# 6.1.5 Capture Data Returned from the API

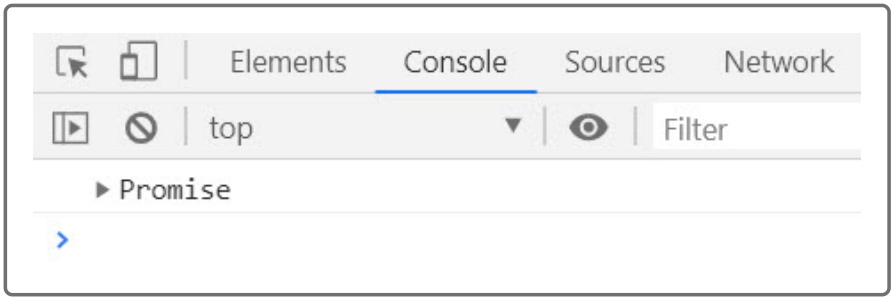
At this point, we've made an HTTP request to the GitHub API using `fetch()`. In turn, GitHub responded with JSON data. Remember that relationship: the request originated from the app, and the response came from GitHub's server.

However, making the request only gets us halfway. We still need to read the data that was returned in the response. You could try console logging whatever the `fetch()` function returns.

In `homepage.js`, update the `getUserRepos()` function to include the following code:

```
var response = fetch("https://api.github.com/users/octocat/repos");
console.log(response);
```

If you check the console, you'll see that the `fetch()` function returned some weird `Promise` object, as the following image shows:




Promises are newer additions to JavaScript that act like more advanced callback functions. We'll dissect Promises in greater detail down the road. For now, note that Promises have a method called `then()`. This method is called when the Promise has been fulfilled.

Update the code inside `getUserRepos()` to look like this:

```
fetch("https://api.github.com/users/octocat/repos").then(function(resp
  console.log("inside", response);
});

console.log("outside");
```

Refresh the browser and watch the console. You'll notice that the second `console.log()` statement prints first: this is an example of **asynchronous** behavior.

What's happening here? Imagine if the GitHub API was running particularly slow and a response didn't come back for several seconds. You wouldn't want the rest of your code to be blocked until then. To avoid that, JavaScript will essentially set aside the fetch request and continue implementing the rest of your code, then come back and run the fetch callback when the response is ready—thus working asynchronously.



## REWIND

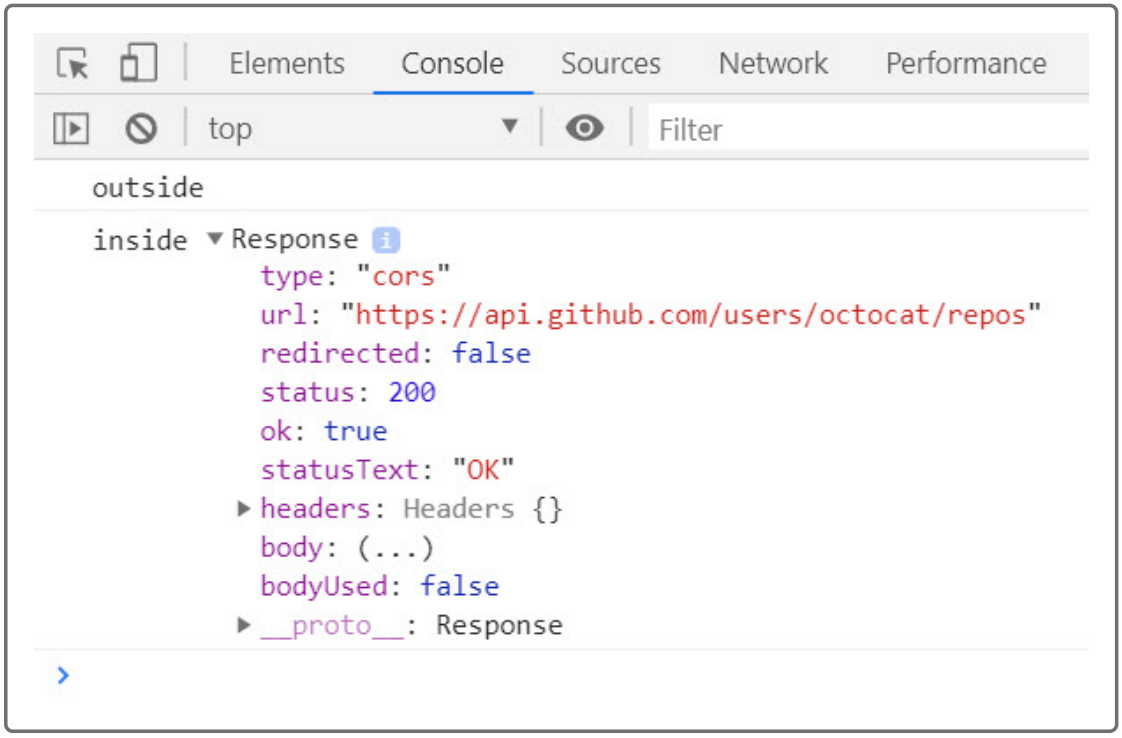
You've come across asynchronous methods in JavaScript before. Event callbacks and timers are both asynchronous in that they don't block the code underneath from running. For example:

```
document.querySelector("#submit-btn").addEventListener("cli
 // only called when user clicks the button
 console.log("hello 1");
});

setTimeout(function() {
 // called after 5 seconds, regardless if button was clicke
 console.log("hello 2");
}, 5000);

// called immediately on page load
console.log("hello 3");
```

In this case, the third `console.log()` will print first because it's not dependent on any asynchronous actions.

This kind of asynchronous communication with a server is often referred to as **AJAX**, which stands for **Asynchronous JavaScript and XML**. The XML in this term refers to an old-fashioned way of formatting data. XML has been largely replaced by JSON, but the name has persisted.

Now that we've received a response, let's see if we can access the JSON data returned from this HTTP/AJAX request. Take a closer look at the `response` object that we logged. In the console, expand the object to display the following image:




Studying this object should tell us a couple of things. First, the `url` property confirms where the response came from. Second, because the `status` property has a value of 200 (which means success), we know that the request went through successfully. But where's the data? Where's the array of repos?

Before you can use the data in your code, you need to format the response. Update your `fetch()` logic to look like this:

```
fetch("https://api.github.com/users/octocat/repos").then(function(resp
  response.json().then(function(data) {
    console.log(data);
  });
});
```

Refresh the browser, and you'll now see the array logged in the console.

Notice how the `response` object in your `fetch()` logic has a method called `json()`. This method formats the response as JSON. But

sometimes a resource may return non-JSON data, and in those cases, a different method, like `text()`, would be used.

The `json()` method returns another Promise, hence the extra `then()` method, whose callback function captures the actual data. This nested `then()` syntax might confuse you at first, but the more API calls you make, the more comfortable you'll start to feel with it.

## LEGACY LORE

Developers can talk to server-side APIs by means other than the Fetch API. For example, the `XMLHttpRequest` object has been a part of JavaScript for a long time. However, its methods for handling responses aren't very intuitive.

Libraries like jQuery have sought to make using `XMLHttpRequest` easier by wrapping up its complexities in a single method: `$.ajax()`. jQuery even used a Promise-like syntax by chaining a `done()` method to the end. Consider the following code example:

```
$.ajax("https://api.github.com/users/octocat/repos").done(function(data)
 console.log(data);
});
```

Of course, the JavaScript engine has since caught up to jQuery, and many developers prefer to use the built-in Fetch API now instead of loading up a third-party library just for its `$.ajax()` shorthand.

In any case, we now have the data we want, but remember that we've hardcoded the request to Octocat's account. Up next, we'll make the `getUserRepos()` function dynamic.