# JavaScript Asynchronous Patterns

Async Await

# 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…

# Callbacks (again!)

# 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");
```

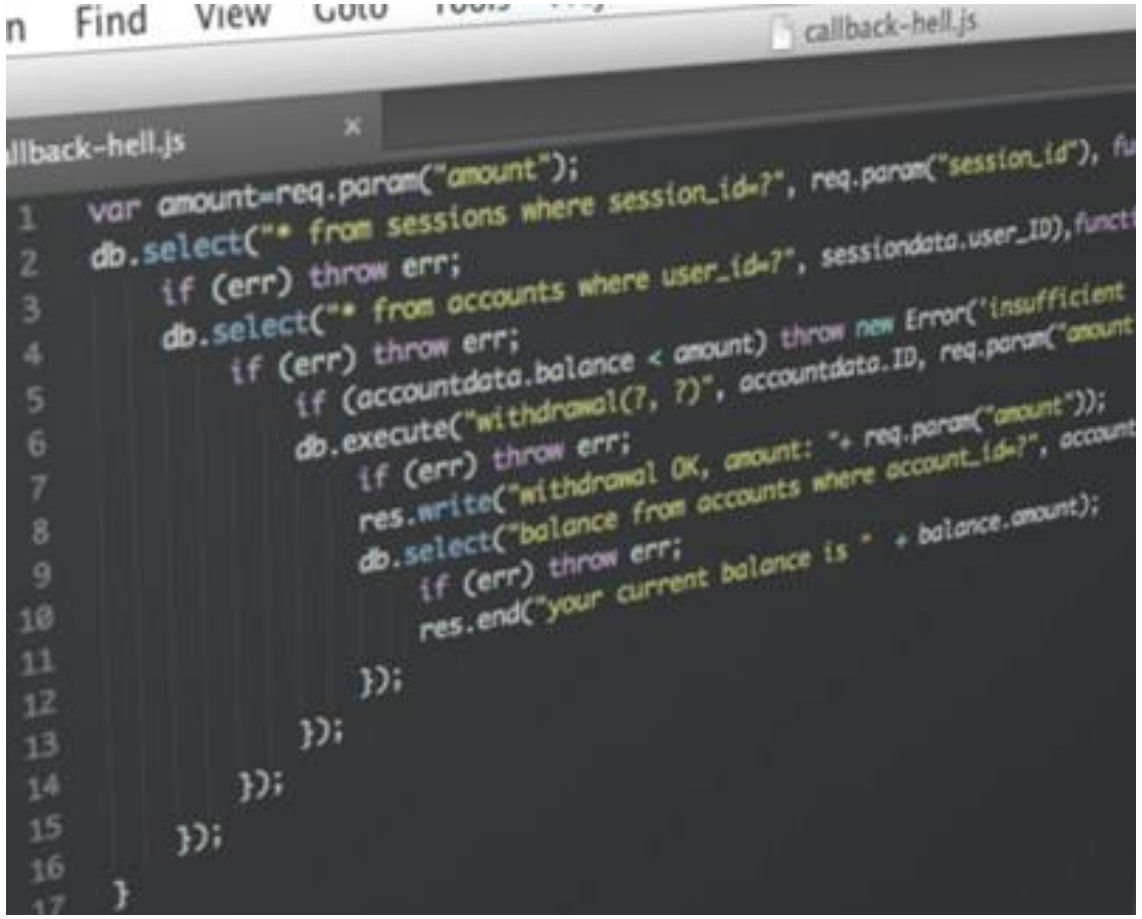
Possible Result

```
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
```

# Callback Hell – Multiple sequential requests




```
function loadUppThatApplication() {
    request("/api/getCustomer", function(response){
        var customerId = response.customer.id;
        request("/api/customer/accounts/"+customerId, function (response2) {
            request("http://facebook/pics/"+response2.faceBookUserName, function (response3) {
                showTheUserThatBeautifulUI(response3, function () {
                    byeByeSpinner();
                });
            });
        });
    });
}
```

Async () => { Await }

# 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

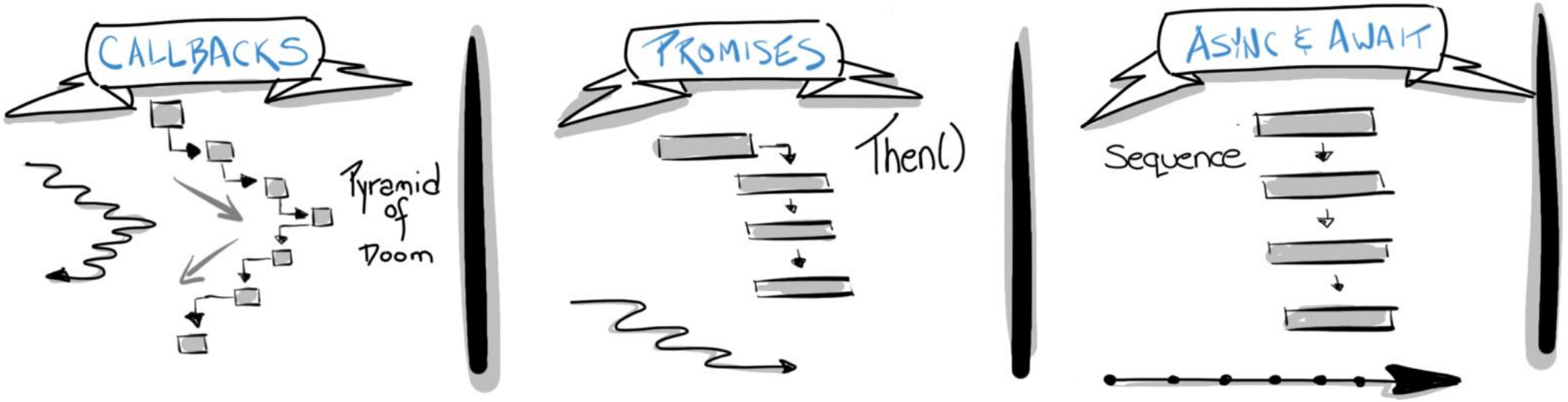
```
console.log("about to read...");

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 – 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");
}
```

# Further Asynchronous features…

# Wrapper Function for Error handling

- Express can’t handle promise errors/rejections
- An Async function always returns a Promise.
  - can *wrap* the async function to catch errors in Express Apps
  - Can drop try/catch in every async function
- Makes code more readable?

```
import asyncHandler from 'express-async-
handler';

const router = express.Router();

// Get all users
router.get('/', asyncHandler(async (req, res
) => {
    const users = await User.find();
    res.status(200).json(users);
}));
```

# 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!";
}
```

# ES6 Classes

# JavaScript Classes

- ECMAScript 2015, also known as ES6, introduced JavaScript Classes.
- Classes are special functions that facilitate the creation of constructors and prototype-based inheritance. Just like in functions, you can declare a class or express it:

```
class Person {
    constructor(name, surname) {
        this.name = name;
        this.surname = surname;
    }
}
```

```
var Person = class {
    function constructor(name, surname) {
        this.name = name;
        this.surname = surname;
    }
};
```

```
var person = new Person('Jack', 'Smith');
console.log(Person.name);
```

# Javascript Classes - inheritance

- To create a class inheritance, use the **extends** keyword.
- A class created with a class inheritance inherits all the methods from another class:

Repository.js

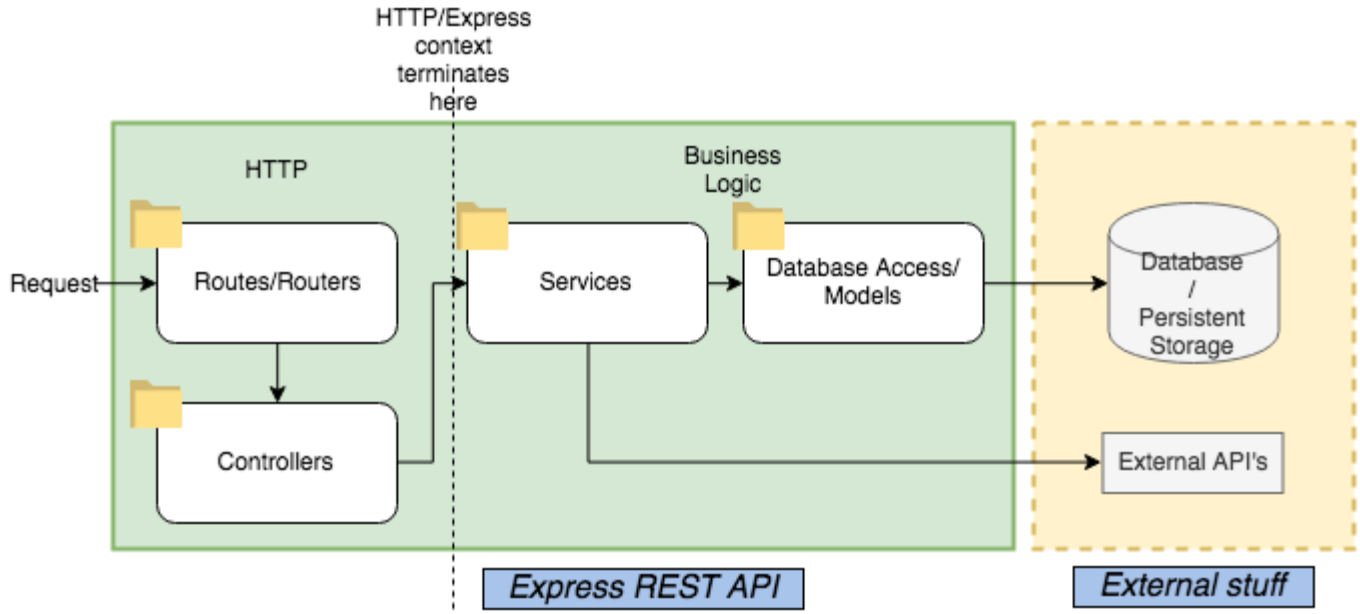
```
export default  class {

  persist(account) {
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');
  }

  merge(account) {
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');
  }

  remove(accountId) {
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');
  }

  get(accountId) {
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');
  }

  getByEmail(email) {
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');
  }

  find() {
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');
  }
```

```
import ContactsRepository from '../Repository'

export default class extends ContactsRepository {

  dataAsArray() {
    return Object.keys(this.data).map(key => this.data[key]);
  }

  constructor() {
    super();
    this.index = 1;
    this.data = {};
  }

  async persist(userEntity) {
    const row = Object.assign({}, userEntity);
    const rowId = this.index++;
    row.id = rowId;
    this.data[rowId] = row;
    return row;
  }

  merge(userEntity) {
    let row = this.data[userEntity.id];
    Object.assign(row, userEntity);
    return Promise.resolve(row);
  }
```

# Express Project Strucure/Architecture

# Express Project Structure

- Lots of different architectures/project Structures
  - Clean, Model/View/Controller, Domain Driven Dev.
- Our Rest API will follow this Layered Arcgitecure



# Simple Layered Approach

- HTTP logic layer
  - Routers:
    - handle the HTTP requests that hits the API and route them to appropriate controller
  - Controllers:
    - Processes request object, pull out data from request, validate, then send to service(s)
- Business logic Layer
  - Services: derived from use cases/business  requirements
  - Data Access: Repository/data store access

# 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