6.4.4 Read and Use Data from Query Parameter

The page `single-repo.html` can still load properly even though the query string was appended, because the browser interprets the query string as optional information and ignores it as a result.

In this step, a browser-based tool called the Location Web API will help us retrieve that query parameter.

DEEP DIVE ▲

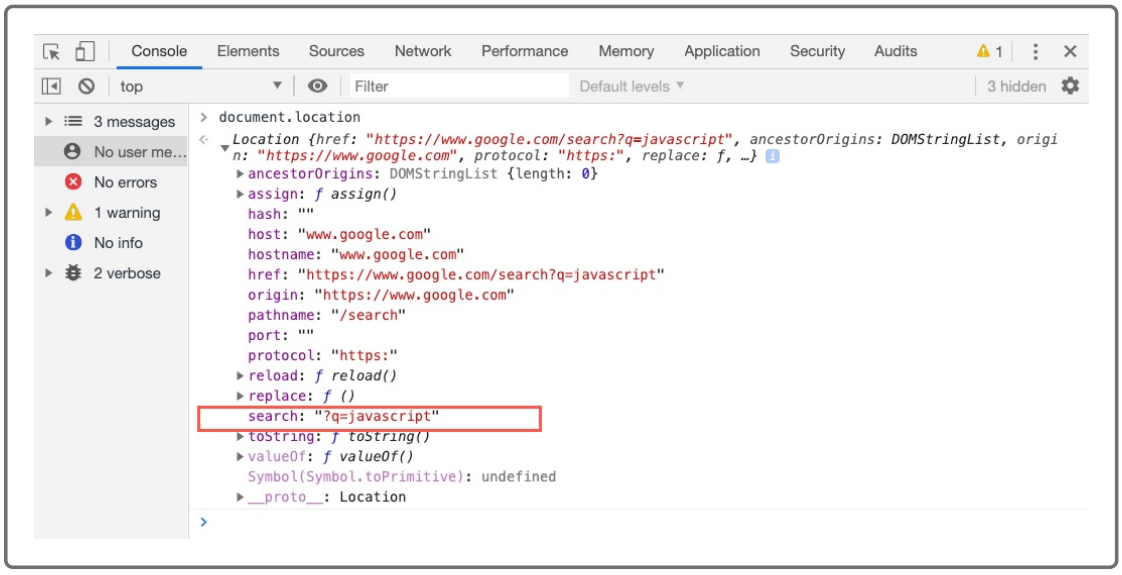
DEEP DIVE

For more information, read **the MDN web docs on the Location Web API (https://developer.mozilla.org/en-US/docs/Web/API/Location)**.

This API is a property of the `document` object we used previously to query elements on the page. In this instance we'll use `document.location` to

locate the query string.

First let's go to the browser, open the Console tab in DevTools, type `document.location`, and press `Enter`. Expand the logged object to see what's shown in the following image:



As we can see in the preceding image, the `location` object contains a few methods and properties that hold information about the URL of the webpage. Of particular interest will be a property called `search`, highlighted in the image shown, because it contains the query string.

Now let's type `document.location.search` directly into the command prompt in DevTools. This will return only the query string portion of the URL, as follows: `?repo=microsoft/activities`.

Excellent work! We've loaded `single-repo.html` with the query string in the URL and accessed it with JavaScript. In the next step, we'll trim this string even further to get the exact data we need from it.

Use the Split Method to Extract the Query Value

Having a string conveniently allows us to use a variety of string methods to retrieve the data, namely the `split()` method. We can use string

methods such as `split()` on all strings in JavaScript, not just query strings. The `split()` method splits a string into an array of substrings and returns the new array. The argument passed into the `split()` method will be used as the separator between those substrings.

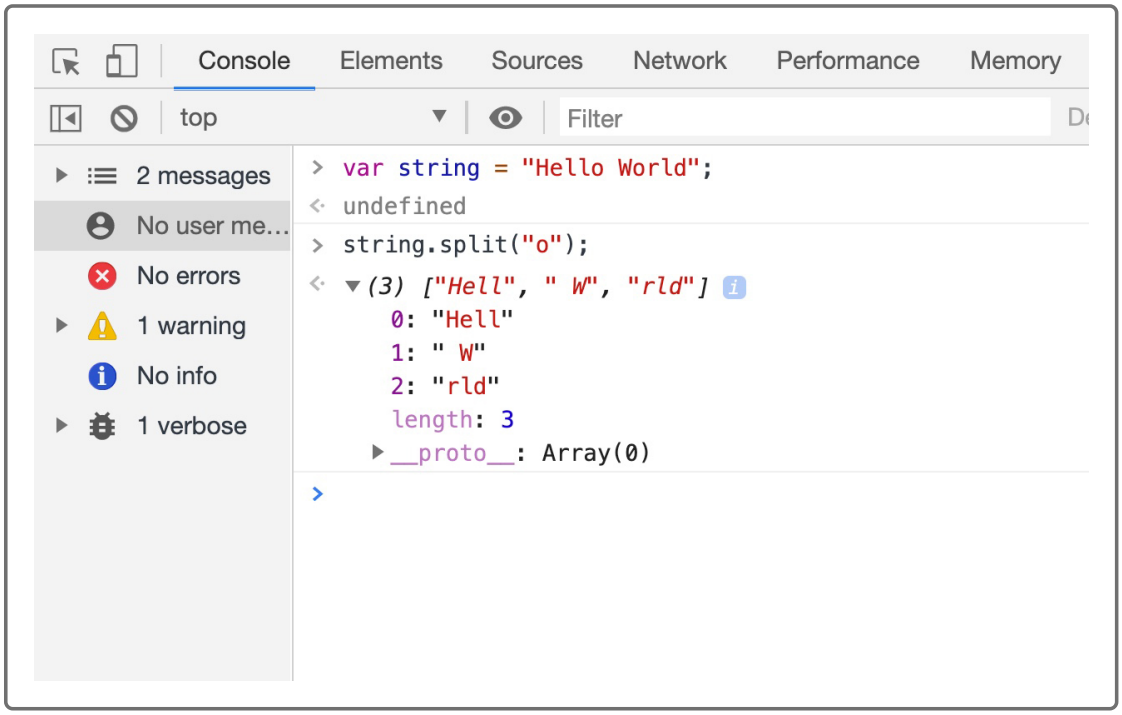
For a quick exercise to demonstrate how this works, navigate to your browser and open the console in DevTools. In the console prompt, type the following expression:

```
var string = "Hello World";
```

Now let's use the `split()` method and pass `o` in as the argument and separator, as shown here:

```
string.split("o");
```

The console will display the result shown in the following image:



The preceding figure shows that the split results in an array with three elements: `"Hell"`, `" W"`, and `"rld"`. Notice how each element of the array corresponds to the letters in the string before or after the letter `"o"`. The `"o"` is the separator that defines where the substrings begin and end, but it isn't added to the array itself.

IMPORTANT

> If an empty string ("") is used as the separator, the string is split between each character. Also, the `split()` method doesn't change the original string.

With these tools available, let's open `single.js` and extract the query value from the query string in the API call function `getRepoIssues()`.

The first step will be to create a new function, called `getRepoName()`, near the top of the file. Remember to place the function call at the bottom of the file as well. The function call will look like this: `getRepoName()`.

Let's also remove the hardcoded function call for `getRepoIssues()`, located at the bottom of the `single.js` file.

In the next step, we'll add to the `getRepoName()` function by using the `location` object and `split()` method to extract the repo name from the query string. Let's start by assigning the query string to a variable:

```
var queryString = document.location.search;
```

Once we have the `queryString`, which currently evaluates to `?repo=microsoft/activities`, how do we break the string apart into the right substrings so that we can isolate the piece we need (`microsoft/activities`)? In other words, which character do we need to pass into the `split()` method to use as the separator?

If you guessed the `=` character, you're correct! By splitting on the `=`, you'll end up with an array with two elements. Using bracket notation, which index number do you need to get the second element from the array?

Remember, indexes start at zero, so we'll need to use `[1]` to get the second element. With that in mind, let's add the following expression to the `getRepoName()` function in the `single.js` file:

```
var repoName = queryString.split("=")[1];
console.log(repoName);
```

Let's save the files and load the `index.html` page in the browser. We need to enter a username and choose a repo. In this example, `microsoft/activities` was used. You should see something like the following image once you open the console:

This image shows that we succeeded in isolating the repo name from the query string.

For the next step, we'll pass the `repoName` variable into the `getRepoIssues()` function, which will use the `repoName` to fetch the related issues from the GitHub API issues endpoint.

Try this yourself and place the function call in the `getRepoName()` function.

Now let's add the repo name to the header of the page. We already have a `<span id="repo-name">` container defined in the HTML. We'll need a DOM

reference to be able to update it.

At the top of `single.js`, add the following:

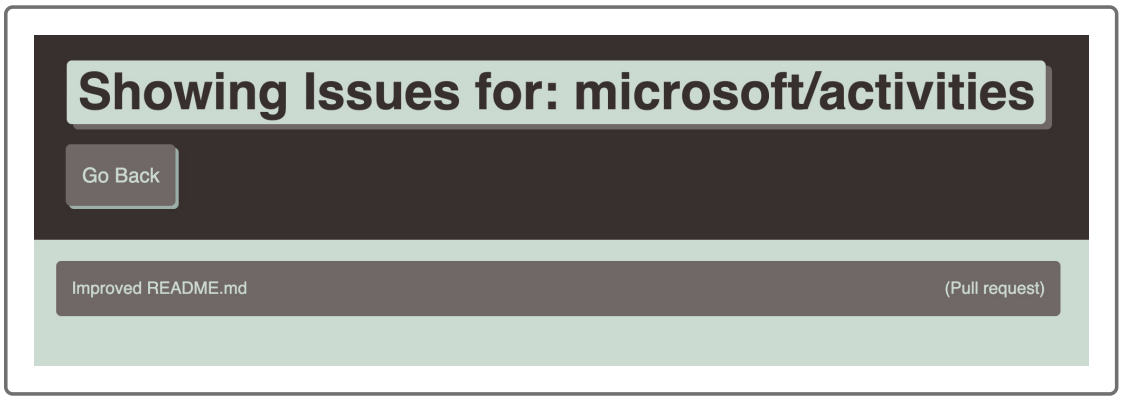
```
var repoNameEl = document.querySelector("#repo-name");
```

In the `getRepoName()` function, use this variable to update the element's text content as such:

```
var getRepoName = function() {
  var queryString = document.location.search;
  var repoName = queryString.split("=")[1];
  getRepoIssues(repoName);
  repoNameEl.textContent = repoName;
}
```

Save your work and load `index.html` in the browser to test if the API call is now dynamic.

The browser should display the issues related to the repo and dynamically call the API. You can test this by selecting various repos and comparing the issues that are rendered. The results should look similar to this image:



The issues page now shows the single issue associated with the selected repository. Return to the search results and click on different repos with open issues to test if the API call is now dynamic.

Great job! The `single-repo.html` page is dynamically rendering the issues based on the repo selected.

The app works just as Amiko has requested. However, to make the app robust and durable, we should add error handling. This means that we should not only make the application work for people who use it as intended but also guide those users who don't.

Moving on, we'll tackle these two main sources of errors:

- A user tries to access the `single-repo.html` page without providing a repository name.
- The request to the GitHub API is unsuccessful (due to server errors or an improperly formatted repository name).