Lesson Weekend

Intermediate JavaScript (/intermediate-javascript) / Asynchrony and APIs (/intermediate-javascript/asynchrony-and-apis) / Exception Handling for API Calls

Text

In this lesson, we'll complete our OpenWeather API website by incorporating error handling into our API request. What this means is having our code do something when an API call is not 200 OK, but instead any of the other possible HTTP status codes, like:

- `400` : Bad request. We'd better check the API request and make sure it's correct.
- `401` : Unauthorized. We aren't authorized to access the resource, which might mean we haven't logged in correctly.
- `403` : Forbidden — we aren't allowed to access that content.
- `404` : Not found. The resource couldn't be found.
- `500` : Internal server error. Not our fault! Something is going on with the API. In general, if we get any `500` errors, the server is having problems.

The error handling that we'll incorporate into our application will be basic: if the API response is not 200 OK, we'll display an error message in the webpage letting the user know what happened, including a status code and short descriptive text. This won't fix all of the errors that could happen, but in some cases it will give the user just enough information to fix some errors they run into, like a typo in their input.

That's why this solution is basic — we're making sure to communicate to the user what happened with the request. More advanced error handling might involve logging all errors that are generated by the API and using a notification system so that developers are notified of these errors and can address them. In fact, there are tech companies who sell products that do monitoring and error reporting, and most website hosting sites have such tools built-in. In this program, we'll stick to basic error handling since learning about software monitoring and reporting tools is out of the scope of this program.

We'll learn two ways of delivering information about errors to the user: using `XMLHttpRequest` object properties, and using the error messages that an API creates and sends independently.

Let's get into it!

The Errors that We'll Encounter

The errors that we'll deal with — and can test for in our apps — are **client** errors, which are errors that come from the computer making the request to the API, also known as the "client". In this lesson, we'll cause a few errors and see how the OpenWeather API responds:

- A bad input
- A bad API key
- A bad request URL (but a good API key)

Other APIs may respond a bit differently to these errors, but using these test cases will be sufficient to ensure we have very solid, albeit basic, error handling in place.

When you add error handling to your API projects in this course section, make sure that your code handles the above three situations.

Error Handling with `XMLHttpRequest.status` and `XMLHttpRequest.statusText`

The first method of error handling is one that will stay the same with every API that you work with, so long as you are using an `XMLHttpRequest` object. That's because it involves using properties from the `XMLHttpRequest` object: `XMLHttpRequest.status` and `XMLHttpRequest.statusText`. We briefly learned about these properties in a previous lesson, so let's do another review here:

- `status`: The status is the HTTP status code of the API response. A 200 means it was successful. As we just reviewed, there are many other codes such as 404 not found and so on.
- `statusText`: This is the short description or title that accompanies the `status` code. For a 200 status, the `statusText` will be "OK". For a 404 status, the `statusText` will be "Not Found".

You should not bother to memorize all status codes, and instead get in the habit of looking them up anytime you run into a new one. You can find a complete list of status codes on Wikipedia (https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) and MDN (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status).

The great thing about these two properties is that they are automatically populated by the `XMLHttpRequest` object when we make an API call. That means we can always rely on them to contain relevant information to the API call we just made. Let's go ahead and use these two properties in our code.

Using `XMLHttpRequest.status` and `XMLHttpRequest.statusText`

The first section of code that we'll want to update is our `"loadend"` event listener. Currently our code looks like this:

```
function getWeather(city) {
  ...

  request.addEventListener("loadend", function() {
    const response = JSON.parse(this.responseText);
    if (this.status === 200) {
      printElements(response, city);
    }
  });

  ...
}
```

We'll add an `else` statement to our branching to handle every other case — that is, when `this.status` does NOT equal `200`.

Here's our new code:

src/index.js

```
function getWeather(city) {
  ...

  request.addEventListener("loadend", function() {
    const response = JSON.parse(this.responseText);
    if (this.status === 200) {
      printElements(response, city);
    } else {
      printError(this, city);
    }
  });

  ...
}
```

Take note that the `printError` function doesn't exist yet. However, we know what direction we're heading in, so we can already define the name of the function and what arguments should be passed into it. Let's break this down:

- We call our function `printError`, because it will handle printing error messages to the DOM.
- We're passing in two arguments: `this` and `city`.
  - Why `this`? Remember that `this` represents the `XMLHttpRequest` object and we need to access the `status` and `statusText` properties inside of it.
  - Why `city`? Remember that the `city` variable represents the user's input. We don't actually have to include this information when we print an error message to the DOM, but it will make our error message more descriptive.

With this new code, we're handling all other status codes at once, in one `else` statement. While we could target specific status codes, something like this:

```
request.addEventListener("loadend", function() {
  const response = JSON.parse(this.responseText);
  if (this.status === 200) {
    printElements(response, city);
  } else if (this.status === 404) {
    printError(this, city);
  } else if (this.status === 401) {
    printError(this, city);
  }
  ...
  ...
});
```

This is verbose and unnecessary. We can capture all non-200 status codes with just one `else` statement.

Alright, let's create our `printError` function next. We can add this to our user interface logic, just above the `printElements` function.

src/index.js

```
// UI Logic

function printError(request, city) {
  document.querySelector('#showResponse').innerText = `There was an error accessing the weather data for ${city}:  ${request.status} ${request.statusText}`;
}
```

We start out by declaring two descriptive parameters that match the arguments we already passed in: `request` and `city`.

Next, we access the P tag with the `id` of `showResponse` and update its inner text with an error message. Here, too, we use template literals to add variables directly to our string.

Since the `request` parameter represents the `XMLHttpRequest` object, we can access the `status` and `statusText` properties with dot notation: `request.status` and `request.statusText`.

The extra touch with this error message is including the city. You can come up with your own error message in your own projects, and include as much extra detail as you want.

Testing the Error Handling

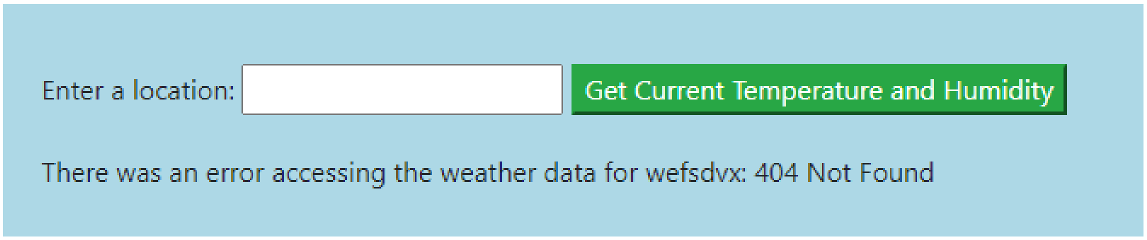
Now that we have our code in place, let's test it out to make sure that it functions as expected. We have three test cases that we'll try out:

- A bad input
- A bad API key
- A bad request URL (but a good API key)

Let's start with a bad user input. This is asking the question, what will the status code be if a resource cannot be found given the user's input?

Start your application, and enter in a phony input like "wefsdvx". We should see our error message print on the page!

```
There was an error accessing the weather data for wefsdvx:
404 Not Found
```



Enter a location: Get Current Temperature and Humidity

There was an error accessing the weather data for wefsdvx: 404 Not Found

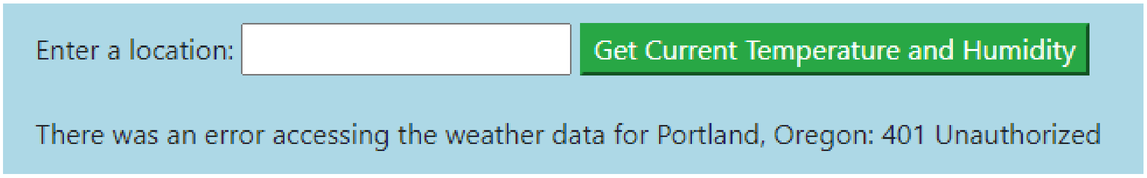
Very cool! We can see our error handling working as expected.

Next, let's try a bad API key. With this test, we're asking the question, what will the status code be if we're not authorized to access a resource on an API?

With this one, we'll need to close our server, change the API key in the `.env`, and then restart the server. We can mess up our API key simply by removing the last character. Go ahead and do that now, and then restart the server.

If we enter a location, even a bogus one, we should see a new status code printed to the webpage.

```
There was an error accessing the weather data for Portland,
Oregon: 401 Unauthorized
```



Enter a location: Get Current Temperature and Humidity

There was an error accessing the weather data for Portland, Oregon: 401 Unauthorized

Alright, let's move onto our last test. Before we do, be sure to close your server, fix your API key, and restart it.

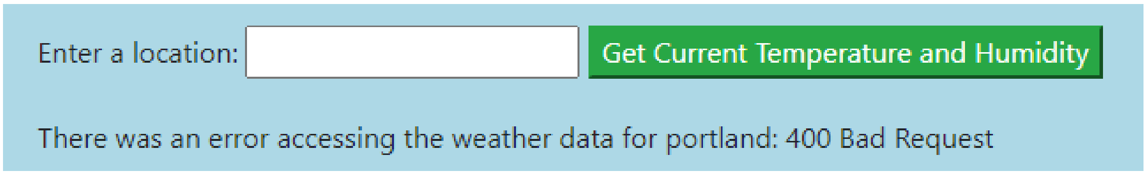
With this next test, we're asking the question, what will the status code be if our request is written incorrectly?

To cause this error, let's change something about our request URL; we'll change the request parameter `q=${city}` to be `p=${city}`:

```
function getWeather() {
  ...
  const url = `http://api.openweathermap.org/data/2.5/weath
er?p=${city}&appid=${process.env.API_KEY}`;
  ...
}
```

Enter in a location to see what sort of error message this typo in our parameter causes:

```
There was an error accessing the weather data for portland:
400 Bad Request
```



Enter a location: Get Current Temperature and Humidity

There was an error accessing the weather data for portland: 400 Bad Request

Sure enough, we'll get an error message about our request being incorrect in some way.

Since we as developers are on the hook for ensuring that our request URLs work as expected, this last situation should only come up in development. However, trying out many different ways to break our code is always a good learning process, and it helps us ensure that our error handling works as expected.

Using an API's Error Messages

Next, let's look at the other way we can handle errors when we make a request to an API: using the API's own error messages. The important gotcha about this method is that no two APIs are the same, which means how errors and relevant info is reported will always vary. Sometimes APIs have extensive internal error messages, and other APIs have none at all. We'll revisit this in just a moment.

One thing to note about API call errors is that the status code will never change for the type of error that's reported whether you are using the API's custom error messages or `XMLHttpRequest.status` and `XMLHttpRequest.statusText`.

So why bother using the API's custom error messages if we can essentially get the same information via the `XMLHttpRequest` object? Well, custom error messages that are crafted by an API are usually more descriptive, which means we can use them to provide a better user and developer experience. Let's look at the same 3 test cases that we just tried out and see what the API returns as far as an error message.

1. A bad input
    - `XMLHttpRequest` object returns 404 Not Found
    - The API returns 404 "city not found"
2. A bad API key
    - `XMLHttpRequest` object returns 401 Unauthorized
    - The API returns 401 "Invalid API key. Please see http://openweathermap.org/faq#error401 for more info."
3. A bad request URL (but a good API key)

  - `XMLHttpRequest` object returns 400 Bad Request
  - The API returns 400 "Nothing to geocode"

As we can see, the OpenWeather API returns error messages that are way more descriptive, which can be a big benefit to users and developers alike.

So, let's learn how to access these errors.

Figuring Out How an API Reports Errors

Keep in mind that every API will report errors differently. To find out how an API reports errors, we need to do two things:

- Cause errors in our API calls and see what gets returned.
- Look for information in the documentation about how errors are structured.

The order in which you do the two tasks above doesn't matter. What matters is that you do both. We'll start with the first one — introducing errors into our API calls and seeing what gets returned. We can easily do this with Postman!

Introducing Errors in an API Call via Postman

We'll look at one example to see how the OpenWeather API structures its errors. We'll look at a bad input (status code 404). To see what a bad input returns we can enter the following URL into Postman:

```
http://api.openweathermap.org/data/2.5/weather?q=a3edf3&appid=[YOUR-API-KEY-HERE]
```

Where `[YOUR-API-KEY-HERE]` is replaced with your API key.

After we send this request, we should get a response object from the API that looks like this:

```
{
    "cod": "404",
    "message": "city not found"
}
```

What this tells us is that when there's an error, the OpenWeather API returns two keys: `"cod"` and `"message"`. The `"cod"` key contains the HTTP status code, and the `"message"` key contains the description for what went wrong.

If we then try out the two remaining errors, one for a bad API key and one for a bad request URL, we'll find that this same structure is returned to us, where `"cod"` contains the HTTP status code and `"message"` contains the description of the error.

With this information already we can update our JS to access and print this information, but before we do that, we should always do a little research on the API documentation. Why? There just may be more information on how errors are reported.

Reviewing the API Documentation on Errors

When you are ready to look through the API's documentation to learn how it reports errors, there's a couple places you should look:

1. If the API you are using has multiple APIs, start in the documentation for the specific API you are using. For the OpenWeather API, we're getting current weather data, so I would start by looking in the current weather documentation (https://openweathermap.org/current). Look at any table of contents, and try using `ctrl + f` to search for "error" to see if you can quickly locate the section that goes over error reporting.

2. If you can't find any information about error reporting for a specific API, go one level up. For the OpenWeather API, this would be looking at their webpage that lists all APIs (https://openweathermap.org/api).
3. If you still can't find any information about error reporting, I would look for a FAQ page, or something similar. Some APIs may include a search option to search their API documentation, which you could also try using.
4. If you still can't find any information about error reporting, you'll need to assume the API doesn't have custom error messages when a response is not 200 OK.

If you found in your testing in Postman that the API does in fact return custom error messages, you can choose to incorporate what you've learned from the testing the API or just stick with the `XMLHttpRequest` object's `status` and `statusText` properties to handle reporting errors.

As we noted earlier, each API is different, so the process of locating its documentation on error reporting (if any) and learning how to use it will vary. For the OpenWeather API, it turns out that it has a section dedicated to API errors at the bottom of its FAQ page (https://openweathermap.org/faq), which was hard for me to find. What's more, it doesn't actually show what the API response object for each error looks like.

So, do your best to research the API documentation to find helpful information on errors, and then always try introducing errors into your request via Postman to test how the API responds. When in doubt, stick to using `XMLHttpRequest.status` and `XMLHttpRequest.statusText` to deliver error messages to the user.

Displaying Custom Error Messages from an API

Since the OpenWeather API does include custom error messages, let's briefly look at how we can access that information in our project. We'll update our existing `printError` function to also print the API's error data along with the information from the `XMLHttpRequest` object.

First, we'll need to update our `"loadend"` event listener in the `getWeather` function.

```
request.addEventListener("loadend", function() {
  const response = JSON.parse(this.responseText);
  console.log(response);
  if (this.status === 200) {
    printElements(response, city);
  } else {
    // there's a new argument
    printError(this, response, city);
  }
});
```

Notice that we've added a new argument to the `printError` function: `response`. Now our `printError` function will have access to the API's response (`response`), as well as the `XMLHttpRequest` object (represented by `this`) and the user input (`city`).

Keep in mind that the API will return different information based on whether the API call was successful or not. If we had an error free request with a status of 200 OK, the `response` variable will represent the weather data:

```
{
    "coord": {
        "lon": -122.6762,
        "lat": 45.5234
    },
    "weather": [
        {
            "id": 800,
            "main": "Clear",
            "description": "clear sky",
            "icon": "01d"
        }
    ],
    "base": "stations",
    "main": {
        "temp": 305.23,
        "feels_like": 306.11,
        "temp_min": 303.01,
        "temp_max": 308.14,
        "pressure": 1012,
        "humidity": 43
    },
    "visibility": 10000,
    "wind": {
        "speed": 2.68,
        "deg": 225,
        "gust": 2.68
    },
    "clouds": {
        "all": 6
    },
    "dt": 1658779129,
    "sys": {
        "type": 2,
        "id": 2008548,
        "country": "US",
        "sunrise": 1658753199,
        "sunset": 1658807250
    },
    "timezone": -25200,
    "id": 5746545,
    "name": "Portland",
    "cod": 200
}
```

If there was an error and the status code is anything other than 200 OK, `response` will be set to an error object that looks something like this:

```
{
    "cod": "404",
    "message": "city not found"
}
```

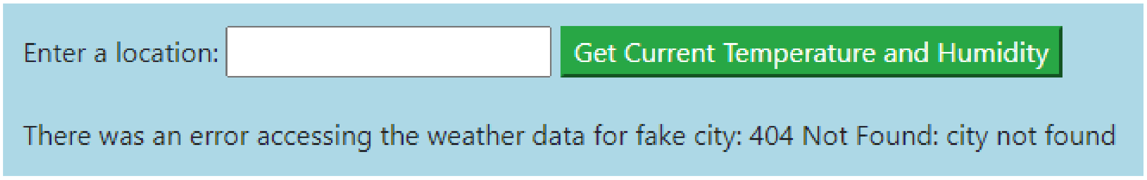
Now it's time to update the `printError` function to have a new parameter `apiResponse`, and to do something with it:

```
function printError(request, apiResponse, city) {
  document.querySelector('#showResponse').innerText = `There was an error accessing the weather data for ${city}: ${request.status} ${request.statusText}: ${apiResponse.message}`;
}
```

Since the `apiResponse` parameter represents the response object from the API, we can access it with object property accessors (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_accessors): `apiResponse.message`.

Now our error messages will print look like this:

```
There was an error accessing the weather data for fake city: 404 Not Found: city not found
```



There was an error accessing the weather data for fake city: 404 Not Found: city not found

Of course, you don't have to stick with this formatting for the error message — as long as you inform the user why there was an error, including the HTTP status code and description, that's sufficient for basic error handling.

Summary

In this lesson we learned how to add basic error handling to our API calls. The goal of basic error handling is to inform the user why an API call went wrong, which includes relaying two key pieces of information: the HTTP status code and the title/description of that status code.

We learned two ways to communicate error messages to the user:

1. By printing the `XMLHttpRequest.status` and `XMLHttpRequest.statusText` to the webpage.
2. By learning how the API structures custom error messages (if it has any) and printing those to the webpage.

Not all APIs have custom error messages, which is why it's important to test out error responses with your API and review its documentation. When APIs do include custom error messages, they are usually more descriptive than reporting the HTTP status code and title from the `XMLHttpRequest` object, which is why it's good to research and use in your code.

To test out your error handling, we recommend these three test cases:

- Using a bad input, like a non-existent city, which causes a "404 Not Found" error.
- Using a bad API key, which causes a "401 Not Authorized" error.
- Using a bad request URL (but a good API key), like messing up the request parameters, which causes a "400 Bad Request" error.

Including basic error handling for API calls is required on this section's independent project, so make sure to practice it during this course section.

Later on in this course section, we'll continue to learn how to handle errors with the new tools we learn to use to handle asynchrony and making API calls. Up next, we'll learn about the DevTools *Network* tab, which we can use to inspect our API calls and test how our error handling deals with one more type of error: a network error.

📂Example GitHub Repo for API Project (https://github.com/epicodus-lessons/section-6-js-api-call-with-webpack)

The above link takes you to a branch within a repo. Make sure that you are referencing the code from the branch called `1_xhr_api_call`. This is the default branch, so running `git clone...` with the URL of the repo home page will automatically clone down the branch called `1_xhr_api_call`. As needed, review the lesson on accessing code from different branches (https://www.learnhowtoprogram.com/intermediate-javascript/object-oriented-javascript/accessing-code-from-different-branches).

Previous (/intermediate-javascript/asynchrony-and-apis/protecting-api-keys)
Next (/intermediate-javascript/asynchrony-and-apis/the-devtools-network-tab)

Lesson 9 of 33
Last updated more than 3 months ago.

disable dark mode

(http://www.epicodus.com)