19th January 2020

Resource loading: onload and onerror

The browser allows us to track the loading of external resources – scripts, iframes, pictures and so on.

There are two events for it:

- `onload` – successful load,
- `onerror` – an error occurred.

Loading a script

Let's say we need to load a third-party script and call a function that resides there.

We can load it dynamically, like this:

```
let script = document.createElement('script');
script.src = "my.js";

document.head.append(script);
```

…But how to run the function that is declared inside that script? We need to wait until the script loads, and only then we can call it.

Please note:

For our own scripts we could use JavaScript modules here, but they are not widely adopted by third-party libraries.

script.onload

The main helper is the `load` event. It triggers after the script was loaded and executed.

For instance:

```
let script = document.createElement('script');

// can load any script, from any domain
script.src = "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.j
document.head.append(script);

script.onload = function() {
  // the script creates a helper function "_"
  alert(_); // the function is available
};
```

So in `onload` we can use script variables, run functions etc.

…And what if the loading failed? For instance, there’s no such script (error 404) or the server is down (unavailable).

script.onerror

Errors that occur during the loading of the script can be tracked in an `error` event.

For instance, let’s request a script that doesn’t exist:

```
let script = document.createElement('script');
script.src = "https://example.com/404.js"; // no such script
document.head.append(script);

script.onerror = function() {
  alert("Error loading " + this.src); // Error loading https://example.com/40
};
```

Please note that we can’t get HTTP error details here. We don’t know if it was an error 404 or 500 or something else. Just that the loading failed.

Important:

Events `onload` / `onerror` track only the loading itself.

Errors that may occur during script processing and execution are out of scope for these events. That is: if a script loaded successfully, then `onload` triggers, even if it has programming errors in it. To track script errors, one can use `window.onerror` global handler.

Other resources

The `load` and `error` events also work for other resources, basically for any resource that has an external `src`.

For example:

```
let img = document.createElement('img');
img.src = "https://js.cx/clipart/train.gif"; // (*)

img.onload = function() {
  alert(`Image loaded, size ${img.width}x${img.height}`);
};

img.onerror = function() {
  alert("Error occurred while loading image");
};
```

There are some notes though:

- Most resources start loading when they are added to the document. But `<img>` is an exception. It starts loading when it gets a src `(*)`.
- For `<iframe>`, the `iframe.onload` event triggers when the iframe loading finished, both for successful load and in case of an error.

That's for historical reasons.

Crossorigin policy

There's a rule: scripts from one site can't access contents of the other site. So, e.g. a script at `https://facebook.com` can't read the user's mailbox at `https://gmail.com`.

Or, to be more precise, one origin (domain/port/protocol triplet) can't access the content from another one. So even if we have a subdomain, or just another port, these are different origins with no access to each other.

This rule also affects resources from other domains.

If we're using a script from another domain, and there's an error in it, we can't get error details.

For example, let's take a script `error.js` that consists of a single (bad) function call:

```
// 📁 error.js
noSuchFunction();
```

Now load it from the same site where it's located:

```
<script>
window.onerror = function(message, url, line, col, errorObj) {
  alert(`${message}\n${url}, ${line}:${col}`);
};
</script>
<script src="/article/onload-onerror/crossorigin/error.js"></script>
```

We can see a good error report, like this:

```
Uncaught ReferenceError: noSuchFunction is not defined
https://javascript.info/article/onload-onerror/crossorigin/error.js, 1:1
```

Now let's load the same script from another domain:

```
<script>
window.onerror = function(message, url, line, col, errorObj) {
  alert(`${message}\n${url}, ${line}:${col}`);
};
</script>
<script src="https://cors.javascript.info/article/onload-onerror/crossorigin/
```

The report is different, like this:

```
Script error.
, 0:0
```

Details may vary depending on the browser, but the idea is the same: any information about the internals of a script, including error stack traces, is hidden. Exactly because it's from another domain.

Why do we need error details?

There are many services (and we can build our own) that listen for global errors using `window.onerror`, save errors and provide an interface to access and analyze them. That's great, as we can see real errors, triggered by our users. But if a script comes from another origin, then there's not much information about errors in it, as we've just seen.

Similar cross-origin policy (CORS) is enforced for other types of resources as well.

To allow cross-origin access, the `<script>` tag needs to have the `crossorigin` attribute, plus the remote server must provide special headers.

There are three levels of cross-origin access:

1. **No `crossorigin` attribute** – access prohibited.
2. **`crossorigin="anonymous"`** – access allowed if the server responds with the header `Access-Control-Allow-Origin` with `*` or our origin. Browser does not send authorization information and cookies to remote server.
3. **`crossorigin="use-credentials"`** – access allowed if the server sends back the header `Access-Control-Allow-Origin` with our origin and `Access-Control-Allow-Credentials: true`. Browser sends authorization information and cookies to remote server.

Please note:

You can read more about cross-origin access in the chapter Fetch: Cross-Origin Requests. It describes the `fetch` method for network requests, but the policy is exactly the same.

Such thing as "cookies" is out of our current scope, but you can read about them in the chapter Cookies, document.cookie.

In our case, we didn't have any crossorigin attribute. So the cross-origin access was prohibited. Let's add it.

We can choose between `"anonymous"` (no cookies sent, one server-side header needed) and `"use-credentials"` (sends cookies too, two server-side headers needed).

If we don't care about cookies, then `"anonymous"` is the way to go:

```
<script>
window.onerror = function(message, url, line, col, errorObj) {
  alert(`${message}\n${url}, ${line}:${col}`);
};
</script>
<script crossorigin="anonymous" src="https://cors.javascript.info/article/onl
```

Now, assuming that the server provides an `Access-Control-Allow-Origin` header, everything's fine. We have the full error report.

Summary

Images `<img>` , external styles, scripts and other resources provide `load` and `error` events to track their loading:

- `load` triggers on a successful load,
- `error` triggers on a failed load.

The only exception is `<iframe>` : for historical reasons it always triggers `load` , for any load completion, even if the page is not found.

The `readystatechange` event also works for resources, but is rarely used, because `load/error` events are simpler.

Tasks

Load images with a callback

importance: 4

Normally, images are loaded when they are created. So when we add `<img>` to the page, the user does not see the picture immediately. The browser needs to load it first.

To show an image immediately, we can create it "in advance", like this:

```
let img = document.createElement('img');
img.src = 'my.jpg';
```

The browser starts loading the image and remembers it in the cache. Later, when the same image appears in the document (no matter how), it shows up immediately.

Create a function `preloadImages(sources, callback)` that loads all images from the array `sources` and, when ready, runs `callback` .

For instance, this will show an `alert` after the images are loaded:

```
function loaded() {
  alert("Images loaded")
}

preloadImages(["1.jpg", "2.jpg", "3.jpg"], loaded);
```

In case of an error, the function should still assume the picture "loaded".

In other words, the `callback` is executed when all images are either loaded or errored out.

The function is useful, for instance, when we plan to show a gallery with many scrollable images, and want to be sure that all images are loaded.

In the source document you can find links to test images, and also the code to check whether they are loaded or not. It should output `300` .

Open a sandbox for the task.

solution

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…)