Criterion C: Development

Techniques used to develop the solution

- **Microsoft MySQL Database**
  - Database Connection
  - Database Queries
    - Select
    - Insert
    - Update
    - Delete

- **Object Oriented Programming**
  - Classes and Objects
  - Inheritance
  - Overriding

- **Sending Emails**

- **Creating a Dynamic Calendar**

- **Creating a notepad**

- **Algorithmic thinking**
  - Arrays
  - Linked List
  - Hashing passwords

- **Validation in forms**
  - Implicit validation
  - Validation upon submission

- **Error Handling**

- **Styling conventions and Interactive GUI**

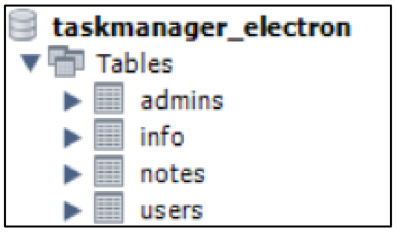
Word count: 1208 words

Technique 1: SQL Database

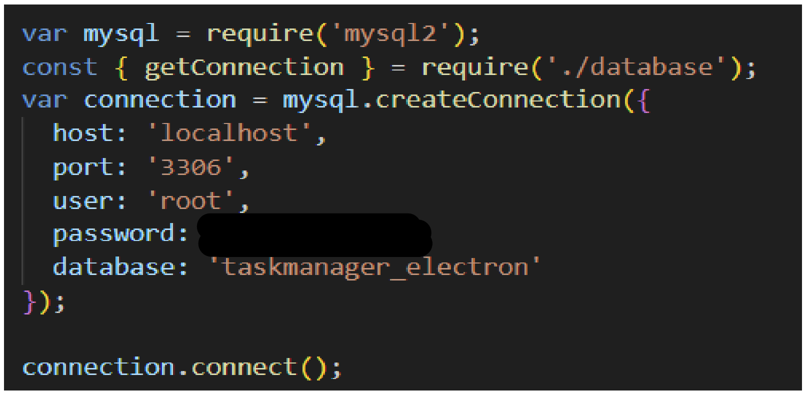
Database Name: `taskmanager_electron`

Database Tables:

- Users
- Admins
- Notes
- Info



1. **Database Connection**



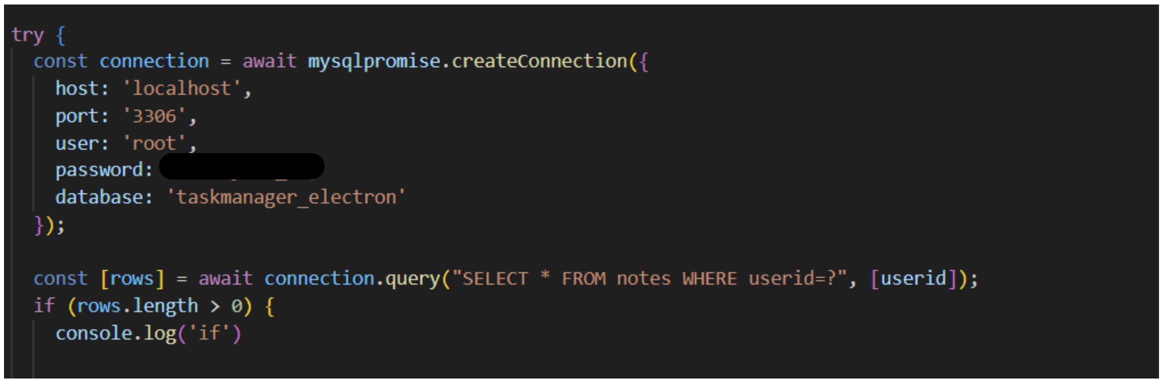
```
var mysql = require('mysql2');
const { getConnection } = require('./database');
var connection = mysql.createConnection({
  host: 'localhost',
  port: '3306',
  user: 'root',
  password: [redacted]
  database: 'taskmanager_electron'
});

connection.connect();
```

This statement uses the `MySQL` library to connect the application to the database. It also loads and executes the queries that are part of database.js.

2. **Database Queries**

   Ingenuity can be seen from the complicated working of each type of command triggered by different buttons is hidden from the user, boosting their experience with the app.

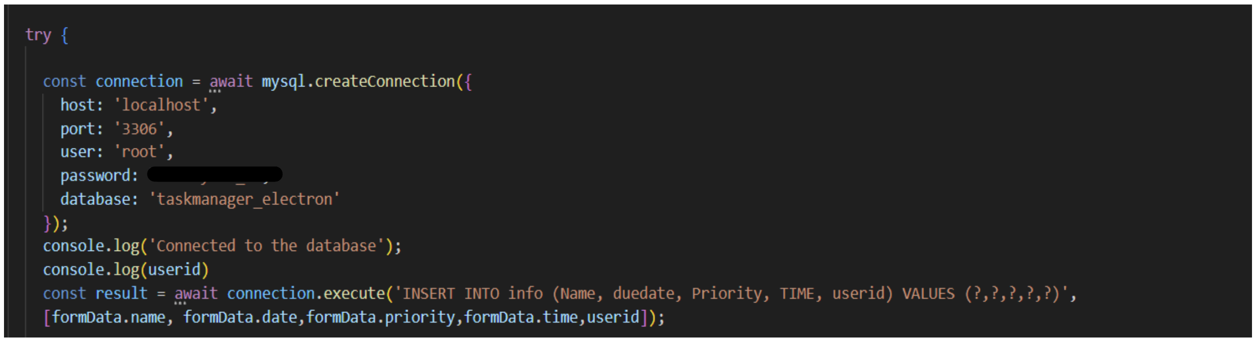
   a. **Select**



```
try {
  const connection = await mysqlpromise.createConnection({
    host: 'localhost',
    port: '3306',
    user: 'root',
    password: [redacted]
    database: 'taskmanager_electron'
  });

  const [rows] = await connection.query("SELECT * FROM notes WHERE userid=?", [userid]);
  if (rows.length > 0) {
    console.log('if')
```

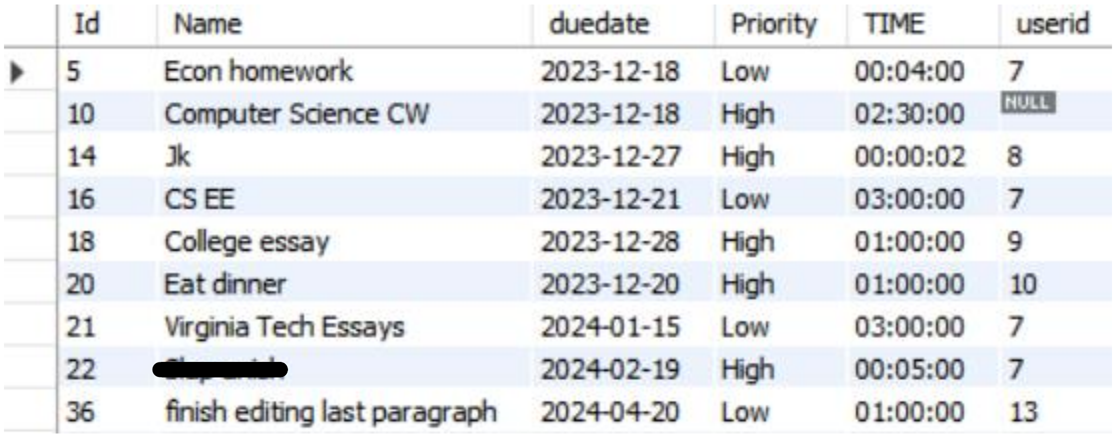
This query retrieves the note from the database using the user ID of the current user, allowing only a specific note to be retrieved. This logic is also used to retrieve tasks and log-in info from other tables throughout the project.

b. Insert



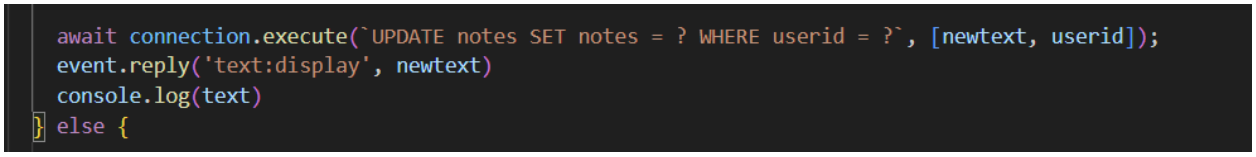
```
try {

  const connection = await mysql.createConnection({
    host: 'localhost',
    port: '3306',
    user: 'root',
    password: [redacted]
    database: 'taskmanager_electron'
  });
  console.log('Connected to the database');
  console.log(userid)
  const result = await connection.execute('INSERT INTO info (Name, duedate, Priority, TIME, userid) VALUES (?,?,?,?,?)',
  [formData.name, formData.date,formData.priority,formData.time,userid]);
```

This query enters the relevant details of each task in the info table. The user ID is later used to retrieve specific tasks based on the previously input user ID.



Id	Name	duedate	Priority	TIME	userid
5	Econ homework	2023-12-18	Low	00:04:00	7
10	Computer Science CW	2023-12-18	High	02:30:00	NULL
14	Jk	2023-12-27	High	00:00:02	8
16	CS EE	2023-12-21	Low	03:00:00	7
18	College essay	2023-12-28	High	01:00:00	9
20	Eat dinner	2023-12-20	High	01:00:00	10
21	Virginia Tech Essays	2024-01-15	Low	03:00:00	7
22	[redacted]	2024-02-19	High	00:05:00	7
36	finish editing last paragraph	2024-04-20	Low	01:00:00	13

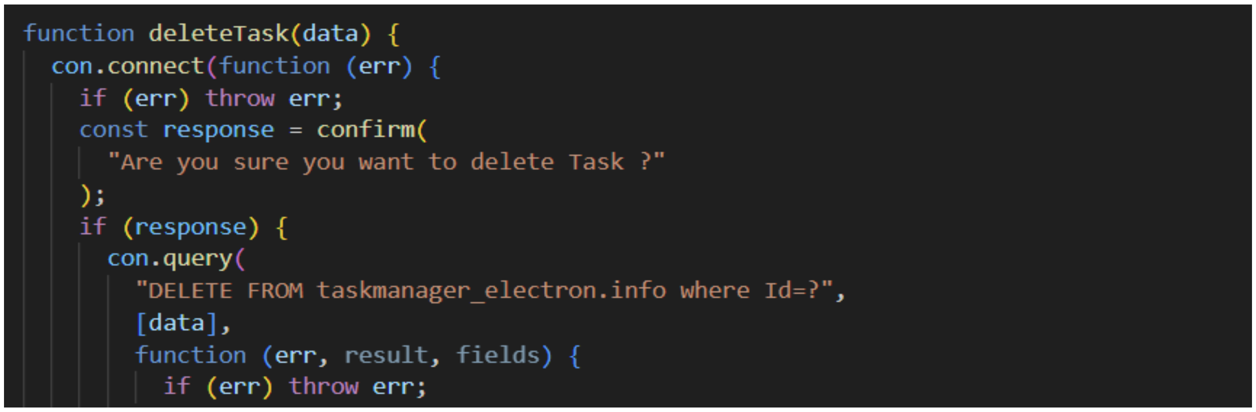
c. Update



```
  await connection.execute(`UPDATE notes SET notes = ? WHERE userid = ?`, [newtext, userid]);
  event.reply('text:display', newtext)
  console.log(text)
} else {
```

This query is part of a function that is initialized using a variable ‘`newText`’ that has the updated text in the notepad. It re-connects the app to the database and changes the values in its notes field.

d. **Delete**

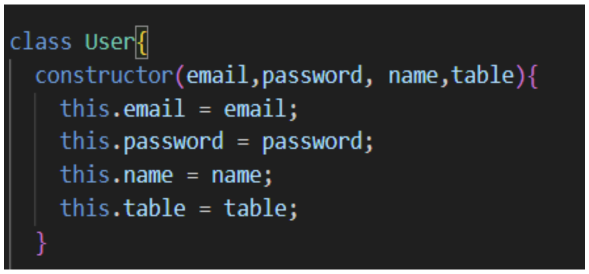


```
function deleteTask(data) {
  con.connect(function (err) {
    if (err) throw err;
    const response = confirm(
      "Are you sure you want to delete Task ?"
    );
    if (response) {
      con.query(
        "DELETE FROM taskmanager_electron.info where Id=?",
        [data],
        function (err, result, fields) {
          if (err) throw err;
```

This query deletes all the information of a task from the table ‘`info`’ using a pre-determined task ID. The task ID is retrieved when a button is clicked, activating the `deleteTask(data)` function with data being the task ID.

Technique 2: Object Oriented Programming

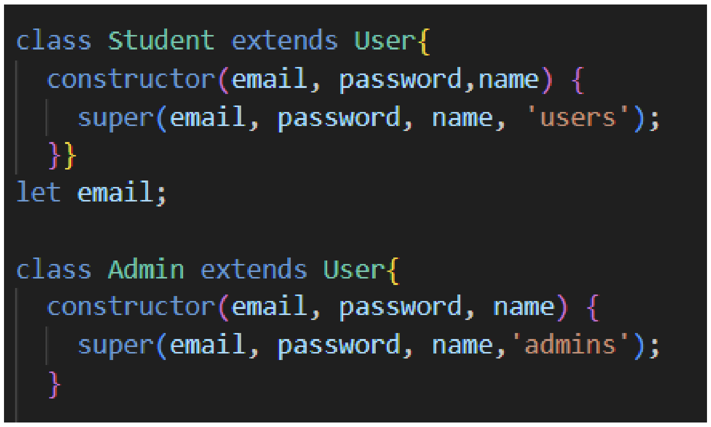
1. **Classes**



```
class User{
  constructor(email,password, name,table){
    this.email = email;
    this.password = password;
    this.name = name;
    this.table = table;
  }
```

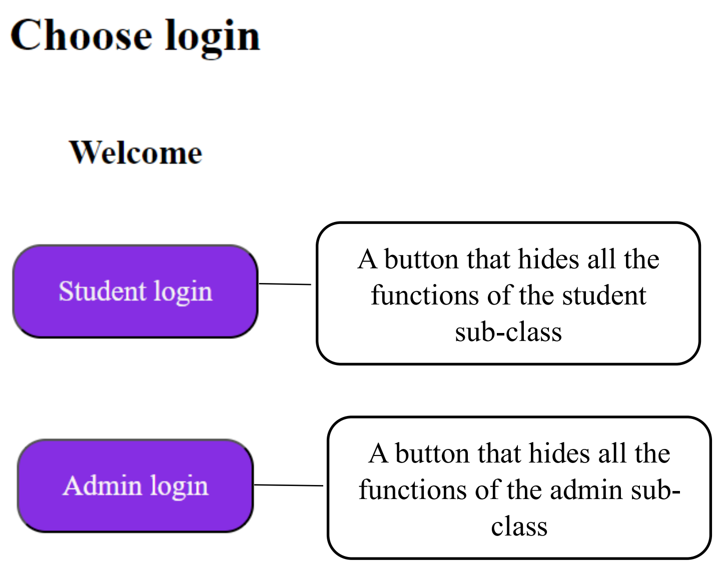
This class is used to handle the login of users into the app. Each user class generically has the above attributes as they correspond to fields in the database table and will facilitate smooth input of values into it. To do this, the table the user belongs to is also an attribute of the class. Ingenuity was employed by selecting OOPs due to its ability to logically organize the code. OOPs also allowed the complicated login in processes to become more modular. This made development and maintenance easier, boosting the app’s extensibility, especially with handling the complex login processes of various types of users.

2. Inheritance

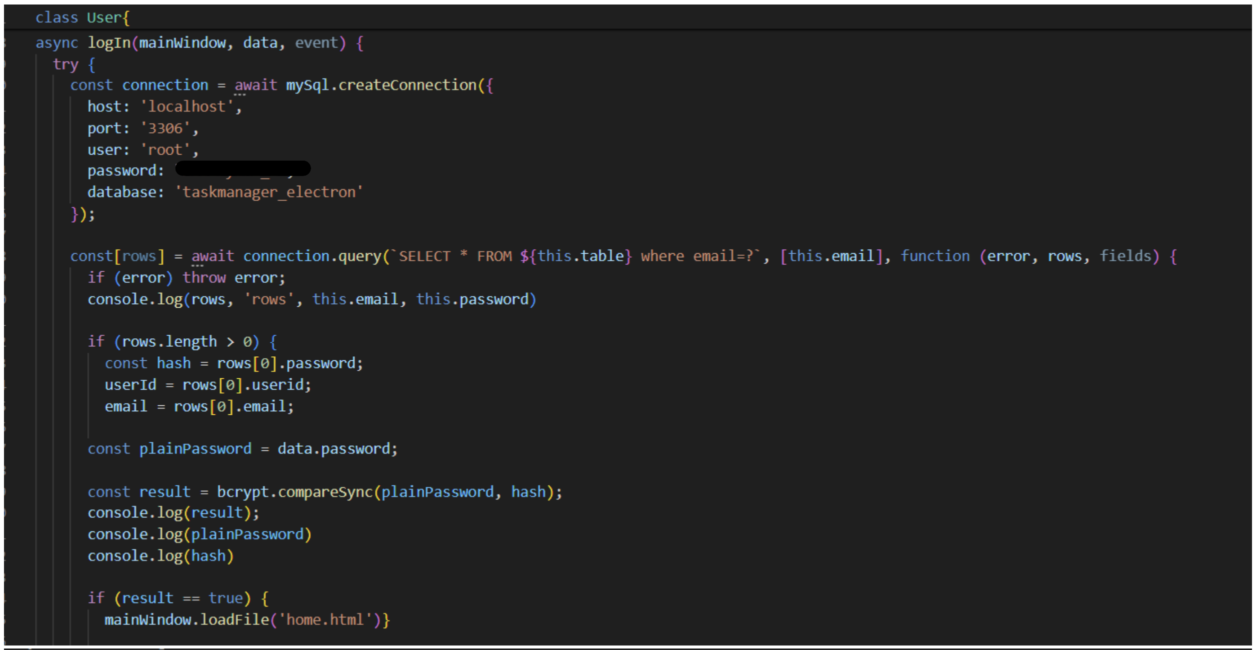


```
class Student extends User{
  constructor(email, password,name) {
    super(email, password, name, 'users');
  }}
let email;

class Admin extends User{
  constructor(email, password, name) {
    super(email, password, name,'admins');
  }
```

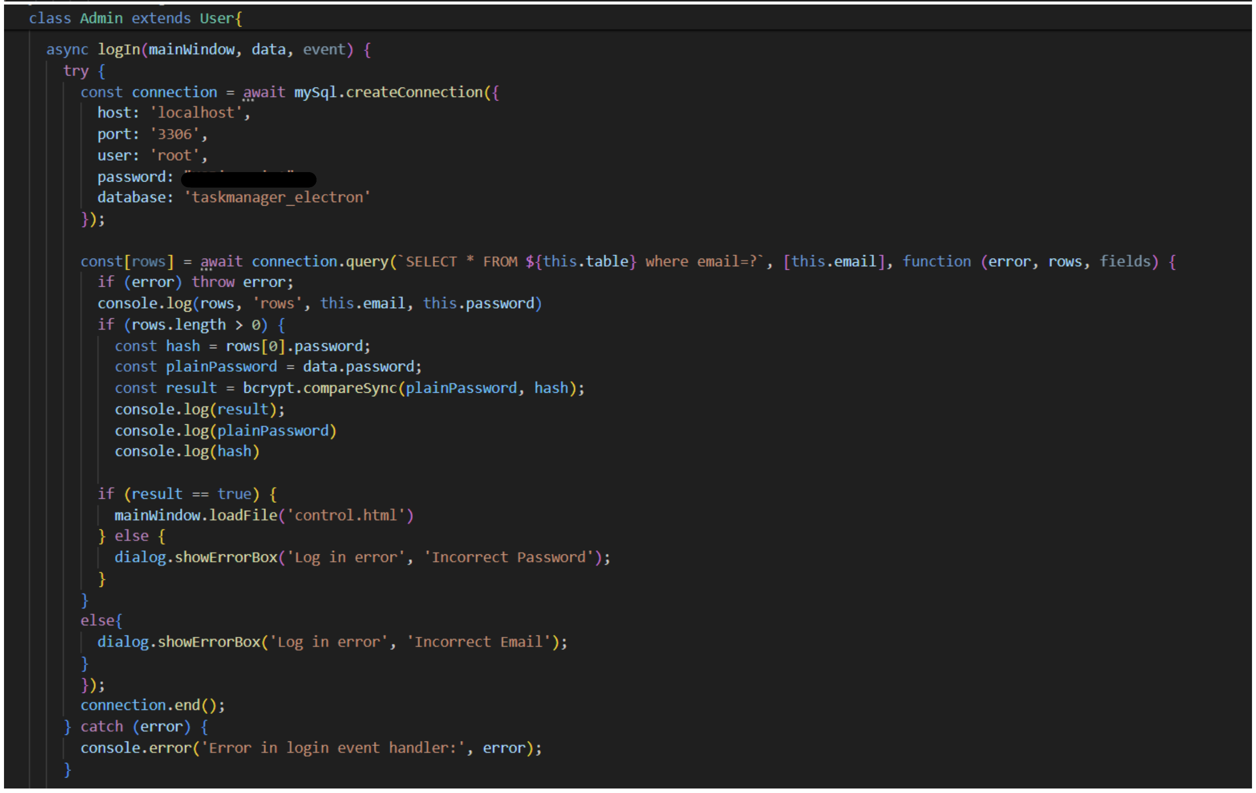
The student and admin class inherits from the user superclass, this allows it to use all the methods without re-coding them and build on them for the specific needs of that class. This process allows for abstraction, which made debugging and programming easier. Both types of users are initiated as users with predefined tables, to make database connection easier.


Choose login
Welcome
Student login
A button that hides all the functions of the student sub-class
Admin login
A button that hides all the functions of the admin sub-class

3. Polymorphism (overriding)



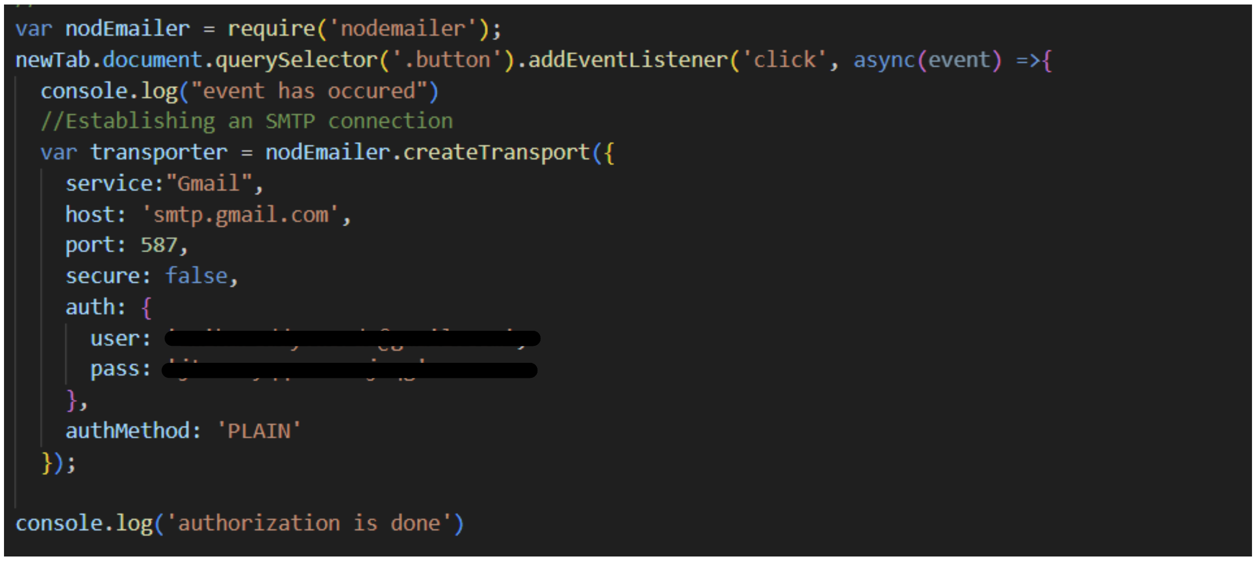
```
class User{
async logIn(mainWindow, data, event) {
  try {
    const connection = await mySql.createConnection({
      host: 'localhost',
      port: '3306',
      user: 'root',
      password: [redacted]
      database: 'taskmanager_electron'
    });

    const[rows] = await connection.query(`SELECT * FROM ${this.table} where email=?`, [this.email], function (error, rows, fields) {
      if (error) throw error;
      console.log(rows, 'rows', this.email, this.password)

      if (rows.length > 0) {
        const hash = rows[0].password;
        userId = rows[0].userid;
        email = rows[0].email;

      const plainPassword = data.password;

      const result = bcrypt.compareSync(plainPassword, hash);
      console.log(result);
      console.log(plainPassword)
      console.log(hash)

      if (result == true) {
        mainWindow.loadFile('home.html')}
```



```
class Admin extends User{

  async logIn(mainWindow, data, event) {
    try {
      const connection = await mySql.createConnection({
        host: 'localhost',
        port: '3306',
        user: 'root',
        password: [redacted]
        database: 'taskmanager_electron'
      });

      const[rows] = await connection.query(`SELECT * FROM ${this.table} where email=?`, [this.email], function (error, rows, fields) {
        if (error) throw error;
        console.log(rows, 'rows', this.email, this.password)
        if (rows.length > 0) {
          const hash = rows[0].password;
          const plainPassword = data.password;
          const result = bcrypt.compareSync(plainPassword, hash);
          console.log(result);
          console.log(plainPassword)
          console.log(hash)

        if (result == true) {
          mainWindow.loadFile('control.html')
        } else {
          dialog.showErrorBox('Log in error', 'Incorrect Password');
        }
      }
      else{
        dialog.showErrorBox('Log in error', 'Incorrect Email');
      }
      });
      connection.end();
    } catch (error) {
      console.error('Error in login event handler:', error);
    }
```

As depicted by the image, the admin subclass overrides the login function of the superclass and modifies it to open the `control.html` document instead of the `home.html` document that is opened by the login method in the superclass. Overriding was also used for creating a new admin as the process is different for students and admins.

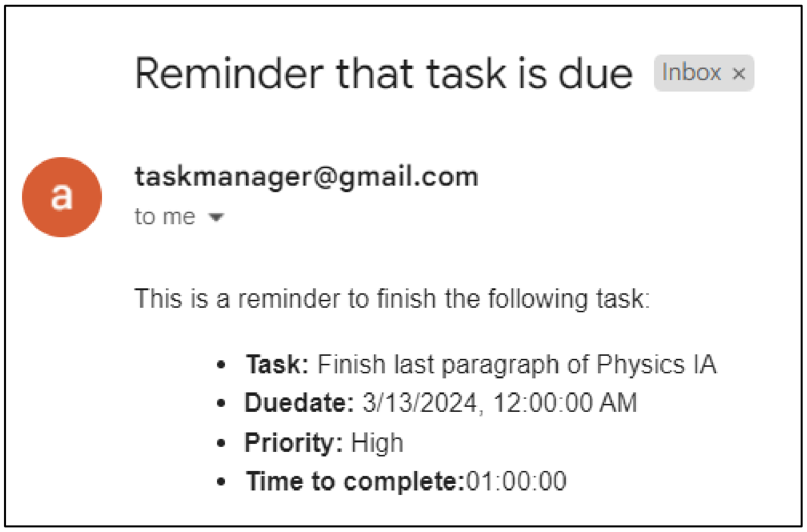
Technique 3: Sending Emails



```
var nodEmailer = require('nodemailer');
newTab.document.querySelector('.button').addEventListener('click', async(event) =>{
  console.log("event has occured")
  //Establishing an SMTP connection
  var transporter = nodEmailer.createTransport({
    service:"Gmail",
    host: 'smtp.gmail.com',
    port: 587,
    secure: false,
    auth: {
      user: [redacted]
      pass: [redacted]
    },
    authMethod: 'PLAIN'
  });

console.log('authorization is done')
```

The library: `nodeemailer` was used to send reminder emails. As seen above, it uses the service Gmail to send emails using the SMTP protocol. Retrieving the recipient email needed the use of a global variable, email. These were required due to the complexity and redundancy that come with providing each user with customized functioning as required by the client.

Once the email is retrieved, an HTML-formatted email is sent:



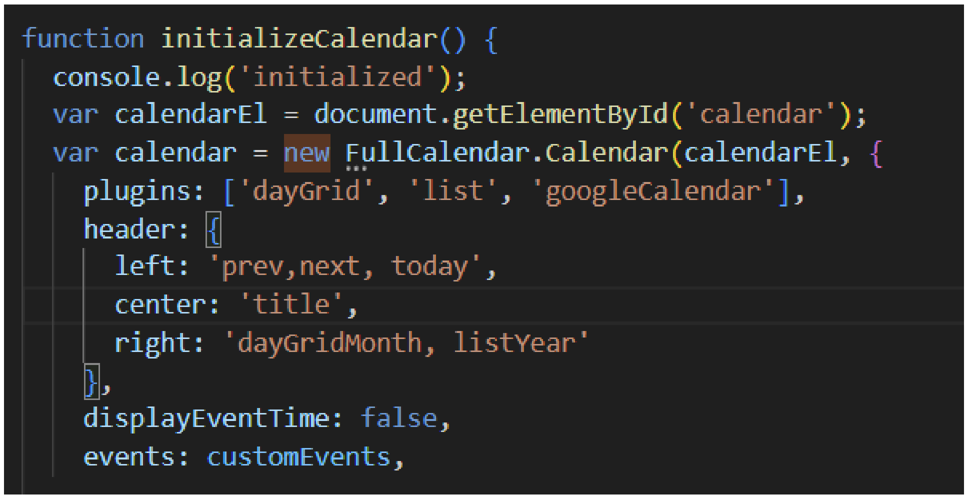
Reminder that task is due Inbox ×

taskmanager@gmail.com
to me

This is a reminder to finish the following task:

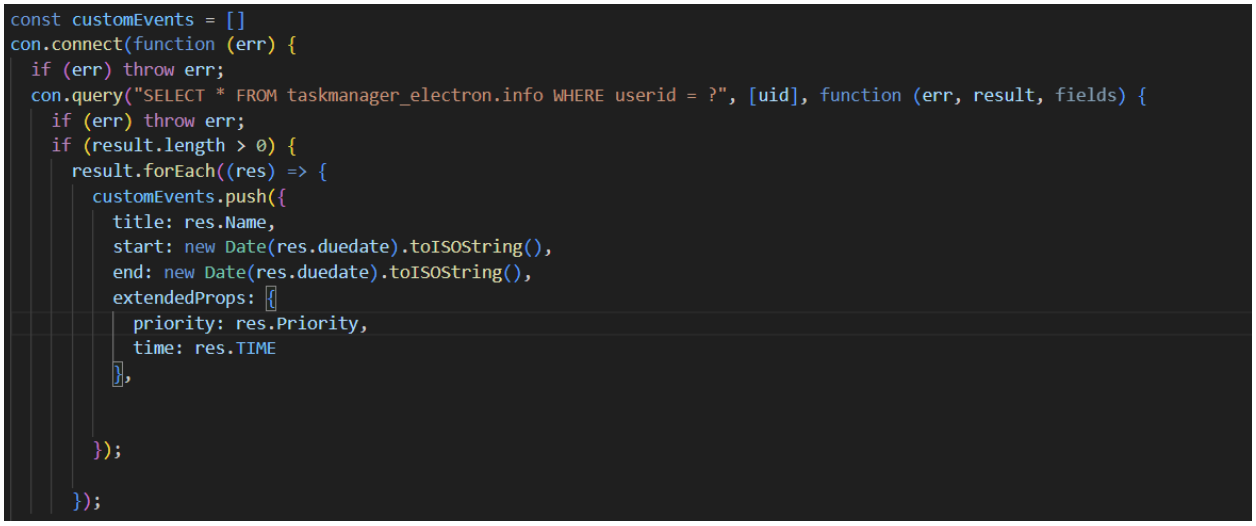
- **Task:** Finish last paragraph of Physics IA
- **Duedate:** 3/13/2024, 12:00:00 AM
- **Priority:** High
- **Time to complete:**01:00:00

Technique 4: Creating a Dynamic Calendar



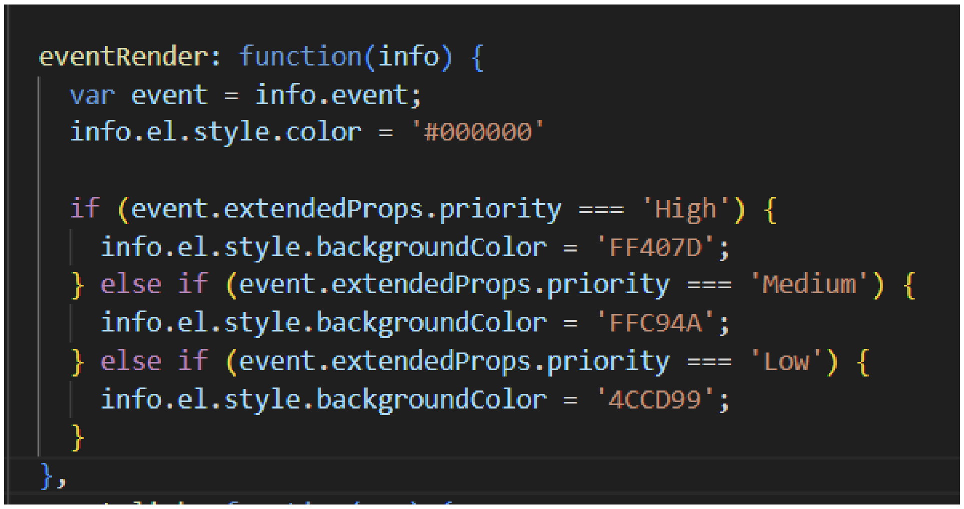
```
function initializeCalendar() {
  console.log('initialized');
  var calendarEl = document.getElementById('calendar');
  var calendar = new FullCalendar.Calendar(calendarEl, {
    plugins: ['dayGrid', 'list', 'googleCalendar'],
    header: {
      left: 'prev,next, today',
      center: 'title',
      right: 'dayGridMonth, listYear'
    },
    displayEventTime: false,
    events: customEvents,
```

The `FullCalendar` library was used to create a dynamic Calendar using its Calendar object. The above function displays the specific inputs given to the object in its constructor to suit this application's needs.



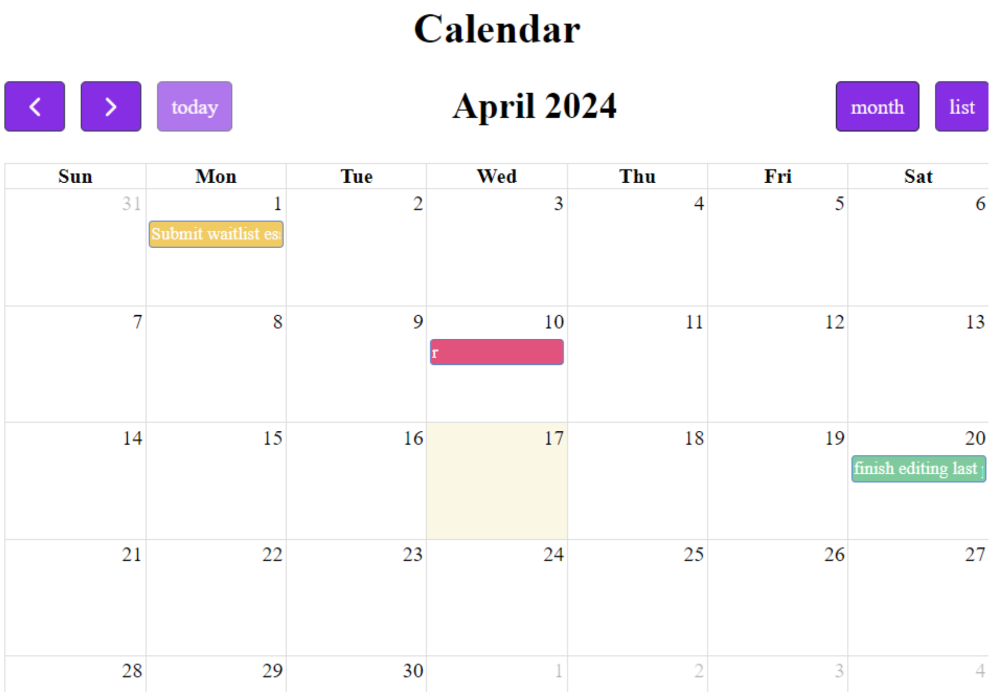
```
const customEvents = []
con.connect(function (err) {
  if (err) throw err;
  con.query("SELECT * FROM taskmanager_electron.info WHERE userid = ?", [uid], function (err, result, fields) {
    if (err) throw err;
    if (result.length > 0) {
      result.forEach((res) => {
        customEvents.push({
          title: res.Name,
          start: new Date(res.duedate).toISOString(),
          end: new Date(res.duedate).toISOString(),
          extendedProps: {
            priority: res.Priority,
            time: res.TIME
          },


        });

      });
```

Custom Events were created by assigning each event to an instance of data in the database.



```
eventRender: function(info) {
  var event = info.event;
  info.el.style.color = '#000000'

  if (event.extendedProps.priority === 'High') {
    info.el.style.backgroundColor = 'FF407D';
  } else if (event.extendedProps.priority === 'Medium') {
    info.el.style.backgroundColor = 'FFC94A';
  } else if (event.extendedProps.priority === 'Low') {
    info.el.style.backgroundColor = '4CCD99';
  }
},
```

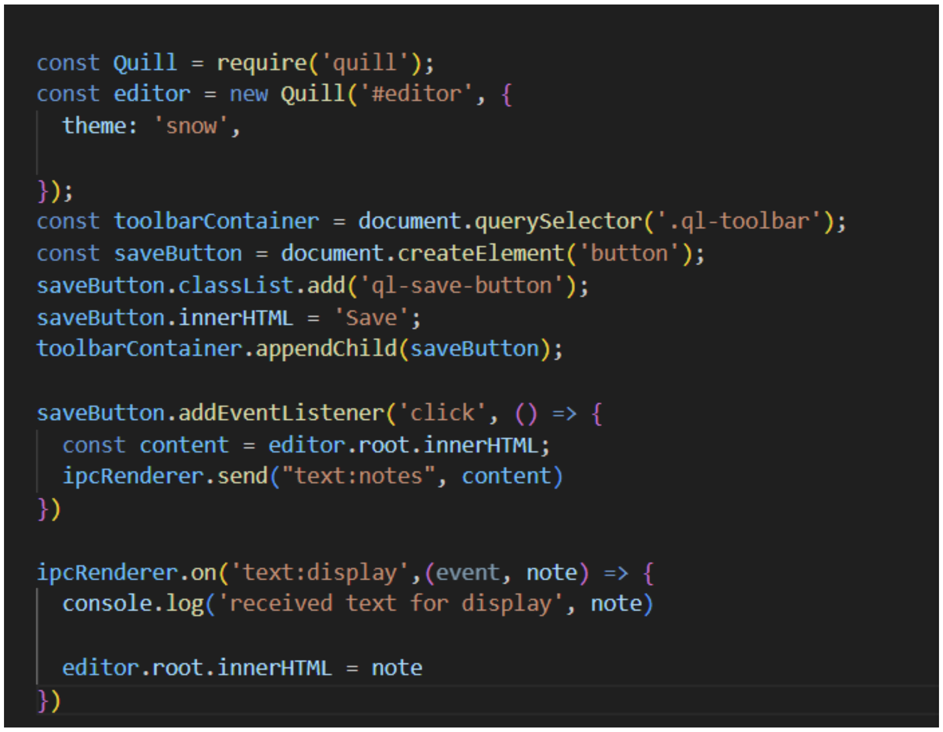
Using the `eventRender` function of the library, I created an extensible function. It was imperative to the client that the color of the tasks was ordered according to priority and looked appealing to students. Hence, a function that uses a ladder if conditions to set different types of events to different colors was created. This enhanced the intuitiveness of the GUI, allowing darker colors of high-priority tasks to be easily associated with the urgency of the task.



April 2024

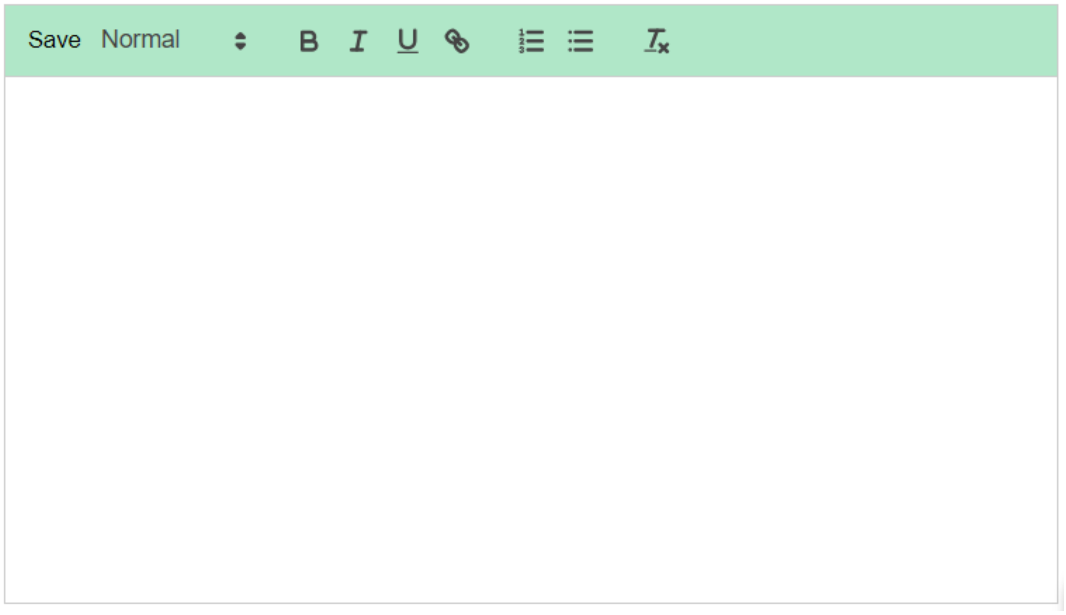
Sun	Mon	Tue	Wed	Thu	Fri	Sat
31	1 Submit waitlist es	2	3	4	5	6
7	8	9	10 r	11	12	13
14	15	16	17	18	19	20 finish editing last
21	22	23	24	25	26	27
28	29	30	1	2	3	4

Technique 5: Creating a notepad



```
const Quill = require('quill');
const editor = new Quill('#editor', {
  theme: 'snow',

});
const toolbarContainer = document.querySelector('.ql-toolbar');
const saveButton = document.createElement('button');
saveButton.classList.add('ql-save-button');
saveButton.innerHTML = 'Save';
toolbarContainer.appendChild(saveButton);

saveButton.addEventListener('click', () => {
  const content = editor.root.innerHTML;
  ipcRenderer.send("text:notes", content)
})

ipcRenderer.on('text:display',(event, note) => {
  console.log('received text for display', note)

  editor.root.innerHTML = note
})
```

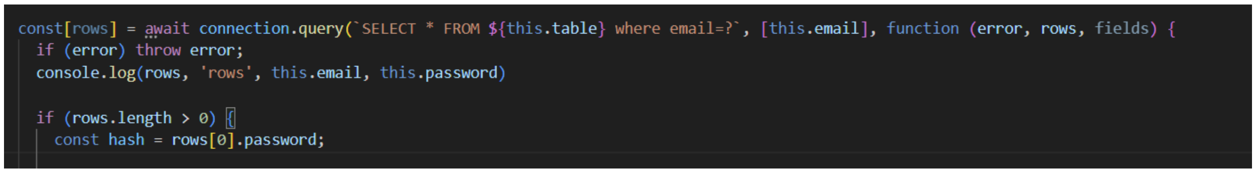
The `quill` library was used to create a functioning notepad that had the basic functionality of a Google Doc, and the toolbar was adapted to create a custom save button on the toolbar that saves and updates notes in the MySQL database as illustrated in the first technique.



The use of the library boosts extensibility as more options can be added to the toolbar using the append function depending on the client's needs.

Technique 6: Algorithmic thinking

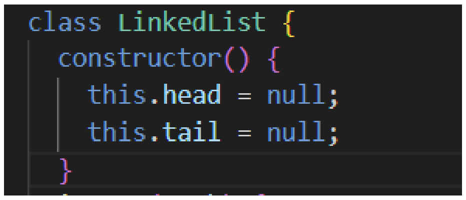
1. **Arrays**



```
const[rows] = await connection.query(`SELECT * FROM ${this.table} where email=?`, [this.email], function (error, rows, fields) {
  if (error) throw error;
  console.log(rows, 'rows', this.email, this.password)

  if (rows.length > 0) {
    const hash = rows[0].password;
```

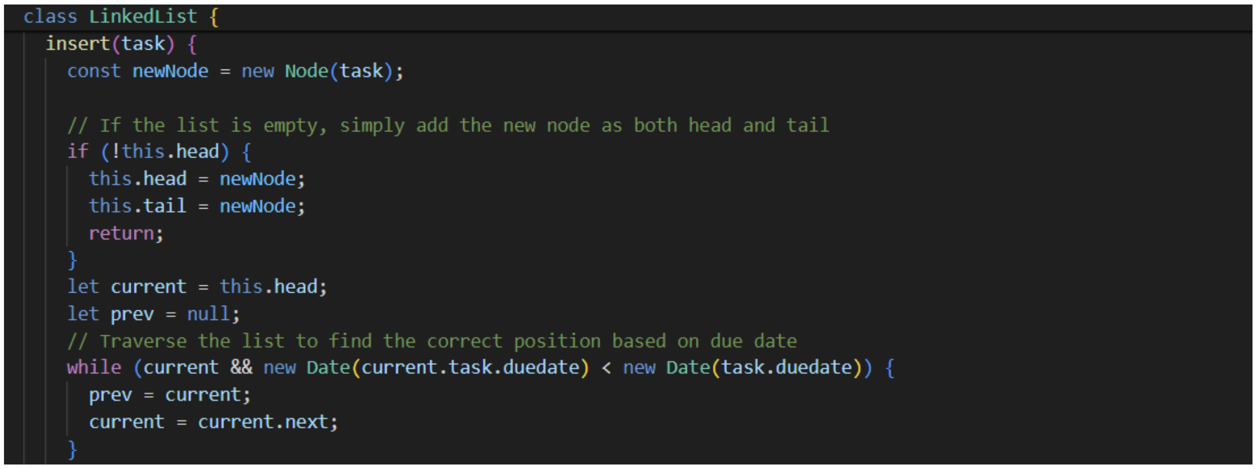
Arrays were used when recovering data from the database for the app to use. In this instance, verify whether a user has entered the right password for login, the database is searched using the email id. If the email id matches, the password will be input into the `rows` array. The if condition is only executed if the array's length attribute indicates that the database has an email entry that matches that inputted by the user.

2. **Linked Lists**



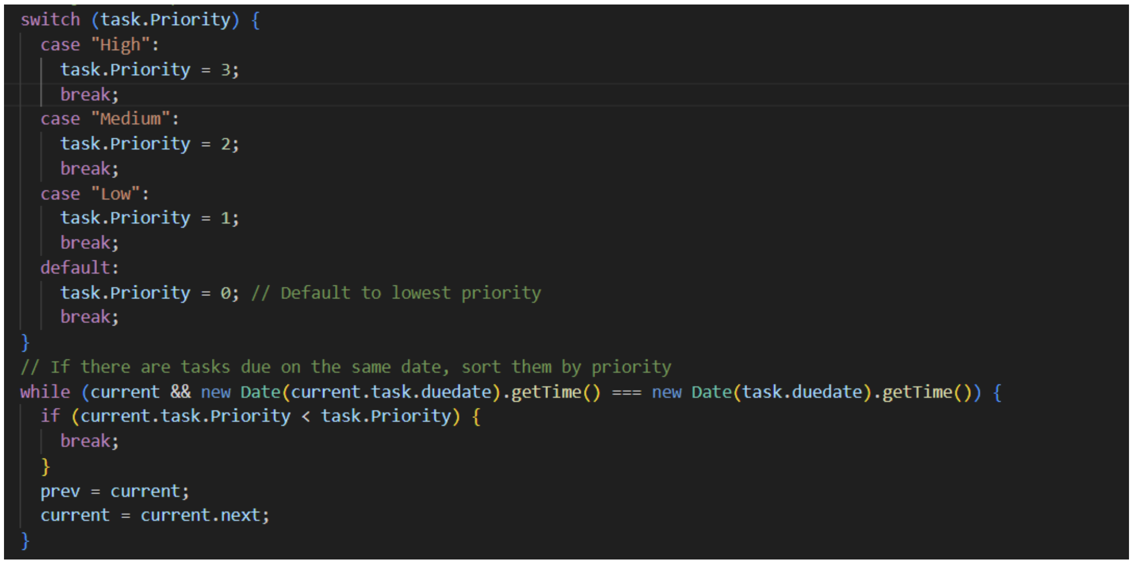
```
class LinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
  }
```

The `LinkedList` class was used to order the tasks after they were retrieved from the database.



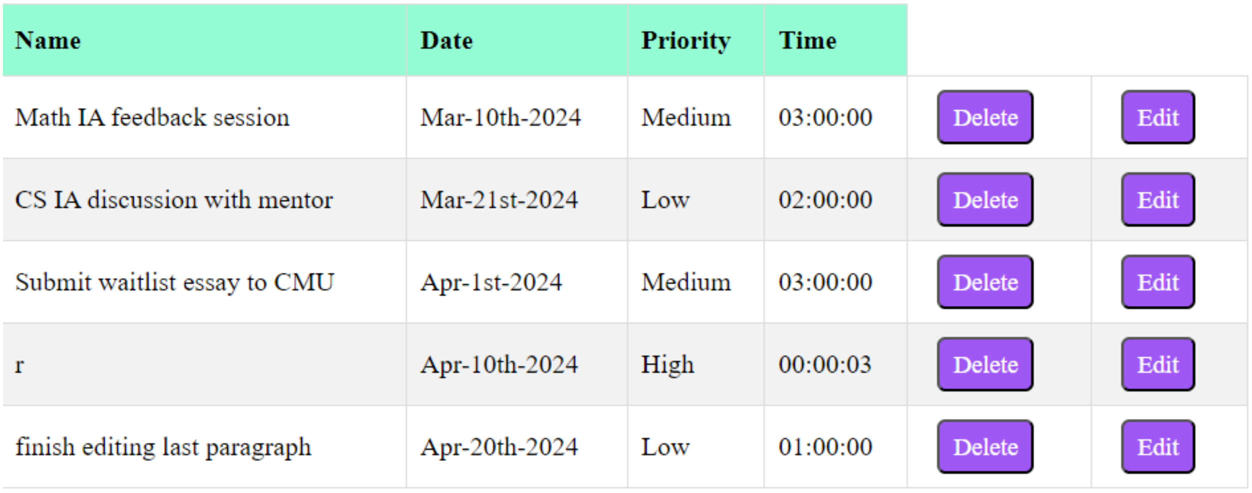
```
class LinkedList {
  insert(task) {
    const newNode = new Node(task);

    // If the list is empty, simply add the new node as both head and tail
    if (!this.head) {
      this.head = newNode;
      this.tail = newNode;
      return;
    }
    let current = this.head;
    let prev = null;
    // Traverse the list to find the correct position based on due date
    while (current && new Date(current.task.duedate) < new Date(task.duedate)) {
      prev = current;
      current = current.next;
    }
```

The original function `insert(task)` was created to incorporate complex algorithmic thinking so that nodes can be easily inserted and ordered in the LinkedList according to the closest due date and highest priority.



```
switch (task.Priority) {
  case "High":
    task.Priority = 3;
    break;
  case "Medium":
    task.Priority = 2;
    break;
  case "Low":
    task.Priority = 1;
    break;
  default:
    task.Priority = 0; // Default to lowest priority
    break;
}
// If there are tasks due on the same date, sort them by priority
while (current && new Date(current.task.duedate).getTime() === new Date(task.duedate).getTime()) {
  if (current.task.Priority < task.Priority) {
    break;
  }
  prev = current;
  current = current.next;
}
```

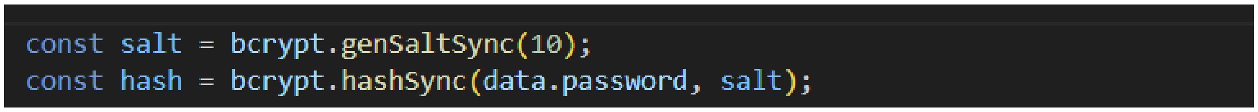
Ingenuity was required to convert the priorities into a number format to be compared by the algorithm. It uses these comparisons to shift tasks with the same due date to a position based on its priority. Another function is used after the swapping to convert the numbers back to strings.



Name	Date	Priority	Time		
Math IA feedback session	Mar-10th-2024	Medium	03:00:00	Delete	Edit
CS IA discussion with mentor	Mar-21st-2024	Low	02:00:00	Delete	Edit
Submit waitlist essay to CMU	Apr-1st-2024	Medium	03:00:00	Delete	Edit
r	Apr-10th-2024	High	00:00:03	Delete	Edit
finish editing last paragraph	Apr-20th-2024	Low	01:00:00	Delete	Edit

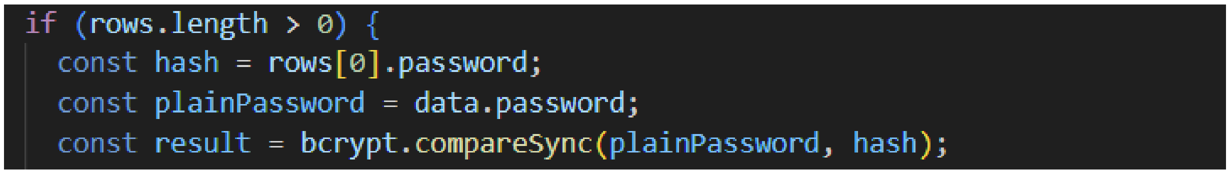
3. Hashing Passwords

To fulfill the need for security, hashing passwords seemed to provide the optimal security required for such a platform.



```
const salt = bcrypt.genSaltSync(10);
const hash = bcrypt.hashSync(data.password, salt);
```

The `bycrypt` library is used to generate a salt using a hashing algorithm over 10 rounds. The `hashSync` command hashes the password provided by the user during their sign-in process according to the salt. The salts ensure that even if two users have the same password, their hashed passwords are different.

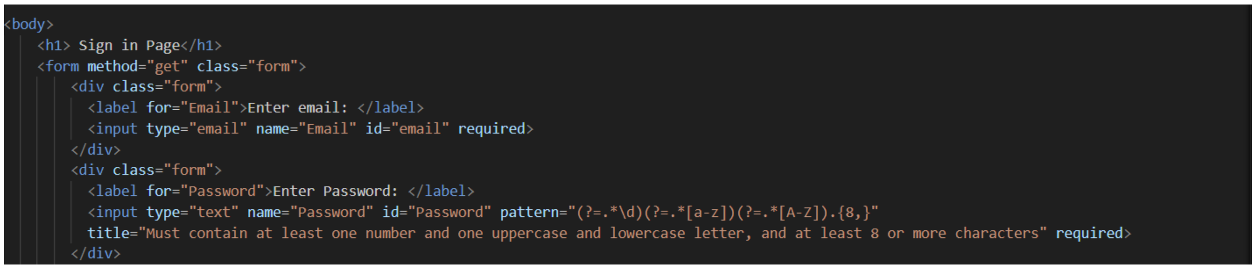


```
if (rows.length > 0) {
  const hash = rows[0].password;
  const plainPassword = data.password;
  const result = bcrypt.compareSync(plainPassword, hash);
```

When a user logs in, the hashed password is retrieved from the database. Using the `bycrypt` library's `compareSync` function, the password entered by the user is hashed and compared with the existing password in the database. If the results match, the user is allowed to log in.

Technique 7: Validation in forms

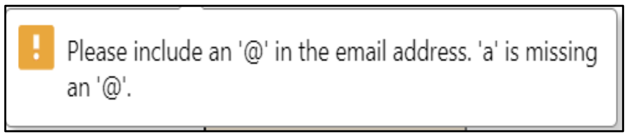
- **Implicit Validation**

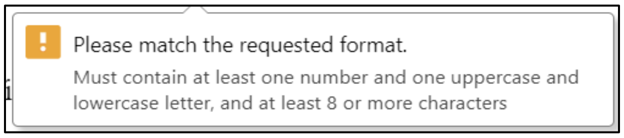


```
<body>
    <h1> Sign in Page</h1>
    <form method="get" class="form">
        <div class="form">
          <label for="Email">Enter email: </label>
          <input type="email" name="Email" id="email" required>
        </div>
        <div class="form">
          <label for="Password">Enter Password: </label>
          <input type="text" name="Password" id="Password" pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}"
          title="Must contain at least one number and one uppercase and lowercase letter, and at least 8 or more characters" required>
        </div>
```

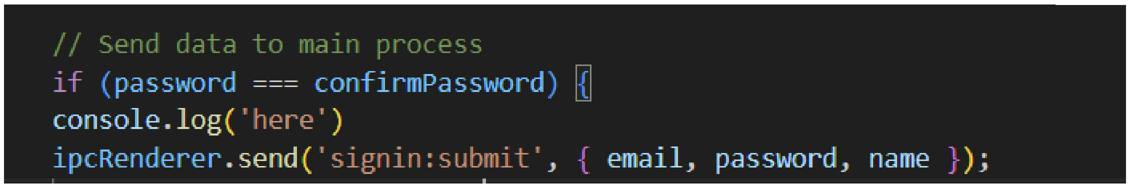
This validates the input that the user enters the sign-in field. It checks if the password is strong, and whether the email has an @ in it.

If not, the user is prompted to re-enter their information.


Please include an '@' in the email address. 'a' is missing an '@'.


Please match the requested format.
Must contain at least one number and one uppercase and lowercase letter, and at least 8 or more characters

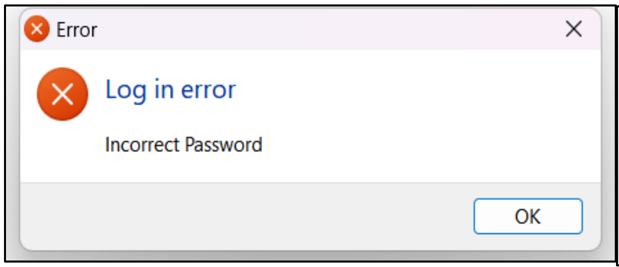
Whether the password field matches the confirm password field is also checked.


```
// Send data to main process
if (password === confirmPassword) {
console.log('here')
ipcRenderer.send('signin:submit', { email, password, name });
```

Only if all these conditions are met will the data be saved and input into the database.

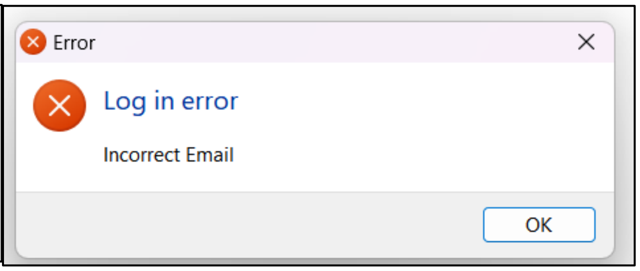
- **Validation upon submission**

If the user inputted login details do not match those in the database, ingenuity is used to provide a different dialogue box for each type of issue, using the API provided by Electron.


Error

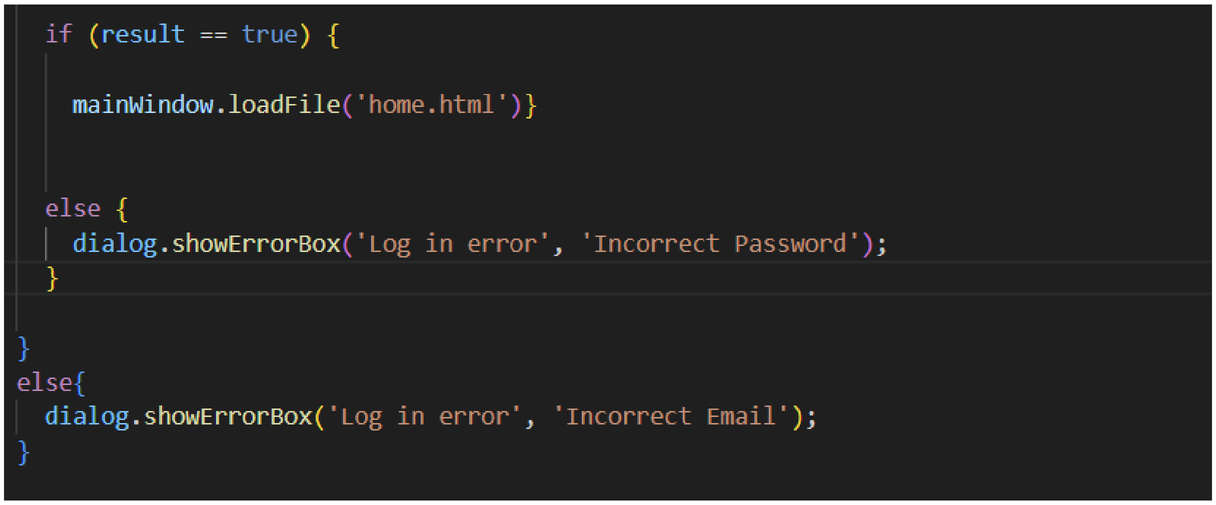
Log in error

Incorrect Password

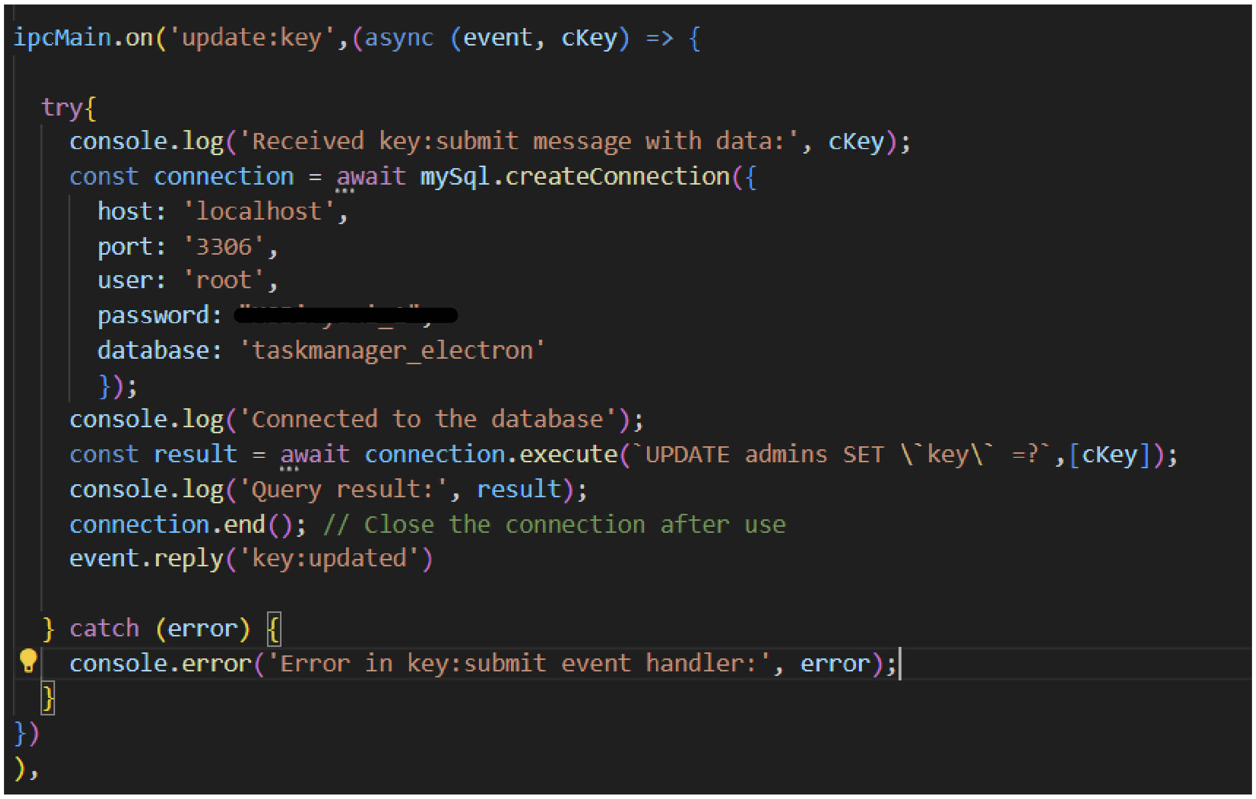

Error

Log in error

Incorrect Email


```
  if (result == true) {

    mainWindow.loadFile('home.html')}


  else {
    dialog.showErrorBox('Log in error', 'Incorrect Password');
  }

}
else{
  dialog.showErrorBox('Log in error', 'Incorrect Email');
}
```

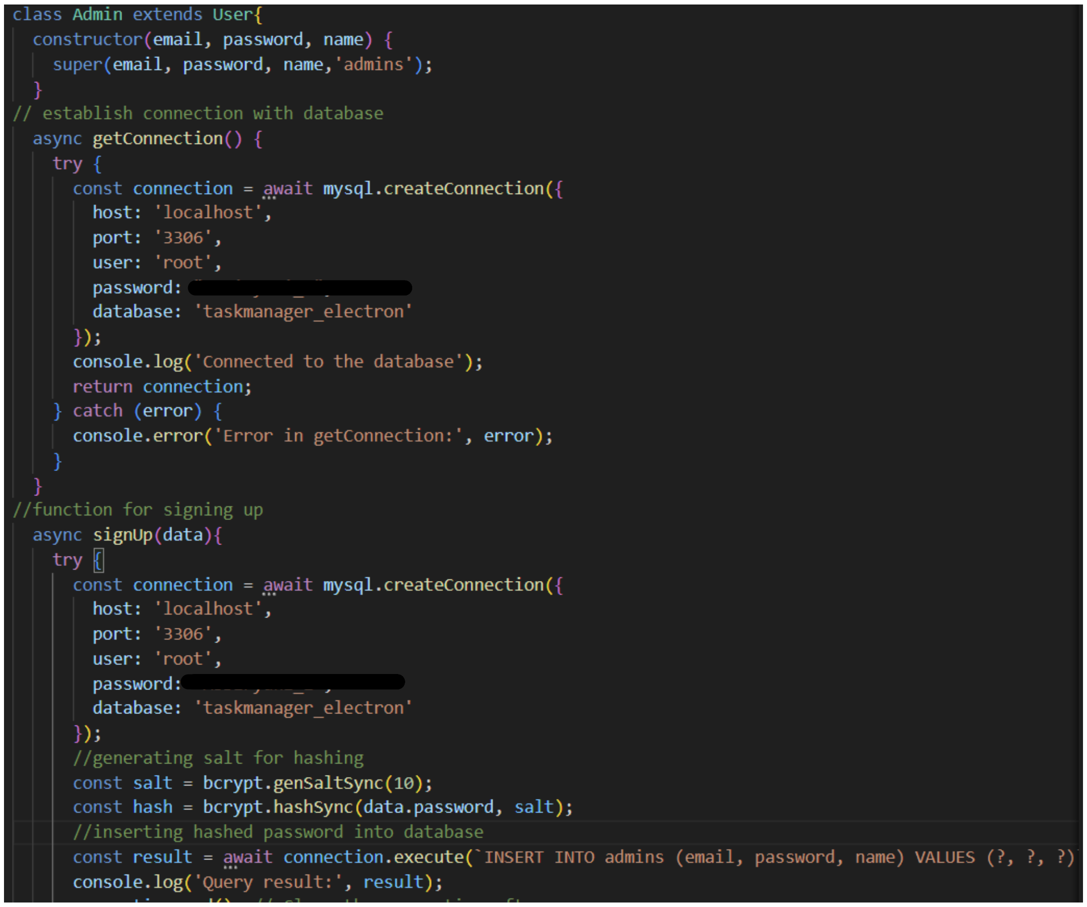
Technique 8: Error handling



```
ipcMain.on('update:key',(async (event, cKey) => {

  try{
    console.log('Received key:submit message with data:', cKey);
    const connection = await mySql.createConnection({
      host: 'localhost',
      port: '3306',
      user: 'root',
      password: [redacted]
      database: 'taskmanager_electron'
      });
    console.log('Connected to the database');
    const result = await connection.execute(`UPDATE admins SET \`key\` =?`,[cKey]);
    console.log('Query result:', result);
    connection.end(); // Close the connection after use
    event.reply('key:updated')

  } catch (error) {
    console.error('Error in key:submit event handler:', error);
  }
})
),
```

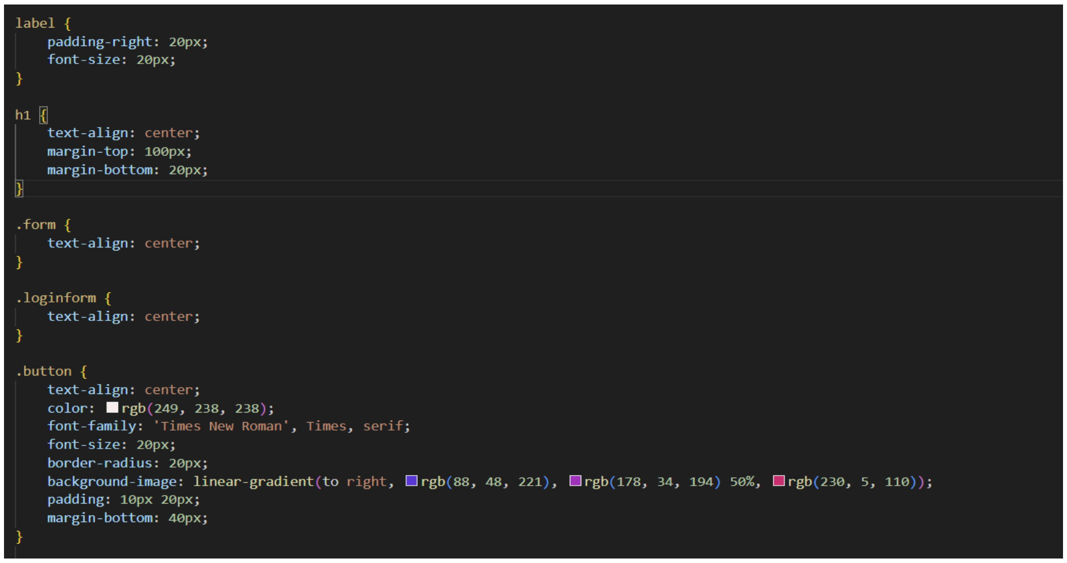
To ensure that all connections to the database are carried out without any errors, try-and-catch statements are used to debug the app by revealing the error occurring during attempts of connection. In this specific case, it is to update the ket when the admin wishes to in the database.

Technique 9: Styling conventions and Interactive GUI



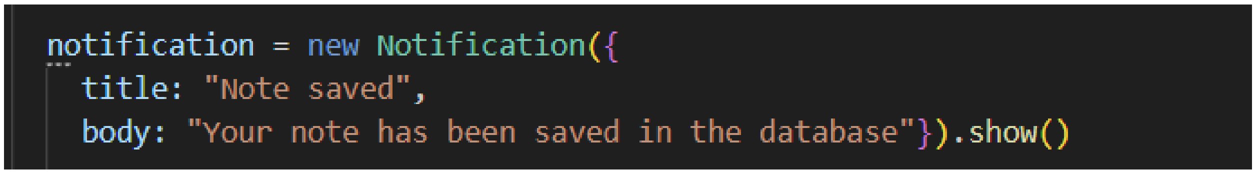
```
class Admin extends User{
  constructor(email, password, name) {
    super(email, password, name,'admins');
  }
// establish connection with database
  async getConnection() {
    try {
      const connection = await mysql.createConnection({
        host: 'localhost',
        port: '3306',
        user: 'root',
        password: [redacted]
        database: 'taskmanager_electron'
      });
      console.log('Connected to the database');
      return connection;
    } catch (error) {
      console.error('Error in getConnection:', error);
    }
  }
//function for signing up
  async signUp(data){
    try {
      const connection = await mysql.createConnection({
        host: 'localhost',
        port: '3306',
        user: 'root',
        password:[redacted]
        database: 'taskmanager_electron'
      });
      //generating salt for hashing
      const salt = bcrypt.genSaltSync(10);
      const hash = bcrypt.hashSync(data.password, salt);
      //inserting hashed password into database
      const result = await connection.execute(`INSERT INTO admins (email, password, name) VALUES (?, ?, ?)
      console.log('Query result:', result);
```

For the backend, conventions were used to allow future developers to comprehend the code properly. camelCase was used for functions and variable names PascalCase was used for classes.



```css
label {
    padding-right: 20px;
    font-size: 20px;
}

h1 {
    text-align: center;
    margin-top: 100px;
    margin-bottom: 20px;
}

.form {
    text-align: center;
}

.loginform {
    text-align: center;
}

.button {
    text-align: center;
    color: rgb(249, 238, 238);
    font-family: 'Times New Roman', Times, serif;
    font-size: 20px;
    border-radius: 20px;
    background-image: linear-gradient(to right, rgb(88, 48, 221), rgb(178, 34, 194) 50%, rgb(230, 5, 110));
    padding: 10px 20px;
    margin-bottom: 40px;
}
```

For the front end, CSS was used to create a customizable GUI, as it was easily implementable with electron, and allowed the introduction of the specific color scheme agreed upon with the client. To ensure the extensibility of the product, programming conventions were maintained in both HTML and CSS documents.



```javascript
notification = new Notification({
  title: "Note saved",
  body: "Your note has been saved in the database"}).show()
```

GUI elements such as Notifications use the inbuilt notification API and class so that users can be notified that the complex process of saving ate in the database has been completed

Bibliography

Database connections:

Node.js MySQL Insert Into. (n.d.). Www.w3schools.com. Retrieved December 12, 2023. https://www.w3schools.com/nodejs/nodejs_mysql_insert.asp

Node.js MySQL Delete. (n.d.). Www.w3schools.com. Retrieved December 12, 2023. https://www.w3schools.com/nodejs/nodejs_mysql_delete.asp

Node.js MySQL Update. (n.d.). Www.w3schools.com. Retrieved December 12, 2023. https://www.w3schools.com/nodejs/nodejs_mysql_update.asp

Sending emails

Shotola, F. (2023, July 3). *Understanding Nodemailer: A Thorough Guide to Email Sending with Node.js*. Medium. Retrieved December 30, 2023. https://medium.com/@femishotolaa/understanding-nodemailer-a-thorough-guide-for-email-sending-with-node-js-ebb45417c64d

Dynamic Calendar

Getting Started - Docs | FullCalendar. (n.d.). Fullcalendar.io. Retrieved December 20, 2023. https://fullcalendar.io/docs/getting-started

Linked Lists:

Southard, A. (2017, July 24). *Linked List in JavaScript*. Medium. Retrieved December 25, 2023. https://medium.com/@andrewsouthard1/linked-list-in-javascript-634fc2e0897b

Hashing Passwords:

bcryptjs. (n.d.). Npm. Retrieved December 10, 2023.

https://www.npmjs.com/package/bcryptjs