Table of Contents

BASH = Bourne Again Shell

Unix based cernal interface

Commands:

Ls – list cd~ root directory cd.. back one level

Cd – change directory

HOLD ALT to:put cursor anywhere using mouse

CTRL-A beginning of command line

CTRL-E end of command line

CTRL-U clears the entire command line

Mkdir – make directory

Touch – creates a new file

Open – uses default application such as text to open a file

Open -a – allows users to specify what application to use to open the file with

Rm – removes files

Pwd – prints the working directory you are in

Rm-r deletes directory including all files and folders within

Back-End Programming

The only things that a browser can work with are:

HTML CSS and js therefore:

There are frameworks such as **React Angular** and Vue and **GOOGLE TOOLKIT**

Frameworks are easements for delivering javascript to the browser for rendering

The back end is typically programmed in:

Java->Spring Framework

Rubi->Rails

PHP->Larval or Cake

C#->ASP.net

Python->Flask or Django

Javascript->**Node**.js

Breakdown by most popular backend frameworks:

Node.js 46%

React.js 44

Jquery 29.21

Express 23.19

Angular 23

ASP.NET Core 20

Vue.js 19

ASP.NET 16

Next.js 14

Django 13

Flask 13

Laravel 10

Angular.js 9.94

FastAPI 6

Blazor 5

PHP and Rubi are declining -> don't bother with learning either

Node.js

Why do you need a framework?

So you do not have to build all behaviours from scratch but can instead leverage pre-existing libraries:

Node.js is:

"An asynchronous event-driven JavaScript runtime, Node.js is designed to build sclable network applications"

In the past javaScript could only run on browsers well… no longer!

Node provides the javaScript runtime engine called V8 which frees us from only using JavaScript inside browsers!

Node.js is written uses something called the V8 engine written in C++ which writes javaScript to the browser…

This means that we can do anything we could normally did with Java now we do it in javaScript

Pros. For using NodeJs:

JS Fullstack is more easily managed

Scales

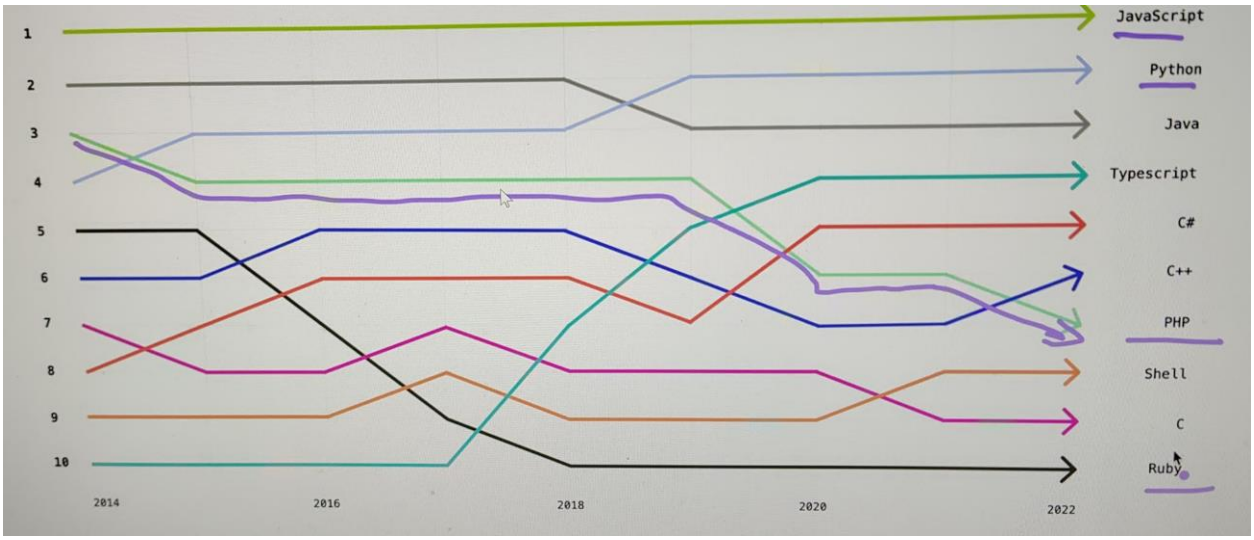
Non=Blocking – Does not suffer from starvation conditions as it does not rely on coded semaphores

Ecosystem: Thousands of open source modules are available!

LinkedIn ebay, uber Wikipedia twitter paypal Netflix its all nodejs as well as NASA

Machine Learning and Machine learning AI uses Python!!!!!!

See 100 days of python with flask



Node.Js

Node REPL or Read Eval Print Loop

It is a computer environment where user inputs (in the form of code) are read and evaluated, and then the results are returned to the user

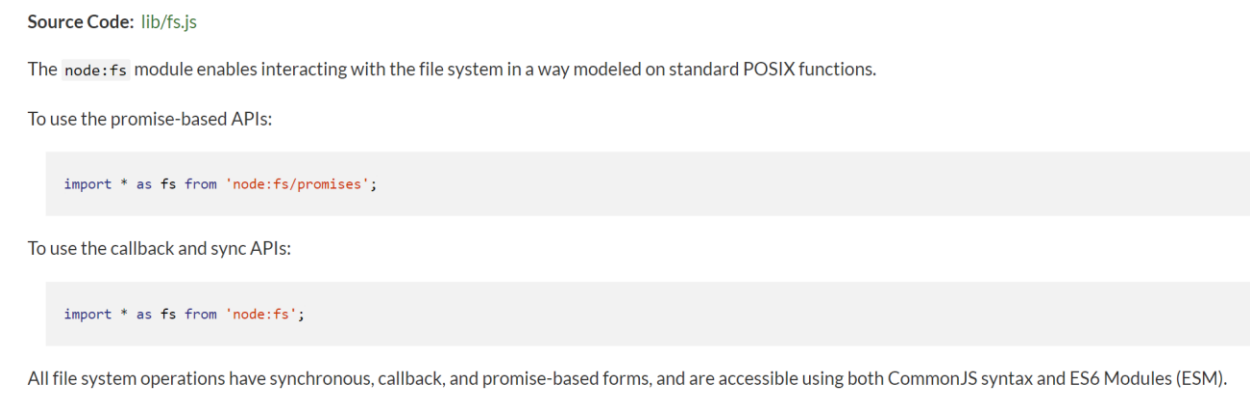
Native NODE modules came pre-bundled to provide additional access to objects and behaviors ready for usage such as:

Reading and writing to your file system

Carry out networking operations

Find out more about Node.js in: https://nodejs.org/docs/latest-v18.x/api/index.html

When using the system modules, import “fs” by either require or import than:



Source Code: lib/fs.js

The `node:fs` module enables interacting with the file system in a way modeled on standard POSIX functions.

To use the promise-based APIs:

```
import * as fs from 'node:fs/promises';
```

To use the callback and sync APIs:

```
import * as fs from 'node:fs';
```

All file system operations have synchronous, callback, and promise-based forms, and are accessible using both CommonJS syntax and ES6 Modules (ESM).

```
import { writeFile } from 'node:fs/promises';
import { Buffer } from 'node:buffer';

try {
  const controller = new AbortController();
  const { signal } = controller;
  const data = new Uint8Array(Buffer.from('Hello Node.js'));

  const promise = writeFile('message.txt', data, { signal });

  // Abort the request before the promise settles.
  controller.abort();

  await promise;
} catch (err) {
  // When a request is aborted - err is an AbortError
  console.error(err);
}
```

(NPM) Node Package Manager

Community tool library found here. This is a place where other people have developed and uploaded code for node

What is ESM (ECMAScript Modules)?

- ESM (EcmaScript modules) is a newer system which has been added to the JavaScript specification. Browsers already support ES modules, and Node is adding support. Let’s take an in-depth look at how this new module system works. When you’re developing with modules, you build up a graph of dependencies.

ESM provides the ability to use the “import” keyword sicj as in:

Import * as fs from ‘node:fs/promises’;

However to use ECMAScript in my code **I have to add “type”:”module” in package.json** configuration text file.

QR code generator project

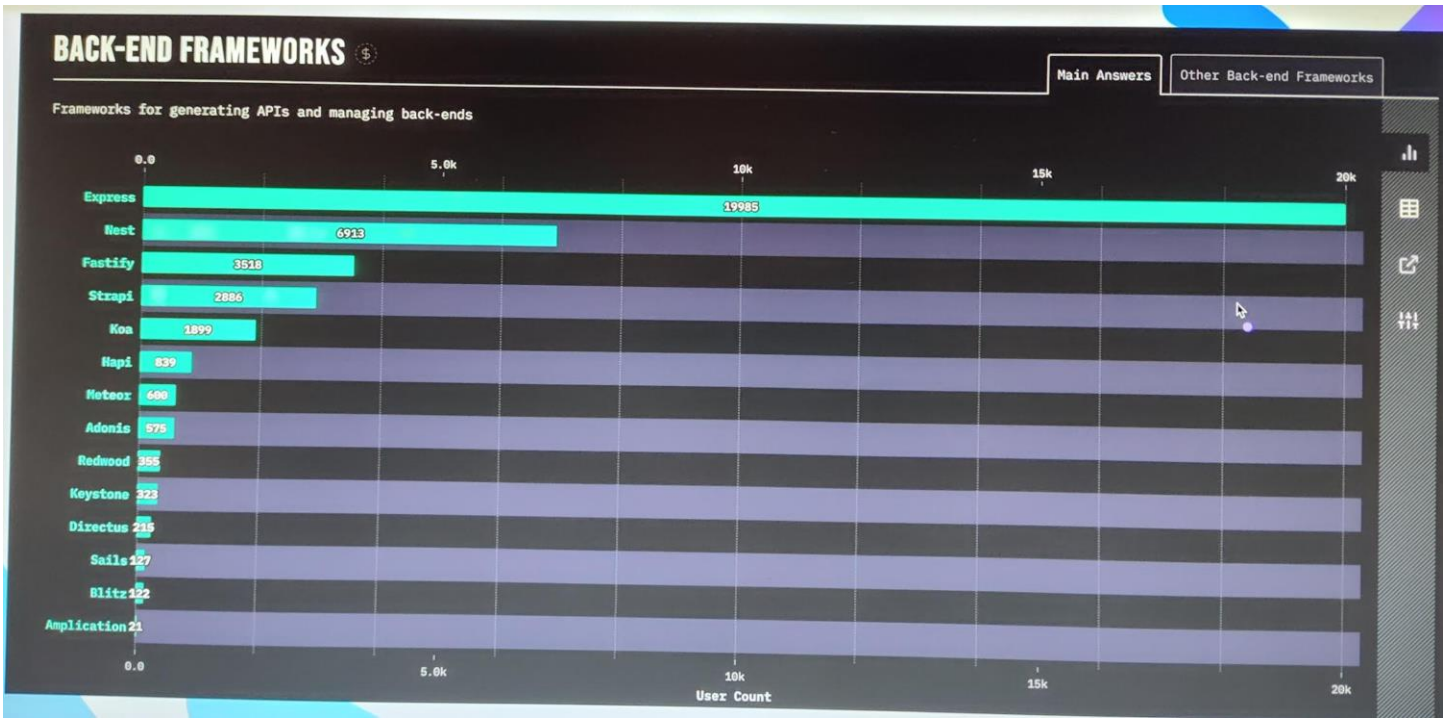
In order to create this project we need two packages from npm:

One is called ‘Inquirer’ -> This allows us to get inputs from the user

and ‘qr-image’ -> which generates png images for our local file system

1. Navigate to the working directory inside visual studio terminal
2. Run the ‘npm init’ command in terminal
3. Import inquirer and qr-image
4. Edit the newly generated configuration json file
    a. Add “Type”:”module”, to allow ‘import’
5. Determine the best usage of the inquirer library from examples provided

```
/*
1. Use the inquirer npm package to get user input.
2. Use the qr-image npm package to turn the user entered URL into a QR code image.
3. Create a txt file to save the user input using the native fs node module.
*/
import inquirer from 'inquirer';
import qr from 'qr-image';
import fs from 'node:fs';
const questions = [
    {
        type: 'input',
        name: 'comments',
        message: 'Please enter URL that you wish to have converted to QR code',
        default: 'google.com',
      }
    ];
  inquirer.prompt(questions).then((answers) => {
    var jsonStringObject = JSON.stringify(answers, null, '  ');
    var jsonObject = JSON.parse(jsonStringObject);
    console.log('\nWhole JSON:');
    console.log(jsonStringObject);
    console.log('\nParsed JSON:');
    console.log(jsonObject.comments);


    fs.writeFile('userData.txt',jsonObject.comments, 'utf8', (err) => {
      if (err) throw err;
      console.log('The file has been saved!');
    });
    var qr_png=qr.image(jsonObject.comments, { type: 'png' });
    qr_png.pipe(fs.createWriteStream('myQR.png'));
    var svg_string = qr.imageSync('myQR', { type: 'png' });
  });
```



Express Framework – A framework on top of Node

Never forget that Node is a runtime environment

Node is what enables us to run javaScript on a computer and not just on a browser.

Express is a framework which allows us to build the backend of our server

The power of Node and Express together allows us to develop servers

Steps to creating an Express Server

1. Create Directory
2. Create index.js file
3. Initialize NPM
4. Install the Express package
5. Write Server application in index.js
6. Server should be able to handle “/”(home),”.contact” and “/about” endpoints
7. Use nodemon to Start Server

For information about how to install express package go to: https://expressjs.com/en/starter/installing.html

Use this command to see what is listening on your computer to see what ports are open

netstat -ano | findstr "LISTENING"

findstr is equal to the unix ‘grep’ command

Express Route parameters

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the req.params object, with the name of the route parameter specified in the path as their respective keys for example:

```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }
```

To define routes with route parameters, simply specify the route parameters in the path of the route as shown:

```
app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params)
})
```

Since the hyphen (-) and the dot (.) are interpreted literally, they can be used along with route parameters for useful purposes.

```
Route path: /flights/:from-:to
Request URL: http://localhost:3000/flights/LAX-SFO
req.params: { "from": "LAX", "to": "SFO" }
Route path: /plantae/:genus.:species
Request URL: http://localhost:3000/plantae/Prunus.persica
req.params: { "genus": "Prunus", "species": "persica" }
```

To have more control over the exact string that can be matched by a route parameter, you can append a regular expression in parentheses (`()`):

```
Route path: /user/:userId(\d+)
Request URL: http://localhost:3000/user/42
req.params: {"userId": "42"}
```

Route Handlers

Express allows us to use route parameters in order to create dynamic routes so if I wanted to have just the index page act as the source of all routing I can provide multiple callback functions within the index response method which behave like middleware to handle a request. The only exception is that these callbacks might invoke next('route') to bypass the remaining route callbacks. I can use this mechanism to impose preconditions on a route then pass control to subsequent routes if there's no reason to proceed with the current route. Route handlers can be in the form of a function , an array of functions, or combinations of both.

A combination of independent functions and arrays of functions can handle a route. For example:

```
const cb0 = function (req, res, next) {
  console.log('CB0')
  next()
}

const cb1 = function (req, res, next) {
  console.log('CB1')
  next()
}

app.get('/example/d', [cb0, cb1], (req, res, next) => {
  console.log('the response will be sent by the next function ...')
  next()
}, (req, res) => {
  res.send('Hello from D!')
})
```

Response methods

The methods on the response object (`res`) in the following table can send a response to the client, and terminate the request-response cycle. If none of these methods are called from a route handler, the client request will be left hanging.

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

app.route()

You can create chainable route handlers for a route path by using `app.route()`. Because the path is specified at a single location, creating modular routes is helpful, as is reducing redundancy and typos. For more information about routes, see: Router() documentation.

Here is an example of chained route handlers that are defined by using `app.route()`.

```
app.route('/book')
  .get((req, res) => {
    res.send('Get a random book')
  })
  .post((req, res) => {
    res.send('Add a book')
  })
  .put((req, res) => {
    res.send('Update the book')
  })
```

express.Router

Use the `express.Router` class to create modular, mountable route handlers. A `Router` instance is a complete middleware and routing system; for this reason, it is often referred to as a “mini-app”.

The following example creates a router as a module, loads a middleware function in it, defines some routes, and mounts the router module on a path in the main app.

Create a router file named `birds.js` in the app directory, with the following content:

```
const express = require('express')
const router = express.Router()
```

```
// middleware that is specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
})
// define the home page route
router.get('/', (req, res) => {
  res.send('Birds home page')
})
// define the about route
router.get('/about', (req, res) => {
  res.send('About birds')
})

module.exports = router
```

Then, load the router module in the app:

```
const birds = require('./birds')

// ...

app.use('/birds', birds)
```

The app will now be able to handle requests to `/birds` and `/birds/about`, as well as call the `timeLog` middleware function that is specific to the route.

HTTP Request Vocabulary

Get – Request a resource from server

Post – Sending a resource to the server like a form

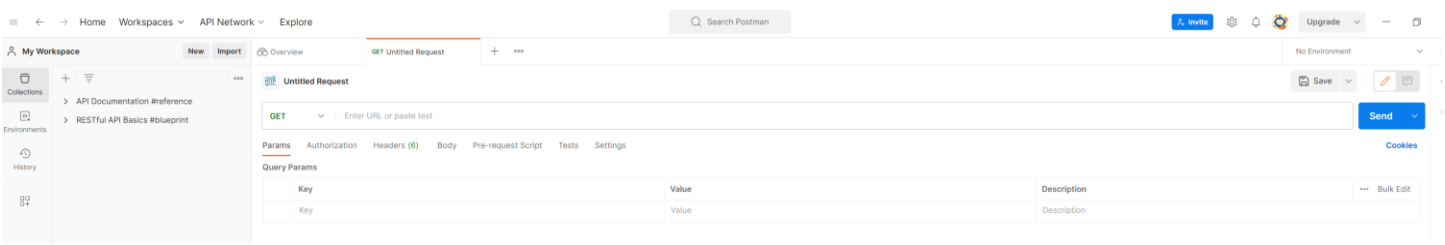
Put – To update something by replacing an entire resource

Patch – To update something by replacing a component of a resource

Delete – Dete resource

Postman and HTTP Routehandlers

Postman allows you to synthesize interactions with the server tailoring the type of HTTP interaction with fields or properties that can be assigned with values.



HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes found in: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

200 is successful get server response for example

You can see all your response codes in the network tab of the browser in dev mode

Middleware

Responsible for processing,logging,authentication,process errors and determine their final routing (get,post,put,patch,delete)

Body Parser

Found in: https://www.npmjs.com/package/body-parser

Node.js body parsing middleware.

Parse incoming request bodies in a middleware before your handlers, available under the req.body property.

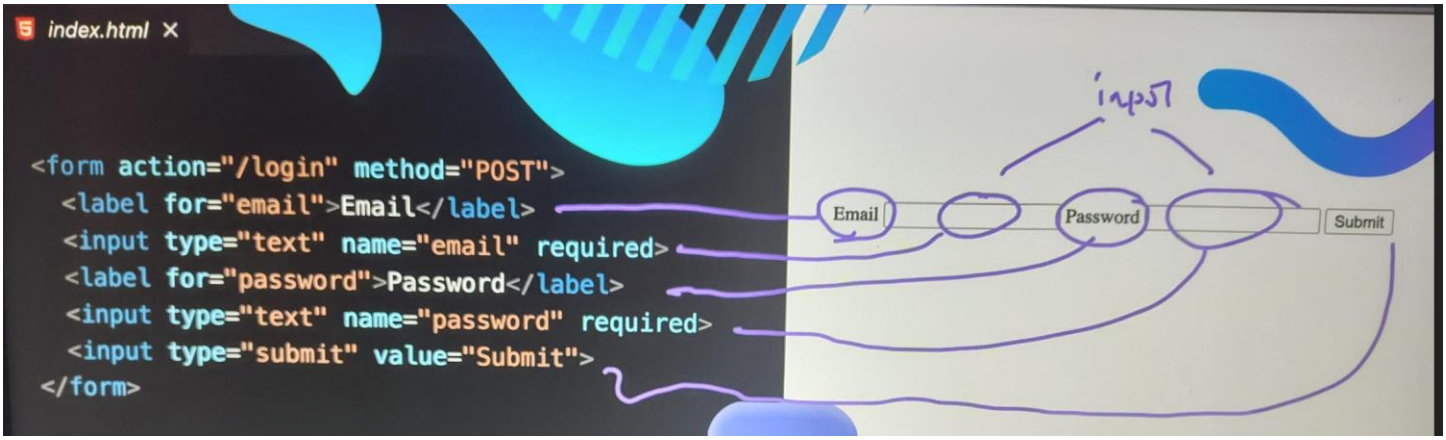
Gives our request the 'body' property, so in the case of an authentication form you could use Body-Parser to pass the parsed body to the back end rather than the entire html code which could look like this:

```
<form action="/login" method="POST">
  <label for="email">Email</label>
  <input type="text" name="email" required>
  <label for="password">Password</label>
  <input type="text" name="password" required>
  <input type="submit" value="Submit">
</form>
```

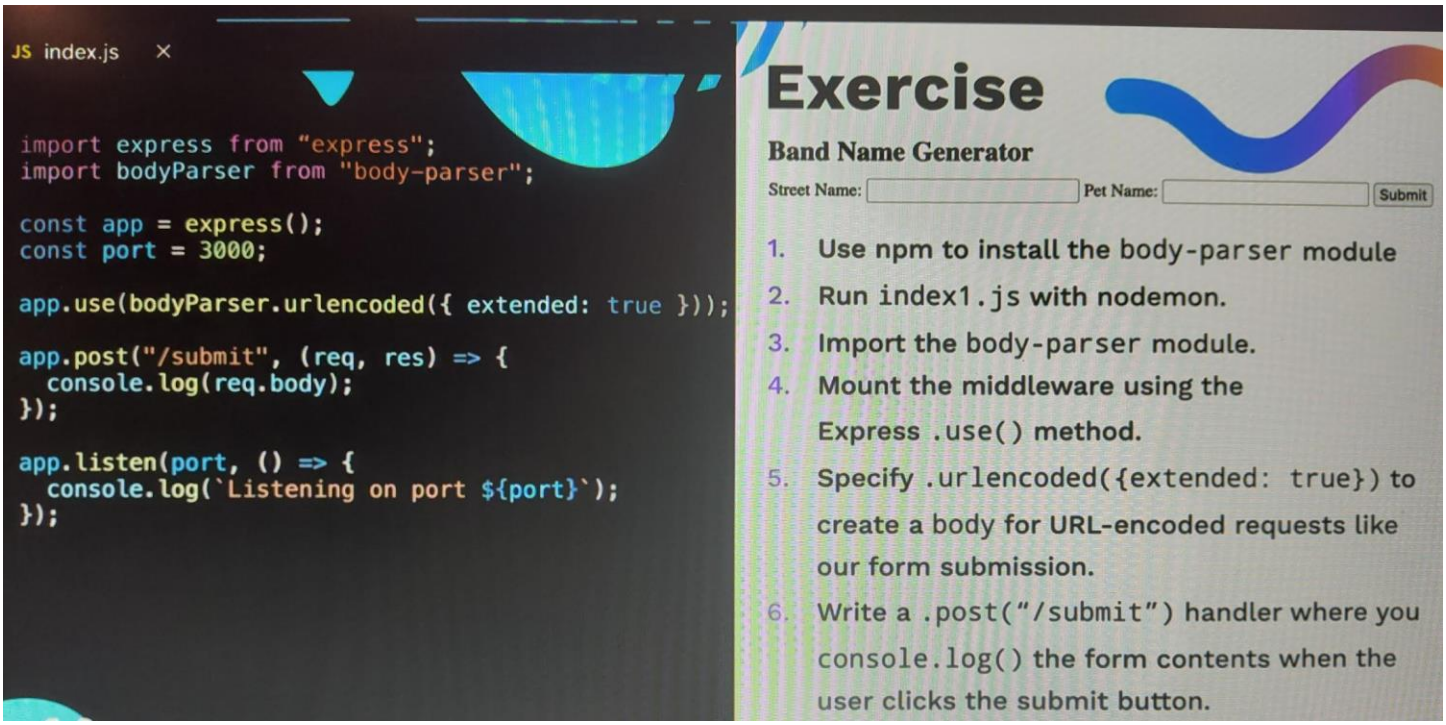
The 'Form' element starts with a form tag typicaly with an inner property of action which indicates how you want your server to route your request as well as a method which indicates how you want this form to be processed by the bac-end.

As seen above notice the typical pairing of the label element followed by an input element. The input element will have a property of type which could be text or radio button or something similar. A type submit input is a button

The name property in the Input is the key element of the key value pair. The required element is there to ensure that a key exhists or the server will not process this request.



Middleware Exercise

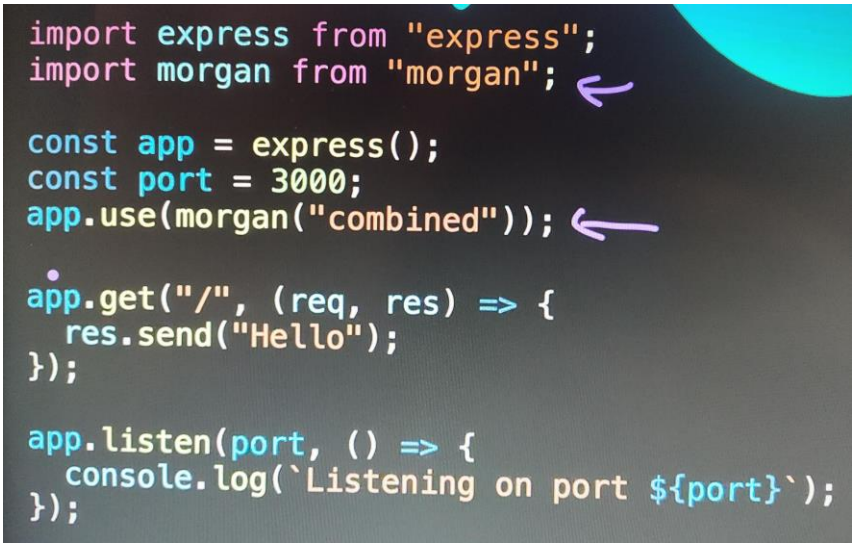


1. Close server if running Lookup the syntax for installing body-parser found to be: npm install body-parser
2. (assuming inside working directory) terminal command nodemon index1.js
3. Add import bp from "body-parser" to index1.js https://www.npmjs.com/package/body-parser
4. Lookup https://expressjs.com/en/guide/using-middleware.html
5. Add app.use(bp.urlencoded({extended: true}));
6. Add app.post("/submit") as above

Custom Middleware

The body-parser belongs to the pre-processing category of middleware.

Morgan is primarily used for logging. https://www.npmjs.com/package/morgan the requests that come to the server



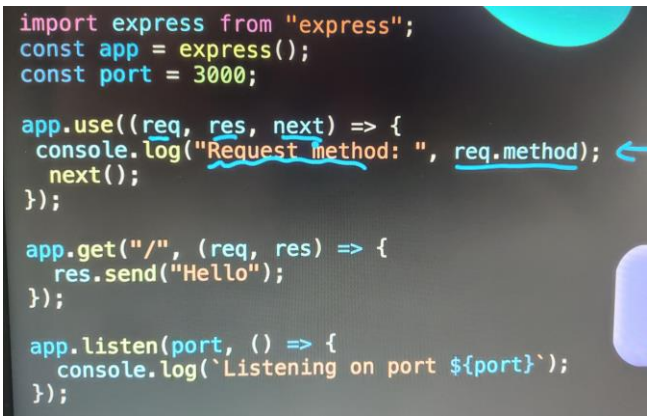
```
import express from "express";
import morgan from "morgan";

const app = express();
const port = 3000;
app.use(morgan("combined"));

app.get("/", (req, res) => {
  res.send("Hello");
});

app.listen(port, () => {
  console.log(`Listening on port ${port}`);
});
```

To add Morgan, do the following:

1. Use npm to install morgan package
2. Start server by envoking nodemon
3. Import the morgan module with the type of logging you need
4. Mount the middleware using Express.use() method
5. Test the logging on localhost with Postman.
6. Add the next() function as the next stage in processing

Create Custom Middleware

The order of execution becomes very important in cases when you want to authenticate first and don't forget to include the next() function or the server will hang.



```
import express from "express";
const app = express();
const port = 3000;

app.use((req, res, next) => {
  console.log("Request method: ", req.method);
  next();
});

app.get("/", (req, res) => {
  res.send("Hello");
});

app.listen(port, () => {
  console.log(`Listening on port ${port}`);
});
```

```
import express from "express";

const app = express();
const port = 3000;

function logger(req,res,next){
  console.log("Request method: ",req.method);
  next();
}
app.use(logger);

app.get("/", (req, res) => {
  res.send("Hello");
});

app.listen(port, () => {
  console.log(`Listening on port ${port}`);
});
```

Middleware Project Band Name Generator

1. Install the right packages to install the right modules in order to serve up the index.html when a 'get' is made
2. On posting through submit send back <h1> Your Band is <h1> then <h2> the result.

```
3. import express from "express";
4. import bp from "body-parser";
5. import { dirname } from "path";
6. import { fileURLToPath } from "url";
7.
8. const app = express();
9. const port = 3000;
10.const __dirname = dirname(fileURLToPath(import.meta.url));
11.app.use(bp.urlencoded({extended: true}));
12.
13.app.listen(port, () => {
14.  console.log(`Listening on port ${port}`);
15.});
16.app.post("/submit", (req,res)=>{// the /submit has to match the route or action inside
   public/index.html 'form' and post is the method
17.  console.log(req.body);//req.body only possible thanks to middlware (app.use(body-
   parser)...)
18.  res.send("<h1>Your band name is:</h1>"+"<h2>"+req.body.street+req.body.pet+"</h2>");
19.
20.})
21.app.get("/", (req, res) => {
22.  console.log(__dirname + "/public/index.html");
23.  res.sendFile(__dirname + "/public/index.html");
24.});
```

4.0 EJS embedded javaScript or how to create dynamic files

Answers the question how to render an object from the back-end to an html. We do it with embedded javaScript code.

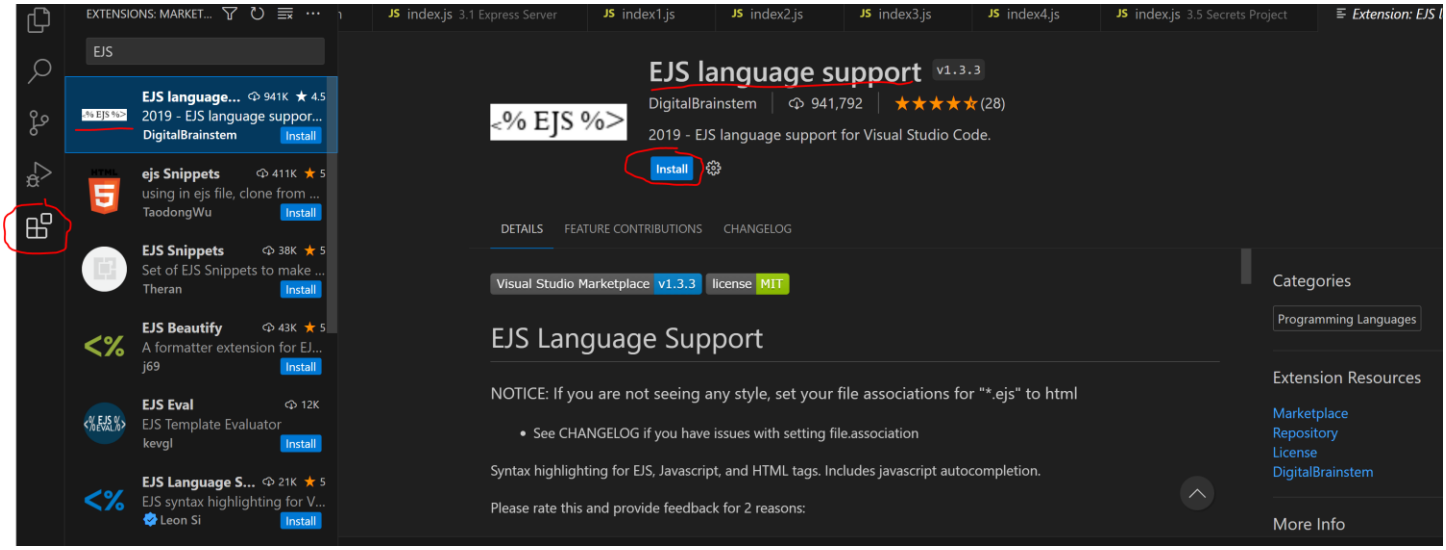
How does EJS work?

We use:

```
res.render("index.ejs",{ name: req.body["name"] });});
```

Steps to EJS

1. Install EJS language support



2. Create a new folder for example: "4.0 EJS" and cd into it.
3. Initialise NPM and install express and ejs. (npm i express ejs)(add: "type":"module")
4. Create index.js, views/index.ejs (views folder is required to use EJS templates)(use ! to populate standard HTML)
5. Use the JS getDay() method to build a website that gives advice based on the day of the week. (new Date())

4.1 EJS Tags Templating or running JS in HTML

What is EJS

EJS is a template system. You define HTML pages in the EJS syntax and you specify where various data will go in the page. Then, your app combines data with the template and "renders" a complete HTML page where EJS takes your data and inserts it into the web page according to how you've defined the template. For example, you could have a table of dynamic data from a database and you want EJS to generate the table of data according to your display rules. It saves you from the drudgery of writing code to dynamically generate HTML based on data.

EJS is compatible with Express for back-end use as it hooks into the View engine architecture that Express provides and lets you render web pages to the client with res.render() in Express.

FYI, there are dozens of competing template systems for use in node.js. EJS is a popular one and people typically choose one based on features that match your needs, how their layout language fits what you want to use, what seems easiest to you to use, etc... I've used Pug, Handlebars, Nunjucks and EJS. Nunjucks is my current favorite.

EJS (along with all the other competing template engines) allows you to generate full-blown HTML pages which certainly enables a "proper front-end".

EJS is a tool for generating web pages that can include dynamic data and can share templated pieces with other web pages (such as common headers/footers). It is not a front-end framework. While EJS can be used by client-side Javascript to generate HTML on the client-side, it is more typically used by your back-end to generate web pages in response to some URL request. EJS is not a client-side framework like Angular or React and does not dictate what client-side framework you do or don't use (if any). It is mostly covers a separate solution space.

EJS Tags

`<%= variable %>`	JS Output
`<% console.log("hello") %>`	JS Execute
`<%- <h1>Hello</h1> %>`	Render HTML
`<%% %%>`	Show <% or %>
`<%# This is a comment %>`	Stop Execution
`<%- include("header.ejs") %>`	Insert another EJS file

Running JS in HTML

JS index.js

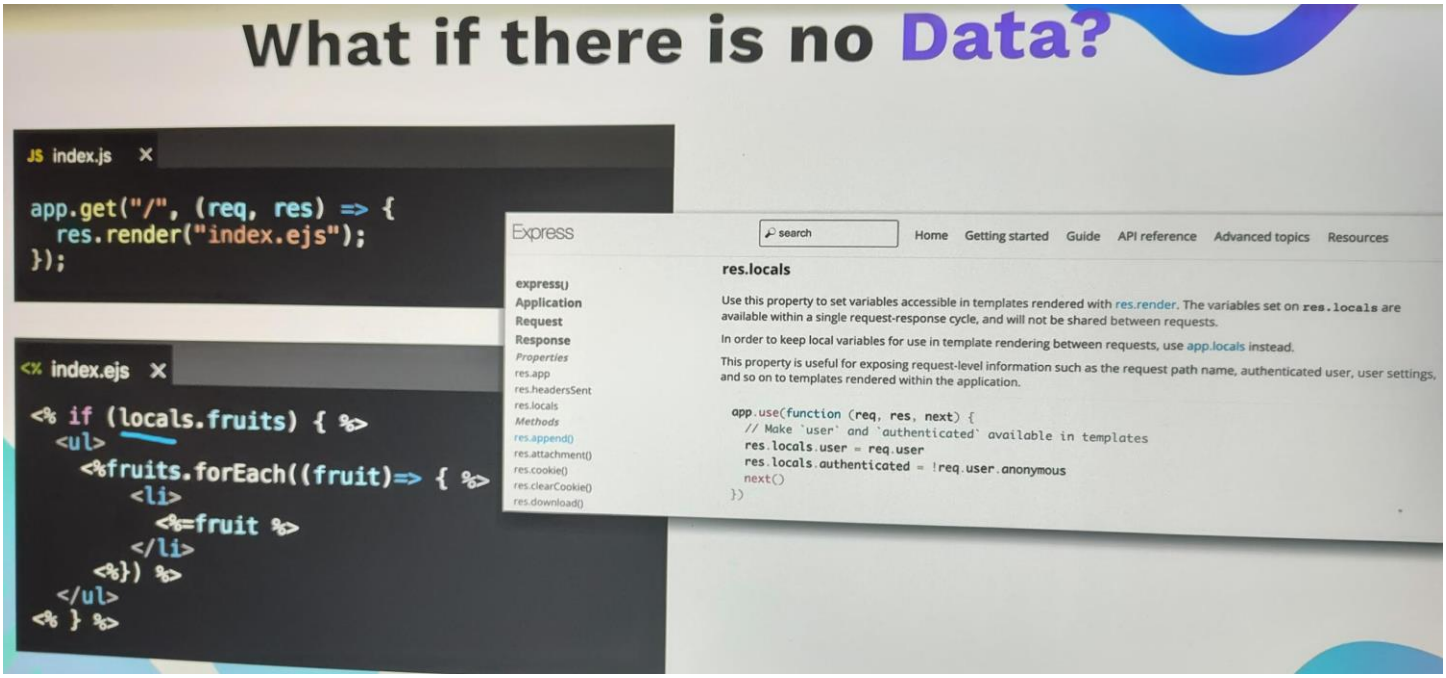
```
let bowl = ["Apples", "Oranges", "Pears"];

app.get("/", (req, res) => {
  res.render("index.ejs", { fruits: bowl });
});
```

<% index.ejs

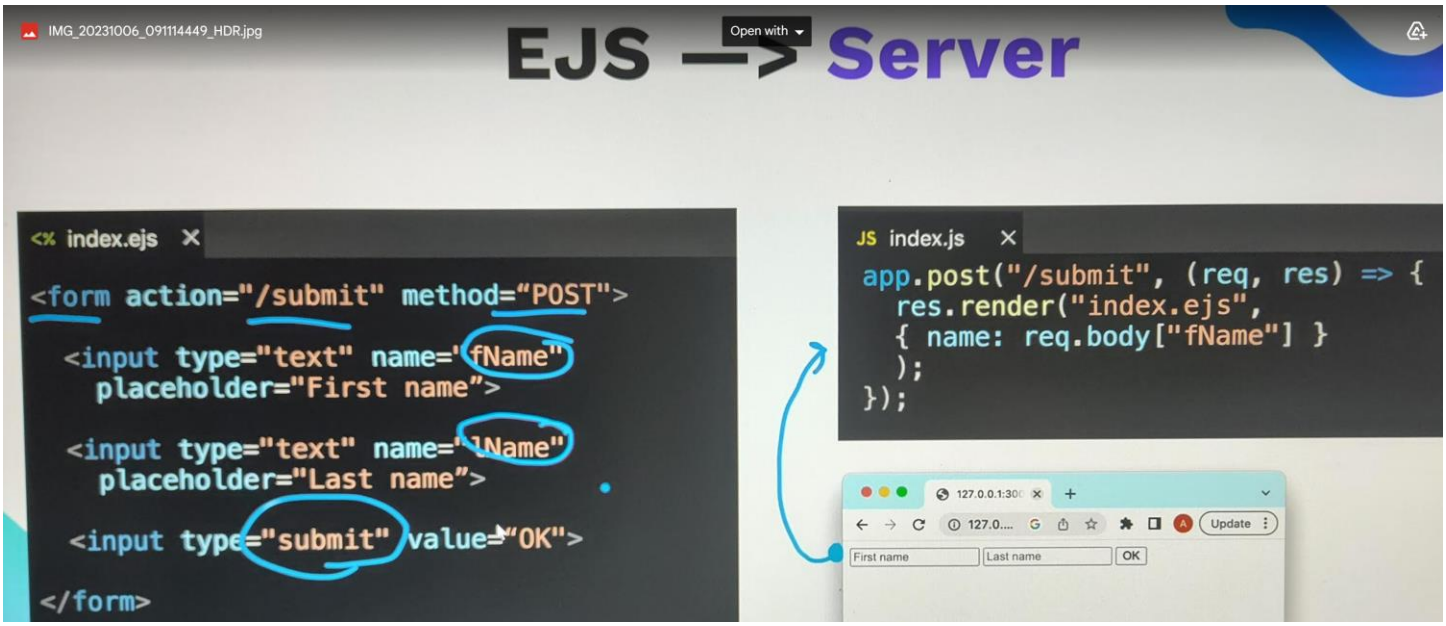
```
<body>
  <ul>
    <% for(let i=0; i<fruits.length; i++) { %>
      <li>
        <%= fruits[i] %>
      </li>
    <% } %>
  </ul>
</body>
```

only code

output Ejs tag → HTML



The locals variable always exists allowing us to test for its content without risking page crash



Partials and Layouts

To use a static resource such as an image or css file you structure it into the public folder and add a pointer to the public folder using express middleware assigned to “app” variable looking like:

App.use(express.static(“public”)); where this public is the reference to the name of the folder containing static files.

Then you link to each within the ejs file

<link rel=”stylesheet” href=”/styles/layout.css”> where these urls are relative to your “public” folder.

Partials

Example: <%- include(“header.ejs”)%>

Example: <%- include(“footer.ejs”)%>

```
<%- include("partials/header.ejs") %>
<!----the header url is always relative to the views folder -->
<!--  the header.ejs contains links that are relative to the public folder -->
```

Setup

1. Make sure that static files are linked using middleware to and the CSS shows up. Create this link to all static resources and css files located within the public folder inside index.js:
    a. **App.use(express.static(“public”));** Without this line the relative link to css and resources in header partial will fail.
2. Add:

```
app.get("/", (req, res) => {
  res.render("index.ejs");
});
```

    a. To Render the home-page
3. Add the CSS link in header.ejs

Todo list Web application

Project Description

The goal of this project is to create a ToDo List web application using Node.js, Express.js, and EJS. The application will **allow users to create and view tasks**. Tasks will not persist between sessions as no database will be used in this version of the application. **Styling will be an important aspect of this project to ensure a good user experience**.

Features design (17:15 10/07/2023)-> (17:47 10/07/2023)[30min]

1. **Task Creation:** Users should be able to create new tasks.
    a. Two ejs files:
        i. Index.ejs the default page with todays tasks
        ii. WorkRelatedTasks.ejs page with work related tasks
    b. 2Tasks array of objects declared inside index.js and passed to locals.tasks in index.ejs form and WorkRelatedTasks Form and therefore objects to include the following:
        i. Radio done button
        ii. Date object (index.ejs file includes date picker)
    c. New tasks are added by filling out the form and submit, posting to server
2. **Task Viewing:** Users should be able to view a list of all their tasks.
    a. Top of index.ejs must contain a 'todays tasks' icon and a 'future tasks' icon
    b. The right of the screen will contain a table with top row identifying each column
3. **Task Completion**: Users should be able to strike through their completed tasks in the todo list. Tasks do not need to be deleted.
    a. Pushing the Done Radio causes the text of that task to be struck trhrough
4. **Segmented Lists**: Users should be able to access at least 2 lists, one for the day's tasks and another for their work related tasks. (See example website)
    a. **See task creation**

3. **Styling:** The application should be well-styled and responsive, ensuring a good user experience on both desktop and mobile devices.

Technical Requirements

1. **Node.js & Express.js:** The application will be a web server built using Node.js and Express.js. Express.js will handle routing and middleware.

2. **EJS:** EJS will be used as the templating engine to generate dynamic HTML based on the application's state.

Timeline

Hour 0: Planning

- Gather content and design ideas, create wireframes and mockups.

Hour 1: Setup

- Set up the project repository, initialize the Node.js application, and install necessary dependencies (Express.js, EJS).
- Plan out the application structure, including routes, views, and static files.
- Set up the Express.js server and define the necessary routes.

Hour 2-3: Implementing Features

- Implement the task creation feature. This includes creating the form on the home page and handling the form submission in the server.
- Implement the task viewing feature. This includes displaying the list of tasks on the home page.
- Implement the task cross-off feature. This allows the user to check a checkbox and make the task appear striked through.
- Implement the alternative list to show a different set of todo tasks.
- Test the application to ensure that task creation and viewing are working correctly.

Hour 4-5: Styling and Polishing

- Style the application. This includes creating a CSS file, linking it to your EJS templates, and writing CSS or using Bootstrap/Flexbox/Grid to style the application.
- Test the application on different devices and browsers to ensure the styling works correctly.
- Fix any bugs or issues that came up during testing.

How do I onclick/addeventlistener to an EJS template?

Answer can be found here: https://www.w3schools.com/jsref/event_onclick.asp

how do i access the document dom object in ejs?

My EJS template is rendered on the server, not in the browser, so you don't have access to the DOM in your templates.

so what do i do for setting eventListeners on the client-side?

Client side JS code (regular <script> tags).

Git

Git commands

Set Project folder: navigate to top of project and run: git init

Add artifacts to staging area git add (name of file or '.') to get everything.

Commit artifacts: git commit

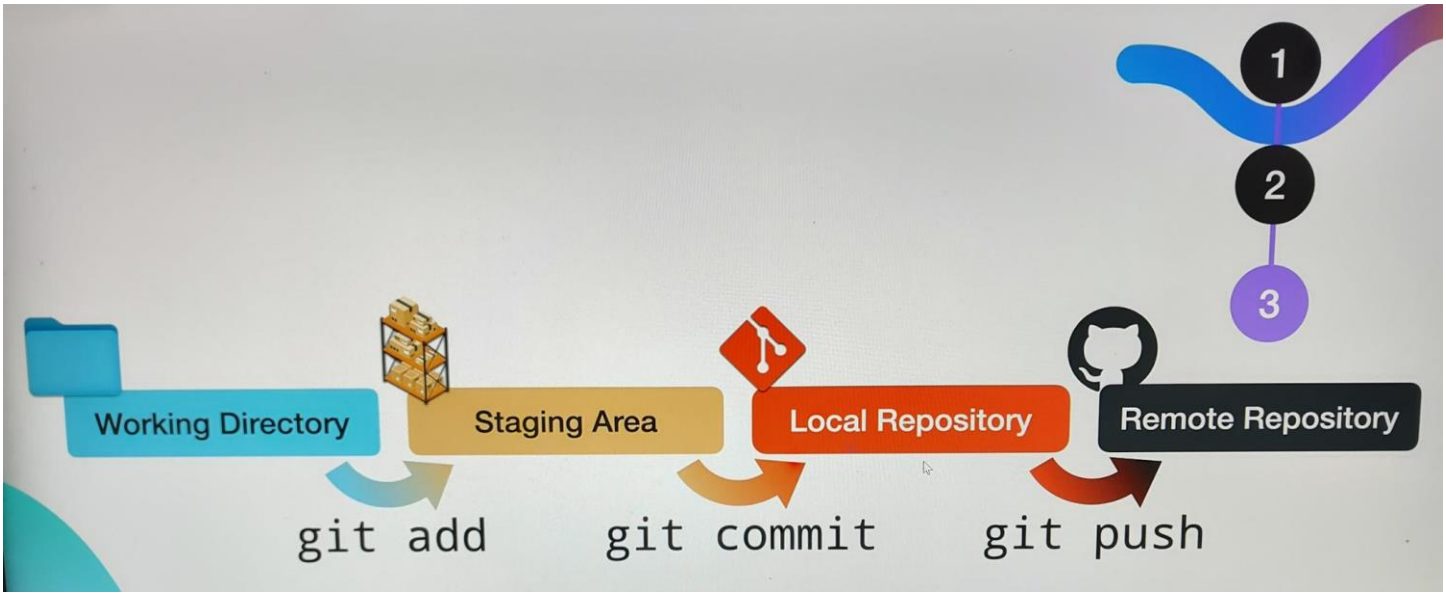
Restore previous version: git checkout

Show what has been committed: git log

Remove files from staging area git rm –cached -r (identify file or . for all files)

git push -u origin main

GitHub and remote repositories



Git Ignore to set rules to prevent accidental push of sensitive information

1. Create a ".gitignore" text file where you will store all the files git should ignore
2. Use '*' as a typcal wild card to specify types sharing common type or name to ignore
3. https://github.com/github/gitignore/blob/main/Node.gitignore has great templates for the .gitignore file.

Git Clone

1. Run "git clone url" command pulls down every artifact of a particular project to your computer.
2. Run npm init

Git Branching

To Branch use: git branch (name of branch without quotes)

To read all branches in project use: branch (the return will contain a branch with an asterix – this is the branch you are currently on.

To switch to a branch use: git checkout (branch name)

Git Merging

To merge a branch with the branch you are on run: git merge (target branch name), this will not eliminate the target branch.

Typical Git Process

1. Git init (top of project folder)
2. Create new Project name in Git Hub
3. Git add views/.
4. Git add public/.
5. Git add index.js
6. Git commit -m “something about this commit”
7. Visual Studio terminal:
    a. git remote add origin https://github.com/AirLinkDev/”GitHubCreatedName”.git
    b. git branch -M main
    c. git push -u origin main (only this one when updating next commit)

API's

REST:API is the most commonly used API in exhistance

You use the HTTP protocol (GET,POST,PUT,PATCH,DELTE) to communicate with the API

You can use POSTMAN to test the API

Get familiar with a generic REST api by looking at https://wheretheiss.at/w/developer

An API url is composed of the base url, the endpoint, and the query which is a ?key =value pair each query delimited by '&'

http://bored-api.appbrewery.com/random?query=value&query2=value

Each resource has a unique key designation so to pull that resource we can use an endpoint followed by '/uniqueKey'

JASON or JavaScript Object Notation

It is structured after a javaScript object making it serializable and client readable. The only difference is that JSON objects need quotation marks around their keys while javaScript objects do not. JavaScript objects have a more relaxed form because they are interpreted by code interpreters and code editors and do not have to be deserialized first.

To serialize JASON object into flat packed string

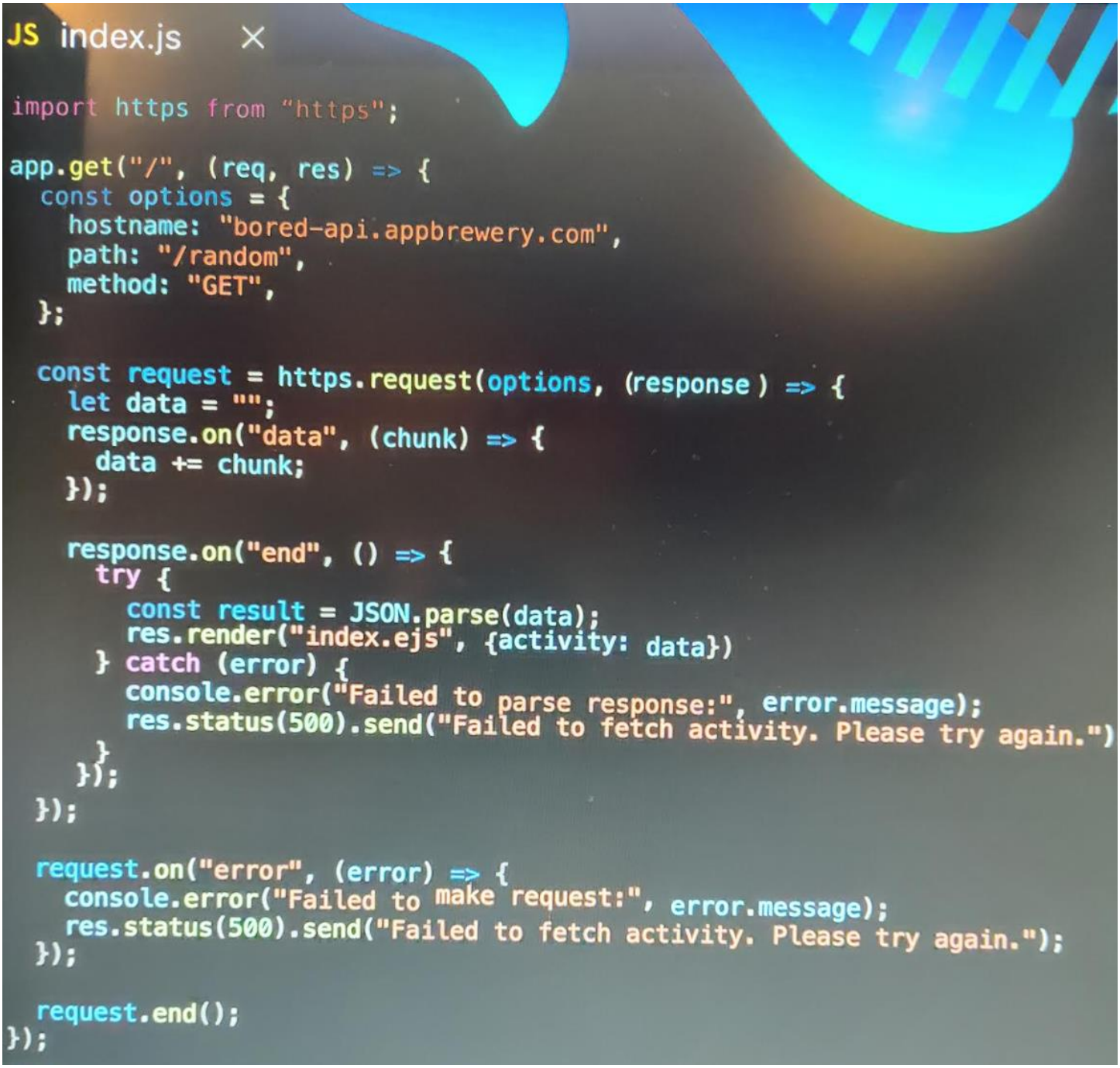
Const jsonData = JSON.stringify(data);

where data is a javaScript object and Stringify simply makes the object into a flat packed string from an JavaScript object.

To deserialize flat packed string into JavaScript object

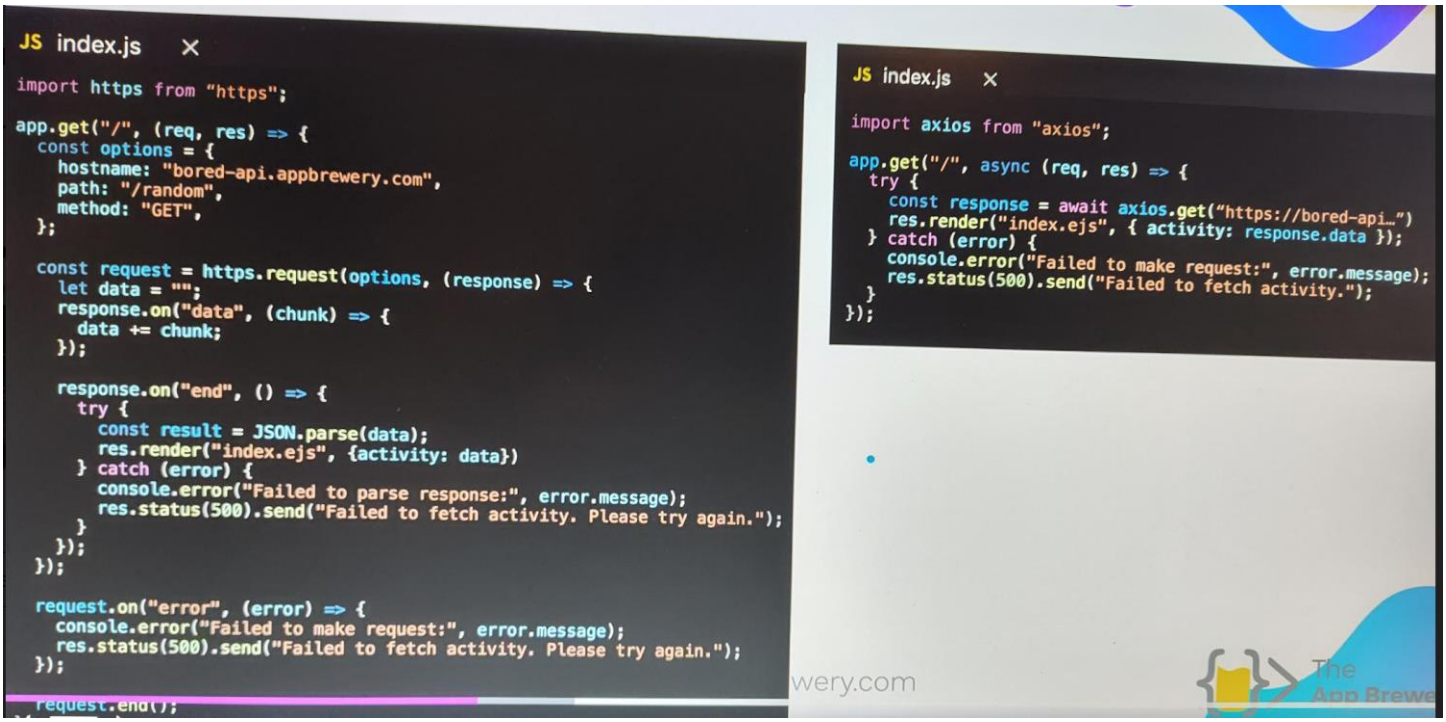
Const data = JSON.parse(jsonStringData)

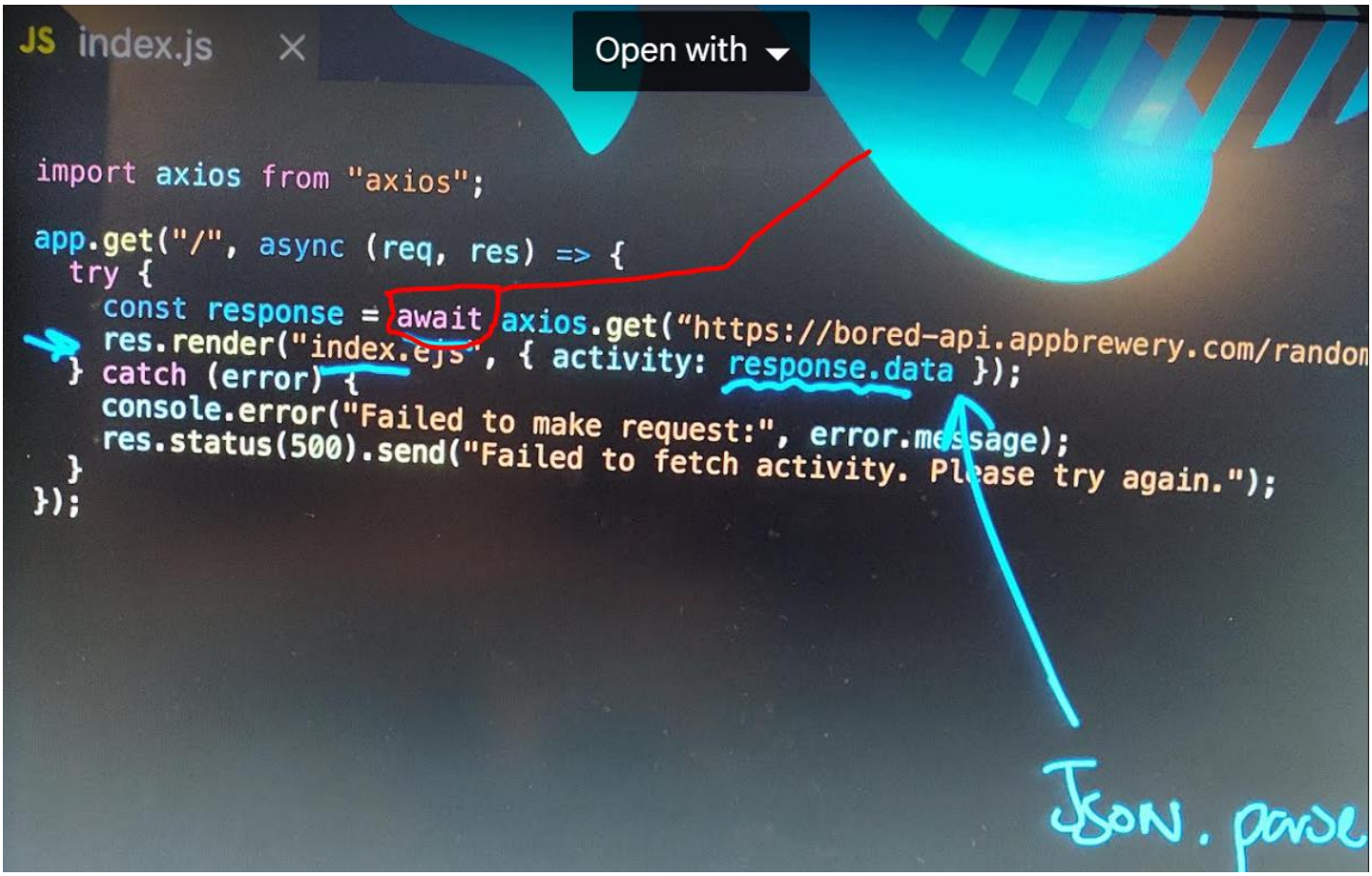
Without Axios



With Axios

Axios provides commonly used methods and requirements to do such things as summoning data from an API.


JS index.js
import https from "https";
app.get("/", (req, res) => {
const options = {
hostname: "bored-api.appbrewery.com",
path: "/random",
method: "GET",
};
const request = https.request(options, (response) => {
let data = "";
response.on("data", (chunk) => {
data += chunk;
});
response.on("end", () => {
try {
const result = JSON.parse(data);
res.render("index.ejs", {activity: data})
} catch (error) {
console.error("Failed to parse response:", error.message);
res.status(500).send("Failed to fetch activity. Please try again.");
}
});
});
request.on("error", (error) => {
console.error("Failed to make request:", error.message);
res.status(500).send("Failed to fetch activity. Please try again.");
});
request.end();
JS index.js
import axios from "axios";
app.get("/", async (req, res) => {
try {
const response = await axios.get("https://bored-api…")
res.render("index.ejs", { activity: response.data });
} catch (error) {
console.error("Failed to make request:", error.message);
res.status(500).send("Failed to fetch activity.");
}
});


JS index.js
Open with
import axios from "axios";
app.get("/", async (req, res) => {
try {
const response = await axios.get("https://bored-api.appbrewery.com/randon
res.render("index.ejs", { activity: response.data });
} catch (error) {
console.error("Failed to make request:", error.message);
res.status(500).send("Failed to fetch activity. Please try again.");
}
});
JSON.parse

API authentication

The three levels of authentication are:

1. Basic Authentication
2. API Key Authorisation
3. Token Based Authentication

Basic Authentication (users authenticate themselves with a password)

You provide a username and password when you make your API request. The prerequisite is that you have an account with the API provider which holds your username and password. Usually this is done by passing a base 64 encoded username and password in the header of the request:

Header Base64 encoding of (username:password) where ascii text characters are passed into client javaScript function to be encoded so your username and password end up in Base64 and added to an authentication header and it gets sent to the server when you make your request.

Although it is easy to decrypt base 64 encoding, using https encodes everything even further resulting in a harder to decode username password exchange however you are still passing your username and password so a sufficiently motivated hacker will decode the key value pair.

API Key Authorization (an authorized client wishing to use your service simply by having the key)

To use api keys correctly sometimes the api will require you to put the api key into the header of the request other times as a query parameter, read api docs to find out which is required.

API key is more secure because no-where are you passing your username and password in the message there is only the key being passed which may have a very short time to live.

Token Based Authentication

User logs in using their username and password which then generates a token for the API, this way the API does not have anything to do with username and password authentication only verifies the token used is genuine.

OAUTH is the industry standard for using token based authentication.

Sequence of operation:

1. Login to the API provider website with your Username and password
2. The API provider issues a token to the user
3. The user uses this token to access the API

The token acts as a proxy for anyone else to interact with the api on the users behalf. In the case of a 3'rd party service such as say a weather app that would like to gain access to the users calander in order to provide weather alerts based on users appointments, this could be done by the user signing into the calendar, acquiring the token and passing the token onto the weather app so as to provide access to calendar by means of this user acquired token.

The Axios Get,Delete method takes two parameters: axios.get(url,config)

The Axios Post,Put,Patch method takes 3 parameters:axios.post(url,body,config)

Incredible API resource: https://github.com/appbrewery/public-api-lists

DIY API

Checkout Rapid API leverages a service api hosting service

What characteristics makes API's monetizable?

1. Data Collection (example weather station)
2. Algorithm/service (example: some sort of service like Chat GPT)
3. Simplified Interface (example: Simplified ordering)
4. Internal API (example: inventory application)

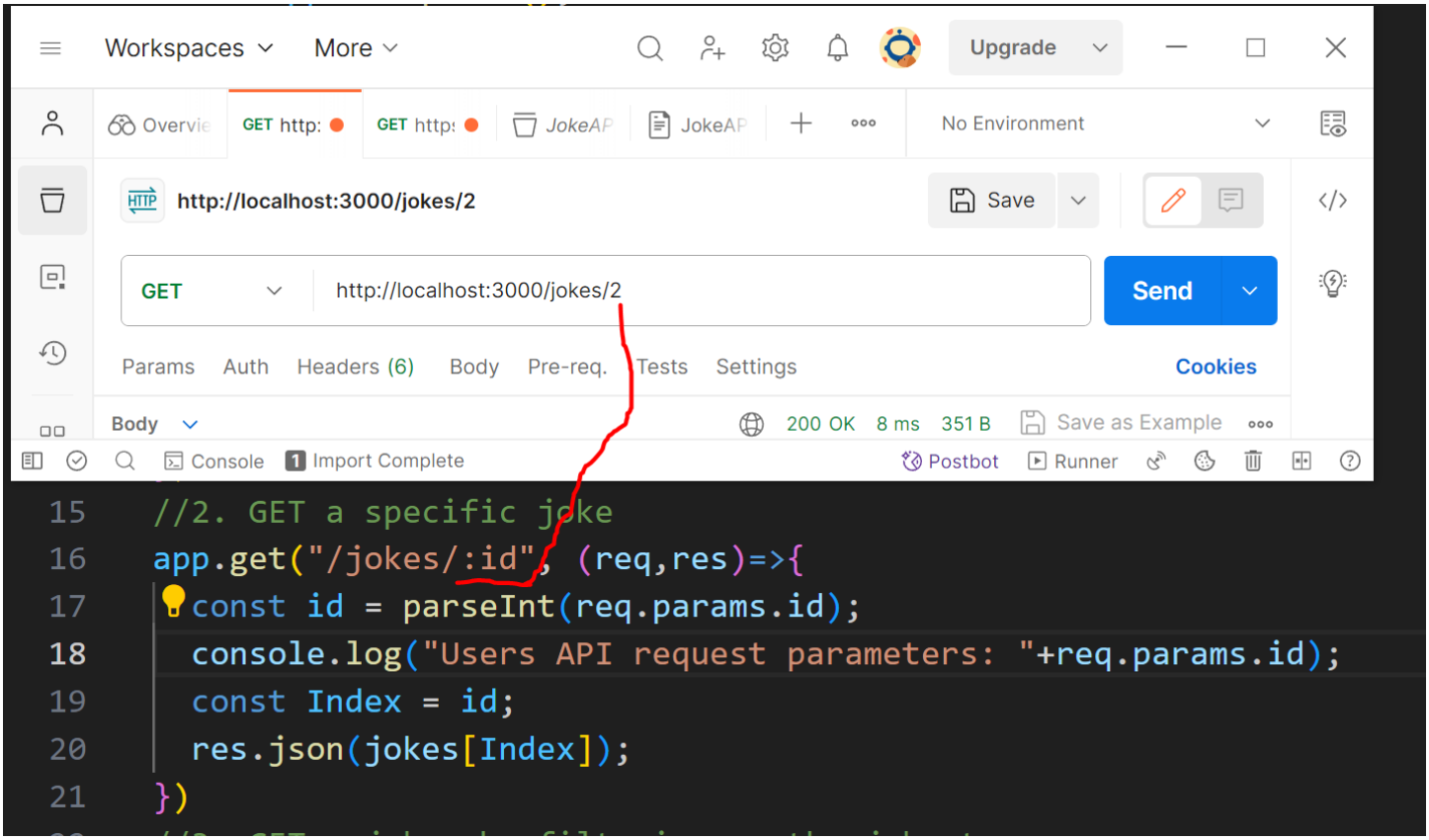
REST:API

Representational State Transfer API

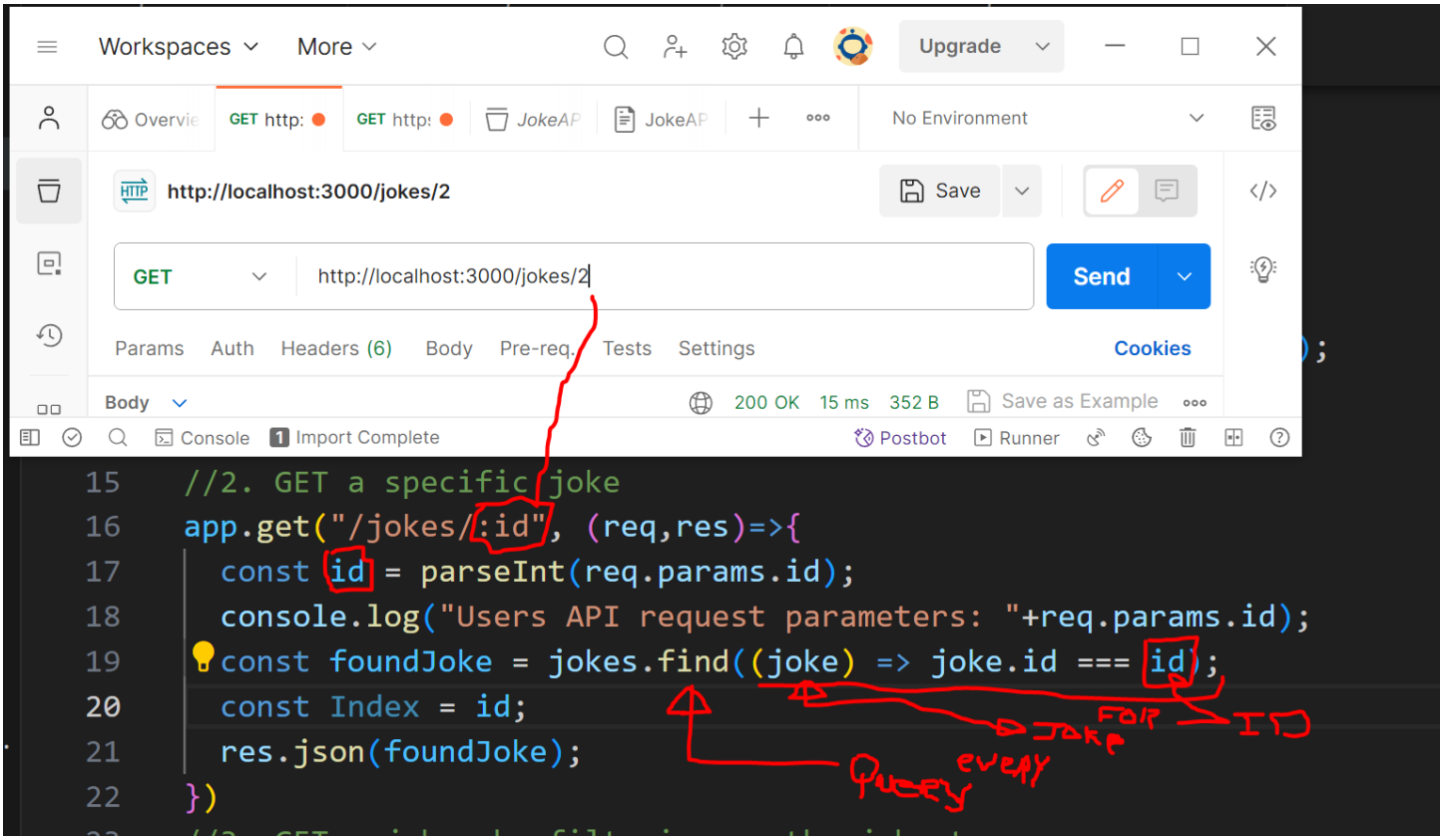
What makes an API restful?

1. It uses standard HTTP methods (GET,POST,PATCH,PUT,DELETE)
2. Uses a standard data format for payload comms.(JSON,XML…) Representation part
3. Clients and servers are not on the same system **this allows each side to scale independent of eachother**.
4. Statelessness: The server should not be storing any client side data, this ensures that each single transaction with the REST API is complete. This also improves scalability as well as simplifying server side implementation.
    a. The client request must contain all necessary information for the server to respond back.
    b. Were this not the case than multiple requests could very quickly overwhelm the server.
5. API must be resource based, that is it is centered around providing access to a resource.
    a. Uses a Universal Resource Identifier (URI) mapped to a Universal Resource Locator (URL)

A get method incorporating a colon allows us to access the path written into the URL directly as parameter:



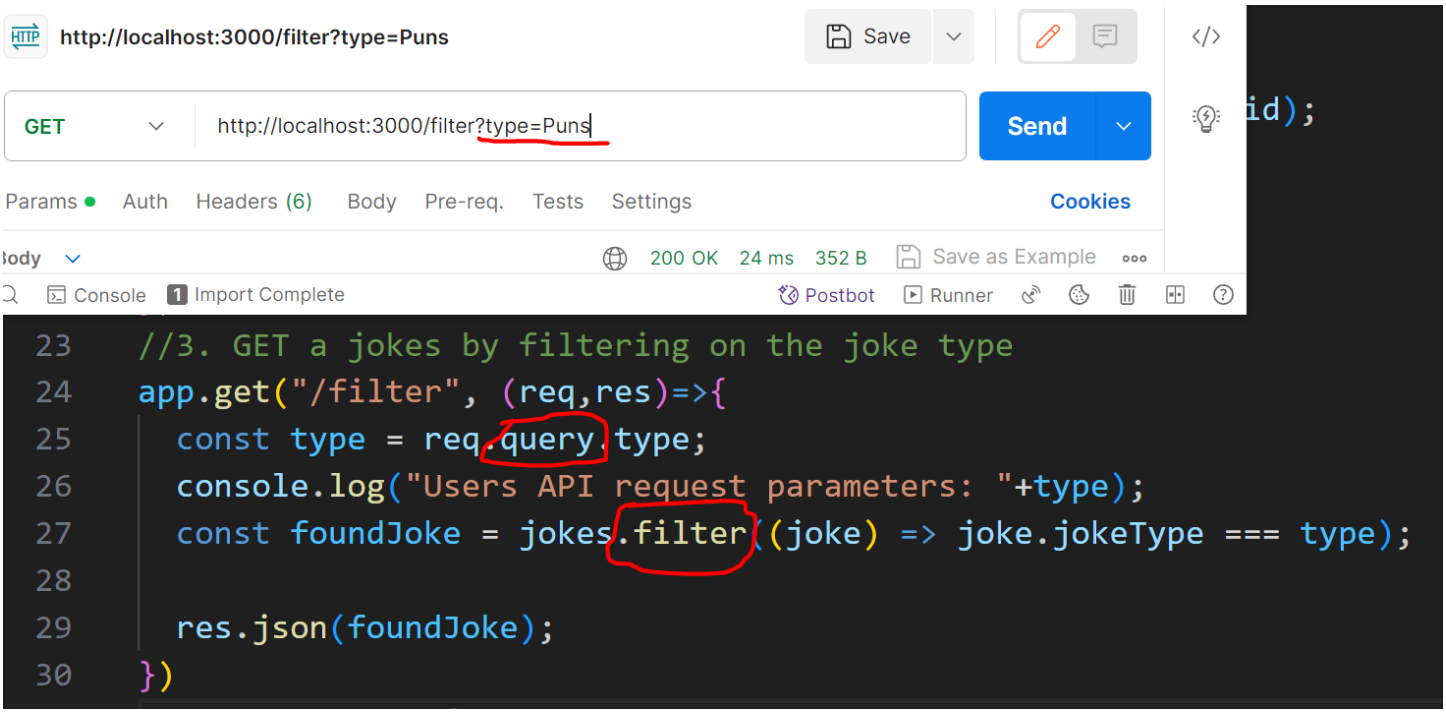
Example of Array.find and req.params



Prototype: array.find(callback) the above callback returns true on a particular condition the find method creates a loop sourcing every element of the jokes array which contains a matching id. The same thing could be accomplished with the array forEach method or just a regular for loop.

Array.find returns the array element value if any of the elements in the array satisfy the condition, otherwise it returns undefined.

Example of array.filter and req.query




```
23  //3. GET a jokes by filtering on the joke type
24  app.get("/filter", (req,res)=>{
25    const type = req.query.type;
26    console.log("Users API request parameters: "+type);
27    const foundJoke = jokes.filter((joke) => joke.jokeType === type);
28
29    res.json(foundJoke);
30  })
```

The return of the array.filter method is an array of all the elements which matched the criteria specified.

SQL

CRUD (Create Read Update Destroy)

```
CREATE TABLE Persons (
    Personid int NOT NULL AUTO_INCREMENT,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (Personid)
);
```

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

SQL foreign key constraint

MySQL FOREIGN KEY Constraint

The `FOREIGN KEY` constraint is used to prevent actions that would destroy links between tables.

A `FOREIGN KEY` is a field (or collection of fields) in one table, that refers to the `PRIMARY KEY` in another table.

The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

Look at the following two tables:

Persons Table

PersonID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

Orders Table

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3
3	22456	2
4	24562	1

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the `PRIMARY KEY` in the "Persons" table.

The "PersonID" column in the "Orders" table is a `FOREIGN KEY` in the "Orders" table.

The `FOREIGN KEY` constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

To allow naming of a `FOREIGN KEY` constraint, and for defining a `FOREIGN KEY` constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
    REFERENCES Persons(PersonID)
);
```

FOREIGN KEY on ALTER TABLE

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

```
ALTER TABLE Orders
ADD CONSTRAINT FK_PersonOrder
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

The INNER JOIN keyword selects records that have matching values in both tables.

INNER JOIN Syntax

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

The following SQL statement selects all orders with customer information:

Example Get your own SQL Server

```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

Note: The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not be shown!

JOIN Three Tables

The following SQL statement selects all orders with customer and shipper information:

Example

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);
```

MongoDB

Note: The first thing to do when getting started with a new database is to lookup the CRUD operations.

Collections in Mongo are analogs of Tables from the SQL world

A mongoshell command db.products is a reference to a collection called products therefore:

Db.products.insertOne({_id:1, name: “Pen”, price: 1.20})

Shell commands to create a mongodb:

1. nano .bash_profile
    a. alias mongod="/C/Program\ Files/MongoDB/Server/7.0/bin/mongod.exe"
    b. alias mongos="/C/Program\ Files/MongoDB/Server/7.0/bin/mongos.exe"
    c. alias mongosh="/E/mongosh-2.0.2-win32-x64/bin/mongosh.exe"
2. mongosh
3. use shopDB
4. db.products.insertOne({_id:1, name: "Pen", price: 1.20})

How to create relationships in MongoDB

A lot depends on how your data is structured:

1. In a simple one to many relationship such as when one product has many reviews- You could simply embed an array of references within the document ala reviews: [{},{}….]
2. In order to form relationships with other abstractions such as products ordered abstraction you could simply have an array of orders submitted which refereces product deffinitions by their id numbers to create a heterogeneous mix of ordered materials listed by individual customer order.

How to connect to mongoDB programmatically

Steps to create a new nodejs project

1. Create folder in development directory
2. Add main js file with: “touch main.js”
3. “npm init -y”
4. Decide whether you want to use the nodejs native driver or mongoose

MongoDB native driver

5. Install mongoDB driver: “npm install mongodb@6.1”
6. Copy and paste startup code from: https://www.mongodb.com/docs/drivers/node/current/quick-start/connect-to-mongodb/
7. Convert top line in code to: import { MongoClient } from "mongodb";
8. Insert Connection String from mongoDB Compass : "mongodb://**localhost**:27017"
    a. Localhost does not work!! Use: mongodb://127.0.0.1:27017/
        i. instead

The preferred way is to use mongoose, an ODM (Object-Document-Mapper) to connect to mongoDB programmatically

5. Install mongoose: npm i mongoose

```
import { mongoose } from "mongoose";
// Replace the uri string with your connection string.
const uri = "mongodb://127.0.0.1:27017/fruitsDB";
mongoose.connect(uri);
const peopleSchema = mongoose.Schema({
    name: String,
    age: Number
});
const Person = mongoose.model("Person",peopleSchema);

const person = new Person({
    name: "Jhon",
    age: 37
});
person.save();
```

21. Create the db by placing it in the url ….:27017:/"name of DB" and mongoose.connect into it
22. Create the collection by enstantiating the schema into a model using the singular form in the name which becomes pluralized in the collection name
23. We can see this by executing the shell command db.people.find()
24. Lookup all the commands you can access through mongoose at: https://mongoosejs.com/docs/api.html#model
25. See also Model.insertMany([])
    a. You specify the name of the mongoose model which will allow it to connect to the relevant collection and also know what the schema is it should work with.
    b. The argument consists of the array of JSONS to be inserted.

Reading a MONGO DB using mongoose

1. Tap into the model and call the Find() function on it
2. The argument to the find function is a callback function
3. The function returns a fully functional javaScript object.
4. Close the database when the query is completed
    a. Mongoose.connection.close();

Validating using mongoose

1. Instead of building a whole bunch of validation statements we can use the built-in mongoose functions which are a lot easier and a lot quicker to use. See: https://mongoosejs.com/docs/validation.html
    a. Validation is defined in the SchemaType
    b. Validation is middleware. Mongoose registers validation as a pre('save') hook on every schema by default.
    c. Validation always runs as the first pre('save') hook. This means that validation doesn't run on any changes you make in pre('save') hooks.
    d. You can disable automatic validation before save by setting the validateBeforeSave option
    e. You can manually run validation using doc.validate() or doc.validateSync()
    f. You can manually mark a field as invalid (causing validation to fail) by using doc.invalidate(...)
    g. Validators are not run on undefined values. The only exception is the required validator.
    h. When you call Model#save, Mongoose also runs subdocument validation. If an error occurs, your Model#save promise rejects
    i. Validation is customizable
2. Mongoose has several built-in validators.
    a. All SchemaTypes have the built-in required validator. The required validator uses the SchemaType's checkRequired() function to determine if the value satisfies the required validator.
    b. Numbers have min and max validators.
    c. Strings have enum, match, minLength, and maxLength validators.

Creating Relationships using Mongoose

We can use elements from other collections within the definition of the object to be inserted by:

1. When declaring the schema for an object embed the element you wish to source from another collection within the the schema definition by giving a name as its data-type where its data-type is the schema it is a member of.
2. When instantiating the model put the embedded object from the other collection inside the object you are enstantiating then save. (does this get update when the embedded object changes? – yes it is linked!

```
3.  const uri = "mongodb://127.0.0.1:27017/fruitsDB";
4.  mongoose.connect(uri);
5.  const fruitSchema = new mongoose.Schema({
6.      name: String,
7.      rating: Number,
8.      review: String
9.  });
10. const newpersonSchema = new mongoose.Schema({
11.     name: String,
12.     age: Number,
13.     favouriteFruit: fruitSchema
14. });
15. const oldpersonSchema = new mongoose.Schema({
16.     name: String,
17.     age: Number
18. });
19. const Fruit = mongoose.model("Fruit",fruitSchema);
20. const mango = new Fruit({
21.     name: "mango",
22.     rating: 8,
23.     review: "Sweet and nice"
24. });
```

```
25. mango.save();
26.
27. const Person = mongoose.model("Person",newpersonSchema);
```

Reading params from URL and Creating Mongo relationships using wrapper around an array of objects

```
const listSchema = new mongoose.Schema({
  name: String,
  items: [itemsSchema]
  });
const List = mongoose.model("List",listSchema);

const item1 = new Item({
  task: "Welcome to your todo list!"
  // not requireddate: new Date()
});
app.get("/:customListName", async(req, res) => {
  console.log("Route: "+req.params.customListName);
  const tempList = await List.findOne({name : req.params.customListName});
  if(tempList){
  console.log("we found a "+req.params.customListName+" list: ");
  }else{
    console.log("we created a "+req.params.customListName+" list: ");
  const list = new List({
    name: req.params.customListName,
    items: defaultItems
  })
  list.save();
}
```

.ejs Lessons learned

1. The browser will keep asking for a favicon.ico in the URL parameter until you deliver one by creating a link reference in the HTML header (see tasks)
2. To deliver back-end functionality to a checkbox or textbox it must be part of a form
3. You can use mongos unique identifier to target the specific element the checkbox is related to.

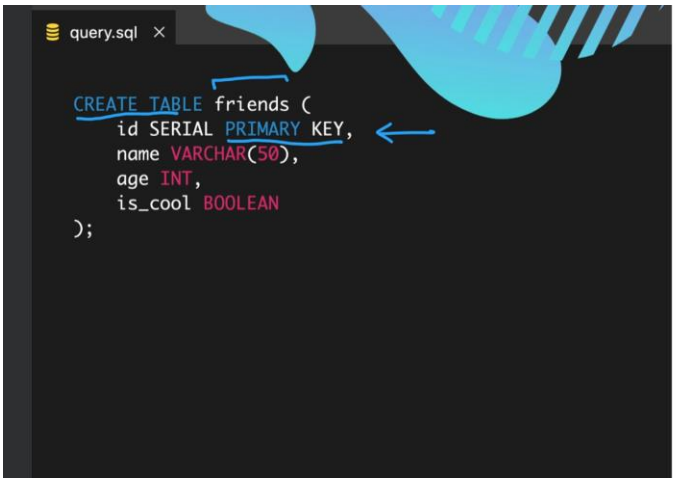
```
4.        <%  if(locals.Tasks){%>
5.          <% locals.Tasks.forEach(task =>{ %>
6.          <tr>
7.              <form action="/delete" method="post">
8.                <td><input type="checkbox" name="checkbox" value="<%=task._id%>"
   onChange="this.form.submit()"></td>
9.                <td><%= task.task %></td>
10.               <td><%= formatDateTimeToString(task.date) %></td>
11.             </form>
12.         </tr>
13.         <%});}%>
```

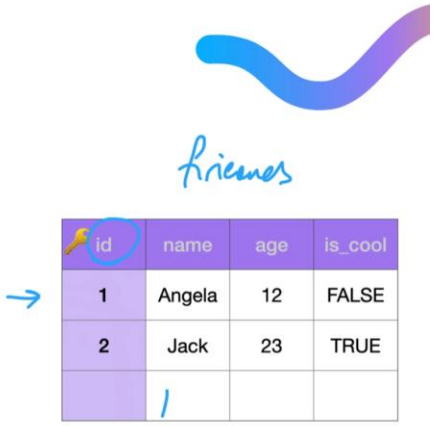
PostgreSQL Open Source Relational Database Management System (RDBMS)

Why Choose PostgreSQL

1. The World's most advanced open source relational database
2. Widely used in industry
3. Free
4. Career Opportunities
5. Community Support

Table Creation





Id SERIAL means the element will autoincrement every time a new friend object is added

Postgres Read Query

```
import pg from "pg";

const db = new pg.Client({
  user: "postgres",
  host: "localhost",
  database: "world",
  password: "123456",
  port: 5432,
});

db.connect();

db.query("SELECT * FROM capitals", (err, res) => {
  if (err) {
    console.error("Error executing query", err.stack);
  } else {
    quiz = res.rows;
  }

  db.end();
});
```

Steps to setup postgresSQL in pgAdmin

1. Create Database
2. Create Table:
    a. CREATE TABLE world_food(
    b. id SERIAL PRIMARY KEY,
    c. Country VARCHAR(45),
    d. Rice float,
    e. Wheat float
    f.);
3. Import Data

Steps to setup postgresql in your development environment

2. "npm i pg"
3. import pg from "pg";
4. Construct new pg.Client() object:
    a. const db = **new pg.Client**({
    b. **user**: "postgres",
    c. **host**: "localhost",
    d. **database**: "world",
    e. **password**: "pilsucki",
    f. **port**: 5432,
    g. });
5. db.connect();
6. Perform Query:
    a. db.query("SELECT * FROM capitals", (err, res) => {
    b. if (err) {
    c. console.error("Error executing query", err.stack);
    d. } else {
    e. //This is where quiz is populated with an array of all the capitals
    f. quiz = **res.rows**;
    g. }
    h. db.end();
    i. });

SELECT queries

1. SELECT <COLUMN> FROM <TABLE>;
2. Do not forget to use single quotes on the condition
    a. SELECT * FROM public.world_food WHERE country = 'China'

Note:

Use double quotes or no quotes for column names and single for string values

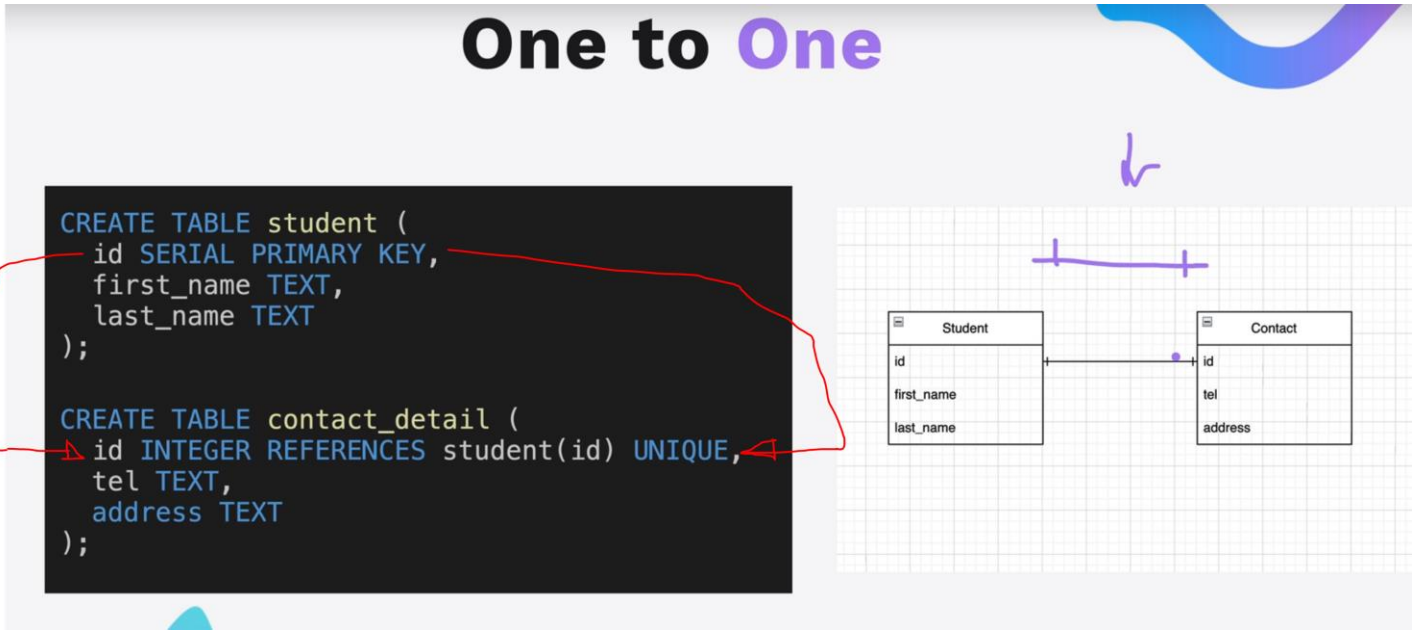
INSERT query

Using the 'pg' library use:

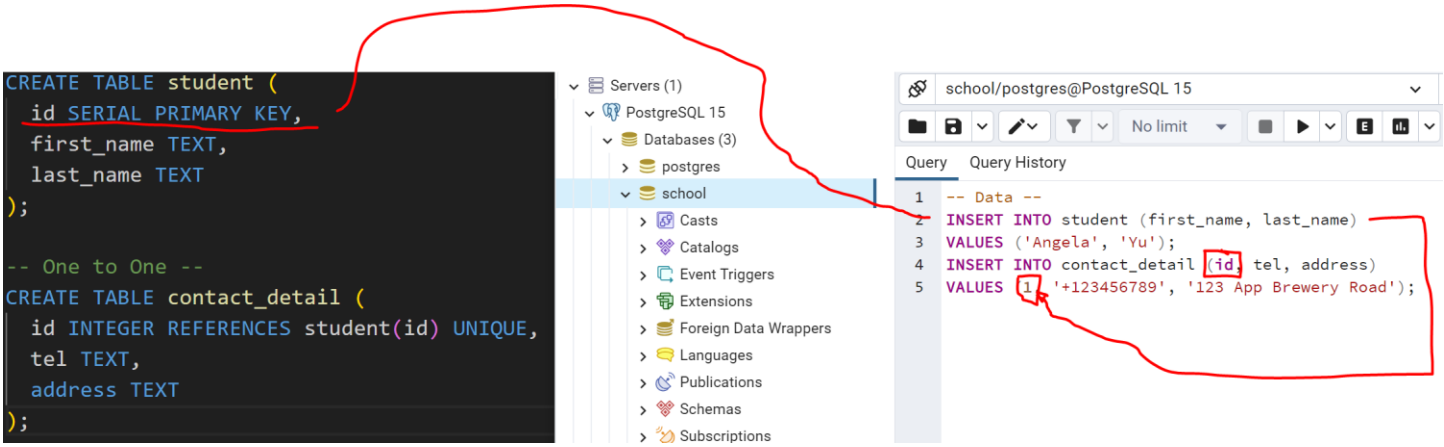
Db.query("INSERT INTO <NAME OF TABLE> (<COL1>,<COL2>,<COL3>) VALUES (**$**1,**$**2,**$**3)",[V1,V2,V3]);

SQL One-to-One

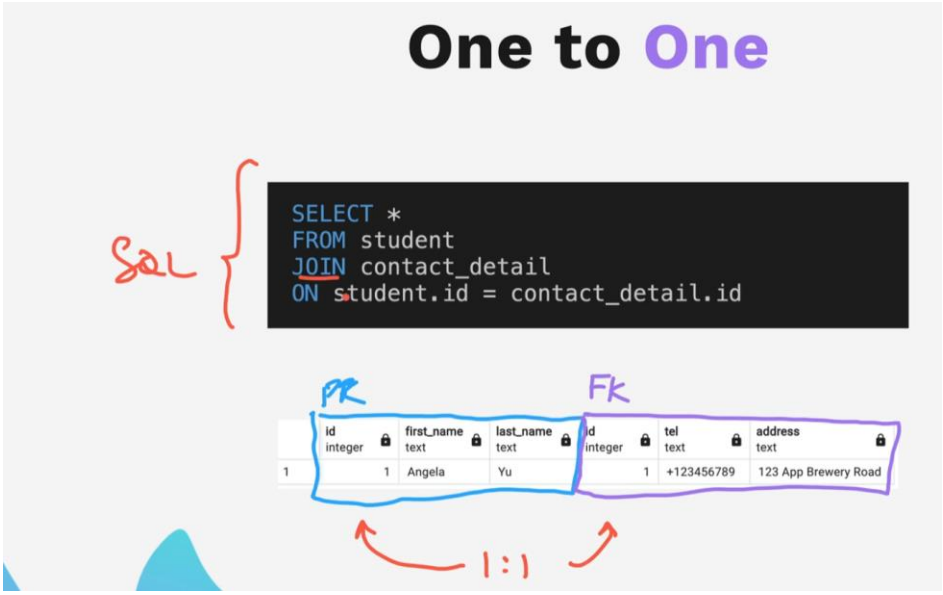
How to Use foreign keys and inner joins in one-to-one relationships



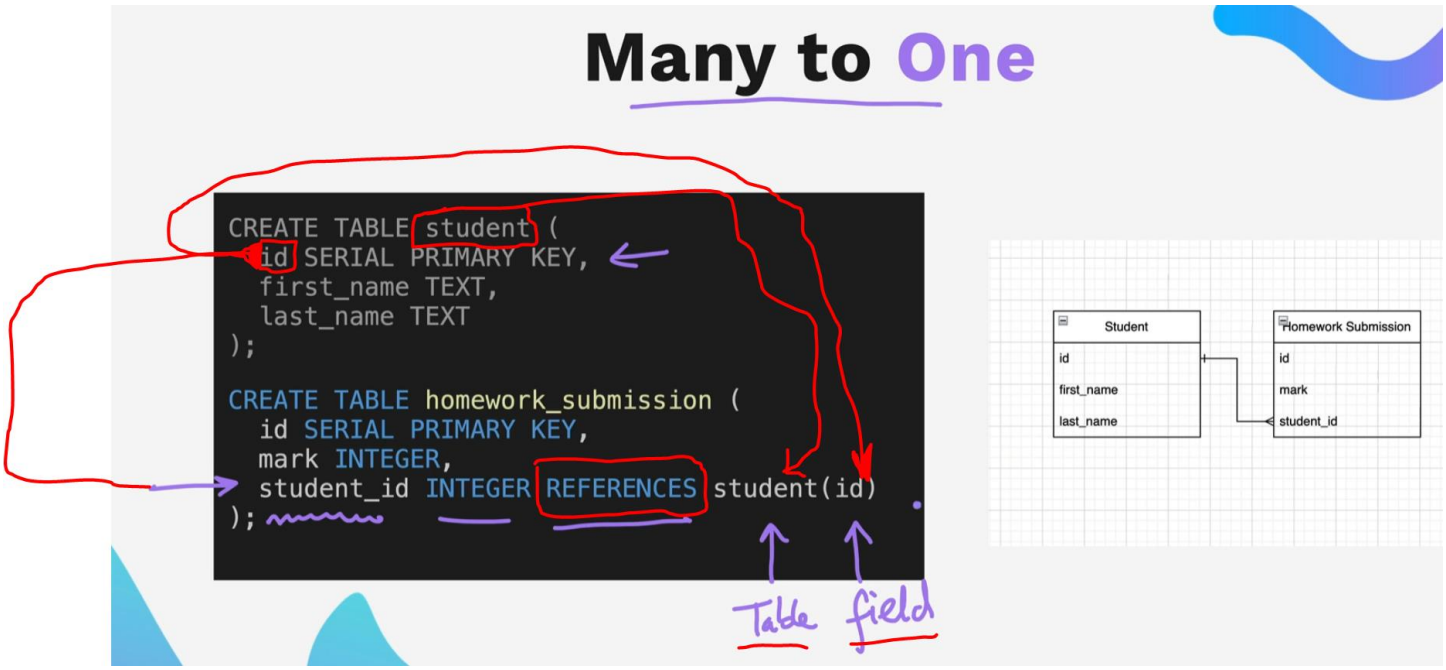
The above sets up the ability to reference a foreign key from the primary key.



Join



SQL One to many relationship



A handy way of performing multiple insertions akin to trendlog:

```
-- Data --
INSERT INTO homework_submission (mark, student_id)
VALUES (98, 1), (87, 1), (88, 1)
```

```
1  SELECT * FROM student
2  join homework_submission
3  on student.id=student_id
4
```

Bool Condition

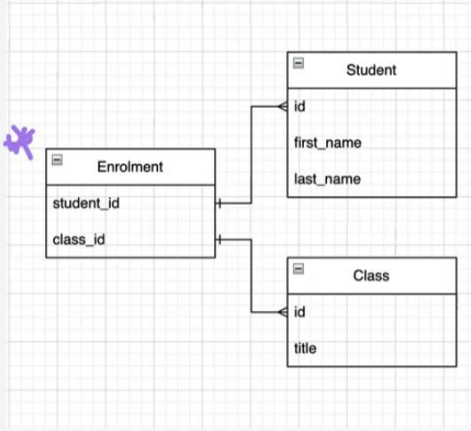
Data Output | Messages | Notifications

	id integer	first_name text	last_name text	id integer	mark integer	student_id integer
1	1	Angela	Yu	1	98	1
2	1	Angela	Yu	2	87	1
3	1	Angela	Yu	3	88	1

SQL Many to Many relationship

Many to Many

```
CREATE TABLE student (
  id SERIAL PRIMARY KEY,
  first_name TEXT,
  last_name TEXT
);

CREATE TABLE class (
  id SERIAL PRIMARY KEY,
  title VARCHAR(45)
);

CREATE TABLE enrollment (
  student_id INTEGER REFERENCES student(id),
  class_id INTEGER REFERENCES class(id),
  PRIMARY KEY (student_id, class_id)
);
```

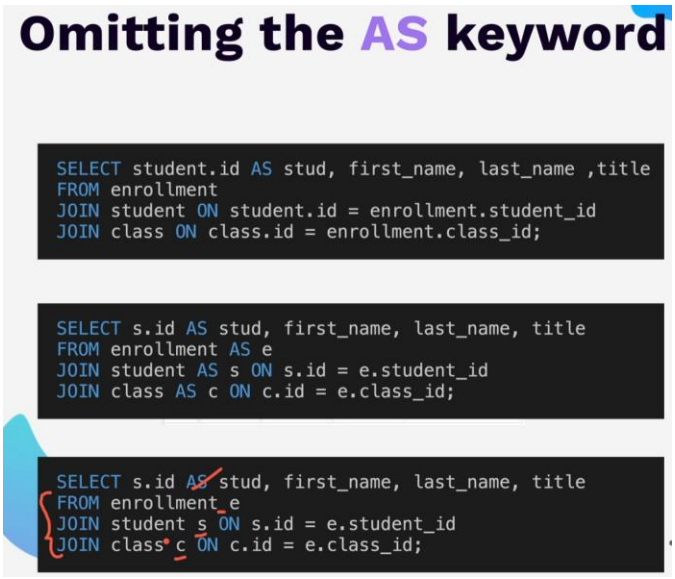



```
SELECT *
FROM enrollment
JOIN student ON student.id = enrollment.student_id
JOIN class ON class.id = enrollment.class_id;
```

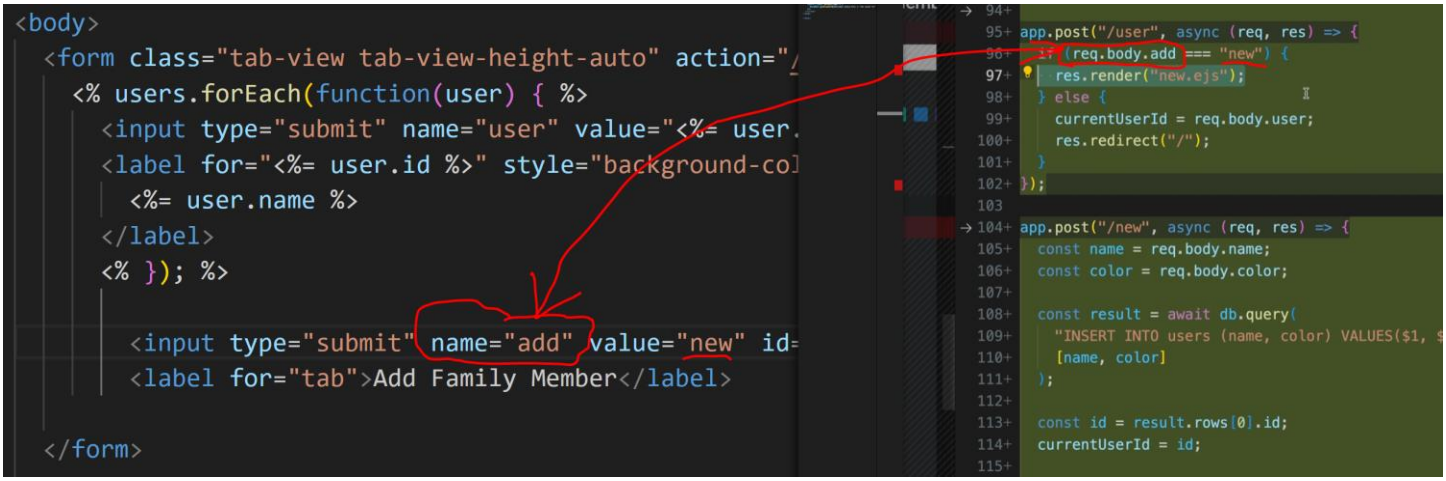
	student_id integer	class_id integer	id integer	first_name text	last_name text	id integer	title character varying (45)
1	1	1	1	Angela	Yu	1	English Literature
2	1	2	1	Angela	Yu	2	Maths
3	2	2	2	Jack	Bauer	2	Maths
4	2	3	2	Jack	Bauer	3	Physics

Good reasons for usage of Aliases

1. If the referenced field in each table I wish to join contains a renamed copy an easy way to aggregate all these back to just one is to use the 'AS' command to create an alias.
2. To shorten length of command in places where the table names are excessively long
3. A space followed by a string name gets around using the 'AS' command because to SQL this is implicit

a. 

Project Errata



I did not find/use the body.add object which would have had the value of add… looked for undefined instead:

```
app.post("/user", async (req, res) => {
  if(req.body.user === undefined){
    res.render("new.ejs");
    return;
  }
  console.log("User Name Tab selected: "+req.body.user);
  //const countries = await checkVisisted(req.body.user);
  let users = await getUsers();
  user = await getUser(req.body.user);
  res.redirect("/");
```

Returning Data from modified rows

The INSERT, UPDATE, and DELETE commands all have an optional RETURNING clause that supports this. Use of RETURNING avoids performing an extra database query to collect the data, and is especially valuable when it would otherwise be difficult to identify the modified rows reliably.

The allowed contents of a RETURNING clause are the same as a SELECT command's output list (see **Section 7.3**). It can contain column names of the command's target table, or value expressions using those columns. A common shorthand is RETURNING *, which selects all columns of the target table in order.

In an INSERT, the data available to RETURNING is the row as it was inserted. This is not so useful in trivial inserts, since it would just repeat the data provided by the client. But it can be very handy when relying on computed default values. For example, when using a serial column to provide unique identifiers, RETURNING can return the ID assigned to a new row:

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);


INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;
```

The RETURNING clause is also very useful with INSERT ... SELECT.

In an UPDATE, the data available to RETURNING is the new content of the modified row. For example:

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

In a DELETE, the data available to RETURNING is the content of the deleted row. For example:

```
DELETE FROM products
  WHERE obsoletion_date = 'today'
  RETURNING *;
```

PostgreSQL Alter, Drop, Update, Delete

Alter is used to change the table schema:

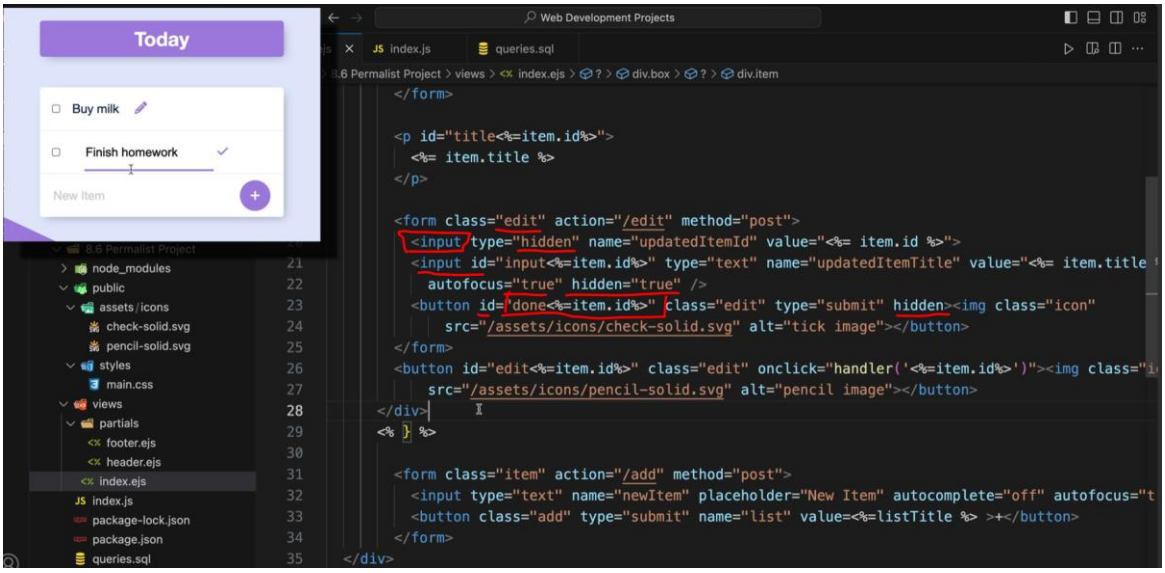
```
ALTER TABLE <TABLE TO ALTER>
      <DO SOMETHING>


ALTER TABLE student
      RENAME TO user;


ALTER TABLE user
      ALTER COLUMN first_name TYPE VARCHAR(20);
```

```
ALTER TABLE contact_detail
      ADD email TEXT
```

EJS template with <script>

1. When .ejs does templating it inserts a template string therefore in cases where an array is needed a comma delimited phrase can be passed up to the template inside the <script> portion than parsed back into an array.
2. If my res.render sends an array of strings this becomes a single comma delimited String phrase.
3. The ternary operator (an operator accepting three operands):
4. Like all expressions, the conditional operator can also be used as a standalone statement with side-effects, though this is unusual outside of minification:
    a. userIsYoungerThan21 **?** serveGrapeJuice() **:** serveWine();
5. They can even be chained:
    a. serveDrink(userIsYoungerThan4 **?** 'Milk' **:** userIsYoungerThan21 **?** 'Grape Juice' **:** 'Wine');

How to easily enable renaming through the hidden attribute.

1. Button could be used to trigger a handler function which
2.
```
        <button id="edit<%=item.id%>" class="edit" onclick="handler('<%=item.id%>')"><img class="i
            src="/assets/icons/pencil-solid.svg" alt="pencil image"></button>
      </div>
      <% } %>

      <form class="item" action="/add" method="post">
        <input type="text" name="newItem" placeholder="New Item" autocomplete="off" autofocus="t
        <button class="add" type="submit" name="list" value=<%=listTitle %> >+</button>
      </form>
</div>

<script>
  function handler(id) {
    document.getElementById("title" + id).setAttribute("hidden", true)
    document.getElementById("edit" + id).setAttribute("hidden", true)
    document.getElementById("done" + id).removeAttribute("hidden")
    document.getElementById("input" + id).removeAttribute("hidden")
  }
```
3. When the edit button is clicked it calls the handler passing the id of the row as its argument then performs each of the four attribute changes to the element which in this case are:
    a. Make the title invisible
    b. Make the edit symbol invisible
    c. Then remove the ‘hidden’ attribute from the done icon and make the line into an input by removing the ‘hidden’ attribute from the paragraph attribute
    d. .
4. 

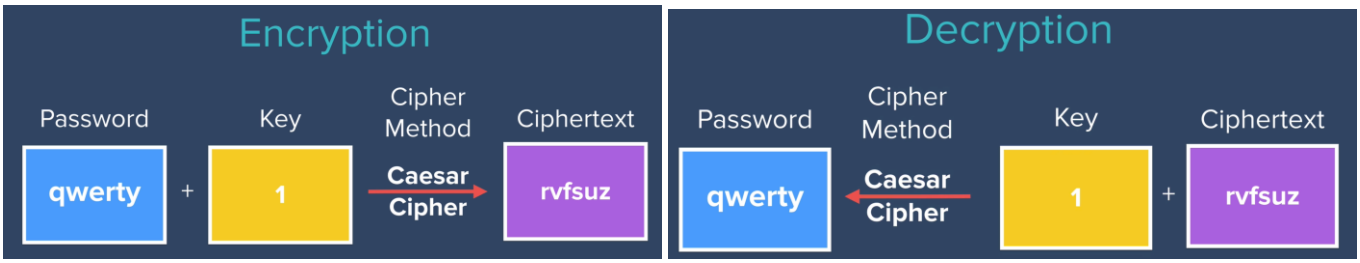
5. In the “SELECT” query order everything by id ascending or descending to maintain the order after editing.

Authentication and security with mongo

Level 1 mongoose-encryption

Make usernames and passwords encrypted in database.

Introduction: The oldest cipher, the Caesar cipher shifted every character in alphabetical order by a specific offset.



1. https://www.npmjs.com/package/mongoose-encryption
2. Mongoose will encrypt when you call save and decrypt when you call find if you add the following:
   a.

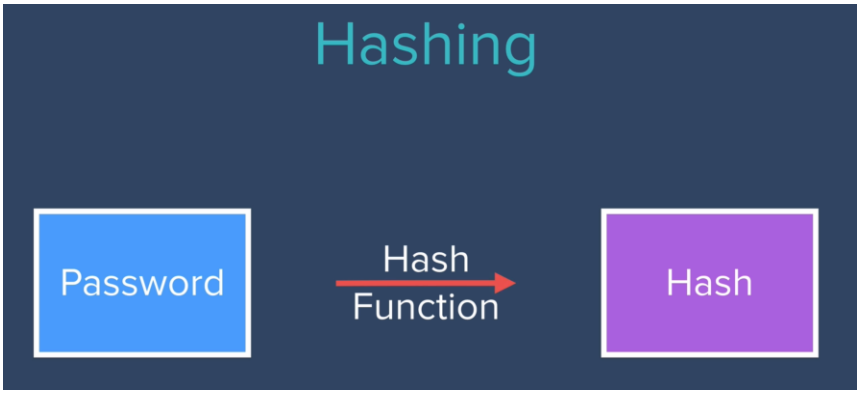
```
const secret = "Iamconffidentinmyabylitytoimplementanyencryptionanddecryptionstrategy";
const userSchema = new mongoose.Schema({
    email: String,
    password: String
});
userSchema.plugin(encrypt,{secret : secret,encryptedFields : ["password"] });
//add additional fields into array as needed
const User = new mongoose.model("user",userSchema);

const app = express();
app.use(express.static("public"));
app.set('view engine','ejs');
app.use(bodyParser.urlencoded({
    extended: true
}));
```

3. Use dotenv to keep the “secret” cypher harder for hackers to find.
4. https://www.npmjs.com/package/dotenv
5.

```
import dotenv from "dotenv";
dotenv.config();
const uri = "mongodb://127.0.0.1:27017/userDB";
//mongoose.connect("mongodb://localhost:27017/userDB");
//IMPORTANT: mongoose.connect does not work with "localhost"
mongoose.connect(uri)
.then(()=>console.log("DB Connected"))
.catch((err)=>console.log(err));

const userSchema = new mongoose.Schema({
    email: String,
    password: String
});
userSchema.plugin(encrypt,{secret : process.env.SECRET,encryptedFields : ["password"] });
//add additional fields into array as needed
const User = new mongoose.model("user",userSchema);

const app = express();
app.use(express.static("public"));
app.set('view engine','ejs');
```

```
SECRET=Iamconffidentinmyabylitytoimplem
```

Level 2 from encryption to Hashing



Its almost impossible to reverse the hash function but the hash function is consistent so you can compare one hash to another for authenticating a user.

1. Npm I md5 then import md5 from “md5”;
2.

```
app.post("/register", function(req,res){

    const newUser = new User({
        email: req.body.username,
        password: md5(req.body.password)
    });
    console.log("Trying to save new user: "+JSON.stringify(newUser));
    res.render("secrets");
    try{
        newUser.save();
    }catch(err){
        if(err){
            console.log("we got Error: " + err);
        }
    }
});
```

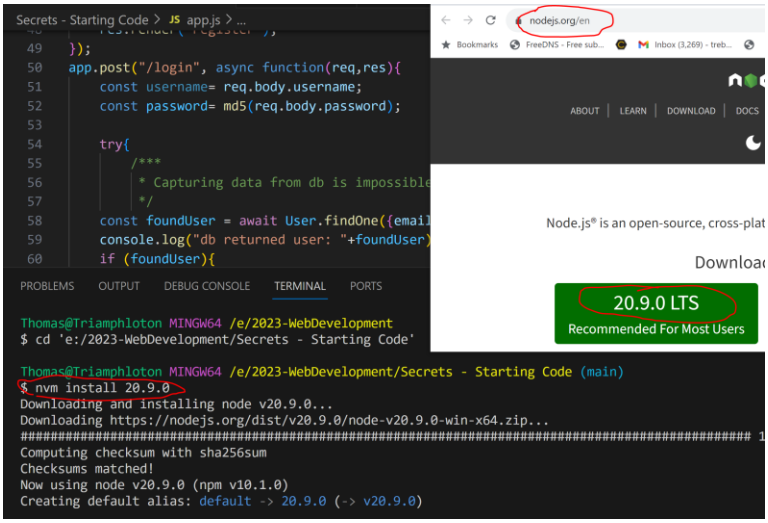
```
app.post("/login", async function(req,res){
    const username= req.body.username;
    const password= md5(req.body.password);

    try{
        /***
         * Capturing data from db is impossible without async and await
         */
    const foundUser = await User.findOne({email: req.body.username});
    console.log("db returned user: "+foundUser);
    if (foundUser){
        console.log("user found!! password: "+foundUser.password+" expected: "+password);
        if(foundUser.password==password){
            console.log("password matched!!");
        res.render("secrets");
        }else{
            console.log("password NOT matched!!");
            res.render("login");
        }
    }else{
        res.render("login");
```

Salting

Hashing is still vulnerable to hackers that lookup common words converted to hash so to make the hashing stronger a random number is added to the string so the resulting hash is created out of the original string plus the random number.

Setting up node.bcrypt

Bcrypt is very sensitive to the node version.(node --version)

1. Install nvm (node version manager)
    a. https://github.com/nvm-sh/nvm
2. Install latest node:
    a. 

```
Secrets - Starting Code > JS app.js > ...
49  });
50  app.post("/login", async function(req,res){
51      const username= req.body.username;
52      const password= md5(req.body.password);
53
54      try{
55          /***
56           * Capturing data from db is impossible
57           */
58      const foundUser = await User.findOne({email
59      console.log("db returned user: "+foundUser
60      if (foundUser){

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Thomas@Triamphloton MINGW64 /e/2023-WebDevelopment
$ cd 'e:/2023-WebDevelopment/Secrets - Starting Code'

Thomas@Triamphloton MINGW64 /e/2023-WebDevelopment/Secrets - Starting Code (main)
$ nvm install 20.9.0
Downloading and installing node v20.9.0...
Downloading https://nodejs.org/dist/v20.9.0/node-v20.9.0-win-x64.zip...
######################################################################################################## 1
Computing checksum with sha256sum
Checksums matched!
Now using node v20.9.0 (npm v10.1.0)
Creating default alias: default -> 20.9.0 (-> v20.9.0)
```

```
nodejs.org/en
Bookmarks  FreeDNS - Free sub...  Inbox (3,269) - treb...
ABOUT | LEARN | DOWNLOAD | DOCS
Node.js® is an open-source, cross-plat
Downloa
20.9.0 LTS
Recommended For Most Users
```

3. Npm I bcrypt https://www.npmjs.com/package/bcrypt
4. Swap out md5 for decrypt
5. Add saltRounds

Usage

async (recommended)

```
const bcrypt = require('bcrypt');
const saltRounds = 10;
const myPlaintextPassword = 's0/\/\P4$$w0rD';
const someOtherPlaintextPassword = 'not_bacon';
```

To hash a password:

Technique 1 (generate a salt and hash on separate function calls):

```
bcrypt.genSalt(saltRounds, function(err, salt) {
    bcrypt.hash(myPlaintextPassword, salt, function(err, hash) {
        // Store hash in your password DB.
    });
});
```

Technique 2 (auto-gen a salt and hash):

```
bcrypt.hash(myPlaintextPassword, saltRounds, function(err, hash) {
    // Store hash in your password DB.
});
```

6.

To check a password:

```
// Load hash from your password DB.
bcrypt.compare(myPlaintextPassword, hash, function(err, result) {
    // result == true
});
```

7.

Cookies and session maintenance

Sessions remain open to prevent re-authentication until the point you log out

Passport (implementing cookies and sessions)

Passport is middleware for Node.js that “salts” and “hashes” our user authentication for us making it a lot easier to plug in different methods of authentication

https://www.passportjs.org/docs/

Npm install:

1. Npm i passport passport-local passport-local-mongoose express-session

```
import session from "express-session";
import passport from "passport";
import passportLocalMongoose from "passport-local-mongoose";
```

2. Please note that `secure: true` is a **recommended** option. However, it requires an https-enabled website, i.e., HTTPS is necessary for secure cookies. If `secure` is set, and you access your site over HTTP, the cookie will not be set. If you have your node.js behind a proxy and are using `secure: true`, you need to set "trust proxy" in express:

```
var app = express()
app.set('trust proxy', 1) // trust first proxy
app.use(session({
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true,
  cookie: { secure: true }
}))
```

For using secure cookies in production, but allowing for testing in development, the following is an example of enabling this setup based on `NODE_ENV` in express:

```
var app = express()
var sess = {
  secret: 'keyboard cat',
  cookie: {}
}

if (app.get('env') === 'production') {
  app.set('trust proxy', 1) // trust first proxy
  sess.cookie.secure = true // serve secure cookies
}
```

3. Configure passport-local-mongoose https://www.npmjs.com/package/passport-local-mongoose

Plugin Passport-Local Mongoose

First you need to plugin Passport-Local Mongoose into your User schema

4.

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const passportLocalMongoose = require('passport-local-mongoose');

const User = new Schema({});

User.plugin(passportLocalMongoose);

module.exports = mongoose.model('User', User);
```

5.
```
mongoose.connect("mongodb://localhost:27017/userDB", {useNewUrlParser: true});

const userSchema = new mongoose.Schema ({
  email: String,
  password: String
});

userSchema.plugin(passportLocalMongoose);
```

6.
```
mongoose.set("useCreateIndex",true);
```

7.
Simplified Passport/Passport-Local Configuration

Starting from version 0.2.1, passport-local-mongoose adds a helper method `createStrategy` as static method to your schema. The `createStrategy` is responsible to setup passport-local `LocalStrategy` with the correct options.

```
const User = require('./models/user');

// CHANGE: USE "createStrategy" INSTEAD OF "authenticate"
passport.use(User.createStrategy());

passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());
```

The reason for this functionality is that when using the `usernameField` option to specify an alternative usernameField name, for example "email" passport-local would still expect your frontend login form to contain an input field with name "username" instead of email. This can be configured for passport-local but this is double the work. So we got this shortcut implemented.

8.
Async/Await

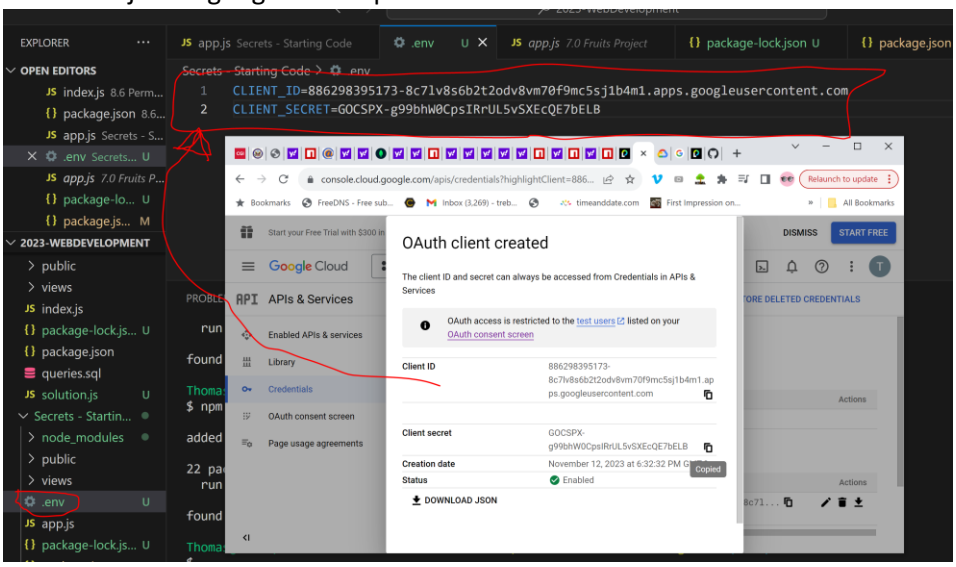
Starting from version `5.0.0`, passport-local-mongoose is async/await enabled by returning Promises for all instance and static methods except `serializeUser` and `deserializeUser`.

```
const user = new DefaultUser({username: 'user'});
await user.setPassword('password');
await user.save();
const { user } = await DefaultUser.authenticate()('user', 'password');
```

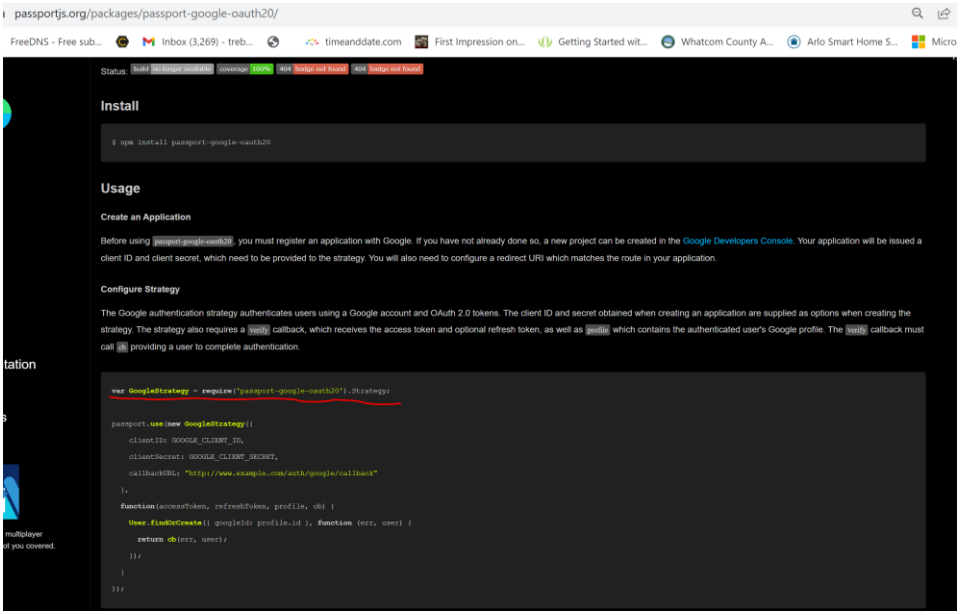
OAuth – Open Authorization

It is an open standard for token authorization.

Setup

1. https://www.passportjs.org/packages/passport-google-oauth20/
    a. This is the latest implementation of passport-google-auth
2. npm install passport-google-oauth20
3. https://www.passportjs.org/packages/passport-google-oauth20/
4. https://console.cloud.google.com/projectselector2/apis/dashboard?supportedpurview=project
    a. Create Project in google developer:
    b. 

5. Copy and paste code from passport-google-oauth20 usage:



a. Implement the pseudocode circled in red:

b.
```
var GoogleStrategy = require('passport-google-oauth20').Strategy;

passport.use(new GoogleStrategy({
    clientID: process.env.CLIENT_ID,
    clientSecret: process.env.CLIENT_SECRET,
    callbackURL: "http://localhost:3000/auth/google/secrets",
    userProfileURL: "https://www.googleapis.com/oauth2/v3/userinfo"
  },
  function(accessToken, refreshToken, profile, cb) {
    User.findOrCreate({ googleId: profile.id }, function (err, user) {
      return cb(err, user);
    });
  }
));
```

c. Fortunately this has been done for us

i. Install “npm install mongoose-findorcreate”

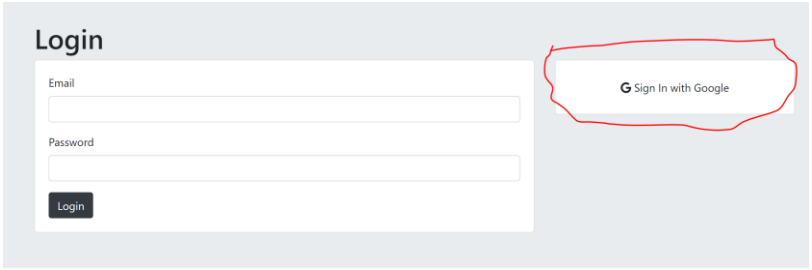
6. Add googleID to the User Schema to prevent db storing the same user twice

a.
```
const userSchema = new mongoose.Schema({
    email: String,
    password: String,
    googleId: String
});
```

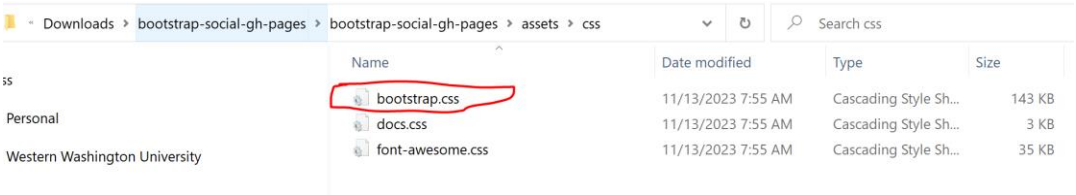
7. Redefine serialize/deserialize user:

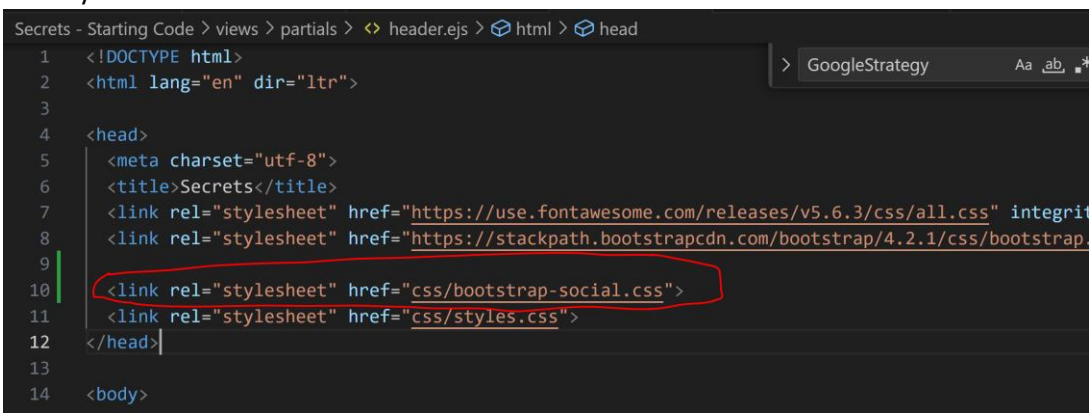
a.
```
passport.serializeUser(function(user, done) {//works with local authentication
    done(null, user);
  });

  passport.deserializeUser(function(user, done) {
    done(null, user);
  });
```

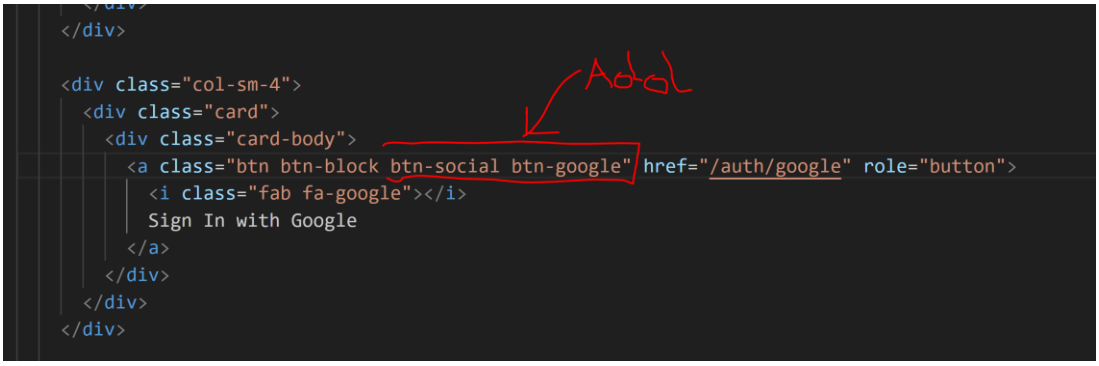
8.
```
app.get("/secrets", function(req,res){
    if(req.isAuthenticated()){
        res.render("secrets");
    }else{
        res.redirect("/login");
    }
});
```

9. Customize sign in with google
   a. 

   b. Download social buttons for bootstrap found in: https://lipis.github.io/bootstrap-social/
   c. Copy bootstrap.css into my project

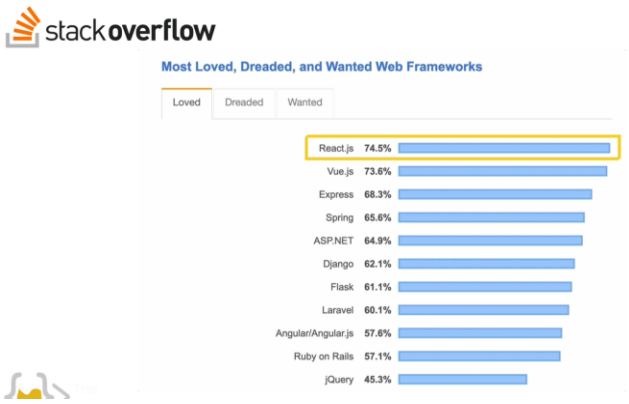


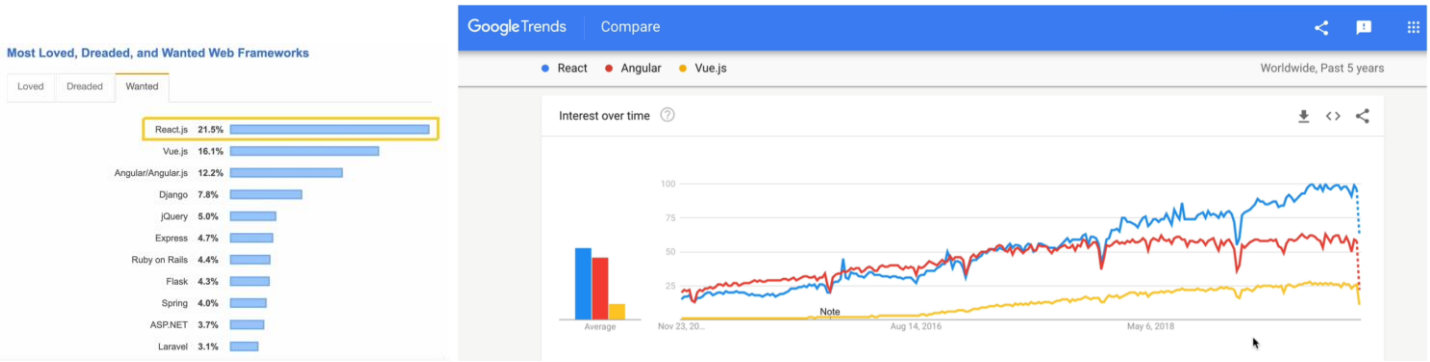
   d. Add stylesheet to header
   e. 

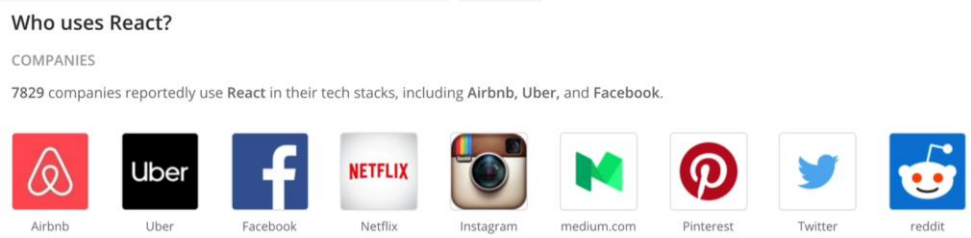
   f. Adapt the register and login ejs for the new stylesheet
   g. 

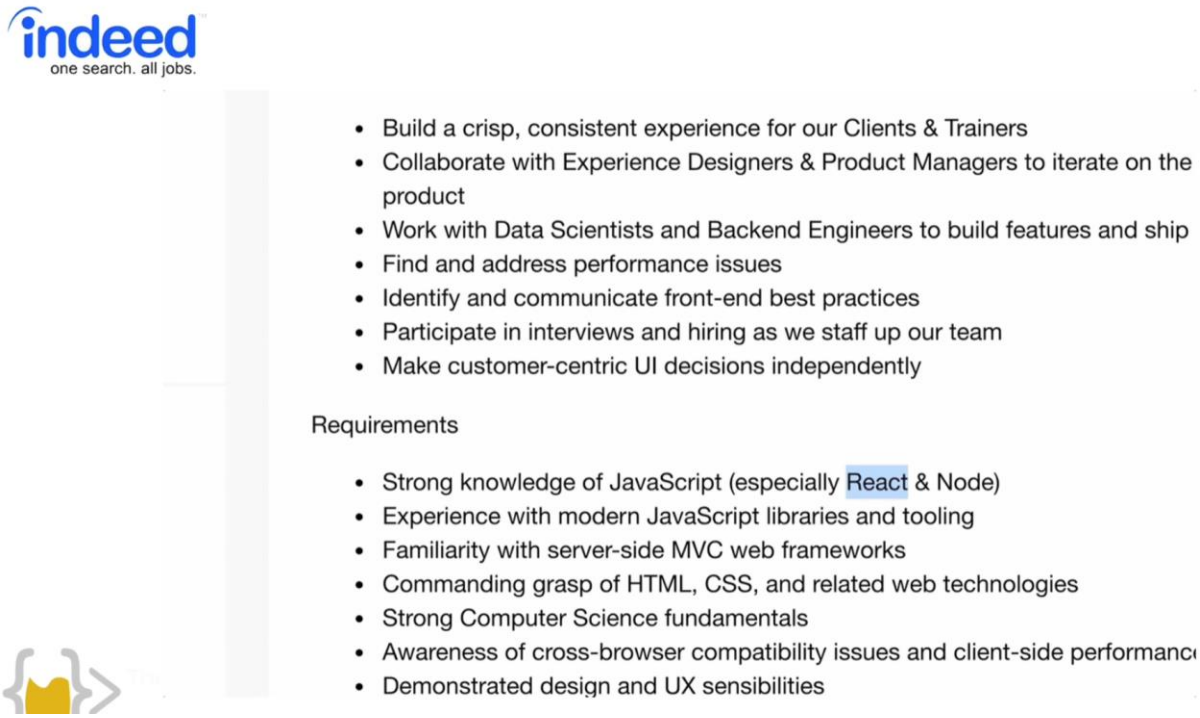
React.js

React is a front-end framework – it is the most loved and wanted JavaScript library for building user interfaces. (see: Airbnb)



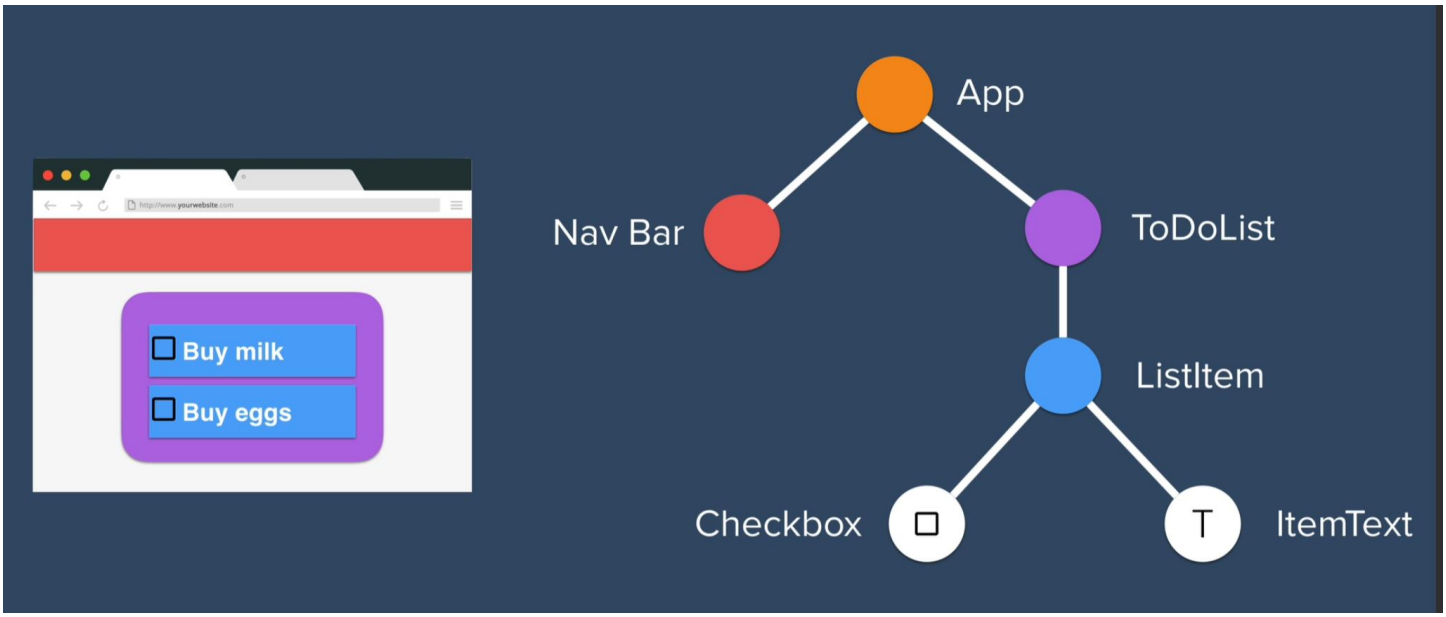






Fundementals

React.js breaks down a very complex component structure into a component tree:



This approach vastly simplifies the organization of your website, it makes your code modular reusable.

This is achieved through mixing of HTML, CSS and JS into a single component.

Client User performance is improved through refreshing only the dynamic components of a page not the static.

Setting up Development Environment

1. Access codesandbox from: https://codesandbox.io/s/react-new
2. Add Root:
   a.
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>React App</title>
    <link rel="stylesheet" href="styles.css" />
  </head>

  <body>
    <div id="root"></div>
    <script src="../src/index.js" type="text/javascript"></script>
  </body>
</html>
```
3. The index.html file does not get touched again….
4. Go to index.js and import react and react-dom
   a. In code Sandbox it is as easy as searching for a dependency then clicking on it to install it.
5.
```
ReactDOM.render(WHAT TO SHOW, WHERE TO SHOW IT);
```

```
6. import { StrictMode } from "react";//allows us to use JSX
7. import { createRoot } from "react-dom/client";
8.
9. const rootElement = document.getElementById("root");
10.     const root = createRoot(rootElement);
11.
12.     root.render(
13.     <h1>Hello World</h1>
14.     );
```

15. The babel engine works behind the scenes to generate browser compatible java script code
    a. Try it out at: https://babeljs.io/
    b. Babell renders the newest versions of javaScript es6 into plain old javaScript in order to allow universal browser compatibility.

16.
17. Root.render allows you to render one lelement and only one element however children can be embedded within.
    a. You can use the element and put a bunch of stuff inside it:

```
import { createRoot } from "react-dom/client";

const rootElement = document.getElementById("root");
const root = createRoot(rootElement);

root.render(
  <div>
    <h1>Hello World</h1>
    <ul>
      <li>Coffee</li>
      <li>Tea</li>
      <li>Milk</li>
    </ul>
  </div>
);
```

    b.

18. JSX allows client webpage to be embedded with javaScript containing HTML which contains inner javaScript

19. To insert a javaScript variable into html code simply wrap curly braces around it and presto:

20. The caveat is that you can only place expressions akin go <%= …%> inside the curly braces but not an entire statement such as if.. then ..else …return.

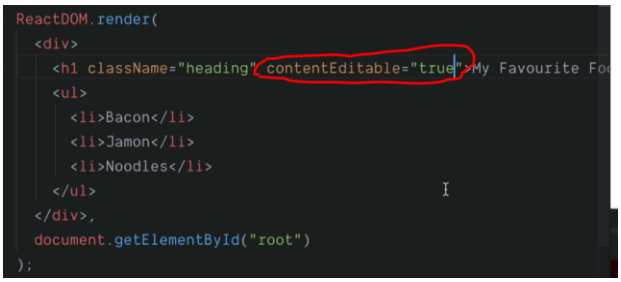
JSX Attribute and Styling

Here we are building JSX files and not just regular javaScript files. JSX epressions are a bit different so to set attributes in html we use standard attributes found in https://www.w3schools.com/html/html_attributes.asp which are not camel case so the following would make sense:

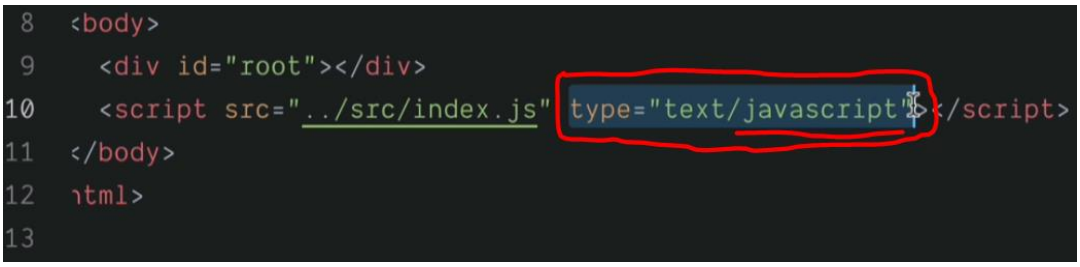
```
ReactDOM.render(
  <div>
    <h1 class="heading">My Favourite Foods</h1>
    <ul>
      <li>Bacon</li>
      <li>Jamon</li>
      <li>Noodles</li>
    </ul>
  </div>,
  document.getElementById("root")
);
```

But in JSX we use camel case such as the term className instead of class

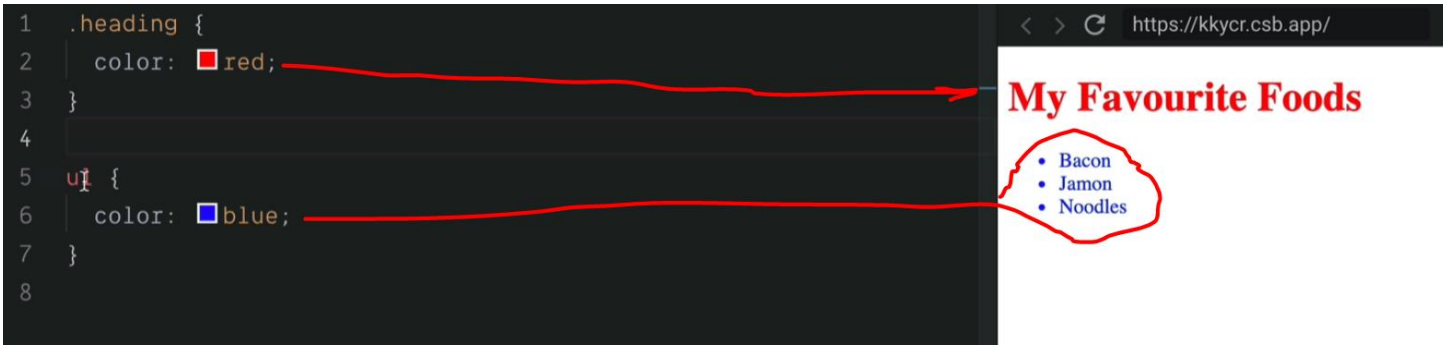
```
ReactDOM.render(
  <div>
    <h1 className="heading">My Favourite Foods</h1>
    <ul>
      <li>Bacon</li>
      <li>Jamon</li>
      <li>Noodles</li>
    </ul>
  </div>,
  document.getElementById("root")
);
```



'contenteditable' is a standard html attribute that you set to camel case 'contentEditable' in order to be JSX. Change class to className and within the script source designation of index.html change the javascript designation to JSX designation:



You can target elements for css styling by id, by elemet type (and all children) such as



See: https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementsByClassName

Examples

Get all elements that have a class of 'test':

JS

```
document.getElementsByClassName("test");
```

Get all elements that have both the 'red' and 'test' classes:

JS

```
document.getElementsByClassName("red test");
```

Get all elements that have a class of 'test', inside of an element that has the ID of 'main':

JS

```
document.getElementById("main").getElementsByClassName("test");
```

Get the first element with a class of 'test', or `undefined` if there is no matching element:

JS

```
document.getElementsByClassName("test")[0];
```

We can also use methods of Array.prototype on any `HTMLCollection` by passing the `HTMLCollection` as the method's *this* value. Here we'll find all div elements that have a class of 'test':

JS

```
const testElements = document.getElementsByClassName("test");
const testDivs = Array.prototype.filter.call(
  testElements,
  (testElement) => testElement.nodeName === "DIV",
);
```

Inline Styling

Use inline styles if you want to update styles on the fly.

```
1  import React from "react";
2  import ReactDOM from "react-dom";
3
4
5
6  ReactDOM.render(<h1 style={{color: "red"}}>Hello World!</h1>,document.getEle
7
```

1. Inline styles require javaScript objects to be passed as inputs.. these are key value pairs in curly brace blocks. The first curly brace designates a parameter entry point.
    a. The style property requires a javaScript object input.

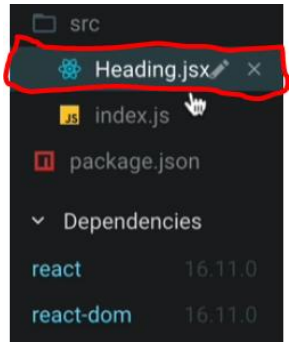
```
import React from "react";
import ReactDOM from "react-dom";
const newStyle = {backgroundColor: "powderblue",color: "Black"}
newStyle.color="red";

ReactDOM.render(<h1 style = {newStyle}>Hello World!</h1>, document.getElementById("root"));
```

React Components

JSX is html script intermingling with javaScript.

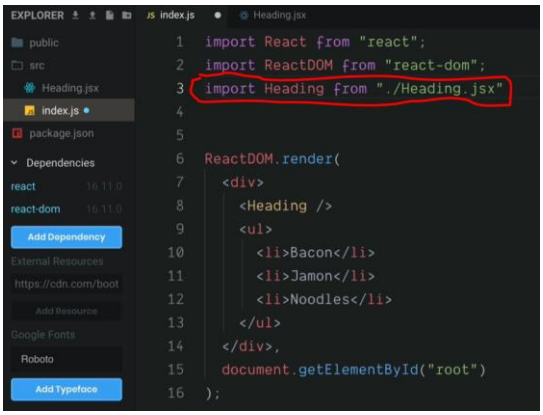
JSX does this by converting html into javaScript functions

Components allow us to split up a large complex file.

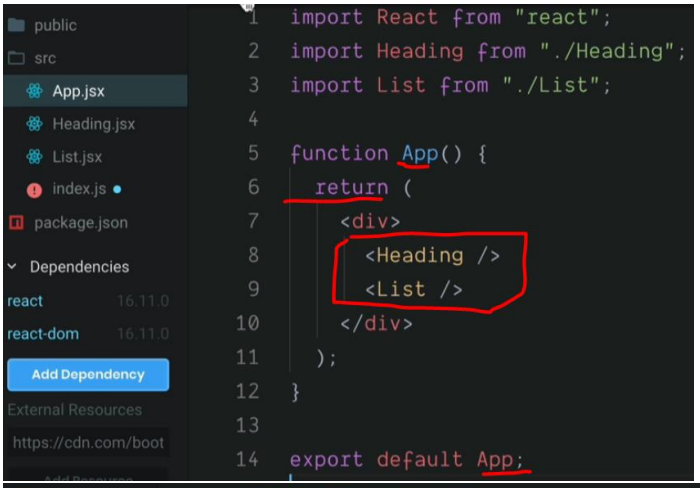
1. Name functions by the pascal method where all functions have the first letter capitalized.
2. It will look for a definition of the function within the render method if capitalization is used:
   a.
```
function Heading() {
  return <h1>My Favourite Foods</h1>;
}

ReactDOM.render(
  <div>
    <Heading></Heading>
    <ul>
      <li>Bacon</li>
      <li>Jamon</li>
      <li>Noodles</li>
    </ul>
  </div>,
  document.getElementById("root")
);
```
   b. That is how we can differentiate between common html components and our own dynamic behavioral components.
3. JSX best practices can be found in: https://airbnb.io/javascript/react/
4. Add components as individual .jsx files to the source folder
   a.


   b. Move the heading component from index.js to Heading.jsx
   c. Don't forget to import "react" into the Heading.jsx component so we can use JSX code.
   d. Export heading so it can be used as a function:
   e.


   f. Now import the heading file into index.js

g. 

h. The most common structure is to wrap all components inside an app file then deliver the app return to the index.js file

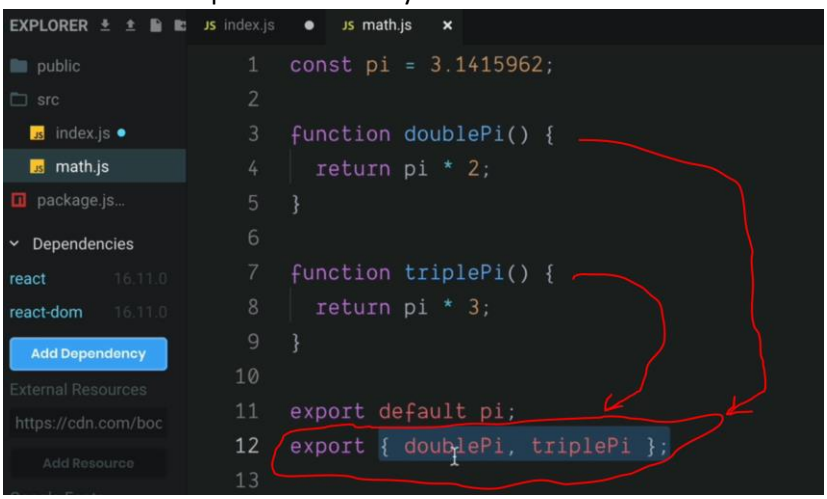
i. 

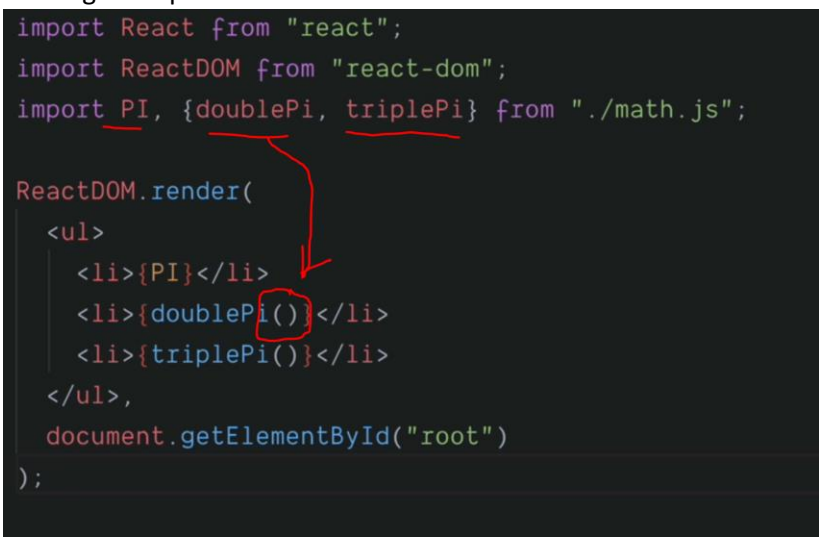
j. 

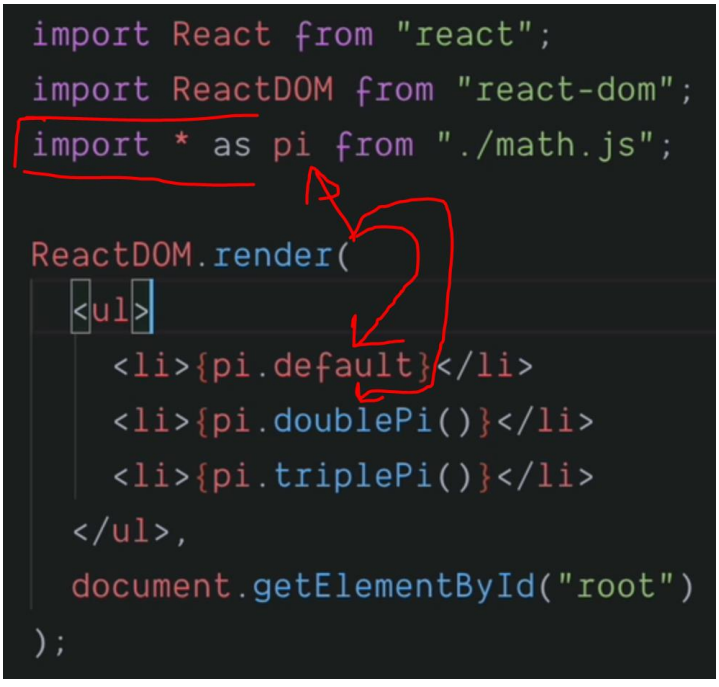
k. Create file folder hierarchy with components at the top to organize files.

l. 

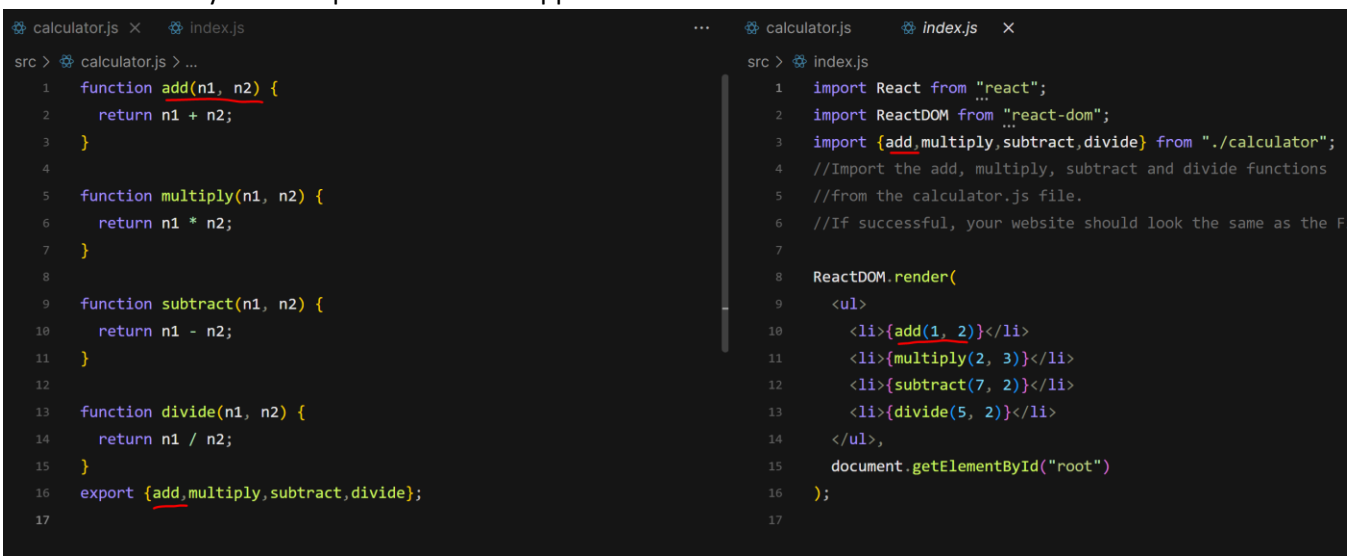
Multiple Export

1. The default export remains the same.
2. Add additional exports inside curly braces:
3. 

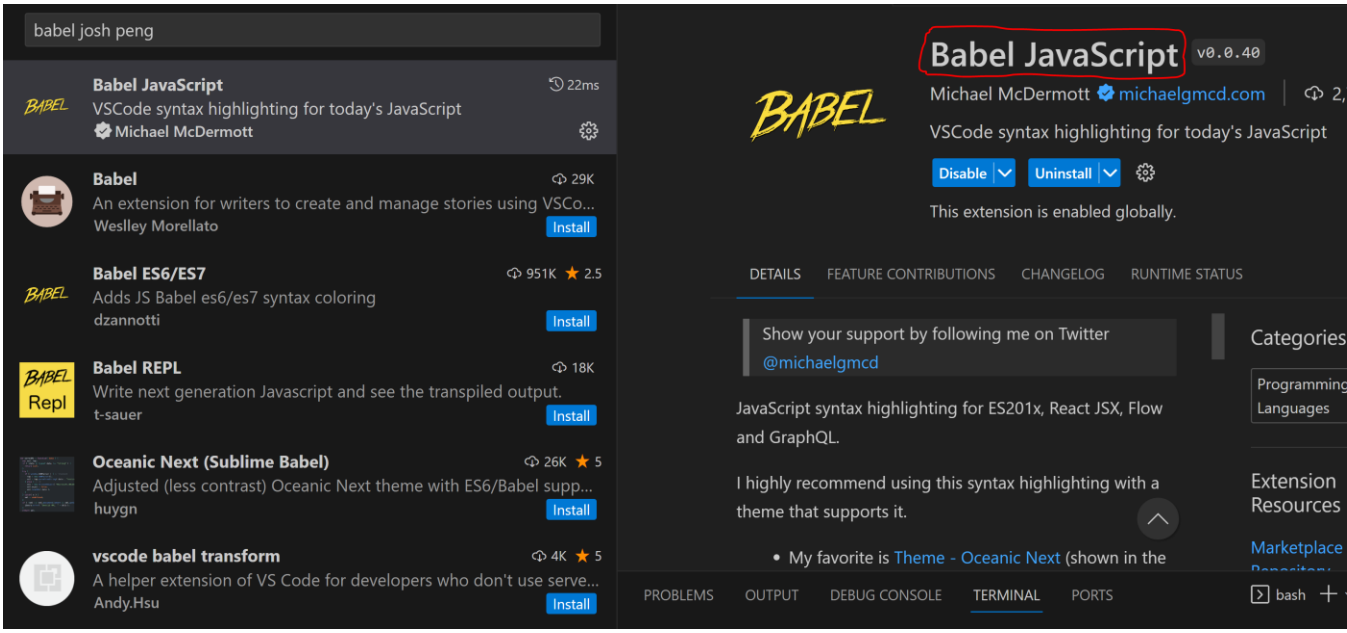
4. If we are Calling an imported method than this too has to be accounted for with brackets in the call:
   a. 

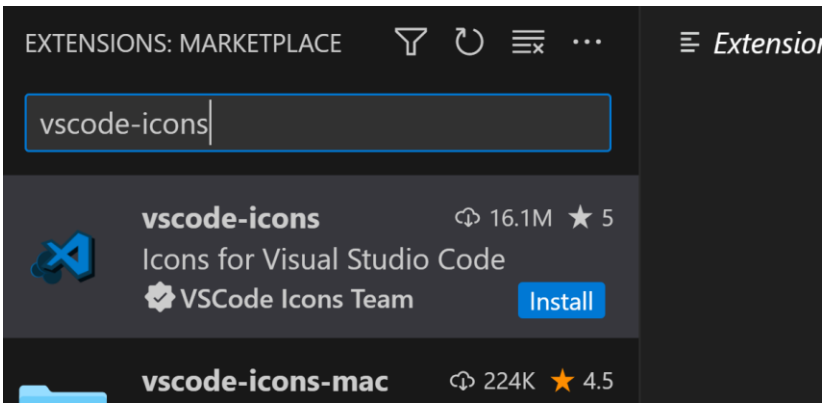
   b. You can also import everything (my preferred method discouraged in style guides):
   c. 

d. When exporting functions that take arguments do not represent them in either the export or the import statement …. Only in the imported function application



Setting up local environment for React development in VS Code

1. In VS Code click File->Preferences->Extensions and enter “Babel” into search and install
   a. 

2. Next add vscode-icons
   a. 

3. Create React app by following the steps in the react website: https://react.dev/learn/start-a-new-react-project
   a. Open terminal and enter: npx create-react-app my-app
      i. Where “my-app” is the app name

b. Cd over to the root project directory “cd my-app/”
c. Enter “npm start”:
   i.
```
Thomas@Triamphloton MINGW64 /e/2023-WebDevelopment
$ cd my-app/

Thomas@Triamphloton MINGW64 /e/2023-WebDevelopment/my-app (master)
$ npm start
Compiled successfully!

You can now view my-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://172.16.0.17:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```
d. Go to “Explorer” in VS Code
e. Select Public/index.html and add:
   i. <script src="./src/index.js" type="text/jsx"></script>
f. Select src/index.js
   i. Remove all imports except react and reactDom
   ii. Remove code below render line

4. Put Hello World into React.DOM.render(<h1>Hellow World</h`>, documents.getElement.ById(“root”));
5. Make sure the import of React DOM looks like this:
6.
```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App.jsx';
```
7. Step two is to create a header jsx component and export the header into index.js.
8. Always make sure that the name of object in import is capitalized or code will not work:

keeper > src > components > JS Header.jsx > header

```
1  //3. Create a Header.jsx component that renders a <header> element
2  //to show the Keeper App name in an <h1>.
3  import React from "react";
4
5  function header(){
6    return <div className = "header"><h1>Keeper App</h1></div>;
7  }
8
9  export default header;
```

9. Go back to App.js and import the header component, remember that React differentiates between standard html components and my components by using capital letters at the start of my component
10.

keeper > src > components > JS App.jsx > app

```
1   //2. Create a App.jsx component.
2   import React from "react";
3   import Header from "./Header.jsx";
4   import Note from "./Note.jsx";
5   import Footer from "./Footer.jsx";
6   function app(){
7       return (
8           <div>
9               <Header />
10              <Note />
11              <Footer />
12          </div>
13      );
14  }
15  export default app;
```

    a. Now test it to see if it works in Pascal case with the new heading and styling
    b.

keeper > src > components > JS App.jsx > app

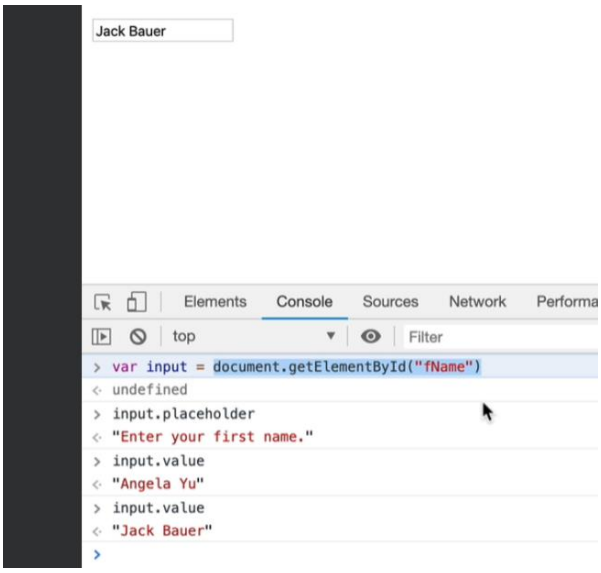
```
1   //2. Create a App.jsx component.
2   import React from "react";
3   import Header from "./Header.jsx";
4   function app(){
5       return (
6
7           <Header />
8
9       );
10  }
11  export default app;
```

    c. To implicitly assign a css style without using the .element notation simply return the element wrapped in the name of the element which has a style assigned to it such as header:
    d.

```
//3. Create a Header.jsx component that renders a <header> element
//to show the Keeper App name in an <h1>.
import React from "react";

function header(){
    return <header><h1>Keeper App</h1></header>;
}

export default header;
```

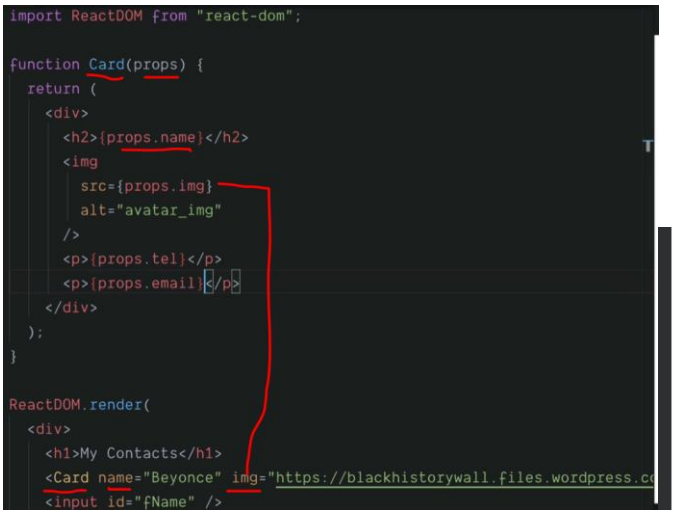
React Properties

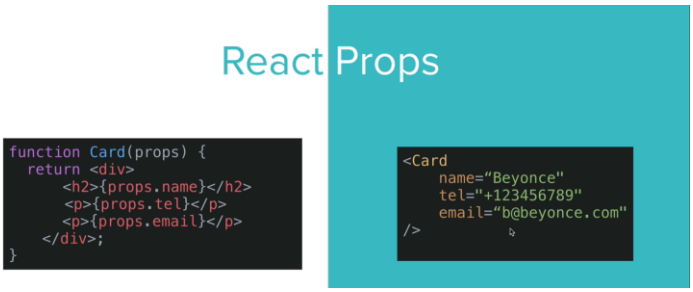
You can tap into any ‘document’ element using getElementById:



These component attributes are standard but if we build our own React components and develop attributes for them than the door is open to create custom attributes or PROPERTIES.

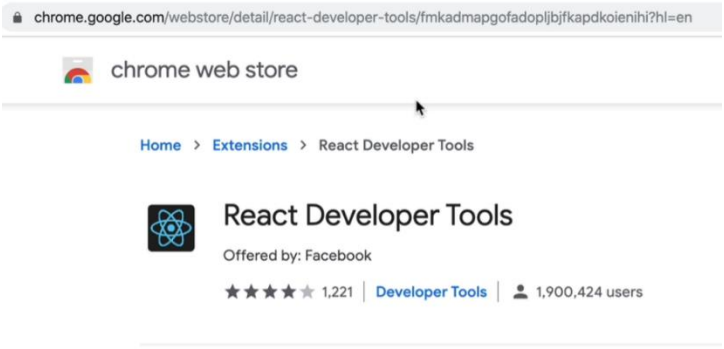
Props pass arguments to a kind of element constructor. When parsing out the props within the Card function/constructor that is the time when styling element names can be utilized.





React Dev Tools

Install React Developer Tools



Mapping Components (loops and functional programming(function within function))

The map function is useful in handling arrays.

The array.map(expects a function to be passed as argument to the function)

Remember that the “key” is not for us to use!! The “key” component in react is a special property and it is used to ensure the order of the objects is rendered efficiently it is not something we can tap into.

```
function createCard(contact){
  return <Card
  key = {contact.id}
  name = {contact.name}
  imgURL = {contact.imgURL}
  tel = {contact.phone}
  email = {contact.email}

  />
}

function App() {
  return (
    <div>
        <Avatar
        image = "https://tse4.mm.bir
        />
      <h1> My Contacts </h1>
      {contacts.map(createCard)}
```

The content of JavaScript react functions is always assumed to be HTML elements.

Mapping components process.

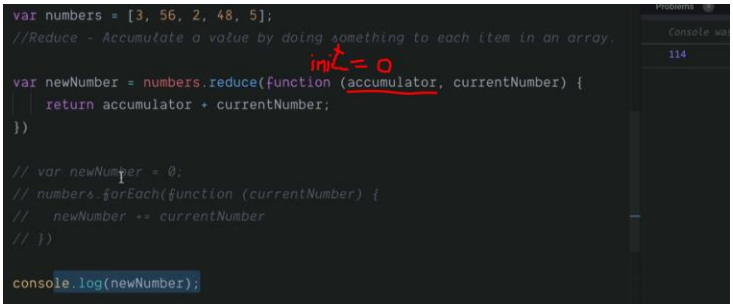
Getting Started:

1. npx create-react-app *appname_which_will_become_new_directory*
2. cd into the new directory and remove superfluous files.
3. Put some components into the App.js file
4. Decide on how you would like your website to look by modeling components directly in App.js (exported)

```
<dl className="dictionary">
  <div className="term">
    <dt>
      <span className="emoji" role="img" aria-label="Tense Biceps">
        💪
      </span>
      <span>Tense Biceps</span>
    </dt>
    <dd>
      "You can do that!" or "I feel strong!" Arm with tense biceps. Also
      used in connection with doing sports, e.g. at the gym.
    </dd>
  </div>
  <div className="term">
    <dt>
      <span className="emoji" role="img" aria-label="Tense Biceps">
        🙏
```

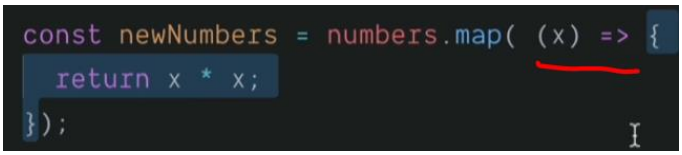
The <dl> component is description list and is described: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dl

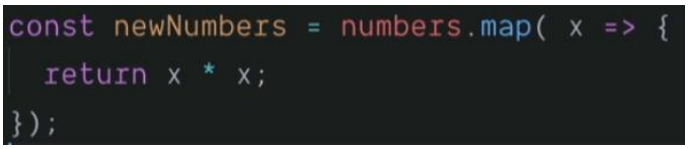
5. Start the react server with:
    a. 'npm start'
6. Take array data and mutate each element through functional programming:
    a.
```
//Map -Create a new array by doing something with each item in an array.

// function double(x) {
//   return x * 2;
// }
// const newNumbers = numbers.map(double);

// var newNumbers = [];
// numbers.forEach(function (x) {
//   newNumbers.push(x * 2);
// });

const newNumbers = numbers.map(function (x) {
  return x * 2;
});

console.log(newNumbers);
```
    b. Map method forgoes the need of creating an array in order to call the method on the array as seen above.
7. The Filter method keeps only those object in the array that return true.
    a.
```
var numbers = [3, 56, 2, 48, 5];
//Filter - Create a new array by keeping the items that return true.

const newNumbers = numbers.filter(function (num) {
  return num > 10
});

console.log(newNumbers);

//Reduce - Accumulate a value by doing something to each item in an array.
```
Problems
Console was
▸ [56, 48]

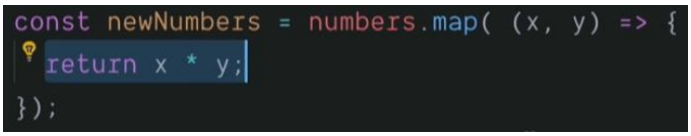
8. The Reduce Function can be used to sum up all members of an array:
    a. 
    ```
    var numbers = [3, 56, 2, 48, 5];
    //Reduce - Accumulate a value by doing something to each item in an array.

    var newNumber = numbers.reduce(function (accumulator, currentNumber) {
        return accumulator + currentNumber;
    })

    // var newNumber = 0;
    // numbers.forEach(function (currentNumber) {
    //     newNumber += currentNumber
    // })

    console.log(newNumber);
    ```
    b. More information can be found: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce
9. The Find method on an array returns the first element which satisfies the conditions:
    a. 
    ```
    var numbers = [3, 56, 2, 48, 5];


    //Find - find the first item that matches from an array.

    const newNumber = numbers.find(function (num) {
        return num > 10;
    })

    console.log(newNumber);

    //FindIndex - find the index of the first item that matches.
    ```

ES6 Arrow Functions

Arrow functions are a **shorter** way of writing javaScript functions.


```
const newNumbers = numbers.map( (x) => {
    return x * x;
});
```


```
const newNumbers = numbers.map( x => {
    return x * x;
});
```


```
const newNumbers = numbers.map( (x, y) => {
    return x * y;
});
```


```
const newNumbers = numbers.map( x => x * x);
```

For more information on arrow functions see: https://hacks.mozilla.org/2015/06/es6-in-depth-arrow-functions/

Conditional Rendering

The means by which authentication allows access.

Setup React Project:

1. npx create-react-app conditional_rendering
2. cd conditional_rendering
3. Remove superfluous files and add desired ones.
4. npm start

Single responsibility principle

See: https://en.wikipedia.org/wiki/Single-responsibility_principle

Setup React for conditional rendering

5. Define Boolean for logged in state
6. Test for logged in condition inside separate function which is called within the App() return(…)
7. Each component should have a single responsibility so break up app into Single-Responsibility principle components.
8. Do not forget that what goes into the curly braces of a react function must resolve to a statement without any intervening conditions therefore in order to perform inline conditional statements you must use the Ternary Operator recall: “CONDITION ? (DO IF TRUE) : (DO IF FALSE)

Ternary Operator

```
CONDITION ? DO IF TRUE : DO IF FALSE
```

a.
```
isCloudy === true ? bringUmbrella() : bringSunscreen()
```

b.
```
var isLoggedIn = false;



function App() {
  return <div className="container">{
    isLoggedIn === true ? <h1>Hello</h1> : <Login />

  }</div>;
}

export default App;
```

c. The AND operator in place of the Ternary operator:

i.
```
currentTime > 12 ? <h1>Why are you still working?</h1>
```

ii.
```
currentTime > 12 && <h1>Why are you still working?</h1>
```

d. This works because of the left to right evaluation implicit in this scripted operation… therefor if the left side “currentTime>12” evaluates to true than the right side gets evaluated and in react that means it gets rendered.

e. If the left side evaluated to false than the right side would not get evaluated resulting in null

State in React

State is central to how React does things. User interface behavior is a function of the State or else:

Declarative programming

This is done by tracking a variable which is central in determining type of behavior of the user interface.

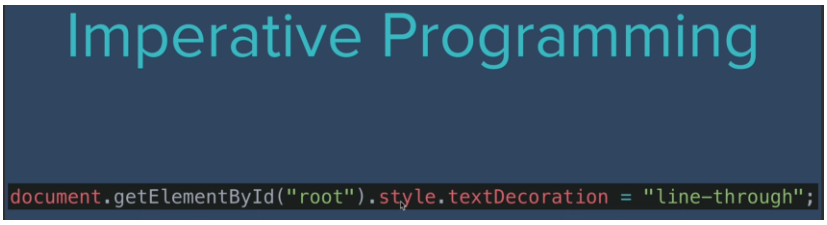
…isDone is the state

```
function App() {
  var isDone = true;

  const strikeThrough = { textDecoration: "line-through" };

  return <p style={isDone ? strikeThrough : null}>Buy milk</p>;
}
```

This approach is called declarative programming because it is dependent on the state!

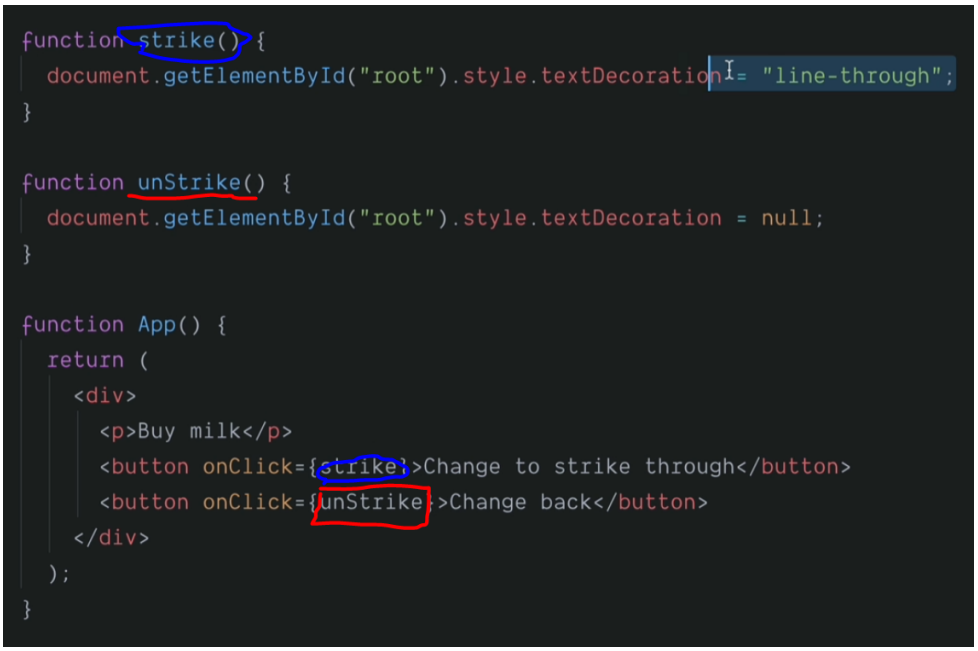
Imperative Programming

The other kind of programming is called Imperative Programming



Imperative Programming is when we tap into an element by id and we set it equal to something, than this is us imperatively telling this element to do something different

This approach requires a listener such as <button onClick….



Hooks

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

https://www.w3schools.com/react/react_hooks.asp

The button elements are being rendered and they are not changeable therefore the have to be RE-RENDERED onto the screen in order for the changes in their properties such as their style property to be seen updated.

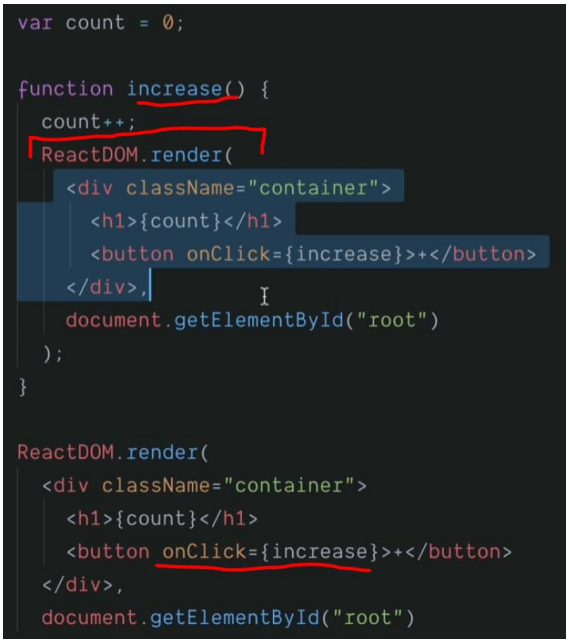
Hooks hook into the state of the App in order to make changes or modify it:

Things to keep in mind

1. All button elements have built-in attribute called on-click and we can set that to some sort of function to be triggered when a click occurs.
2. Convert standard HTML element names to camel-case in order to use React components in the program, in this way we can point the onClick to point to a function in our code
    a. 

3. If Hooks are not used than we are relying on the ReactDom.render inside index.js to render changes to the client therefore in order for my component to show an update to the client, the method ReactDom.render(<div className = …….) has to be called again when the onClick listener triggers the react increase function as seen below:

a.



```
var count = 0;

function increase() {
  count++;
  ReactDOM.render(
    <div className="container">
      <h1>{count}</h1>
      <button onClick={increase}>+</button>
    </div>,
    document.getElementById("root")
  );
}

ReactDOM.render(
  <div className="container">
    <h1>{count}</h1>
    <button onClick={increase}>+</button>
  </div>,
  document.getElementById("root")
```

4. However using Hooks requires abiding by certain Rules:
    a. You must use a HOOK inside a functional component
        i. You have to create a function which renders that component and then you can use a hook inside of that function.
    b. You can Call a hook function from react by envoking React.function or by importing this function from the React file with a set of curly braces to denote separate import then use it directly:
        i.
```
import React, { useState } from "react";

function App() {
  //var count = 0;
  const state = React.useState();
  function increase(){
    count++;
  }
```
    c. ..useState() takes initial or starting state as its parameter:
        i.
```
import React, { useState } from "react";

function App() {
  const state = useState(123);

  console.log(state);

  function increase() {
    //count++;
  }

  return (
    <div className="container">
      <h1>0</h1>
      <button onClick={increase}>+</button>
    </div>
  );
}
```
Initial State
https://ylxqj.csb.app/
Problems
React DevTools
[123, function bound dispatchAction()]
    d. In order to read the value stored in useState we reference it like any two element array with the first element containing our value.
        i.
```
function App() {
  const state = useState(123);

  console.log(state[0]);

  function increase() {
    //count++;
  }

  return (
    <div className="container">
      <h1>{state[0]}</h1>
      <button onClick={increase}>+</button>
    </div>
  );
}
```

Find great color palets encoded in RGB at https://flatuicolors.com/

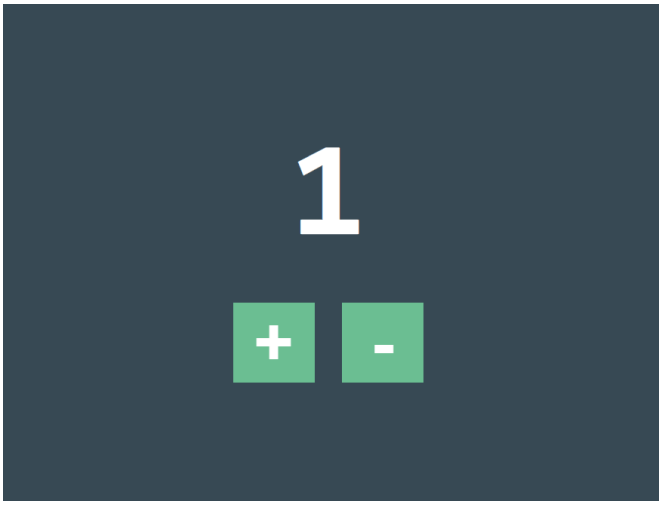
Destructuring – for useState

```
const [red, green, blue] = [9, 132, 227]
```

This allows us to map “red” to the first value of the array

The React.useState(initialValue) hook returns a two element array, the first element provides access to read the value while the second is a setter function.

```
const [count, setCount] = useState(0);
function increase(){

  setCount(count+1);
}
function decrease(){

  setCount(count-1);
}
return (
<div className = "container">
<h1>{count}</h1>
<button onClick={increase}>+</button>
<button onClick={decrease}>-</button>
</div>
);
```



Destructuring Arrays vs Objects

Use square brackets when destructuring an array with one assigned name per element in array.

<u>*Note:You do not have to do it for all of the keys and values!</u>

Use curly brackets when destructuring an object with one assigned name per object property:

```
import animals from "./data";

console.log(animals);
const [cat, dog] = animals;
console.log(cat);

const {name, sound} = cat;
```

```
import animals from "./data";

console.log(animals);
const [cat, dog] = animals;
console.log(cat);

const { name: catName, sound: catSound } = cat;
console.log(catName);
```

The object property names **<u>must match the actual object properties!! Unless you rename it with colon operator</u>**

Assigning default values to destructured objects

A default property value can be assigned to an object which does not contain that property automatically during destructuring:

```
console.log(animals);
const [cat, dog] = animals;
console.log(cat);

// const { name, sound} = cat;

// const { name: catName, sound: catSound } = cat;

const { name = "Fluffy", sound = "Purr" } = cat;
console.log(name);
```

If the cat object did not already have a name property assigned to it than “Fluffy” would automatically be assigned.

Destructuring Nested Objects

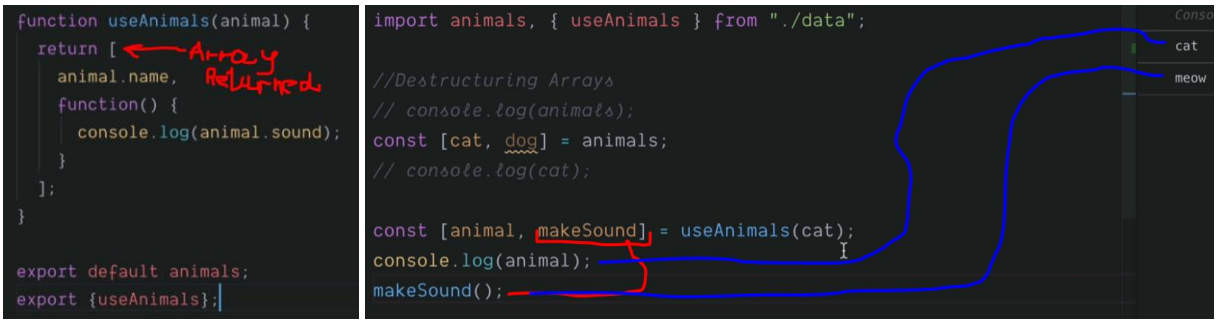
```
const animals = [
  {
    name: "cat",
    sound: "meow",
    feedingRequirements: {
      food: 2,
      water: 3
    }
  },
  { name: "dog", sound: "woof" }
];

export default animals;
```

```
const [cat, dog] = animals;
console.log(cat);

// const { name, sound} = cat;

// const { name: catName, sound: catSound } = cat;

// const { name = "Fluffy", sound = "Purr" } = cat;

const { name, sound, feedingRequirements: {food, water} } = cat;

console.log(food);
```

```
const cars = [
  {
    model: "Honda Civic",
    //The top colour refers to the first item in the
    //i.e. hondaTopColour = "black"
    coloursByPopularity: ["black", "silver"],
    speedStats: {
      topspeed: 140,
      zeroToSixty: 8.5
    }
  },
  {
    model: "Tesla Model 3",
    coloursByPopularity: ["red", "white"],
    speedStats: {
      topSpeed: 150,
      zeroToSixty: 3.2
    }
  }
];
export default cars;
```

```
const {coloursByPopularity:[teslaTopColour],speedStats:{topSpeed:teslaTopSpeed}}=tesla;
const {coloursByPopularity:[hondaTopColour],speedStats:{topSpeed:hondaTopSpeed}}=honda;
ReactDOM.render(
  <table>
    <tr>
      <th>Brand</th>
      <th>Top Speed</th>
    </tr>
    <tr>
      <td>{tesla.model}</td>
      <td>{teslaTopSpeed}</td>
      <td>{teslaTopColour}</td>
    </tr>
    <tr>
      <td>{honda.model}</td>
      <td>{hondaTopSpeed}</td>
      <td>{hondaTopColour}</td>
    </tr>
  </table>,
  document.getElementById("root")
);
```

Destructuring useState analog



Event Handling

HTML offers a wide range of event handlers, see: https://www.w3schools.com/tags/ref_eventattributes.asp

However when referencing them in JSX remember to convert it to camel case.

```jsx
import React,{useState} from "react";
function App() {

const [headingText,setHeadingText] = useState("Hello");
const [getColor,setColor] = useState("white");
function handleClick(){
  console.log("stop touching me")
  setHeadingText("Submitted");
}
function handleHover(){
  setColor("black");
}
function handleMouseOut(){
  setColor("white");
}
  return (
    <div className="container">
      <h1>{headingText}</h1>
      <input type="text" placeholder="What's your name?" />
      <button style={{backgroundColor : getColor}} onClick={handleClick} onMouseOver={handleHover} onMouseOut={handleMouseOut}>Submit</button>
    </div>
  );
}
```