Service Calls are Asynchronous (async)

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

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

So async may stand out as difficult

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

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: async 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

We will NOT be using async/await

- They ARE NOT bad!
- But they hide what you need to know
- Once you know promises (after this course)
    - You can use `async`/`await`
    - And you will understand them better

REPEAT: No `async`, no `await`

- **Great tools** once you understand promises
- Using them now hides promises

If you use `async` or `await` keywords in code:

- I will assume you have learned nothing
    - Code isn't showing *your* understanding
- Assignment will get very poor grade
    - or even `0`
- We don't have time to waste

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

Callbacks let you "wait" for situation

"Wait for click"

- *then* call this function

Can connect these waits:

"Connect to DB" (wait for connection)

- *then* "login to DB" (wait for response)
    - *then* "prepare a DB command" (wait)
        - *then* "execute w/params" (wait)

Pyramid of Doom

When you have nested async callbacks:

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

It gets ugly, fast

Known as the **Pyramid of Doom**

- Nested callbacks make an indented triangle

Promises are objects

- Track a status
  - **pending** (not done)
  - **resolved** (done successfully)
  - **rejected** (didn't succeed)
- Tracks callbacks to call
  - `.then()` - callbacks to call when **resolved**

Simple promise example

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

Always logs `1 3 2`

Always

Even if the promise is already **resolved**

Why never `1 2 3`?

Promises call callbacks asynchronously

Callbacks put in **queue**

- Never interrupt currently running code
- Just like JS event handlers

`.then()` itself is **synchronous**

- Like `.addEventListener`
- Passed callback called **asynchronously**
    - Like event handler

Promises So Far

Promises track a task and related callbacks

- Has a status: **pending**, **resolved**, or **rejected**
- Has a `.then()` method
- You pass callbacks to `.then()`
    - Callbacks called when promise **resolves**
    - If already resolved
        - Callbacks are **queued** immediately
        - Nothing interrupts current code
- If Promise rejects
    - `.then()` callbacks NOT called

Promise object .then() method

Calling `.then()` w/callback

- Returns a NEW promise object!
    - This promise returned by `.then()`
- Resolves after callback runs

Chaining

```
const one = Promise.resolve(); // returns a resolved promise
const two = one.then( () => console.log(1) );
const three = two.then( () => console.log(2) );
console.log(3);
```

A more compact version using **chaining**:

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

Result is **always** `3 1 2`

How many Promises involved?

When do they resolve?

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

Chained example

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

1. `Promise.resolve()` returns a resolved promise
2. First `.then()` returns a new pending promise
    - Cannot and DOES NOT call callback yet
3. Second `.then()` returns a new pending promise
4. `console.log(3)` runs as part of current sync code
5. Sync code done, `console.log(1)` can and does run
6. That pending promise is now resolved
7. `console.log(2)` now can and does run
8. That pending promise is now resolved

Chaining - Details

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

/* A */ const one = Promise.resolve();
/* B */ const two = one.then( () => console.log(1)/* D */);
/* C */ const three = two.then( () => console.log(2)/* E */);
```

- `A` runs, `one` is a **resolved** promise
- `B` runs, `two` is a **pending** promise
    - `D` placed on event queue as `one` is resolved
- `C` runs, `three` is a **pending** promise
- sync code is done, checks queue
- `D` runs, `two` is **resolved**, `E` placed on queue
- checks queue, `E` runs, `three` is **resolved**

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

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

Results:

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

Remember: `then()` returns a new pending 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

```
const one = Promise.resolve("Jorts");

const two = one.then( cat => {
    console.log(cat);
    return "Jean";
});

two.then( mystery => {
    console.log(mystery); // ???
});
```

Promises, Promises

- `.then()` on a promise returns a new promise
- The new promise resolves with return of callback
- **EXCEPT: If return value is a promise**
    - Uses resolution of THAT promise instead

```
const one = Promise.resolve("Jorts");

const two = one.then( cat => {
    console.log(cat);
    return Promise.resolve("Jean");
});

two.then( mystery => {
    console.log(mystery); // "Jean", not a promise of Jean
});
```

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) );
console.log(result);
```

What is the output?

Common Promise issues!

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

`result` is a PROMISE

- I'm not (just) being annoying
- This is a common misunderstanding
- `await` would match what you "expected"
- But would also silently delay further code!
- **We are not using `await` in this course**

Try/Catch catches synchronous errors

```
try {
  console.log(1);
  throw new Error("poop"); // deliberate error
  console.log(2); // won't run due to error
} catch(err) {
  console.log(`caught ${err}`)' // caught poop
}
console.log(3); // still runs after error
```

Result:

```
1
caught poop
3
```

Try/Catch is useless with Promises!

```
try {
  Promise.resolve()
    .then( () => {
      console.log(1);
      throw new Error("poop"); // deliberate error
      console.log(2); // won't run due to error
    });
} catch(err) {
  // Doesn't happen
  console.log(`caught ${err}`);
}
console.log(3);
```

Why? (Hint: output is `3 1`)

Try/Catch done by time async code runs!

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

- `Promise.resolve()` gives **resolved** promise
- `.then()` places callback on queue
- `.then()` returns **pending** promise
- `catch` not needed, try/catch complete
- Callback on queue runs, throws uncaught error
- The pending promise is now **rejected**

`.catch()`

Promises `catch()` method covers "failures"

- any fatal errors INSIDE a callback
    - (such promises become **rejected**)
- any returned **rejected** Promises

`.catch()` is passed a callback

- Just like `.then()`

`.catch()` also returns a promise

- **resolves** if callback runs (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 normal `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, **then** use `async`/`await`