ecx.io Frontend Bootcamp

—

Javascript Day 08
18.08.2021.

ecx.io
an IBM Company

IBM iX

Agenda

- Templates
- URL/Location
- Async JS

Templates

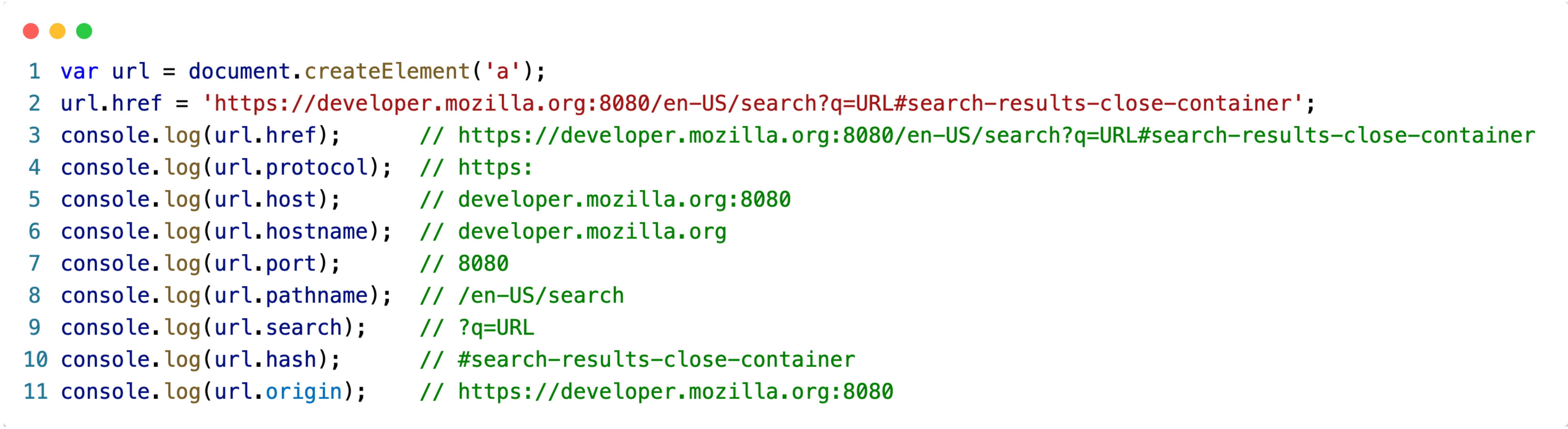
Handlebars & Ractive

Take Friday task from Day 6 and refactor it to use handlebars or ractive.

Location/URL

Location

The Location interface represents the
location (URL) of the object it is linked to.

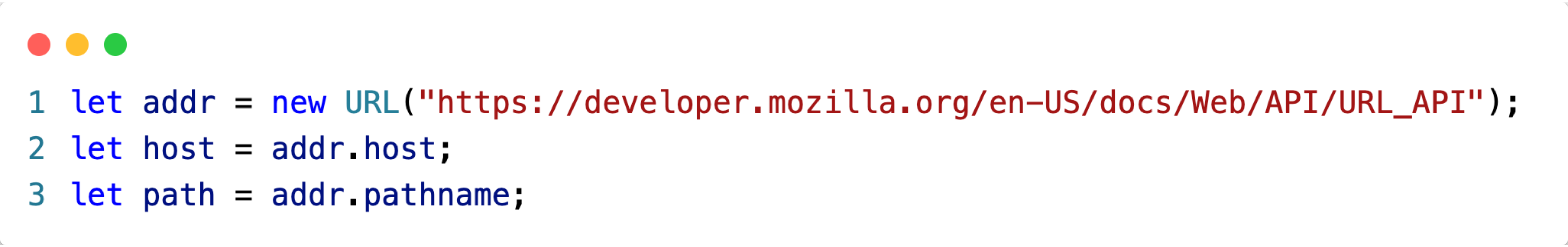


```
var url = document.createElement('a');
url.href = 'https://developer.mozilla.org:8080/en-US/search?q=URL#search-results-close-container';
console.log(url.href);      // https://developer.mozilla.org:8080/en-US/search?q=URL#search-results-close-container
console.log(url.protocol);  // https:
console.log(url.host);      // developer.mozilla.org:8080
console.log(url.hostname);  // developer.mozilla.org
console.log(url.port);      // 8080
console.log(url.pathname);  // /en-US/search
console.log(url.search);    // ?q=URL
console.log(url.hash);      // #search-results-close-container
console.log(url.origin);    // https://developer.mozilla.org:8080
```

URL API

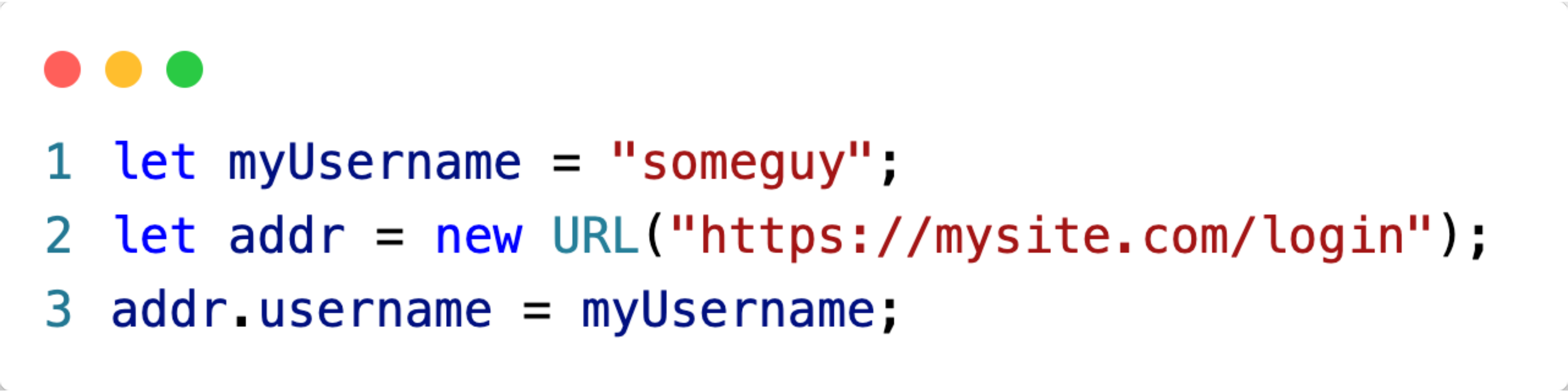
The URL API is a component of the URL standard, which defines what constitutes a valid Uniform Resource Locator and the API that accesses and manipulates URLs.

Accessing URL components



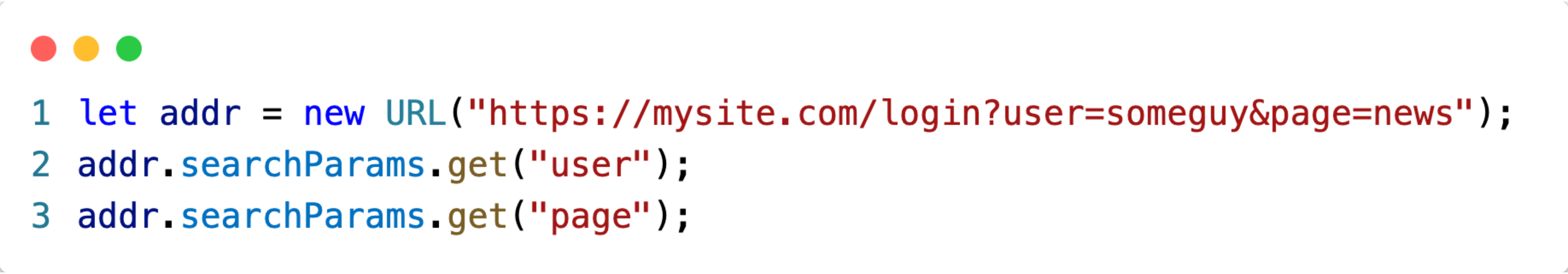
```
let addr = new URL("https://developer.mozilla.org/en-US/docs/Web/API/URL_API");
let host = addr.host;
let path = addr.pathname;
```

Changing the URL



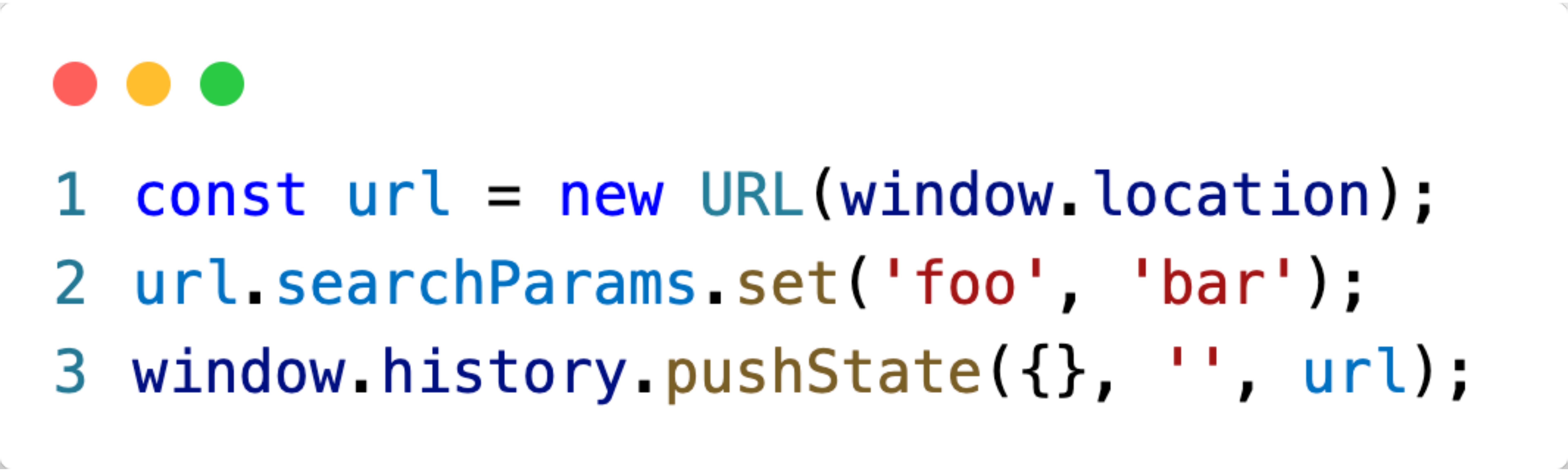
```
let myUsername = "someguy";
let addr = new URL("https://mysite.com/login");
addr.username = myUsername;
```

Queries



```
let addr = new URL("https://mysite.com/login?user=someguy&page=news");
addr.searchParams.get("user");
addr.searchParams.get("page");
```

Change a query parameter



```
const url = new URL(window.location);
url.searchParams.set('foo', 'bar');
window.history.pushState({}, '', url);
```

Exercise

To the Friday task that we worked on in previous topic add logic that will add id of the clicked the card to the url.

Asynchronous JavaScript

Synchronous JavaScript

Normally, a given program's code runs straight along, with only one thing happening at once. If a function relies on the result of another function, it has to wait for the other function to finish and return, and until that happens, the entire program is essentially stopped from the perspective of the user.

When a web app runs in a browser and it executes an intensive chunk of code without returning control to the browser, the browser can appear to be frozen. This is called blocking; the browser is blocked from continuing to handle user input and perform other tasks until the web app returns control of the processor.

Blocking Example

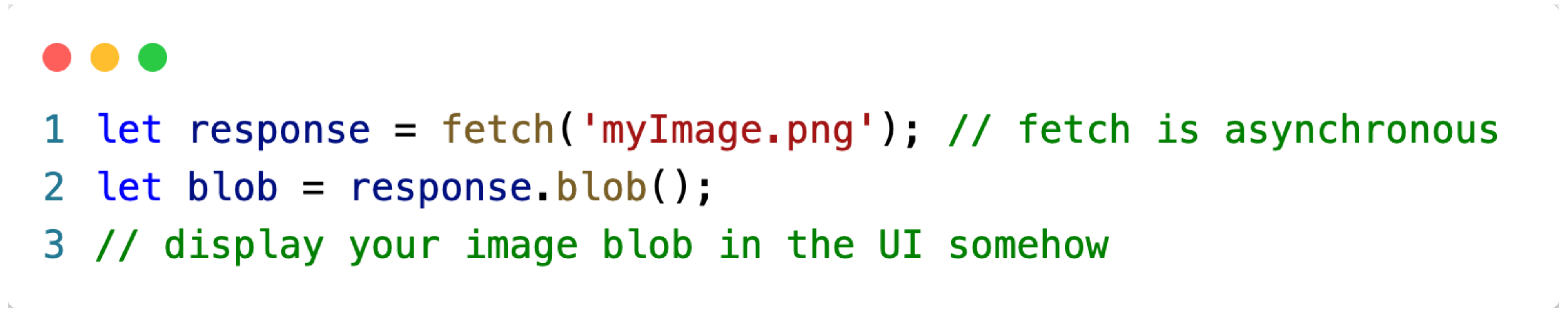
https://mdn.github.io/learning-area/javascript/asynchronous/introducing/simple-sync.html

https://mdn.github.io/learning-area/javascript/asynchronous/introducing/basic-function.html

Asynchronous JavaScript

Many Web API features now use asynchronous code to run, especially those that access or fetch some kind of resource from an external device, such as fetching a file from the network, accessing a database and returning data from it, accessing a video stream from a web cam, or broadcasting the display to a VR headset.

There are two main types of asynchronous code style you'll come across in JavaScript code, old-style callbacks and newer promise-style code. In the below sections we'll review each of these in turn.



```javascript
let response = fetch('myImage.png'); // fetch is asynchronous
let blob = response.blob();
// display your image blob in the UI somehow
```

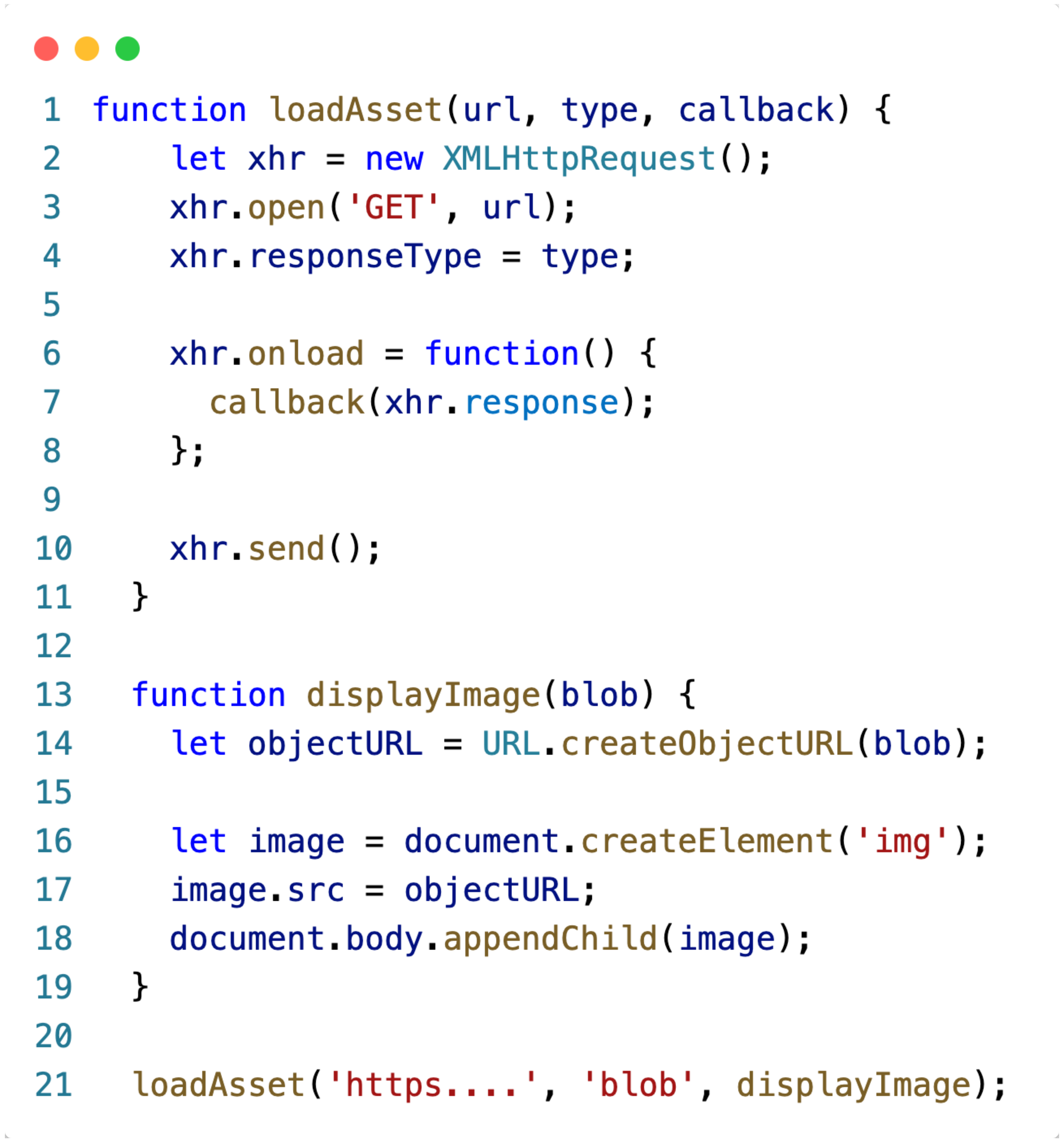
Async callbacks

Async callbacks are functions that are specified as arguments when calling a function which will start executing code in the background. When the background code finishes running, it calls the callback function to let you know the work is done, or to let you know that something of interest has happened. Using callbacks is slightly old-fashioned now, but you'll still see them in use in a number of older-but-still-commonly-used APIs.

An example of an async callback is the second parameter of the addEventListener() method.

Async callbacks

Another example that loads a resource via the XMLHttpRequest API



```javascript
function loadAsset(url, type, callback) {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.responseType = type;

    xhr.onload = function() {
      callback(xhr.response);
    };

    xhr.send();
  }

  function displayImage(blob) {
    let objectURL = URL.createObjectURL(blob);

    let image = document.createElement('img');
    image.src = objectURL;
    document.body.appendChild(image);
  }

  loadAsset('https....', 'blob', displayImage);
```

Exercise

Fetch data using XMLHttpRequest
from https://jsonplaceholder.typicode.com/todos
with type json.
One callback function should return first 5 todos.

Second callback function should return title of the first todo.

Promise

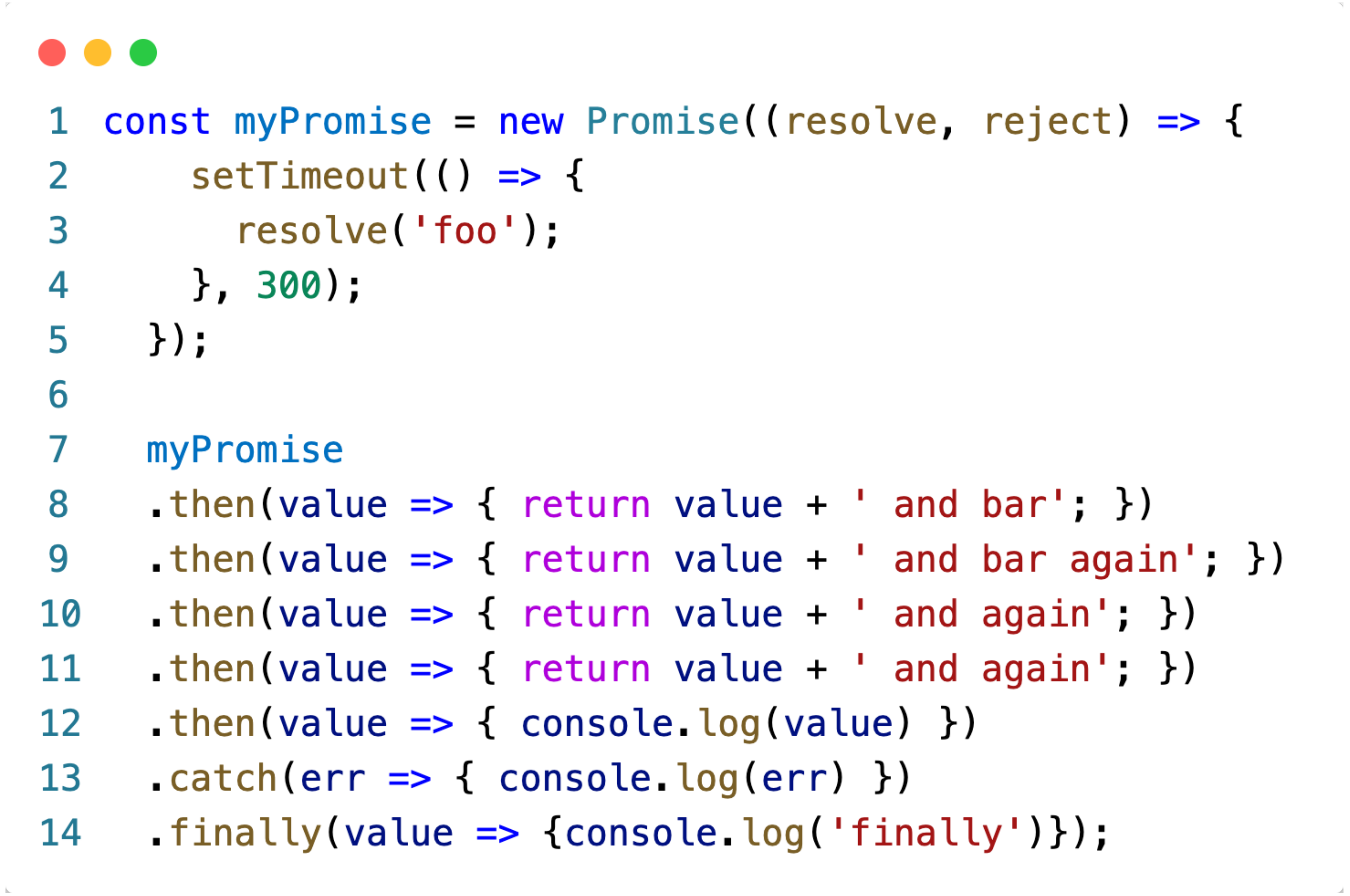
The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.

A Promise is in one of these states:

- pending: initial state, neither fulfilled nor rejected.
- fulfilled: meaning that the operation was completed successfully.
- rejected: meaning that the operation failed.

Promise



```javascript
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
   resolve('foo');
  }, 300);
 });

 myPromise
 .then(value => { return value + ' and bar'; })
 .then(value => { return value + ' and bar again'; })
 .then(value => { return value + ' and again'; })
 .then(value => { return value + ' and again'; })
 .then(value => { console.log(value) })
 .catch(err => { console.log(err) })
 .finally(value => {console.log('finally')});
```

Exercise

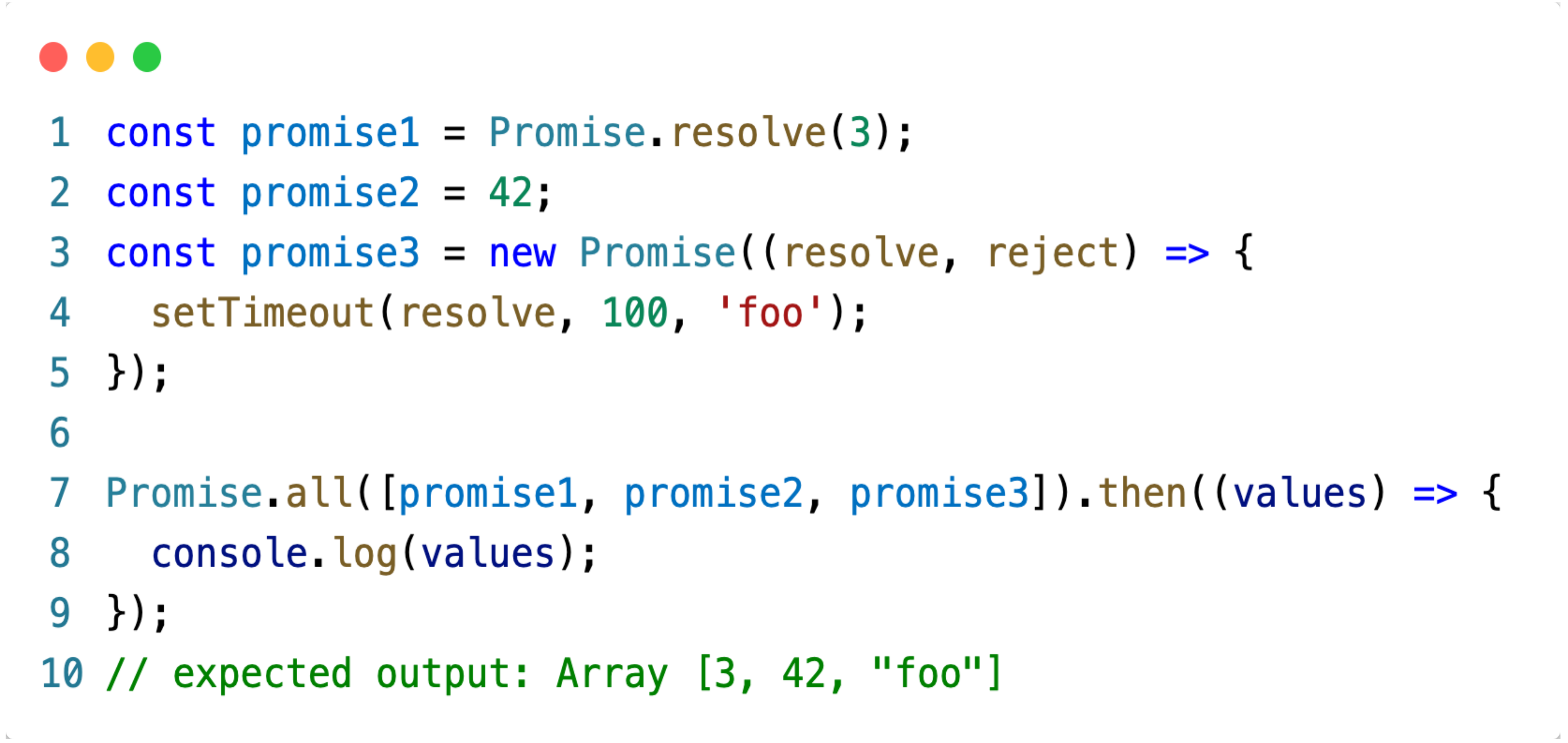
Write promise that will output success message after 500ms.

Exercise

What is the output?

Promise.all()

The Promise.all() method takes an iterable of promises as an input, and returns a single Promise that resolves to an array of the results of the input promises. This returned promise will resolve when all of the input's promises have resolved, or if the input iterable contains no promises. It rejects immediately upon any of the input promises rejecting or non-promises throwing an error, and will reject with this first rejection message / error.



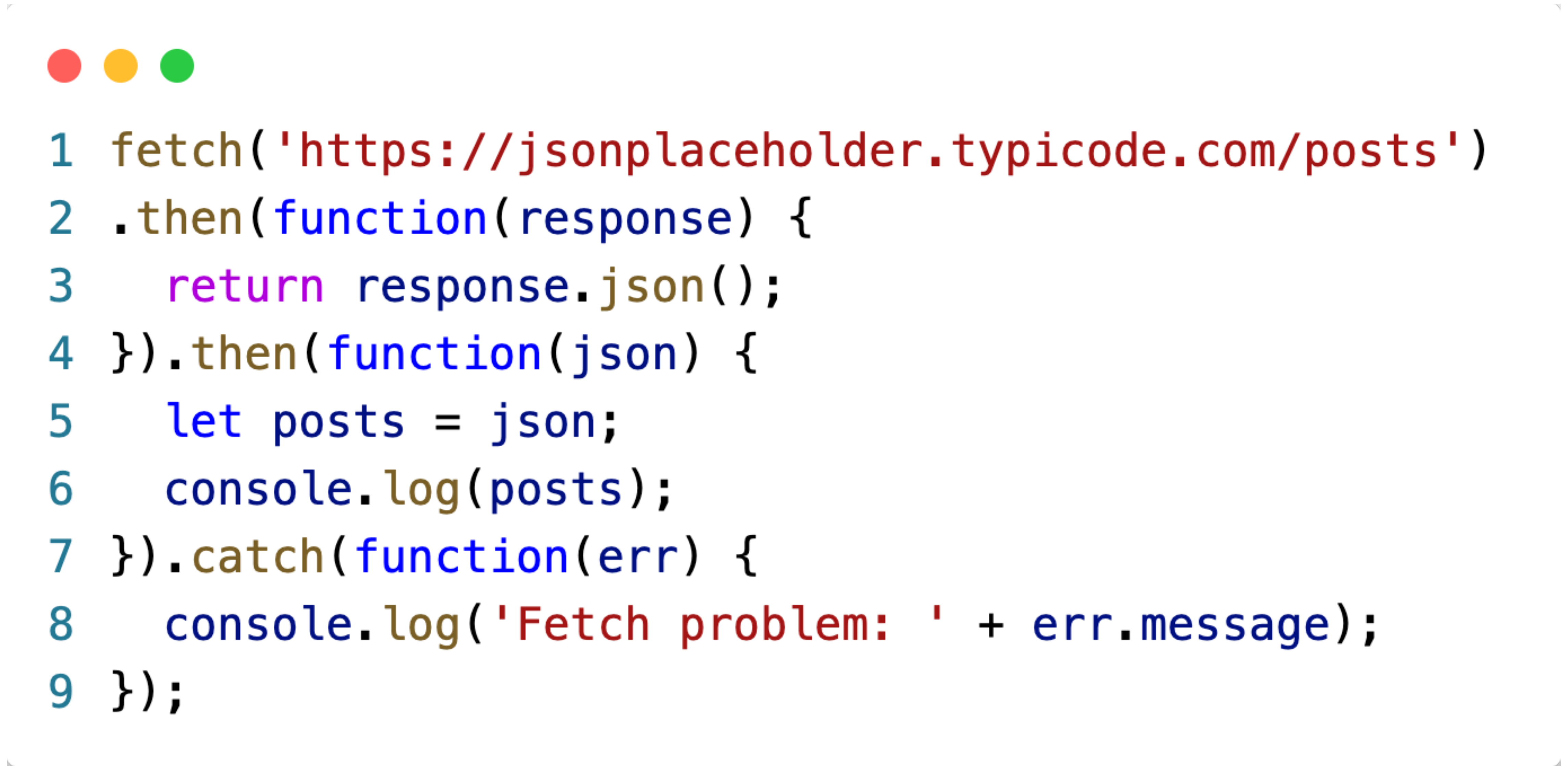
```javascript
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log(values);
});
// expected output: Array [3, 42, "foo"]
```

Fetch API

The fetch() method of the WindowOrWorkerGlobalScope mixin starts the process of fetching a resource from the network, returning a promise which is fulfilled once the response is available. The promise resolves to the Response object representing the response to your request. The promise does not reject on HTTP errors — it only rejects on network errors. You must use then handlers to check for HTTP errors.

`*fetch()*` API is basically like a modern, more efficient version of XMLHttpRequest.

Fetch API



```
fetch('https://jsonplaceholder.typicode.com/posts')
.then(function(response) {
  return response.json();
}).then(function(json) {
  let posts = json;
  console.log(posts);
}).catch(function(err) {
  console.log('Fetch problem: ' + err.message);
});
```

Exercise

Refactor friday task to get results from the jsonplaceholder API.

Exercise

Get users
from https://jsonplaceholder.typicode.com/users
and after that get all posts from the user Ervin Howell.
To get posts for user use this url with correct userId: https://jsonplaceholder.typicode.com/posts?userId=xx

Thank you

ecx.io
an IBM Company