20th December 2019

Cross-window communication

The “Same Origin” (same site) policy limits access of windows and frames to each other.

The idea is that if a user has two pages open: one from `john-smith.com`, and another one is `gmail.com`, then they wouldn’t want a script from `john-smith.com` to read our mail from `gmail.com`. So, the purpose of the “Same Origin” policy is to protect users from information theft.

Same Origin

Two URLs are said to have the “same origin” if they have the same protocol, domain and port.

These URLs all share the same origin:

- `http://site.com`
- `http://site.com/`
- `http://site.com/my/page.html`

These ones do not:

- `http://www.site.com` (another domain: `www.` matters)
- `http://site.org` (another domain: `.org` matters)
- `https://site.com` (another protocol: `https`)
- `http://site.com:8080` (another port: `8080`)

The “Same Origin” policy states that:

- if we have a reference to another window, e.g. a popup created by `window.open` or a window inside `<iframe>`, and that window comes from the same origin, then we have full access to that window.
- otherwise, if it comes from another origin, then we can’t access the content of that window: variables, document, anything. The only exception is `location`: we can change it (thus redirecting the user). But we cannot *read* location (so we can’t see where the user is now, no information leak).

In action: iframe

An `<iframe>` tag hosts a separate embedded window, with its own separate `document` and `window` objects.

We can access them using properties:

- `iframe.contentWindow` to get the window inside the `<iframe>`.
- `iframe.contentDocument` to get the document inside the `<iframe>`, shorthand for `iframe.contentWindow.document`.

When we access something inside the embedded window, the browser checks if the iframe has the same origin. If that's not so then the access is denied (writing to `location` is an exception, it's still permitted).

For instance, let's try reading and writing to `<iframe>` from another origin:

```
<iframe src="https://example.com" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // we can get the reference to the inner window
    let iframeWindow = iframe.contentWindow; // OK
    try {
      // ...but not to the document inside it
      let doc = iframe.contentDocument; // ERROR
    } catch(e) {
      alert(e); // Security Error (another origin)
    }

    // also we can't READ the URL of the page in iframe
    try {
      // Can't read URL from the Location object
      let href = iframe.contentWindow.location.href; // ERROR
    } catch(e) {
      alert(e); // Security Error
    }

    // ...we can WRITE into location (and thus load something else into the i
    iframe.contentWindow.location = '/'; // OK

    iframe.onload = null; // clear the handler, not to run it after the locat
  };
</script>
```

The code above shows errors for any operations except:

- Getting the reference to the inner window `iframe.contentWindow` – that's allowed.
- Writing to `location`.

Contrary to that, if the `<iframe>` has the same origin, we can do anything with it:

```
<!-- iframe from the same site -->
<iframe src="/" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // just do anything
    iframe.contentDocument.body.prepend("Hello, world!");
  };
</script>
```

`iframe.onload` vs `iframe.contentWindow.onload`

The `iframe.onload` event (on the `<iframe>` tag) is essentially the same as `iframe.contentWindow.onload` (on the embedded window object). It triggers when the embedded window fully loads with all resources.

...But we can't access `iframe.contentWindow.onload` for an iframe from another origin, so using `iframe.onload`.

Windows on subdomains: document.domain

By definition, two URLs with different domains have different origins.

But if windows share the same second-level domain, for instance `john.site.com`, `peter.site.com` and `site.com` (so that their common second-level domain is `site.com`), we can make the browser ignore that difference, so that they can be treated as coming from the "same origin" for the purposes of cross-window communication.

To make it work, each such window should run the code:

```
document.domain = 'site.com';
```

That's all. Now they can interact without limitations. Again, that's only possible for pages with the same second-level domain.

Iframe: wrong document pitfall

When an iframe comes from the same origin, and we may access its `document`, there's a pitfall. It's not related to cross-origin things, but important to know.

Upon its creation an iframe immediately has a document. But that document is different from the one that loads into it!

So if we do something with the document immediately, that will probably be lost.

Here, look:

```
<iframe src="/" id="iframe"></iframe>

<script>
  let oldDoc = iframe.contentDocument;
  iframe.onload = function() {
    let newDoc = iframe.contentDocument;
    // the loaded document is not the same as initial!
    alert(oldDoc == newDoc); // false
  };
</script>
```

We shouldn't work with the document of a not-yet-loaded iframe, because that's the *wrong document*. If we set any event handlers on it, they will be ignored.

How to detect the moment when the document is there?

The right document is definitely at place when `iframe.onload` triggers. But it only triggers when the whole iframe with all resources is loaded.

We can try to catch the moment earlier using checks in `setInterval`:

```
<iframe src="/" id="iframe"></iframe>

<script>
  let oldDoc = iframe.contentDocument;

  // every 100 ms check if the document is the new one
  let timer = setInterval(() => {
    let newDoc = iframe.contentDocument;
    if (newDoc == oldDoc) return;

    alert("New document is here!");

    clearInterval(timer); // cancel setInterval, don't need it any more
  }, 100);
</script>
```

Collection: window.frames

An alternative way to get a window object for `<iframe>` – is to get it from the named collection `window.frames`:

- By number: `window.frames[0]` – the window object for the first frame in the document.
- By name: `window.frames.iframeName` – the window object for the frame with `name="iframeName"`.

For instance:

```
<iframe src="/" style="height:80px" name="win" id="iframe"></iframe>

<script>
  alert(iframe.contentWindow == frames[0]); // true
  alert(iframe.contentWindow == frames.win); // true
</script>
```

An iframe may have other iframes inside. The corresponding `window` objects form a hierarchy.

Navigation links are:

- `window.frames` – the collection of "children" windows (for nested frames).
- `window.parent` – the reference to the "parent" (outer) window.
- `window.top` – the reference to the topmost parent window.

For instance:

```
window.frames[0].parent === window; // true
```

We can use the `top` property to check if the current document is open inside a frame or not:

```
if (window == top) { // current window == window.top?
  alert('The script is in the topmost window, not in a frame');
} else {
  alert('The script runs in a frame!');
}
```

The “sandbox” iframe attribute

The `sandbox` attribute allows for the exclusion of certain actions inside an `<iframe>` in order to prevent it executing untrusted code. It “sandboxes” the iframe by treating it as coming from another origin and/or applying other limitations.

There’s a “default set” of restrictions applied for `<iframe sandbox src="...">`. But it can be relaxed if we provide a space-separated list of restrictions that should not be applied as a value of the attribute, like this: `<iframe sandbox="allow-forms allow-popups">`.

In other words, an empty `"sandbox"` attribute puts the strictest limitations possible, but we can put a space-delimited list of those that we want to lift.

Here’s a list of limitations:

`allow-same-origin`

By default `"sandbox"` forces the “different origin” policy for the iframe. In other words, it makes the browser to treat the `iframe` as coming from another origin, even if its `src` points to the same site. With all implied restrictions for scripts. This option removes that feature.

`allow-top-navigation`

Allows the `iframe` to change `parent.location`.

`allow-forms`

Allows to submit forms from `iframe`.

`allow-scripts`

Allows to run scripts from the `iframe`.

`allow-popups`

Allows to `window.open` popups from the `iframe`

See the manual for more.

The example below demonstrates a sandboxed iframe with the default set of restrictions: `<iframe sandbox src="...">`. It has some JavaScript and a form.

Please note that nothing works. So the default set is really harsh:

Result index.html sandboxed.html

The iframe below is has `sandbox` attribute.

<iframe sandbox="" srcdoc="<button>Click to run a script (doesn't work)</button><input><button>Submit (doesn't work)</button>"></iframe>

Please note:

The purpose of the `"sandbox"` attribute is only to *add more* restrictions. It cannot remove them. In particular, it can't relax same-origin restrictions if the iframe comes from another origin.

Cross-window messaging

The `postMessage` interface allows windows to talk to each other no matter which origin they are from.

So, it's a way around the "Same Origin" policy. It allows a window from `john-smith.com` to talk to `gmail.com` and exchange information, but only if they both agree and call corresponding JavaScript functions. That makes it safe for users.

The interface has two parts.

postMessage

The window that wants to send a message calls postMessage method of the receiving window. In other words, if we want to send the message to `win`, we should call `win.postMessage(data, targetOrigin)`.

Arguments:

`data`

The data to send. Can be any object, the data is cloned using the "structured cloning algorithm". IE supports only strings, so we should `JSON.stringify` complex objects to support that browser.

`targetOrigin`

Specifies the origin for the target window, so that only a window from the given origin will get the message.

The `targetOrigin` is a safety measure. Remember, if the target window comes from another origin, we can't read it's `location` in the sender window. So we can't be sure which site is open in the intended window right now: the user could navigate away, and the sender window has no idea about it.

Specifying `targetOrigin` ensures that the window only receives the data if it's still at the right site. Important when the data is sensitive.

For instance, here `win` will only receive the message if it has a document from the origin `http://example.com`:

```
1 <iframe src="http://example.com" name="example">
2
3 <script>
4   let win = window.frames.example;
```

```
5
6   win.postMessage("message", "http://example.com");
7 </script>
```

If we don't want that check, we can set `targetOrigin` to `*`.

```
1 <iframe src="http://example.com" name="example">
2
3 <script>
4   let win = window.frames.example;
5
6   win.postMessage("message", "*");
7 </script>
```

onmessage

To receive a message, the target window should have a handler on the `message` event. It triggers when `postMessage` is called (and `targetOrigin` check is successful).

The event object has special properties:

`data`

The data from `postMessage`.

`origin`

The origin of the sender, for instance `http://javascript.info`.

`source`

The reference to the sender window. We can immediately `source.postMessage(...)` back if we want.

To assign that handler, we should use `addEventListener`, a short syntax `window.onmessage` does not work.

Here's an example:

```
1 window.addEventListener("message", function(event) {
2   if (event.origin != 'http://javascript.info') {
3     // something from an unknown domain, let's ignore it
4     return;
5   }
6
7   alert( "received: " + event.data );
8
9   // can message back using event.source.postMessage(...)
10 });
```

The full example:


Enter message Click to send

Receiving iframe.

Summary

To call methods and access the content of another window, we should first have a reference to it.

For popups we have these references:

- From the opener window: `window.open` – opens a new window and returns a reference to it,
- From the popup: `window.opener` – is a reference to the opener window from a popup.

For iframes, we can access parent/children windows using:

- `window.frames` – a collection of nested window objects,
- `window.parent`, `window.top` are the references to parent and top windows,
- `iframe.contentWindow` is the window inside an `<iframe>` tag.

If windows share the same origin (host, port, protocol), then windows can do whatever they want with each other.

Otherwise, only possible actions are:

- Change the `location` of another window (write-only access).
- Post a message to it.

Exceptions are:

- Windows that share the same second-level domain: `a.site.com` and `b.site.com`. Then setting `document.domain='site.com'` in both of them puts them into the “same origin” state.
- If an iframe has a `sandbox` attribute, it is forcefully put into the “different origin” state, unless the `allow-same-origin` is specified in the attribute value. That can be used to run untrusted code in iframes from the same site.

The `postMessage` interface allows two windows with any origins to talk:

1. The sender calls `targetWin.postMessage(data, targetOrigin)`.
2. If `targetOrigin` is not `'*'`, then the browser checks if window `targetWin` has the origin `targetOrigin`.
3. If it is so, then `targetWin` triggers the `message` event with special properties:
   - `origin` – the origin of the sender window (like `http://my.site.com`)
   - `source` – the reference to the sender window.
   - `data` – the data, any object in everywhere except IE that supports only strings.

We should use `addEventListener` to set the handler for this event inside the target window.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)