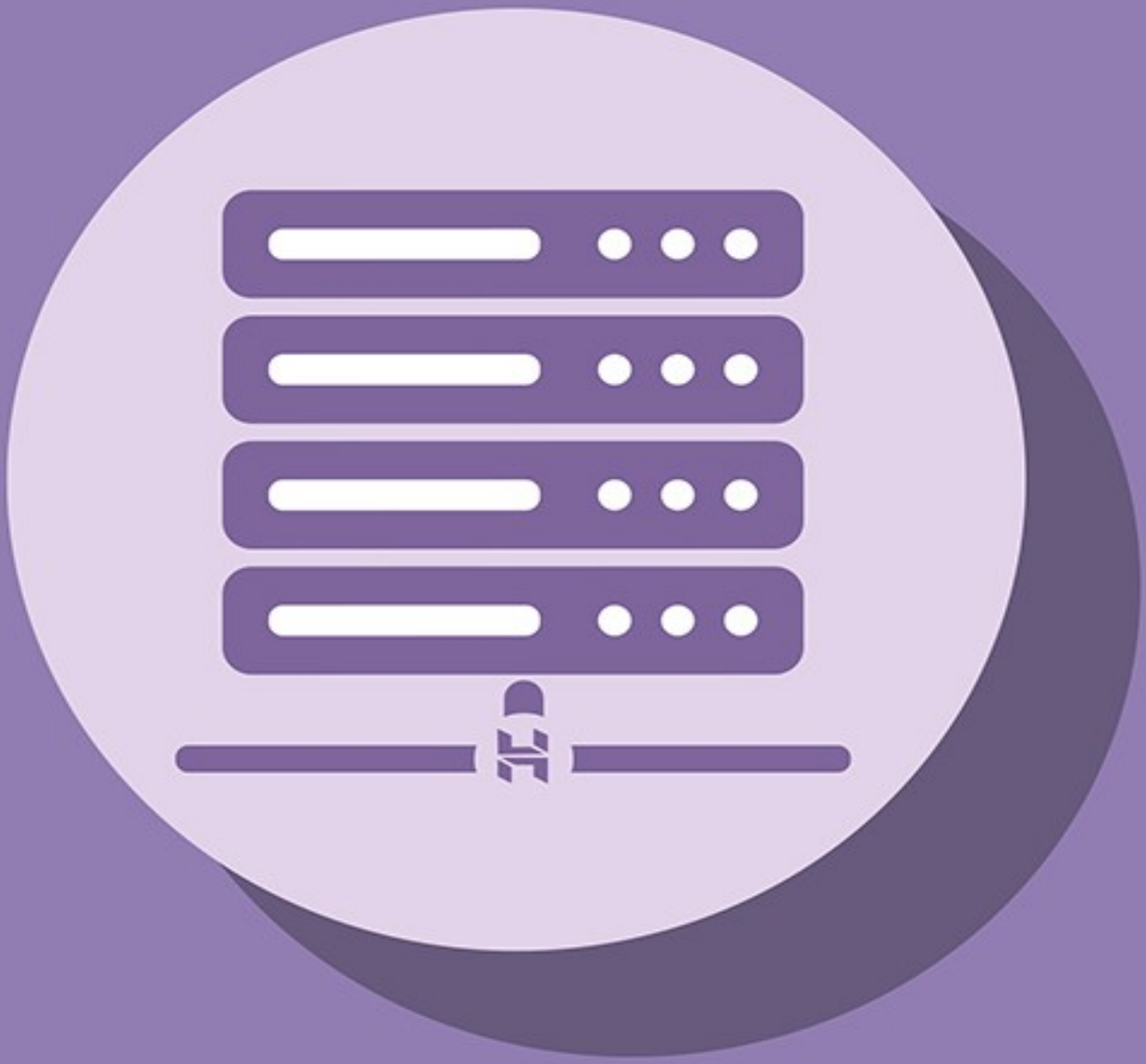
- Local/session storage cannot store arrays and objects. JSON encode/decode if you want to store them.
- Just like "normal variables", the names are also case-sensitive. I.E. "name" is different from "Name".
- Both local storage and session storage have very limited storage space… Don't even bother to store anything large with it.

LINKS & REFERENCES

- Local Storage – MDN
- Session Storage – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Local Storage
- Session Storage



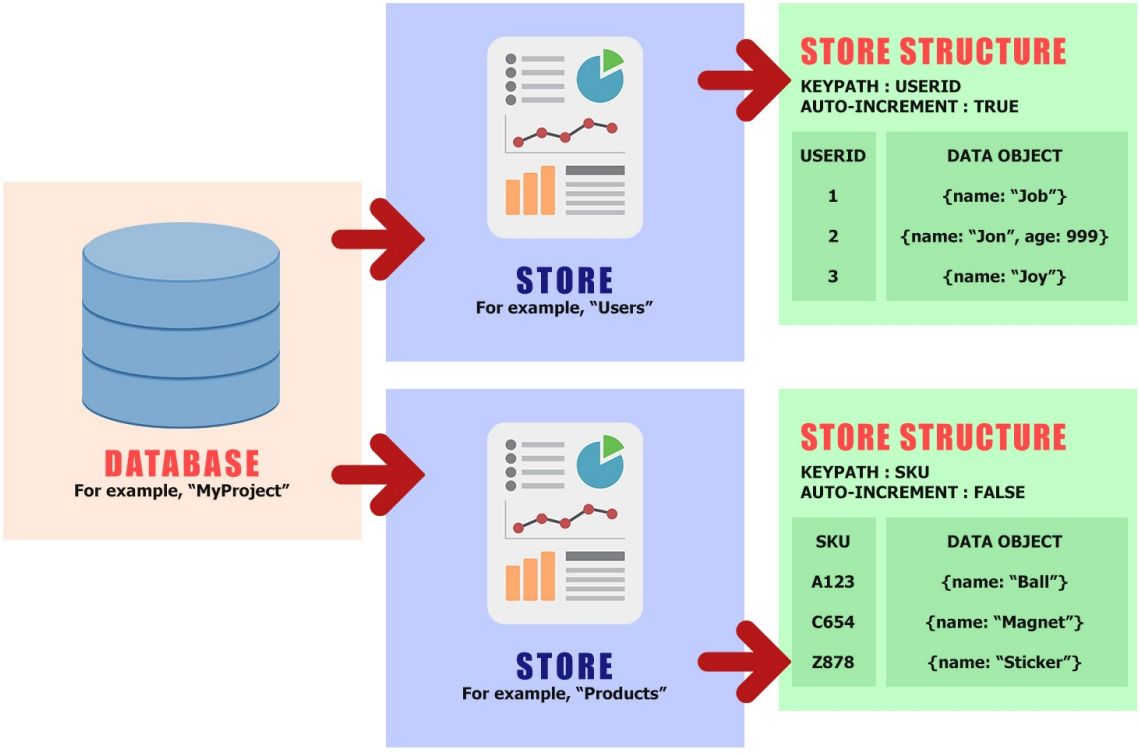
- CHAPTER G -

INDEXED DATABASE

JAVASCRIPT DATABASE

You read that right. We can create a database in client-side Javascript, save a large amount of data in the user's browser for the long term. Indexed databases have actually been around for some time, but it was never too popular due to its "unstable changing procedures"... Still, things seem to be slowly finalizing, and this is a good alternative if you need to store data on a substantial level.

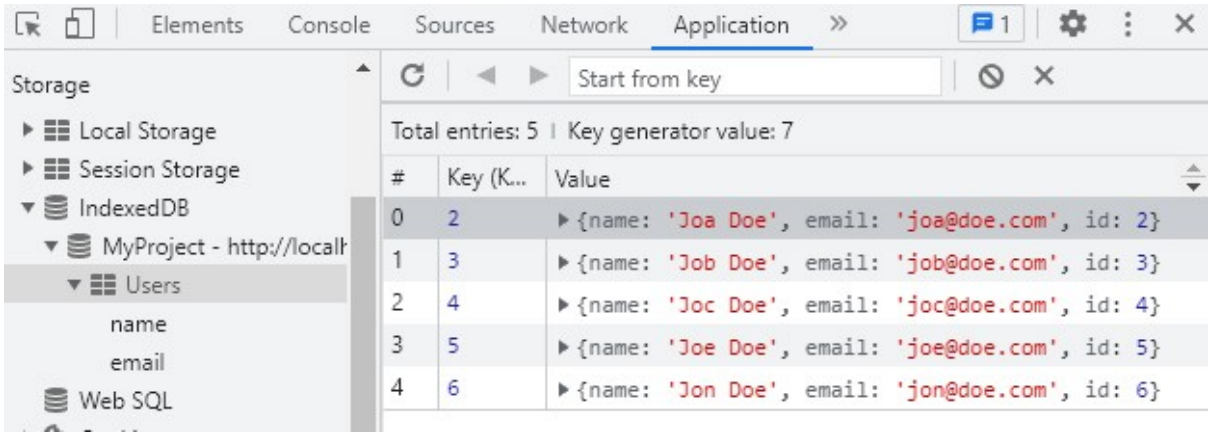
INDEXED DB STRUCTURE IN A NUTSHELL



Before we go into the actual code, let us start with the structure of an indexed database (IDB). Some of you who have used relational databases (RDB) before are probably going to brain freeze for a while – IDB shares a few similarities with RDB, but they are totally different.

- At the "top level", we have the `database`. For example, we create a test database called "MyProject".
- A database can contain a number of `object stores`. For example, `Users` and `Products`. You guys who have used a RDB before, these are the so-called "tables".
- How an object store work is very simple. We only need to define a `KEY` for the object store, and save `OBJECTS` into the store.
- Easy example (see above illustration):
  - We create a `Users` store. Define `USERID` as the `KEY`, set it as an auto-increment running number.
  - When we save an object into `Users`, a `USERID` will automatically be assigned to the newly added object.
  - To retrieve an object, we simply have to refer it by the `KEY`, the `USERID`. For example, `USERID 3` will refer to `Joy`.
  - RDB users – An object store simply keeps objects and assigns a `KEY` to it. Period. There are no restrictions on what properties the object must have, there is no such thing as "table structure".
- Next example, we have a `Products` store.
  - This time, the `KEY` is the item `SKU` (stock-keeping unit). Not a running number.
  - Very simply, we have to supply the `SKU` when adding objects to the Products store. Retrieve products by referencing the `SKU`.

WHAT'S IN MY INDEXED DB?



To check the contents of your indexed DB, open the developer's console:

- In Chromium-Based browsers (Chrome, Opera, Edge) - Check under Application > Storage > IndexedDB.
- In Firefox, it is under Storage > IndexedDB.

Don't forget to hit the "refresh" button, the developer's console does not update the contents on-the-fly.

INITIALIZING & CREATING A DATABASE

CHAPTER-G/1-OPEN-INIT.JS

```
// (A) OPEN "MYPROJECT" DATABASE
var idb = window.indexedDB,
    req = idb.open("MyProject");

// (B) ON DATABASE ERROR
req.onerror = (evt) => { console.log(evt); };

// (C) ON UPGRADE NEEDED (CREATE OR UPDATE DATABASE)
req.onupgradeneeded = (evt) => {
```

```
  // (C1) GET DATABASE
  idb = evt.target.result;
  idb.onerror = (evt) => { console.log(evt); };

  // (C2) CREATE STORE
  var store = idb.createObjectStore("Users", {
    keyPath: "id",
    autoIncrement: true
  });
  console.log("Database Created");
};

// (D) OPEN DATABASE OK
req.onsuccess = (evt) => {
  idb = evt.target.result;
  console.log("READY!");
};
```

A) Captain Obvious, we open a database using `window.indexedDB.open("DATABASE")`.

B) Handle any database errors. This is optional.

C) Create the object stores and structure in the user's browser.

D) What to do on a successful database connection. Here, we will just reference `var idb` back to the `MyProject` database.

BASIC TRANSACTIONS

BASIC TRANSACTION (HELPER FUNCTION)

CHAPTER-G/2-TRANSACTIONS.JS

```
// (A) TRANSACTION HELPER FUNCTION
function tx () {
```

```
  return idb
    .transaction("Users", "readwrite")
    .objectStore("Users");
}
```

To process a database transaction – We need to start with the access permission `transaction("STORE", "PERMISSION")`, then select the store `objectStore("STORE")`. Annoying, but that is how it works.

ADD ENTRY (SAVE)

CHAPTER-G/2-TRANSACTIONS.JS

```
// (B) ADD NEW ENTRY
function add () {
  tx().add({
    name: "John Doe",
    email: "john@doe.com"
  });
  console.log("Added!");
}
```

Self-explanatory. Use `add(OBJECT)` to save an object into the object store.

UPDATE ENTRY (PUT)

CHAPTER-G/2-TRANSACTIONS.JS

```
// (C) PUT (UPDATE) AN ENTRY
function put () {
  tx().put({
    id: 1,
    name: "John Doezzz",
```

```
    email: "john@doe.com",
    gender : "male"
  });
  console.log("Updated");
}
```

Use `put(OBJECT)` and define the `KEY` to update an object in the store.

Quick reminder –

- The `KEY` for the `Users` store is `id`.
- If the provided `id` already exists, the entry will be replaced. If not, a new entry will be created.
- Special service for RDB users, take note of the extra `gender`. An object store does not have "fixed columns", it does not care what properties the object has; It simply saves as long as it is an object.

GET ENTRY

CHAPTER-G/2-TRANSACTIONS.JS

```
// (D) GET (RETRIEVE) ENTRY
function get () {
  var req = tx().get(1);
  req.onsuccess = (evt) => {
    var john = req.result;
    console.log(john);
  };
}
```

Self-explanatory again – Use `get(ID)` to retrieve an entry from the store.

DELETE ENTRY

CHAPTER-G/2-TRANSACTIONS.JS

```
// (E) DELETE ENTRY
function del () {
  var req = tx().delete(1);
  req.onsuccess = (evt) => {
    console.log("Deleted");
  };
}
```

Self-explanatory yet again – Use `delete(ID)` to remove an entry.

UPGRADING A DATABASE

CHAPTER-G/3-UPGRADE.JS

```
// (A) OPEN "MYPROJECT" DATABASE - NOTE : VERSION 2
var idb = window.indexedDB,
    req = idb.open("MyProject", 2);

// (B) ON DATABASE ERROR
req.onerror = (evt) => { console.log(evt); };

// (C) UPGRADE NEEDED (CREATE OR UPDATE DATABASE)
req.onupgradeneeded = (evt) => {
  // (C1) GET DATABASE
  idb = evt.target.result;
  idb.onerror = (evt) => { console.log(evt); };

  // (C2) VERSION 1 - CREATE "USER" STORE
  if (evt.oldVersion < 1) {
    var store = idb.createObjectStore("Users", {
      keyPath: "id",
      autoIncrement: true
```

```
    });
    console.log("Database Created");
  }

  // (C3) VERSION 2 - ADD INDEX "NAME", UNIQUE “EMAIL"
  if (evt.oldVersion < 2) {
    var store = req.transaction.objectStore("Users");
    store.createIndex("name", "name", { unique: false });
    store.createIndex("email", "email", { unique: true });
    // deleteIndex("NAME")
    // deleteObjectStore("NAME")
    console.log("Upgraded To V2");
  }
};

// (D) OPEN DATABASE OK
req.onsuccess = (evt) => {
  idb = evt.target.result;
  console.log("READY!");
};
```

Sometimes, it is just inevitable. We need to add or remove stores, change the indexes. This example is a slightly modified `1-open-init.js` to deal with database upgrades:

- **(A)** The `open()` function actually takes in 2 parameters, the database name and version number. The version number is an integer, please do not enter decimals (such as 1.234), it will be rounded off…
- **(C)** Take note of the changes in this section. We can use `evt.oldVersion` to determine the current database version on the user’s device, and update accordingly. Also, keep a note on how we add an index to the store with `createIndex()`.

INDEXES

HELPER FUNCTION & DUMMY DATA

CHAPTER-G/4-INDEX.JS

```
// (A) TRANSACTION HELPER FUNCTION
function tx () {
  return idb
    .transaction("Users", "readwrite")
    .objectStore("Users");
}

// (B) GENERATE DUMMY DATA
function dummy () {
  // (B1) DUMMY USERS
  var users = [
    {name: "Joa Doe", email: "joa@doe.com"},
    {name: "Job Doe", email: "job@doe.com"},
    {name: "Joc Doe", email: "joc@doe.com"},
    {name: "Joe Doe", email: "joe@doe.com"},
    {name: "Jon Doe", email: "jon@doe.com"}
  ];

  // (B2) ADD TO STORE
  var store = tx();
  for (let u of users) {
    store.add(u);
    console.log(u);
  }
}
```

What are indexes used for? Let us add some dummy users first.

GET BY INDEX

CHAPTER-G/4-INDEX.JS

```
// (C) GET BY NAME
function getName () {
  // (C1) GET "JON DOE"
  var req = tx().index("name").get("Jon Doe");

  // (C2) RESULT
  req.onsuccess = () => {
    console.log(req.result);
  };
}

// (D) GET BY EMAIL
function getEmail () {
  // (D1) GET "JOB@DOE.COM"
  var req = tx().index("email").get("job@doe.com");

  // (D2) RESULT
  req.onsuccess = () => {
    console.log(req.result);
  };
}
```

Now that the users store has some dummy data, indexes should be pretty self-explanatory. In the first version, we can only get users by the `id`. But when we add indexes to the store, it is also possible to get users by their name and email address.

GET ALL & CURSORS

GET ALL ENTRIES

CHAPTER-G/5-ALL-CURSOR.JS

```
// (A) GET ALL
function getAll () {
  // (A1) GET ALL
  var req = idb
    .transaction("Users", "readwrite")
    .objectStore("Users")
    .getAll();

  // (A2) FETCH RESULTS
  req.onsuccess = (evt) => {
    var users = evt.target.result;
    console.log(users);
  };
}
```

So far so good? Let us move into the next stage, getting all entries in one go. Don't think this needs any explanation, just use `getAll()`.

GET CURSOR

CHAPTER-G/5-ALL-CURSOR.JS

```
// (B) GET CURSOR
function getCursor () {
  // (B1) OPEN CURSOR
  var req = idb
    .transaction("Users", "readwrite")
    .objectStore("Users")
    .openCursor();
```

```
  // (B) "CURSOR RUN"
  req.onsuccess = (evt) => {
    let cursor = evt.target.result;
    if (cursor) {
      var user = cursor.value;
      console.log(user);
      cursor.continue();
    } else {
      console.log("DONE!");
    }
  };
}
```

Does this not do the same as `getAll()`? Fetch all users? What's the point of a cursor then? For the uninitiated:

- `getAll()` will get everything in one shot. If the store has thousands of entries, the system will probably run into a performance or "out-of-memory" problem.
- `openCursor()` will get one-by-one instead, this is the safer way to deal with large datasets.

KEY RANGE (LIMIT)

CHAPTER-G/6-RANGE.JS

```
const irange = window.IDBKeyRange;
function getRange () {
  // (A) RANGE LIMIT
  // GET FROM ID 2 TO ID 4
  var limit = irange.bound(2, 4);

  // GET ALL IDS ABOVE 4 (INCLUSIVE OF 4)
  // var limit = irange.lowerBound(4);
```

```
  // GET ALL IDS BELOW 4 (INCLUSIVE OF 4)
  // var limit = irange.upperBound(4);

  // GET ONLY ID 3
  // var limit = irange.only(3);

  // (B) LIMITED GET
  var req = idb
    .transaction("Users", "readonly")
    .objectStore("Users")
    .getAll(limit);

  // (C) RESULTS
  req.onsuccess = (evt) => {
    var users = evt.target.result;
    console.log(users);
  };
}
```

By default, `getAll()` and `openCursor()` will get all entries from the object store. To limit the results, we need to specify a range.

- `bound(X, Y)` will limit the results to `>=X and <=Y`.
- `lowerBound(X)` will limit the results to `>=X`.
- `upperBound(X)` will limit the results to `<=X`.
- `only(X)` is the funky one… It is as good as using `get(ID)`.

SEARCH

CHAPTER-G/7-SEARCH.JS

```
function getSearch () {
  // (A) OPEN CURSOR
```

```
  var req = idb
    .transaction("Users", "readwrite")
    .objectStore("Users")
    .openCursor();

  // (B) SEARCH TERM & RESULTS HOLDER
  var search = "Job", results = [];

  // (C) COLLECT INTO RESULTS IF SEARCH TERM MATCHES
  req.onsuccess = (evt) => {
    let cursor = evt.target.result;
    if (cursor) {
      let user = cursor.value;
      if (user.name.indexOf(search) !== -1) {
        results.push(cursor.value);
      } else if (user.email.indexOf(search) !== -1) {
        results.push(cursor.value);
      }
      cursor.continue();
    } else {
      console.log("DONE!");
      console.log(results);
    }
  };
}
```

Sadly, there is no such thing as a search feature or `SELECT * FROM` `` `TABLE` WHERE XYZ `` in IDB at the time of writing. The only way to do a search is to manually run through all the entries and match your desired search term. By the way, it's a case sensitive search.

PERSISTENT STORAGE

CHAPTER-G/7-SEARCH.JS

```
if (navigator.storage && navigator.storage.persist) {
  navigator.storage.persist()
  .then((persisted) => {
    if (persisted) { console.log("Storage will only be
    cleared when the user manually does it."); }

    else { console.log("Storage can be automatically
    cleared by the browser."); }
  });
}
```

One final bit on the IDB, it works just like the browser cache by default. Whenever there is a storage space crunch, or whenever the browser deems it is appropriate to delete old data – The IDB will be cleared off automatically.

To prevent that from happening, we can ask for persistent storage – `navigator.storage.persist()`. Take note, this requires `HTTPS`.

LINKS & REFERENCES

- Using Indexed DB – MDN
- Working with IndexedDB – Google Developers
- Indexed DB API – MDN
- IDB Key Range – MDN
- Persistent Storage – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Indexed DB
- Persistent Storage

- CHAPTER H -

READ FILES

READING FILES IN NODEJS

CHAPTER-H/1-NODE-READ-FILE.JS

```
// (A) REQUIRE MODULES
const fs = require("fs"), // FILE SYSTEM
      rl = require("readline"), // READ LINE
      https = require("https");

// (B) READ FILE (SYNC)
var data = fs.readFileSync("dummy.txt", "utf8");
console.log(data);

// (C) READ FILE (ASYNC)
fs.readFile("dummy.txt", "utf8", (err, data) => {
  console.log(data);
});

// (D) READ LINE-BY-LINE
const reader = rl.createInterface({
  input: fs.createReadStream("dummy.txt")
});
reader.on("line", (row) => {
  console.log(row);
});

// (E) GET FROM REMOTE SERVER
https.get("https://code-boxx.com", (res) => {
  res.on("data", (d) => {
    process.stdout.write(d);
  });
});
```

Reading files in NodeJS? Piece of cake. No sweat.

READ FILE AS TEXT

CHAPTER-H/2-READ-TEXT.HTML

```
<script>
function readFile () {
  // (A) GET SELECTED FILE
  let selected = document.getElementById("demoPick")
                 .files[0];

  // (B) READ SELECTED FILE
  let reader = new FileReader();
  reader.addEventListener("loadend", () => {
    document.getElementById("demoShow").innerHTML =
    reader.result;
  });
  reader.readAsText(selected);
}
</script>

<div id="demoShow"></div>
<input type="file" value="Choose TXT File" id="demoPick"
       accept="text/plain" onchange="readFile()"/>
```

That's right, your eyes did not deceive you. This is client-side Javascript and HTML, it is possible to read files on the user's device without uploading anything to the server-side.

- Start by creating an `<input type="file"/>`.
- In the Javascript, we can get the selected file(s) with `document.getElementById(FIELD).files`.
- Lastly, we only need to create a `FileReader()` object and use `readAsText()` to read the selected file.

READ FILE AS DATA URL (BASE 64 ENCODED)

CHAPTER-H/3-READ-IMG.HTML

```
<script>
function readFile () {
  // (A) GET SELECTED FILE
  let selected = document.getElementById("demoPick")
                 .files[0];

  // (B) READ SELECTED FILE
  let reader = new FileReader();
  reader.addEventListener("load", () => {
    document.getElementById("demoShow").src =
    reader.result;
  });
  reader.readAsDataURL(selected);
}
</script>

<img id="demoShow"/>
<input type="file" value="Choose JPG File" id="demoPick"
       accept="image/*" onchange="readFile()"/>
```

We can also directly read a file, represented in the base 64 encoding – Just use `readAsDataURL()` instead of `readAsText()`.

READ FILE AS BINARY DATA

CHAPTER-H/4-READ-BIN.HTML

```
<script>
function readFile (mode) {
  // (A) GET SELECTED FILE
  let selected = document.getElementById("demoPick")
                 .files[0];
```

```
  // (B) READ SELECTED FILE
  let reader = new FileReader();
  reader.addEventListener("load", () => {
    console.log(reader.result);
  });
  if (mode == 1) { reader.readAsBinaryString(selected); }
  else { reader.readAsArrayBuffer(selected); }
}
</script>

<input type="file" value="Choose File" id="demoPick"/>
<input type="button" value="Read As BIN"
       onclick="readFile(1)"/>
<input type="button" value="Read As Array Buffer"
       onclick="readFile(2)"/>
```

Lastly, we have `readAsBinaryString()` and `readAsArrayBuffer()` if you need to read a file in its "raw binary form".

CALLING THE FILE PICKER PROGRAMMATICALLY

CHAPTER-H/5-CALL-PICKER.HTML

```
<script>
async function readFile() {
  // (A) OPEN FILE PICKER
  [handler] = await window.showOpenFilePicker({
    excludeAcceptAllOption: true,
    types: [{
      description: "Text",
      accept: { "text/plain": [".txt", ".text"] }
    }]
  });
```

```
  // (B) GET SELECTED FILE
  let selected = await handler.getFile();

  // (C) READ SELECTED FILE
  let reader = new FileReader();
  reader.addEventListener("loadend", () => {
    document.getElementById("demoShow").innerHTML =
    reader.result;
  });
  reader.readAsText(selected);
}
</script>

<div id="demoShow"></div>
<input type="button" value="Choose TXT File"
       onclick="readFile()"/>
```

Traditionally, the only way to bring up the "choose a file" dialog box is from the `<input type="file"/>` itself. But now, we can also use `window.showOpenFilePicker()` to open up the file picker dialog box. Take note though, this will only work on Chrome, Edge, and Opera at the time of writing.

CLIENT-SIDE RESTRICTION – EXPLICIT PERMISSION

Noticed how the user must always choose and open a file before we can read it? Yep, that's the restriction in client-side Javascript. We don't have direct access to any of the user's files.

LINKS & REFERENCES

- File System Access API – MDN
- File Reader – MDN
- Show Open File Picker – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- File Reader
- Show Open File Picker

- CHAPTER I -

WRITE FILES

WRITING FILES IN NODEJS

CHAPTER-I/1-NODE-WRITE-FILE.JS

```
// (A) FILE SYSTEM MODULE
const fs = require("fs");

// (B) WRITE TO FILE (ASYNC)
fs.writeFile("demoA.txt", "Hello World", (err) => {});

// (C) APPEND TO FILE (ASYNC)
fs.appendFile("demoA.txt", "Goodbye World", (err) => {});

// (D) WRITE FILE (SYNC)
fs.writeFileSync("demoB.txt", "Hello World");

// (E) APPEND FILE (SYNC)
fs.appendFileSync("demoB.txt", "Goodbye World");

// (F) WRITE FILE STREAM
var data = ["Apple", "Banana", "Cherry", "Durian"];
    stream = fs.createWriteStream("demoC.txt");
// stream = fs.createWriteStream("demoC.txt",
//          {"flags": "a"});
for (let i of data) { stream.write(i + "\r\n"); }
stream.end();
```

Writing files in NodeJS? Piece of cake. No sweat.

BLOB DOWNLOAD

CHAPTER-I/2-BLOB-DOWNLOAD.HTML

```
<script>
function writeFile () {
  // (A) CREATE BLOB OBJECT
  var myBlob = new Blob(["CONTENT"],
```

```
                {type: "text/plain"});

  // (B) CREATE DOWNLOAD LINK
  var url = window.URL.createObjectURL(myBlob);
  var anchor = document.createElement("a");
  anchor.href = url;
  anchor.download = "demo.txt";

  // (C) "FORCE DOWNLOAD"
  // NOTE: MAY NOT ALWAYS WORK DUE TO BROWSER SECURITY
  // BETTER TO LET USERS CLICK ON THEIR OWN
  anchor.click();
  window.URL.revokeObjectURL(url);
  document.removeChild(anchor);
}
</script>

<input type="button" value="Write File"
       onclick="writeFile()"/>
```

You guys who are looking to "directly save a file" into the system – That cannot be done with client-side Javascript due to security restrictions. This is one of the better cross-browser solutions, we create a blob object, and offer it as a download via a hidden HTML anchor tag.

WRITABLE FILE STREAM

CHAPTER-I/3-WRITABLE-STREAM.HTML

```
<script>
async function writeFile () {
  // (A) CREATE BLOB OBJECT
  var myBlob = new Blob(["CONTENT"],
                {type: "text/plain"});
```

```
  // (B) FILE HANDLER & FILE STREAM
  const fileHandle = await window.showSaveFilePicker({
    types: [{
      description: "Text file",
      accept: {"text/plain": [".txt"]}
    }]
  });
  const fileStream = await fileHandle.createWritable();

  // (C) WRITE FILE
  await fileStream.write(myBlob);
  await fileStream.close();
}
</script>

<input type="button" value="Write File"
       onclick="writeFile()"/>
```

Alternatively, this is a more modern approach.

- Start by creating the blob object.
- Open a "save file as" dialog with `fileHandle = window.showSaveFilePicker()`. Take note though, this will only work on Chrome, Edge, and Opera at the time of writing.
- Create a writable file stream `fileStream = fileHandle.createWritable()` to the file that the user chose.
- Write to the file and close it – `fileStream.write(BLOB)`, `fileStream.close()`.

LOCAL STORAGE BASE 64 ENCODED STRING

CHAPTER-I/4-BLOB-LOCAL-STORAGE.HTML

```
// (A) CREATE BLOB OBJECT
var myBlob = new Blob(["CONTENT"], {type: "text/plain"});

// (B) BLOB TO BASE 64 ENCODED
var reader = new FileReader();
reader.readAsDataURL(myBlob);
reader.onloadend = function() {
  var encoded = reader.result;

  // (C) STORE INTO LOCAL STORAGE
  localStorage.setItem("myBlob", encoded);
  console.log("OK");
};
```

Remember `localStorage`? Yep, here is the last alternative where we turn a blob object into a base 64 encoded string, and store it into the `localStorage`... It may not offer a lot of storage space, but at least this is the most "silent" method – We can just directly save without having to nag "save as" at the user.

BASE 64 TO BLOB

CHAPTER-I/5-BASE64-BLOB.HTML

```
async function decode () {
  // (A) GET BASE 64 ENCODED STRING
  var encoded = localStorage.getItem("myBlob");

  // (B) PARSE BACK INTO BLOB
  const res = await fetch(encoded);
  var myBlob = await res.blob();
```

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

Credits to Ionic Framework for this snippet, the base 64 encoded string can pretty much be used "as it is". But if you need to parse it back into a blob object – Simply pass the encoded string into `fetch()` and call `blob()`.

LINKS & REFERENCES

- File System Access API – MDN
- Writable Streams – MDN
- Show Save File Picker – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Create Writable
- Show Save File Picker

- CHAPTER J -

FETCH

MODERN HTTP COMMUNICATIONS

Once upon a time, there was `XMLHttpRequest`. It still works great, but it became limited in some ways as the Cyber world moved forward with crazy security and processes. To address the needs of modern asynchronous HTTP calls, the Fetch API was introduced – Yes, this is the same `fetch()` that we used in the earlier service worker chapter.

BASIC FETCH CONTENT

CHAPTER-J/1A-FETCH-CONTENT.HTML

```
<script>
function fetchHTML () {
  // (A) FETCH HTML FILE
  fetch("1b-dummy.html")

  // (B) RETURN RESPONSE AS TEXT
  .then((response) => {
    if (response.status!=200)
      { throw new Error("Bad server response"); }
    return response.text();
  })

  // (C) PUT FETCHED CONTENT INTO <DIV>
  .then((result) => {
    document.getElementById("demoShow")
      .innerHTML = result;
  });
}
</script>

<!-- (D) TEST DIV & BUTTON -->
<div id="demoShow"></div>
```

```
<input type="button" value="Fetch" onclick="fetchHTML()"/>
```

CHAPTER-J/1B-DUMMY.HTML

```
<p>This is loaded via Fetch!</p>
```

For a start, this is how a "general" fetch request looks like – Typically a three steps process.

A) Fire the fetch request itself. Specify the target URL, set the options (more on that later).

B) Parse and return the server response – As text, as JSON, or as binary data.

C) Do something with the parsed result.

FETCH SEND DATA

FETCH POST

CHAPTER-J/2A-FETCH-POST.HTML

```
<!-- (A) HTML FORM -->
<form id="form" onsubmit="return fetchPOST()">
  <input type="text" name="name" value="Jon Doe"
         required/>
  <input type="email" name="email" value="jon@doe.com"
         required/>
  <input type="submit" value="Go!"/>
</form>

<script>
function fetchPOST () {
  // (B) INIT FETCH POST
```

```
  fetch("YOUR-SERVER-SCRIPT", {
    method: "POST",
    body: new FormData(document.getElementById("form"))
  })

  // (C) SERVER RESPONSE
  .then((result) => {
    if (result.status != 200)
      { throw new Error("Bad Server Response"); }
    return result.text();
  })
  .then((response) => { console.log(response); })
  .catch((error) => { console.log(error); });
  return false;
}
</script>
```

Remember from 1 minute ago that we mentioned "pass options into `fetch()`"? Yes, just like the good old `XMLHttpRequest`, we can set options to let `fetch()` submit as a POST request and append `FormData` along with it.

FETCH GET

CHAPTER-J/2B-FETCH-GET.HTML

```
<!-- (A) HTML FORM -->
<form id="form" onsubmit="return fetchGET()">
  <input type="text" name="name" value="Jon Doe"
         required/>
  <input type="email" name="email" value="jon@doe.com"
         required/>
  <input type="submit" value="Go!"/>
</form>
```

```
<script>
function fetchGET () {
  // (B) INIT FETCH GET
  fetch("YOUR-SERVER-SCRIPT?" + new URLSearchParams(
    new FormData(document.getElementById("form"))
  ).toString())

  // (C) SERVER RESPONSE
  .then((result) => {
    if (result.status != 200)
      { throw new Error("Bad Server Response"); }
    return result.text();
  })
  .then((response) => { console.log(response); })
  .catch((error) => { console.log(error); });
  return false;
}
</script>
```

If you are wondering how to do the "GET" counterpart – Just append a query string to the URL.

FETCH JSON

FETCH SEND JSON DATA

CHAPTER-J/3A-FETCH-SEND-JSON.HTML

```
<script>
function fetchSendJSON () {
  // (A) DATA OBJECT
  var data = {
    name : "Jon Doe",
    email : "jon@doe.com"
  };
```

```
  // (B) FETCH POST JSON
  fetch("YOUR-SERVER-SCRIPT", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(data)
  })

  // (C) SERVER RESPONSE
  .then((result) => {
    console.log(result);
    if (result.status != 200)
      { throw new Error("Bad Server Response"); }
    return result.text();
  })
  .then((response) => { console.log(response); })
  .catch((error) => { console.log(error); });
}
</script>

<input type="button" value="JSON Send"
       onclick="fetchSendJSON()"/>
```

If you need to work with arrays and objects – We can also directly send as JSON using `fetch()`.

FETCH RECEIVE JSON DATA

CHAPTER-J/3B-FETCH-GET-JSON.HTML

```
<script>
function fetchGetJSON () {
  // (A) FETCH REQUEST
  fetch("3c-dummy.json")

  // (B) RETURN SERVER RESPONSE AS JSON
  .then((result) => {
```

```
    if (result.status != 200)
      { throw new Error("Bad Server Response"); }
    return result.json();
  })
  .then((response) => { console.log(response); })
  .catch((error) => { console.log(error); });
}
</script>

<input type="button" value="JSON Get"
       onclick="fetchGetJSON()"/>
```

Remember from the first example that we can parse the server response as text, JSON, or binary? Yes, this is a small improvement over the traditional `XMLHttpRequest`, we do not need to manually JSON parse the results.

FETCH HTTP BASIC AUTH

CHAPTER-J/4-FETCH-AUTH.HTML

```
<script>
function fetchAuth () {
  // (A) URL & CREDENTIALS
  var url = "protected/secret.html",
      credentials = btoa("USER:PASS");

  // (B) FETCH WITH HTTP AUTH
  fetch (url, {
    headers: {"Authorization": `Basic ${credentials}`}
  })

  // (C) SERVER RESPONSE
  .then((result) => {
    if (result.status != 200)
```

```
      { throw new Error("Bad Server Response"); }
    return result.text();
  })
  .then((response) => { console.log(response); })
  .catch((error) => { console.log(error); });
}
</script>

<input type="button" value="Fetch Auth"
       onclick="fetchAuth()"/>
```

So far so good? The previous examples have been rather "basic". Let us move the level up a little – Yes, we can also send an `Authorization` header to challenge HTTP basic authentication with `fetch()`.

FETCH AS BINARY

CHAPTER-J/5-FETCH-BINARY.HTML

```
<script>
function fetchToCanvas () {
  // (A) FETCH IMAGE & RETURN AS BINARY
  fetch ("orange.jpg")
  .then((result) => {
    if (result.status != 200)
      { throw new Error("Bad Server Response"); }
    return result.blob();
  })

  // (B) BLOB TO BASE64 ENCODE TO IMAGE TAG
  .then((blob) => {
    var reader = new FileReader() ;
    reader.onload = function () {
      var img = new Image();
      img.src = this.result;
```

```
      document.getElementById("demoShow")
        .appendChild(img);
    };
    reader.readAsDataURL(blob);
  })
  .catch((error) => { console.log(error); });
}
</script>

<div id="demoShow"></div>
<input type="button" value="Fetch Canvas"
       onclick="fetchToCanvas()"/>
```

I know, this is not the smartest example, but it serves its purpose well enough – We can fetch a file as binary data, encode it, and directly embed into an HTML page.

LINKS & REFERENCES

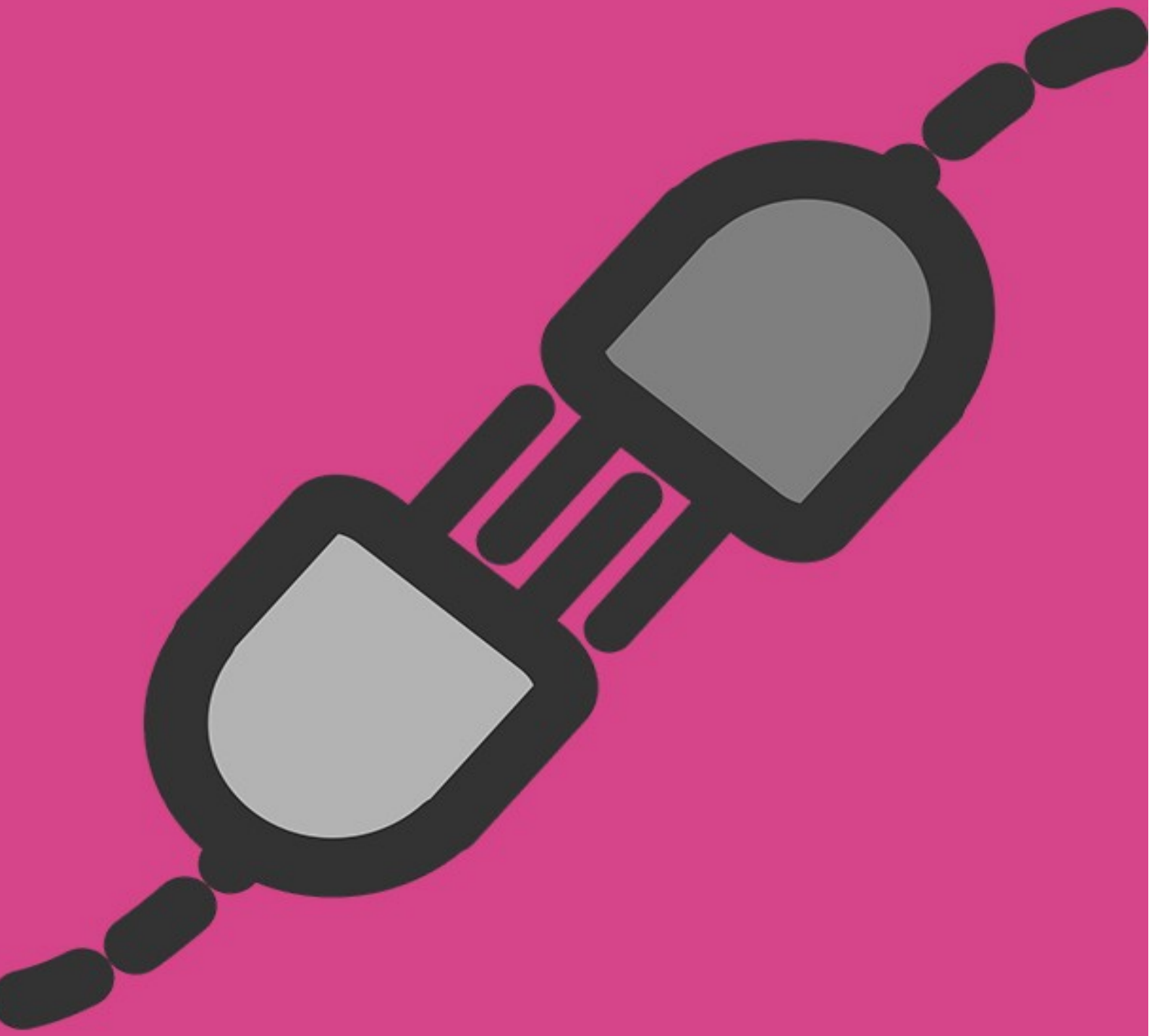
- Fetch API – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Fetch API

- CHAPTER K -

WEB SOCKET



WHAT IS A SOCKET?

If you do not already know, HTTP is stateless and non-persistent.

- Connect to the server, request for resource (page, document, image, video, audio, etc...)
- Download the requested resource and close the connection. The end, case closed.

HTTP works great for short, simple, anonymous requests. But it sucks when it comes to live applications (live chat, live updates, live stream, online games) – How is it going to communicate with the server continuously? Spam reload every microsecond to check for updates?

So for live applications, it is smarter to use a persistent "will not disconnect until you say so" connection – This is called a "socket".

BASIC WEB SOCKET

NODEJS WEB SOCKET SERVER

CHAPTER-K/1A-SERVER.JS

```
// (A) CREATE WEBSOCKET SERVER AT PORT 8080
const ws = require("ws"),
      wss = new ws.Server({ port: 8080 });

// (B) ON CLIENT CONNECT
wss.on("connection", (socket, req) => {
  // (B1) SEND MESSAGE TO CLIENT
  socket.send("Welcome!");

  // (B2) ON RECEIVING MESSAGE FROM CLIENT
  socket.on("message", (msg) => {
```

```
    let message = msg.toString(); // MSG IS BUFFER OBJECT
    console.log(message);
  });

  // (B3) ON CLIENT DISCONNECT
  socket.on("close", (code, reason) => {
    console.log(code);
    console.log(reason);
  });
});

// (C) NOT-SO-CRITICAL EVENTS
wss.on("listening", () => { console.log("READY"); });
wss.on("close", () => { console.log("STOPPED"); });
wss.on("error", (err) => { console.log(err); });
```

To get started with this "socket" thing, let us create a simple web socket server using NodeJS.

- The `ws` module is required. Simply navigate to your project folder in the command line and run `npm install ws`.
- **(A)** We create a `new ws.Server({ port: 8080 })`, that will open the socket at `localhost:8080`.
- **(B & C)** Listen and manage the socket server.
  - `connection` When a client connects to the server.
  - `listening` The socket server is ready and listening.
  - `close` When the server is closed.
  - `error` An error has occurred.
- **(B)** This dummy script pretty much just sends a "welcome" message to the client on connect, and show all messages received

from the client.

- When you are ready, run this script in the command line – `node 1a-server.js`. Press `ctrl-c` (`cmd-c` on Mac) to stop running.

WEB SOCKET CLIENT

CHAPTER-K/1B-CLIENT.HTML

```
// (A) CONNECT TO WEB SOCKET SERVER
const socket = new WebSocket("ws://localhost:8080");

// (B) ON CONNECTING TO THE SERVER
socket.addEventListener("open", () => {
  socket.send("Hello Server!"); // SEND MESSAGE TO SERVER
});

// (C) ON RECEIVING MESSAGE FROM SERVER
socket.addEventListener("message", (evt) => {
  console.log(evt.data);
});

// (D) ON CONNECTION CLOSE
socket.addEventListener("close", () => {
  console.log("Connection Closed");
});

// (E) ON ERROR
socket.addEventListener("error", (err) => {
  console.log(err);
});
```

Don't think this needs a lot of explanation.

- **(A)** `new WebSocket("ws://localhost:8080")` connects to the NodeJS socket server we created earlier.

- **(B to E)** Listen to the socket and manage the connection.
  - `open` On connecting to the server. In this example, we just send "Hello Server" to the server.
  - `message` On receiving a message from the server.
  - `close` When the connection is closed.
  - `error` Handle errors here.

That covers the basics of a web socket connection. But of course, this dummy example does nothing productive… Let's go through a live chat application next.

LIVE CHAT WITH WEB SOCKET

NODEJS LIVE CHAT SERVER

CHAPTER-K/2A-CHAT-SERVER.JS

```
// (A) INIT + CREATE WEBSOCKET SERVER AT PORT 8080
var ws = require("ws"),
    wss = new ws.Server({ port: 8080 }),
    users = {};

// (B) ON CLIENT CONNECT
wss.on("connection", (socket, req) => {
  // (B1) REGISTER CLIENT
  let id = 0;
  while (true) {
    if (!users.hasOwnProperty(id))
      { users[id] = socket; break; }
    id++;
  }
```

```
  // (B2) DEREGISTER CLIENT ON DISCONNECT
  socket.on("close", () => { delete users[id]; });

  // (B3) FORWARD MESSAGE TO ALL ON RECEIVING MESSAGE
  socket.on("message", (msg) => {
    let message = msg.toString();
    for (let u in users) { users[u].send(message); }
  });
});
```

This is pretty much a "modified example" of the basic web socket server.

- When a client connects to the socket server, we store the connection in `users`.
- On receiving a message from any client, we simply forward it to all connected `users`.

Yep, that's all it takes to create a simple live chat server.

LIVE CHAT HTML PAGE

CHAPTER-K/2B-CHAT-CLIENT.HTML

```
<!-- (A) CHAT HISTORY -->
<div id="chatShow"></div>

<!-- (B) CHAT FORM -->
<form id="chatForm" onsubmit="return chat.send();">
  <input id="chatMsg" type="text" required disabled/>
  <input id="chatGo" type="submit" value="Go" disabled/>
</form>
```

- We show the chat messages in `<div id="chatShow">`.
- Use `<form id="chatForm">` to send a message.

LIVE CHAT CLIENT JAVASCRIPT

CHAPTER-K/2B-CHAT-CLIENT.JS

```
var chat = {
  // (A) INIT CHAT
  name : null, // USER'S NAME
  socket : null, // CHAT WEBSOCKET
  ewrap : null, // HTML CHAT HISTORY
  emsg : null, // HTML CHAT MESSAGE
  ego : null, // HTML CHAT GO BUTTON
  init : () => {
    // (A1) GET HTML ELEMENTS
    chat.ewrap = document.getElementById("chatShow");
    chat.emsg = document.getElementById("chatMsg");
    chat.ego = document.getElementById("chatGo");

    // (A2) USER'S NAME
    chat.name = prompt("What is your name?", "John");
    if (chat.name == null || chat.name=="")
      { chat.name = "Mysterious"; }

    // (A3) CONNECT TO CHAT SERVER
    chat.socket = new WebSocket("ws://localhost:8080");

    // (A4) ON CONNECT - ANNOUNCE "I AM HERE" TO THE WORLD
    chat.socket.addEventListener("open", () => {
      chat.controls(1);
      chat.send("Joined the chat room.");
    });

    // (A5) ON RECEIVE MESSAGE - DRAW IN HTML
    chat.socket.addEventListener("message", (evt) => {
      chat.draw(evt.data);
    });

    // (A6) ON ERROR & CONNECTION LOST
```

```
    chat.socket.addEventListener("close", () => {
      chat.controls();
      alert("Websocket connection lost!");
    });
    chat.socket.addEventListener("error", (err) => {
      chat.controls();
      console.log(err);
      alert("Websocket connection error!");
    });
  },

  // (B) TOGGLE HTML CONTROLS
  controls : (enable) => {
    if (enable) {
      chat.emsg.disabled = false;
      chat.ego.disabled = false;
    } else {
      chat.emsg.disabled = true;
      chat.ego.disabled = true;
    }
  },

  // (C) SEND MESSAGE TO CHAT SERVER
  send : (msg) => {
    if (msg == undefined) {
      msg = chat.emsg.value;
      chat.emsg.value = "";
    }
    chat.socket.send(JSON.stringify({
      name: chat.name,
      msg: msg
    }));
    return false;
  },

  // (D) DRAW MESSAGE IN HTML
  draw : (msg) => {
```

```
    // (D1) PARSE JSON
    msg = JSON.parse(msg);
    console.log(msg);

    // (D2) CREATE NEW ROW
    let row = document.createElement("div");
    row.className = "chatRow";
    row.innerHTML = `<div class="chatName">
      ${msg["name"]}</div> <div class="chatMsg">
      ${msg["msg"]}</div>`;
    chat.ewrap.appendChild(row);

    // AUTO SCROLL TO BOTTOM MAY NOT BE THE BEST...
    window.scrollTo(0, document.body.scrollHeight);
  }
};
window.addEventListener("DOMContentLoaded", chat.init);
```

What the heck is this!? Keep calm and study slowly:

- **(A & B)** On page load, `chat.init()` will run. Pretty straightforward, get the user's name, connect to the chat server, and enable the HTML "send message" form.
- **(C)** Send a message to the chat server. Duh.
- **(A5 & D)** Remember that the chat server simply forwards a message to all connected clients? All we need to do here, is to draw any received message into `<div id="chatShow">`.

The end. It's not really that difficult… Just long-winded.

MORE USES

So what other uses do web sockets have? There's plenty I can think of:

- Live scoreboard.
- Live bidding.
- Live streaming – Yes, web socket can stream raw binary data… It's a persistent connection to begin with.
- File exchange.
- Group meeting.
- Live support.
- Live restaurant orders management.
- Live tracking.

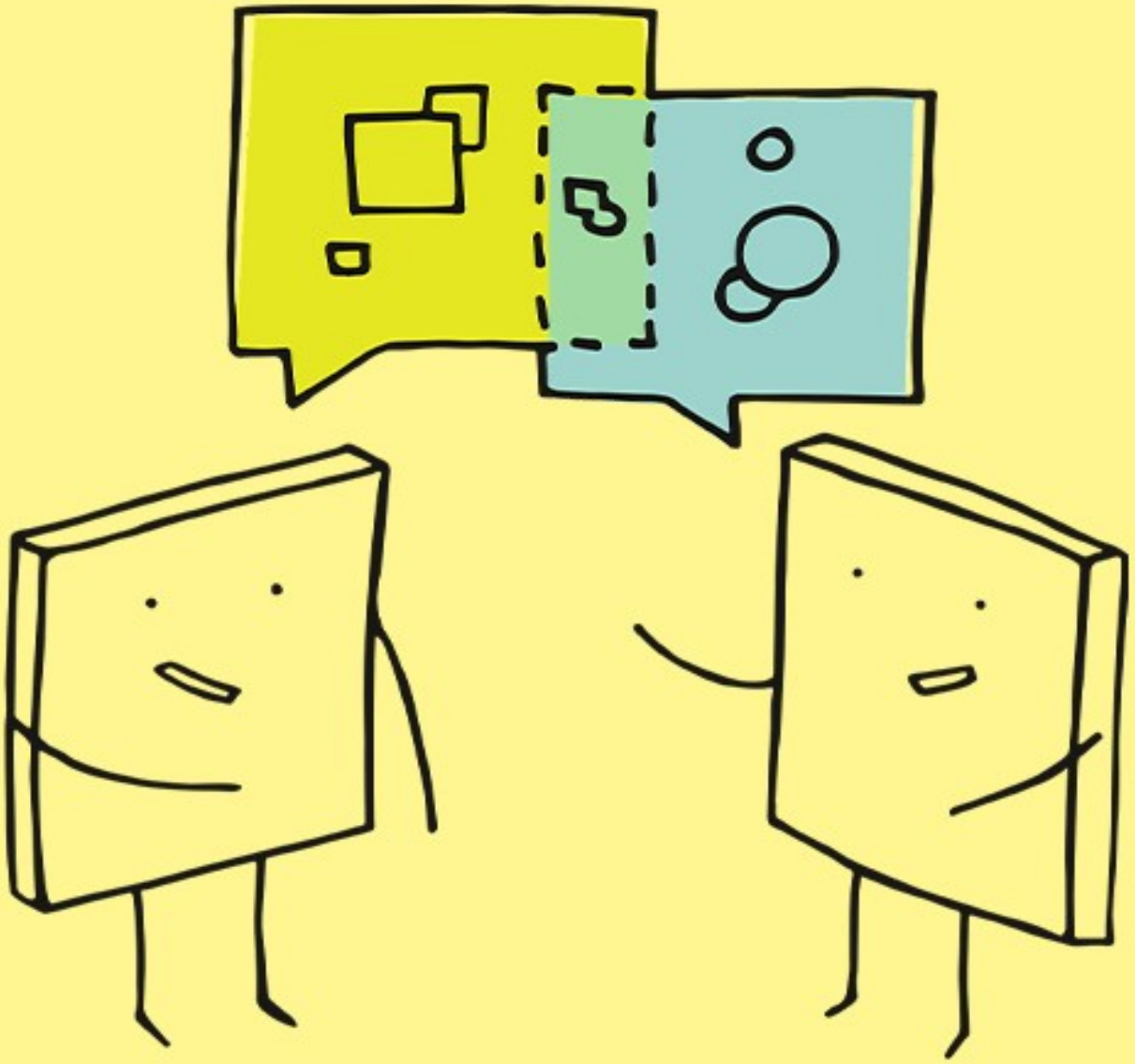
This list can go on forever, but I am going to end here...

LINKS & REFERENCES

- WebSocket API – MDN
- ws Module Documentation – GitHub

COMPATIBILITY CHECKS (WITH CANIUSE)

- WebSocket



- CHAPTER L -

WEBRTC (PEER TO PEER)

DIRECT EXCHANGE

RTC stands for "real time communication", and yes, this refers to a real-time peer-to-peer connection with web technologies. However, a peer server is still required to facilitate the handshaking – Thereafter, data exchange between the peers are done without going through the server.

VERY SIMPLE PEER-TO-PEER

STEP 1) PEER LIBRARIES

Working with the "native" WebRTC is... raw and painful. To speed things up, we will be using the PeerJS and PeerServer libraries.

- The client-side PeerJS can be easily loaded from CDNJS.
- The Peer Server can be downloaded with `npm install peer`.

STEP 2) NODEJS PEER SERVER

CHAPTER-L/1A-SERVER.JS

```
const { PeerServer } = require("peer");
const peerServer = PeerServer({
  port: 9000,
  path: "/myapp"
});
```

A basic peer server is as simple as that. Running this will deploy it at `http://localhost:9000/myapp`.

STEP 3) PEER CLIENT A

CHAPTER-L/1B-PEER-A.HTML

```
// (B1) HANDSHAKE WITH PEER SERVER
const peer = new Peer("PEER-A", {
  host: "localhost",
  port: 9000,
  path: "/myapp"
});

// (B2) ON RECEIVING MESSAGE FROM OTHER PEERS
peer.on("connection", (conn) => {
  conn.on("data", (data) => { console.log(data); });
});
```

Go ahead, access `http://localhost/1b-peer-a.html` in your browser. This should be self-explanatory.

- **(B1)** `new Peer(ID, OPTIONS)` will do a handshake with the peer server. Take note of the `ID` here, we manually set it to `PEER-A`, and this must be a unique user ID. More on that later.
- **(B2)** When another peer connects, we show the message that is sent over.

STEP 4) PEER CLIENT B

CHAPTER-L/1C-PEER-B.HTML

```
// (B1) HANDSHAKE WITH PEER SERVER
const peer = new Peer("PEER-B", {
  host: "localhost",
  port: 9000,
  path: "/myapp"
```

```
});

// (B2) READY - CONNECT & SEND MESSAGE TO PEER A
peer.on("open", (id) => {
  var conn = peer.connect("PEER-A");
  conn.on("open", () => {
    conn.send("Hi from PEER-B!");
  });
});
```

- **(B1)** This is the very same "handshake with peer server", but take note that we are assigning `PEER-B` as the user ID this time.
- **(B2)** On successful handshake with the server, we open a connection to `PEER-A` and send a "hi" message.

Now, open a new window or use a different browser. Access `http://localhost/1c-peer-b.html` and see the "hi" message appear in the console panel of `PEER-A`. Yes, the message is sent directly from peer-to-peer. The server is not involved.

SIMPLE PEER-TO-PEER CHAT

Next, let us upgrade the very simple P2P exchange into a simple P2P chat app. Yes, a reminder that this is peer-to-peer (one to one). This is different from the previous WebSocket chat that is broadcast (one to many).

P2P CHAT HTML

CHAPTER-L/2A-CHAT.HTML

```
<!-- (B) HTML INTERFACE -->
<form id="chatForm">
  <input type="text" id="chatTxt" disabled required/>
  <input type="submit" id="chatGo" disabled value="Send"/>
  <input type="button" id="chatCx" disabled
         value="Disconnect"/>
</form>
<div id="chatView"></div>
```

For this simple chat, we have a simple HTML interface.

- `chatForm` – Chat form on top. With a text box, send and disconnect buttons.
- `chatView` – Chat messages below.

P2P CHAT JAVASCRIPT – THE PROPERTIES

CHAPTER-L/2C-CHAT.JS

```
var chat = {
  // (A) PROPERTIES
  // (A1) CHAT SERVER
  peer : null, conn : null,
  opt : { host: "localhost", port: 9000, path: "/myapp" },

  // (A2) HTML ELEMENTS
  hView : null, hForm : null, hTxt : null,
  hGo : null, hCx : null,
}
```

First, we begin by defining a `chat` object and some properties. All of these should be self-explanatory.

P2P CHAT JAVASCRIPT – INITIALIZE

CHAPTER-L/2C-CHAT.JS

```
// (B) INIT
init : (id, after) => {
  // (B1) GET + SET HTML ELEMENTS
  chat.hView = document.getElementById("chatView");
  chat.hForm = document.getElementById("chatForm");
  chat.hTxt = document.getElementById("chatTxt");
  chat.hGo = document.getElementById("chatGo");
  chat.hCx = document.getElementById("chatCx");
  chat.hForm.onsubmit = chat.send;
  chat.hCx.onclick = chat.disconnect;

  // (B2) HANDSHAKE WITH PEER SERVER
  chat.peer = new Peer(id, chat.opt);

  // (B3) READY
  chat.peer.on("open", (id) => {
    if (after) { after(); }
  });

  // (B4) ON PEER CONNECT
  chat.peer.on("connection", (conn) => {
    chat.conn = conn;
    chat.connector();
  });

  // (B5) ALL OTHER PEER EVENTS
  chat.peer.on("close", () => { console.log("close"); });
  chat.peer.on("disconnected", () =>
  { console.log("dc"); });
  chat.peer.on("error", (err) => { console.log(err); });
  // chat.peer.on("call", () => { console.log("call"); });
}
```

We run the `init()` function on page load to start the chat. This is just long-winded, not complicated.

- **(B1)** Get the HTML elements, set the "chat actions".
- **(B2 To B4)** Do the peer server handshake as usual. But keep the peer object in `chat.peer` and connection in `chat.conn` to better manage the chat app.
- **(B5)** Do take some time to go through the peer events. Read their official documentation if you want.

P2P CHAT JAVASCRIPT – PEER CONNECT & DISCONNECT

CHAPTER-L/2C-CHAT.JS

```
// (C) CONNECT TO PEER
connect : (id) => {
  chat.conn = chat.peer.connect(id);
  chat.connector();
},

// (D) ATTACH PEER LISTENERS
connector : () => {
  chat.conn.on("open", () => { chat.tog(); });
  chat.conn.on("close", () => { chat.tog(true); });
  chat.conn.on("data", chat.recv);
  chat.conn.on("error", (err) => { console.log(err); });
},

// (D) DISCONNECT PEER
disconnect : () => { chat.conn.close(); }
```

These are the same with the previous very simple example once again, but broken into different functions to better manage the connection.

P2P CHAT JAVASCRIPT – DRAW MESSAGE

CHAPTER-L/2C-CHAT.JS

```
// (E) HELPER - DRAW MESSAGE
draw : (txt, css) => {
  let row = document.createElement("div");
  row.innerHTML = txt;
  row.className = css;
  chat.hView.appendChild(row);
},

// (F) SEND MESSAGE
send : () => {
  chat.conn.send(chat.hTxt.value);
  chat.draw(chat.hTxt.value, "send");
  chat.hTxt.value = "";
  return false;
},

// (G) RECEIVE MESSAGE
recv : (txt) => {
  chat.draw(txt, "recv");
}
```

Self-explanatory. Draw the sent/received messages in HTML.

P2P CHAT JAVASCRIPT – LOCK/UNLOCK INTERFACE

CHAPTER-L/2C-CHAT.JS

```
// (H) TOGGLE INTERFACE
tog : (lock) => {
  if (lock) {
    chat.hForm.disabled = true;
    chat.hTxt.disabled = true;
```

```
    chat.hGo.disabled = true;
    chat.hCx.disabled = true;
  } else {
    chat.hForm.disabled = false;
    chat.hTxt.disabled = false;
    chat.hGo.disabled = false;
    chat.hCx.disabled = false;
  }
}
```

This final bit is used to lock the HTML interface when no peers are connected, enable the chat when a peer is connected.

LAUNCH P2P CHAT JAVASCRIPT

CHAPTER-L/2A-CHAT.HTML

```
window.addEventListener("load", () => {
  chat.init("PEER-A");
});
```

Now that things are ready, launch `http://localhost/2a-chat.html` in your browser. This will wait for a peer connection just like the previous example.

CHAPTER-L/2B-CHAT.HTML

```
window.addEventListener("load", () => {
  chat.init("PEER-B", () => {
    chat.connect("PEER-A");
  });
});
```

Next, open `http://localhost/2b-chat.html` in another browser or window. This will connect to `PEER-A` and the chat example is ready.

MYSTERY OF THE USER ID

Must we always manually assign `PEER-A` and `PEER-B`? How does that even work? This is a difficult question to answer, as every app development is different. Here are a few examples to better illustrate:

- If you already have a user database, it makes sense to tie in and use it – `new Peer(USER-ID-IN-DATABASE, OPTIONS)`.
- `new Peer(null, OPTIONS)` will automatically generate a random ID that you can obtain in `peer.on("open", (id) => { … })`. Maybe use it to create a URL for the user to share – `http://site.com/chatwith/?u=ID`
- If not, you can create your own concept of a "private room" or sorts, generate and set your own unique ID.

WEBRTC AUDIO VIDEO STREAM

If you have noticed that `peer.on("call")` event, yes, we can do audio and video calls with WebRTC. But we will hold on and cover that in the later chapter on screen sharing and webcams.

LINKS & REFERENCES

- WebRTC API – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- WebRTC Compatibility – CanIUse

- CHAPTER M -

PUSH NOTIFICATIONS

STEP-BY-STEP PUSH

It is no secret that we can do push notifications in browsers. But the actual implementation is not the easiest, as it involves multiple different technologies. So instead of throwing out "one massive and confusing example", let us take a step-by-step approach instead.

STEP 1) DOWNLOAD NODE MODULES

First, let us get all the required Node modules first. Open the command line, navigate to your project folder, and run `npm install express body-parser web-push`.

STEP 2) GENERATE VAPID KEYS

CHAPTER-M/2-VAPID-KEYS.JS

```
const vapidKeys = require("web-push").generateVAPIDKeys();
console.log(vapidKeys);
```

Next, run this snippet to get your own pair of public/private keys – `node 2-vapid-keys.js`. VAPID stands for Voluntary Application Server Identification. Basically security and authentication stuff, so people don't hijack your push notifications to send funky stuff.

Just hold on to the keys first, we will use it later. But a quick reminder, always use the private key on the server-side only. Do not expose it publicly.

STEP 3) CLIENT-SIDE

GET PERMISSION TO SHOW NOTIFICATIONS

CHAPTER-M/3-CLIENT.HTML

```
// (A) OBTAIN USER PERMISSION
// (A1) ASK FOR PERMISSION
if (Notification.permission === "default") {
  Notification.requestPermission().then((perm) => {
    if (Notification.permission === "granted") {
      regWorker().catch((err) =>
      { console.error(err); } );
    } else { alert("Please allow notifications."); }
  });
}

// (A2) GRANTED
else if (Notification.permission === "granted") {
  regWorker().catch((err) => { console.error(err); } );
}

// (A3) DENIED
else { alert("Please allow notifications."); }
```

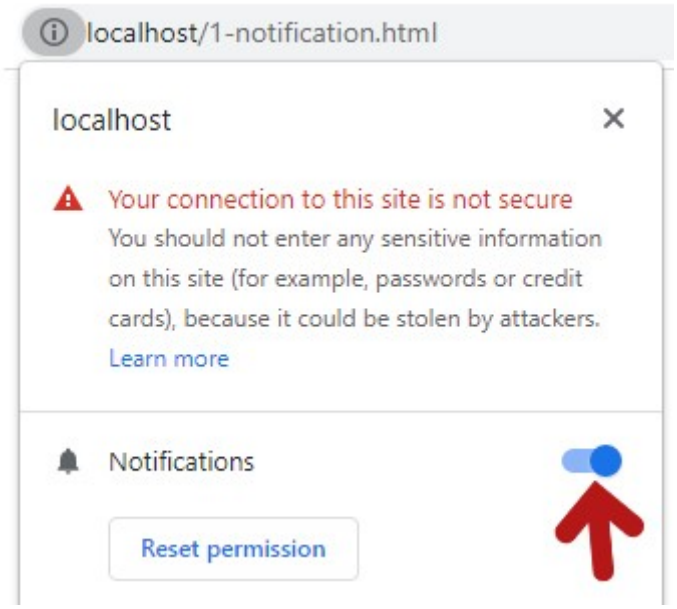
The first thing we do on the HTML page is to get the user's permission to display notifications. Very simple, just a couple of things to take note:

- `Notification.permission` contains the status.
  - `default` – The user has not allowed nor denied permission.
  - `granted` – The user allowed notifications to be shown.
  - `denied` – The user has denied permission.
- If the user has not given permission, we use `Notification.requestPermission()` to ask for it.

- We proceed to register the service worker when the user has given permission to display push notifications.

PERMISSIONS & RESTRICTIONS

The Notification API requires a valid `https://` website to work, but as always, `http://localhost` is an exception. Also, once the user denied permission, `Notification.requestPermission()` will not show the "grant permission" dialog box again. You will have to show your own "manual remedy instructions", or just live with it.



P.S. If you do not already know, click on the website icon in the URL bar to bring up the permissions.

REGISTER SERVICE WORKER

CHAPTER-M/3-CLIENT.HTML

```
// (B) REGISTER SERVICE WORKER
```

```
async function regWorker () {
  // (B1) YOUR PUBLIC KEY - CHANGE TO YOUR OWN!
  const publicKey = "YOUR-PUBLIC-KEY";

  // (B2) REGISTER SERVICE WORKER
  const reg = await navigator.serviceWorker.
    register("4-sw.js", { scope: "/" });

  // (B3) SUBSCRIBE TO PUSH SERVER
  const sub = await reg.pushManager.subscribe({
    userVisibleOnly: true,
    applicationServerKey: publicKey
  });

  // (B4) TEST PUSH
  await fetch("/mypush", {
    method: "POST",
    body: JSON.stringify(sub),
    headers: { "content-type": "application/json" }
  });
}
```

- **(B1)** Insert your public key here.
- **(B2)** Looks familiar? Yes, it's the same old register service worker.
- **(B3)** After registering the service worker, we also subscribe to the push server – Pass in the options and public key. Take note that this returns a subscription object into `const sub`.
- **(B4)** After a complete client-side setup, we immediately fire a fetch request to `/mypush`. This is an endpoint that we will build later on the server, to send out a dummy push notification. Take note, we pass the subscription object `sub` to the server.

STEP 4) SERVICE WORKER

CHAPTER-M/4-SW.JS

```
self.addEventListener("push", (evt) => {
  const data = evt.data.json();
  self.registration.showNotification(data.title, {
    body: data.body,
    icon: data.icon,
    image: data.image
  });
});
```

Simply listen to the `push` event and use `registration.showNotification()` to show push messages.

STEP 5) SERVER-SIDE

TURN OFF YOUR WEB SERVER!

Hold your horses, things get complicated here. Now, most hosts in the real world should already be running an HTTP web server (Apache, Nginx, IIS, etc…). Here's the sticky situation:

- If we also deploy the Node push server on port 80, it will clash with the existing web server.
- We can avert this by deploying it on another port (8080 for example), but same origin restriction kicks in and this still fails.
- There are 2 possible solutions to further remedy this situation.
  - Run the push server on port 8080, then set an HTTP proxy in the web server. I.E. "Map" `https://site.com/mypush` to `https://site.com:8080/mypush`.

- Turn off the existing web server, set the Node server to cover both HTTP and push services.

Since setting up a proxy is grossly out of topic for this book, we will go with the simpler "hostile take over" method. Remember to turn off your web server… But in your live application, you will want to set up a proper proxy instead. Or use a server-side language supported on your server to rebuild the entire push server below.

NODE HTTP & PUSH SERVER

CHAPTER-M/5-SERVER.JS

```
// (A) MODULES & SETTINGS
const express = require("express"),
      bodyParser = require("body-parser"),
      webpush = require("web-push"),
      mail = "your@email.com",
      port = 80,
      publicKey = "YOUR-PUBLIC-KEY",
      privateKey = "YOUR-PRIVATE-KEY";

// (B) SETUP SERVER
webpush.setVapidDetails("mailto:" + mail, publicKey,
privateKey);
const app = express();
app.use(express.static(__dirname)); // SERVE STATIC FILES
app.use(bodyParser.json()); // JSON PARSER

// (C) SEND TEST PUSH NOTIFICATION
app.post("/mypush", (req, res) => {
  res.status(201).json({}); // REPLY WITH 201 (CREATED)
  webpush.sendNotification(req.body, JSON.stringify({
    title: "YES!",
```

```
    body: "It works!",
    icon: "note-potato.png",
    image: "note-banner.png"
  }))
  .catch((err) => { console.log(err); });
});

// (D) START!
app.listen(port, () => {
  console.log(`Server deployed at ${port}`)
});
```

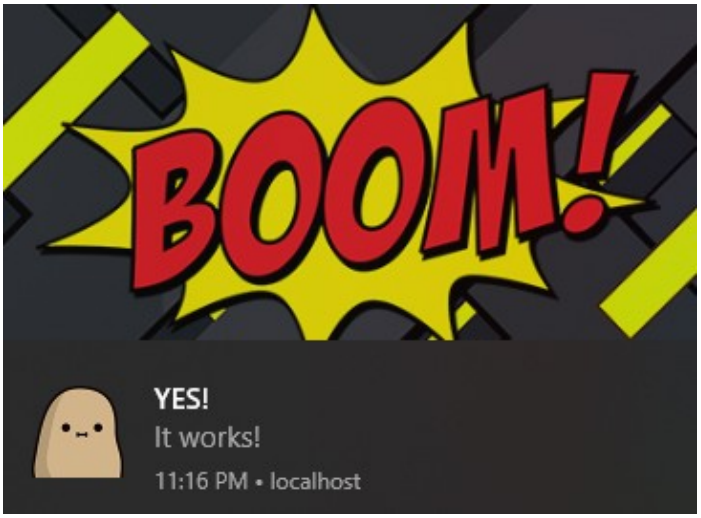
A) Insert your public and private keys.

B) Setup the push server, set Express to use the JSON parser and serve static files from your project folder.

C) The dummy `/mypush` endpoint mentioned earlier. Notice how we "reuse and forward" the subscription object – `webpush.sendNotification(req.body, NOTIFICATION)`.

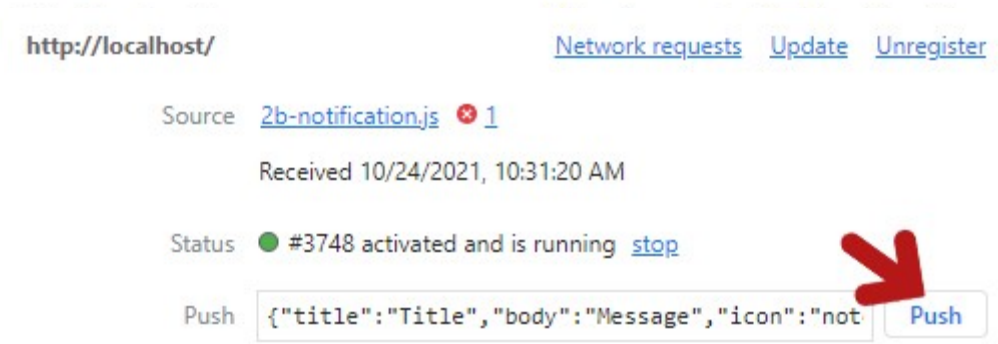
D) Start the server. What else?

STEP 6) LAUNCH!



- Start the server – `node 5-server.js`.
- Access `http://localhost/3-client.html` in the browser.
- Give permission and see the notification roll.

PUSH TEST



Now that the service worker is registered, you can actually do a test push at any time without the push server. Open up the developer's console, go under Application > Service Worker > Try sending this test push - `{"title":"Title","body":"Message","icon":"note-potato.png","image":"note-banner.png"}`

P.S. Try to use absolute URL for the icon and image, it is more reliable that way.

SECURITY & MORE

Of course, this example only scratched the surface. In your own project, you will need to tie all of these together:

- **Client-side:** Ask for user permission, register service worker,

subscribe to push server.

- **Service Worker:** Listen to push requests, show the notification.
- **Server-side:** NodeJS is not the only one that can work with push notifications. There are also web push servers for PHP, Python, Java, C#, etc… Feel free to adopt whichever you are familiar with, just do a quick search on GitHub. Also, the `mypush/` endpoint should be restricted to "admin only".

LINKS & REFERENCES

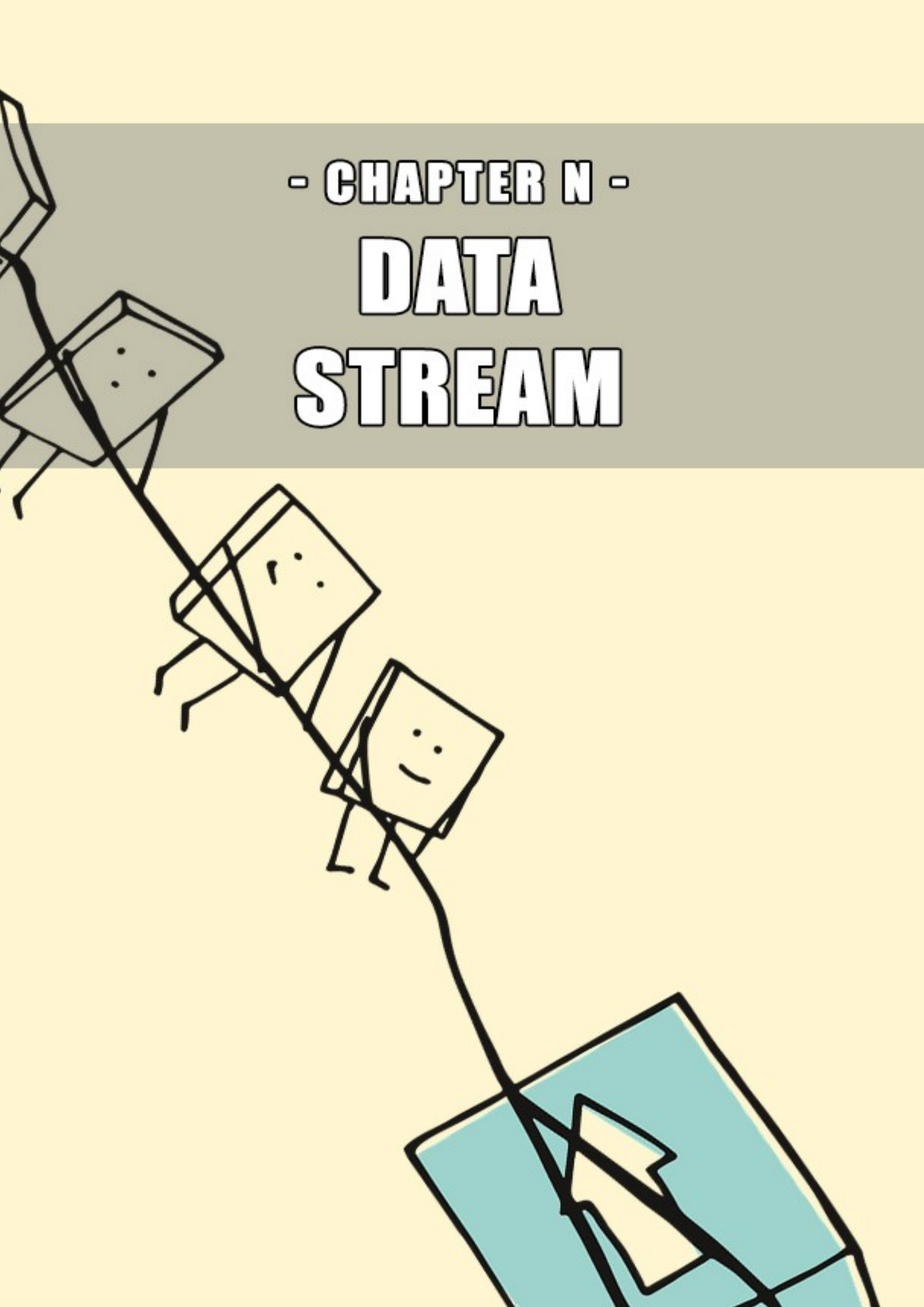
- Notification API – MDN
- Push API – MDN
- Push Manager API – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Web Notification
- Push Manager

- CHAPTER N -

DATA STREAM



DRY TOPIC

Streaming, it is nothing new these days. But just what does it mean? In layman terms – Instead of dealing with "one big file", we split it into smaller chunks. While this sounds easy, the concepts behind streaming can be dry and boring. So let's get over it, with simple examples.

NODEJS WRITABLE FILE STREAM

CHAPTER-N/1-WRITE-FILE-STREAM.JS

```
// (A) WRITE STREAM
let stream = require("fs").createWriteStream("dummy.txt");

// (B) COMMON WRITE STREAM EVENTS
// (B1) ERROR
stream.on("error", (err) => { console.log(err); });
// (B2) FINISHED WRITING
stream.on("finish", () => { console.log("Finished"); });
// (B3) STREAM CLOSED
stream.on("close", () => { console.log("Closed"); });

// (C) WRITE DUMMY DATA
for (let i=0; i<99; i++) {
  stream.write("Lorem ipsum dolor sit amet, consectetur
  adipiscing elit. Curabitur ut odio viverra turpis
  interdum pulvinar.");
}
stream.end();
```

Yes, you should have already seen this in the "write files" chapter. Using a writable file stream in Node is as straightforward as can be:

- Open file stream – `STREAM = fs.createWriteStream(FILE)`.

- Write into the file stream – `STREAM.write(DATA)`.
- Properly close the stream – `STREAM.end()`.

Take note of the events though, these may come in handy in your app:

- `error` – Handle when an error occurs.
- `finish` – On write finish, but take note that the stream remains open. In plain English, this simply means "finished writing everything to the file, buffer is emptied".
- `close` – Fired after `STREAM.end()`, when the stream is closed.

NODEJS READABLE FILE STREAM

CHAPTER-N/2-READ-FILE-STREAM.JS

```
// (A) READ STREAM
let stream = require("fs").createReadStream("dummy.txt");
// TO READ SPECIFIC BYTES ONLY
// ("dummy.txt", {start: 0, end: 99})

// (B) COMMON WRITE STREAM EVENTS
// (B1) ERROR
stream.on("error", (err) => { console.log(err); });
// (B2) REACHED END OF FILE
stream.on("end", () => { console.log("EOF"); });
// (B3) STREAM CLOSED
stream.on("close", () => { console.log("Closed"); });

// (C) READ FROM FILE STREAM
stream.on("data", (chunk) => {
  console.log("Read", chunk);
});
```

The readable stream counterpart should be self-explanatory too.

- Open file stream – `STREAM = fs.createReadStream(FILE)`.
- Read the file chunk-by-chunk – `STREAM.on("data", (chunk) => { … })`.

Similarly, the common events that may be useful:

- `error` – When an error occurs.
- `end` – When the file pointer reaches the end of file.
- `close` – When the stream is closed.

PIPE CHAIN

CHAPTER-N/3-PIPE.JS

```
// (A) LOAD MODULES
const fs = require("fs"),
      http = require("http");

// (B) PIPE DUMMY FILE TO HTTP RESPONSE
const server = http.createServer();
server.on("request", (req, res) => {
  fs.createReadStream("dummy.txt").pipe(res);
});

// (C) START!
server.listen(8080, () => {
  console.log("Server deployed at 8080")
});
```

With the basic read and write out of the way, let us now introduce a concept called "pipe chain" – It is quite literally what it means. We "plug" the stream into a pipe, let the data flow into another process. In this example:

- When a user connects to this dummy HTTP server, we will start reading `dummy.txt` into a file stream.
- Pipe the stream into the HTTP response. That is, we serve the contents of `dummy.txt` to the user chunk by chunk.
- Go ahead and see this for yourself, run `node 3-pipe.js` then access `http://localhost:8080` in your browser.

Piping is actually a clever technique that can transform data on-the-fly. For example, we can install a zip module and serve a ZIP file instead – `fs.createReadStream("dummy.txt").pipe(ZIP).pipe(res)`

"RAW" READ & WRITE STREAMS

CHAPTER-N/4-READ-WRITE.JS

```
// (A) READABLE & WRITABLE STREAMS
const Stream = require("stream"),
      rStream = new Stream.Readable(),
      wStream = new Stream.Writable();

// (B) SET STREAMS TO ECHO DATA
// (B1) CONSUMER
rStream.on("data", (chunk) => {
  console.log("Read Stream", chunk.toString());
});
// (B2) SINK
wStream._write = (chunk, encoding, next) => {
```

```
  console.log("Write Stream", chunk.toString());
  next();
};

// (C) DUMB TEST - PIPE READ STREAM INTO WRITE
rStream.pipe(wStream);
rStream.push("Hello World!");

// (D) CLOSE STREAM
rStream.destroy();
```

So far so good? Let us take things up one notch, literally. Streams don't always have to be associated with files. In this example:

A) We create 2 "raw" readable and writable data streams.

B) Set both streams to echo data. By the way, the process in which read streams deal with data is called a "consumer", while the write stream is called a "sink".

C) Pipe the readable stream into the writable stream.

Well, this is not productive, but should serve its purpose for demonstration. A better real-world example is:

- Feed web cam data into the read stream.
- Pipe the read stream into a video compressor or encoder.
- Lastly, pipe the compressed data into a file write stream.

DUPLEX STREAM

CHAPTER-N/5-DUPLEX.JS

```
// (A) DUPLEX STREAM
const Stream = require("stream"),
```

```
      dStream = new Stream.Duplex();

// (B) SET STREAMS TO ECHO DATA
dStream._read = () => {};
dStream._write = (chunk, encoding, next) => {
  console.log("Write Stream", chunk.toString());
  next();
};
dStream.on("data", (chunk) => {
  console.log("Read Stream", chunk.toString());
});

// (C) DUPLEX IS ESSENTIALLY 2-IN-1
dStream.push("Hello World!");
dStream.write("Goodbye World!");
```

Some of you guys may be thinking "why so stupid, why do we need 2 streams for read and write"? Well, we also have a duplex stream in Node. Essentially, the duplex is just a read and write stream 2-in-1 instant mix – Duplex inherits whatever the read and write stream has, but take note, they are still separate channels within the duplex.

VIDEO STREAM

SETTING UP THIS DEMO

All right, here's one last "actually pretty useful" demo for the server-side. But before that, do three things:

- `npm install express.`
- Do an online search for "free stock video", download any free video for testing.
- Move the video file into your project folder. Edit `6b-vid-`

`server.js` and change (C2) `const vid` to your own.

DUMMY VIDEO PAGE

CHAPTER-N/6A-VID-PAGE.HTML

```
<video src="http://localhost:8080/video"
       style="width:100%" controls autoplay></video>
```

There is only one video tag on this test page. Take note that the source points to `localhost:8080/video`.

NODE HTTP VIDEO STREAM

CHAPTER-N/6B-VID-SERVER.JS

```
// (A) MODULES & EXPRESS SERVER
const express = require("express"),
      fs = require("fs"),
      app = express();

// (B) SERVE DEFAULT HTML PAGE
app.get("/", (req, res) => {
  res.sendFile(__dirname + "/6a-vid-page.html");
});

// (C) SERVE VIDEO STREAM
app.get("/video", (req, res) => {
  // (C1) VIDEO RANGE MUST BE SPECIFIED
  const range = req.headers.range;
  if (!range) {
    res.status(400).send("Requires range header");
  }
```

```
  // (C2) VIDEO STATS
  const vid = __dirname + "/video.mp4",
        vidSize = fs.statSync(vid).size,
        chunkSize = 10 ** 6, // 1MB
        start = Number(range.replace(/\D/g, "")),
        end = Math.min(start + chunkSize, vidSize - 1),
        contentLength = end - start + 1;

  // (C3) SERVE HTTP 206 (PARTIAL CONTENT)
  res.writeHead(206, {
    "Content-Range": `bytes ${start}-${end}/${vidSize}`,
    "Accept-Ranges": "bytes",
    "Content-Length": contentLength,
    "Content-Type": "video/mp4",
  });

  // (C4) SERVE VIDEO STREAM
  fs.createReadStream(vid, { start, end }).pipe(res);
});

// (D) LAUNCH!
app.listen(8080, () => {
  console.log("Server deployed at 8080!");
});
```

Credits go to Abdisalan, this post on dev.to. Go ahead, run this `node 6b-vid-server.js` and access `http://localhost:8080`. What the heck does this script do?

- **(B)** Serve `6a-vid-page.html` when you access `localhost:8080`.
- **(C)** Serve the video when you access `localhost:8080/video`. This is the juicy part. Take note of how the file stream reads a chunk of the video file and outputs it to HTTP response.

Sure thing, most web servers already do this video streaming "automatically". But this example is a quick appreciation to the technologies behind – Even the `<video>` tag automatically does streaming and buffering behind the scenes, without requiring you to write a single line of code.

CLIENT-SIDE READABLE STREAM

CHAPTER-N/7-READ-STREAM.HTML

```
// (A) FETCH DUMMY TEXT FILE & RETURN STREAM
fetch("dummy.txt")
.then((res) => { return res.body.getReader(); })

// (B) READ THE STREAM
.then((reader) => {
  reader.read().then(function process({ done, value }) {
    // (B1) NO MORE DATA TO READ
    if (done) {
      console.log("End of stream.");
      return;
    }

    // (B2) READ CHUNK
    console.log(value); // UINT8ARRAY

    // (B3) IF YOU WANT TO CONVERT TO STRING
    var string = new TextDecoder().decode(value);
    console.log(string);

    // (B4) CONTINUE READ NEXT CHUNK
    return reader.read().then(process);
  });
});
```

Streaming is obviously not a "server-side only" thing – Readable streams also exist on client-side Javascript.

CLIENT-SIDE WRITABLE STREAM

CHAPTER-N/8-WRITE-STREAM.HTML

```
<!-- (A) TO PUT CONTENTS INTO -->
<div id="demo"></div>

<script>
// (B) GET HTML <DIV> + TEXT DECODER
const demo = document.getElementById("demo"),
      decoder = new TextDecoder();

// (C) CREATE WRITABLE STREAM
const writer = new WritableStream({
  // (C1) SINK
  write (chunk) {
    console.log(chunk);
    demo.innerHTML += decoder.decode(chunk);
  },

  // (C2) ON STREAM CLOSE
  close () { console.log("Done"); }
});

// (D) FETCH & PIPE TO WRITABLE STREAM
fetch("dummy.txt")
.then((res) => { res.body.pipeTo(writer); });
</script>
```

Warning: Writable streams are not supported in Firefox at the time of writing.

Yep, this is the writable stream counterpart. This is yet another "super

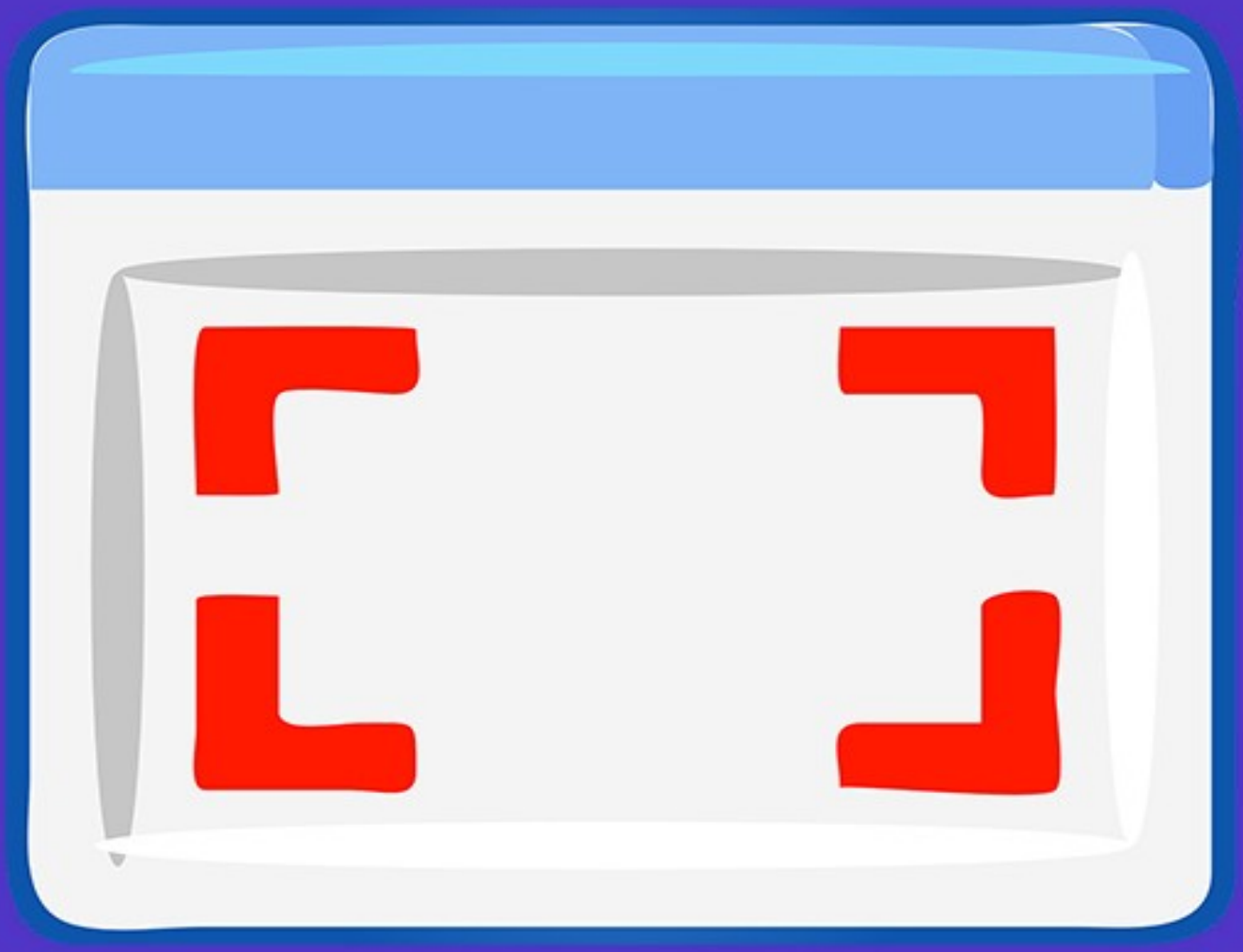
inefficient and unproductive" example of piping the fetch read stream into a write stream... That pretty much outputs the results into an HTML `<div>`. In your own project, you will want to change the sink (C1) to do something productive instead.

LINKS & REFERENCES

- Stream – NodeJS
- Streams API – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- ReadableStream API
- WritableStream API



- CHAPTER 0 -

GOING FULLSCREEN

SUPER EASY FULLSCREEN MODE

CHAPTER-O/1-FULLSCREEN.HTML

```
<!-- (A) IMAGE FOR TESTING FULLSCREEN -->
<img id="demo" src="doge.png"/>

<!-- (B) FULLSCREEN BUTTONS -->
<div>
  <!-- (B1) ENTIRE PAGE -->
  <input type="button" value="Entire Page"
         onclick="document.documentElement
         .requestFullscreen()"/>

  <!-- (B2) IMAGE ONLY -->
  <input type="button" value="Image Only"
         onclick="document.getElementById('demo')
         .requestFullscreen()"/>

  <!-- (B3) EXIT FULLSCREEN -->
  <input type="button" value="Exit"
         onclick="document.exitFullscreen();"/>
</div>
```

- To engage fullscreen – `ELEMENT.requestFullscreen()`
- To disengage fullscreen – `document.exitFullscreen()`

Yes, this pretty much sums up the entire chapter.

- `document.documentElement` refers to the whole HTML page, so `document.documentElement.requestFullscreen()` engages fullscreen on the entire page.
- Self-explanatory, to engage fullscreen on a section only – `document.getElementById(ID).requestFullscreen()`

DETECTING FULLSCREEN TOGGLE

CHAPTER-O/1-FULLSCREEN.HTML

```
// (C) LISTEN TO FULLSCREEN TOGGLE
document.addEventListener("fullscreenchange", () => {
  if (document.fullscreenElement===null) {
    console.log("Exited fullscreen");
  } else {
    console.log("Entered fullscreen");
  }
});

// (D) ON FULLSCREEN ERROR
document.addEventListener("fullscreenerror", (evt) => {
  console.error(evt);
});
```

- The `fullscreenchange` event is triggered when the user toggles fullscreen mode.
- When fullscreen mode encounters an error, `fullscreenerror` is triggered
- `document.fullscreenElement` holds the current fullscreen element. This is `null` when fullscreen mode is not engaged.

LINKS & REFERENCES

- Fullscreen API – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Fullscreen API Compatibility – CanIUse

- CHAPTER P -

AUDIO PLAYER

MAKE SOME SOUND

Once upon a time in the Iron Age of the Internet, we have to use all sorts third-party plugins and implement our own server-side solutions to stream audio files. Now? It's as simple as `<audio>`, or:

- `var audio = new Audio("SONG.MP3");`
- `audio.play();`
- `audio.pause();`
- `audio.volume = 0.0 TO 1.0;`
- `audio.currentTime = SECONDS;`

Yep, no idea why some people still complain "it's so difficult". That absolutely makes no sense. Javascript audio is long-winded, but it is by all means, far from being "complicated". Let's walk through a custom audio player (with playlist) in this chapter.

P.S. There are no demo songs in the zip file (possible copyright issues) – Search for "free stock music" and download your own.

STEP 1) THE HTML

CHAPTER-P/1-PLAYER.HTML

```
<!-- MATERIAL ICON -->
<link rel="stylesheet" href="https://fonts.googleapis.com/
icon?family=Material+Icons">

<div id="aWrap">
  <!-- (A) PLAY/PAUSE BUTTON -->
  <button id="aPlay" disabled>
```

```
  <span id="aPlayIco" class="material-icons">
    play_arrow
  </span></button>

  <!-- (B) TIME -->
  <div id="aCron">
    <span id="aNow"></span> / <span id="aTime"></span>
  </div>

  <!-- (C) SEEK BAR -->
  <input id="aSeek" type="range" min="0" value="0"
         step="1" disabled/>

  <!-- (D) VOLUME SLIDE -->
  <span id="aVolIco" class="material-icons">
    volume_up
  </span>
  <input id="aVolume" type="range" min="0" max="1"
         value="1" step="0.1" disabled/>

  <!-- (E) PLAYBACK SPEED -->
  <select id="aRate">
    <option value="0.5">0.5</option>
    <option value="1.0" selected>1.0</option>
    <option value="1.5">1.5</option>
    <option value="2.0">2.0</option>
  </select>

  <!-- (F) PLAYLIST -->
  <div id="aList"></div>
</div>
```

There are 6 components in this custom audio player:

A) `aPlay` Play/pause button. `aPlayIco` Holds the play/pause icon, I have used Google's Material Icon here.

B) `aNow` The current playtime. `aTime` Length of the current track.

C) `aSeek` Time seek bar, the value is in seconds.

D) `aVolume` Volume slider bar, this is a number from 0.0 (mute) to 1.0 (loudest). `aVolIco` Volume icon.

E) `aRate` Playback speed selector. There doesn't seem to have a hard limit on how fast this can go… But we will stop at 2.0, any faster, and you will be listening to gibberish.

F) `aList` Audio playlist. Will be generated with Javascript.

STEP 2) INITIALIZING THE PLAYER

CHAPTER-P/1-PLAYER.JS

```
window.addEventListener("DOMContentLoaded", () => {
  // (A) PLAYER INIT
  // (A1) PLAYLIST - CHANGE TO YOUR OWN!
  let playlist = [
    {name: "Sugar Plum Fairy", src: "S1.mp3"},
    {name: "If I Had A Chicken", src: "S2.mp3"},
    {name: "Run Little Chicken", src: "S3.mp3"}
  ];

  // (A2) AUDIO PLAYER & GET HTML CONTROLS
  const audio = new Audio(),
        aPlay = document.getElementById("aPlay"),
        aPlayIco = document.getElementById("aPlayIco"),
        aNow = document.getElementById("aNow"),
        aTime = document.getElementById("aTime"),
        aSeek = document.getElementById("aSeek"),
        aVolume = document.getElementById("aVolume"),
        aVolIco = document.getElementById("aVolIco"),
```

```
        aRate = document.getElementById("aRate"),
        aList = document.getElementById("aList");
  // (A3) BUILD PLAYLIST
  for (let i in playlist) {
    let row = document.createElement("div");
    row.className = "aRow";
    row.innerHTML = playlist[i]["name"];
    row.addEventListener("click", () => { audPlay(i); });
    playlist[i]["row"] = row;
    aList.appendChild(row);
  }
});
```

- **(A1)** Start by defining the `playlist`, this is an array of objects. Each entry should have the `name` and `src`.
- **(A2)** Self-explanatory, create the `new Audio()` object, and get all the HTML controls.
- **(A3)** Loop through `playlist` to build the HTML. Take note that clicking on a row will trigger `audPlay()`.

STEP 3) PLAY MECHANISM

CHAPTER-P/1-PLAYER.JS

```
// (B) PLAY MECHANISM
// (B1) FLAGS
var audNow = 0, // current song
    audStart = false, // auto start next song

// (B2) PLAY SELECTED SONG
audPlay = (idx, nostart) => {
  audNow = idx;
  audStart = nostart ? false : true;
```

```
  audio.src = playlist[idx]["src"];
  audio.playbackRate = aRate.value;
  for (let i in playlist) {
    if (i == idx) {
      playlist[i]["row"].classList.add("now");
    } else {
      playlist[i]["row"].classList.remove("now");
    }
  }
};

// (B3) AUTO START WHEN SUFFICIENTLY BUFFERED
audio.addEventListener("canplay", () => { if (audStart) {
  audio.play();
  audStart = false;
}});

// (B4) AUTOPLAY NEXT SONG IN THE PLAYLIST
audio.addEventListener("ended", () => {
  audNow++;
  if (audNow >= playlist.length) { audNow = 0; }
  audPlay(audNow);
});

// (B5) INIT SET FIRST SONG
audPlay(0, true);
```

The rather painful part…

- **(B1)** We have to use 2 flags to control audio player.
  - Since `playlist` is an array, `audNow` is used the track the current song. E.G. `audNow = 2` will indicate that the third song is currently playing.
  - `audStart` is used to control the autostart in (B4).

- **(B2)** Remember that clicking on an item in the HTML list will trigger `audPlay()`? This plays the selected song by simply setting `audio.src = playlist[N]["src"]`. Take note of how this sets the `audStart` flag as well.
- **(B3)** Autoplay mechanism and control.
  - When the audio file is sufficiently buffered and `canplay`, we simply start playing. But take note of the use of the `audStart` flag here.
  - Basically, `canplay` can be triggered by 2 possible means – When we set `audio.src` and after the user recovers from a bad connection.
  - We do not want to force a choppy playback, so `audStart` is used to trigger autoplay for manual "click on playlist" and "autoplay next song" only.
- **(B4)** When the current song has ended, we do a `audNow++` and play the next song automatically.
- **(B5)** Init set the first song, but don't autoplay.

STEP 4) PLAY/PAUSE BUTTON

CHAPTER-P/1-PLAYER.JS

```
// (C) PLAY/PAUSE BUTTON
// (C1) AUTO SET PLAY/PAUSE TEXT
audio.addEventListener("play", () => {
  aPlayIco.innerHTML = "pause";
});
```

```
audio.addEventListener("pause", () => {
  aPlayIco.innerHTML = "play_arrow";
});
// (C2) CLICK TO PLAY/PAUSE
aPlay.addEventListener("click", () => {
  if (audio.paused) { audio.play(); }
  else { audio.pause(); }
});
```

- **(C1)** Auto set the play/pause icon.
- **(C2)** If the audio is paused, click to play. If the audio is playing, click to pause.

STEP 5) TRACK PROGESS (TIME)

CHAPTER-P/1-PLAYER.JS

```
// (D) TRACK PROGRESS
// (D1) SUPPORT FUNCTION - FORMAT HH:MM:SS
var timeString = (secs) => {
  // HOURS, MINUTES, SECONDS
  let ss = Math.floor(secs),
      hh = Math.floor(ss / 3600),
      mm = Math.floor((ss - (hh * 3600)) / 60);
  ss = ss - (hh * 3600) - (mm * 60);

  // RETURN FORMATTED TIME
  if (hh>0) { mm = mm<10 ? "0"+mm : mm; }
  ss = ss<10 ? "0"+ss : ss;
  return hh>0 ? `${hh}:${mm}:${ss}` : `${mm}:${ss}` ;
};

// (D2) INIT SET TRACK TIME
audio.addEventListener("loadedmetadata", () => {
  aNow.innerHTML = timeString(0);
```

```
  aTime.innerHTML = timeString(audio.duration);
});

// (D3) UPDATE TIME ON PLAYING
audio.addEventListener("timeupdate", () => {
  aNow.innerHTML = timeString(audio.currentTime);
});
```

- **(D1)** The track length `audio.duration` and `audio.currentTime` are both in seconds (with 2 decimal places microseconds). We need this support function to manually format a "nice display time".
- **(D2)** When the metadata is loaded, we reset the HTML current time to 0:00, and set the total time.
- **(D3)** As the track plays, we update the HTML current time.

STEP 6) TIME SEEK BAR

CHAPTER-P/1-PLAYER.JS

```
// (E) SEEK BAR
audio.addEventListener("loadedmetadata", () => {
  // (E1) SET SEEK BAR MAX TIME
  aSeek.max = Math.floor(audio.duration);

  // (E2) USER CHANGE SEEK BAR TIME
  var aSeeking = false; // USER IS NOW CHANGING TIME
  aSeek.addEventListener("input", () => {
    aSeeking = true; // PREVENTS CLASH WITH (E3)
  });
  aSeek.addEventListener("change", () => {
    audio.currentTime = aSeek.value;
    if (!audio.paused) { audio.play(); }
```

```
    aSeeking = false;
  });

  // (E3) UPDATE SEEK BAR ON PLAYING
  audio.addEventListener("timeupdate", () => {
    if (!aSeeking) {
      aSeek.value = Math.floor(audio.currentTime);
    }
  });
});
```

- **(E1)** When the metadata is loaded, we set the `max` of the time seek bar.
- **(E3)** Update the seek bar as the track plays.
- **(E2 & E3)** Time seek action.
  - Very simply, we set `audio.currentTime = SELECTED TIME` when the user changes the seek bar.
  - But take note of that when the user is changing the seek bar (`input`) we engage `aSeeking = true`. This is to "disable" (E3), so that the seek bar don't "skip" as the user is trying to drag the seek bar.

STEP 7) SET VOLUME

CHAPTER-P/1-PLAYER.JS

```
// (F) VOLUME
aVolume.addEventListener("change", () => {
  audio.volume = aVolume.value;
  aVolIco.innerHTML = (aVolume.value==0 ? "volume_mute" :
  "volume_up");
```

```
});
```

Probably the easiest part, no explanation required.

STEP 8) PLAYBACK SPEED

CHAPTER-P/1-PLAYER.JS

```
// (G) PLAYBACK SPEED
aRate.addEventListener("change", () => {
  audio.playbackRate = aRate.value;
});
```

Another self-explanatory part.

STEP 9) ENABLE/DISABLE

CHAPTER-P/1-PLAYER.JS

```
// (H) ENABLE/DISABLE CONTROLS
audio.addEventListener("canplay", () => {
  aPlay.disabled = false;
  aVolume.disabled = false;
  aSeek.disabled = false;
  aRate.disabled = false;
});
audio.addEventListener("waiting", () => {
  aPlay.disabled = true;
  aVolume.disabled = true;
  aSeek.disabled = true;
  aRate.disabled = true;
});
```

Enable the controls when playback is possible, disable when the audio is loading.

EXTRA) EFFECTS WITH WEB AUDIO API

THE HTML

CHAPTER-P/2-WEB-AUDIO-API.HTML

```
<input type="button" value="Start!" onclick="start()"/>
```

Adding audio effects with Javascript? Yes we can. But take note that the web audio API cannot start unless the user clicks or interacts with something first, thus the need for this HTML button.

THE JAVASCRIPT

CHAPTER-P/2-WEB-AUDIO-API.JS

```
function start () {
  // (A) AUDIO CONTEXT
  const audio = new Audio("S2.mp3"),
        audioCTX = new AudioContext(),
        audioSRC = audioCTX
          .createMediaElementSource(audio);

  // (B) GAIN
  const audioGAIN = audioCTX.createGain();
  audioGAIN.gain.value = 1.5; // 1.5 TIMES LOUDER

  // (C) PANNING
  // -1 LEFT, 0 CENTER, 1 RIGHT
  const audioPAN = audioCTX.createStereoPanner();
  audioPAN.pan.value = 0.7;

  // (D) COMPRESSOR
  const audioCOM = audioCTX.createDynamicsCompressor();
  // NOT AN AUDIO ENGINEER - DON'T KNOW WHAT I AM DOING
  audioCOM.threshold.setValueAtTime(-50,
```

```
    audioCTX.currentTime);
  audioCOM.attack.setValueAtTime(5, audioCTX.currentTime);
  audioCOM.knee.setValueAtTime(40, audioCTX.currentTime);
  audioCOM.ratio.setValueAtTime(12, audioCTX.currentTime);
  audioCOM.release.setValueAtTime(0.25,
    audioCTX.currentTime);

  // (E) CONNECT THE DOTS
  audioSRC
    .connect(audioGAIN)
    .connect(audioPAN)
    .connect(audioCOM)
    .connect(audioCTX.destination);
  audio.play();
}
```

To use the Web Audio API:

1. Create an `<audio>` or `new Audio()` as usual.
2. Create an `audioCTX = new AudioContext()` object.
3. "Connect" the audio context to the source – `audioSRC = audioCTX.createMediaElementSource(HTML AUDIO OR AUDIO OBJECT)`.
4. Set all the effects you like – Pan, gain, compressor, modulator, delay, etc…
5. Lastly chain all of them together – `audioSRC.connect(EFFECT).connect(EFFECT)`.

Sorry guys, I am not a sound engineer. So I literally don't know much about what is going on here… But if you want to "tweak the sound", there's all sorts filters available. Check out the Web Audio API link below.

LINKS & REFERENCES

- Media Element – MDN
- Web Audio API – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Audio
- Web Audio API

- CHAPTER Q -

VIDEO PLAYER

LOOKS FAMILIAR...

Oh no! It's a video! Things are going to be difficult... Not! This chapter is going to look very familiar. Yes, this chapter is literally a copy-and-paste of the previous chapter, but for video.

P.S. No videos are included again, please download your own.

STEP 1) THE HTML

CHAPTER-Q/VIDEO.HTML

```
<!-- MATERIAL ICON -->
<link rel="stylesheet" href="https://fonts.googleapis.com/
icon?family=Material+Icons">

<div id="vWrap">
  <!-- (A) VIDEO TAG -->
  <video id="vVid"></video>

  <!-- (B) PLAY/PAUSE BUTTON -->
  <button id="vPlay" disabled>
  <span id="vPlayIco" class="material-icons">
    play_arrow
  </span></button>

  <!-- (C) TIME -->
  <div id="vCron">
    <span id="vNow"></span> / <span id="vTime"></span>
  </div>

  <!-- (D) SEEK BAR -->
  <input id="vSeek" type="range" min="0" value="0"
         step="1" disabled/>
  <!-- (E) VOLUME SLIDE -->
```

```
  <span id="vVolIco" class="material-icons">
    volume_up
  </span>
  <input id="vVolume" type="range" min="0" max="1"
         value="1" step="0.1" disabled/>

  <!-- (F) PLAYBACK SPEED -->
  <select id="vRate">
    <option value="0.5">0.5</option>
    <option value="1.0" selected>1.0</option>
    <option value="1.5">1.5</option>
    <option value="2.0">2.0</option>
  </select>

  <!-- (G) PLAYLIST -->
  <div id="vList"></div>
</div>
```

This is just about the same as the audio player.

A) `vPlay` The HTML video tag itself.

B) `vPlay` Play/pause button, `vPlayIco` is the play/pause icon. Same old Google's Material Icon.

C) `vNow` current playtime, `vTime` length of the current video.

D) `vSeek` Time seek bar, in seconds.

E) `vVolume` Volume slider bar, from 0.0 (mute) to 1.0 (loudest). `vVolIco` Volume icon.

F) `vRate` Playback speed selector.

G) `vList` Video playlist.

STEP 2) INITIALIZING THE PLAYER

CHAPTER-Q/VIDEO.JS

```
window.addEventListener("DOMContentLoaded", () => {
  // (A) PLAYER INIT
  // (A1) PLAYLIST - CHANGE TO YOUR OWN!
  let playlist = [
    {name: "Video A", src: "v1.mp4"},
    {name: "Video B", src: "v2.mp4"},
    {name: "Video C", src: "v3.mp4"}
  ];

  // (A2) VIDEO PLAYER & GET HTML CONTROLS
  const video = document.getElementById("vVid"),
        vPlay = document.getElementById("vPlay"),
        vPlayIco = document.getElementById("vPlayIco"),
        vNow = document.getElementById("vNow"),
        vTime = document.getElementById("vTime"),
        vSeek = document.getElementById("vSeek"),
        vVolume = document.getElementById("vVolume"),
        vVolIco = document.getElementById("vVolIco"),
        vRate = document.getElementById("vRate"),
        vList = document.getElementById("vList");

  // (A3) BUILD PLAYLIST
  for (let i in playlist) {
    let row = document.createElement("div");
    row.className = "vRow";
    row.innerHTML = playlist[i]["name"];
    row.addEventListener("click", () => { vidPlay(i); });
    playlist[i]["row"] = row;
    vList.appendChild(row);
  }
});
```

- **(A1)** First, we start with defining the video `playlist`. Each entry should have the `name` and `src`.
- **(A2)** Get all the HTML elements.
- **(A3)** Loop through `playlist` to build the HTML. Clicking on a playlist item will trigger `vidPlay()`.

STEP 3) PLAY MECHANISM

CHAPTER-Q/VIDEO.JS

```
// (B) PLAY MECHANISM
// (B1) FLAGS
var vidNow = 0, // current video
    vidStart = false, // auto start next video

// (B2) PLAY SELECTED VIDEO
vidPlay = (idx, nostart) => {
  vidNow = idx;
  vidStart = nostart ? false : true;
  video.src = playlist[idx]["src"];
  video.playbackRate = vRate.value;
  for (let i in playlist) {
    if (i == idx)
    { playlist[i]["row"].classList.add("now"); }
    else { playlist[i]["row"].classList.remove("now"); }
  }
};

// (B3) AUTO START WHEN SUFFICIENTLY BUFFERED
video.addEventListener("canplay", () => { if (vidStart) {
  video.play();
  vidStart = false;
}});
```

```
// (B4) AUTOPLAY NEXT VIDEO IN THE PLAYLIST
video.addEventListener("ended", () => {
  vidNow++;
  if (vidNow >= playlist.length) { vidNow = 0; }
  vidPlay(vidNow);
});

// (B5) INIT SET FIRST VIDEO
vidPlay(0, true);
```

- **(B1)** Same old flags to control the video player.
  - `vidNow` Index of the currently playing video of `playlist`.
  - `vidStart` used to indicate if a video should auto play in (B4).
- **(B2)** `vidPlay()` plays the selected video by setting `video.src = playlist[N]["src"]`.
- **(B3)** Autoplay mechanism and control.
  - Start playing when the video `canplay`.
  - Same problem, `canplay` can either be triggered by setting `video.src`, or after the user recovers from a bad connection.
  - We don't want to force a choppy playback, so `vidStart` only triggers auto play for manual "click on playlist" and "play next video" only.
- **(B4)** `vidNow++` automatically plays the next video when the current ends.
- **(B5)** Init set the first video, but don't start playing (user needs to click on something to start playing anyway).

STEP 4) PLAY/PAUSE BUTTON

CHAPTER-Q/VIDEO.JS

```
// (C) PLAY/PAUSE BUTTON
// (C1) AUTO SET PLAY/PAUSE TEXT
video.addEventListener("play", () => {
  vPlayIco.innerHTML = "pause";
});
video.addEventListener("pause", () => {
  vPlayIco.innerHTML = "play_arrow";
});

// (C2) CLICK TO PLAY/PAUSE
vPlay.addEventListener("click", () => {
  if (video.paused) { video.play(); }
  else { video.pause(); }
});
```

- **(C1)** Automatic set the play/pause icon.
- **(C2)** Click to play when the video is paused. Click to pause if the video is playing.

STEP 5) TRACK PROGESS (TIME)

CHAPTER-Q/VIDEO.JS

```
// (D) TRACK PROGRESS
// (D1) SUPPORT FUNCTION - FORMAT HH:MM:SS
var timeString = (secs) => {
  // HOURS, MINUTES, SECONDS
  let ss = Math.floor(secs),
      hh = Math.floor(ss / 3600),
      mm = Math.floor((ss - (hh * 3600)) / 60);
  ss = ss - (hh * 3600) - (mm * 60);
```

```
  // RETURN FORMATTED TIME
  if (hh>0) { mm = mm<10 ? "0"+mm : mm; }
  ss = ss<10 ? "0"+ss : ss;
  return hh>0 ? `${hh}:${mm}:${ss}` : `${mm}:${ss}` ;
};

// (D2) INIT SET TRACK TIME
video.addEventListener("loadedmetadata", () => {
  vNow.innerHTML = timeString(0);
  vTime.innerHTML = timeString(video.duration);
});

// (D3) UPDATE TIME ON PLAYING
video.addEventListener("timeupdate", () => {
  vNow.innerHTML = timeString(video.currentTime);
});
```

- **(D1)** Same story, `video.duration` and `video.currentTime` are in seconds. We need a support "nice display time" function.
- **(D2)** When the metadata is loaded, reset the HTML current time to 0:00, and set the total play time.
- **(D3)** As the video plays, update the HTML current time.

STEP 6) TIME SEEK BAR

CHAPTER-Q/VIDEO.JS

```
// (E) SEEK BAR
video.addEventListener("loadedmetadata", () => {
  // (E1) SET SEEK BAR MAX TIME
  vSeek.max = Math.floor(video.duration);

  // (E2) USER CHANGE SEEK BAR TIME
  var vSeeking = false; // USER IS NOW CHANGING TIME
```

```
  vSeek.addEventListener("input", () => {
    vSeeking = true; // PREVENTS CLASH WITH (E3)
  });
  vSeek.addEventListener("change", () => {
    video.currentTime = vSeek.value;
    if (!video.paused) { video.play(); }
    vSeeking = false;
  });

  // (E3) UPDATE SEEK BAR ON PLAYING
  video.addEventListener("timeupdate", () => {
    if (!vSeeking) { vSeek.value =
    Math.floor(video.currentTime); }
  });
});
```

- **(E1)** Set the `max` of the time seek bar.
- **(E3)** Update the seek bar as the video plays.
- **(E2 & E3)** Time seek action. Same old "use a flag to prevent a jumping time bar when the user drags the bar".

STEP 7) SET VOLUME

CHAPTER-Q/VIDEO.JS

```
// (F) VOLUME
vVolume.addEventListener("change", () => {
  video.volume = vVolume.value;
  vVolIco.innerHTML = (vVolume.value==0 ? "volume_mute" :
  "volume_up");
});
```

Same as audio...

STEP 8) PLAYBACK SPEED

CHAPTER-Q/VIDEO.JS

```
// (G) PLAYBACK SPEED
vRate.addEventListener("change", () => {
  video.playbackRate = vRate.value;
});
```

Same as audio again...

STEP 9) ENABLE/DISABLE

CHAPTER-Q/VIDEO.JS

```
// (H) ENABLE/DISABLE CONTROLS
video.addEventListener("canplay", () => {
  vPlay.disabled = false;
  vVolume.disabled = false;
  vSeek.disabled = false;
  vRate.disabled = false;
});
video.addEventListener("waiting", () => {
  vPlay.disabled = true;
  vVolume.disabled = true;
  vSeek.disabled = true;
  vRate.disabled = true;
});
```

Enable the controls when playback is possible, disable when loading.

LINKS & REFERENCES

- Media Element – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Video

- CHAPTER R -

HTML-JAVASCRIPT CANVAS

JAVASCRIPT PAINT

The HTML canvas element has actually been around for a long time now. All those "crop image", "add watermark", "combine images", and whatever "online draw apps" – They are probably driven with canvas. It's powerful, but canvas can go in very deep and broad. Everything from 2D to 3D drawing, and even image manipulation.

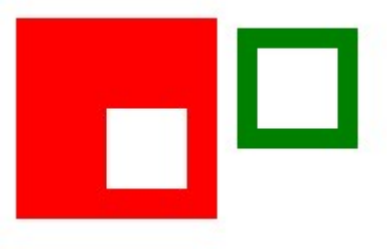
Since the title of this book is not "how to create a browser game", we will keep things simple and not bore you to tears. We will only touch on the quick basics and walk through some useful image editing magic in this chapter.

CANVAS BASICS – FILL, STROKE, CLEAR

CHAPTER-R/1-FILL-STROKE-CLEAR.HTML

```
<!-- (A) HTML CANVAS -->
<canvas id="mycanvas" width="200" height="200"></canvas>
<script>
// (B) GET CANVAS CONTEXT
var ctx =
document.getElementById("mycanvas").getContext("2d");

// (C) FILL RECTANGLE
ctx.fillStyle = "red";
ctx.fillRect(5, 5, 100, 100);

// (D) STROKE RECTANGLE
ctx.strokeStyle = "green";
ctx.lineWidth = 10;
ctx.strokeRect(120, 15, 50, 50);
```

```
// (E) CLEAR RECTANGLE
ctx.clearRect(50, 50, 40, 40);
</script>
```



- **(A)** To work with the canvas, we need to first create a `<canvas>`. Setting the `width` and `height` are optional, but highly recommended (give fixed dimensions to not break the layout of the page). You can also set these in CSS.
- **(B)** Get the 2D drawing context. Yes, canvas is also capable of drawing in 3D space with `webgl`... But we won't touch on that.
- **(C To E)** In 2D drawings, there are 3 "common drawing modes" that you need to know:
  - **Fill –** Draws a solid shape. In this example, `fillRect(X, Y, WIDTH, HEIGHT)` draws a solid rectangle/square. We can also use `fillStyle` to control the fill color, this takes in a CSS color value (`rgb, rgba, #RRGGBB`).
  - **Stroke –** Draws the outline, `strokeRect(X, Y, WIDTH, HEIGHT)` draws the outline of a rectangle/square. Use `lineWidth` to specify the thickness, and `strokeStyle` to specify the color (same CSS color values).
  - **Clear** – Self-explanatory, `clearRect(X, Y, WIDTH, HEIGHT)` will clear the specified area on the canvas.

CANVAS BASICS – WRITE TEXT

CHAPTER-R/2-TEXT.HTML

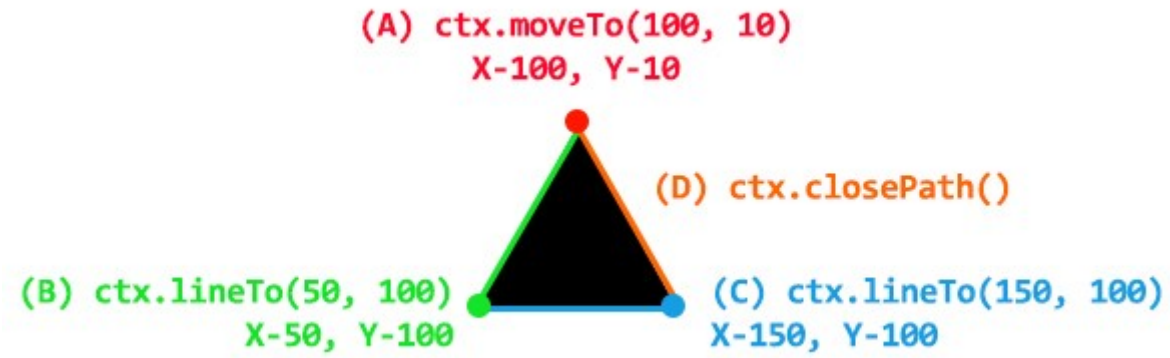
```
<!-- (A) HTML CANVAS -->
<canvas id="mycanvas" width="200" height="200"></canvas>
<script>
// (B) GET CANVAS CONTEXT
var ctx =
document.getElementById("mycanvas").getContext("2d");

// (C) FILL TEXT
ctx.font = "bold 24px Arial";
ctx.fillStyle = "red";
ctx.fillText("FOO", 10, 30);

// (D) STROKE TEXT
ctx.font = "italic 30px Arial";
ctx.strokeStyle = "green";
ctx.strokeText("BAR", 80, 30);
</script>
```

- **(A & B)** Same old canvas and drawing context.
- **(C)** Remember fill? We use `fillText(TEXT, X, Y)` to draw "solid text", use `font` to set the font itself – This takes in a regular CSS font property.
- **(D)** Remember stroke? Yep, use `strokeText(TEXT, X, Y)` to draw "outline text".

TRIANGLES & OTHER SHAPES – USE PATH

CHAPTER-R/3-PATH-TRIANGLE.HTML

```
<!-- (A) HTML CANVAS -->
<canvas id="mycanvas" width="200" height="200"></canvas>
<script>
// (B) GET CANVAS CONTEXT
var ctx =
document.getElementById("mycanvas").getContext("2d");

// (C) TRIANGLE PATH
ctx.beginPath();
ctx.moveTo(100, 10);
ctx.lineTo(50, 100);
ctx.lineTo(150, 100);
ctx.closePath();
ctx.fill();
</script>
```



Yes, it's crazy. For any other shapes, you have to manually draw it by defining a path.

- `beginPath()` Start drawing a path.
- `moveTo(X, Y)` Move to coordinates, but don't draw.
- `lineTo(X, Y)` Move to coordinates and draw a line.

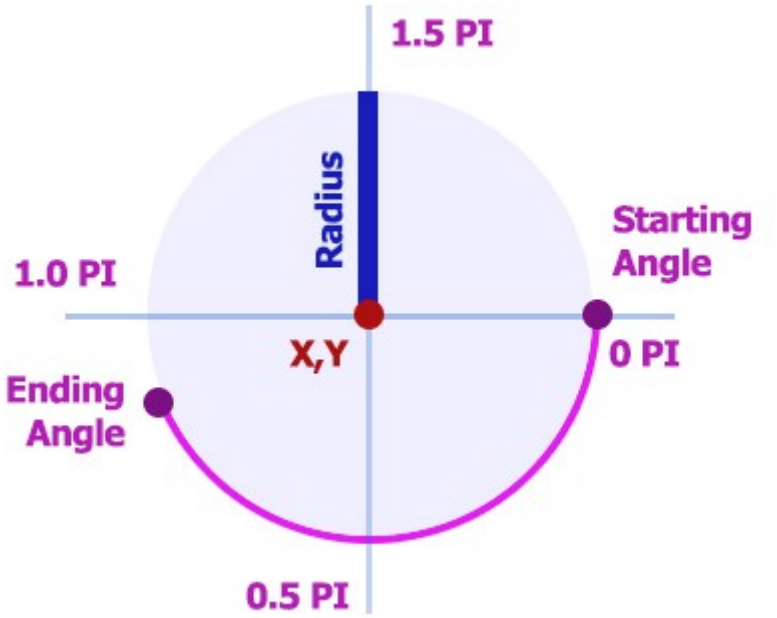
- `closePath()` Automatically connect to the nearest point to close the path.
- `fill()` Fill with solid color.
- `stroke()` Draw the outline.

ARCS & CIRCLES

CHAPTER-R/4-ARC.HTML

```
<!-- (A) HTML CANVAS -->
<canvas id="mycanvas" width="200" height="200"></canvas>
<script>
// (B) GET CANVAS CONTEXT
var ctx =
document.getElementById("mycanvas").getContext("2d");

// (C) DRAW ARC
// ARC (X, Y, RADIUS, START ANGLE, END ANGLE, DIRECTION)

// (C1) 1/4 CIRCLE
ctx.beginPath();
ctx.arc(50, 30, 20, 0, 0.5 * Math.PI);
ctx.strokeStyle = "red";
ctx.stroke();

// (C2) 1/2 CIRCLE
ctx.beginPath();
ctx.arc(120, 30, 20, 0, 1 * Math.PI);
ctx.strokeStyle = "green";
ctx.stroke();

// (C3) 3/4 CIRCLE
ctx.beginPath();
ctx.arc(50, 100, 20, 0, 1.5 * Math.PI);
```

```
ctx.strokeStyle = "blue";
ctx.stroke();

// (C4) CIRCLE
ctx.beginPath();
ctx.arc(120, 100, 20, 0, 2 * Math.PI);
ctx.fillStyle = "red";
ctx.fill();
</script>
```



Arcs and circles are the worst. Let's take this step-by-step.

- We use `arc()` to draw an arc or circle, and it takes in 6 parameters.
- The first two parameters are `X and Y`, the center of where to draw the circle or arc on the canvas.
- The third parameter is the `radius`… Self-explanatory.
- Followed by the `starting` and `ending` angle. Confusing, but let's do it the easy way:

- Start `0 PI`, end `0.5 PI` creates a "bottom right quarter circle".
- Start `0 PI`, end `1 PI` creates a "bottom half circle".
- Start `0 PI`, end `1.5 PI` creates a "missing top right circle".
- Start `0 PI`, end `2 PI` creates a "full circle".

- Lastly, we have `direction`. This is `false` by default, which draws the circle in a clockwise manner. Change this to `true` to draw in an anti-clockwise manner.

SAVE CANVAS – DOWNLOAD

CHAPTER-R/5-DOWNLOAD.HTML

```
<!-- (A) DUMMY CANVAS -->
<canvas id="demo" width="200" height="200"></canvas>

<script>
// (B) GET CANVAS + 2D
let canvas = document.getElementById("demo");
    ctx = canvas.getContext("2d");

// (C) DRAW DUMMY BLACK SQUARES
ctx.fillRect(0, 0, 100, 100);
ctx.fillRect(50, 50, 100, 100);

// (D) CREATE DOWNLOAD LINK
// image/jpg, image/png, image/gif, image/webp
let anchor = document.createElement("a");
anchor.download = "download.png";
anchor.href = canvas.toDataURL("image/png");

// (E) "FORCE DOWNLOAD"
```

```
anchor.click();
anchor.remove();

// (F) SAFER ALTERNATIVE - LET USER CLICK ON LINK
anchor.innerHTML = "Download";
document.body.appendChild(anchor);
</script>
```

So far so good? What good is a canvas if we cannot save it? In most modern browsers, we can actually right click on the canvas and pick the "save as" option. In Javascript, we have a roundabout trick:

- Create an HTML anchor – `var anc = document.createElement("a");`
- Set the link to the canvas data URL (base 64 encoded) – `anc.href = canvas.toDataURL(MIME TYPE);`
- "Force download" with `anc.click();`
- Take note, this may not always work due to security restriction. The safer way is to attach the link to the page, let the user manually click on it.

SAVE CANVAS – UPLOAD

CHAPTER-R/6-UPLOAD.HTML

```
<!-- (A) DUMMY CANVAS -->
<canvas id="demo" width="200" height="200"></canvas>

<script>
// (B) GET CANVAS + 2D
let canvas = document.getElementById("demo");
    ctx = canvas.getContext("2d");
```

```
// (C) DRAW DUMMY BLACK SQUARES
ctx.fillRect(0, 0, 100, 100);
ctx.fillRect(50, 50, 100, 100);

// (D) TO BLOB & UPLOAD
canvas.toBlob((blob) => {
  // (D1) CREATE FILE
  let file = new File([blob], "demo.png", { type:
  "image/png" });

  // (D2) UPLOAD
  var data = new FormData();
  data.append("up", file);
  fetch("SERVER-SCRIPT", { method:"POST", body:data })
  .then(res => res.text())
  .then((txt) => { console.log(txt); });
});
</script>
```

How about uploading? Remember `new blob()` from earlier? We simply turn the canvas into a file and upload it "as usual".

RESIZING IMAGES WITH CANVAS

CHAPTER-R/7-RESIZE.HTML

```
<!-- (A) DEMO CANVAS -->
<canvas id="demo"></canvas>

<script>
// (B) RESIZE ON IMAGE LOAD
var img = new Image();
img.onload = () => {
  // (B1) NEW DIMENSIONS - 50%
  let width = Math.ceil(0.5 * img.naturalWidth),
      height = Math.ceil(0.5 * img.naturalHeight);
```

```
  // (B2) CANVAS RESIZE
  let canvas = document.getElementById("demo"),
      ctx = canvas.getContext("2d");
  canvas.width = width;
  canvas.height = height;
  ctx.drawImage(img, 0, 0, width, height);
};

// (C) GO!
img.src = "eggs.jpg";
</script>
```

Manually drawing on a canvas is probably not very useful, so here's a practical example – Resizing an image. Not going to explain step-by-step, but the essentials:

- We load the source image with `var img = new Image()` and `img.src="IMAGE.PNG"`.
- Proceed with the resize only when the image is fully loaded – `img.onload = () => { RESIZE };`
- The resizing is as simple as `drawImage(img, X, Y, WIDTH, HEIGHT)`. Basically, copy the source image onto the canvas at the specified coordinates (`X, Y`) with the specified dimensions (`WIDTH, HEIGHT`).

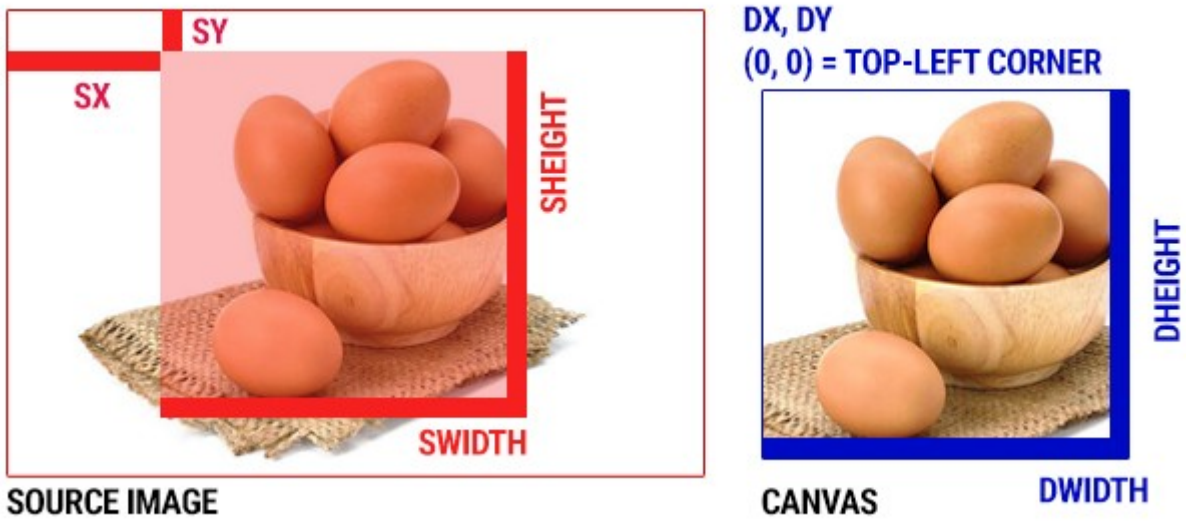
CROP IMAGES WITH CANVAS

CHAPTER-R/8-CROP.HTML

```
<!-- (A) DEMO CANVAS -->
<canvas id="demo" width="300" height="300"></canvas>
```

```
<script>
// (B) CROP ON IMAGE LOAD
var img = new Image();
img.onload = () => {
  // (B1) GET CANVAS
  let canvas = document.getElementById("demo"),
      ctx = canvas.getContext("2d");

  // (B2) DO YOUR OWN CROP CALCULATIONS...
  ctx.drawImage(img, 170, 20, 300, 300, 0, 0, 300, 300);
};

// (C) GO!
img.src = "eggs.jpg";
</script>
```

If we can resize an image, we can also crop it. Not going to explain line-by-line again, but the star is `ctx.drawImage()` once again. As you can see, there are are a lot of parameters now – `IMAGE, SX, SY, SWIDTH, SHEIGHT, DX, DY, DWIDTH, DHEIGHT`. Not to be confused, we are just copying a part of the source image onto the canvas to do the "crop". This diagram will explain all the parameters:



WATERMARK (COMPOSITE) IMAGES

CHAPTER-R/9-WATERMARK.HTML

```
<!-- (A) DEMO CANVAS -->
<canvas id="demo"></canvas>

<script>
// (B) IMAGES + CANVAS
var iBack = new Image(),
    iMark = new Image(),
    iText = "FOO BAR",
    loaded = 0;
    canvas = document.getElementById("demo"),
    ctx = canvas.getContext("2d");

// (C) PROCEED WHEN ALL IMAGES LOADED
function cmark () { loaded++; if (loaded==2) {
  // (C1) MAIN IMAGE
  canvas.width = iBack.naturalWidth;
  canvas.height = iBack.naturalHeight;
  ctx.drawImage(iBack, 0, 0, iBack.naturalWidth,
iBack.naturalHeight);

  // (C2) WATERMARK IMAGE
  ctx.drawImage(iMark, 0, 0, iMark.naturalWidth,
iMark.naturalHeight);

  // (C3) TEXT
  ctx.font = "bold 24px Arial";
  ctx.fillStyle = "rgba(255, 0, 0, 0.5)";
  ctx.fillText(iText, 100, 30);
}}

// (D) GO!
iBack.onload = cmark;
iMark.onload = cmark;
```

```
iBack.src = "eggs.jpg";
iMark.src = "potato.png";
</script>
```

Add watermark or text on an image? No problem – Just use what we have walked through earlier. Add the "main background" image to the canvas first, then composite whatever you want on top.

PICK IMAGE & SHOW ON CANVAS

CHAPTER-R/10-PICK.HTML

```
<!-- (A) FILE PICKER + CANVAS -->
<input type="file" id="picker"
accept="image/jpeg,image/png,image/webp"/>
<canvas id="demo"></canvas>

<script>
// (B) GET PICKER + CANVAS
let picker = document.getElementById("picker"),
    canvas = document.getElementById("demo"),
    ctx = canvas.getContext("2d");

// (B) AUTO START ON SELECT FILE
picker.onchange = () => {
  // (B1) CREATE NEW IMAGE + URL TO SELECTED FILE
  let img = new Image(),
      surl = URL.createObjectURL(picker.files[0]);

  // (B2) PUT ONTO CANVAS ON IMAGE LOAD
  img.onload = () => {
    canvas.width = img.naturalWidth;
    canvas.height = img.naturalHeight;
    ctx.drawImage(img, 0, 0, img.naturalWidth,
    img.naturalHeight);
    URL.revokeObjectURL(surl);
```

```
  };

  // (B3) GO!
  img.src = surl;
};
</script>
```

Of course, we can. This just needs a little roundabout:

- Create an object URL to the selected file – `URL.createObjectURL(picker.files[0])`
- The rest are the same. Create an image object, set the source to the above URL, draw the image onto the canvas.

LINKS & REFERENCES

- Canvas API – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Canvas

- CHAPTER 5 -

SPEECH RECOGNITION

EASIER THAN EXPECTED

Voice recognition. Most people will probably think that it is a complicated process. Actually, no. It's long-winded, but still pretty straightforward.

SPEECH TO TEXT

CHAPTER-S/1-SPEECH-TEXT.HTML

```
<!-- (A) TEST HTML -->
<div id="demo"></div>

<script>
// (B) GET HTML
let demo = document.getElementById("demo");

// (C) ASK FOR MIC PERMISSION
navigator.mediaDevices.getUserMedia({ audio: true })

// (D) OK - INIT
.then((stream) => {
  // (D1) SPEECH RECOGNITION OBJECT & SETTINGS
  const SR = window.SpeechRecognition ||
             window.webkitSpeechRecognition;
  let recog = new SR();
  recog.lang = "en-US";
  recog.continuous = false;
  recog.interimResults = false;

  // (D2) ON SPEECH RECOGNITION - SHOW TRANSCRIPT
  recog.onresult = (evt) => {
    demo.innerHTML += evt.results[0][0]
                      .transcript.toLowerCase();
    recog.stop();
  };
```

```
  // (D3) ON SPEECH RECOGNITION ERROR
  recog.onerror = (err) =>
  { demo.innerHTML = err.message; };

  // (D4) START!
  demo.innerHTML = "Speak now - ";
  recog.start();
})

// (E) ERROR
.catch((err) => { demo.innerHTML = err.message; });
</script>
```

For the first example, we have a simple speech-to-text – Which is essentially what the voice recognition API does. Important parts:

- **(C)** `navigator.mediaDevices.getUserMedia({ audio: true })` – Ask for access to the microphone. Same as push notifications, once denied, we can only ask the user to manually give permission (click on icon beside URL).
- **(D1)** `let recog = new SR()` and the settings – Yes, we can set speech recognition non-English languages, but this is still a gray area at the time of writing. We do not know if the browser will support other languages, and there is no reliable way to check.
- **(D2)** `recog.onresult` should be self-explanatory enough. But getting the text transcript is kind of a dumb-dumb thing – `evt.results[0][0].transcript.toLowerCase()`. It is what it is, we will just use `results[0][0].transcript` as the developers set.
- `recog.start()` to start recognition, `recog.stop()` to end.

VOICE SEARCH

CHAPTER-S/2-SPEECH-SEARCH.HTML

```
<form onsubmit="return false;">
  <!-- (A) USUAL SEARCH FIELD & BUTTON -->
  <input type="text" id="sfield"/>
  <button type="submit" disabled>
    <span class="material-icons">search</span>
  </button>

  <!-- (B) SPEECH SEARCH -->
  <button id="sspeech" disabled>
    <span id="ssicon" class="material-icons">mic</span>
  </button>
</form>
```

Let's put that "speech-to-text" into good use next. Here, we have a "usual search form" with an additional voice search button.

CHAPTER-S/2-SPEECH-SEARCH.JS

```
var voice = {
  // (A) INIT SPEECH RECOGNITION
  sfield : null, // HTML SEARCH FIELD
  sspeech : null, // HTML VOICE SEARCH BUTTON
  ssicon : null, // HTML VOICE SEARCH ICON
  recog : null, // SPEECH RECOGNITION OBJECT
  init : () => {
    // (A1) GET HTML ELEMENTS
    voice.sfield = document.getElementById("sfield");
    voice.sspeech = document.getElementById("sspeech");
    voice.ssicon = document.getElementById("ssicon");

    // (A2) GET MICROPHONE ACCESS
    navigator.mediaDevices.getUserMedia({ audio: true })
```

```
  .then((stream) => {
    // (A3) SPEECH RECOGNITION OBJECT + SETTINGS
    const SR = window.SpeechRecognition ||
               window.webkitSpeechRecognition;
    voice.recog = new SR();
    voice.recog.lang = "en-US";
    voice.recog.continuous = false;
    voice.recog.interimResults = false;

    // (A4) POPUPLATE SEARCH FIELD ON SPEECH RECOGNITION
    voice.recog.onresult = (evt) => {
      let said = evt.results[0][0]
                 .transcript.toLowerCase();
      voice.sfield.value = said;
      voice.stop();
      // SUBMIT SEARCH FORM OR AJAX SEARCH IF YOU WANT
    };

    // (A5) ON SPEECH RECOGNITION ERROR
    voice.recog.onerror = (err) =>
                          { console.error(err); };

    // (A6) READY!
    voice.sspeech.disabled = false;
    voice.stop();
  })
  .catch((err) => { console.error(err); });
},

// (B) START SPEECH RECOGNITION
start : () => {
  voice.recog.start();
  voice.sspeech.onclick = voice.stop;
  voice.ssicon.innerHTML = "record_voice_over";
},

// (C) STOP/CANCEL SPEECH RECOGNITION
```

```
  stop : () => {
    voice.recog.stop();
    voice.sspeech.onclick = voice.start;
    voice.ssicon.innerHTML = "mic";
  }
};
window.addEventListener("DOMContentLoaded", voice.init);
```

This may look confusing, but all of these are essentially the same as the speech-to-text. Except that it has some bells and whistles, plus, it populates a search field instead.

VOICE COMMANDS

CHAPTER-S/3-SPEECH-CMD.HTML

```
<!-- (A) DEMO WRAPPER -->
<div id="vwrap"></div>

<!-- (B) DEMO BUTTON -->
<input type="button" id="vbtn" value="Loading" disabled/>
<div>Try "power on", "power off", or "say hello".</div>
```

Ever wonder if we can create our own voice command driven web page? Yes, we can.

CHAPTER-S/3-SPEECH-CMD.JS

```
var voice = {
  // (A) INIT VOICE COMMAND
  wrap : null, // HTML DEMO <DIV> WRAPPER
  btn : null, // HTML DEMO BUTTON
  recog : null, // SPEECH RECOGNITION OBJECT
  init : () => {
    // (A1) GET HTML ELEMENTS
```

```
  voice.wrap = document.getElementById("vwrap");
  voice.btn = document.getElementById("vbtn");

  // (A2) GET MIC ACCESS PERMISSION
  navigator.mediaDevices.getUserMedia({ audio: true })
  .then((stream) => {
    // (A3) SPEECH RECOGNITION OBJECT & SETTINGS
    const SR = window.SpeechRecognition ||
               window.webkitSpeechRecognition;
    voice.recog = new SR();
    voice.recog.lang = "en-US";
    voice.recog.continuous = false;
    voice.recog.interimResults = false;

    // (A4) ON SPEECH RECOGNITION - RUN COMMAND
    voice.recog.onresult = (evt) => {
      let said = evt.results[0][0].
                 transcript.toLowerCase();
      if (cmd[said]) { cmd[said](); }
      else { said += " (command not found)"; }
      voice.wrap.innerHTML = said;
      voice.stop();
    };

    // (A5) ON SPEECH RECOGNITION ERROR
    voice.recog.onerror = (err) =>
                          { console.error(evt); };

    // (A6) READY!
    voice.btn.disabled = false;
    voice.stop();
  })
  .catch((err) => {
    console.error(err);
    voice.wrap.innerHTML = "Please enable access and
                           attach a microphone.";
  });
```

```
  },

  // (B) START SPEECH RECOGNITION
  start : () => {
    voice.recog.start();
    voice.btn.onclick = voice.stop;
    voice.btn.value = "Speak Now Or Click To Cancel";
  },

  // (C) STOP/CANCEL SPEECH RECOGNITION
  stop : () => {
    voice.recog.stop();
    voice.btn.onclick = voice.start;
    voice.btn.value = "Press To Speak";
  }
};
window.addEventListener("DOMContentLoaded", voice.init);

// (D) COMMANDS LIST
var cmd = {
  "power on" : () => {
    voice.wrap.style.backgroundColor = "yellow";
    voice.wrap.style.color = "black";
  },

  "power off" : () => {
    voice.wrap.style.backgroundColor = "black";
    voice.wrap.style.color = "white";
  },

  "say hello" : () => {
    alert("Hello World!");
  }
};
```

Once again, this is long-winded, but does the same speech-to-text. The essentials:

- **(D)** We define a whole list of functions in an object. `var cmd = { "VOICE COMMAND" => FUNCTION }`
- **(A4)** On getting the text transcription, we attempt to "map and run" the voice command – `cmd[SAID]()`.

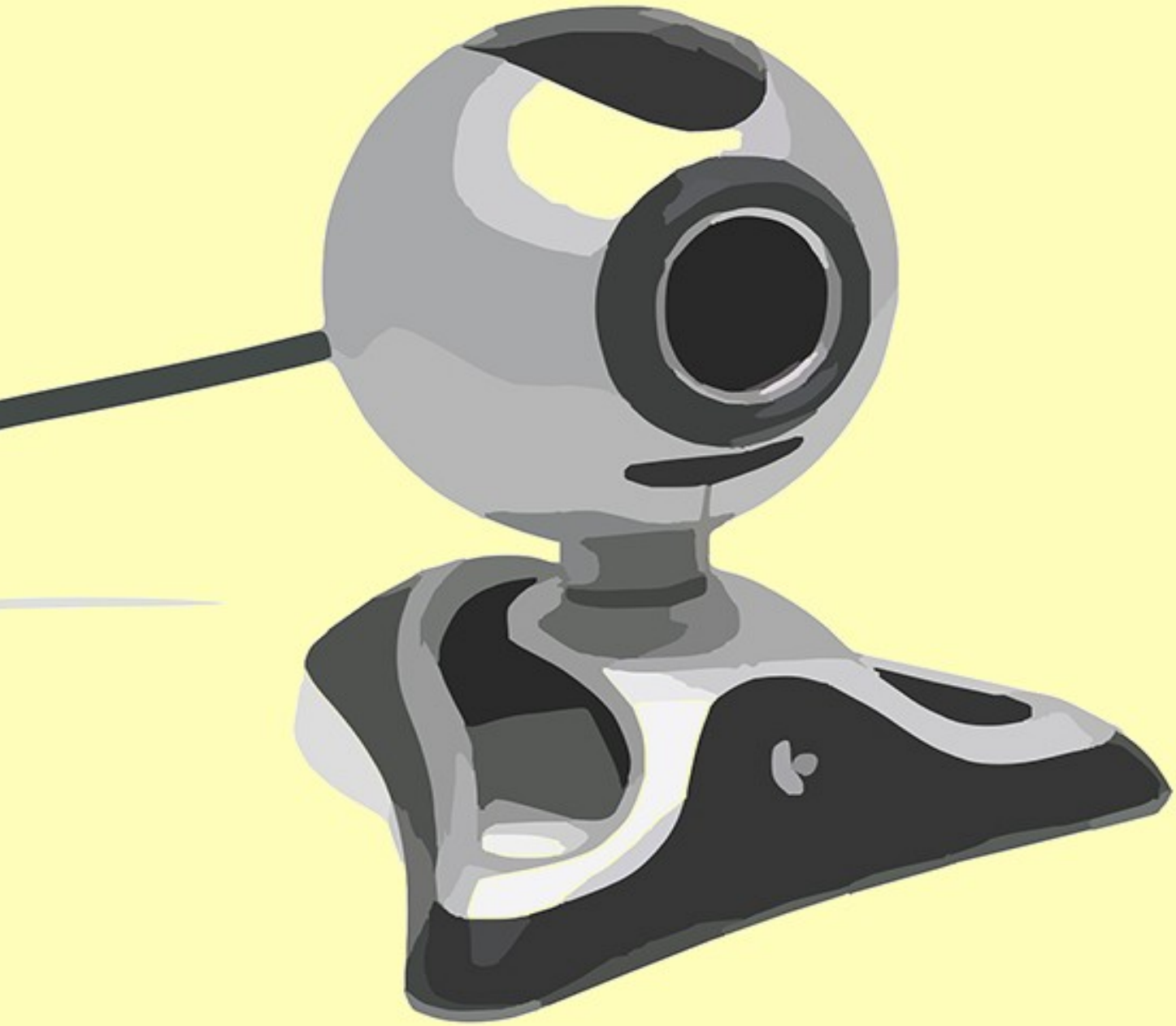
Yes, it's just a simple trick.

LINKS & REFERENCES

- Speech Recognition API – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Speech Recognition



- CHAPTER T -

JAVASCRIPT WEBCAM

JAVASCRIPT CAMERA

Webcams have been around for a long time now, and it is made even more popular with mobile devices. This should not be a surprise by now – We can access webcams with Javascript, take photos and videos.

VERY SIMPLE LIVE FEED

CHAPTER-T/1-LIVE-FEED.HTML

```
<!-- (A) VIDEO TAG -->
<video id="kam-live" autoplay></video>

<script>
// (B) GET PERMISSION TO ACCESS CAMERA
navigator.mediaDevices.getUserMedia({ video: true
/*, audio: true*/ })

// (C) ATTACH LIVE STREAM TO <VIDEO>
.then((stream) => {
  document.getElementById("kam-live").srcObject = stream;
})

// (D) HANDLE ERRORS
.catch((err) => { console.error(err); });
</script>
```

A) Start by defining an empty HTML video tag with `autoplay`.

B) Then, we need to get access permission to the webcam with `navigator.mediaDevices.getUserMedia({ video: true })`. If you need to capture the audio, also include `audio: true` in the options.

C) Attach the webcam stream into the `<video>` tag.

That's all for a simple live feed… But this does nothing productive. Let's build a simple "camera" app next.

TAKING PHOTOS WITH WEBCAM

STEP 1) WEBCAM HTML

CHAPTER-T/2-WEBCAM.HTML

```
<!-- (A) VIDEO LIVE FEED -->
<video id="kam-live" autoplay></video>

<!-- (B) SNAPSHOTS -->
<div id="kam-snaps"></div>

<!-- (C) CONTROLS -->
<div id="kam-controls">
  <button id="kam-take" disabled>
    <span class="material-icons">photo_camera</span>
  </button>
  <button id="kam-save" disabled>
    <span class="material-icons">save</span>
  </button>
</div>
```

A) Same old <video> tag for the live feed.

B) Camera snapshots will be placed here.

C) A button to take a photo, another to offer a "save as".

STEP 2) WEBCAM PROPERTIES

CHAPTER-T/2-WEBCAM.JS

```
// (A) INITIALIZE
hVid : null, hSnaps :null, hTake : null, hSave : null,
init : () => {
  // (A1) GET HTML ELEMENTS
  webkam.hVid = document.getElementById("kam-live"),
  webkam.hSnaps = document.getElementById("kam-snaps"),
  webkam.hTake = document.getElementById("kam-take"),
  webkam.hSave = document.getElementById("kam-save");

  // (A2) GET USER PERMISSION TO ACCESS CAMERA
  navigator.mediaDevices.getUserMedia({ video: true })
  .then((stream) => {
    // "LIVE FEED" WEB CAM TO <VIDEO>
    webkam.hVid.srcObject = stream;

    // ENABLE BUTTONS
    webkam.hTake.onclick = webkam.take;
    webkam.hSave.onclick = webkam.save;
    webkam.hTake.disabled = false;
    webkam.hSave.disabled = false;
  })
  .catch((err) => { console.error(err); });
}
window.addEventListener("load", webkam.init);
```

`webkam.init()` will run on window load to initialize the app:

- **(A1)** Get the HTML elements.
- **(A2)** Same old get webcam permission, set live feed. But small extra now, also enable the HTML buttons.

STEP 3) TAKE A SNAPSHOT

CHAPTER-T/2-WEBCAM.JS

```
// (B) HELPER - SNAP VIDEO FRAME TO CANVAS
snap : () => {
  // (B1) CREATE NEW CANVAS
  let canvas = document.createElement("canvas"),
      ctx = canvas.getContext("2d"),
      vWidth = webkam.hVid.videoWidth,
      vHeight = webkam.hVid.videoHeight;

  // (B2) CAPTURE VIDEO FRAME TO CANVAS
  canvas.width = vWidth;
  canvas.height = vHeight;
  ctx.drawImage(webkam.hVid, 0, 0, vWidth, vHeight);

  // (B3) DONE
  return canvas;
},

// (C) TAKE A SNAPSHOT - PUT CANVAS INTO <DIV> WRAPPER
take : () => {
  webkam.hSnaps.appendChild(webkam.snap());
}
```

Remember the canvas? Yes, to "take a photo", we literally:

- Create a new HTML canvas element.
- Capture the current video frame onto the canvas.

STEP 4) SAVE SNAPSHOT AS IMAGE

CHAPTER-T/2-WEBCAM.JS

```
// (D) SAVE SNAPSHOT
save : () => {
  // (D1) TAKE A SNAPSHOT, CREATE DOWNLOAD LINK
  let canvas = webkam.snap(),
      anchor = document.createElement("a");
  anchor.href = canvas.toDataURL("image/png");
  anchor.download = "snap.png";

  // (D2) "FORCE DOWNLOAD" - MAY NOT ALWAYS WORK!
  anchor.click();
  anchor.remove();
  canvas.remove();

  // (D3) SAFER - LET USERS MANUAL CLICK
  // anchor.appendChild(canvas);
  // webkam.hSnaps.appendChild(anchor);
}
```

Remember how to save an HTML canvas as an image? This is the same.

P2P LIVE VIDEO CHAT

STEP 1) PEERJS SERVER

CHAPTER-T/3A-SERVER.JS

```
const { PeerServer } = require("peer");
const peerServer = PeerServer({
  port: 9000,
  path: "/myapp"
  /*
  ssl : {
```

```
    key: fs.readFileSync("/path/ssl.key"),
    cert: fs.readFileSync("/pathcertificate.crt")
  }
  */
});
```

Remember this fella from earlier? We will now use this to create a simple P2P video chat. Run `npm install peer` if you have deleted the package, then launch this in the command line – `node 3a-server.js`. Take note of SSL here. In your live application, please make sure that you have a valid SSL cert, or things will not work – More on that below.

STEP 2) PEER A

CHAPTER-T/3B-PEER-A.HTML

```
<!-- (A) VIDEO TAGS -->
You: <video id="vMe" autoplay></video>
Caller: <video id="vCaller" autoplay></video>

<!-- (B) VIDEO CALL -->
<script
src="https://cdnjs.cloudflare.com/ajax/libs/peerjs/1.3.2/p
eerjs.min.js"></script>
<script>
async function kamcall (id) {
  // (B1) VIDEO LIVE FEED
  let streamMe = null;
  try {
    streamMe = await navigator.mediaDevices.getUserMedia({
    video: true, audio: true });
  } catch(err) {
    console.error(err);
    return false;
```

```
  }
  document.getElementById("vMe").srcObject = streamMe;

  // (B2) HANDSHAKE WITH PEER SERVER
  const peer = new Peer(id, {
    host: "192.168.0.101", // CHANGE TO YOUR OWN!
    port: 9000,
    path: "/myapp"
  });

  // (B3) AUTO ANSWER CALLS
  peer.on("call", (call) => {
    call.answer(streamMe);
    call.on("stream", (streamCaller) => {
      document.getElementById("vCaller").srcObject =
      streamCaller;
    });
  });
}
kamcall("PEER-A");
</script>
```

This should look very familiar, here's a quick walk through:

- **(A)** We now have 2 `<video>` tags. One for your own live feed, the other to show the caller's video feed.
- **(B1)** Same old "get video AND audio permission", feed into <video> tag.
- **(B2)** Same old handshake with peer server.
- **(B3)** Take note, it's `peer.on("call")` now. In this example, we simply accept a call, then put the caller's video feed into the other `<video>` tag.

STEP 3) PEER B

CHAPTER-T/3C-PEER-B.HTML

```
// (B3) CALL "PEER-A"
peer.on("open", () => {
  console.log("READY!");
  let call = peer.call("PEER-A", streamMe);
  call.on("stream", (streamCaller) => {
    document.getElementById("vCaller").srcObject =
    streamCaller;
  });
});
```

Look no further, this is the same as Peer A. Except that we make a video call to Peer A once the handshake is complete.

MANY OTHER USES

There are many more uses for the webcam that I can think of:

- Do an online search for "Javascript QR code scanner" or "Barcode scanner" – Yes, we can bake these directly into the web app itself.
- Live stream. Remember web socket? Send the video stream to the server, then broadcast it live to everyone. Although the bit rate, chunking, and whatever controls are going to be a royal pain to deal with.
- If you want, you can even save the video on the server.
- Home security or build your own web-based local intercom system.

This is probably endless, so I shall stop here.

LINKS & REFERENCES

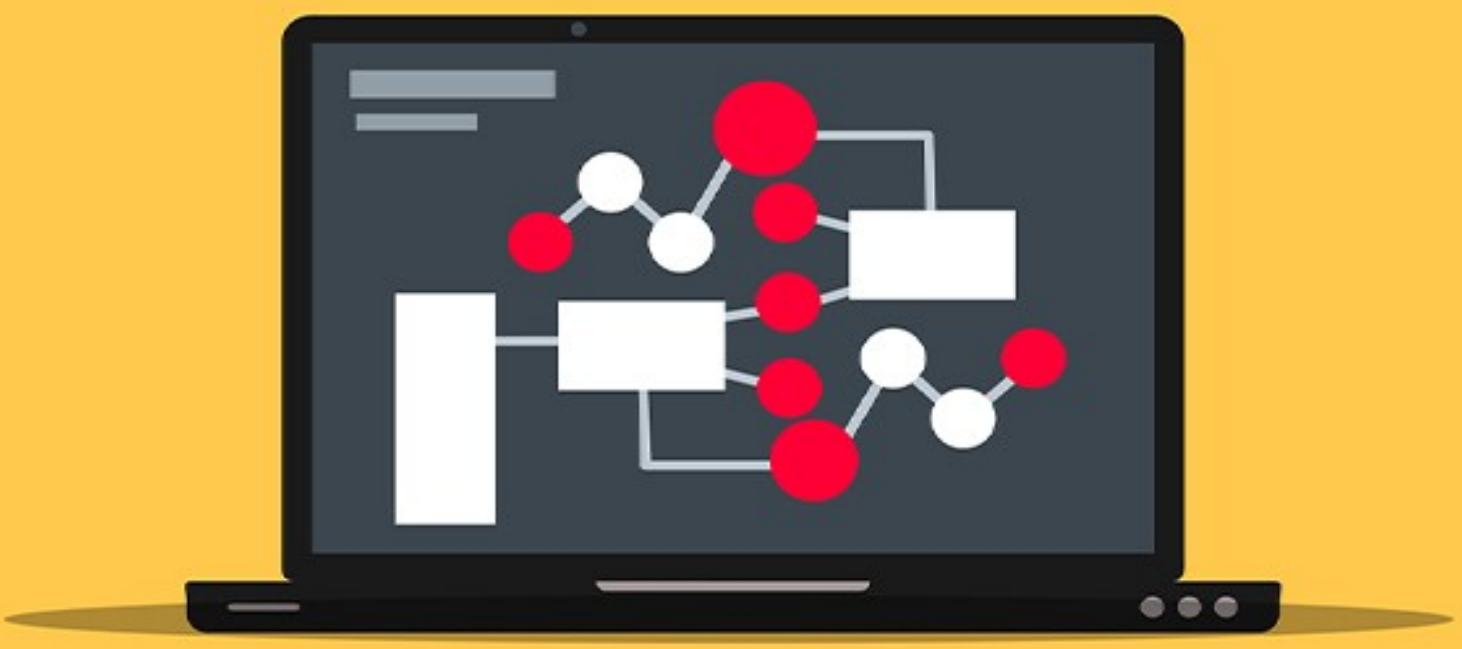
- Get User Media – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Get User Media (Stream) Compatibility - CanIUse

- CHAPTER U -

SCREEN CAPTURE



"ALTERNATE" VIDEO STREAM

If we can stream audio and video, what's stopping us from sharing screens? Yes, this feature is also baked directly inside most "Grade A" modern browsers as well.

SIMPLE SCREEN SHARING

THE HTML

CHAPTER-U/SCREEN-CAP.HTML

```
<!-- (A) HTML INTERFACE -->
<video id="video" style="width:100%;" autoplay></video>
<button id="start">Start Capture</button>
<button id="stop">Stop Capture</button>
```

It's the same old `<video>` tag again. You should already have an idea of what we will do in the Javascript next.

START SCREEN CAPTURE

CHAPTER-U/SCREEN-CAP.HTML

```
// (B) GET HTML ELEMENTS
const hVid = document.getElementById("video"),
      hStart = document.getElementById("start"),
      hStop = document.getElementById("stop");

// (C) START CAPTURE
hStart.onclick = () => {
  navigator.mediaDevices.getDisplayMedia({
    video: { cursor: "always" },
    audio: false
```

```
  })
  .then((stream) => { hVid.srcObject = stream; })
  .catch((err) => { console.error(err); });
};
```

Does this look familiar? I am sure it does.

- To get permission to the webcam, we use `getUserMedia()`.
- To get permission to the screen, we use `getDisplayMedia()`.

That's all. The rest are all the same – Once the user gives permission, we "map" the stream into the `<video>` tag.

STOP SCREEN CAPTURE

CHAPTER-U/SCREEN-CAP.HTML

```
// (D) STOP CAPTURE
hStop.onclick = () => {
  hVid.srcObject.getTracks().forEach(track =>
  track.stop());
  hVid.srcObject = null;
};
```

To stop the sharing, we simply run through all the "video tracks" and stop the one-by-one.

TAKING A SCREENSHOT

All right, I am getting lazy, but it's the same as the webcam. Create a canvas, capture a frame from the video onto the canvas.

SCREEN SHARING

It's the same as again, use PeerJS to do the same video call. But instead of feeding in the webcam, we feed in the screen capture instead.

LINKS & REFERENCES

- Get Display Media – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Get Display Media

- CHAPTER V -

SENSORS

SMALL BUT ADVANCED

Mobile devices. They fit into pockets, and we use them without thinking much these days. But the amount of technologies packed inside is actually amazing – GPS, camera, audio, video, gyroscope, light sensors, wireless, etc… Let us walk through how to use some of these mobile sensors in this chapter.

GPS

CHAPTER-V/1-GPS.HTML

```
<!-- (A) DEMO HTML -->
Lat <div id="lat"></div>
Lng <div id="lng"></div>
<input type="button" value="GPS" onclick="getGPS()"/>

<script>
function getGPS () {
  // (B) GET HTML ELEMENTS
  let lat = document.getElementById("lat"),
      lng = document.getElementById("lng");

  // (C) GET GPS COORDINATES
  navigator.geolocation.getCurrentPosition(
    // (C1) ON SUCCESS
    (pos) => {
      lat.innerHTML = pos.coords.latitude;
      lng.innerHTML = pos.coords.longitude;
    },
    // (C2) ON FAILURE - OPTIONAL
    (err) => {
      lat.innerHTML = err.message;
      console.error(err);
```

```
    },
    // (C3) OPTIONS - OPTIONAL
    {
      enableHighAccuracy: true,
      timeout: 5000,
      maximumAge: 0
    }
  );
}
</script>
```

- Use `navigator.geolocation.getCurrentPosition()` to request for the GPS coordinates, it takes in 3 parameters.
- The first parameter is a function to call on successfully getting the GPS coordinates – `pos.coords.latitude` and `pos.coords.longitude` are probably all we are interested in.
- The second parameter is optional, a function to deal with errors. This is triggered when the user denies access to the GPS and on GPS timeout. Yes, when the device fails to get the coordinates.
- Lastly, the GPS options. Optional again.
  - `enableHighAccuracy` Self-explanatory. Drains a little more battery, but high accuracy.
  - `timeout` When to give up on not getting a GPS signal.
  - `maximumAge` In milliseconds. If set, the device will cache the GPS coordinates for the set time. This is to save the battery.

HOW ABOUT MAPS?

There are a dozen of map services on the Internet. It is not possible to

cover all of them, so check out some of these in your own time:

- Google Maps
- HERE Maps
- Apple Maps
- Bing Maps
- Mapbox
- Mapzen
- Leaflet
- Open Layers
- Thunderforest

GYROSCOPE

CHAPTER-V/2-GYRO.HTML

```
<!-- (A) DEMO HTML -->
X <div id="gx"></div>
Y <div id="gy"></div>
Z <div id="gz"></div>
<input type="button" value="Gyroscope"
onclick="getGyro()"/>

<script>
function getGyro () {
  // (B) GET HTML ELEMENTS
  let gx = document.getElementById("gx"),
      gy = document.getElementById("gy"),
      gz = document.getElementById("gz");
```

```
  // (C) READ GYRO
  let gyro = new Gyroscope({frequency: 10});
  gyro.addEventListener("reading", (e) => {
    gx.innerHTML = gyro.x;
    gy.innerHTML = gyro.y;
    gz.innerHTML = gyro.z;
  });
  gyro.start();

  // (D) ON ERROR - OPTIONAL
  gyro.addEventListener("error", (e) => {
    gx.innerHTML = e.message;
  });
}
</script>
```

- Create `let gyro = new Gyroscope()` object. The frequency is the number of scans per second. The higher the number, the more "sensitive" it is .
- Define what to do on getting a reading – `gyro.onreading = DO SOMEHTING`. This will return the XYZ "angular velocity of the device".
- Start – `gyro.start()`.

Sorry, I am not a physicist. My layman understanding of a Gyroscope is "how much in which direction is the device tilted". For a better explanation, I will just point to a Google image search on "Gyroscope XYZ".

ACCELEROMETER

CHAPTER-V/3-ACCELEROMETER.HTML

```
<!-- (A) DEMO HTML -->
X <div id="ax"></div>
Y <div id="ay"></div>
Z <div id="az"></div>
<input type="button" value="Accelerometer"
onclick="getAccel()"/>

<script>
function getAccel () {
  // (B) GET HTML ELEMENTS
  let ax = document.getElementById("ax"),
      ay = document.getElementById("ay"),
      az = document.getElementById("az");

  // (C) READ ACCELEROMETER
  let accel = new Accelerometer({frequency: 10});
  accel.addEventListener("reading", (e) => {
    ax.innerHTML = accel.x;
    ay.innerHTML = accel.y;
    az.innerHTML = accel.z;
  });
  accel.start();

  // (D) ON ERROR - OPTIONAL
  accel.addEventListener("error", (e) => {
    ax.innerHTML = e.message;
  });
}
</script>
```

Right. As it seems, an "accelerometer" is different from "gyroscope".

According to online technical terms:

- Gyroscope – Angular velocity.
- Accelerometer - Acceleration of the device.

So my layman understanding:

- Gyroscope – How much in which direction is the device tilted.
- Accelerometer – How fast in which direction is the device tilted or moved at.

LIGHT METER

CHAPTER-V/4-LIGHT.HTML

```
<!-- (A) DEMO HTML -->
Illuminance <div id="illum"></div>
<input type="button" value="Light Meter"
onclick="getLum()"/>

<script>
function getLum () {
  // (B) GET HTML ELEMENTS
  let il = document.getElementById("illum");

  // (C) READ LIGHT SENSOR
  let ls = new AmbientLightSensor();
  ls.addEventListener("reading", (e) => {
    il.innerHTML = ls.illuminance;
  });
  ls.start();

  // (D) ON ERROR - OPTIONAL
  ls.addEventListener("error", (e) => {
    il.innerHTML = e.message;
  });
```

```
}
</script>
```

This should be self-explanatory. The light sensor will return the current light level in LUX.

EVEN MORE SENSORS

We have covered some of the common mobile device sensors, but there are actually more.

- Gravity Sensor
- Magnetometer
- Proximity Sensor
- Linear Acceleration Sensor

No idea what these even do…

DO YOUR FEATURE CHECKS!

Yes, not all devices have all the sensors built in. A reminder to do your own homework on checking for the required feature, implement your own fallback and warning messages.

LINKS & REFERENCES

- Get Current Position – MDN
- Gyroscope – MDN
- Accelerometer – MDN

- Light Sensor – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

- Get Current Position
- Gyroscope
- Accelerometer
- Light Sensor

- CHAPTER W -

VIBRATION

SUPER SHORT CHAPTER

Let's go.

VIBRATION CONTROL

CHAPTER-W/VIBRATE.HTML

```
<!-- (A) DEMO HTML -->
<input type="button" value="Single" onclick="vOne()"/>
<input type="button" value="Many" onclick="vMany()"/>
<input type="button" value="Stop" onclick="vStop()"/>

<script>
function vOne () {
  navigator.vibrate(1000);
}
function vMany () {
  navigator.vibrate([200, 50, 200, 50]);
}
function vStop () {
  navigator.vibrate(0);
  navigator.vibrate([]);
}
</script>
```

- Use `navigator.vibrate(MS)` to vibrate once.
- Use `navigator.vibrate([MS, MS, …])` for a vibration pattern.
- Use `navigator.vibrate(0)` or `navigator.vibrate([])` to stop vibrating.

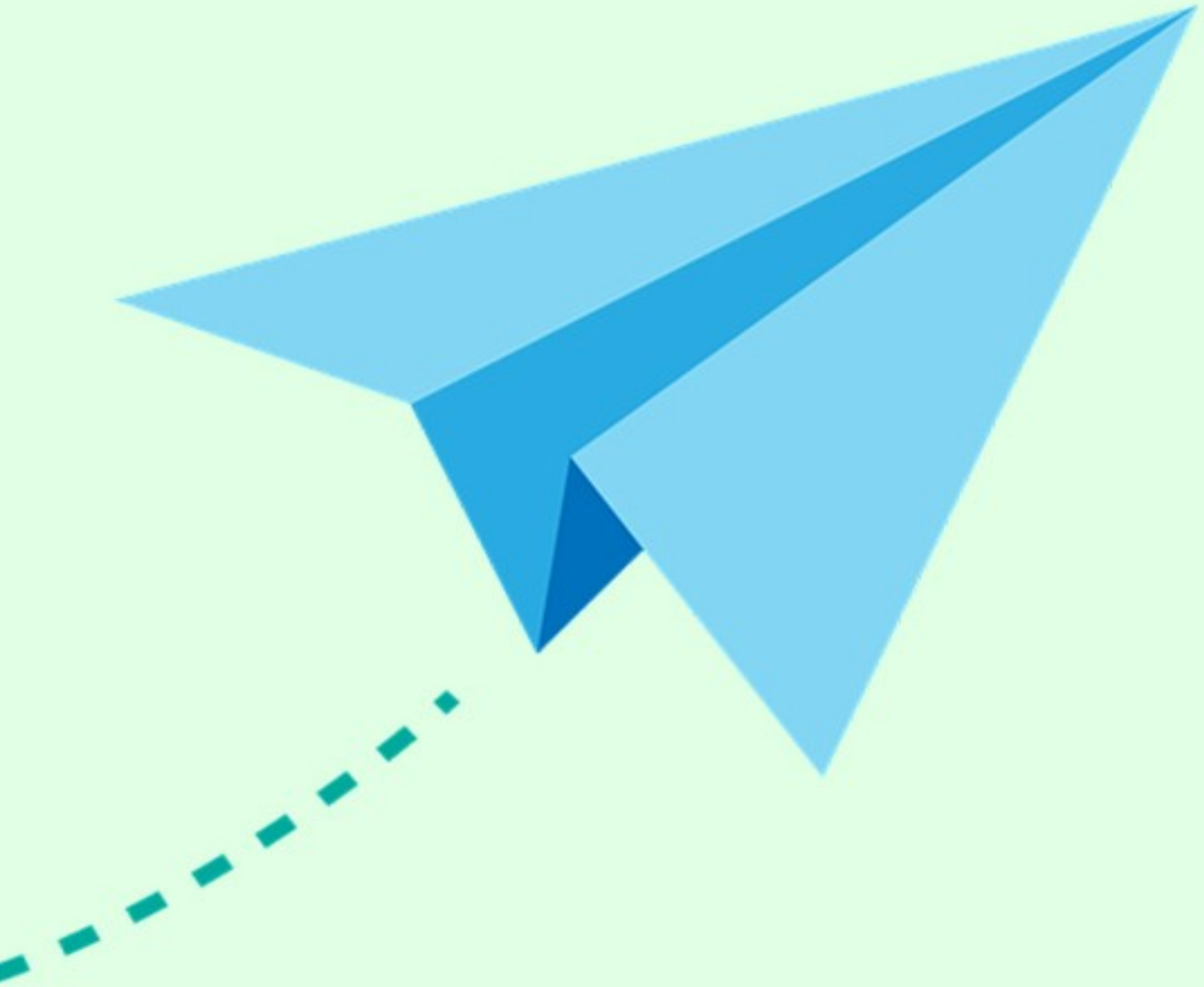
The end.

LINKS & REFERENCES

- Vibrate – MDN

COMPATIBILITY CHECKS (WITH CANIUSE)

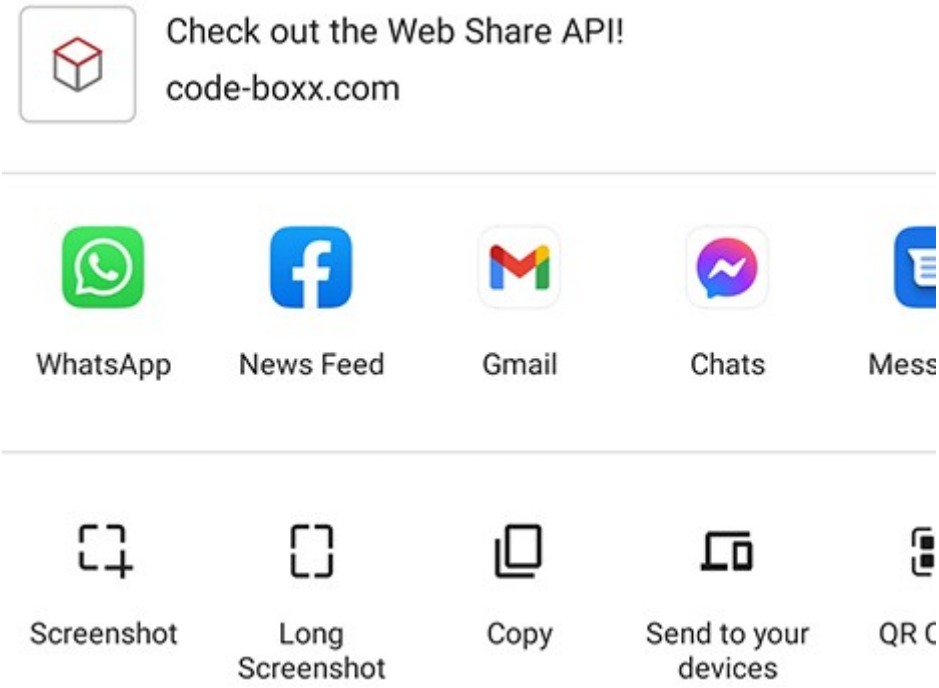
- Vibrate



- CHAPTER X -

WEB SHARE

NATIVE SHARE BUTTON



I am sure that you are not a stranger to "social share buttons" in this age. But know what? There is an actual native "share" implementation in Javascript. Take note though, this mostly only applies to mobile devices at the time of writing.

JAVASCRIPT SHARE

CHAPTER-X/WEB-SHARE.HTML

```
<!-- (A) DEMO BUTTON -->
<input type="button" value="Share" id="wshare"
onclick="wshare()"/>

<script>
// (B) SHARE
function wshare () {
  // (B1) DATA TO SHARE
```

```
  const data = {
    title: "Code Boxx",
    text: "Check out the Web Share API!",
    url: "https://code-boxx.com"
    // files: [FILE OBJECT, FILE OBJECT, ...]
  };

  // (B2) SHARE DIALOG
  navigator.share(data)
  .then(() => { alert("SHARED"); })
  .catch((err) => { alert(err.message); });
}
</script>
```

- Just use `navigator.share(DATA)` to activate the share dialog.
- The share data takes in 4 parameters.
  - `title` Title of the share.
  - `text` Description or share text.
  - `url` Self-explanatory.
  - `files` If sharing files, this is an array of file objects.

LINKS & REFERENCES

- Web Share – MDN

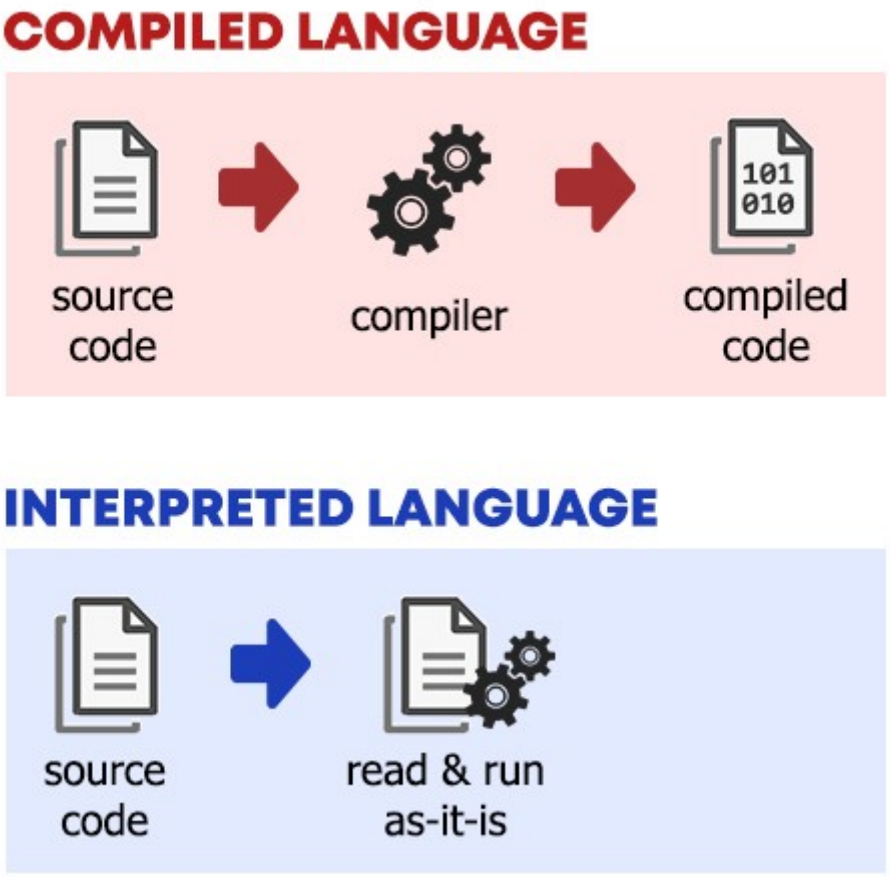
COMPATIBILITY CHECKS (WITH CANIUSE)

- Web Share

- CHAPTER Y -

WEB ASSEMBLY

COMPILED VS INTERPRETED



For you guys who somehow missed this out:

- Javascript is an interpreted language. The source code is read as-it-is and executed on-the-fly.
- But compiled languages (such as C, C++, Java) require the source code to be compiled into an executable first.

FREAKY JAVASCRIPT COLLAB

So you can pretty much guess what Web Assembly is all about. Letting Javascript run a piece of compiled code. Yes, it's kind of freaky, but compiled code does offer better overall performance.

INSTALLATION MAYHEM

To get started with Web Assembly, we have to go through a baptism by installation first.

PYTHON

Yes, Python is required by the tool we are going to use next. So install it if you have not already done so.

EMSCRIPTEN

The tool that we are going to use is called EMScripten. If you have Git installed, just create a folder and run `git clone https://github.com/emscripten-core/emsdk.git`. If not, you can also manually download it from Github.

EMSCRIPTEN TOOLCHAIN

Following up, we need to install the Toolchain.

```
FOR WINDOWS
cd WHERE/YOU/INSTALLED/EMSCRIPTEN
emsdk update
emsdk install latest
emsdk activate latest
emsdk_env.bat
emcmdprompt.bat
```

FOR LINUX/MAC

```
cd WHERE/YOU/INSTALLED/EMSCRIPTEN
./emsdk install latest
./emsdk activate latest
source ./emsdk_env.sh
```

P.S. If you get a "Python not found" error, the system path to Python is probably not properly set. Do your own research on how to fix.

HELLO WORLD IN C

CHAPTER-Y/2-HELLO-WORLD.C

```
#include <stdio.h>
int main(int argc, char ** argv) {
  printf("Hello world!\n");
}
```

Now that the installation nightmare is over, let us create a "Hello World" script. In C, that is.

COMPILE

Then, compile the script with `emcc 2-hello-world.c -s WASM=1 -o hello-world.html`.

- `-s WASM=1` specifies that we want a WASM output.
- `-o hello-world.html` saves the output to `hello-world.html`

LAUNCH

Lastly, just open `http://localhost/hello-world.html`. Well done, you have created your first WASM application.

LINKS & REFERENCES

Of course, this is only the first step into WASM. Read up more if you are interested:

- Web Assembly – MDN
- EMScripten
- WebAssembly.org

COMPATIBILITY CHECKS (WITH CANIUSE)

- Web Assembly

- CHAPTER Z -

JS MOBILE APP & PWA

NEXT GENERATION JAVASCRIPT

No joke, we have come to the last chapter, but there is still so much more that cannot be contained within this book. So I shall pave one last step toward the "next generation Javascript". That is, making Javascript work just like a native mobile app.

DEVELOPING MOBILE APPS WITH WEB TECHNOLOGY

Not going deep into this one, as this topic can be an entire book on its own. But it is possible to develop mobile apps using HTML, CSS, and Javascript. There are 2 "main schools" in this:

- Bridge or "convert" Javascript into a native app.
- The "mobile app" itself is just a browser running a "mini localhost".

I shall point to some projects worth checking out:

- Apache Cordova
- React Native
- Native Script
- Flutter
- Ionic

PROGRESSIVE WEB APPLICATION (PWA)

Apart from mobile apps, there is something called "PWA". Long story short, PWA is pretty much "an installable website that can function like a native app even when the user is offline".

As to the specific requirements of what makes a PWA, there's none. Yep, PWA is still a pretty loose term, but I will point to [3 main principles that Google pointed out – dev.to]

- **Capable** – The web app can perform all kinds of things. Live calls, store data, retrieve data, GPS maps, and whatever else funky under the sun.
- **Reliable** – Works offline. Has fallback for older browsers. Handles errors and graceful fails.
- **Installable** – The user can choose to install and create an icon on the home screen. This makes the PWA run in a standalone manner, much like a native app.

Basically, a PWA is everything covered in this book – Service workers, offline caching, sync, indexed databases, WebRTC, Web Sockets, GPS, etc… Except that we are missing on the "installable" part. Let's get into a simple example next.

BAREBONES PWA

THE HTML

CHAPTER-Z/1-INDEX.HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- (A) TITLE + CHARSET + DESCRIPTION -->
    <title>Barebones PWA</title>
    <meta charset="utf-8">
    <meta name="description" content="Barebones PWA page">
```

```
  <!-- (B) VIEWPORT (ALLOW ZOOM IN, NOT OUT) -->
  <meta name="viewport" content="width=device-width,
  initial-scale=1.0, maximum-scale=1.5">

  <!-- (C) WEB APP MANIFEST -->
  <!-- https://web.dev/add-manifest/ -->
  <link rel="manifest" href="4-manifest.json">

  <!-- (D) GOOD OLD FAVICON -->
  <link rel="icon" href="favicon.png" type="image/png">

  <!-- (E) ANDROID/CHROME -->
  <meta name="mobile-web-app-capable" content="yes">
  <meta name="theme-color" content="white">

  <!-- (F) IOS APP ICON + MOBILE SAFARI -->
  <link rel="apple-touch-icon" href="icon-512.png">
  <meta name="apple-mobile-web-app-capable"
  content="yes">
  <meta name="apple-mobile-web-app-status-bar-style"
  content="black">
  <meta name="apple-mobile-web-app-title" content="Hello
  World">

  <!-- (G) WINDOWS -->
  <meta name="msapplication-TileImage" content="icon-
  512.png">
  <meta name="msapplication-TileColor"
  content="#ffffff">

  <!-- (H) STYLESHEET + JAVASCRIPT -->
  <link rel="stylesheet" href="2-style.css">
  <script defer src="3a-main.js"></script>
</head>
<body>
  It Works!
</body>
```

```
</html>
```

As you can see, a "PWA website" is still the same old HTML. Except that it has a ton of meta tags.

- **(A)** The usual title, character set, description.
- **(B)** Viewport. AKA "pinch zoom control".
- **(C)** The web app manifest, very important for PWA. This is nothing but a JSON file.
- **(D To G)** Irritating. Everyone has their own set of meta tags to define the icon and browser-specific color theme. Not going too deep into that, just keep things simple – Serve 64X64 for favicon, give the rest a huge 512X512 icon.
- **(H)** The usual CSS and Javascript.

THE CSS

CHAPTER-Z/2-STYLE.CSS

```
* { font-family: arial, sans-serif; }
body {
  background: #000;
  color: #fff;
}
```

Just some dummy styles.

JAVASCRIPT – MAIN PAGE

CHAPTER-Z/3A-MAIN.JS

```
if ("serviceWorker" in navigator) {
  navigator.serviceWorker.register("3b-sw.js");
}
```

Looks familiar? Yes, register a service worker.

JAVASCRIPT – SERVICE WORKER CACHE

CHAPTER-Z/3A-MAIN.JS

```
// (A) FILES TO CACHE
const cName = "demo-pwa",
cFiles = [
  "1-index.html",
  "2-style.css",
  "3a-main.js"
];

// (B) CREATE/INSTALL CACHE
self.addEventListener("install", (evt) => {
  evt.waitUntil(
    caches.open(cName)
    .then((cache) => { return cache.addAll(cFiles); })
    .catch((err) => { console.error(err) })
  );
});

// (C) LOAD FROM CACHE, FALLBACK TO NETWORK IF NOT FOUND
self.addEventListener("fetch", (evt) => {
  evt.respondWith(
    caches.match(evt.request)
    .then((res) => { return res || fetch(evt.request); })
```

```
  );
});
```

Looks familiar again? Yes, offline cache storage.

WEB APP MANIFEST

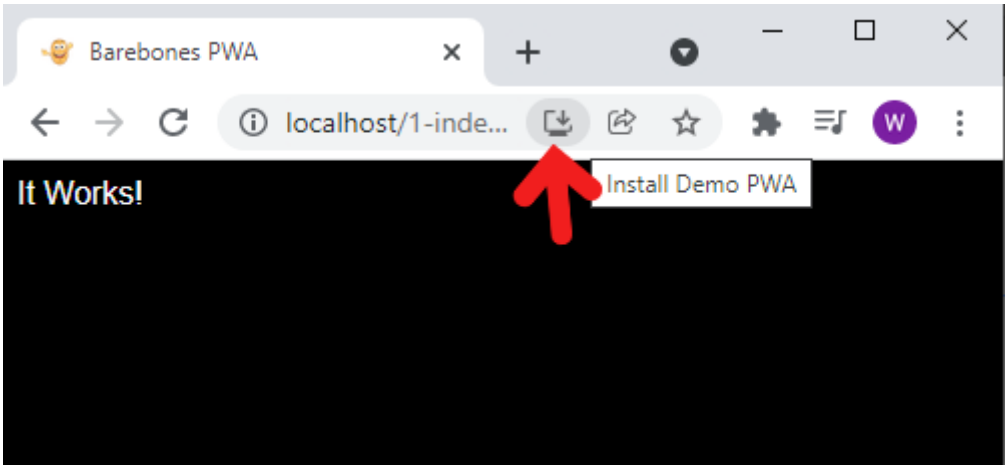
CHAPTER-Z/4-MANIFEST.JSON

```
{
  "short_name": "DPWA",
  "name": "Demo PWA",
  "icons": [{
    "src": "favicon.png",
    "sizes": "64x64",
    "type": "image/png"
  }, {
    "src": "icon-512.png",
    "sizes": "512x512",
    "type": "image/png"
  }],
  "start_url": "1-index.html",
  "scope": "/",
  "background_color": "white",
  "theme_color": "white",
  "display": "standalone"
}
```

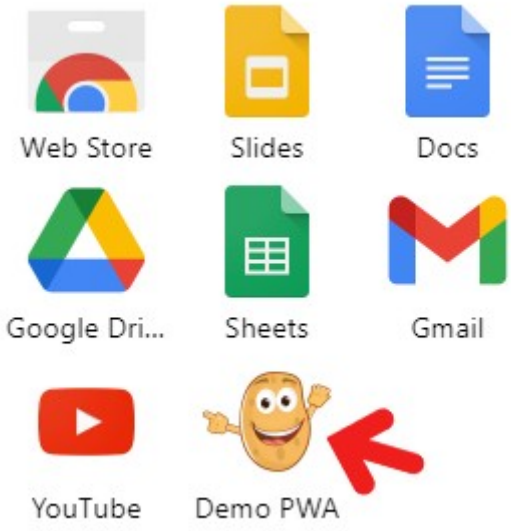
The manifest file pretty much describes your web app, the icons, and all the settings stuff. See this post for the full list of manifest properties.

LAUNCH!

Simply access `http://localhost/1-index.html` in your browser, there should be an icon to install the PWA.



Go ahead. This will create an icon on your desktop/home screen, and also the apps page in your browser. Open `chrome://apps OR edge://apps OR opera://apps`.



To remove the app, you have to do it in the browser. Deleting the icon on the desktop/home screen will not remove it.

MANUAL INSTALL

CHAPTER-Z/5-MANUAL-INSTALL.HTML

```
<!-- (A) INSTALL PWA BUTTON -->
<input type="button" id="iBtn" style="display:none"
       value="Add To Home Screen"/>

<script>
// (B) PRE-INSTALL
let iBtn = document.getElementById("iBtn"), // HTML BUTTON
    iPrompt; // TO HOLD THE "INSTALL APP" PROMPT

// (C) LISTEN FOR "BEFORE INSTALL PROMPT" EVENT
window.addEventListener("beforeinstallprompt", (evt) => {
  // (C1) STOP DEFAULT "INSTALL APP" PROMPT
  evt.preventDefault();

  // (C2) "STORE" THE "INSTALL APP" PROMPT
  iPrompt = evt;

  // (C3) SHOW "INSTALL APP" ON CLICK
  iBtn.addEventListener("click", () => {
    iPrompt.prompt();
    iPrompt.userChoice.then((res) => {
      if (res.outcome === "accepted") {
        console.log("ACCEPTED INSTALL");
      } else {
        console.log("DISMISSED INSTALL");
      }
      iBtn.remove();
      iPrompt = null;
    });
  });
  iBtn.style.display = "block";
});
</script>
```

If you want to create your own "manual install button" or "nice-looking install app offer":

- **(A)** Create your HTML. For this example, we will just use a simple HTML button – Hidden by default.
- **(B)** In the Javascript, we put the HTML button into `iBtn` and create an empty `iPrompt` to hold the "install prompt" event.
- **(C)** If the browser deems your PWA as "installable", the `beforeinstallprompt` event will fire.
  - **(C1 & C2)** We "hijack and precent" the default behavior to show an installation prompt, and "store" it into `iPrompt`.
  - **(C3)** Self-explanatory, we show the "install app" button, and the user has to click on it to show the install prompt.

STARTED AS STANDALONE APP OR ACCESS FROM WEB?

CHAPTER-Z/5-MANUAL-INSTALL.HTML

```
// (D) DETECT STANDALONE PWA OR ACCESSED FROM WEB
const isStandalone = window.matchMedia("(display-mode:
standalone)").matches || navigator.standalone;
console.log(isStandalone ? "Standalone APP" : "Web");
```

For this final little bit, if you need to check if the user has started as a PWA (by clicking on the icon) or accessed from the Internet – Just do a quick check with `window.matchMedia()` and `navigator.standalone`.

LINKS & REFERENCES

- Progressive Web App – MDN
- Progressive Web App – web.dev

COMPATIBILITY CHECKS (WITH CANIUSE)

- Add To Home Screen

- CHAPTER α -

THANK YOU

BUGS SPOTTED?!

I have done some due diligence of running through the code after each chapter to make sure that it at least works. But sadly, I don't have a team of editors… So my apologies if there are still some bugs scattered around. Kindly drop a note on Code Boxx and let me know… If you want to. I will revise that in future editions of this book.

code-boxx@mail.com

DISCOUNT COUPON

A big thank you for getting this eBook! Should you be interested in getting more goodies from the **Code Boxx Store**, please use the following promo code at to get a 10% off –

THANKYOU4896