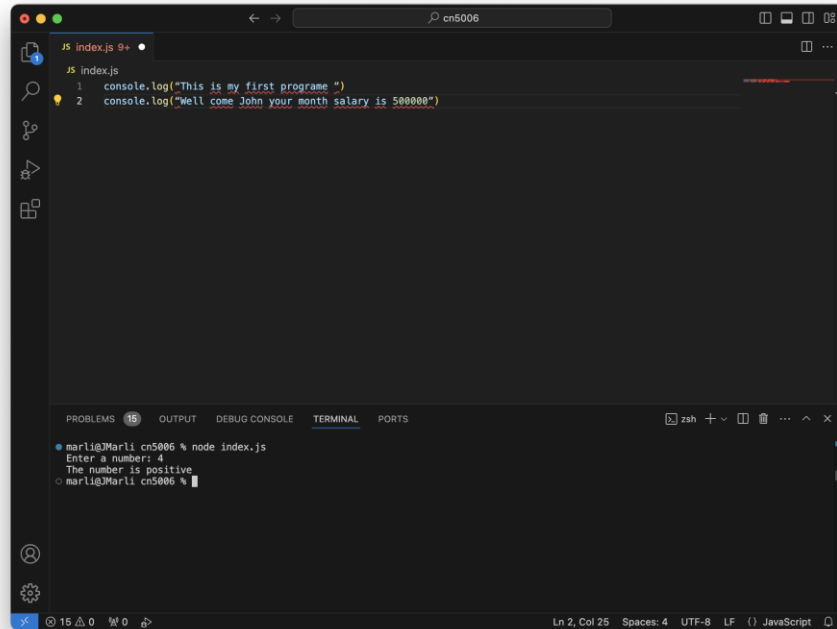


CN5006 PORTFOLIO:

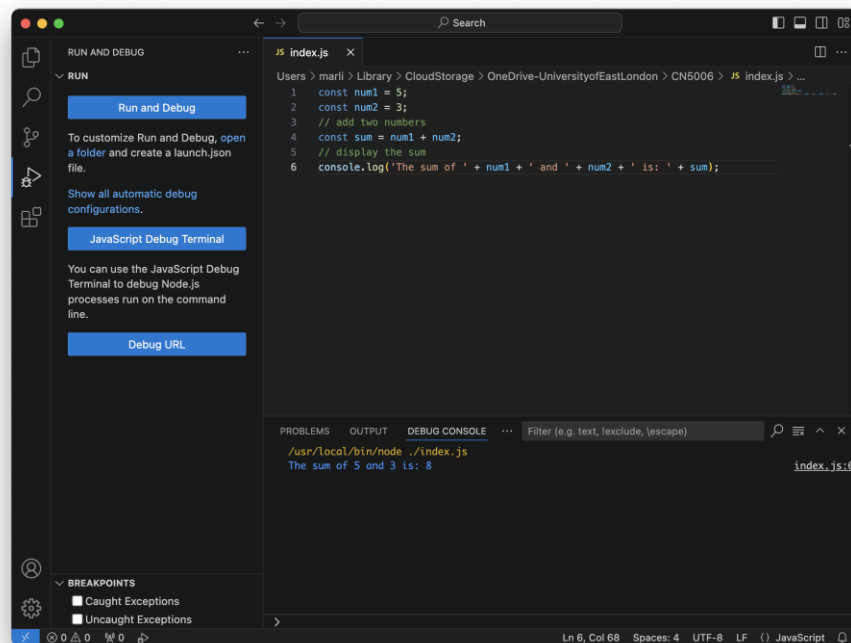
Week 1 code:



The screenshot shows the Visual Studio Code editor with a file named `index.js` open. The code contains two `console.log` statements. The first logs the string "This is my first programe ". The second logs the string "Well come John your month salary is 500000". Below the editor, the TERMINAL panel is active, showing the command `marli@Marli: cn5006 % node index.js` and its output: "Enter a number: 4", "The number is positive", and the prompt `marli@Marli: cn5006 %`.

```
JS index.js 9+
1 console.log("This is my first programe ")
2 console.log("Well come John your month salary is 500000")

PROBLEMS 15 OUTPUT DEBUG CONSOLE TERMINAL PORTS
marli@Marli: cn5006 % node index.js
Enter a number: 4
The number is positive
marli@Marli: cn5006 %
```



The screenshot shows the Visual Studio Code editor with a file named `index.js` open. The code defines two variables, `num1` and `num2`, with values 5 and 3 respectively. It then calculates their sum and logs it using `console.log`. The output of the program is "The sum of 5 and 3 is: 8". The left sidebar shows the "RUN AND DEBUG" panel with the "Run and Debug" button highlighted. The bottom status bar indicates the file is at line 6, column 68, using UTF-8 encoding and LF line endings.

```
JS index.js x
1 const num1 = 5;
2 const num2 = 3;
3 // add two numbers
4 const sum = num1 + num2;
5 // display the sum
6 console.log('The sum of ' + num1 + ' and ' + num2 + ' is: ' + sum);

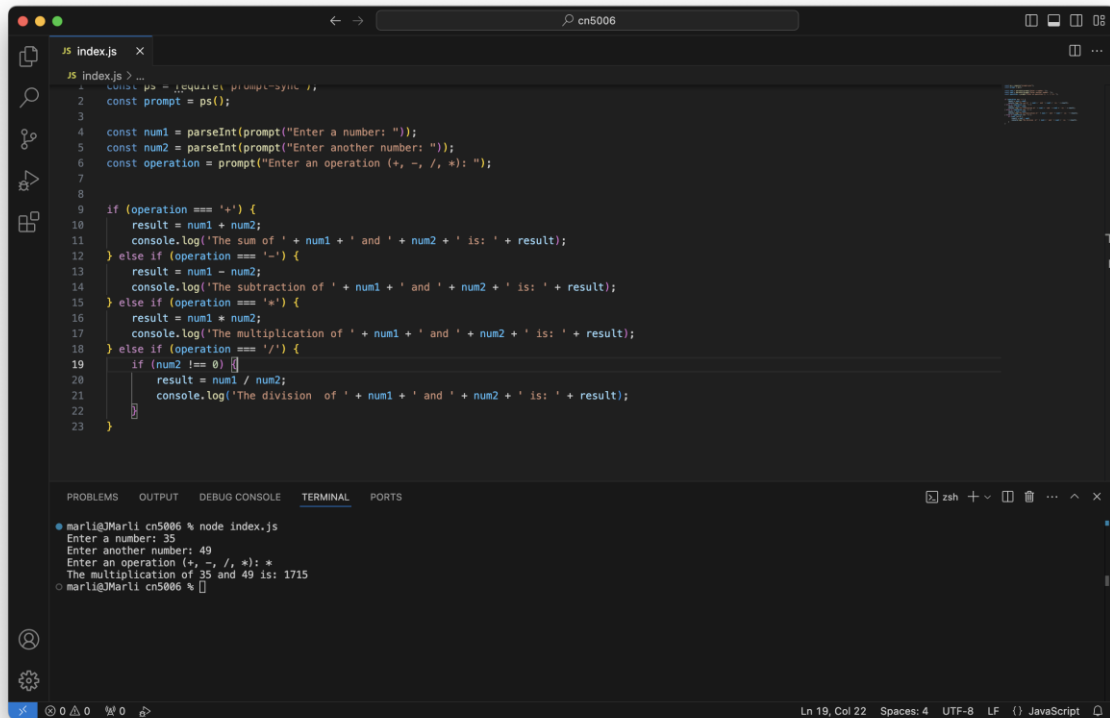
PROBLEMS OUTPUT DEBUG CONSOLE ... Filter (e.g. text, exclude, escape)
/usr/local/bin/node ./index.js
The sum of 5 and 3 is: 8
index.js:6
```

Self-evaluation:

I have improved my java script language skills and will work more to improve them. This was a great refresher after the summer holiday.

Exercise:

U2283556



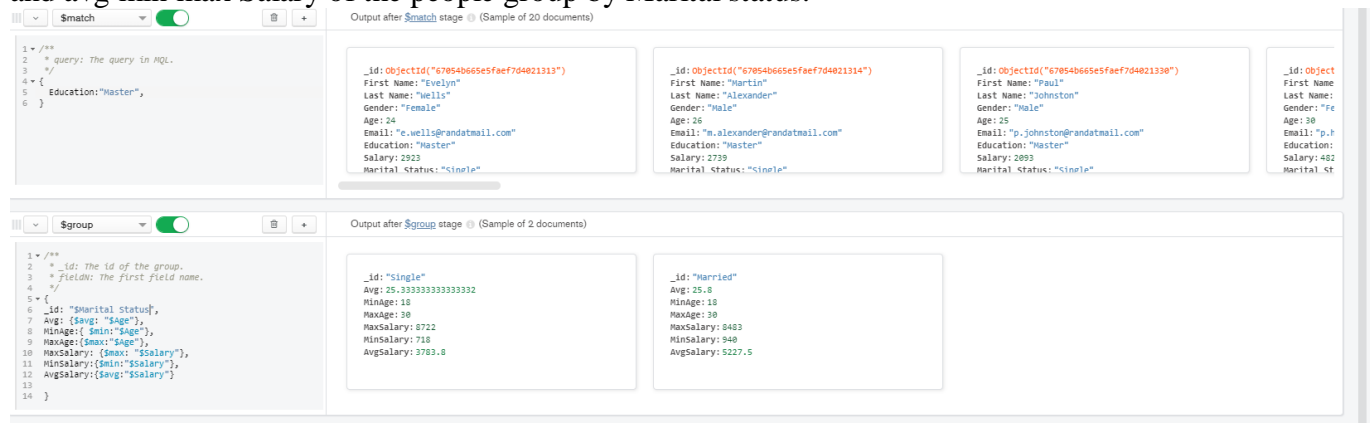
```
1 // index.js
2 const ps = require('prompt-sync');
3 const prompt = ps();
4
5 const num1 = parseInt(prompt("Enter a number: "));
6 const num2 = parseInt(prompt("Enter another number: "));
7 const operation = prompt("Enter an operation (+, -, /, *): ");
8
9 if (operation === '+') {
10   result = num1 + num2;
11   console.log('The sum of ' + num1 + ' and ' + num2 + ' is: ' + result);
12 } else if (operation === '-') {
13   result = num1 - num2;
14   console.log('The subtraction of ' + num1 + ' and ' + num2 + ' is: ' + result);
15 } else if (operation === '*') {
16   result = num1 * num2;
17   console.log('The multiplication of ' + num1 + ' and ' + num2 + ' is: ' + result);
18 } else if (operation === '/') {
19   if (num2 !== 0) {
20     result = num1 / num2;
21     console.log('The division of ' + num1 + ' and ' + num2 + ' is: ' + result);
22   }
23 }
```

```
marli@Marli cn5006 % node index.js
Enter a number: 35
Enter another number: 49
Enter an operation (+, -, /, *): *
The multiplication of 35 and 49 is: 1715
marli@Marli cn5006 %
```

This program was created to function as a calculator that is capable of four kinds of operations.

Week 2 lab tasks:

1) Repeat the same process to search Education for Master and .Find the avg,min,max age and avg min max Salary of the people group by Marital status.



The screenshot shows the MongoDB Compass interface. The left pane displays a query: `{ "Education": "Master" }`. The right pane shows the results of the query, which are three documents. Each document contains fields: `_id`, `First Name`, `Last Name`, `Gender`, `Age`, `Email`, `Education`, `Salary`, and `Marital Status`.

Document 1	Document 2	Document 3
<code>{ "_id": "ObjectId('67954b665efae764821313')", "First Name": "Evelyn", "Last Name": "Wells", "Gender": "Female", "Age": 24, "Email": "e.wells@randatmail.com", "Education": "Master", "Salary": 2923, "Marital Status": "Single" }</code>	<code>{ "_id": "ObjectId('67954b665efae764821314')", "First Name": "Martin", "Last Name": "Alexander", "Gender": "Male", "Age": 26, "Email": "m.alexander@randatmail.com", "Education": "Master", "Salary": 2739, "Marital Status": "Single" }</code>	<code>{ "_id": "ObjectId('67954b665efae764821336')", "First Name": "Paul", "Last Name": "Johnston", "Gender": "Male", "Age": 25, "Email": "p.johnston@randatmail.com", "Education": "Master", "Salary": 2093, "Marital Status": "Single" }</code>

Below the first set of results, there is a section for the `$group` stage. The query is: `{ "$group": { "_id": "$Marital status", "Avg": { "$avg": "$Age" }, "MinAge": { "$min": "$Age" }, "MaxAge": { "$max": "$Age" }, "MinSalary": { "$min": "$Salary" }, "MaxSalary": { "$max": "$Salary" }, "AvgSalary": { "$avg": "$Salary" } } }`. The results show two documents: one for `"Single"` and one for `"Married"`.

Document 1 (Single)	Document 2 (Married)
<code>{ "_id": "Single", "Avg": 25.333333333333332, "MinAge": 18, "MaxAge": 30, "MinSalary": 7722, "MaxSalary": 7718, "AvgSalary": 3783.8 }</code>	<code>{ "_id": "Married", "Avg": 25.8, "MinAge": 18, "MaxAge": 30, "MinSalary": 6403, "MaxSalary": 940, "AvgSalary": 5227.5 }</code>

U2283556

2) 2. find min,max average salary of each age group of female

Output after **\$match** stage (i) (Sample of 20 documents)

```
1 /*
2 * query: The query in MQL.
3 */
4 {
5   Gender: "Female",
6 }
```

Output after **\$group** stage (i) (Sample of 13 documents)

```
1 /*
2 * _id: The id of the group.
3 * fields: The first field name.
4 */
5 {
6   _id: "$age",
7   Avg: { $avg: "$age" },
8   MinAge: { $min: "$age" },
9   MaxAge: { $max: "$age" },
10  MaxSalary: { $max: "$salary" },
11  MinSalary: { $min: "$salary" },
12  AvgSalary: { $avg: "$salary" }
13 }
14 }
```

3) find min,max average salary of each age group of male

Output after **\$match** stage (i) (Sample of 20 documents)

```
1 /*
2 * query: The query in MQL.
3 */
4 {
5   Gender: "Male",
6 }
```

Output after **\$group** stage (i) (Sample of 13 documents)

```
1 /*
2 * _id: The id of the group.
3 * fields: The first field name.
4 */
5 {
6   _id: "$age",
7   Avg: { $avg: "$age" },
8   MinAge: { $min: "$age" },
9   MaxAge: { $max: "$age" },
10  MaxSalary: { $max: "$salary" },
11  MinSalary: { $min: "$salary" },
12  AvgSalary: { $avg: "$salary" }
13 }
14 }
```

4) Count married and unmarried females and males.

Documents Aggregations Schema Explain Plan Indexes Validation

200 Documents in the Collection

Select an operator to construct expressions used in the aggregation pipeline stages. [Learn more](#)

Output after **\$match** stage (i) (Sample of 20 documents)

```
1 /*
2 * query: The query in MQL.
3 */
4 {
5   'Marital Status': 'Married',
6 }
```

Output after **\$group** stage (i) (Sample of 2 documents)

```
1 /*
2 * _id: The id of the group.
3 * fields: The first field name.
4 */
5 {
6   _id: "$gender",
7   $sum: 1
8 }
9 }
10 }
```

U2283556

The screenshot shows the MongoDB Atlas Aggregations interface. At the top, there are tabs for Documents, Aggregations (selected), Schema, Explain Plan, Indexes, and Validation. The top right corner displays statistics: 200 documents, 38.1KB total size, 195B avg size, 1 index, 36.0KB total size, and 36.0KB avg size. The main area shows a pipeline with three stages: \$match, \$group, and \$sample. The \$match stage is currently selected, and its output is displayed as a sample of 20 documents. The \$group stage is also visible, and its output is shown as a sample of 2 documents. The \$sample stage is at the bottom, and its output is shown as a sample of 2 documents. The interface includes a 'SAVE' button and a 'SAMPLE MODE' toggle.

```
1 // **
2 * query: The query in HQL.
3 */
4 {
5   "Marital Status": "Single",
6 }
```

```
1 // **
2 * _id: The id of the group.
3 * fieldN: The first field name.
4 */
5 {
6   "_id": "Male",
7   "fieldN": 1
8 }
9
```

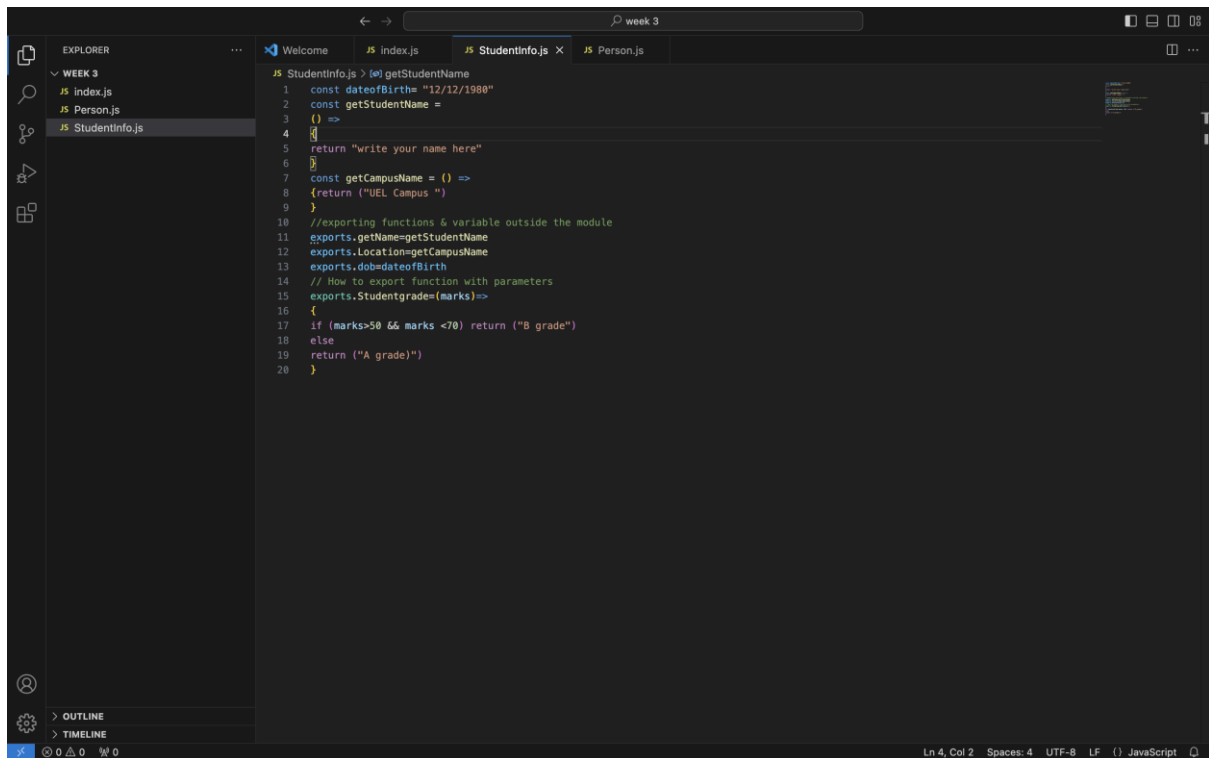
```
1 // **
2 * _id: The id of the group.
3 * fieldN: The first field name.
4 */
5 {
6   "_id": "Female",
7   "fieldN": 1
8 }
9
```

Report:

The work that been done with this code has been used to separate the data that has been given from a spread sheet then sorted into different categories depending on the need.

WEEK 3:

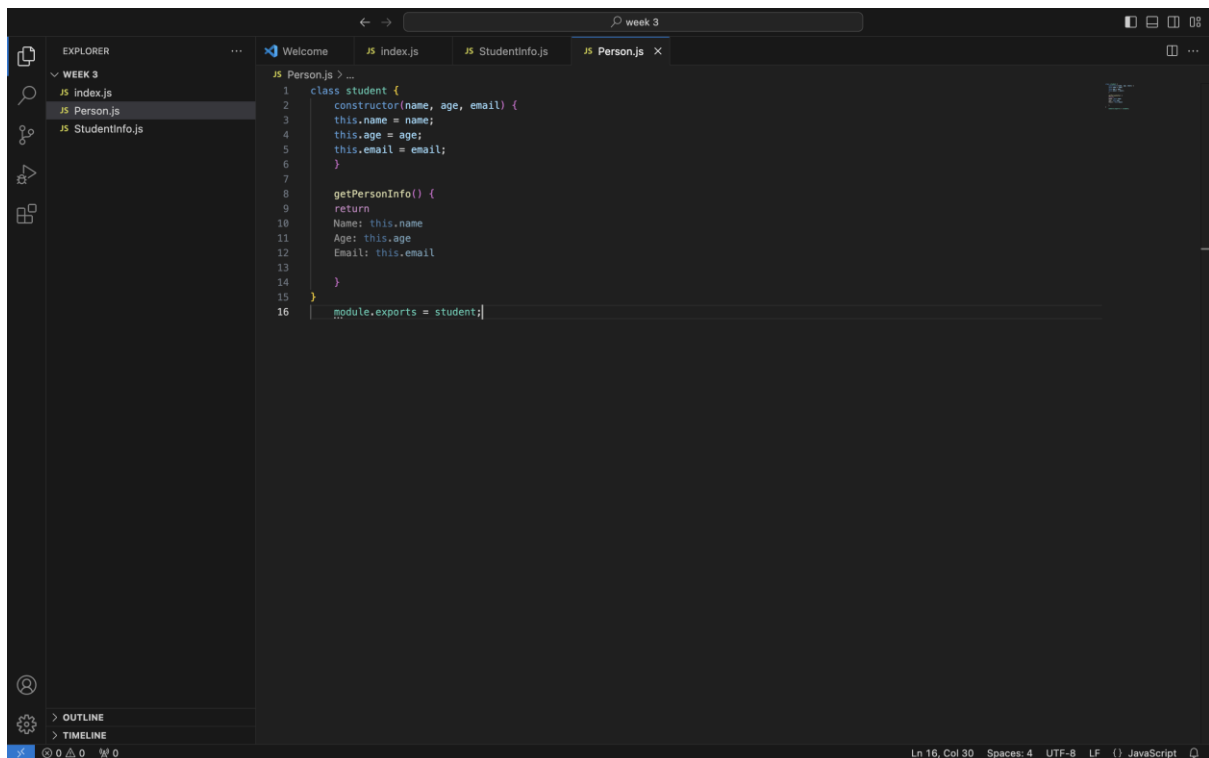
Submit the code for the completed Exercises i.e.: i) Index.js ii) Person.js iii) Employeeinfo.js iv) Exercise 4.js



This screenshot shows the Visual Studio Code editor with the file `StudentInfo.js` open. The Explorer sidebar on the left shows a project structure for 'WEEK 3' containing `index.js`, `Person.js`, and `StudentInfo.js`. The main editor area displays the following JavaScript code:

```
1  const dateOfBirth = "12/12/1980"
2  const getStudentName = () =>
3  {
4    // Write your name here
5    return "write your name here"
6  }
7  const getCampusName = () =>
8  {
9    return ("UEL Campus ")
10 }
11 //exporting functions & variable outside the module
12 exports.getName=getStudentName
13 exports.Location=getCampusName
14 exports.dob=dateOfBirth
15 // How to export function with parameters
16 exports.Studentgrade=(marks)=>
17 {
18   if (marks>50 && marks <70) return ("B grade")
19   else
20     return ("A grade")
21 }
```

The status bar at the bottom indicates the cursor is at Line 4, Column 2, with 4 spaces, in UTF-8 encoding, using LF line endings, in a JavaScript file.



This screenshot shows the Visual Studio Code editor with the file `Person.js` open. The Explorer sidebar on the left shows the same project structure as the previous screenshot. The main editor area displays the following JavaScript code:

```
1  class student {
2    constructor(name, age, email) {
3      this.name = name;
4      this.age = age;
5      this.email = email;
6    }
7
8    getPersonInfo() {
9      return
10     Name: this.name
11     Age: this.age
12     Email: this.email
13   }
14 }
15
16 module.exports = student;
```

The status bar at the bottom indicates the cursor is at Line 16, Column 30, with 4 spaces, in UTF-8 encoding, using LF line endings, in a JavaScript file.

```

1 // definition of the function EmployeeInfo
2 function EmployeeInfo(name,Salary)
3 {
4   console.log("Wellcome " + name+ " Your monthly Salary is "+ Salary)
5 }
6 var EmpName="John"
7 var EmpSalary= 50000
8 // calling of the function EmployeeInfo
9 EmployeeInfo(EmpName,EmpSalary)
10 //Code for Second Exercise starts from here:
11 const EmpSkills= (skills)=> {
12   console.log("Expert in "+ skills)
13 }
14 EmpSkills("java")
15
16 const student= require('./StudentInfo')
17 const person = require('./Person')
18 // because getName is the function so we use ()
19 console.log("Student Name:" +student.getName())
20 console.log(student.Location())
21 console.log(student.dob)
22 // because dob is a variable so we do nt use ()
23 console.log(student.Studentgrade())
24 console.log("grade is "+student.Studentgrade(55) )
25 // creating new Person
26 person= new person("Jim","USA","myemail@gmail.com")
27 console.log("using Person Module",person1.getPersonInfo())
28 os=require("os")
29 const util=require('util')
30 console.log("temporary directory"+ os.tmpdir() )
31 console.log("hostname: " + os.hostname())
32 console.log("OS : " + os.platform() +"release:"+ os.release())
33 console.log("Uptime"+ (os.uptime())/3600 +" hours")
34 console.log("userInfo" + util.inspect(os.userInfo()))
35 console.log("Memory "+ os.totalmem()/1000000000 + "Giga byte")
36 console.log(" free: "+os.freemem()/1000000000 + "Giga byte")
37 console.log("CPU "+ util.inspect(os.cpus()))
38 console.log("Network"+ util.inspect(os.networkInterfaces()))
39 console.log("programe end"]
40 console.log("Programe ended")

```

Report:

The coding that I've done during this lesson displays the information of the device that is being used. It also displays the information of the student information that has been given. I learned how to link different new methods together.

WEEK 4:

```

{
  "status": true,
  "Status_Code": 200,
  "request": "/GetStudents",
  "request Method": "GET",
  "studentdata": [
    {
      "Student1": {
        "name": "Jonhthon",
        "Age": "33",
        "Qualification": "BSC",
        "Email": "std123@gm.com",
        "id": 1
      }
    },
    {
      "Student2": {
        "name": "David",
        "Age": "23",
        "Qualification": "HNC",
        "Email": "Abc@gm.com",
        "id": 2
      }
    },
    {
      "Student3": {
        "name": "Emily",
        "Age": "25",
        "Qualification": "A-level",
        "Email": "email@gm.com",
        "id": 3
      }
    }
  ]
}

```

```

{"status":true,"Status_Code":200,"request":"/GetStudents","request Method":"GET","studentdata":{"Student1":{"name":"Jonhthon","Age":"33","Qualification":"BSC","Email":"std123@gm.com","id":1},"Student2":{"name":"David","Age":"23","Qualification":"HNC","Email":"Abc@gm.com","id":2},"Student3":{"name":"Emily","Age":"25","Qualification":"A-level","Email":"email@gm.com","id":3}}}

```

localhost:5000/GetStudents

name:	"Jonhthon"
Age:	"33"
Qualification:	"BSC"
Email:	"std123@gm.com"
id:	1

Localhost:5000/ GetStudentid/1

name:	"David"
Age:	"23"
Qualification:	"HNC"
Email:	"Abc@gm.com"
id:	2

Localhost:5000/ GetStudentid/2

name:	"Emily"
Age:	"25"
Qualification:	"A-level"
Email:	"email@gm.com"
id:	3

Localhost:5000/ GetStudentid/3

status:	true
Status_Code:	200
requrl:	"/GetStudentid/4"
request Method:	"GET"
▼ studentdata:	
▼ Student1:	
name:	"Jonhthon"
Age:	"33"
Qualification:	"BSC"
Email:	"std123@gm.com"
id:	1
▼ Student2:	
name:	"David"
Age:	"23"
Qualification:	"HNC"
Email:	"Abc@gm.com"
id:	2
▼ Student3:	
name:	"Emily"
Age:	"25"
Qualification:	"A-level"
Email:	"email@gm.com"
id:	3

Localhost:5000/

GetStudentid/4

Student Details

First Name:

Last Name :

Email:

Age :

Please select your gender:

- ☒ **Male**
☐ **Female**
☐ **Other**

Qualifications

- ☐ **GCSE**
☐ **A- level**
☐ **Higher National Certificate/Level 4**
☒ **Foundation Degree/HND/DipHE/Level 5**
☐ **Bachelor Degree/Graduate diploma or Certificate/Level 6**
☐ **Master Degree/PGCE/Level7**
☐ **PhD/Level8**

```
status: true
message: "form Details"
data:
  name: "jahmarli hibbert "
  age: "21 Gender: male"
  Qualification: " QualificationHND"
```

```
{"status":true,"message":"form Details","data":{"name":"jahmarli hibbert ","age":" Gender: male","Qualification":" QualificationHND"}}
```

Report:

This week I learnt how to implement data that has been provided and display it in a format that separates the data into different categories. The data has also been displayed in different formats.

WEEK 5:

For todays Lab submission, After you complete the lab write down a word document answering following questions for your portfolio:

1. What is React

React is a JavaScript library for building a user interface.

2. What do you understand by React component and what command do you use to create a React component with or without property

A React Component is a piece of UI logic that uses HTML, CSS, and JavaScript code to represent a part of the user interface.

without

```
1 function MyComponent() {
2   return <h1>Hello, World!</h1>;
3 }
4 |
```

with

```
1 function MyComponent(props) {
2   return <h1>{props.heading}</h1>;
3 }
4 |
```

3. What command will you use to render the the newly created component named as myREACT

```
1 import React from 'react';
2 import ReactDOM from 'react-dom/client';
3 import MyREACT from './MyREACT';
4
5 const root = ReactDOM.createRoot(document.getElementById('root'));
6 root.render(<MyREACT />);
```

4. Suppose the MyReact Component has a property heading, write down the code that could be used to render the MYReact Component, and pass the message to the property heading as “this is my first element”

```
1 import React from 'react';
2 import ReactDOM from 'react-dom/client';
3 import MyReact from './MyReact';
4
5 const root = ReactDOM.createRoot(document.getElementById('root'));
6 root.render(<MyReact heading="This is my first element" />);
```

5. Observe this code and answer the questions below

```
<AppColor heading="This is first element" lbl  
="Name :" color="green"/>
```

What is the name of the React Component

[AppColor](#)

How many properties this component uses

[heading](#), [lbl](#), and [color](#).

6. Look at the following Code:

```
function GreetingElementwithProp(props) {  
  return (  
    <div className="App">  
      <h1>Wellcome , {props.studentname}</h1>;  
    </div>  
  );  
}  
export default ??????
```

what will you write to make this export this function correctly?

Hint you need to replace ?????? with the correct word.

Add a function that takes two properties as numbers ,add these numbers on the click event of the button and display the sum.

Hint you will be using in jsx

```
<button value={props.color} onClick={Namdofyourfunc  
tion}
```

```

1  import React, { useState } from 'react';
2
3  function GreetingElementwithProp(props) {
4      const [sum, setSum] = useState(null); // State to store the sum
5
6      // Function to add num1 and num2
7      const handleAddition = () => {
8          const result = props.num1 + props.num2;
9          setSum(result); // Update the sum state
10     };
11
12     return (
13         <div className="App">
14             <h1>Welcome, {props.studentname}</h1>
15             <button value={props.color} onClick={handleAddition}>
16                 Add Numbers
17             </button>
18             {sum !== null && <p>The sum is: {sum}</p>}
19         </div>
20     );
21 }
22
23 export default GreetingElementwithProp;

```

Week 7:

Write one page reflective what did you learn about React Hook API during this week Q2. Study the code in EmojeeCounters.js, Please note, You Do not need to submit the full code rather you need to answer the following questions for your this week portfolio

- What is Name of the Component you have created in EmojeeCounters.js

EmojeeCounter

- Identify the line of code that uses the EmojeeCounter in index.js

`import EmojeeCounter from './EmojiCounter';`

- Declares the states of each of the html elements defined in the EmojeeCounters.js (identify these lines and explain only those lines)

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

- Lines of codes that are used to associate the event handler used.

`<button onClick={ClickHandle}>{count }`

- Explain the line : `<EmojeeCounter pic='Love'/>` , what is `pic='Love'` means in this line.

`This means that the component can access the image called 'love'.`

- What is useEffect and why you think we have used it in the Component.

It can be used for tasks like data fetching.

- Explain these line of the codes in functional component EmojeeCounter.js:

```
return (
  <div className="App">
    <p>{props.pic} <span></span>
    <button onClick={ClickHandle}>{count }
    <img src={pic} alt=""/>
    </button>
  </p>
</div>
)
```

`{props.pic}`: Shows the picture prop that was supplied to the component.

This button's `onClick={ClickHandle}>{count}`: A button element that, when clicked, launches the ClickHandle function and displays the current count.

``: Shows an image in which the source is the pic state or prop

Q3

Create a code for a Component that takes two HTML one text box and one label. Label will be used to display the images. So it should be like this

If I write “Happy” in the text box the label should show happy face (You can use any image)

If I write “Like” in the text box the label should show Like icon

If I write “sad “ the label should show sad emoji.

Run this component take the screen shot of your newly run component and write a

Emoji Display App



paragraph how did you develop this component.

Emoji Display App



Emoji Display App



Using React and `useState` to control the input text state, I created the `EmojiDisplayComponent`. To enable fast lookup, I made an `imageMap` object that links particular words ("happy," "like," and "sad") to image URLs. When the user types, the component updates the state and shows the matching image if the input contains a keyword.

Week 8:

```
removed in the next major version
Connected to mongodb://localhost:27017/Week8
New document has been added into your database: {
  name: 'JACK',
  age: 20,
  Gender: 'Male',
  Salary: 3456,
  _id: new ObjectId('673cc97f616a8a136ff8382a'),
  __v: 0
}
Data inserted successfully!
```

```
_id: ObjectId('673cc8568b623cc87abb31ad')
name: "Simon"
age: 42
Gender: "Male"
Salary: 3456
__v: 0
```

```
_id: ObjectId('673cc8568b623cc87abb31ae')
name: "Neesha"
age: 23
Gender: "Female"
Salary: 1000
__v: 0
```

```
_id: ObjectId('673cc8568b623cc87abb31af')
name: "Mary"
age: 27
Gender: "Female"
Salary: 5402
__v: 0
```

```
_id: ObjectId('673cc8568b623cc87abb31b0')
name: "Mike"
age: 40
Gender: "Male"
Salary: 4519
__v: 0
```

Showing 5 documents with all information:

```
{
  _id: new ObjectId('673cc8568b623cc87abb31ac'),
  name: 'Yousuf',
  age: 44,
  Gender: 'Male',
  Salary: 3456,
  __v: 0
}
{
  _id: new ObjectId('673cc8568b623cc87abb31ad'),
  name: 'Simon',
  age: 42,
  Gender: 'Male',
  Salary: 3456,
  __v: 0
}
{
  _id: new ObjectId('673cc8568b623cc87abb31ae'),
  name: 'Neesha',
  age: 23,
  Gender: 'Female',
  Salary: 1000,
  __v: 0
}
{
  _id: new ObjectId('673cc8568b623cc87abb31af'),
  name: 'Mary',
  age: 27,
  Gender: 'Female',
  Salary: 5402,
  __v: 0
}
{
  _id: new ObjectId('673cc8568b623cc87abb31b0'),
  name: 'Mike',
  age: 40,
  Gender: 'Male',
  Salary: 4519,
  __v: 0
}
```

New document has been added into your database.

Showing unique names with Salary greater than 3000:

JACK

Mary

Mike

Simon

Yousuf

Data inserted successfully!

□

1. mongoose.connect(uri, options)

- **Purpose:** Establishes a connection to the MongoDB database.
- **Parameters:**
 - uri: The MongoDB connection string (e.g., mongodb://localhost:27017/Week8).
 - options: An object specifying additional options, such as:
 - useUnifiedTopology: Enables the new MongoDB driver's unified topology layer for better connection management.
 - useNewUrlParser: Ensures the new URL string parser is used.
- **Returns:** A Promise that resolves when the connection is successful.

2. db.on(event, callback) and db.once(event, callback)

- **Purpose:** Listen for specific events on the database connection object.
- **Events:**
 - error: Triggered when a connection error occurs.
 - connected: Triggered when the database connection is successfully established.
- **Usage:**
 - on: Attaches a listener that runs every time the specified event occurs.
 - once: Attaches a listener that runs only the first time the specified event occurs.

3. mongoose.Schema

- **Purpose:** Defines the structure and constraints of documents in a collection.
- **Parameters:**
 - An object representing the fields (e.g., name, age, Gender, Salary) and their configurations.
- **Example Configurations:**
 - type: Specifies the data type (e.g., String, Number).
 - required: Marks the field as mandatory.
 - default: Specifies a default value for the field.
 - unique: Ensures the field's values are unique across the collection.

4. mongoose.model(name, schema, collection)

- **Purpose:** Creates a model for interacting with the database collection.
- **Parameters:**
 - name: The name of the model (e.g., Person).
 - schema: The schema to apply (e.g., PersonSchema).
 - collection (optional): The name of the MongoDB collection (e.g., personCollection).

5. document.save()

- **Purpose:** Saves a single document (instance of a model) to the database.
- **Returns:** A Promise that resolves with the saved document or rejects with an error.
- **Example Usage:**

6. Model.insertMany(documents)

- **Purpose:** Inserts multiple documents into the database in a single operation.
- **Parameters:**
 - **documents:** An array of objects, where each object represents a document.
- **Returns:** A Promise that resolves when all documents are successfully inserted or rejects with an error.

7. Model.find(query)

- **Purpose:** Retrieves documents from the collection that match the query criteria.
- **Parameters:**
 - **query:** An object defining the conditions for retrieval (e.g., { Gender: "Female", age: { \$gte: 30 } }).
- **Returns:** A Query object that can be further refined (e.g., sorted, limited).

8. Query Modifiers

Modifiers are chained to refine the results of a query. Here are the ones used in your script:

- **.sort(fields):**
 - Sorts the results based on the specified fields.
 - Example: { Salary: 1 } sorts by Salary in ascending order; { Salary: -1 } for descending.
- **.select(fields):**
 - Specifies the fields to include or exclude in the result.
 - Example: 'name Salary age' includes only name, Salary, and age.
- **.limit(number):**
 - Limits the number of documents returned.
 - Example: 10 restricts the output to 10 documents.
- **.exec():**
 - Executes the query and returns a Promise.
 - Commonly used to handle asynchronous results.

9. .then() and .catch()

- **Purpose:** Handle the resolved or rejected state of Promises.
- **Usage:**
 - **.then(callback):** Runs the callback function if the Promise resolves successfully.
 - **.catch(callback):** Runs the callback function if the Promise is rejected (error occurs).

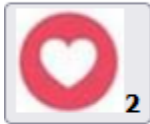
WEEK 9:

U2283556

It is 5 Like.



It is 2 Love.



This is output of Task 2: sad



<https://github.com/Marli1741/WorkSchool>

Functional components are simpler and more concise than class components. They use the `useState` hook to manage state, eliminating the need for constructors and `this.setState()`. This makes them easier to understand and maintain. Additionally, hooks like `useEffect` allow functional components to handle side effects, making them more versatile. For these reasons, functional components are often the preferred choice for new React projects.

Week 10:

U2283556

POST

http://localhost:5003/updatebook/674f42cbc9a088b1d87461af

Send

Params

Authorization

Headers (9)

Body

Scripts

Settings

Cookies

☐ none

☐ form-data

☐ x-www-form-urlencoded

☒ raw

☐ binary

☐ GraphQL

JSON

Beautify

```
1 {
2   "booktitle": "Mice and Men",
3   "PubYear": 1925,
4   "author": "F. Scott Fitzgerald",
5   "Topic": "Fiction",
6   "formate": "Hardcover"
7 }
8
```

Body

Cookies

Headers (7)

Test Results

200 OK

73 ms

272 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "books": "book updated successfully"
3 }
```

POST

http://localhost:5003/addbooks

Params

Authorization

Headers (9)

Body

Scripts

Settings

☐ none

☐ form-data

☐ x-www-form-urlencoded

☒ raw

☐ binary

☐ GraphQL

```
1 {
2   "booktitle": "darkness",
3   "PubYear": 1958,
4   "author": "Gregg",
5   "Topic": "horror",
6   "format": "web novel"
7 }
```

Body

Cookies

Headers (7)

Test Results

Pretty

Raw

Preview

Visualize

JSON

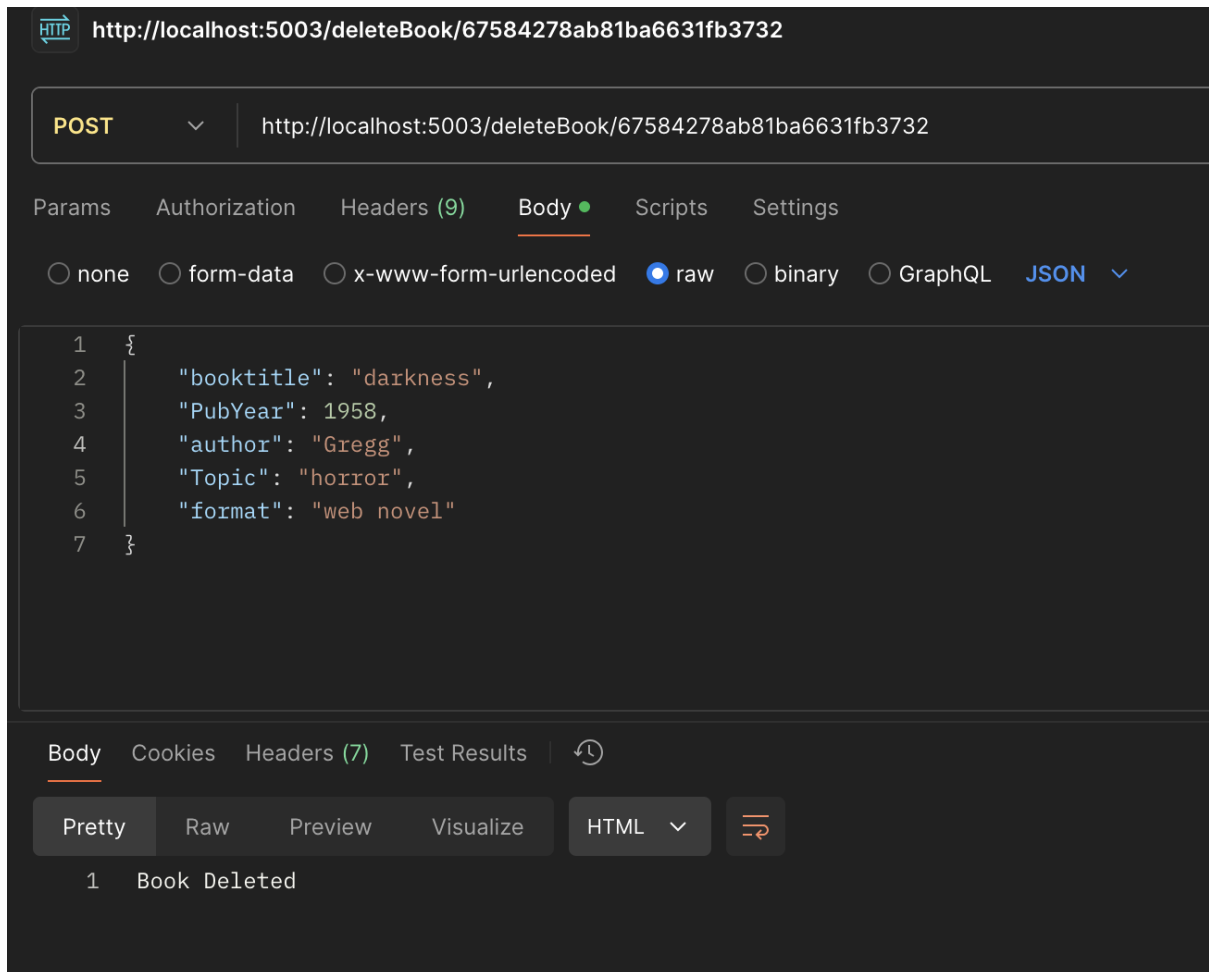
```
1 {
2   "books": "book added successfully"
3 }
```

localhost:5003/allbooks

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```
▼ 0:
  _id: "674f42cbc9a088b1d87461af"
  booktitle: "Mice and Men"
  PubYear: 1925
  author: "F. Scott Fitzgerald"
  Topic: "Fiction"
  __v: 0
▼ 1:
  _id: "67584278ab81ba6631fb3732"
  booktitle: "darkness"
  PubYear: 1958
  author: "Gregg"
  Topic: "horror"
  format: "web novel"
  __v: 0
```



This coursework's backend is a Node.js application using Express.js for routing and MongoDB as the database. It follows RESTful principles for easy integration with the frontend.

The backend has five main API endpoints:

- **GET /books:** Retrieves all books using the find query.
- **GET /books/:id:** Fetches a specific book by ID using the findOne query.
- **POST /addbooks:** Adds a new book to the database using the insertOne query.
- **PUT /books/:id:** Updates an existing book's details using the updateOne query.
- **DELETE /books/:id:** Removes a book using the deleteOne query.

The backend supports all basic CRUD operations (Create, Read, Update, Delete) and includes input validation middleware. It follows a modular MVC architecture, making it easy to maintain and scale. This design provides a strong foundation for future additions like user authentication or more book-related features.

<https://github.com/Marli1741/WorkSchool>

Week 11:

Add Book Display Books

Add Book

Book Title:

Mice and Men

Author:

Gregg

Topic:

Fiction

Format:

☐ Hard Copy ☒ Electronic Copy

Publication Year: 1920

Add Book

I built the front-end of this project using React.js. This framework's component-based structure and efficient state management helped me create a user-friendly interface. To connect with the backend, I used Axios to send HTTP requests to the REST API. For example, I sent form data as JSON to the /addbooks endpoint to add books to the MongoDB database and fetched book data from the /books endpoint to populate the interface. I managed form inputs and application state with React's useState hook. I also implemented error handling with .then() and .catch() blocks to provide feedback to the user during operations like adding or updating books.

I planned to create five main components: a form for adding books, a list to display all books, a detailed view for individual books, a form for editing book details, and a function for deleting books.

Working on this project improved my skills in React and Axios. I learned communication between the front-end and backend in building interactive web applications.

<https://github.com/Marli1741/WorkSchool>