Using Fetch

BY **ZELL LIEW** ON MAY 2, 2017

AJAX, **FETCH**

Whenever we send or retrieve information with JavaScript, we initiate a thing known as an Ajax call. Ajax is a technique to send and retrieve information behind the scenes without needing to refresh the page. It allows browsers to send and retrieve information, then do things with what it gets back, like add or change HTML on the page.

Let's take a look at the history of that and then bring ourselves up-to-date.

Another note here, we're going to be using ES6 syntax for all the demos in this article.

A few years ago, the easiest way to initiate an Ajax call was through the use of jQuery's `ajax` method:

jQuery

```
$.ajax('some-url', {
  success: (data) => { /* do something with the data */ },
  error: (err) => { /* do something when an error happens */}
});
```

We could do Ajax without jQuery, but we had to write an `XMLHttpRequest`, which is pretty complicated.

Thankfully, browsers nowadays have improved so much that they support the Fetch API, which is a modern way to Ajax without helper libraries like jQuery or Axios. In this article, I'll show you how to use Fetch to handle both success and errors.

Support for Fetch

Let's get support out of the way first.

This browser support data is from Caniuse, which has more detail. A number indicates that browser supports the feature at that version and up.

Desktop

Chrome	Opera	Firefox	IE	Edge	Safari
42	29	39	No	14	10.1

Mobile / Tablet

iOS Safari: 10.3	Opera Mobile: 37	Opera Mini: No	Android: 56	Android Chrome: 57
Android Firefox: 52				

Support for Fetch is pretty good! All major browsers (with the exception of Opera Mini and old IE) support it natively, which means you can safely use it in your projects. If you need support anywhere it isn't natively supported, you can always depend on this handy polyfill.

Getting data with Fetch

Getting data with Fetch is easy. You just need to provide Fetch with the resource you're trying to fetch (so meta!).

Let's say we're trying to get a list of Chris' repositories on Github. According to Github's API, we need to make a `get` request for `api.github.com/users/chriscoyier/repos` .

This would be the fetch request:

JS

```js
fetch('https://api.github.com/users/chriscoyier/repos');
```

So simple! What's next?

Fetch returns a Promise, which is a way to handle asynchronous operations without the need for a callback.

To do something after the resource is fetched, you write it in a `.then` call:

JS

```js
fetch('https://api.github.com/users/chriscoyier/repos')
  .then(response => {/* do something */})
```

If this is your first encounter with Fetch, you'll likely be surprised by the `response` Fetch returns. If you `console.log` the response, you'll get the following information:

JS

```
{
  body: ReadableStream
  bodyUsed: false
  headers: Headers
  ok : true
  redirected : false
  status : 200
  statusText : "OK"
  type : "cors"
  url : "http://some-website.com/some-url"
  __proto__ : Response
}
```

Here, you can see that Fetch returns a response that tells you the status of the request. We can see that the request is successful (`ok` is true and `status` is 200), but a list of Chris' repos isn't present anywhere!

Turns out, what we requested from Github is hidden in `body` as a readable stream. We need to call an appropriate method to convert this readable stream into data we can consume.

Since we're working with GitHub, we know the response is JSON. We can call `response.json` to convert the data.

There are other methods to deal with different types of response. If you're requesting an XML file, then you should call `response.text` . If you're requesting an image, you call `response.blob` .

All these conversion methods (`response.json` et all) returns another Promise, so we can get the data we wanted with yet another `.then` call.

JS

```
fetch('https://api.github.com/users/chriscoyier/repos')
  .then(response => response.json())
  .then(data => {
    // Here's a list of repos!
    console.log(data)
  });
```

Phew! That's all you need to do to get data with Fetch! Short and simple, isn't it? :)

Next, let's take a look at sending some data with Fetch.

Sending data with Fetch

Sending data with Fetch is pretty simple as well. You just need to configure your fetch request with three options.

JS

```js
fetch('some-url', options);
```

The **first option** you need to set is your *request method* to `post` , `put` or `del` . Fetch automatically sets the `method` to `get` if you leave it out, which is why getting a resource takes lesser steps.

The **second option** is to set your *headers*. Since we're primarily sending JSON data in this day and age, we need to set `Content-Type` to be `application/json` .

The **third option** is to set a body that contains JSON content. Since JSON content is required, you often need to call `JSON.stringify` when you set the `body` .

In practice, a `post` request with these three options looks like:

JS

```js
let content = {some: 'content'};

// The actual fetch request
fetch('some-url', {
  method: 'post',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(content)
})
// .then()...
```

For the sharp-eyed, you'll notice there's some boilerplate code for every `post` , `put` or `del` request. Ideally, we can reuse our headers and call `JSON.stringify` on the content before sending since we already know we're sending JSON data.

But even with the boilerplate code, Fetch is still pretty nice for sending any request.

Handling errors with Fetch, however, isn't as straightforward as handling success messages. You'll see why in a moment.

Handling errors with Fetch

Although we always hope for Ajax requests to be successful, they can fail. There are many reasons why requests may fail, including but not limited to the following:

1. You tried to fetch a non-existent resource.
2. You're unauthorized to fetch the resource.
3. You entered some arguments wrongly
4. The server throws an error.
5. The server timed out.
6. The server crashed.
7. The API changed.
8. ...

Things aren't going to be pretty if your request fails. Just imagine a scenario you tried to buy something online. An error occured, but it remains unhandled by the people who coded the website. As a result, after clicking buy, nothing moves. The page just hangs there... You have no idea if anything happened. Did your card go through? 😟.

Now, let's try to fetch a non-existent error and learn how to handle errors with Fetch. For this example, let's say we misspelled `chriscoyier` as `chrissycoyier`

JS

```
// Fetching chrissycoyier's repos instead of chriscoyier's repos
fetch('https://api.github.com/users/chrissycoyier/repos')
```

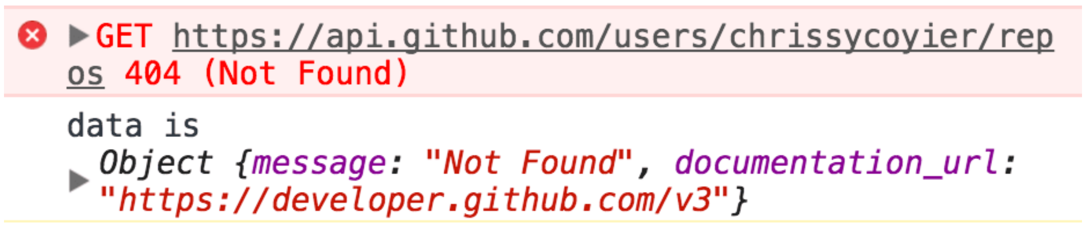
We already know we should get an error since there's no `chrissycoyier` on Github. To handle errors in promises, we use a `catch` call.

Given what we know now, you'll probably come up with this code:

JS

```
fetch('https://api.github.com/users/chrissycoyier/repos')
  .then(response => response.json())
  .then(data => console.log('data is', data))
  .catch(error => console.log('error is', error));
```

Fire your fetch request. This is what you'll get:



```
▶GET https://api.github.com/users/chrissycoyier/repos 404 (Not Found)
data is
▶ Object {message: "Not Found", documentation_url:
  "https://developer.github.com/v3"}
```

Fetch failed, but the code that gets executed is the second `.then` instead of `.catch`

Why did our second `.then` call execute? Aren't promises supposed to handle errors with `.catch`? Horrible! 😨😨😨

If you `console.log` the response now, you'll see slightly different values:

JS

```
{
  body: ReadableStream
  bodyUsed: true
  headers: Headers
  ok: false // Response is not ok
  redirected: false
  status: 404 // HTTP status is 404.
  statusText: "Not Found" // Request not found
  type: "cors"
  url: "https://api.github.com/users/chrissycoyier/repos"
}
```

Most of the response remain the same, except `ok`, `status` and `statusText`. As expected, we didn't find chrissycoyier on Github.

This response tells us Fetch doesn't care whether your AJAX request succeeded. It only cares about sending a request and receiving a response from the server, which means we need to throw an error if the request failed.

Hence, the initial `then` call needs to be rewritten such that it only calls `response.json` if the request succeeded. The easiest way to do so to check if the `response` is `ok`.

JS

```
fetch('some-url')
  .then(response => {
    if (response.ok) {
      return response.json()
    } else {
      // Find some way to get to execute .catch()
    }
  });
```

Once we know the request is unsuccessful, we can either `throw` an Error or `reject` a Promise to activate the `catch` call.

JS

```
// throwing an Error
else {
  throw new Error('something went wrong!')
}

// rejecting a Promise
else {
  return Promise.reject('something went wrong!')
}
```

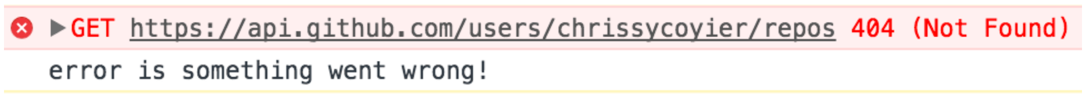
Choose either one, because they both activate the `.catch` call.

Here, I choose to use `Promise.reject` because it's easier to implement. Errors are cool too, but they're harder to implement, and the only benefit of an Error is a stack trace, which would be non-existent in a Fetch request anyway.

So, the code looks like this so far:

JS

```
fetch('https://api.github.com/users/chrissycoyier/repos')
  .then(response => {
    if (response.ok) {
      return response.json()
    } else {
      return Promise.reject('something went wrong!')
    }
  })
  .then(data => console.log('data is', data))
  .catch(error => console.log('error is', error));
}
```

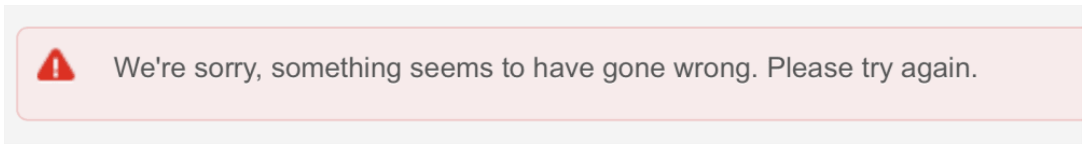


```
✖ ▶GET https://api.github.com/users/chrissycoyier/repos 404 (Not Found)
  error is something went wrong!
```

Failed request, but error gets passed into catch correctly

This is great. We're getting somewhere since we now have a way to handle errors.

But rejecting the promise (or throwing an Error) with a generic message isn't good enough. We won't be able to know what went wrong. I'm pretty sure you don't want to be on the receiving end for an error like this...



We're sorry, something seems to have gone wrong. Please try again.

Yeah... I get it that something went wrong... but what exactly? □

What went wrong? Did the server time out? Was my connection cut? There's no way for me to know! What we need is a way to tell what's wrong with the request so we can handle it appropriately.

Let's take a look at the response again and see what we can do:

JS

```
{
  body: ReadableStream
  bodyUsed: true
  headers: Headers
  ok: false // Response is not ok
  redirected: false
  status: 404 // HTTP status is 404.
  statusText: "Not Found" // Request not found
  type: "cors"
  url: "https://api.github.com/users/chrissycoyier/repos"
}
```

Okay great. In this case, we know the resource is non-existent. We can return a `404` status or `Not Found` status text and we'll know what to do with it.

To get `status` and `statusText` into the `.catch` call, we can reject a JavaScript object:

JS

```js
fetch('some-url')
  .then(response => {
    if (response.ok) {
      return response.json()
    } else {
      return Promise.reject({
        status: response.status,
        statusText: response.statusText
      })
    }
  })
  .catch(error => {
    if (error.status === 404) {
      // do something about 404
    }
  })
```

Now we're getting somewhere again! Yay! 😄.

Let's make this better! 😌.

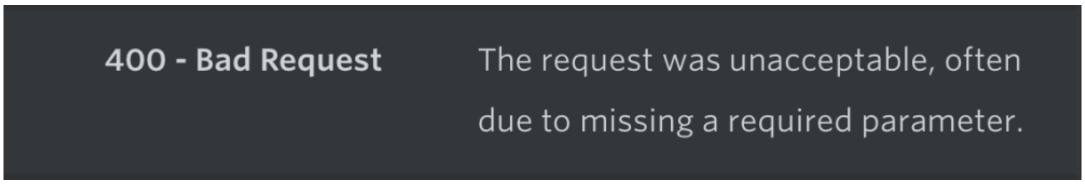
The above error handling method is good enough for certain HTTP statuses which doesn't require further explanation, like:

- 401: Unauthorized
- 404: Not found
- 408: Connection timeout
- ...

But it's not good enough for this particular badass:

- **400: Bad request**.

What constitutes bad request? It can be a whole slew of things! For example, Stripe returns 400 if the request is missing a required parameter.



400 - Bad Request	The request was unacceptable, often due to missing a required parameter.

Stripe's explains it returns a 400 error if the request is missing a required field

It's not enough to just tell our `.catch` statement there's a bad request. We need more information to tell what's missing. Did your user forget their first name? Email? Or maybe their credit card information? We won't know!

Ideally, in such cases, your server would return an object, telling you what happened together with the failed request. If you use Node and Express, such a response can look like this.

JS

```js
res.status(400).send({
  err: 'no first name'
})
```

Here, we can't reject a Promise in the initial `.then` call because the error object from the server can only be read after `response.json`.

The solution is to return a promise that contains two `then` calls. This way, we can first read what's in `response.json`, then decide what to do with it.

Here's what the code looks like:

JS

```js
fetch('some-error')
  .then(handleResponse)

function handleResponse(response) {
  return response.json()
    .then(json => {
      if (response.ok) {
        return json
      } else {
        return Promise.reject(json)
      }
    })
}
```

Let's break the code down. First, we call `response.json` to read the json data the server sent. Since, `response.json` returns a Promise, we can immediately call `.then` to read what's in it.

We want to call this second `.then` within the first `.then` because we still need to access `response.ok` to determine if the response was successful.

If you want to send the status and statusText along with the json into `.catch` , you can combine them into one object with `Object.assign()` .

JS

```js
let error = Object.assign({}, json, {
  status: response.status,
  statusText: response.statusText
})
return Promise.reject(error)
```

With this new `handleResponse` function, you get to write your code this way, and your data gets passed into `.then` and `.catch` automatically

JS

```js
fetch('some-url')
  .then(handleResponse)
  .then(data => console.log(data))
  .catch(error => console.log(error))
```

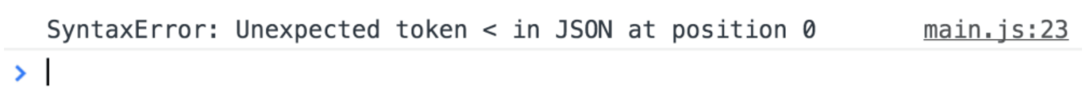
Unfortunately, we're not done with handling the response just yet :(

Handling other response types

So far, we've only touched on handling JSON responses with Fetch. This already solves 90% of use cases since APIs return JSON nowadays.

What about the other 10%?

Let's say you received an XML response with the above code. Immediately, you'll get an error in your catch statement that says:



```
SyntaxError: Unexpected token < in JSON at position 0        main.js:23
> |
```

Parsing an invalid JSON produces a Syntax error

This is because XML isn't JSON. We simply can't return `response.json` . Instead, we need to return `response.text` . To do so, we need to check for the content type by accessing the response headers:

```js
.then(response => {
  let contentType = response.headers.get('content-type')

  if (contentType.includes('application/json')) {
    return response.json()
    // ...
  }

  else if (contentType.includes('text/html')) {
    return response.text()
    // ...
  }

  else {
    // Handle other responses accordingly...
  }
});
```

Wondering why you'll ever get an XML response?

Well, I encountered it when I tried using ExpressJWT to handle authentication on my server. At that time, I didn't know you can send JSON as a response, so I left it as its default, XML. This is just one of the many unexpected possibilities you'll encounter. Want another? Try fetching `some-url` :)

Anyway, here's the entire code we've covered so far:

JS

```js
fetch('some-url')
  .then(handleResponse)
  .then(data => console.log(data))
  .then(error => console.log(error))

function handleResponse (response) {
  let contentType = response.headers.get('content-type')
  if (contentType.includes('application/json')) {
    return handleJSONResponse(response)
  } else if (contentType.includes('text/html')) {
    return handleTextResponse(response)
  } else {
    // Other response types as necessary. I haven't found a need for
    throw new Error(`Sorry, content-type ${contentType} not supportec
  }
}

function handleJSONResponse (response) {
  return response.json()
    .then(json => {
      if (response.ok) {
        return json
      } else {
        return Promise.reject(Object.assign({}, json, {
          status: response.status,
          statusText: response.statusText
        }))
      }
    })
}
function handleTextResponse (response) {
  return response.text()
    .then(text => {
      if (response.ok) {
        return json
```

```
} else {
```

It's a lot of code to write/copy and paste into if you use Fetch. Since I use Fetch heavily in my projects, I create a library around Fetch that does exactly what I described in this article (plus a little more).

Introducing zlFetch

zlFetch is a library that abstracts away the `handleResponse` function so you can skip ahead to and handle both your data and errors without worrying about the response.

A typical zlFetch look like this:

JS

```js
zlFetch('some-url', options)
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

To use zlFetch, you first have to install it.

Command Line

```
npm install zl-fetch --save
```

Then, you'll import it into your code. (Take note of `default` if you aren't importing with ES6 imports). If you need a polyfill, make sure you import it before adding zlFetch.

JS

```js
// Polyfills (if needed)
require('isomorphic-fetch') // or whatwg-fetch or node-fetch if you p

// ES6 Imports
import zlFetch from 'zl-fetch';

// CommonJS Imports
const zlFetch = require('zl-fetch');
```

zlFetch does a bit more than removing the need to handle a Fetch response. It also helps you send JSON data without needing to write headers or converting your body to JSON.

The below the functions do the same thing. zlFetch adds a `Content-Type` and converts your content into JSON under the hood.

```js
let content = {some: 'content'}

// Post request with fetch
fetch('some-url', {
  method: 'post',
  headers: {'Content-Type': 'application/json'}
  body: JSON.stringify(content)
});

// Post request with zlFetch
zlFetch('some-url', {
  method: 'post',
  body: content
});
```

zlFetch also makes authentication with JSON Web Tokens easy.

The standard practice for authentication is to add an `Authorization` key in the headers. The contents of this `Authorization` key is set to `Bearer your-token-here` . zlFetch helps to create this field if you add a `token` option.

So, the following two pieces of code are equivalent.

```js
let token = 'someToken'
zlFetch('some-url', {
  headers: {
    Authorization: `Bearer ${token}`
  }
});

// Authentication with JSON Web Tokens with zlFetch
zlFetch('some-url', {token});
```

That's all zlFetch does. It's just a convenient wrapper function that helps you write less code whenever you use Fetch. Do check out zlFetch if you find it interesting. Otherwise, feel free to roll your own!

Here's a Pen for playing around with zlFetch:

Wrapping up

Fetch is a piece of amazing technology that makes sending and receiving data a cinch. We no longer need to write XHR requests manually or depend on larger libraries like jQuery.

Although Fetch is awesome, error handling with Fetch isn't straightforward. Before you can handle errors properly, you need quite a bit of boilerplate code to pass information go to your `.catch` call.

With zlFetch (and the info presented in this article), there's no reason why we can't handle errors properly anymore. Go out there and put some fun into your error messages too :)

By the way, if you liked this post, you may also like other front-end-related articles I write on my blog. Feel free to pop by and ask any questions you have. I'll get back to you as soon as I can.

Comments

goose

MAY 2, 2017

I've got a question and an opinion.

Why is there two async calls? It only calls the resource once, right? Surely it doesn't call the URL a second time to get the body. If it's not calling the URL twice, why does it need to be async, isn't the data of the body already in the client?

Second, Fetch seems incredibly cool. I'll say though that it doesn't compete directly with jQuery. For most programmers, we won't be switching to several smaller libraries like zlFetch and Sizzle. This does seem incredible for those situations when I'm working without jQuery already. Great piece!

Agop

MAY 2, 2017

When you say "two async calls," you're talking about `fetch()` and `response.json()`, right?

The reason that `response.json()` (as well as `.blob()` and `.text()` and so on) is async is because when `fetch()` completes, the body of the response isn't necessarily all there yet (e.g. the server could have sent only 50% of the response so far). In order for `.json()` to return an object, it needs to wait for the complete body of the response first (again, same for `.text()` and so on).

This means that you can actually stream the response as it comes in. Imagine `fetch()` ing a giant CSV file with 1 million rows, but you only need the 50th row. Calling `.text()` and parsing it as a CSV is going to be *really, really slow* because it has to wait for the entire file. On the other hand, calling `.getReader().read()` (and calling `.read()` again and again as necessary) allows you to get to the 50th row much faster and ignore the rest.

See the example here: https://fetch.spec.whatwg.org/#fetch-api

James Edwards

MAY 2, 2017

Given all the messing about you have to do to handle errors and invalid responses, exactly how is this simpler than XMLHttpRequest?

Agop

MAY 2, 2017

I think that the example in the article might be a bit exaggerated in order to show every possible scenario.

In a more realistic situation, you'd only need the boilerplate code that handles your specific API responses. E.g.

```
async function callAPI(url, options)
{
    const resp = await fetch(url, options),
        body = await resp.json();

    if (!resp.ok || body.error)
    {
        throw new APIError(body.error || `Unknown
    }

    return body;
}
```

I'm sure most people would find this a lot cleaner than the XMLHttpRequest alternative.

Jull Weber

MAY 2, 2017

In your handleTextResponse() you are return json var instead of the var text you defined in the arrow function…

Oliver Williams

MAY 2, 2017

"Fetch returns a Promise, which is a way to handle asynchronous operations without the need for a callback."
This is really bad wording! Promises use callback functions – they're just structured in a way that is more logical to reason about

Šime Vidas

MAY 2, 2017

Not necessarily; you can `await` a promise ;)

FND

MAY 3, 2017

Thanks for this write-up! It's worth noting that, unlike with XHR, you need to be explicit for protected resources: `{ credentials: "same-origin" }` (or `"include"` for CORS).

Phil

MAY 3, 2017

Thanks for the great write up on fetch.

I think that the fetch API is great and easier to use than XMLHttpRequest, however as you have shown, fetch like XMLHttpRequest still requires a fair bit of extra handling to make it work nice.

Where I used jQuery to wrap XMLHttpRequest, I need something similar for Fetch, zlFetch seems to do the job, but why would I use it in place of Axios? Which seems to be more popular and works with the same API in the browser and Node.js.

Zell

MAY 12, 2017

Axios is cool too. Choose whichever floats your boat :)

Aslam

MAY 5, 2017

Really good tutorial. Thanks :)

eigenstates

MAY 8, 2017

The thing about the whole promise stuff is that it looks and reads to the eye much more complicated that traditional JS. You also draw comparisons in the article to jQuery but then at the end throw in yet another library.

My point here is that every dev should have an XMLHttpRequest utility function- you only need to write it once. The code ends up much more readable than the new

Promise idiom. With what people call callback hell it seems prudent to remember .then is still a callback.

Chang Wang

MAY 12, 2017

Fetch does not support aborting/cancelling requests (and it likely never will since cancellation of promises is a dead feature), it does not support specifying a timeout, it does not support upload progress.

Perhaps those aren't things that you need right away, but unless you can guarantee that none of those things will be needed in your project's lifetime, using fetch would not be a prudent decision for your project.

Mike

MAY 13, 2017

Exactly !
I don't understand why everybody starts using fetch.
I's so limited comparatively to XMLHttpRequest.
And it's easy to wrap XMLHttpRequest in a promise if necessary.

Ununderstandable …

Yuqing Jiang

MAY 14, 2017

I use reader.cancel() to cancel the request.
reader = response.body.getReader()
reader.cancel()

like this.

But Firefox currently has no response.body..

when the request is cancelled

the val.done is true

like

reader.read().then(function(val) {
// val.done is true when cancelled
});

Jérémy

MAY 12, 2017

I think that with fetch you can't track a progress of an upload for instance...

CroModder

MAY 12, 2017

"A few years ago, the easiest way to initiate an Ajax call was through the use of jQuery's ajax method"
– From the article, I don't see that changed :)

Good write up, but I can't understand why devs are pretending that fetch API is **easier** than jQuery.
I don't say jQuery justifies its weight and all, but why pushing too much how cool, simple, clean, magnificent... fetch API is?

Sharlaan

MAY 12, 2017

Awesome lib zlFetch, exactly what i were searching for: a wrapper for fetch, like Axios does for xmlHTTPrequest !

I think it would be nice this article ends with a comparison with Axios for nowadays usecases, especially progress handling (like for image fetching). If i remember correctly, fetch does not support yet progress and cancel, while xmlHTTPrequest does (and by extension Axios).

Bogdan

MAY 16, 2017

In handleTextResponse() you should return "text" not "json" if (response.ok)