Shared web workers -- A new way to manage token refresh requests

A crazy idea brought to you by
Camille Kaniecki, SE III at ACV Auctions

Forward

Welcome!

This session brought you you by a conference newbie. Constructive feedback and questions following this presentation welcome! Please lower your expectations and resign yourself to some blatherings from a madwoman.

This presentation also assumes that you have some basic knowledge of JavaScript, Vue.js, Vuex, and token refresh cycles.

Enjoy!

Definition time!

- oAuth
  - Open standard authorization protocol
  - Uses authorization tokens as proof of identity rather than username/password
  - Uses JSON for payload data
  - Uses API calls rather than session cookies. Better for short, relatively infrequent transitions than something which needs constant access (i.e. medical system, regulatory technologies or mission-critical services)
- Authorization
  - Access rights or grants of an authenticated user or service
  - Different from authentication
- Authentication
  - Verifying the identity of a user or service

A Brief Introduction to Token Refresh Cycles

More definitions! (sorry)

- Token
  - Pieces of data (JSON) in this case, which, when decrypted, provide information about authorization grants for a particular user or service.
  - Multiple types involved for oauth:
    - JWT tokens
      - “JSON Web Token token”, but JWT is just…missing something
      - “Bearer” token (anyone who “holds” the token can use it). (think RFID badge)
      - How prevent you giving your ID badge to a friend? Have them be short-lived.
    - Refresh tokens
      - Think travel visa. (as long as your visa is still valid, you can get a new badge)
      - Do not carry any sensitive information

SPAs and Token Refresh Cycles

(Definitions again?? Yes.)

- SPA
  - Single-page application
- Refresh Token Rotation
  - Need a refresh token to get a new JWT
  - When you request a new JWT, you also receive a new refresh token
  - Previous refresh token becomes invalidated immediately

SPAs and Token Refresh Cycles

Need to strike a balance between security and user experience. We need to ask users to authenticate themselves enough times that we can still verify their identity, but not so many times that the user becomes frustrated or the service hits volume limits.

In the case of SPAs, a common option is to call a token refresh endpoint to get a new refresh token on a set interval. Exchanging the refresh token on this interval invalidates the old one in the off chance it becomes compromised. Each refresh token is then used to renew JWTs as well, to ensure continued authorization of any necessary api endpoints.

Sounds simple!

...but not quite.

Hard to avoid token refresh requests without stale data, or ending up with race conditions on you token refresh endpoint between different api calls, especially in multiple browser tabs.

There are many ways to solve this problem, including (but not limited to):

- A custom token refresh locking mechanism
- “Free-for-all” requests with graceful failing of unauthorized calls
- Silent authentication
- Utilization of background threading (web workers)

The Proposition!

This presentation aims to persuade you to give Shared Web Workers a try for managing token refresh cycles on a timed interval.

What is a web worker?

A web worker is any browser-provided service which allows scripting to be run in a background thread (client side). JavaScript is single-threaded, and this was by design to coordinate with the browser's main rendering loop.

Now supported by nearly every web browser.

Adoption of web workers has been slow…no one seems to know why 🤷🏻‍♀️

Time to break the cycle!

Couple of Caveats

1. For a web worker to collaborate with a web application, they must all be within the same domain (CORS same-origin policy).
2. Web workers are unable to access DOM elements or the window object.
3. Shared web workers (covered next) are not supported by many mobile web browsers. Arguably, this is not much of a detriment to adoption because most mobile application usage does not rely on multiple browser tabs.

Types of Web Workers

There are mainly two kinds:

1. Web worker
    a. Can only be accessed by the script which created it
2. Shared web worker
    a. Can be accessed by any script in any browser tab (from the same domain)

(We are disregarding service worker and others from the scope of this presentation)

The use case

Shared web workers have the potential to serve as the perfect vehicle to manage token refresh cycles for SPAs since they can “listen” to multiple open tabs simultaneously.

In fact, shared web workers are accessible to not only the browser application, but also other applications and even other web workers themselves.

Under the hood

The mechanism by which shared web workers communicate with these objects is via the browser MessagePort object.

MessagePorts are an interface for the Channel Messaging API and come equipped with some basic functionality for handling events, such as connecting, sending messages, handling errors, and closing connections.

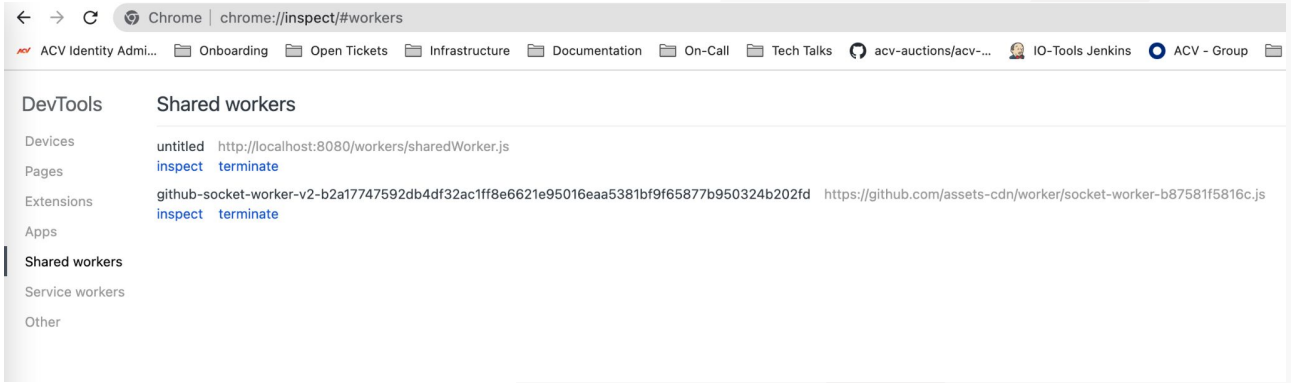
You'll see this in action in the examples to follow.

Prove it!

How can I inspect a web worker? Depends on the browser and the type of worker.

In Chrome, you can view any running web workers by opening up a new tab and navigating to `chrome://inspect/#workers` in the address bar.

A new window will appear with your web workers, as well as a separate console available for logging which is specific to your workers.



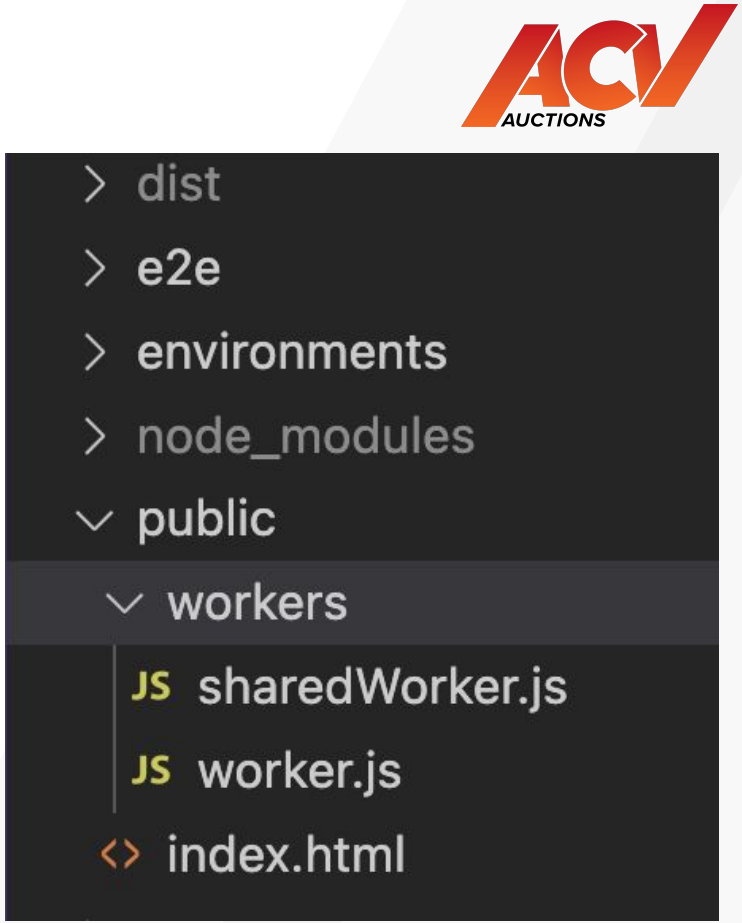
Implementation

Rather than give you a full working example with a brand-new SPA, since we don't have a readily available authorization service, we are instead going to focus on a more generic approach and walk you through the universal components of the implementation which you can then utilize in multiple frameworks of your choosing.

In our example, we will use components of:

- Vue.js (Vue2)
- Webpack
- Vuex

Components, Part 1



Worker Initialization Files

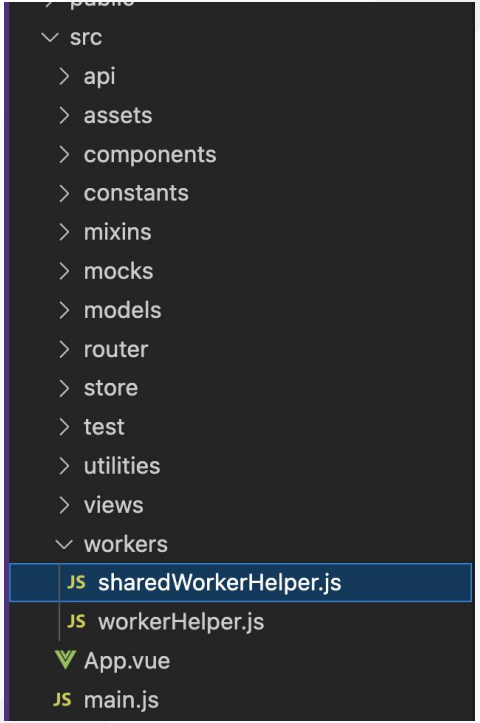
These are the files which call upon the browser's MessagePort Object

Since these will run within the shared web worker, they need to be outside of our src folder (hence they are in our /public folder, which will be copied into our /dist folder via Webpack)

Components: Part 2

Helper files (*custom utilities scripts*)

These are the two files used to manage connections to the web workers. Since these will actually need to interact with our main application thread, we include this in our src folder.



Components: Part 3

In App.js, we'll have a few lines of code to actually initialize our web workers as well as our token refresh interval.

We will cover this briefly and focus on the important bits, as each application will be different and have different utilization of `App.vue`.

Let’s break this down...

We’re going to look at some lines of code outside of their use case context.

For more information and better formatting (aka not slides!), refer to this article: https://acv.engineering/posts/managing-refresh-tokens-with-a-shared-web-worker/

Keep in mind:

1. Your individual use case may be different with regards to token refresh cycles
2. This is not a fully bootstrapped example, but will instead aim to show you the components’ potentiality that you can then appropriate and adapt to fit the needs of your application

/public/sharedWorker.js

```
let timers = [];

const alreadyHaveTimerType = (dataType, obj) => {

  return obj.name === dataType;

};

onconnect = (ev) => {

  const [port] = ev.ports;

  port.onmessage = (MessageEvent) => {

    const { eventName, data = {} } = MessageEvent.data;

        if (eventName === 'setInterval') {
```

Firstly, we initialize an array of timers.

This step is not necessary, but it makes the code more reusable in that you can can simply add or remove as many different timers as you like with the same script.

For example, you can use one timer for managing token refresh cycles, and another for polling on changes for some other data set, all with the same shared web worker.

Next, we include a method (`alreadyHaveTimerType`) which will be used to make sure we don't end up having concurrent timers performing the same function.

/public/sharedWorker.js

```
let timers = [];

const alreadyHaveTimerType = (dataType, obj) => {

  return obj.name === dataType;

};

onconnect = (ev) => {

  const [port] = ev.ports;

  port.onmessage = (MessageEvent) => {

    const { eventName, data = {} } = MessageEvent.data;

        if (eventName === 'setInterval') {
```

The next bit of code is mostly standard re: web browsers as it will be connecting to (and listening for messages on) the browser's MessagePort object.

Once we set up a port connection and start receiving events, then we can customize how our application interacts with them.

/public/sharedWorker.js

```
if (eventName === 'setInterval') {

      const dataTypeObjTimersExist =
timers.some(alreadyHaveTimerType.bind(null, data.type));

      if (dataTypeObjTimersExist) {
        const prunedTimers = timers.filter(timerObj => {
          if (timerObj.name === data.type) {
            clearInterval(timerObj.timer);
          }
          return timerObj.name !== data.type;
        });
        timers = prunedTimers;
      }


      const intervalObj = setInterval(() => {
        port.postMessage({
          event: eventName,
          Data,
        });
      }, data.interval);


      const newTimerObj = {
        name: data.type,
        timer: intervalObj,
        appUuid: data.appUuid,
      };
      timers.push(newTimerObj);
    }
```

When the MessagePort detects a `setInterval` event, we firstly want to check whether a timer of that type already exists, as we only ever want one timer of each type for this application.

If there is already a timer of that same type, we remove it, create a new interval object, and then add it to our timer array.

Not shown here, we also detect if a `clearAllTimers` event is fired, which empties the `timers` array.

/src/workers/sharedWorkerHerlper.js

```
export default class SharedWorkerHelper {
  constructor() {

    if (window.SharedWorker) {
      this.worker = new SharedWorker('/workers/sharedWorker.js');
      this.events = {};
      this.worker.port.start();

      this.worker.port.addEventListener(
        'Message',
        (e) => this.callEventCallback(e.data),
        False,
      );

      this.worker.port.addEventListener(
        'Error',
        () => this.callEventCallback('error'),
        False,
      );
    }
  }
```

Firstly, we check to make sure our browser actually supports shared web workers.

If we do, we start one up and then open up a browser MessagePort Object.

Then we add some event listeners. This is all pretty standard as far as web workers go.

/src/workers/sharedWorkerHerlper.js

```
trigger(eventName, data) {
    this.worker.port.postMessage({
      eventName,
      data,
    });
  }

  clearAllTimers() {
    this.worker.port.postMessage({
      eventName: 'clearAllTimers',
    });
  }

  terminate() {
    this.clearAllTimers();
    this.worker.port.postMessage({
      type: 'cmd',
      action: 'die',
    });
  }

 on(eventName, callback) {
    this.events[eventName] = callback;
  }

  callEventCallback({ event, data }) {
    if (Object.prototype.hasOwnProperty.call(this.events, event)) {
      this.events[event].call(null, data);
    }
  }
}
```

The rest is all pretty standard as far as event listening and callbacks are concerned, so for the sake of time, we are going to gloss over these but leave the code in the slides for you go to back and review after this presentation

App.vue

After importing our SharedWorkerHelper.js file, we'll define 3 pieces of state in our `data()` method:

1. A `uuid` for this application instance (initialized to `null`)
2. A worker object (also initialized to `null`)
3. A refresh interval (in milliseconds) for how long we want each token refresh cycle to be

```
],
data() {
  return {
    currentRefreshInterval: 300000, // 5 mins
    uuid: null,
    worker: null, // for token refresh
  };
},
```

App.vue

We'll also add a few computed properties to verify whether or not our state has an active shared web worker and supports them:

```
haveActiveWorker() {
  return !!this.worker;
},
supportsSharedWorkers() {
  return !!window.SharedWorker;
},
supportsWebWorkers() {
  return !!window.Worker;
},
```

App.vue

Now comes the fun part!

In the `created()` Vue lifecycle hook, we'll use the npm uuid package to generate a unique identifier for this instance of our application.

Then, we check for previously-existing web workers, and 🔪 them. Then we create a new worker.

```
try {
  this.handleWorkerEnd();
} catch (e) {
  // do nothing, we have no straggling workers
}
if (!this.worker) {
  if (this.supportsSharedWorkers) {
    this.worker = new SharedWorkerHelper(this.uuid);
  } else if (this.supportsWebWorkers) {
    this.worker = new WorkerHelper(this.uuid);
  } else {
    console.log('web workers not supported in this browser');
  }
}
```

App.vue

Next, we'll make sure that we have a refresh interval defined and that we are not on a page which does not require authentication (such as the login page), before actually firing up our shared web worker.

```
if (this.currentRefreshInterval > 0 && !this.isAuthenticationView) {
  this.setupWorkerActions();
  this.handleWorkerStart();
}
```

App.vue

```
async handleWorkerStart() {
  await this.createWorker();
  await this.setupWorkerActions();
  this.worker.on('setInterval', () => {
    this.handleWorkerUpdate('refreshTokens');
  });
},
handleWorkerUpdate(type) {
  switch (type) {
    case 'refreshTokens':
      this.refreshTokens(this.currentRefreshToken);
      if (this.authErrors.length > 0) {
        this.handleWorkerEnd();
        this.$router.push('/loggingyouout');
      }
      break;
    default:
      break;
  }
},
handleWorkerEnd() {
  if (this.worker) {
    this.worker.terminate();
    this.worker = null;
  }
},
setupWorkerActions() {
  this.worker.trigger('setInterval', {
    interval: this.currentRefreshInterval,
    type: 'refreshTokens',
    appUuid: this.uuid,
  });
},
```

We also need to define a few methods to manage our worker's lifecycle.

Note: `this.authErrors` and `this.currentRefreshTokens` are both Vuex getters in our authorization module, as is the action `this.refreshTokens`.

Right now, handleWorkerUpdate is generic, and has only one case, but is built to be expanded as needs dictate.

Why do this?

The Why of the Web Worker

Using the shared web worker lets us:

1. Keep the logic (and state data) for the token refresh within the application itself, avoiding unnecessary data exposure
2. Prevents a race condition by having each open instance or tab of our application supercede control from any previously-running timers
3. Maintain a more constant timer outside of the main application loop (mitigating delays from heavy processing tasks in the main thread)

Here's an illustration of this process…

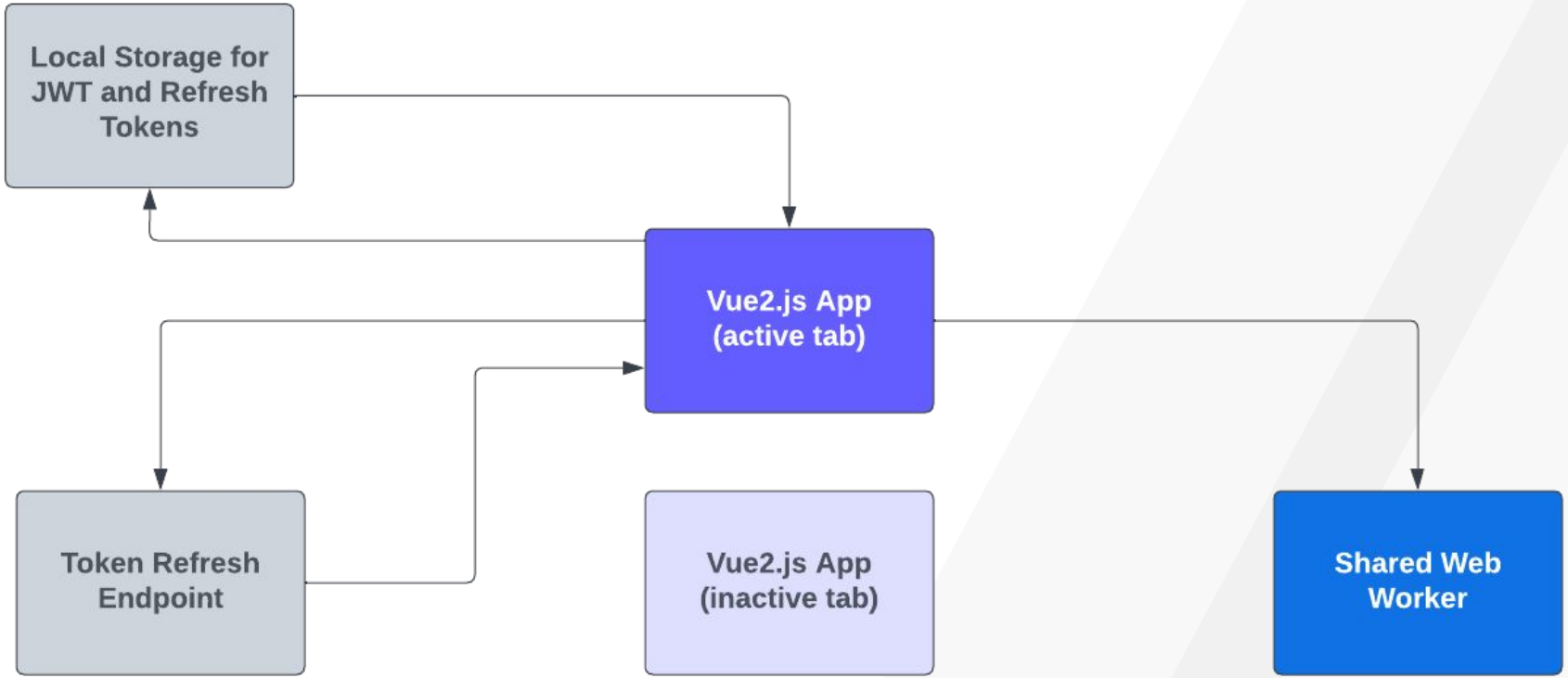
The Why of the Web Worker

Let’s suppose our application is open in two browser tabs.

The “active” tab is the one which has initialized the web worker and already sent over its `uuid` and timer request.

The shared web worker is managing the **timer** for the `refreshToken` method within the application, but the **application itself** is pulling the token information from local storage as well as calling the token refresh endpoint.

The Why of the Web Worker



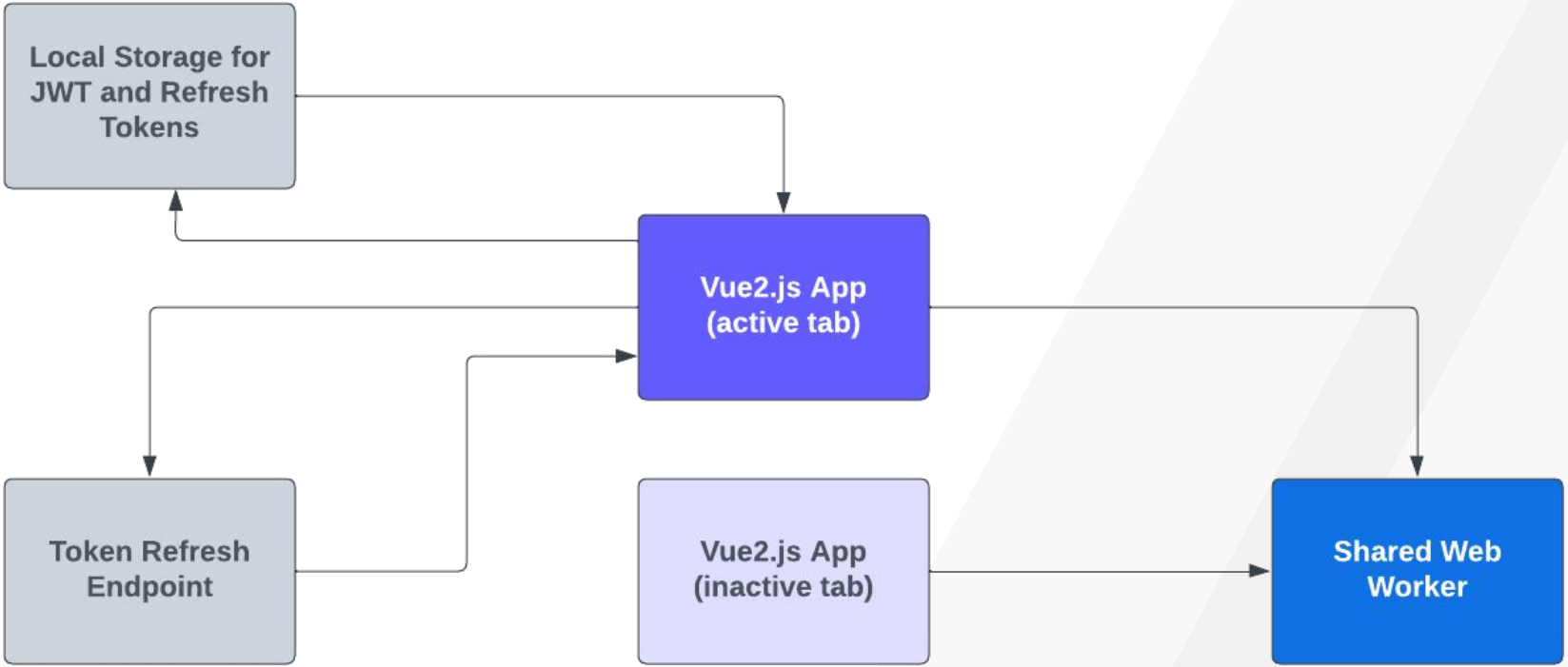
The Why of the Web Worker

Then let’s suppose our user switches to the second tab.

This new tab tries to make a timer request to the shared web worker with its unique uuid. The shared web worker realized this is a different timer of the same instance, and creates a new timer, overriding the old one.

This means the active tab then takes over making the calls to the token refresh endpoint as well as managing the JWT and refresh tokens within local storage.

The Why of the Web Worker



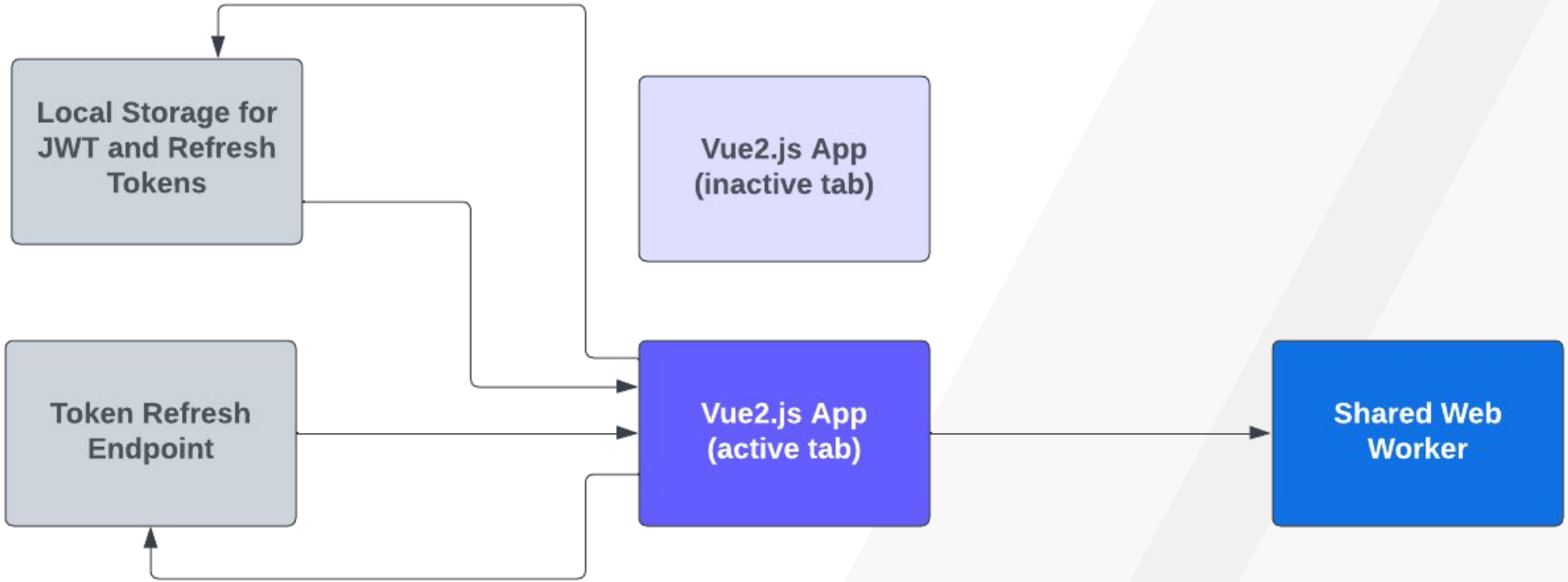
The Why of the Web Worker

The current active tab then takes over making the calls to the token refresh endpoint as well as managing the JWT and refresh tokens within local storage.

Note: There are many ways to save application token data other than local storage. In this example, the important part is that it is stored somewhere outside of state in a location reachable by all open application tabs to preserve data integrity and mitigate stale tokens.

The Why of the Web Worker

The current active tab then takes over making the calls to the token refresh endpoint as well as managing the JWT and refresh tokens within local storage.



The Why of the Web Worker

This process is repeated every time a new tab is opened for this application (within the same domain).

This process prevents a race condition when calling the `tokenRefresh` endpoint, as well as prevents the endpoint from being called with stale token information.

Additionally, the background threading of the shared web worker means that there are no gaps or delays in processing, as the original thread running the application does not need to worry about finishing any calculations before the timer event is fired to the port (it only needs to listen on the shared MessagePort object).

One last note...

In our repo with code samples, we fall back to regular web workers when shared web workers are not supported.

See repo linked at the end of this presentation for the code.

You may not actually even need token refresh cycles on an interval

But that is a story for another day…

Thank you for attending!!

More resources below:

Tech blog article on this topic:
https://acv.engineering/posts/managing-refresh-tokens-with-a-shared-web-worker/

Code snippets:
https://github.com/camillekaniecki/connect-tech-2022-shared-web-workers