Asynchronous JavaScript

CS5610: Web Development

Joydeep Mitra

Pre-class Activity

- Fork the repo https://github.com/CSE-316-Software-Development/learn-async-js
- Make a new branch with today's date.
- Push all work to this branch.
- Submit forked URL and branch name to Canvas class activity.

Motivation

- Asynchronous behavior in JavaScript is enabled by callbacks.
  - E.g., pass a callback to readFile(), which gets called after reading is complete.
- However, callbacks make the code complicated, which often leads to bugs!
- Often large JS codebases can get into a *callback hell*.
  - Callbacks can be hard to read and understand especially if the callbacks also call other functions and those functions also have callbacks.

Callback Hell Example

- An example is shown in *src/callback-hell.js*.
  - Note for every callback, we call the function passed as argument.
  - Does it not look unwieldly?
- An easier way to define asynchronous execution in JavaScript is to use a **promise**.

The Promise API

- With a promise-based API we can execute a function asynchronously
  - return a promise object from it.
  - continue without having to wait for the function to complete.
  - on completion, the promise object is triggered to perform further actions.
- We attach handlers to the promise object, which are executed when the promise has succeeded or failed.
- As an example, we will use the *fetch* API in NodeJS to retrieve data from another service.
  - Example in *src/fetch-promise.js*
  - Note what *fetch* returns and how the returned object is handled.
  - Note the order of the messages logged.

Chaining Promises

- Often, we have to call asynchronous functions one after the other.
- For example, in *src/fetch-promise-results.js* we need to get the JSON result from the response received from the service.
  - What does the call to *response.json()* return?
  - Does the code remind you of callback hell?
- The example in *src/fetch-promise-chain* shows how to chain and execute promises one after the other without creating a callback hell.

Promise Errors

- Runtime errors may occur when processing a request.
- Errors are handled by a catch block similar to a try-catch block.
- The catch block runs if an error is encountered in any associated then block.
- As an example, consider the fetch error in *src/fetch-promise-error.js*. The error occurs due to an invalid URL.

Promise States

- A Promise can be in one of three states:
  - **pending**. A promise is pending when the associated async function has not completed.
  - **fulfilled**. A promise is fulfilled when the async function completes and the then *handler* is called.
  - **rejected**. A promise is rejected when the async function has completed with an error and the else handler is *called*.
- Promise and the event loop:
  - Promise callbacks are scheduled on the microtask queue.
  - Callbacks on the microtask queue before every iteration of the event loop but after the main thread completes.

For You to Do

- In *src/quiz/promise-2darray.js* the calls to *sum2DArray()* do not yield a result or an error message as they should. Why? Change the code so they do.
- Also, explain the code execution based on the order of the messages logged. Add your explanation to *src/quiz/promise-2darray.txt.*

Concurrency using Promise

- Sometimes we need to run several promises concurrently and combine their results later.
- The Promise API provides several static methods to enable concurrency.
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#static_methods

Concurrency using Promise

- **Promise.all()** starts several promises at the same time and triggers then *then()* handler only when all of them are resolved.
  - If any one promise is rejected, the whole thing is rejected.
- In *src/fetch-promise-all.js* we fetch json responses from three different URLs and resolve them if all were successfully received. Otherwise, we handle the error in *catch()*.
  - Make one of the URLs badly formed and see what happens?

Concurrency using Promise

- Sometimes we may need any promise to be fulfilled and don’t care which one.
  - Use **Promise.any()** in such cases.
  - *Promise.any()* is rejected if all promises are rejected.
  - If at least one is completed it triggers the *then()* handler.
- Consider the example in *src/fetch-promise-any.js*

For You to Do

- Suppose we want to compute the sum of all integers in a 2D array (in *src/quiz/promise-sum.js)*. This is an expensive operation O(n^2). Use concurrency in promise to make the operation faster.
- From a 2D array of integers in *src/quiz/promise-neg.js*, log any row that has at least one negative number. Use concurrency methods in promise for faster execution.

Async and Await

- Defining asynchronous behavior with promises, especially promise chaining could also look a lot like callback hell!
  - We haven’t solved the problem just given it a different form.
- The async await keywords allow us to write asynchronous code and make it look like synchronous calls.
  - Add **async** before any function definition that will make asynchronous calls.
  - Add **await** before asynchronous function calls.
    - The call will wait until the promise is resolved and returns expected value.
    - Errors due to the function call can be handled using good old **try catch**.
  - A function defined as **async** always returns a promise which must be further processed using **then** blocks.

Async and Await

- In *src/fetch-products-async-await.js* note
  - how the **await** is used before call to fetch.
  - How the functions that use **await** are defined as **async.**
  - The returns values from **async** functions**.**
  - Note the execution order of the **async** functions.

For You to Do

- The Promise.all function in *src/quiz/promise-sum-2d-array.js* looks complicated. Simplify it using **async await**.
- In *src/quiz/animation/main.js*, we are animating three images *alice1, alice2, alice3*. They must be animated one after the other, that is, *alice2* must begin animation after *alice1* has completed and so on.
  - The *animate()* method is a Web Animation API, which returns an *Animation* object that is used by the browser to perform the animation. This method has a property called *finished*, which returns a promise that can be used to indicate that the animation for the object is complete.
  - The code right now looks a lot like callback hell. Simplify it using **aync await**.