6.3.4 Fetch the API Data

The JavaScript we write in `single.js` will do most of the work on the `single-repo.html` page, so let's add that functionality. On this page, we'll make a request to the GitHub API with a repository's name, and we'll display all the issues associated with that repository.

IMPORTANT

> A repository name includes the creator's username (e.g., octocat or lernantino) because many projects across GitHub might have the same name (e.g., octocat/my-first-app and lernantino/my-first-app).

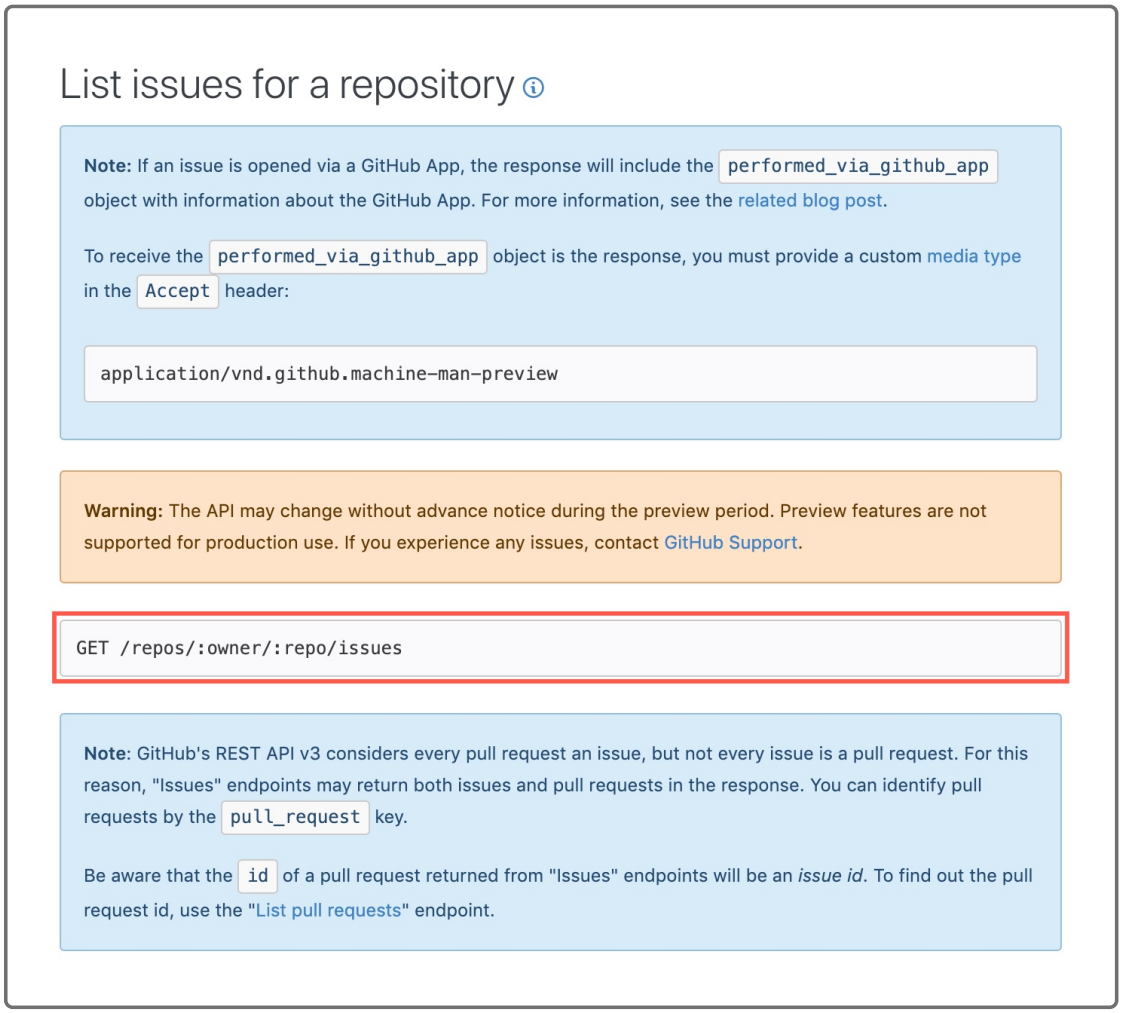
Start by opening `single.js` in VS Code and creating a `getRepoIssues()` function that will take in a repo name as a parameter. For basic testing, `console.log()` the passed repo name as such:

```
var getRepoIssues = function(repo) {
  console.log(repo);
};

getRepoIssues("facebook/react");
```

Navigate back to the `single-repo.html` page in the browser and reload, opening Chrome DevTools to check if the function worked. Great! Let's add in an actual request.

The API endpoint we've been using (`/users/<user>/repos`) doesn't provide enough information for us to display the individual issues for a single repository, so let's look at the **GitHub documentation for issues (https://developer.github.com/v3/issues/)** and get the correct endpoint. Note the section pictured in the following image:



List issues for a repository

Note: If an issue is opened via a GitHub App, the response will include the `performed_via_github_app` object with information about the GitHub App. For more information, see the related blog post.

To receive the `performed_via_github_app` object is the response, you must provide a custom media type in the `Accept` header:

```
application/vnd.github.machine-man-preview
```

Warning: The API may change without advance notice during the preview period. Preview features are not supported for production use. If you experience any issues, contact GitHub Support.

```
GET /repos/:owner/:repo/issues
```

Note: GitHub's REST API v3 considers every pull request an issue, but not every issue is a pull request. For this reason, "Issues" endpoints may return both issues and pull requests in the response. You can identify pull requests by the `pull_request` key.

Be aware that the `id` of a pull request returned from "Issues" endpoints will be an *issue id*. To find out the pull request id, use the "List pull requests" endpoint.

Using the endpoint listed in the documentation, we can format the URL as `https://api.github.com/repos/<repo>/issues`, where `<repo>` encompasses the username and repo name. Note that the documentation says this endpoint will return both issues and pull requests. This is fine; pull requests can still be contributed to!

The documentation might not make this apparent, but we also have the option to append `?direction=asc` to the end of the query URL to specify the sort order. By default, GitHub returns request results in descending order by their created date, meaning that we see newer issues first. The `?direction=asc` option reverses order to return older issues first.

We'll get into these `?` options in more detail in the next lesson. For now, let's focus on using the Fetch API to create an HTTP request to this endpoint. First we'll create a variable to hold the query. So, inside the `getRepoIssues()` function, add the following line:

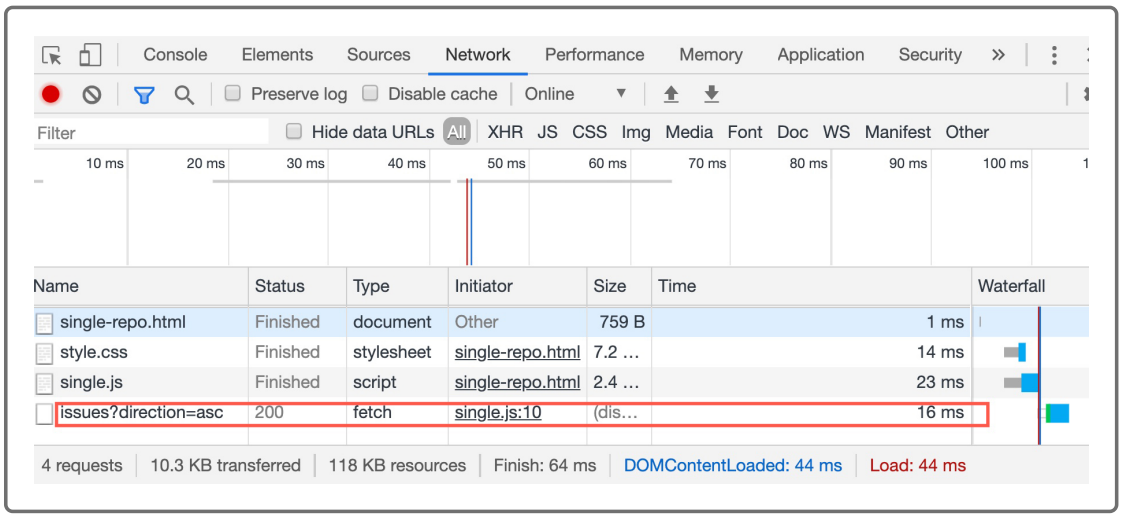
```
var apiUrl = "https://api.github.com/repos/" + repo + "/issues?directi
```

Now let's build out a basic HTTP request to hit this endpoint and check the information returned in the response. Create a request with `fetch()`, and pass in the `apiUrl` variable we created. The `getRepoIssues()` function should resemble the following code block:

```
var apiUrl = "https://api.github.com/repos/" + repo + "/issues?directi

fetch(apiUrl);
```

To view the resulting request, save and navigate back to `single-repo.html` in your browser, making sure to have the Network tab open in Chrome DevTools. Refresh the page, and you should see the HTTP request being made, per the following image:

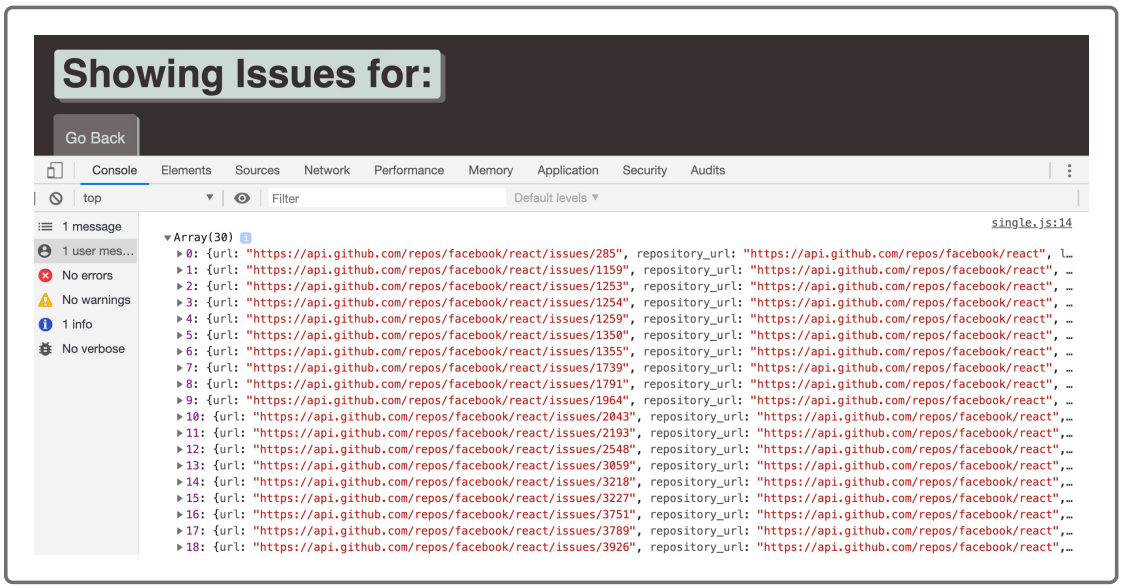


Remember that `fetch()` is asynchronous. We'll have to use its Promise-based syntax to actually access the data contained in the response.

Update the `fetch()` request to receive and handle the server's response:

```
fetch(apiUrl).then(function(response) {
  // request was successful
  if (response.ok) {
    response.json().then(function(data) {
      console.log(data);
    });
  }
  else {
    alert("There was a problem with your request!");
  }
});
```

Notice that we are checking the value of `response.ok`, which indicates a successful request. Inspect the logged `data` parameter in the console, noting some of the properties on the issue objects, as seen in the following image:



You'll need to use those properties to create DOM elements in the next step!