COMP6080: Web Front-End Programming

Tutorial 8

Sidney Pham
sidney.pham@unsw.edu.au

UNSW

Week 8, Term 3 2020

Introduction

Firstly, congrats on making it this far in the course! This is the third last week (and I'm starting to get all nostalgic).

Introduction

Firstly, congrats on making it this far in the course! This is the third last week (and I'm starting to get all nostalgic).

Today's tutorial is hopefully a useful one! I'm going to give you some recommendations on the React exercise from Lab07 and then we're going to go over how the event loop works in JavaScript!

Introduction

Firstly, congrats on making it this far in the course! This is the third last week (and I'm starting to get all nostalgic).

Today's tutorial is hopefully a useful one! I'm going to give you some recommendations on the React exercise from Lab07 and then we're going to go over how the event loop works in JavaScript! This is an area that most JavaScript developers neglect to understand (including me, for a very long time); and hopefully this clarifies lots of questions about *why* things work a certain way.

Clickbait

Let's start with something easy: what's the order of the logs here?

```
Promise.resolve().then(() => console.log("promise done"));
console.log("end");
```

Clickbait

Let's start with something easy: what's the order of the logs here?

```
Promise.resolve().then(() => console.log("promise done"));
console.log("end");
```

A: `end`, `promise done`

Clickbait

Let's start with something easy: what's the order of the logs here?

```
Promise.resolve().then(() => console.log("promise done"));
console.log("end");
```

A: `end`, `promise done`

Now, something a little harder — what about this?

```
console.log('before');
setTimeout(() => console.log('setTimeout'));
Promise.resolve().then(() => console.log('promise'));
console.log('after');
```

Clickbait

Let's start with something easy: what's the order of the logs here?

```
Promise.resolve().then(() => console.log("promise done"));
console.log("end");
```

A: `end`, `promise done`

Now, something a little harder — what about this?

```
console.log('before');
setTimeout(() => console.log('setTimeout'));
Promise.resolve().then(() => console.log('promise'));
console.log('after');
```

A: `before`, `after`, `promise`, `setTimeout`

Clickbait

Let's start with something easy: what's the order of the logs here?

```
Promise.resolve().then(() => console.log("promise done"));
console.log("end");
```

A: `end`, `promise done`

Now, something a little harder — what about this?

```
console.log('before');
setTimeout(() => console.log('setTimeout'));
Promise.resolve().then(() => console.log('promise'));
console.log('after');
```

A: `before`, `after`, `promise`, `setTimeout`

We'll see why this works after this code review!

The Lab07 React App

Let's review this, uh, random lab (let's look at the GitHub profile exercise) and discuss as a class what we think can be improved.

The Tutorial's Solutions

1. *Overall pretty good (I agree)*
2. *Inline CSS in Card.jsx*
3. *Incosistent use of JSX and JS file extensions*
4. *Inconsistent use of semicolons and non semicolons to terminate statements*
5. *App.js has a redundant wrapper element around the input*
6. *Many lines are very long and should be moved onto next line*

Clearing the Interval Nicely

Last week I recommended (for this task) the pattern of storing the `timeoutID` in a global variable, but there's a nicer way of doing it (that many smart students spotted)!

Clearing the Interval Nicely

Last week I recommended (for this task) the pattern of storing the `timeoutID` in a global variable, but there's a nicer way of doing it (that many smart students spotted)!

What we're trying to do is run a side effect if no key is pressed in 500ms.

Clearing the Interval Nicely

Last week I recommended (for this task) the pattern of storing the `timeoutID` in a global variable, but there's a nicer way of doing it (that many smart students spotted)!

What we're trying to do is run a side effect if no key is pressed in 500ms. Recall that `useEffect` lets us specify a cleanup function?

Clearing the Interval Nicely

Last week I recommended (for this task) the pattern of storing the `timeoutID` in a global variable, but there's a nicer way of doing it (that many smart students spotted)!

What we're trying to do is run a side effect if no key is pressed in 500ms. Recall that `useEffect` lets us specify a cleanup function? It runs right before every re-render

Clearing the Interval Nicely

Last week I recommended (for this task) the pattern of storing the `timeoutID` in a global variable, but there's a nicer way of doing it (that many smart students spotted)!

What we're trying to do is run a side effect if no key is pressed in 500ms. Recall that `useEffect` lets us specify a cleanup function? It runs right before every re-render so we can use this to cancel the timeout (i.e. `clearTimeout`) that we schedule in the effect.

My Solution

Here's my solution (which isn't perfect). I think a few things to point out are...

- The discrinated union trick from my Tut05 `timer` code.
- The `Promise.all`.
- The error handling.
- That it hopefully isn't too complicated.

My Solution

Here's my solution (which isn't perfect). I think a few things to point out are...

- The discrinated union trick from my Tut05 `timer` code.
- The `Promise.all`.
- The error handling.
- That it hopefully isn't too complicated.

...and a few more things.

Doing `Promise.all` Twice

Plenty of people ran `Promise.all([fetch(...), fetch(...), ...])` and then with the `results` (an array of the results of the input promises) they did `Promise.all(results => results.map(res => res.json()))`.

Doing `Promise.all` Twice

Plenty of people ran `Promise.all([fetch(...), fetch(...), ...])` and then with the `results` (an array of the results of the input promises) they did `Promise.all(results => results.map(res => res.json()))`.

But the time this takes is the sum of the longest time for the fetch step *plus* the longest time to run `.json()`.

Mixing `async`/`await` with Promise Chaining

You don't have to exclusively use one or the other.

Mixing `async`/`await` with Promise Chaining

You don't have to exclusively use one or the other. One example of a useful line is:

```
const data = await fetch(url).then(res => res.json());
```

Mixing `async`/`await` with Promise Chaining

You don't have to exclusively use one or the other. One example of a useful line is:

```
const data = await fetch(url).then(res => res.json());
```

I like doing this when I want to hide boring details (like the `.json()` step) in `.then`, letting our code still read nicely synchronously.

Fetch's 404 Behaviour

A *lot* of people were very sloppy with not checking for errors in the fetch step. (I get it.)

Fetch's 404 Behaviour

A *lot* of people were very sloppy with not checking for errors in the fetch step. (I get it.)

But be careful:

> *the Promise returned from* `fetch()` *won't reject on HTTP error status even if the response is an HTTP 404 or 500. Instead, it will resolve normally (with* `ok` *status set to false), and it will only reject on network failure or if anything prevented the request from completing.*

Fetch's 404 Behaviour

A *lot* of people were very sloppy with not checking for errors in the fetch step. (I get it.)

But be careful:

> *the Promise returned from* `fetch()` *won't reject on HTTP error status even if the response is an HTTP 404 or 500. Instead, it will resolve normally (with* `ok` *status set to false), and it will only reject on network failure or if anything prevented the request from completing.*

From MDN, an accurate check for a successful `fetch()` would include checking that the promise resolved, then checking that the `Response.ok` property has a value of true.

Fetch's 404 Behaviour (cont.)

It recommends that you do something like:

```
fetch('flowers.jpg')
    .then(response => {
        if (!response.ok) {
            throw new Error('Network response was not ok');
        }
        return response.blob();
    })
    .then(myBlob => {
        myImage.src = URL.createObjectURL(myBlob);
    })
    .catch(error => {
        console.error('Fetch problem!');
    });
```

Isn't JavaScript Single Threaded?

We've heard many times that JavaScript is single threaded, but then how do things like `setTimeout` or `fetch` not block (like if we had a long loop).

Isn't JavaScript Single Threaded?

We've heard many times that JavaScript is single threaded, but then how do things like `setTimeout` or `fetch` not block (like if we had a long loop).

Well, it *is*, but our browser (or other environment like Node) provides us with many Web APIs (like `XMLHttpRequest` and the DOM).

Isn't JavaScript Single Threaded?

We've heard many times that JavaScript is single threaded, but then how do things like `setTimeout` or `fetch` not block (like if we had a long loop).

Well, it *is*, but our browser (or other environment like Node) provides us with many Web APIs (like `XMLHttpRequest` and the DOM). Your *browser* can spin up another thread for these API calls!

The Task Queue

When you make a browser API call like `setTimeout`, the browser goes off and handles it asynchronously for you so your code can continue to run.

The Task Queue

When you make a browser API call like `setTimeout`, the browser goes off and handles it asynchronously for you so your code can continue to run.

When it's done, if there's something that needs to happen (e.g. if you've attached a listener to an event), a task will get added to the task queue!

The Task Queue

When you make a browser API call like `setTimeout`, the browser goes off and handles it asynchronously for you so your code can continue to run.

When it's done, if there's something that needs to happen (e.g. if you've attached a listener to an event), a task will get added to the task queue!

This is where the event loop comes in!

The Event Loop

The event loop is a part of the JavaScript runtime that continuously polls the task queue for work to do.

The Event Loop

The event loop is a part of the JavaScript runtime that continuously polls the task queue for work to do. But it *only* does this if the call stack is clear!

The Event Loop

The event loop is a part of the JavaScript runtime that continuously polls the task queue for work to do. But it *only* does this if the call stack is clear!

Let's have a look at what the call stack is, and how this works. This is a cool visualisation.

Example 1

Demo `foo-bar.js`.

Example 1

Demo `foo-bar.js`.

Hopefully how simple asynchronous code with `setTimeout` is clearer now.

But Promises don't work exactly the same way...

Microtasks

Promises are an example of a microtask, whereas `setTimeout` callbacks are an example of a task, and so your browser actually has *multiple* types of queues! (More info here.)

Our Examples

To understand our Promise examples we need to understand the way the event loop works.

Our Examples

To understand our Promise examples we need to understand the way the event loop works. Whenever the browser finishes executing a task (such as executing the code!), it empties the microtask queue and executes all tasks in it, before continuing with a task from another task queue.

The Event Loop

(Note: I honestly recommend ignoring this slide and referring to the resources on the next slide.)

The Event Loop

(Note: I honestly recommend ignoring this slide and referring to the resources on the next slide.)

A simplified understanding of the event loop algorithm (though much simpler than the specification):

1. Dequeue and run the oldest task from the task queue (e.g. "script").
2. Execute all microtasks:
    a. While the microtask queue is not empty, dequeue and run the oldest microtask.
3. Render any changes.
4. If the task queue is empty, wait till a task appears.
5. Go to step 1.

Resources

I found a lot of really good resources on the Internet. The `latentflip.com/loupe` tool is made by Philip Roberts, whose explanation of the event loop you have to watch!

Resources

I found a lot of really good resources on the Internet. The `latentflip.com/loupe` tool is made by Philip Roberts, whose explanation of the event loop you have to watch!

After watching that, some more great talks are this one on microtasks by Jake Archibald (who you'll see all over the web) and this one that also talks about microtasks.

Resources

I found a lot of really good resources on the Internet. The `latentflip.com/loupe` tool is made by Philip Roberts, whose explanation of the event loop you have to watch!

After watching that, some more great talks are this one on microtasks by Jake Archibald (who you'll see all over the web) and this one that also talks about microtasks.

MDN has a pretty comprehensive explanation here that I think you should read, and it explains really well what tasks and microtasks are exactly. I also found this StackOverflow answer helpful.