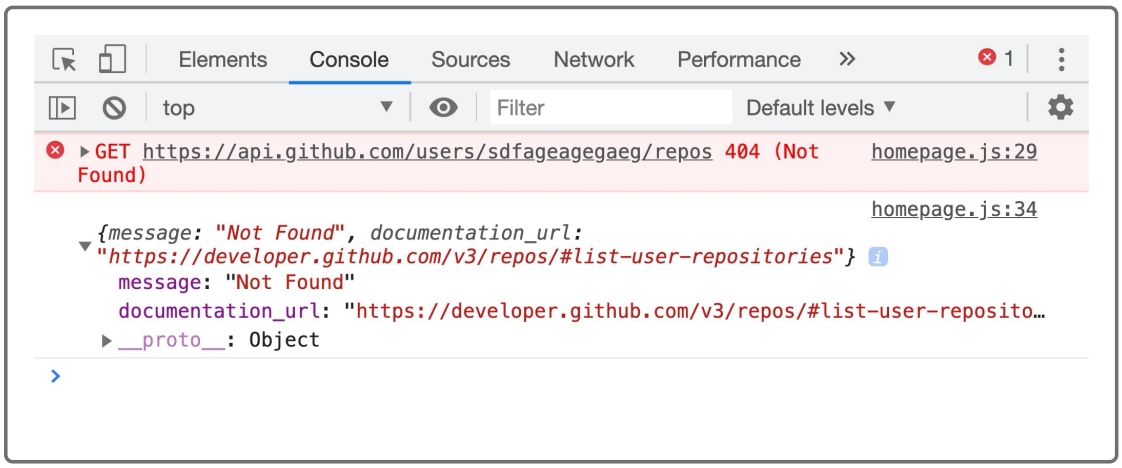
6.2.6 Add Error Handling

Errors are a potential hurdle we can face when working with server-side APIs. Because we're communicating over the internet with another computer, many factors could cause failed responses, including a bad network connection or changes to the API. Let's review possible solutions to deal with these issues.

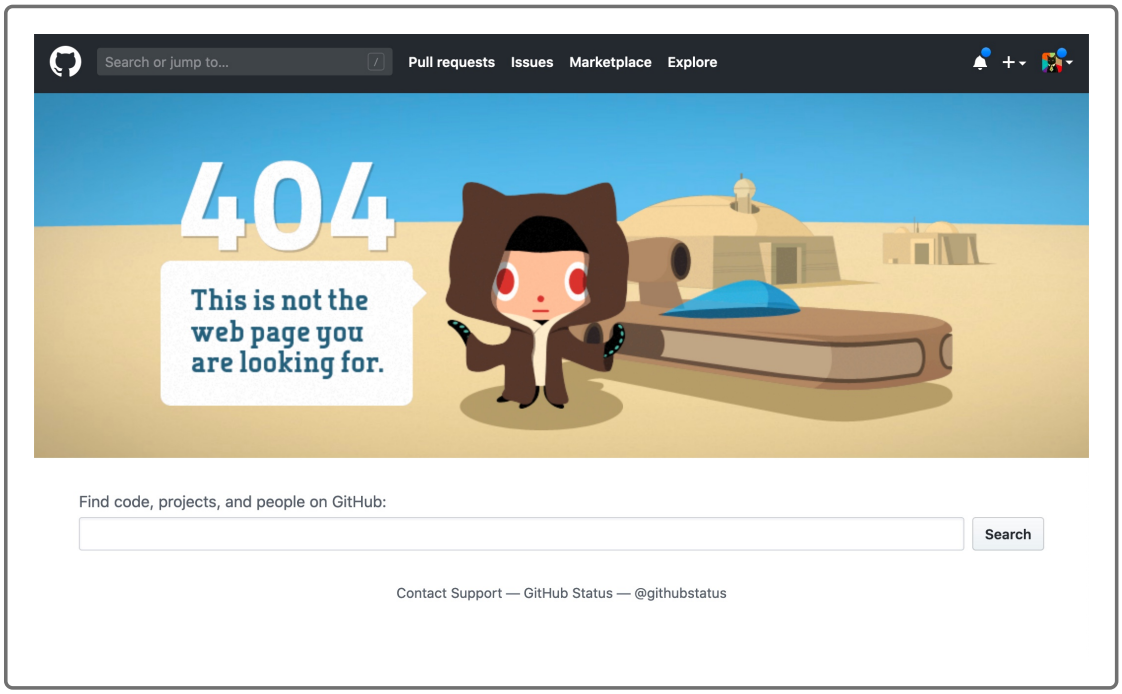
User Not Found Error

A failed response could occur, for example, if we try searching for a user that doesn't exist.

To test this possible error, type in any string of characters and numbers and press Get User. The page will display the name but no repositories. Let's check the console and see what happened:



It's the dreaded 404 error! You've probably seen a 404 error before, most likely when you accidentally entered the wrong address in a URL bar or searched for something that didn't exist, like this image shows:



Now this type of error is happening in your own app! Fear not—the HTTP request 404 status error isn't really a problem at all. As a matter of fact, it's trying to help by explaining that GitHub's API understands your request but that you're looking for something that doesn't exist.

An HTTP request, even when unsuccessful, will always respond with a status code that indicates how the request went. We've now seen two

possible HTTP request status codes in this application: a `200` status and a `404` status. Let's talk about what these status codes mean.

A status code in the 200s means that the HTTP request was successful. Any status code in the 400s indicates that the server received the HTTP request but there's an issue with how we made the request, such as missing information—just as we saw with our search for a user that doesn't exist. We'll also come across status codes in the 500s, which indicate an error with the server we're making a request to or the lack of an internet connection to make the request.

DEEP DIVE

DEEP DIVE

To learn more, read the **MDN web docs on HTTP status codes (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)**.

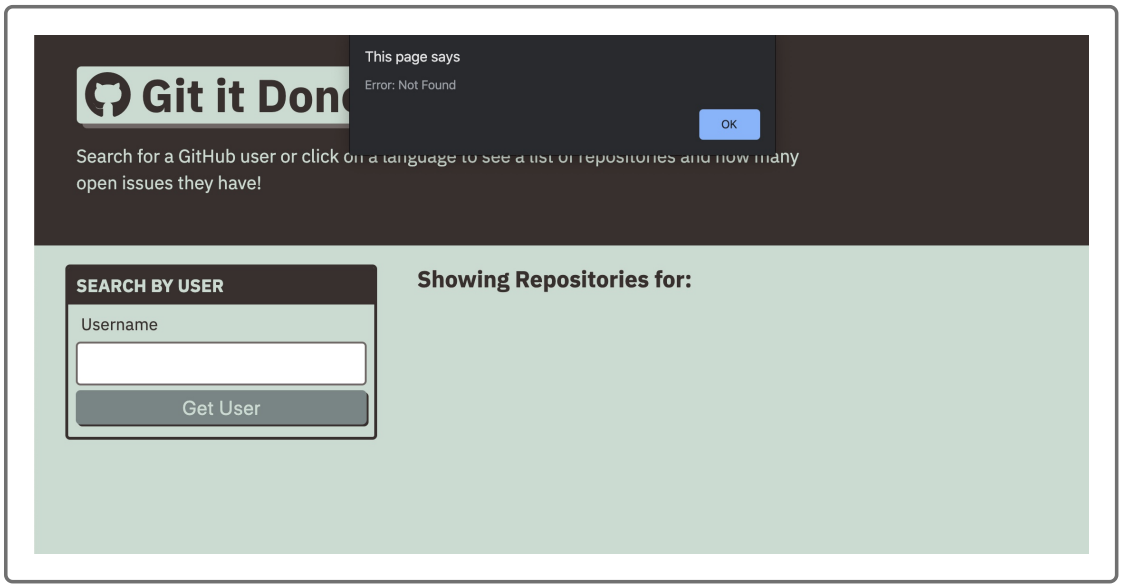
Now, we really can't stop a 404 error unless we prevent users from looking up GitHub accounts that don't exist. That's not realistic, so we can only control how the app reacts to a 404 message.

To handle this error, let's update the `getUserRepos()` function's `fetch()` method to look like this:

```
fetch(apiUrl).then(function(response) {
  if (response.ok) {
    response.json().then(function(data) {
      displayRepos(data, user);
    });
```

```
  } else {
    alert("Error: " + response.statusText);
  }
});
```

Now if we try to search for a user that doesn't exist, the HTTP request will still be made; but instead of an attempt to display the repository data, an alert like this image will appear:



Now it's up to GitHub's API to tell us they couldn't find that user and send a response back. We can check if it was a successful request by using the `ok` property that's bundled in the response object from `fetch()`. When the HTTP request status code is something in the 200s, the `ok` property will be `true`.

If the `ok` property is `false`, we know that something is wrong with the HTTP request, so we can check the `statusText` property to see exactly what the problem is.

Now that we've handled that potential error, let's take a look at another potential issue.

User Has No Repositories

If a GitHub user exists but doesn't have any repos, GitHub will not return a 404—because the user was found but has no repositories to display.

In this situation, the `fetch()` request's response data will still have information about that name, so when we run it through `response.json()` it will still work but will return an empty array. If we just leave the page blank, the user will have no idea if the search function has worked as intended. We should let them know immediately that everything's fine but that no results appeared for that name.

With that in mind, let's update the `displayRepos()` function to check for an empty array and let the user know if there's nothing to display.

At the very top of `displayRepos()`, add the following code:

```
// check if api returned any repos
if (repos.length === 0) {
  repoContainerEl.textContent = "No repositories found.";
  return;
}
```

Now if someone searches for a name without any repositories, we can let the user know that there's nothing to display!

We've handled a couple of potential errors at this point. However, any app that transfers data over the internet—like ours—might experience bugs if the internet goes down or the GitHub server crashes.

Catch Network Errors

While many communities and organizations now have stable, widespread internet access, network connectivity can still cause problems. Wireless

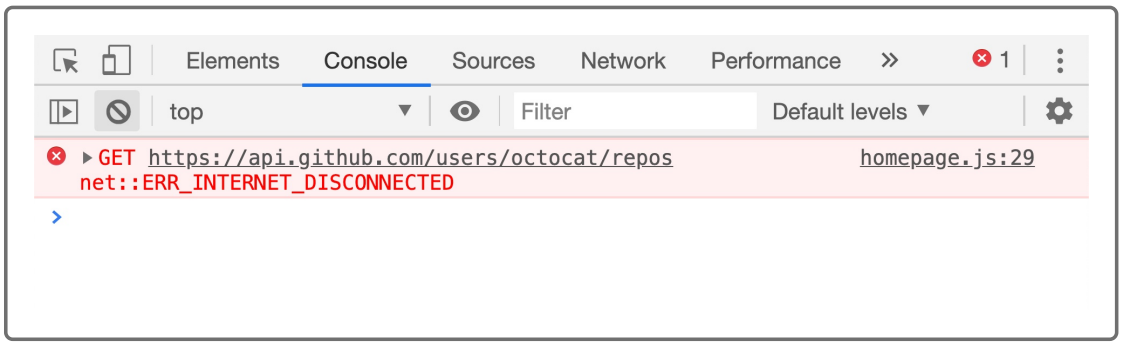
networks can be overloaded; we may find ourselves out of range; and GitHub may even have issues with their API servers' connection. We can't control any of these factors, but we have to ensure that the app responds properly if such an error occurs, to avoid turning off users.

Luckily, a feature built into the Fetch API can help us with connectivity issues. Let's add the following method to the `fetch()` request so that the entire function now looks like this:

```
fetch(apiUrl)
  .then(function(response) {
    // request was successful
    if (response.ok) {
      response.json().then(function(data) {
        displayRepos(data, user);
      });
    } else {
      alert("Error: " + response.statusText);
    }
  })
  .catch(function(error) {
    // Notice this `.catch()` getting chained onto the end of the `.th
    alert("Unable to connect to GitHub");
  });
```

Notice that last part, the `.catch()` method? That's the Fetch API's way of handling network errors. When we use `fetch()` to create a request, the request might go one of two ways: the request may find its destination URL and attempt to get the data in question, which would get returned into the `.then()` method; or if the request fails, that error will be sent to the `.catch()` method.

Requests can fail in a few different ways. To force a failed request for testing purposes, let's turn off the computer's WiFi and disconnect it from the internet. If we do that and try to search for a GitHub user, we'll see an alert stating that we can't connect to GitHub. This is what the console says:



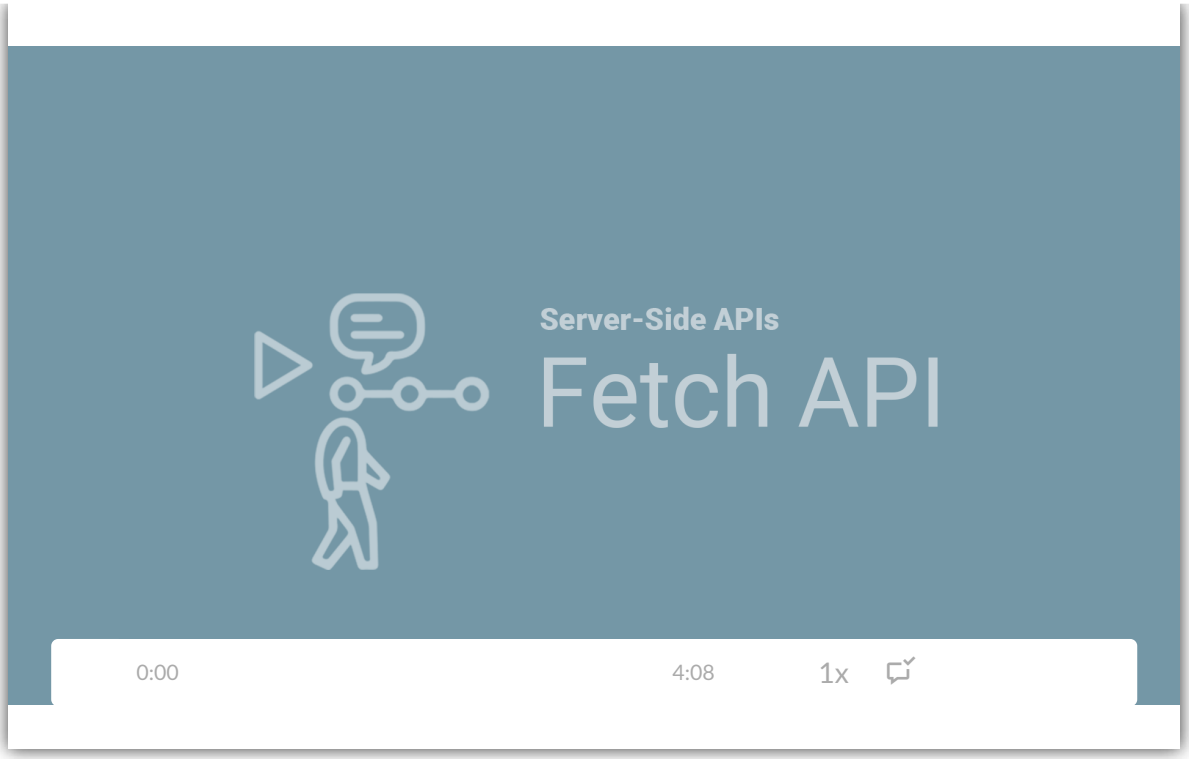
The message explains that there's no internet for us to even make a request. A message like this is helpful, though. If we don't catch the error and display something to the user, they may think the application is broken. Now they know that there's an issue connecting to GitHub and that the issue might be out of our hands.

HIDE PRO TIP

You can also force a failed HTTP request by making a typo in the requested URL.

Great job! You've covered a lot of ground in this lesson, but you'll get plenty of practice capturing input, displaying data, and handling errors often in your career.

For a recap on how to use the Fetch API, watch the following video:



Let's wrap up the first GitHub issue and merge the code into the `develop` branch!