Service Calls are Asynchronous (async)

Async is (IMO) the #2 hardest things in JS

- We mostly skipped #1 (`this` keyword)
- We mostly skipped #3 (prototypes)

So async may stand out as difficult

- But you are here to learn
- Once you learn
    - async and promises are awesome

Async in a nutshell

Async examples:

- A request to an express server
- A `click` event listener

You say:

- When (something) happens
    - Call this callback

The callback doesn't happen immediately

- it happens "asynchronously"
    - "not in order"

Key lesson: single-threaded

Remember that your JS runs single-threaded

This means your code will never interrupt itself

- Any running synchronous code...
    - Such as the callback
- ...will finish before any other code runs

This is great for you the developer

All synchronous code will finish without interruption

Key lesson: results are async

If you want a result that comes from async code

- Such as from a service call
- You can't use that result...
    - ...in the code that triggered async

```
let name = "";
setTimeout( () => {
  name = "Jorts";
}, 0); // Run this callback ASAP
console.log(name); // Never "Jorts"
```

async results must be handled async

Async/Await

We will now be covering **promises**

A newer syntax allows you to avoid much of the syntax

- using **async** and **await** keywords

But we will NOT be using async/await

- They aren't bad at all
- But they hide what you need to know
- Once you know promises (after this course)
    - You can use async/await
    - And they will be easier

Callbacks

Async results are handled with callbacks

For DOM events you register a handler

- called when appropriate

For Service calls, same idea

- Also for filesystem calls on Node
- Or Database interactions on Node

Pyramid of Doom

When you have nested async callbacks:

```
connectToDatabase( "dbinfo", (db) => {
  db.authenticateToDatabase( "user", (db) =>
    db.prepareStatement( "sql", (stmt) => {
      stmt.executeStatement( "variable", (results) => {
        doSomething(results);
      });
    });
  });
});
```

It gets ugly, fast

Known as the **Pyramid of Doom**

- Nested callbacks make an indented triangle

Promises

Promises are a way to track callbacks

- A promise object
- Has a `.then()` method you can pass callbacks to
    - Returns a NEW promise object
- Promise is **pending**, **resolved**, or **rejected**
    - Callback passed to `.then()`
        - Called once promise resolved
        - Not called if promise rejects
- Promise returned by `.then()`
    - Resolves after callback runs

Simple promise example

```
console.log(1);
returnsAPromise().then( () => console.log(2) );
console.log(3);
```

always logs `1 3 2`. **Always**

Why?

Chained example

```
returnsAPromise()
  .then( () => console.log(1) )
  .then( () => console.log(2) );
```

Always `1 2`. **Always**. Why?

Resolve values

Promises might "resolve" with a value

- This value is passed to any `.then()` callbacks
- Value is **NOT** returned by the `then()` call

Examples use `Promise.resolve()`

- To get a resolved promise
- For examples
- Most "real" promises resolve differently

Resolve Value is not returned

```
const promise = Promise.resolve("hi");

const value1 = promise.then(
  (text) => console.log(`callback: ${text}`)
);

console.log(`from then: ${value1}`);
```

Results:

```
from then: [object Promise]
callback: hi
```

Remember: `then()` returns a new promise

Golden rule: Async results must be handled async

Resolve with what

- A promise resolves with a value
- `.then()` on a promise returns a new promise

What value does the new promise resolve with?

- The return value of the callback
- If that return value is a promise
    - uses resolution of THAT promise

Chaining

```
const one = Promise.resolve();
const two = one.then( () => console.log(1) );
const three = two.then( () => console.log(2) );
```

VS

```
Promise.resolve()
  .then( () => console.log(1) )
  .then( () => console.log(2) );
```

Chaining returns

Callback return value (default `undefined`!)

- Becomes resolve value of promise of that `then()`

```
const result = Promise.resolve(1)
  .then( val => {
    console.log(val);
    return val+1;
  })
  .then( val => {
    console.log(val);
    return val+1;
  })
  .then( val => {
    console.log(val);
    return val+1;
  });
```

What is `result`?

Trick question!

```
const result = Promise.resolve(1)
  .then( val => {
    console.log(val);
    return val+1;
  })
  .then( val => {
    console.log(val);
    return val+1;
  })
  .then( val => {
    console.log(val);
    return val+1;
  });
```

`result` is a PROMISE

- that resolved with value 4
- but `result` is NOT 4

No Pyramid!

```
connectToDatabase( "dbinfo", (db) => {
  db.authenticateToDatabase( "user", (db) =>
    db.prepareStatement( "sql", (stmt) => {
      stmt.execute( "variable", (results) => {
        doSomething(results);
      });
    });
  });
});
```

```
connectToDatabase("dbinfo")
  .then( (db) => db.authenticateToDatabase("user") )
  .then( (db) => db.prepareStatement("sql") )
  .then( (stmt) => stmt.execute("variable") )
  .then( (results) => doSomething(results) );
console.log("No callbacks have run yet!");
```

Promise resolve 1

```
const result = Promise.resolve(4)
  .then( (val) => val+1 );
result.then( val => console.log(val) );
```

What is the output?

Promise resolve 2

```
const result = Promise.resolve(4)
  .then( (val) => val+1 )
  .then( () => 2 )
  .then( (val) => val+3 );
result.then( val => console.log(val) );
```

What is the output?

Promise resolve 3

```
const result = Promise.resolve(4)
  .then( (val) => val+1 )
  .then( () => Promise.resolve(2) );
result.then( val => console.log(val) );
```

What is the output?

Promise resolve 4

```
const result = Promise.resolve(1)
  .then( (val) => val+1 )
  .then( () => Promise.resolve(4) )
  .then( (val) => Promise.resolve(val+4) );
```

What is the output?

Try/Catch is useless with Promises!

```
try {
  Promise.resolve()
    .then( () => {
      console.log(1);
      throw new Error("poop");
    });
} catch(err) {
  // Doesn't happen
  console.log(`caught ${err}`);
}
console.log(2);
```

Why? (Hint: output is `2 1`)

`.catch()`

Promises `catch()` method covers "failures"

- any thrown errors INSIDE a callback
- any returned **rejected** Promises

`.catch()` is passed a callback

- Just like `.then()`

`.catch()` also returns a promise

- **resolves** by default! (like `.then()`)
- Allows you to handle errors and keep going

`.catch()` example

```
Promise.resolve()
  .then( () => {
    throw new Error("poop");
  })
  .then( () => console.log('does not happen') )
  .catch( err => console.log(err) )
  .then( () => console.log('happy again') );
```

When a promise **rejects**:

Any promises created by a `.then()` on it

- Do not call their `.then()` callbacks
- Go to "rejected" status themselves
- Call any `.catch()` callbacks on themselves

Async/Await

A newer syntax is `async` and `await`

- A different way to manage promises
- Hides the `.then()` and `.catch()`
- Implicitly sets all following code to be async
- Allows try/catch

Do not use async/await for this course

I want you to become very comfortable with promises

- Hiding things makes that harder

Once out of this course, use async/await