

JavaScript Asynchronous Patterns

Frank Walsh

then

Agenda

- Callbacks
- Promises
- Async Await
- Further/advanced async behaviour



Recap -Javascript Characteristics

- JavaScript is single threaded
- JavaScript is event driven
 - Events happen – we write code to deal with them
 - Can use callbacks to do this
- JavaScript can be Asynchronous
 - Order of operation results may differ from order they were called...

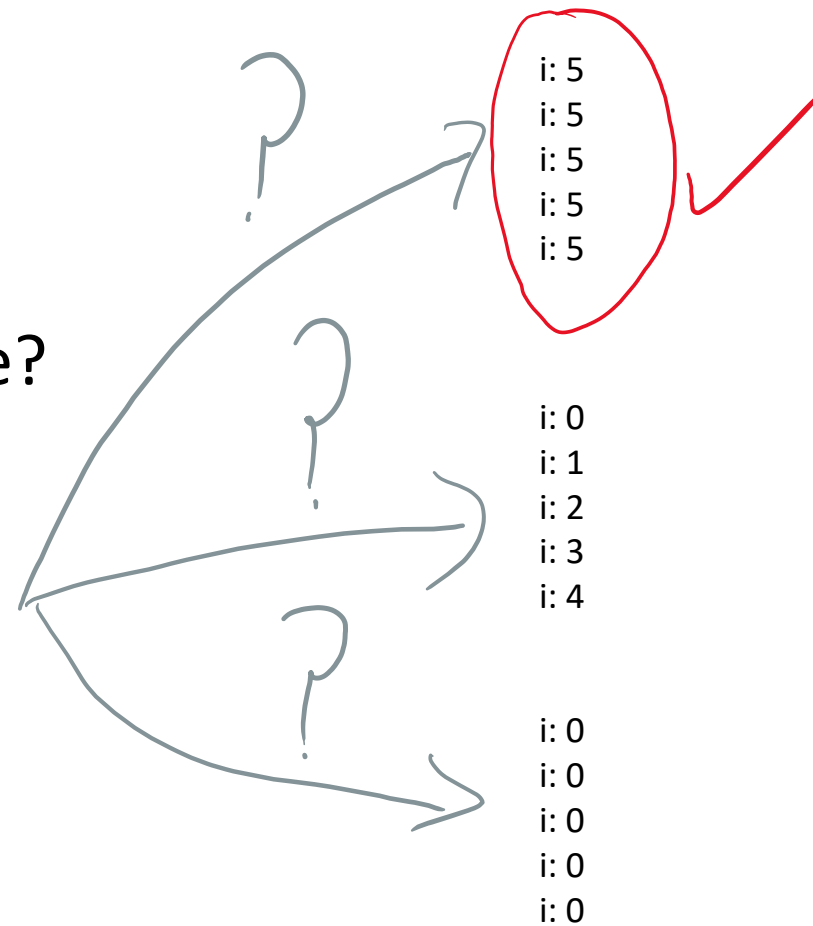
A large, dark gray 'JS' logo is centered on a solid yellow background. The letters are bold and sans-serif, with the 'J' and 'S' being slightly larger than the 'J' alone.

Callbacks (again)

Recap - The Callback

What will be logged to the console?

```
console.log('setTimeout')
let i=0;
while (i <5) {
  setTimeout(() => {
    console.log('i: ' + i)
  }, 0);
  i++;
}
```



Hint: the while loop will finish before the 1st callback executes.

The Callback

- The traditional way of handling asynchronous events
 - Function that is registered as the event handler for something we're fairly sure will happen in the future

```
componentDidMount : ()=> {  
  request.get('http://0.0.0.0:3000/friends')  
    .end( (error, response) => { . . . Callback code ... }  
  }  
  .....  
  filterFriends : (event) => { . . . Callback . . . }  
  render: ()=>{  
    .....  
    <input type="text" .... onChange={this.filterFriends} />  
  }  
}
```

What have Callbacks ever done for us...

- Great for things that can happen multiple times
- Great if you don't really care about what happened before you attached the listener
- Great if it's a straight-forward, stand-alone event with a quick resolution time
- Great if callback is not part of sequential process.

E.g. key press event on control.

```
document.getElementById("demo").addEventListener("keypress", myFunction);
```

- **But...**



Multiple Callbacks

I want this to happen sequentially... but order of callback events is indeterminate.

```
console.log("about to read...");
fs.readFile("test-1.txt", "utf8", (err, contents) => {
  console.log(contents);
});
fs.readFile("test-2.txt", "utf8", (err, contents) => {
  console.log(contents);
});
fs.readFile("test-3.txt", "utf8", (err, contents) => {
  console.log(contents);
});
console.log("...done");
```

Possible Result

```
about to read...
...done
- 4
- 5
- 6
- 7
- 8
- 9
- 1
- 2
- 3
```

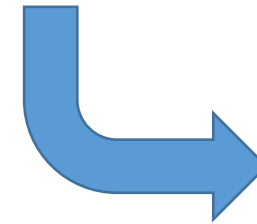
Possible Result

```
about to read...
...done
- 1
- 2
- 3
- 7
- 8
- 9
- 4
- 5
- 6
```


Sequential callbacks.

- A callback in a callback in a callback!
- Sequential callbacks are difficult to read / understand
- Sequential callbacks cause many levels of indentation in source code
- And this is typically just the “success path”
 - It gets more complicated if you want to handle errors

```
console.log("about to read...");
fs.readFile("test-1.txt", "utf8", (err, contents) => {
  if (err) throw err;
  console.log(contents);
  fs.readFile("test-2.txt", "utf8", (err, contents) => {
    if (err) throw err;
    console.log(contents);
    fs.readFile("test-3.txt", "utf8", (err, contents) => {
      if (err) throw err;
      console.log(contents);
    });
  });
});
```



```
about to read...
...done
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
```

Callback Hell – Multiple sequential requests

```
1  var amount=req.param("amount");
2  db.select("* from sessions where session_id=?", req.param("session_id"), function(err, sessions) {
3    if (err) throw err;
4    db.select("* from accounts where user_id=?", sessions[0].user_ID, function(err, accounts) {
5      if (err) throw err;
6      if (accounts[0].balance < amount) throw new Error('insufficient balance');
7      db.execute("withdrawal(?, ?)", accounts[0].ID, req.param("amount"));
8      if (err) throw err;
9      res.write("withdrawal OK, amount: " + req.param("amount"));
10     db.select("balance from accounts where account_id=?", accounts[0].ID, function(err, balance) {
11       if (err) throw err;
12       res.end("your current balance is " + balance[0].amount);
13     });
14   });
15 });
```

```
1
2
3  function loadUpThatApplication() {
4    request("/api/getCustomer", function(response){
5      var customerId = response.customer.id;
6      request("/api/customer/accounts/"+customerId, function (response2) {
7        request("http://facebook/pics/"+response2.faceBookUserName, function (response3) {
8          showTheUserThatBeautifulUI(response3, function () {
9            byeByeSpinner();
10          });
11        });
12      });
13    });
14  }
15
16
```

JS Promise

#Javascript

mduyemvun.vn

This Photo by Unknown Author is licensed under [CC BY-NC-ND](#)

Promises |

Promises

- A promise is the eventual result of an asynchronous operation or computation.
- Promises are:
 - an abstraction useful in async programming
 - an associated API that allows us to use this abstraction in our programs.
- A promise can be:
 - **fulfilled** - The action relating to the promise succeeded
 - **rejected** - The action relating to the promise failed
 - **pending** - Hasn't fulfilled or rejected yet
 - **settled** - Has fulfilled or rejected

fs using promises (experimental)

```
import {promises as fs} from 'fs';

console.log("about to read...");
fs.readFile("test-1.txt", "utf8").then((contents) => {
  console.log(contents);
});

console.log("...done");
```



output

```
about to read...
...done
(node:10072) ExperimentalWarning: The fs.promises API is experimental
- one
- two
- three
```

- If you return a value to a ***then()***, the next ***then()*** is called with that value.
- If you return a promise, the next ***then()*** waits on it, and is only called when that promise settles (i.e. either succeeds/rejects).



fs sequential using promises (chaining)

```
import {promises as fs} from 'fs';

console.log("about to read...");
fs.readFile("test-1.txt", "utf8").then((contents) => {
  console.log(contents);
  return fs.readFile("test-2.txt", "utf8")
}).then((contents) => {
  console.log(contents);
  return fs.readFile("test-3.txt", "utf8")
}).then((contents) => {
  console.log(contents)
}), (error) => {
  console.error('File read failed', error)
};

console.log("...done");
```

output

```
about to read...
...done
(node:8404) ExperimentalWarning: The fs.promises API is experimental
- one
- two
- three
- four
- five
- six
- seven
- eight
- nine
```

Async Functions

using `async/await`

Async/Await !

- **async/await** and **promises** are essentially the same under the hood
- **async** is a keyword
 - Used in function declaration
- **await** is used during the promise handling
 - must be used within an **async function**
- **async** functions return a promise, regardless of what the return value is within the function.
- **Available now!** in most good browsers as well as Node.js

```
import {promises as fs} from 'fs';

console.log("about to read...");

readFiles()

console.log("...doing other stuff")

|

async function readFiles() {
  console.log("starting sequential read...");
  const contents1 = await fs.readFile("test-1.txt", "utf8");
  console.log(contents1);
}
```

Async/Await Sequential read

```
console.log("about to read...");

readFiles()

console.log("...doing other stuff")

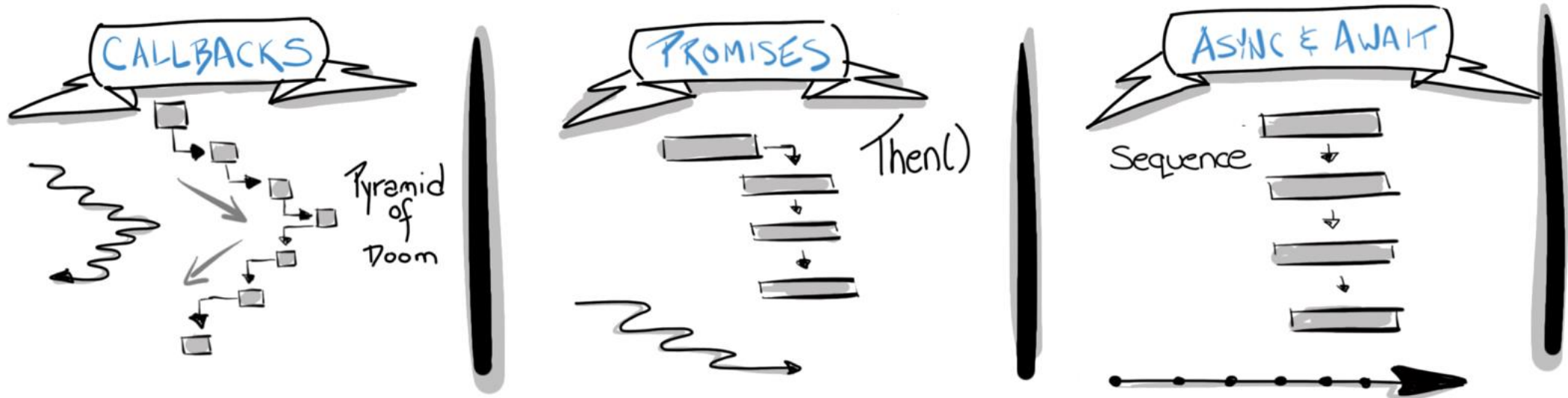
async function readFiles() {
  console.log("starting sequential read...");
  const contents1 = await fs.readFile("test-1.txt", "utf8");
  console.log(contents1);
  const contents2 = await fs.readFile("test-2.txt", "utf8");
  console.log(contents2);
  const contents3 = await fs.readFile("test-3.txt", "utf8");
  console.log(contents3);
  console.log("...done sequential read");
}
```

output

```
about to read...
starting sequential read...
...doing other stuff
(node:1988) ExperimentalWarning:
- one
- two
- three
- four
- five
- six
- seven
- eight
- nine
...done sequential read
```

Callbacks vs Promises vs Async-Await

Asynchronous Javascript



Error Handling

Error Handling - Promises

- **then()** function can take **2 arguments**, one for fulfillment(success), one for rejection(failure)

```
fs.readFile("test-12.txt", "utf8").then((contents) => {  
    console.log(contents);  
}, (error)=>{console.error("Failed to read file!",error)});
```

Error Handling - catch(...)

- You can also use catch() to handle promise rejects:
- Reacts slightly different to previous.
 - Useful in sequential async processes

```
fs.readFile("test-1.txt", "utf8").then((contents) => {  
  console.log(contents);  
  return fs.readFile("test-2.txt", "utf8")  
}).then((contents) => {  
  console.log(contents);  
  return fs.readFile("test-3.txt", "utf8")  
}).then((contents) => {  
  console.log(contents)  
}).catch ((error) => {  
  console.error('File read failed', error)  
});
```

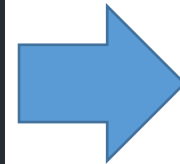
Error Handling – async await

Use try -catch

```
async function readFiles() {  
  try{  
    console.log("starting sequential read...");  
    const contents1 = await fs.readFile("test-1.txt", "utf8");  
    console.log(contents1);  
    const contents2 = await fs.readFile("test-22.txt", "utf8");  
    console.log(contents2);  
    const contents3 = await fs.readFile("test-3.txt", "utf8");  
    console.log(contents3);  
  }catch (error){  
    console.error("failed to read a file!", error)  
  }  
  console.log("...done sequential read");  
}
```

JavaScript promise dummy implementation

```
1  const promise = new Promise((resolve, reject)=> {
2    // do a thing, possibly async, then...
3    console.log('setTimeout');
4    setTimeout(()=> {
5      if (doSomethingThatMightFail()) {
6        resolve('Stuff worked!');
7      } else {
8        reject(Error('It broke'));
9      }
10   }, 1000);
11   }
12 );
13
```



```
14  promise.then((result) => {
15    console.log(result); // "Stuff worked!"
16  }, (err)=>{
17    console.log(err); // Error: It broke
18  });
19
```

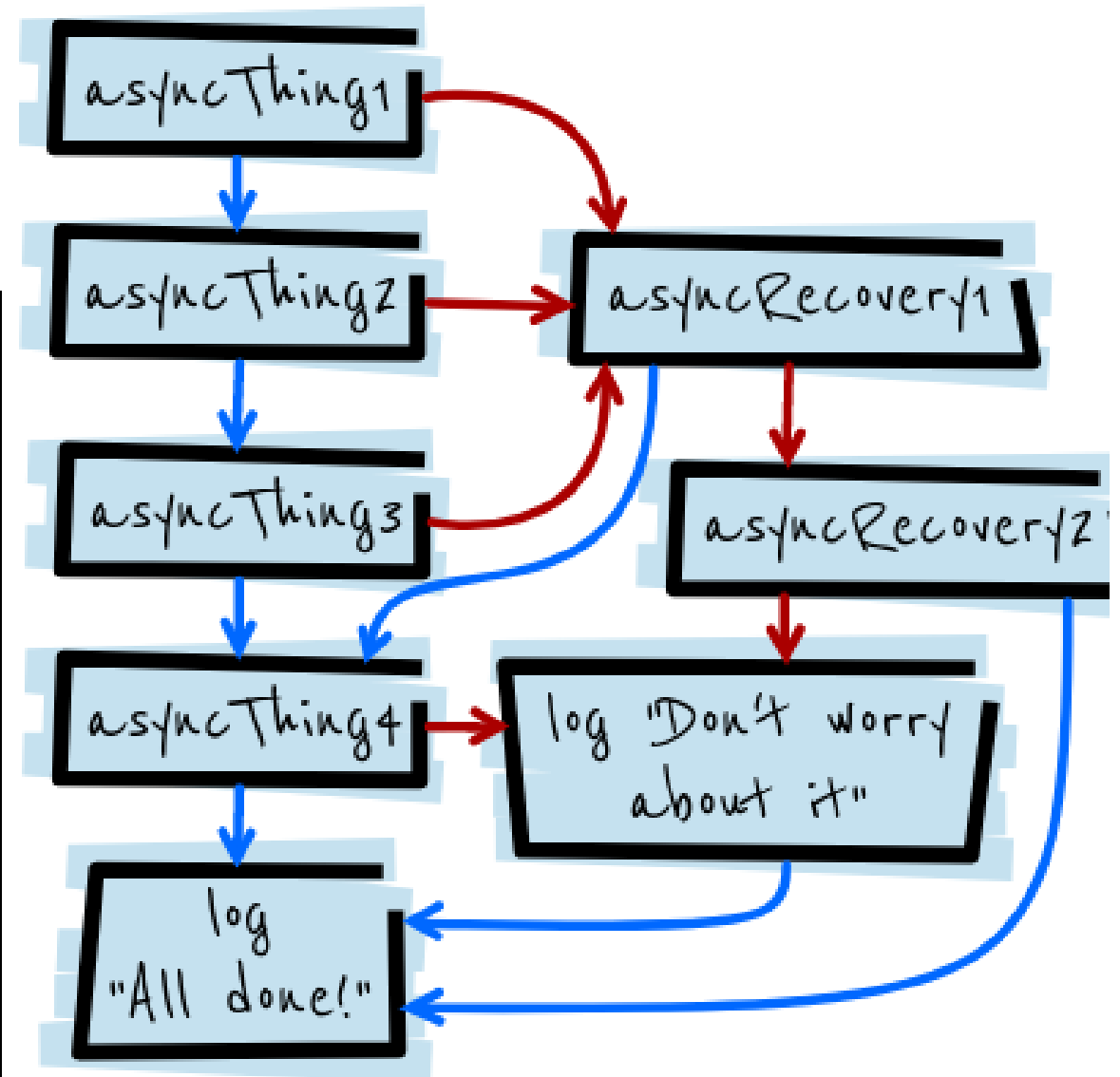
```
20  const doSomethingThatMightFail = ()=>{
21    return result = (Math.random()>.5)? true:false;
22  };
23
```


Further Asynchronous features...

Rejection forwarding

- A Promise rejection will skip forward to the next then() with a rejection callback (or catch()):

```
asyncThing1().then(function() {  
  return asyncThing2();  
}).then(function() {  
  return asyncThing3();  
}).catch(function(err) {  
  return asyncRecovery1();  
}).then(function() {  
  return asyncThing4();  
}, function(err) {  
  return asyncRecovery2();  
}).catch(function(err) {  
  console.log("Don't worry about it");  
}).then(function() {  
  console.log("All done!");  
})
```



Wrapper Function

- As an Async function always returns a Promise.
 - can *wrap* the async function to catch errors...
 - Can drop try/catch.
- Makes code more readable.

```
const asyncWrapper = fn => {  
  return Promise.resolve(fn)  
    .catch(err => {return err.message});  
};
```

```
async function doSomethingAsync() {  
  const result = await asyncWrapper(promise());  
  console.log(result);  
}
```

Chaining

- you can chain then's together to transform values or run additional async actions one after another.

```
1  const promise = new Promise((resolve, reject) => {
2    resolve(1);
3  });
4
5  promise.then((val) => {
6    console.log(val); // 1
7    return val + 2;
8  }).then((val) => {
9    console.log(val); // 3
10 });
```

Parallelism //

- Some processes need to be sequential
 - Eg. Had to get data back from API **BEFORE** getting link URL
- REMEMBER: Should only be sequential if you need to be...

Takes 1000ms

```
async function series() {  
  await wait(500); // Wait 500ms...  
  await wait(500); // ...then wait another 500ms.  
  return "done!";  
}
```

Takes ~500ms

```
async function parallel() {  
  const wait1 = wait(500); // Start a 500ms timer asynchronously...  
  const wait2 = wait(500); // this timer happens in parallel.  
  await wait1; // Wait 500ms for the first timer...  
  await wait2; // ...by which time this timer has already finished.  
  return "done!";  
}
```

Sources

- <https://developers.google.com/web/fundamentals/primers/promises>
- <https://stackoverflow.com/questions/2069763/difference-between-event-handlers-and-callbacks>
- <https://medium.com/@Abazhenov/using-async-await-in-express-with-node-8-b8af872c0016>