BroadcastChannel API: A Message Bus for the Web

By Eric Bidelman
Engineer @ Google working on web tooling: Headless Chrome, Puppeteer, Lighthouse

The BroadcastChannel API allows same-origin scripts to send messages to other browsing contexts. It can be thought of as a simple message bus that allows pub/sub semantics between windows/tabs, iframes, web workers, and service workers.

API basics

The Broadcast Channel API is a simple API that makes communicating between browsing contexts easier. That is, communicating between windows/tabs, iframes, web workers, and service workers. Messages which are posted to a given channel are delivered to all listeners of that channel.

The `BroadcastChannel` constructor takes a single parameter: the name of a channel. The name identifies the channel and lives across browsing contexts.

```
// Connect to the channel named "my_bus".
const channel = new BroadcastChannel('my_bus');

// Send a message on "my_bus".
channel.postMessage('This is a test message.');

// Listen for messages on "my_bus".
channel.onmessage = function(e) {
  console.log('Received', e.data);
};

// Close the channel when you're done.
channel.close();
```

Sending messages

Messages can be strings or anything supported by the structured clone algorithm (Strings, Objects, Arrays, Blobs, ArrayBuffer, Map).

Example - sending a Blob or File

```
channel.postMessage(new Blob(['foo', 'bar'], {type: 'plain/text'}));
```

A channel won't broadcast to itself. So if you have an **`onmessage`** listener on the same page as a **`postMessage()`** to the same channle, that **`message`** event doesn't fire.

Differences with other techniques

At this point you might be wondering how this relates to other techniques for message passing like WebSockets, SharedWorkers, the **`MessageChannel`** API, and **`window.postMessage()`**. The Broadcast Channel API doesn't replace these APIs. Each serves a purpose. The Broadcast Channel API is meant for easy one-to-many communication between scripts **on the same origin**.

Some use cases for broadcast channels:

- Detect user actions in other tabs
- Know when a user logs into an account in another window/tab.
- Instruct a worker to perform some background work
- Know when a service is done performing some action.
- When the user uploads a photo in one window, pass it around to other open pages.

Example - page that knows when the user logs out, even from another open tab on the same site:

```
<button id="logout">Logout</button>

<script>
function doLogout() {
  // update the UI login state for this page.
}

const authChannel = new BroadcastChannel('auth');

const button = document.querySelector('#logout');
button.addEventListener('click', e => {
  // A channel won't broadcast to itself so we invoke doLogout()
  // manually on this page.
  doLogout();
  authChannel.postMessage({cmd: 'logout', user: 'Eric Bidelman'});
});
```

```
authChannel.onmessage = function(e) {
  if (e.data.cmd === 'logout') {
    doLogout();
  }
};
</script>
```

In another example, let's say you wanted to instruct a service worker to remove cached content after the user changes their "offline storage setting" in your app. You could delete their caches using `window.caches`, but the service worker may already contain a utility to do this. We can use the Broadcast Channel API to reuse that code! Without the Broadcast Channel API, you'd have to loop over the results of `self.clients.matchAll()` and call `postMessage()` on each client in order to achieve the communication from a service worker to all of its clients (actual code that does that). Using a Broadcast Channel makes this `O(1)` instead of `O(N)`.

Example - instruct a service worker to remove a cache, reusing its internal utility methods.

```
// In index.html

const channel = new BroadcastChannel('app-channel');
channel.onmessage = function(e) {
  if (e.data.action === 'clearcache') {
    console.log('Cache removed:', e.data.removed);
  }
};

const messageChannel = new MessageChannel();

// Send the service worker a message to clear the cache.
// We can't use a BroadcastChannel for this because the
// service worker may need to be woken up. MessageChannels do that.
navigator.serviceWorker.controller.postMessage({
  action: 'clearcache',
  cacheName: 'v1-cache'
}, [messageChannel.port2]);

// In sw.js

function nukeCache(cacheName) {
  return caches.delete(cacheName).then(removed => {
    // ...do more stuff (internal) to this service worker...
    return removed;
  });
}
```

```
self.onmessage = function(e) {
  const action = e.data.action;
  const cacheName = e.data.cacheName;

  if (action === 'clearcache') {
    nukeCache(cacheName).then(removed => {
      // Send the main page a response via the BroadcastChannel API.
      // We could also use e.ports[0].postMessage(), but the benefit
      // of responding with the BroadcastChannel API is that other
      // subscribers may be listening.
      const channel = new BroadcastChannel('app-channel');
      channel.postMessage({action, removed});
    });
  }
};
```

Difference with `postMessage()`

Unlike **`postMessage()`**, you no longer have to maintain a reference to an iframe or worker in order to communicate with it:

```
// Don't have to save references to window objects.
const popup = window.open('https://another-origin.com', ...);
popup.postMessage('Sup popup!', 'https://another-origin.com');
```

`window.postMessage()` also allows you to communicate across origins. **The Broadcast Channel API is same-origin**. Since messages are guaranteed to come from the same origin, there's no need to validate them like we used to with **`window.postMessage()`**:

```
// Don't have to validate the origin of a message.
const iframe = document.querySelector('iframe');
iframe.contentWindow.onmessage = function(e) {
  if (e.origin !== 'https://expected-origin.com') {
    return;
  }
  e.source.postMessage('Ack!', e.origin);
};
```

Simply "subscribe" to particular channel and have secure, bidirectional communication!

Difference with SharedWorkers

Use `BroadcastChannel` for simple cases where you need to send message to potentially several windows/tabs, or workers.

For fancier use cases like managing locks, shared state, synchronizing resources between a server and multiple clients, or sharing a WebSocket connection with a remote host, shared workers are the most appropriate solution.

Difference with MessageChannel API

The main difference between the Channel Messaging API and `BroadcastChannel` is that the latter is a means to dispatch messages to multiple listeners (one-to-many). `MessageChannel` is meant for one-to-one communication directly between scripts. It's also more involved, requiring you to setup channels with a port on each end.

Feature detection and browser support

Currently, Chrome 54, Firefox 38, and Opera 41 support the Broadcast Channel API.

```
if ('BroadcastChannel' in self) {
  // BroadcastChannel API supported!
}
```

As for polyfills, there are a few out there:

- https://gist.github.com/alexis89x/041a8e20a9193f3c47fb
- https://gist.github.com/inexorabletash/52f437d1451d12145264

I haven't tried these, so your mileage may vary.

Resources

- Chromestatus.com entry
- Caniuse.com entry
- MDN Hacks - "BroadcastChannel API in Firefox 38"

Last updated July 2, 2018.