Exploration — Asynchronous Programming: Motivation

Introduction

Asynchronous programming is an essential feature of modern JavaScript. It is widely used to build scalable, performant web apps. Asynchronous features of JavaScript are used in both client-side and server-side programs. In this exploration, we motivate the need for asynchronous programming. In the next exploration, we will study JavaScript features that help write asynchronous code.

Synchronous JavaScript

By default, JavaScript code run synchronously in a single thread. What does that mean? Let us break it down:

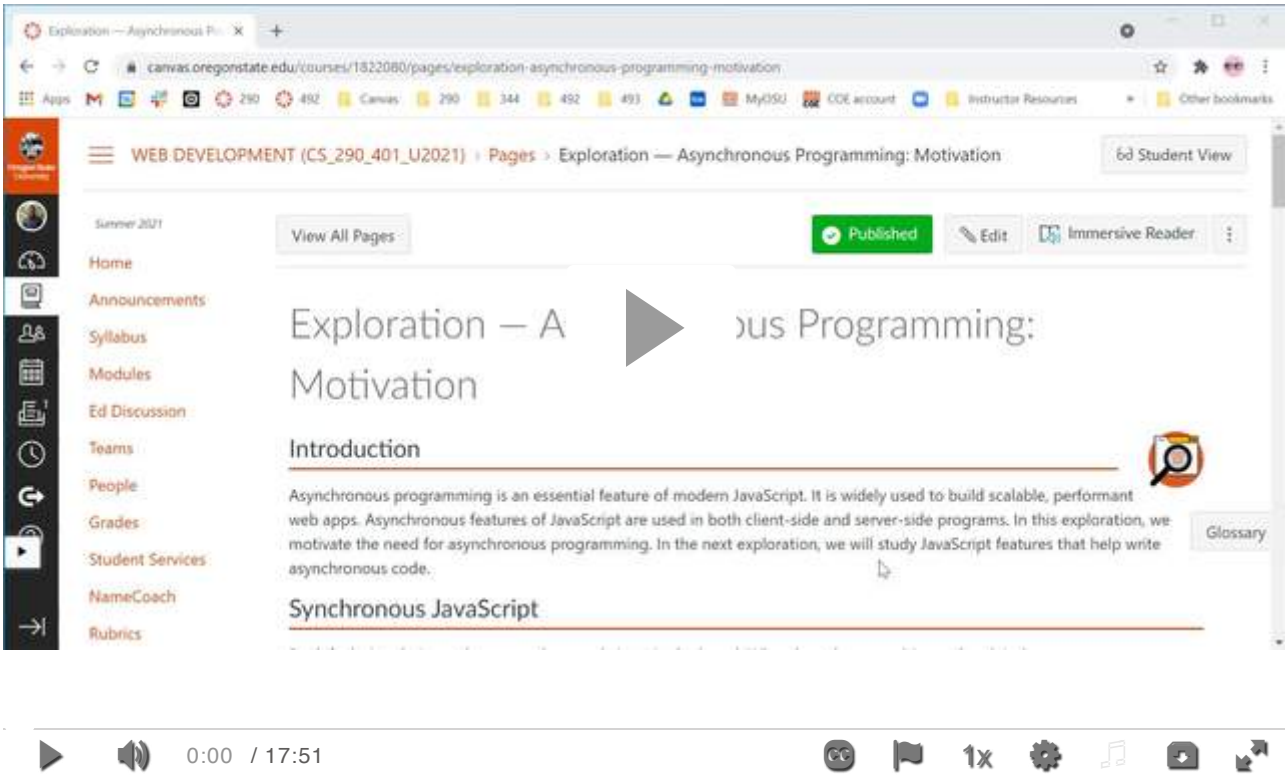
- Synchronous: Each function runs to completion before any other code is executed in the program.
  - For example, when one function `X` calls another function `Y`, nothing happens in `X` until the function `Y` completes its execution and returns.
- Single Threaded: Only one line of code can be executed at any given time.

So in JavaScript, each line of code can potentially **block** the entire program. Blocking means that the entire program is halted until that line of code finishes executing. This would be huge problem when using JavaScript to render webpages. For example, many webpages contain multiple images which the browser loads by sending HTTP requests over the network. Now imagine that the request to load each image blocked all the processing, including preventing the user from interacting with the webpage. Only when one image was loaded, the request for the next image would be sent, and so on. This would surely frustrate the user!

As opposed to this, **asynchronous functions are non-blocking.** As soon as an asynchronous function is called, the next line of code in the calling function can be executed.

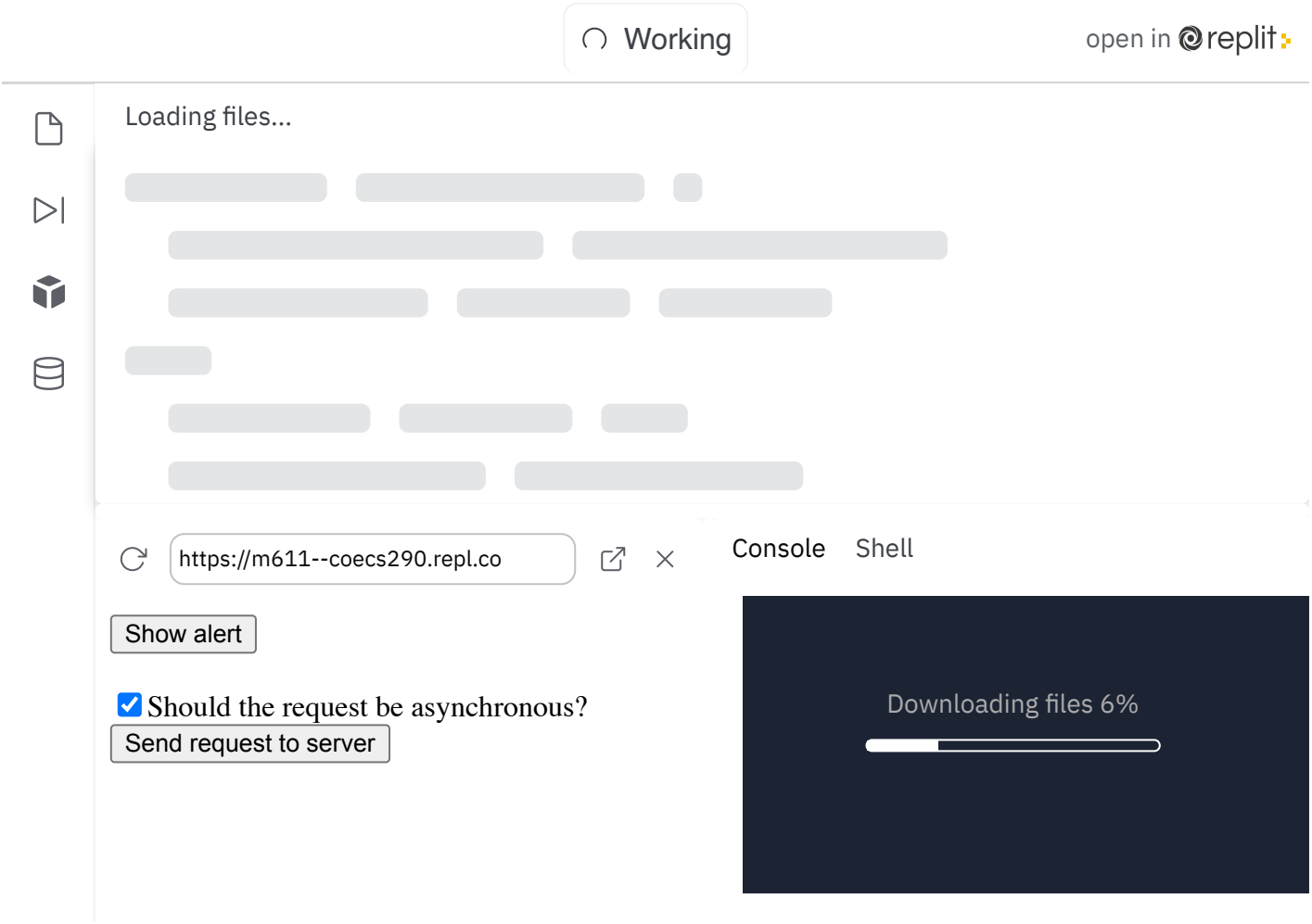
- For example, consider a function `X` calls an asynchronous function `Z`. As soon as `Z` is called, the next line in `X` can be executed and `X` does not have to wait for `Z` to complete. The asynchronous function `Z` is executed by the JavaScript engine separate from the main thread executing the function `X`.

When the asynchronous function completes, its results can be accessed by the main thread using callbacks or via the newer features of Promises and await/async.



Example: Blocking JavaScript

We illustrate the issue of blocking JavaScript via a simple web app. In this web app we simulate a server-side operation that takes a long time. We then study the impact of making a synchronous HTTP request for this operation from our JavaScript code running in the browser. We contrast this behavior from the behavior of the webpage when the request is sent asynchronously.



Let us examine the files that comprise the web app:

- `index.js`
  - This is our server program written using Node and Express.
  - We have been naming our server programs `server.js`. However, this program is named `index.js`, instead of `server.js` to facilitate running the app in the repl.
  - The server serves the static files from the `public` directory.
  - It has a route handler for GET requests for the URL `/longop` which calls the function `doLongOp()`.
  - The function `doLongOp()` simulates an operation that takes a long time by just running a loop a very large number of times and then returns a value from an array. This value is sent back to the client in the HTTP response.
- `index.html`
  - This is the Home Page of our web app.
  - The file is placed under the `public` directory.
  - The Home Page has two buttons, one shows an alert and the other submits a form.
  - The form contains a checkbox which controls whether the client-side script `ajax.js` will send a synchronous or an asynchronous HTTP request to the URL `/longop` in the server.
- `ajax.js`

- This script is loaded by `index.html` and runs in the browser.
- The script registers the function `callServer()` to handle submission of the form in `index.html`.
- The function `getData()` makes the actual HTTP request to the server and is called by `callServer` two times.
- The HTTP request is sent using the function `XMLHttpRequest()`. The details of this function are not important, other than the fact that it can send a request synchronously or it can send a request asynchronously which allows execution to continue .
- Regardless of whether `XMLHttpRequest()` sends the request synchronously or asynchronously, the response is processed by registering callback handlers on the events `load` and `error`.
  - When the response is received the event `load` is raised.
  - If the request errors out without receiving a response, e.g., because of network issues, then the event `error` is raised.
- The callback function registered for the `load` event updates the webpage by adding the text sent by the response in a new element in the DOM tree.

The program has the following behavior:

- When we uncheck the checkbox for “Should the request be asynchronous?” and then submit the form:
  - The HTTP request is sent synchronously.
  - The webpage becomes non-responsive until the response is received.
  - Clicks on the button “Show alert” do not show an alert window until the response is received.
- When we check the checkbox for “Should the request be asynchronous?” and then submit the form:
  - The HTTP request is sent asynchronously.
  - The webpage remains responsive during the time the response has not been received.
  - Clicks on the button “Show alert” keep showing the alert window during this time.

Why Using Callbacks is Unsatisfactory?

Asynchronous functionality has been available in JavaScript before the language added the feature of promises. An example of this is the function `XMLHttpRequest()` which, as we saw, can run asynchronously and execute callback functions when the HTTP request completes. However, when our program needs to make multiple asynchronous calls that depend on each other, implementing asynchronous functionality using callback functions can lead to hard-to-understand code. Consider the following scenario:

1. A program makes an HTTP request and in the response gets the URL of an image.
2. Then this program makes another HTTP request to retrieve the image from that URL.
3. Finally, the program adds the image into the DOM tree.

Both Steps 1 and 2 should be done asynchronously. If these steps are executed asynchronously, we can provide a callback function for Step 1, and in that callback function we execute Step 2. In the callback function for Step 2, we pass in the callback function for Step 3. This style of nested callbacks leads to such hard-to-understand code that it has been given the name “callback hell”.

Summary

In this exploration we presented an example to motivate the need for asynchronous programming in JavaScript. We will next study how we can write asynchronous code in JavaScript. Understanding how JavaScript engines run asynchronous tasks requires an understanding of JavaScript’s concurrency model and event loop. This knowledge is not required for developing web applications and we do not cover it in this course. However, this knowledge will help you become better computer scientists and we have provided a couple of references. We encourage you to learn about this topic. If you find that you do not have the background needed to understand the topic, shelve it for now and revisit it after you have taken CS 271 “Computer Architecture and Assembly Language” which will certainly provide you with the necessary background.

Additional Resources

- For more information on `XMLHttpRequest` **see MDN (https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest)**. Note that the Fetch API, that we will look at in the next exploration, is now commonly used instead of `XMLHttpRequest`.
- A good explanation of JavaScript’s event loop is provided at **MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop)**.
- For a video explaining JavaScript’s event loop see this **Youtube video (https://www.youtube.com/watch?v=8aGhZQkoFbQ)**.