Async Programming And You

Part 2: Return of the callback

By Zain Afzal

@zainafzal08

@blockzain

Chaining With Promises

.then can return a promise in itself allowing you to do multiple async actions in sequence

```
doSomething(result => {
   doSomethingElse(result, newResult => {
     doThirdThing(newResult, finalResult => {
       console.log(finalResult);
     }, handleFailure);
   }, handleFailure);
}, handleFailure);
```

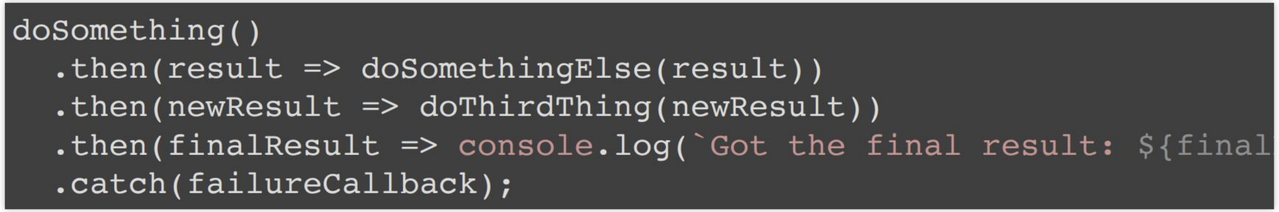
```
doSomething()
 .then(result => doSomethingElse(result))
 .then(newResult => doThirdThing(newResult))
 .then(finalResult => {
       console.log(finalResult);
   })
 .catch(handleFailure);
```

DEMO

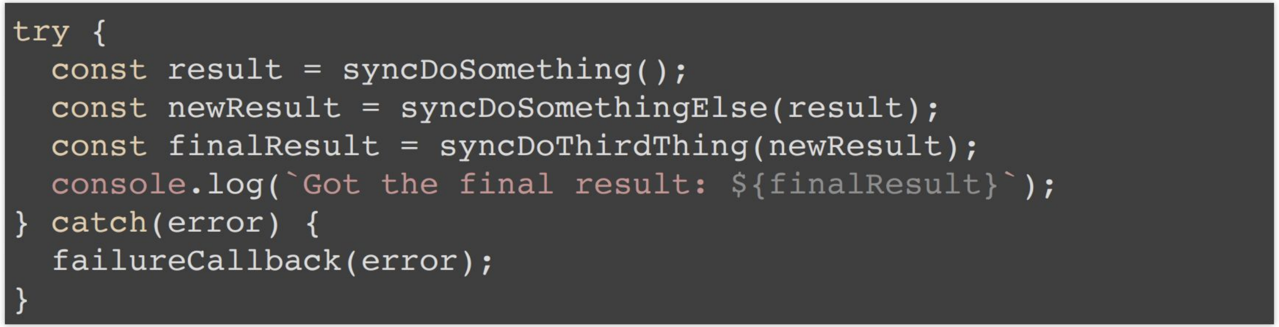
Redo text animation with a wrap

Catching Promises

A promise chain stops if there's an exception and looks down the chain for a catch handler instead



```
doSomething()
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(finalResult => console.log(`Got the final result: ${final
  .catch(failureCallback);
```



```
try {
  const result = syncDoSomething();
  const newResult = syncDoSomethingElse(result);
  const finalResult = syncDoThirdThing(newResult);
  console.log(`Got the final result: ${finalResult}`);
} catch(error) {
  failureCallback(error);
}
```

Catching Promises

A catch statement can be used to continue the chain after a failure as it also returns a promise that resolves immediately

```
fetchUsers()
  .then(() => {
      throw new Error('Something failed');
      console.log('Do this');
  })
  .catch(() => {
      console.log('Do that');
  })
  .then(() => {
      console.log('Do this, no matter what happened before');
  });
```

```
Do that
Do this, no matter what happened before
```

Fetch

You could wrap XMLHTTP to give you all the nice features of promises or you can just use fetch, it brings the magic of promises by default as well as being a bit less confusing to read/use

```
fetch('http://example.com/movies.json')
 .then(response => response.json())
 .then(myJson => {
     console.log(myJson);
 });
```

Fetch/Promise Notes

- Will not throw an error if a HTTP request fails
  - if a request returns 400 or 500 etc. it assumes this may be expected
  - You will have to detect this downstream and raise a result if it is **not** expected
- Request.JSON() returns a promise which returns a json object, not just a json object
- All requests are assumed to be GET unless you specify otherwise
- You can’t cancel a Promise
  - This poses potential issues when using promises for long running tasks like heavy downloads
- You shouldn't use promises
  - When you have a callback situation where the callback is designed to be called multiple times
  - For situations where the action often does not often finish or occur
- You can still get into promise hell if you don’t use them correctly, make sure if you ever have a further async action you return a promise and keep your logic flat

Avoiding Promise Hell

```
fetchBook()
 .then(book => {
   return formatBook(book)
     .then(book => {
       return sendBookToPrinter(book);
     });
 });
```

```
fetchBook()
 .then(book => formatBook(book))
 .then(book => sendBookToPrinter(book));
```

More Advanced Fetch Use Cases

```
const url = “api.com/posts";

const payload = {
    title: 'My fun day!',
    content: 'Had a fun day it\'s in the title'
}

const options = {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(payload)
}


fetch(url, options)
    .then(r => console.log('WOW!');
```

```
const options = {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
        // get this token after you login
        'Authorization': 'Token 2dcc8215'
    },
    body: JSON.stringify({governmentSecrets: 'yes'})
}


fetch('nsa.secret.com', options)
    .then(r => console.log('WOW!');
```

DEMO

Track the ISS

Roadmap

Promise++ And Async Await

But wait! There's more

Promise Helpers

What if i want to fetch 4 things and need all 4 before processing them? Use **promise.all()**!

```
function handle(f) {
    fruits.push(r)
    if (fruits.length === 4)
        make_fruit_salad(fruits)
}
fruits = []
fetch('myapi/apple').then(r => handle(r))
fetch('myapi/orange').then(r => handle(r))
fetch('myapi/banana').then(r => handle(r))
fetch('myapi/pear').then(r => handle(r))
```

```
fruits = [
    fetch('myapi/apple'),
    fetch('myapi/orange'),
    fetch('myapi/banana'),
    fetch('myapi/pear')
]

Promise.all(fruits)
    .then(fruits => make_fruit_salad(fruits))
```

Promise Helpers

What if i want to fetch 4 things and just what whatever i faster? Use **promise.race()**!

```
function handle(f) {
    if (fruit === null) {
      fruit = f
      eat(f)
    }
}
fruit = null;
fetch('myapi/apple').then(r => handle(r))
fetch('myapi/orange').then(r => handle(r))
fetch('myapi/banana').then(r => handle(r))
fetch('myapi/pear').then(r => handle(r))
```

```
fruits = [
    fetch('myapi/apple'),
    fetch('myapi/orange'),
    fetch('myapi/banana'),
    fetch('myapi/pear')
]

Promise.race(fruits)
    .then(fruit => eat(fruits))
```

Async Await

This is just Syntactic sugar around Promises which helps you write code that feels more synchronous.

Rather than registering some handler for a event we can **await** it. This basically returns from the function and resumes it when the response is ready.

Our code becomes so much easier to reason about and additionally takes away the additional energy it takes to write things correctly.

You’ll find similar paradigms in other languages like python and you’ll find this concept invaluable when working with websockets

```javascript
function sleep(time) {
  return new Promise((resolve, reject) => {
    setTimeout(function(){resolve(time);}, time);
  });
}

// you can only await in a async function
async function animate() {
    document.getElementById('output').innerText += 'the...'
    // wait until the promise
    // resolves before continuing
    await sleep(1000)
    document.getElementById('output').innerText += 'suspense'
}
animate()
```

```
function sleep(time) {
  return new Promise((resolve, reject) => {
    setTimeout(function(){resolve(time);}, time);
  });
}

// you can only await in a async function
function animate() {
    document.getElementById('output').innerText += 'the...'
    // wait until the promise
    // resolves before continuing
    return sleep(1000).then(_ => {
        document.getElementById('output').innerText += 'suspense'
        return new Promise.resolve()
    })
}
animate()
```

Demo
Text animation
(again)

Roadmap

1 The Event Loop

2 XMLHTTP And Callback Hell

3 Fetch and Promise Heaven

4 Promise++ And Async Await

5 Assignment

6 Beyond 2041

The Assignment

Seddit (geddit?)

Data Attributes

data-* attributes allow us to store extra information on standard, semantic HTML elements without other hacks

```html
<article

  id="electric-cars"

  data-columns="3"

  data-index-number="12314"

  data-parent="cars">

...

</article>
```

```javascript
const article =
document.querySelector('#electric-cars');

article.dataset.columns // "3"
article.dataset.indexNumber // "12314"
article.dataset.parent // "cars"
```

Pointers

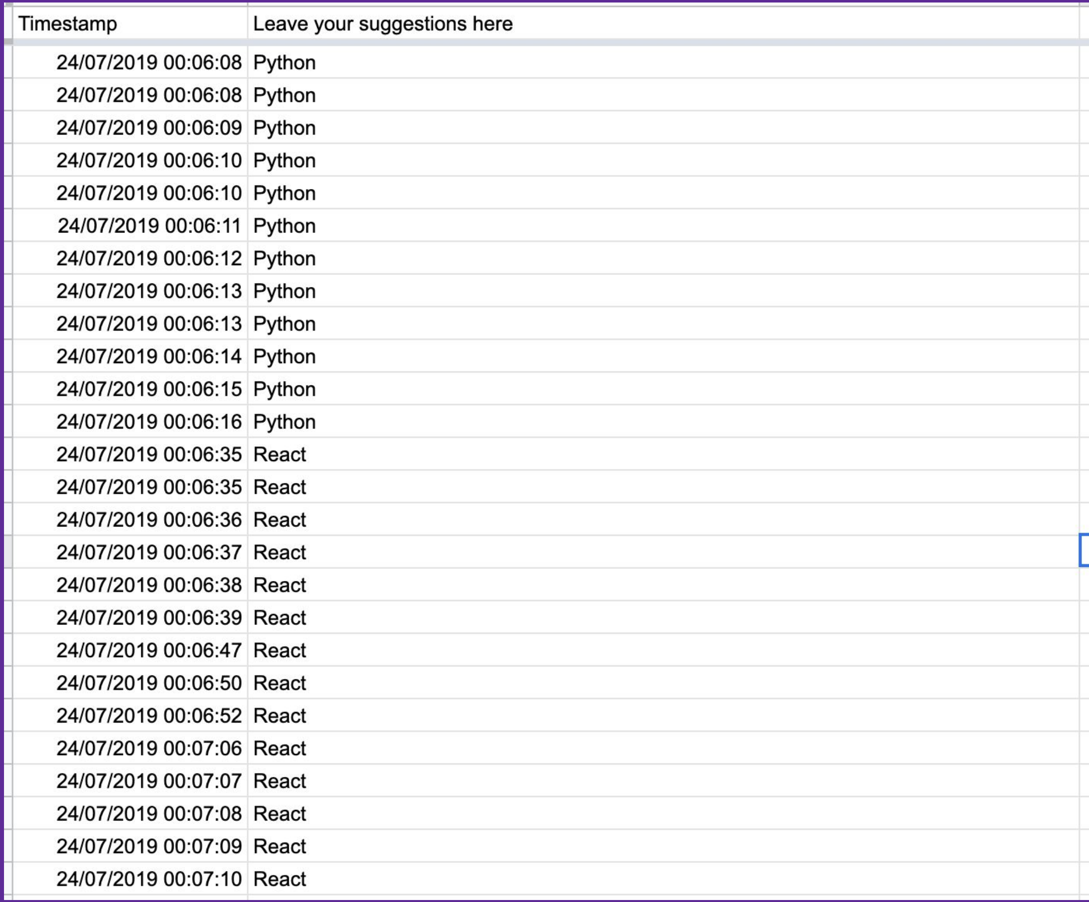
- You must not use .innerHTML
  - slow
  - prone to security risks
  - defeats the purpose of the assignment
  - at scale makes it harder
- Be careful of caching
  - In the network tab you can turn off caching while the tools are open
- Some of the later milestones make previous milestones redundant (as is life with web). Think ahead a bit if your planning on hitting the later milestones
- The images are stored in base64
  - This is just a way to represent binary data, you can display them with a image tag
  - see base64 example
- You can request static json files in the same way you'd make a api request
  - fetch('/file.json').then(r=>r.json())

Running the assignment

Roadmap

Beyond 2041

No.



Timestamp	Leave your suggestions here
24/07/2019 00:06:08	Python
24/07/2019 00:06:08	Python
24/07/2019 00:06:09	Python
24/07/2019 00:06:10	Python
24/07/2019 00:06:10	Python
24/07/2019 00:06:11	Python
24/07/2019 00:06:12	Python
24/07/2019 00:06:13	Python
24/07/2019 00:06:13	Python
24/07/2019 00:06:14	Python
24/07/2019 00:06:15	Python
24/07/2019 00:06:16	Python
24/07/2019 00:06:35	React
24/07/2019 00:06:35	React
24/07/2019 00:06:36	React
24/07/2019 00:06:37	React
24/07/2019 00:06:38	React
24/07/2019 00:06:39	React
24/07/2019 00:06:47	React
24/07/2019 00:06:50	React
24/07/2019 00:06:52	React
24/07/2019 00:07:06	React
24/07/2019 00:07:07	React
24/07/2019 00:07:08	React
24/07/2019 00:07:09	React
24/07/2019 00:07:10	React

Modern Web Design

This is a overview as the field is both very complex and fastly evolving, many JS libraries have been made over the years to do a couple of things:

- Address Javascripts shortcomings
  - better api (jquery for example used $(‘’) rather then ‘document.getElementById etc.’)
  - modules (before they were implemented)
  - Adding more type safety
- Modularise the process of building UI
  - Makes things easier to manage and reuse
  - Helps decrease coupling
- Speed up DOM interactions
  - use smart code to minimise the number of times you touch the DOM to a bare minimum to further increase speed
- Make JS dev faster and less verbose

Frameworks

Some Libraries you'll hear about

A library from facebook where all your UI is contained in a virtual DOM in neat components that then, on changing, trigger a small number of DOM operations to make the REAL dom match. Also can be ported to a native app.

Similar to react with the virtual DOM but more of a focus on HTML, CSS and JS where React focuses more heavily on doing all these things within their higher level language (called JSX). Vue has HTML templates and the ability to be integrated into a project gradually without needing a full rewrite.

In addition Vue has more built in features that React leans on third party apps for. Can not be native just yet.

From google, Angular solves similar issues to the above too, allowing you to break down code into components and use templating to generate HTML. It however does not have a virtual DOM and has strong opinions on how you need to structure your code, where react/vue give you some freedom with how to model data and connect it to your view, angular forces you to do it through it's systems.

What framework do i use?

You Retweeted
Mark Dalgleish
@markdalgleish

When people ask what the best JavaScript framework is.

- Choose what you like, the important thing is you understand the issues the framework is trying to solve
  - for this reason learning vanilla JS to get context on why these libraries exist is A+
- In general however
  - React is gaining popularity very fast in industry
  - Vue has very good docs and a better learning curve then react
  - typescript is taking over everywhere quite quickly

Transpilers

Sinces browsers for the time being mostly only run JS many projects have a pipeline where higher level (and often better) languages are transpiled down into JS

BABEL	Transpiles newer JS into backwards compatible JS
TS	A superset of JS with types that transpiles down to normal JS

SPA / PWA

SPA stands for **single page app** and it’s where a website is a single HTML page that uses JS to swap out the entire view rather then doing a fetch for another HTML file. Cut out slow full page requests entirely

This is how we want you to build your assignment.

Progressive Web Apps are usually SPA’s which are designed to replace standard apps, that is to say a app that works in a way that removes the need for a desktop app or mobile app build in a lower level language.

They can do things like push notifications, work offline etc. But there isn’t a strong definition. Regardless they are popular because you have 1 code base for all devices

Web workers

Web Workers are a simple means for web content to run scripts in background threads. The worker thread can perform tasks without interfering with the user interface.

```
const worker = new Worker('incremento.js');
worker.addEventListener('message', (r) => {
      console.log(r.data)
});
worker.postMessage(1)


// we get 2 printed to the console
```

```
onmessage = function(e) {
    const x = e.data + 1;
    postMessage(x);
}
```