1. **Build a responsive web application for shopping cart with registration, login, catalog and cart pages using CSS3 features, flex and grid.**

ALGORITHM:

Open any Text Editor or Notepad

Write the code and save it as file.html.

Define the document type and language.

Set up the page head and metadata.

Create the body layout.

Product Catalog: Where products are displayed.

Shopping Cart: Where added items and the total price are displayed.

Style the Page with CSS

Set the body to use a flexible layout with display: flex; to center content.

Add background color, text color, and padding for the header.

Use Flexbox to arrange the products in a responsive way, making sure they wrap when necessary.

Add borders and padding to make the items visually distinct.

Create an empty cart array.

Create a function to add items to the cart.

When a product button is clicked, the addToCart function adds the product's name and price to the cartItems array and updates the total price.

This function clears the old cart list and updates the displayed items and total price every time the cart is modified.

Create a function to update the cart display.

Add Product List and Cart Display

Clicking "Add to Cart" adds products to the cart.

Cart is updated dynamically with item names, prices, and the total price.

Clicking "Add to Cart" will add the product to the shopping cart and show the updated list and total.

PROGRAM:

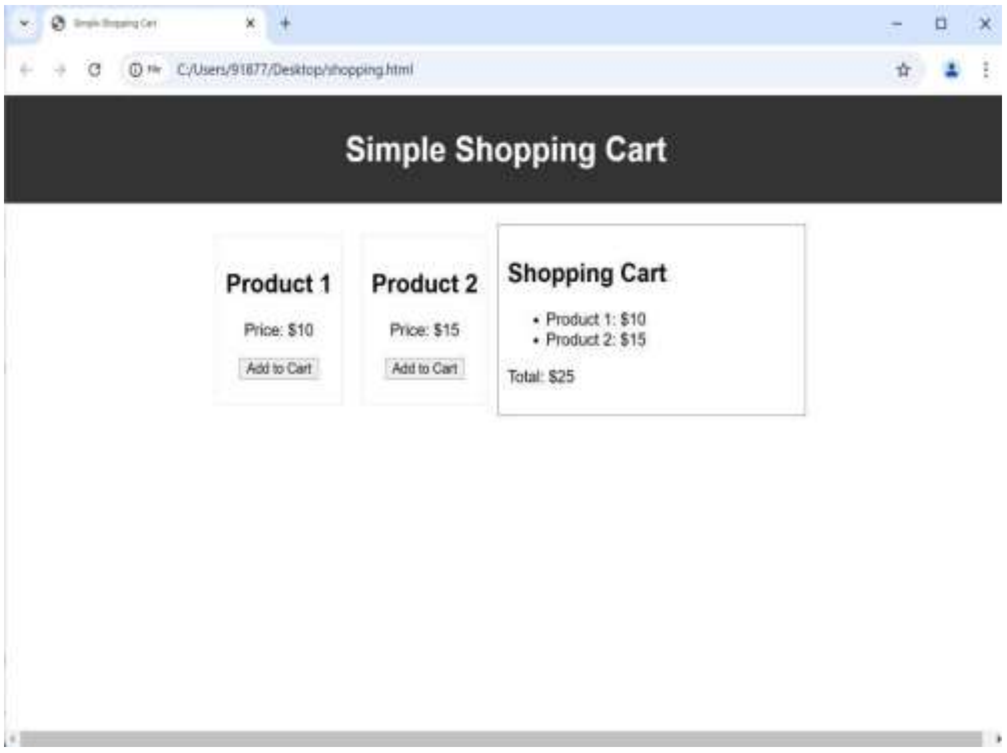
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Simple Shopping Cart</title>
<style>
 /* Basic CSS with Flexbox for layout */
 body {
  font-family: Arial, sans-serif;
  margin: 0;
  padding: 0;
  display: flex;
  flex-direction: column;
  align-items: center;
 }

 header {
  background-color: #333;
  color: #fff;
  padding: 10px;
  width: 100%;
  text-align: center;
 }

 .container {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-around;
  padding: 20px;
  max-width: 800px;
```

```
    }

    .product {
      border: 1px solid #ddd;
      padding: 10px;
      margin: 10px;
      text-align: center;
    }

    .cart {
      border: 1px solid #333;
      padding: 10px;
      width: 300px;
    }
  </style>
</head>
<body>
  <header>
    <h1>Simple Shopping Cart</h1>
  </header>

  <div class="container">
    <!-- Product Catalog -->
    <div class="product">
      <h2>Product 1</h2>
      <p>Price: $10</p>
      <button onclick="addToCart('Product 1', 10)">Add to Cart</button>
    </div>
    <div class="product">
      <h2>Product 2</h2>
      <p>Price: $15</p>
      <button onclick="addToCart('Product 2', 15)">Add to Cart</button>
    </div>

    <!-- Shopping Cart -->
    <div class="cart">
      <h2>Shopping Cart</h2>
      <ul id="cart-items"></ul>
      <p>Total: $<span id="cart-total">0</span></p>
    </div>
  </div>

  <script>
    // JavaScript for handling the shopping cart
    let cartItems = [];
    let cartTotal = 0;

    // Function to add items to the cart
    function addToCart(productName, price) {
      // Add the item to the cart array
      cartItems.push({ productName, price });

      // Update the total price
      cartTotal += price;
```

```
      // Update the cart display
      updateCartDisplay();
    }

    // Function to update the cart display
    function updateCartDisplay() {
      const cartItemsElement = document.getElementById('cart-items');
      const cartTotalElement = document.getElementById('cart-total');

      // Clear existing cart display
      cartItemsElement.innerHTML = '';

      // Populate cart display with current items
      cartItems.forEach(item => {
        const listItem = document.createElement('li');
        listItem.textContent = `${item.productName}: $${item.price}`;
        cartItemsElement.appendChild(listItem);
      });

      // Update total price
      cartTotalElement.textContent = cartTotal;
    }
  </script>
</body>
</html>
```

OUTPUT:



Simple Shopping Cart

Product 1

Price: $10

Add to Cart

Product 2

Price: $15

Add to Cart

Shopping Cart

- Product 1: $10
- Product 2: $15

Total: $25

2 Make the above web application responsive web application using Bootstrap framework

Aim:

To make the above web application responsive web application using Bootstrap framework

Algorithm:

Create an HTML file.

Set the language to English (<html lang="en">).

Include the meta tags for character encoding (UTF-8) and viewport settings for responsive design.

Add a title to the page.

Create a <header> element with a heading (<h1>) to display the title of the shopping cart ("Simple Shopping Cart").

Use Flexbox to arrange elements like the product catalog and shopping cart.

Style the body, header, product container, and individual product items.

Style the shopping cart to have a border and define its width.

Inside the .container, create individual .product divs.

For each product, display the product name, price, and an "Add to Cart" button.

Add an onclick event to the button to trigger the addToCart function, passing the product's name and price as arguments.

Create a div for the shopping cart with a .cart class.

Inside this div, add:

A heading (<h2>) that says "Shopping Cart".

An unordered list (<ul>) with an ID cart-items to display the cart's contents.

A paragraph (<p>) with the total price, which will update dynamically.

Define a JavaScript function addToCart(productName, price):

Push the selected product (name and price) into an array (cartItems).

Add the product price to the total (cartTotal).

Call a function to update the cart display.

Create the updateCartDisplay() function:

Select the HTML elements for the cart's list of items (cart-items) and total price (cart-total).

Clear any existing items in the cart display.

Loop through the cartItems array and create a list item for each product in the cart.

Update the total price in the cart.

Define a JavaScript function addToCart(productName, price):

Push the selected product (name and price) into an array (cartItems).

Add the product price to the total (cartTotal).

Call a function to update the cart display.

Create the updateCartDisplay() function:

Select the HTML elements for the cart's list of items (cart-items) and total price (cart-total).

Clear any existing items in the cart display.

Loop through the cartItems array and create a list item for each product in the cart.

Update the total price in the cart.

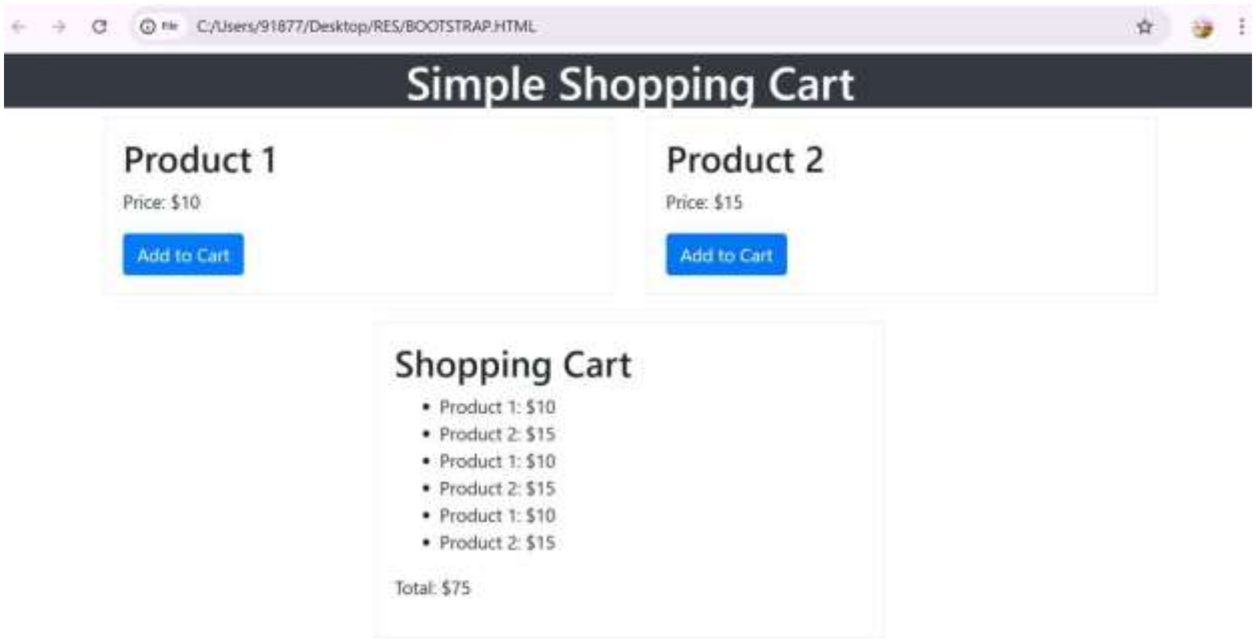
PROGRAM:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Simple Shopping Cart</title>
  <!-- Include Bootstrap CSS -->
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
rel="stylesheet">
  <style>
    /* Custom CSS for changing colors */
    body {
      background-color: #f0f8ff; /* Light blue background for the page */
    }

    header {
      background-color: #2c3e50; /* Dark blue background for the header */
      color: #ecf0f1; /* Light text color */
    }

    .product {
      background-color: #ffffff; /* White background for products */
      border: 1px solid #bdc3c7; /* Light gray border */
    }

    .product h2 {
      color: #2980b9; /* Blue color for product titles */
    }

    .btn-primary {
      background-color: #e74c3c; /* Red color for buttons */
      border-color: #c0392b; /* Darker red border for buttons */
    }

    .btn-primary:hover {
      background-color: #c0392b; /* Darker red on hover */
      border-color: #e74c3c; /* Lighter red border on hover */
    }

    .cart {
      background-color: #ecf0f1; /* Light gray background for the cart */
      border: 1px solid #bdc3c7; /* Light gray border for the cart */
    }

    .cart h2 {
      color: #2c3e50; /* Dark blue color for the cart title */
    }

    .cart p {
      color: #16a085; /* Teal color for total price text */
    }

    ul {
      color: #34495e; /* Dark gray color for cart items */
    }
  </style>
</head>
<body>
  <header class="bg-dark text-white text-center">
    <h1>Simple Shopping Cart</h1>
  </header>
  <div class="container">
    <!-- Product Catalog -->
    <div class="row">
    <div class="col-md-6">
      <div class="product border p-3">
        <h2>Product 1</h2>
        <p>Price: $10</p>
        <button class="btn btn-primary" onclick="addToCart('Product 1', 10)">Add to Cart</button>
      </div>
    </div>
    <div class="col-md-6">
      <div class="product border p-3">
        <h2>Product 2</h2>
        <p>Price: $15</p>
        <button class="btn btn-primary" onclick="addToCart('Product 2', 15)">Add to Cart</button>
      </div>
    </div>
  </div>
  <!-- Shopping Cart -->
  <div class="row mt-4">
    <div class="col-md-6 offset-md-3">
      <div class="cart border p-3">
        <h2>Shopping Cart</h2>
        <ul id="cart-items"></ul>
        <p>Total: $<span id="cart-total">0</span></p>
      </div>
    </div>
  </div>
</div>

<script>
  // JavaScript for handling the shopping cart
```

```
let cartItems = [];
let cartTotal = 0;

// Function to add products to the cart
function addToCart(productName, price) {
cartItems.push({ productName, price });
cartTotal += price;

  // Update the cart display
  updateCartDisplay();
}

// Function to update the cart display
function updateCartDisplay() {
  const cartItemsElement = document.getElementById('cart-items');
  const cartTotalElement = document.getElementById('cart-total');

  // Clear existing cart display
  cartItemsElement.innerHTML = '';

  // Populate cart display with current items
  cartItems.forEach(item => {
    const listItem = document.createElement('li');
    listItem.textContent = `${item.productName}: $${item.price}`;
    cartItemsElement.appendChild(listItem);
  });

  // Update total price
  cartTotalElement.textContent = cartTotal;
```

```
    }
  </script>

  <!-- Include Bootstrap JS -->
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>
```

Output:



C:/Users/91877/Desktop/RES/BOOTSTRAP.HTML

Simple Shopping Cart

Product 1

Price: $10

Add to Cart

Product 2

Price: $15

Add to Cart

Shopping Cart

- Product 1: $10
- Product 2: $15
- Product 1: $10
- Product 2: $15
- Product 1: $10
- Product 2: $15

Total: $75

3 Use JavaScript for doing client – side validation of the pages implemented in experiment 1 and experiment 2.

Aim:

To Use JavaScript for doing client – side validation of the pages

Algorithm:

The HTML structure defines the layout of the page.
It consists of a header displaying the title "Simple Shopping Cart" and a container div for product catalog and shoppingcart.

Basic CSS styles are applied to create a visually appealing layout.
Flexbox is used for layout design, ensuring responsiveness and alignment.

Products are displayed in separate divs with class "product".

Each product has a name, price, input field for quantity, and an "Add to Cart" button.

Two products are included in the catalog with hardcoded prices.

The shopping cart is displayed in a separate div with class "cart".
It contains a title "Shopping Cart", a list element ("ul") for displaying cart items, and a paragraph showing the total price.

JavaScript handles adding products to the shopping cart and updating the cart display.
The addToCart() function is called when the "Add to Cart" button is clicked. It retrieves the product name and price, validates the quantity input, adds the item to the cartItems array, and updates the total price.
The updateCartDisplay() function updates the HTML elements displaying cart items and total price. It clears the existing cart display, iterates over the cartItems array, creates list items for each item, and appends them to the cart display.

Two main variables are declared: cartItems (array to store cart items) and cartTotal (variable to store total price).

Event listeners are added to the "Add to Cart" buttons to trigger the addToCart() function.
Overall, this program provides a simple example of how to create a basic shopping cart functionality using HTML, CSS, and JavaScript, allowing users to add products with quantities to a cart and view the total price.

PROGRAM:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Simple Shopping Cart</title>
<style>
/* Basic CSS with Flexbox for layout */
body {
   font-family: Arial, sans-serif;
   margin: 0;
   padding: 0;
   display: flex;
   flex-direction: column;
   align-items: center;
   background-color: #f0f0f0; /* Light gray background */
}

header {
   background-color: #2c3e50; /* Dark blue header */
   color: #fff;
   padding: 15px;
   width: 100%;
   text-align: center;
}
```

```
.container {
    display: flex;
    flex-wrap: wrap;
    justify-content: space-around;
    padding: 20px;
    max-width: 800px;
}

.product {
    border: 1px solid #ddd;
    padding: 15px;
    margin: 10px;
    text-align: center;
    background-color: #a2c4f4; /* Light blue for product cards */
    border-radius: 8px;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
}

.product h2 {
    color: #2c3e50; /* Dark blue for product titles */
}

.product button {
    background-color: #3498db; /* Blue for buttons */
    color: white;
    padding: 10px 15px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    transition: background-color 0.3s ease;
```

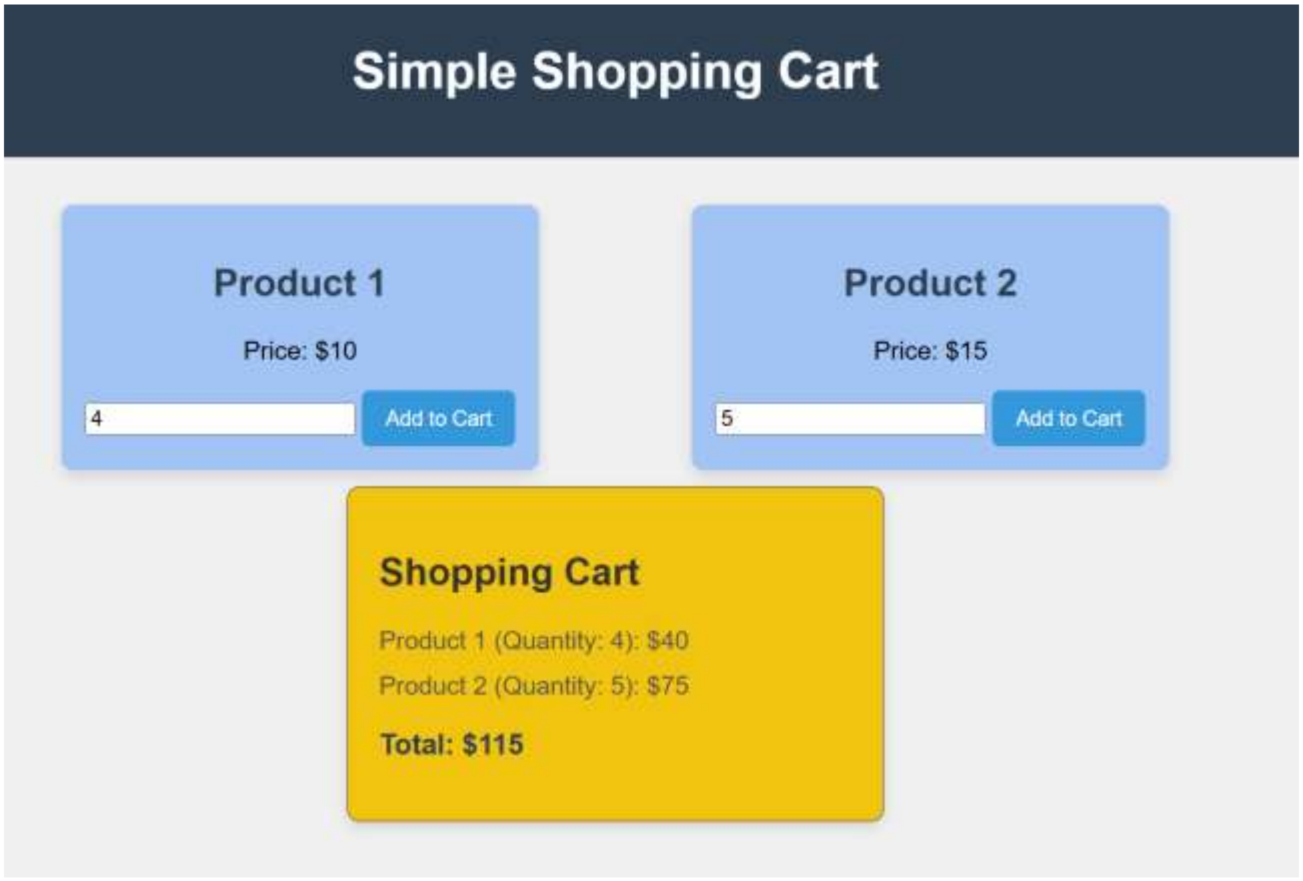
```
}

.product button:hover {
   background-color: #2980b9; /* Darker blue on hover */
}

.cart {
   border: 1px solid #333;
   padding: 20px;
   width: 300px;
   background-color: #f1c40f; /* Soft yellow for the cart */
   border-radius: 8px;
   box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
}

.cart h2 {
   color: #333; /* Dark text for cart header */
}

.cart ul {
   list-style-type: none;
   padding: 0;
}

.cart ul li {
   margin-bottom: 10px;
   color: #555; /* Dark gray text for cart items */
}

.cart p {
   font-size: 18px;
```

```
    color: #2c3e50; /* Dark blue for the total price */
    font-weight: bold;
}
</style>
</head>
<body>
<header>
    <h1>Simple Shopping Cart</h1>
</header>

<div class="container">
    <!-- Product Catalog -->
    <div class="product">
        <h2>Product 1</h2>
        <p>Price: $10</p>
        <input type="number" id="quantity1" placeholder="Quantity" min="1">
        <button onclick="addToCart('Product 1', 10, 'quantity1')">Add to Cart</button>
    </div>
    <div class="product">
        <h2>Product 2</h2>
        <p>Price: $15</p>
        <input type="number" id="quantity2" placeholder="Quantity" min="1">
        <button onclick="addToCart('Product 2', 15, 'quantity2')">Add to Cart</button>
    </div>
    <!-- Shopping Cart -->
    <div class="cart">
        <h2>Shopping Cart</h2>
        <ul id="cart-items"></ul>
        <p>Total: $<span id="cart-total">0</span></p>
    </div>
</div>
```

```
<script>
// JavaScript for handling the shopping cart
let cartItems = [];
let cartTotal = 0;

function addToCart(productName, price, quantityId) {
  const quantityField = document.getElementById(quantityId);
  const quantity = parseInt(quantityField.value);

  // Validate quantity
  if (isNaN(quantity) || quantity < 1) {
    alert("Please enter a valid quantity (1 or more).");
    return;
  }

  // Add to cart
  cartItems.push({ productName, price, quantity });
  cartTotal += price * quantity;

  // Update the cart display
  updateCartDisplay();
}

function updateCartDisplay() {
  const cartItemsElement = document.getElementById('cart-items');
  const cartTotalElement = document.getElementById('cart-total');

  // Clear existing cart display
  cartItemsElement.innerHTML = '';
```

```
    // Populate cart display with current items
    cartItems.forEach(item => {
        const listItem = document.createElement('li');
        listItem.textContent = `${item.productName} (Quantity: ${item.quantity}): $${item.price * item.quantity}`;
        cartItemsElement.appendChild(listItem);
    });

    // Update total price
    cartTotalElement.textContent = cartTotal;
}
</script>


</body>
</html>
```

OUTPUT:



Simple Shopping Cart

Product 1

Price: $10

4 Add to Cart

Product 2

Price: $15

5 Add to Cart

Shopping Cart

Product 1 (Quantity: 4): $40

Product 2 (Quantity: 5): $75

Total: $115

4 Explore the features of ES6 like arrow functions, callbacks, promises, async/await. Implement an application for reading the weather information from openweathermap.org and display the information in the form of a graph on the web page.

Aim: To explore the features of ES6 like arrow functions, callbacks, promises, async/await. Implement an application for reading the weather information from openweathermap.org and display the information in the form of a graph on the web page.

Algorithm:

The HTML structure includes an input field for entering the city name, a button to trigger the weather data fetch, a div to display weather information, and a canvas element to render the bar chart.

The getWeather() function is triggered when the "Get Weather" button is clicked.

It retrieves the city name entered by the user and constructs the API URL with the city name and API key.

It then uses the Fetch API to make a request to the OpenWeatherMap API.
Upon receiving a response, it extracts the temperature data from the JSON response and updates the weatherInfo div with the city name and temperature.

It also dynamically creates a bar chart using Chart.js library, representing the temperature data.
If there's an error during the fetch process, it catches the error and displays an error message in the weatherInfo div.

Chart.js library is included via a CDN link in the head of the HTML document.
Upon successful retrieval of weather data, a bar chart is created using the temperature data.

The chart is rendered inside the canvas element with id "weatherChart".

An API key from OpenWeatherMap is used for making requests to fetch weather data.

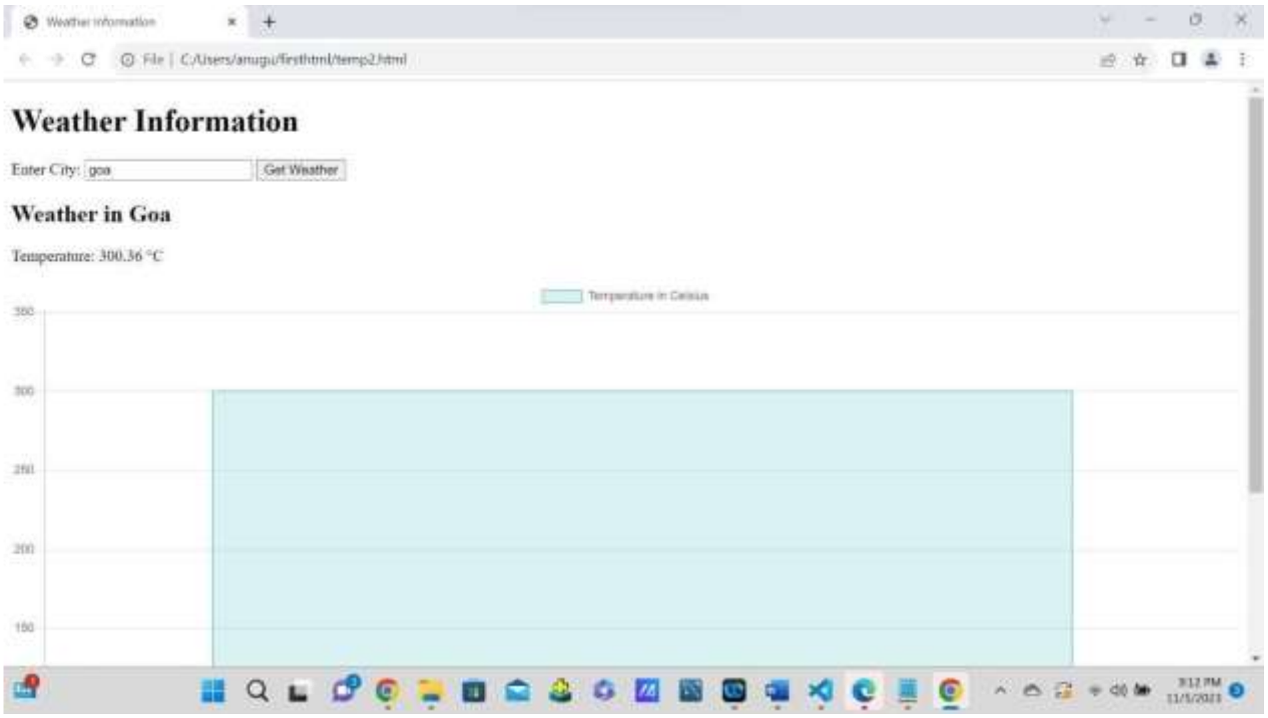
Errors that occur during the fetch process are caught and displayed in the weatherInfo div to notify the user about the issue.
Overall, this program demonstrates how to fetch data from an API, dynamically update the webpage content based on the data received, and integrate a chart library to visualize the data in a graphical format.

Program:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Weather Information</title>
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
</head>
<body>
<h1>Weather Information</h1>
<label for="cityInput">Enter City: </label>
```

```
<input type="text" id="cityInput" placeholder="Enter city name">
<button onclick="getWeather()">Get Weather</button>
<div id="weatherInfo"></div>
<canvas id="weatherChart" width="400" height="200"></canvas>

<script>
function getWeather() {
  const apiKey = 'd073796cc2255e3d2582e5b474c53ee3';
  const cityInput = document.getElementById('cityInput').value;
  const apiUrl = `https://api.openweathermap.org/data/2.5/weather?q=${cityInput}&appid=${apiKey}`;

  fetch(apiUrl)
  .then(response => response.json())
  .then(data => {
    // Convert temperature from Kelvin to Celsius
    const temperatureKelvin = data.main.temp;
    const temperatureCelsius = (temperatureKelvin - 273.15).toFixed(2); // Convert Kelvin to Celsius

    const weatherInfo = document.getElementById('weatherInfo');
    weatherInfo.innerHTML = `<h2>Weather in ${data.name}</h2>`;
    weatherInfo.innerHTML += `<p>Temperature: ${temperatureCelsius} °C</p>`;

    // Create a bar graph
    const ctx = document.getElementById('weatherChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Temperature (°C)'],
        datasets: [{
          label: 'Temperature in Celsius',
          data: [temperatureCelsius],
          backgroundColor: 'rgba(75, 192, 192, 0.2)',
          borderColor: 'rgba(75, 192, 192, 1)',
          borderWidth: 1
        }]
      },
      options: {
        scales: {
          y: {
            beginAtZero: true,
            title: {
              display: true,
              text: 'Temperature (°C)'  // Adding a label for Y-axis
            }
          }
        }
      }
     });
  })
  .catch(error => {
        console.error('Error fetching weather data:', error);
        const weatherInfo = document.getElementById('weatherInfo');
        weatherInfo.innerHTML = '<p>Error fetching weather data. Please try again later.</p>';
    });
}
</script>
</body>
</html>
```

Output:



Weather Information

Enter City: goa

Weather in Goa

Temperature: 300.36 °C

5) Develop a java stand alone application that connects with the database (Oracle / mySql) and perform the CRUD operation on the database tables.

Aim:

To develop a java stand alone application that connects with the database (Oracle / mySql) and perform the CRUD operation on the database tables.

Algorithm:

1. Import Statements:

- Import necessary classes from the java.sql package for JDBC operations.

2. Main Method:

- The `main` method is the entry point of the program.

3. Try-Catch Block:

- The database operations are wrapped inside a try-catch block to handle any potential exceptions that might occur during the execution.

4. Loading JDBC Driver:

-`Class.forName("com.mysql.cj.jdbc.Driver")`loads the MySQL JDBC driver class.

5. Establishing Connection:

- `DriverManager.getConnection()`establishes a connection to the MySQL database.

- The connection string specifies the URL (`jdbc:mysql://localhost:3306/jdbc_first`), username (`root`), and password (`root`) to connect to the database named `jdbc_first` running on `localhost` at port `3306`.

6. Creating a Statement:

- `connection.createStatement()`creates a Statement object for executing SQL queries.

7. Creating a Table:

- `statement.execute("create table user(id int primary key,name varchar(45),phone bigint(10))")` executes an SQL query to create a table named `user` with three columns: `id`, `name`, and `phone`.

8. Inserting Data:

- `statement.execute("insert into user values(12,'chand',9848844129)")` executes an SQL query to insert a row into the `user` table.

9. Closing Resources:

- `connection.close()` closes the database connection to release resources.

10.Exception Handling:

- Catch blocks handle `ClassNotFoundException` and `SQLException` that might occur during driver loading or database operations.

11. Output:

- Various `println`statements are used to output information about the connection and statement objects.

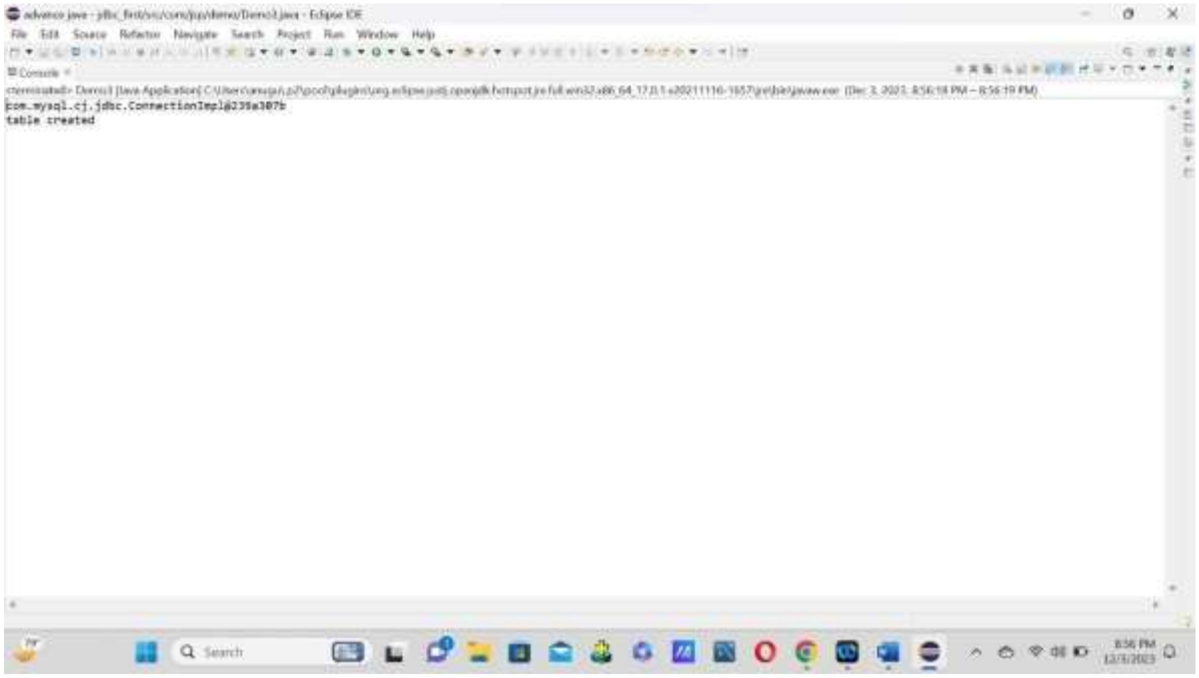
- The program prints "table created" to indicate successful execution.

This program demonstrates how to connect to a MySQL database, create a table, insert data, and handle exceptions that may occur during the process.

Program:

```
package com.jsp.demo;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;


public class Demo3 {
    public static void main(String[] args) {
        // Define the database connection URL and credentials
        String url = "jdbc:mysql://localhost:3306/jdbc_first";
        String username = "root";
        String password = "root";

        // Define the SQL queries
        String createTableSQL = "CREATE TABLE IF NOT EXISTS user ("
                + "id INT PRIMARY KEY, "
                + "name VARCHAR(45), "
                + "phone BIGINT(10))";
        String insertSQL = "INSERT INTO user (id, name, phone) VALUES (?, ?, ?)";

        // Establish the database connection
        try (Connection connection = DriverManager.getConnection(url, username, password);
             Statement statement = connection.createStatement();
             PreparedStatement preparedStatement = connection.prepareStatement(insertSQL)) {

            // Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Execute the CREATE TABLE query (only if the table doesn't exist)
            statement.execute(createTableSQL);
            System.out.println("Table 'user' created or already exists.");

            // Insert a record using PreparedStatement
            preparedStatement.setInt(1, 12); // Set ID
            preparedStatement.setString(2, "chand"); // Set Name
            preparedStatement.setLong(3, 9848844129L); // Set Phone Number
            preparedStatement.executeUpdate();
            System.out.println("Record inserted into 'user' table.");

            // Uncomment to execute UPDATE or DELETE operations
            // statement.execute("UPDATE user SET name='Dil' WHERE id=9");
            // statement.execute("DELETE FROM user WHERE id=9");

        } catch (ClassNotFoundException e) {
            e.printStackTrace(); // JDBC Driver not found
        } catch (SQLException e) {
            e.printStackTrace(); // SQL error (e.g., connection issue, query failure)
        }
    }
}
```

Output:



```
com.mysql.cj.jdbc.ConnectionImpl@235a307b
table created
```

6) Create an xml for the bookstore. Validate the same using both DTD and XSD.

Aim:

To create an xml for the bookstore. Validate the same using both DTD and XSD.

Algorithm:

To Check the Validity :

1. Go to the below link, https://www.liquid-technologies.com/online-xsd-validator
2. Place the **XML code** in the XML Validate.
3. Place the **XSD code** in the XML Schema Data.
4. Then click the **validate** Button.
5. Then it will show the Document as Valid.

Program:

Bookstore.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE bookstore[

<!ELEMENT bookstore (book+)>

<!ELEMENT book (title, author, price)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT author (#PCDATA)>

<!ELEMENT price (#PCDATA)>

]>

<bookstore>

<book>

<title>Introduction to XML</title>

<author>John Doe</author>
```

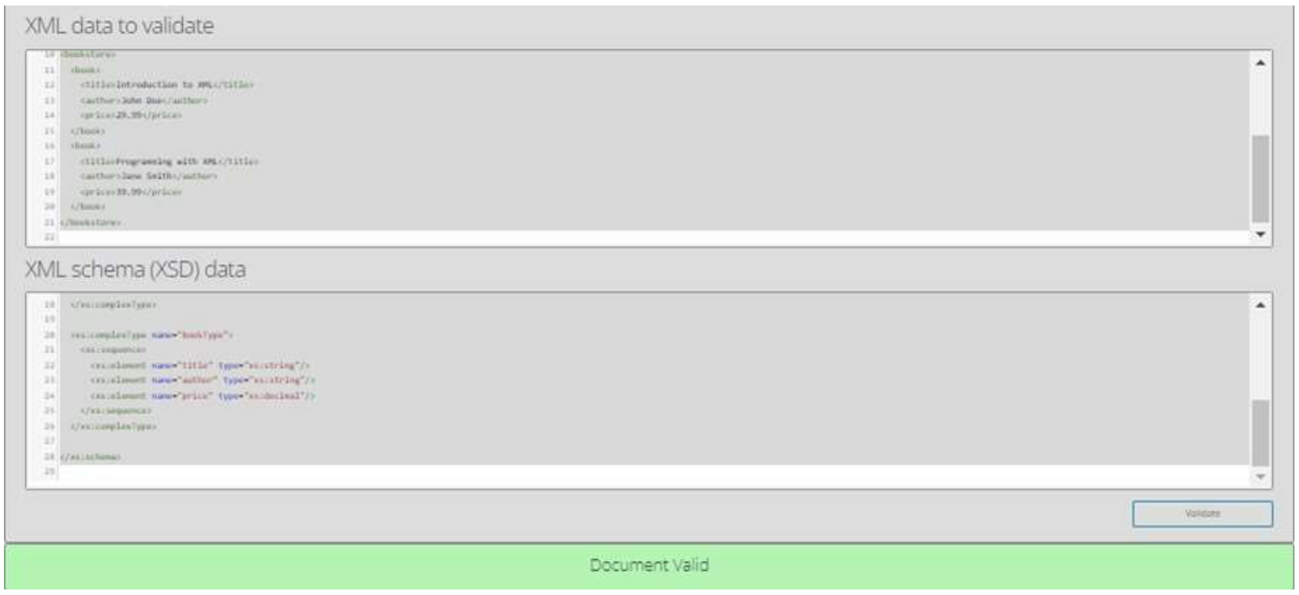
```
<price>29.99</price>

</book>

<book>

<title>Programming with XML</title>

<author>Jane Smith</author>

<price>39.99</price>

</book>

</bookstore>
```

Bookstore.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

          targetNamespace="http://example.com"

          xmlns="http://example.com">

<xs:element name="root">

<xs:complexType>

<xs:sequence>

<xs:element name="bookstore" type="bookstoreType"/>

</xs:sequence>

</xs:complexType>

</xs:element>


<xs:complexType name="bookstoreType">

<xs:sequence>

<xs:element name="book" type="bookType" minOccurs="0" maxOccurs="unbounded"/>

</xs:sequence>
```

```
</xs:complexType>

<xs:complexType name="bookType">

<xs:sequence>

<xs:element name="title" type="xs:string"/>

<xs:element name="author" type="xs:string"/>

<xs:element name="price" type="xs:decimal"/>

</xs:sequence>

</xs:complexType>

</xs:schema>
```

Output :



XML data to validate

```
<bookstore>
  <book>
    <title>Introduction to XML</title>
    <author>John Doe</author>
    <price>29.99</price>
  </book>
  <book>
    <title>Programming with XML</title>
    <author>Jane Smith</author>
    <price>39.99</price>
  </book>
</bookstore>
```

XML schema (XSD) data

```
  </xs:complexType>

  <xs:complexType name="bookType">
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author" type="xs:string"/>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Document Valid

7. Design a controller with servlet that provides the interaction with application developed in experiment 1 and the database created in experiment 5.

Program:

Product.html

```
<div class="container">

  <!-- Product Catalog -->
  <div class="product">
    <h2>Product 1</h2>
    <p>Price: $10</p>
    <button onclick="addToCart('Product 1', 10)">Add to Cart</button>
  </div>
  <div class="product">
    <h2>Product 2</h2>
    <p>Price: $15</p>
    <button onclick="addToCart('Product 2', 15)">Add to Cart</button>
  </div>

  <!-- Shopping Cart -->
  <div class="cart">
    <h2>Shopping Cart</h2>
    <ul id="cart-items"></ul>
    <p>Total: $<span id="cart-total">0</span></p>
  </div>
</div>
```

Product.java

```
package com.jsp.demo;

public class Product {
    private String name;
    private double price;
    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}
```

Cart,java

```
package com.jsp.demo;

import java.util.ArrayList;
import java.util.List;

public class Cart {
    private List<Product> products;
    private double total;

    public Cart() {
        this.products = new ArrayList<>();
        this.total = 0.0;
    }

    public void addProduct(Product product) {
        products.add(product);
        total += product.getPrice();
    }

    public List<Product> getProducts() {
        return products;
    }

    public double getTotal() {
        return total;
    }
}
```

ShoppingCartServlet.java

```
package com.jsp.demo;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
```

```java
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;


@WebServlet("/ShoppingCartServlet")
public class ShoppingCartServlet extends HttpServlet {


    // Initialize some example products
    private static Map<String, Product> productCatalog = new HashMap<>();


    static {
        productCatalog.put("product1", new Product("Product 1", 10.0));
        productCatalog.put("product2", new Product("Product 2", 15.0));
    }


    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {


        // Retrieve the cart from session
        HttpSession session = request.getSession();
        Cart cart = (Cart) session.getAttribute("cart");
        if (cart == null) {
            cart = new Cart();
            session.setAttribute("cart", cart);
        }

        // Get the product to add from the request parameter
        String productId = request.getParameter("productId");
        if (productId != null && productCatalog.containsKey(productId)) {
            Product product = productCatalog.get(productId);
            cart.addProduct(product);
        }

        // Forward to the JSP page to display the cart
        request.getRequestDispatcher("index.jsp").forward(request, response);
    }
}
```

Web.xml

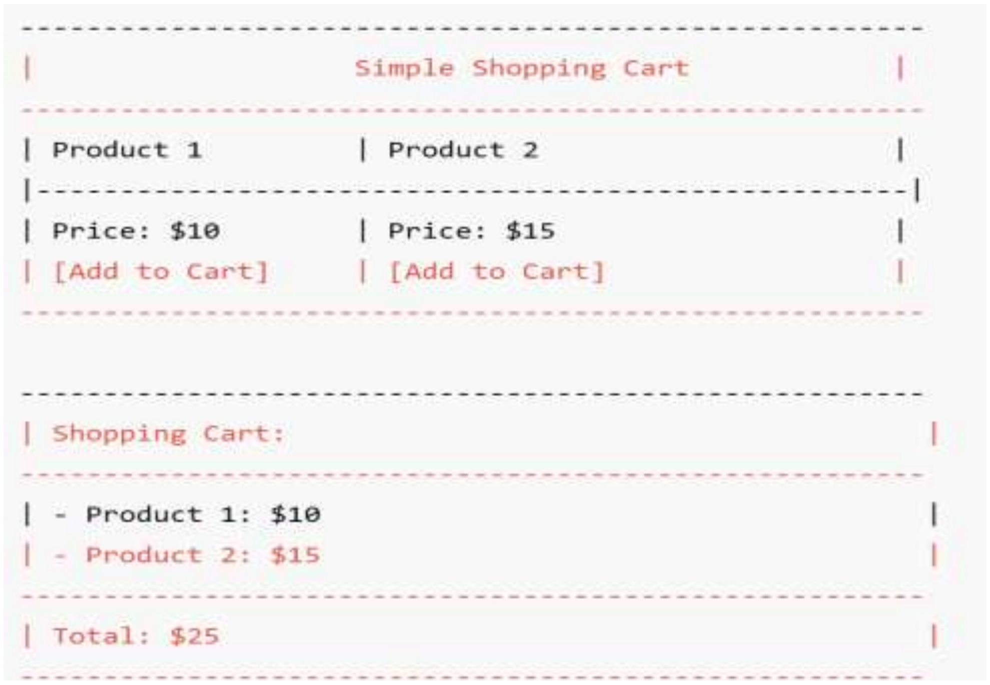
```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd" version="3.0">


    <servlet>
        <servlet-name>ShoppingCartServlet</servlet-name>
        <servlet-class>com.jsp.demo.ShoppingCartServlet</servlet-class>
    </servlet>


    <servlet-mapping>
        <servlet-name>ShoppingCartServlet</servlet-name>
        <url-pattern>/ShoppingCartServlet</url-pattern>
    </servlet-mapping>

</web-app>
```

index.jsp

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Simple Shopping Cart</title>
  <style>
    /* Basic CSS with Flexbox for layout */
    body {
      font-family: Arial, sans-serif;
      margin: 0;
      padding: 0;
      display: flex;
      flex-direction: column;
      align-items: center;
    }

    header {
      background-color: #333;
      color: #fff;
      padding: 10px;
      width: 100%;
      text-align: center;
    }
    .container {
      display: flex;
      flex-wrap: wrap;
      justify-content: space-around;
      padding: 20px;
      max-width: 800px;
    }

    .product {
      border: 1px solid #ddd;
      padding: 10px;
      margin: 10px;
      text-align: center;
    }

    .cart {
      border: 1px solid #333;
      padding: 10px;
      width: 300px;
    }
  </style>
```

```
</head>
<body>
  <header>
    <h1>Simple Shopping Cart</h1>
  </header>

  <div class="container">
    <!-- Product Catalog -->
    <c:forEach var="product" items="${products}">
      <div class="product">
        <h2>${product.name}</h2>
        <p>Price: $${product.price}</p>
        <a href="ShoppingCartServlet?productId=${product.id}">Add to Cart</a>
      </div>
    </c:forEach>

    <!-- Shopping Cart -->
    <div class="cart">
      <h2>Shopping Cart</h2>
      <ul>
        <c:forEach var="cartItem" items="${cart.products}">
          <li>${cartItem.name}: $${cartItem.price}</li>
        </c:forEach>
      </ul>
      <p>Total: $${cart.total}</p>
    </div>

  </div>


</body>

</html>
```

Output:



```
-----------------------------------------------------
|                    Simple Shopping Cart           |
-----------------------------------------------------
| Product 1          | Product 2                    |
|---------------------------------------------------|
| Price: $10         | Price: $15                   |
| [Add to Cart]      | [Add to Cart]                |
-----------------------------------------------------


-----------------------------------------------------
| Shopping Cart:                                      |
-----------------------------------------------------
| - Product 1: $10                                    |
| - Product 2: $15                                    |
-----------------------------------------------------
| Total: $25                                          |
-----------------------------------------------------
```

8. Maintaining the transactional history of any user is very important. Explore the various session tracking mechanism (Cookies, HTTP Session)

AIM: Maintaining the transactional history of any user is very important. Explore the various session tracking mechanism (Cookies, HTTP Session)

DESCRIPTION:

Session tracking mechanisms are crucial for maintaining the state of a user's interactions with a web application. Two common methods for session tracking are Cookies and HTTP Sessions.

1. Cookies: Cookies are small data pieces stored on a user's device by a web browser, used to maintain user-specific information between the client and the server.

Example: Suppose you want to track the user's language preference. Server-side (in a web server script, e.g., in Python with Flask):

```
from flask import Flask, request, render_template, make_response
app = Flask(__name__)
@app.route('/')
def index():
# Check if the language cookie is set
user_language = request.cookies.get('user_language')
return
render_template('index.html',user_language=user_language)
@app.route('/set_language/<language>')
def set_language(language):
# Set the language preference in a cookie
response = make_response(render_template('set_language.html'))
response.set_cookie('user_language', language)
return response
if __name__== '__main__':
app.run(debug=True)
```

HTML Templates (index.html and set_language.html):

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Cookie Example</title>
  </head>
  <body>
    <h1>Welcome to the website!</h1>
    {% if user_language %}
    <p>Your preferred language is: {{ user_language }}</p>
    {% else %}
    <p>Your language preference is not set.</p>
    {% endif %}
    <p><a href="/set_language/en">Set language to English</a></p>
    <p><a href="/set_language/es">Set language to Spanish</a></p>
  </body>
</html>
<!-- set_language.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Set Language</title>
  </head>
  <body>
    <h2>Language set successfully!</h2>
    <p><a href="/">Go back to the home page</a></p>
  </body>
</html>
```

Output:

When a user visits the site for the first time, the language preference is not set. When the user clicks on "Set language to English" or "Set language to Spanish," the preference is stored in a cookie.

On subsequent visits, the site recognizes the user's language preference based on the cookie.

2. HTTP Session: An HTTP session is a way to store information on the server side between requests from the same client. Each client gets a unique session ID, which is used to retrieve session data.

Example: Suppose you want to track the number of visits for each user. Server-side (in a web server script, e.g., in Python with Flask):

```
from flask import Flask, request, render_template, session

app = Flask( _name _)

app.secret_key = 'super_secret_key'

# Set a secret key for session management

@app.route('/')

def index():

# Increment the visit count in the session

session['visit_count'] =

session.get('visit_count', 0) + 1

return render_template('index_session.html',

visit_count=session['visit_count']) if _name__== '_main_':

app.run(debug=True)
```

HTML Template (index_session.html):

```
<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Session Example</title>

  </head>

  <body>

    <h1>Welcome to the website!</h1>

    <p>This is your visit number: {{ visit_count }}</p>

  </body>

</html>
```

Output:

- Each time a user visits the site, the server increments the visit count stored in the session.
- The visit count is unique to each user and persists across multiple requests until the session expires.

These examples demonstrate simple use cases for cookies and HTTP sessions to maintain user-specific information on the client and server sides, respectively.

9. Create a custom server using http module and explore the other modules of Node JS like OS, path, event.

AIM: Create a custom server using http module and explore the other modules of Node JS like OS, path, event

DESCRIPTION: Let's create a simple custom server using the http module in Node.js and then explore the os, path, and events modules with examples.

1. Creating a Custom Server with

http Module: Server-side

(server.js):

```
const http = require('http');

const server = http.createServer((req,

res) => { res.writeHead(200,

{'Content-Type': 'text/plain'

});

res.end('Hello, this is a custom

server!');}); const PORT =

3000;server.listen(PORT, () => {

console.log(`Server is listening on port

${PORT}`);});
```

To run the server: node server.js

Output:

Visit http://localhost:3000 in your browser or use a tool like curl or Postman to make a request, and you should see the response "Hello, this is a custom server!".

2. Exploring Node.js Modules:

A. os Module:

The os module provides operating system-related utility methods and properties.

Example:

```
const os = require('os');

console.log('OS Platform:',

os.platform()); console.log('OS
Architecture:', os.arch());
console.log('Total Memory (in bytes):',
os.totalmem()); console.log('Free
Memory (in bytes):', os.freemem());
```

Output: This will output information about the operating system platform, architecture, total memory, and free memory.

B.path Module:

The **path** module provides methods for working with file and directory paths.

Example:

```
const path = require('path');
const filePath = '/path/to/some/file.txt';
console.log('File Name:',
path.basename(filePath));
console.log('Directory Name:',
path.dirname(filePath));
console.log('File Extension:',
path.extname(filePath));
```

Output: This will output the file name, directory name, and file extension for the given file path.

C. Events Module:

The events module provides an implementation of the Event Emitter pattern, allowing objects to emit and listen for events.

Example:

```
const EventEmitter =
require('events'); class
MyEmitter extends
EventEmitter {} const
myEmitter = new
MyEmitter();
```

10. Develop an express web application that can interact with REST API to perform CRUD operations on student data. (Use Postman)

Firstly we need to create a new folder and open the folder in the command prompt and enter a command as below:

```
npminit -y
```

Open that folder in the vscode by entering code.
Next in the terminal we need to install all the packages we need, so we mainly use express and sqlite3.
The Command to install express and sqlite3 is

```
npm install express sqlite3
```

Then create file named as the app.js and db.js

db.js

```
const sqlite3 = require('sqlite3').verbose();

// Function to initialize the database schema
function initializeDatabase() {
   const db = new sqlite3.Database('./mydatabase.db', (err) => {
      if (err) {
         console.error(err.message);
      } else {
         console.log('Connected to the SQLite database.');
         createStudentsTable(db);
      }
   });

   // Close the database connection when the Node process exits
   process.on('exit', () => {
      db.close((err) => {
         if (err) {
            console.error(err.message);
         } else {
            console.log('Disconnected from the SQLite database.');
         }
      });
   });
}
// Function to create the 'students' table if it doesn't exist
function createStudentsTable(db) {
   const createTableQuery = `
      CREATE TABLE IF NOT EXISTS students (
         id INTEGER PRIMARY KEY AUTOINCREMENT,
         name TEXT,
         age INTEGER,
         grade TEXT
      );
```

```
    `;

    db.run(createTableQuery, (err) => {
        if (err) {
            console.error(err.message);
        } else {
            console.log('The students table has been created or already exists.');
        }
    });
}

module.exports = { initializeDatabase };
```

when we execute both the db.js then the database will be created that is mydatabase.db
app.js

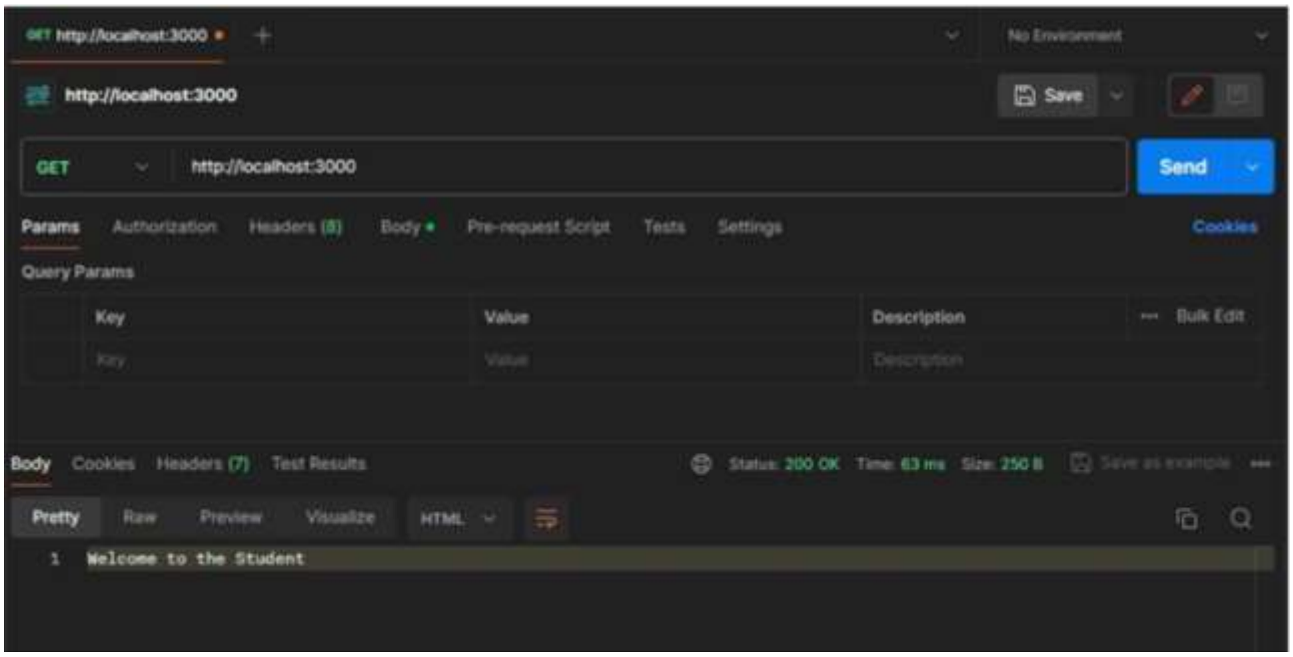
```
const express = require('express');
const sqlite3 = require('sqlite3');
const{ initializeDatabase } = require('./db');
const app = express();
const port = 3000;

// Connect to SQLite database
const db = new sqlite3.Database('./mydatabase.db', (err) => {
    if (err) {
        console.log(err.message);
    } else {
        console.log('Connected to the SQLite database.');
    }
});

// Middleware to parse request body as JSON
app.use(express.json());

app.get('/', (req, res) => {
    res.send('Welcome to the Student');
});

// Get all Students
app.get('/students', (req, res) => {
    db.all('SELECT * FROM students', [], (err, rows) => {
        if (err) {
            return console.error(err.message);
        }
        res.json(rows);
    });
});

// Get a single student by id
```

```
app.get('/students/:id', (req, res) => {
   const id = req.params.id;
   db.all('SELECT * FROM students WHERE id = ?', [id], (err, row) => {
      if (err) {
         return  console.error(err.message);
      }
      res.json(row);
   });
});

// Create a new student
app.post('/students', (req, res) => {
   const{ name, age, grade } = req.body;
   db.run('INSERT INTO students (name, age, grade) VALUES (?, ?, ?)', [name, age,
grade], function (err) {
      if (err) {
         return  console.error(err.message);
      }
      res.status(201).json({  id:this.lastID  });
   });
});

// Update a student
app.put('/students/:id', (req, res) => {
   const id = req.params.id;
   const{ name, age, grade } = req.body;
   db.run('UPDATE students SET name = ?, age = ?, grade = ? WHERE id = ?', [name,
age, grade, id], function (err) {
      if (err) {
         return  console.error(err.message);
      }
      res.json({ updatedID:id });
   });
});

// Delete a student
app.delete('/students/:id', (req, res) => {
   const id = req.params.id;
   db.run('DELETE FROM students WHERE id = ?', id, function (err) {
      if (err) {
         return  console.error(err.message);
      }
      res.json({  deletedID:id  });
   });
});

app.listen(port, () => {
   console.log('Server running at http://localhost:${port}');
```

});
OUTPUT:

GET:

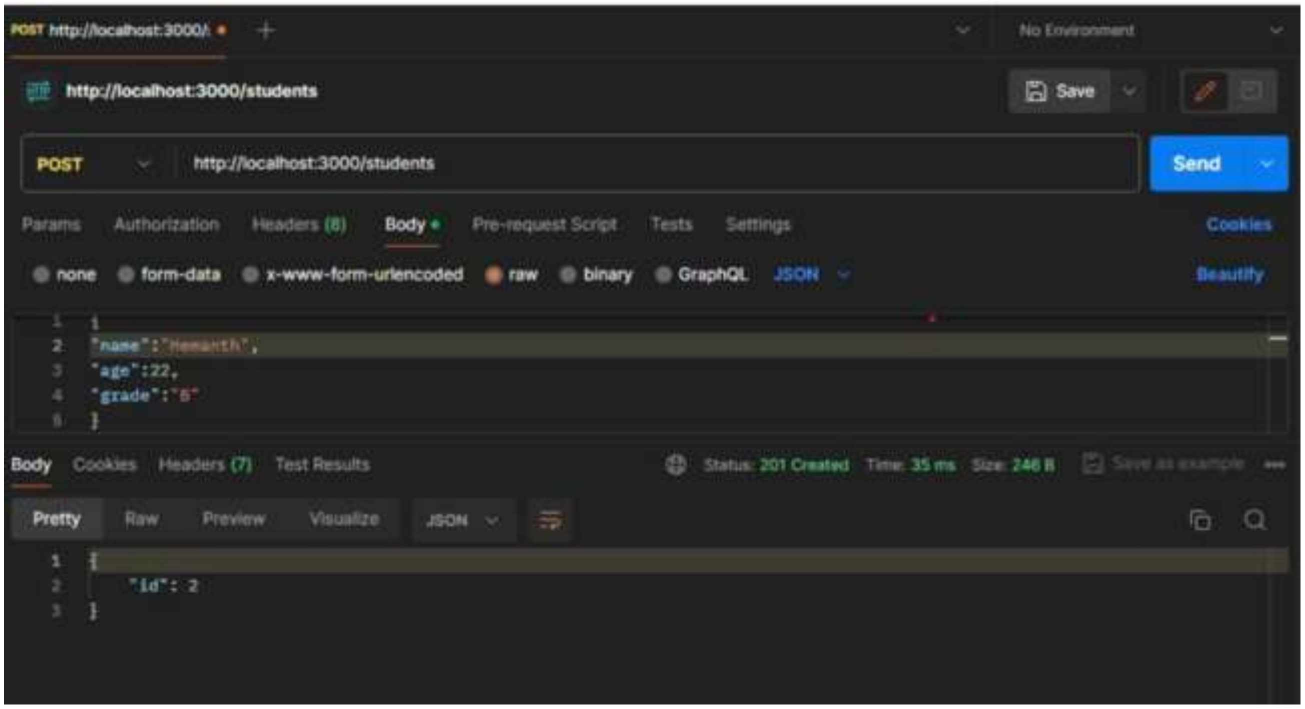
- Open Postman.
- Set the request type to GET.
- Enter the URL: *http://localhost:3000/students.*



```
Welcome to the Student
```

POST : Create a New Student

- Open Postman.
- Set the request type to POST.
- Enter the URL: *http://localhost:3000/students.*
- Go to the **"Body"** tab.
- Select raw and set the body to JSON format.



http://localhost:3000/students

POST http://localhost:3000/students

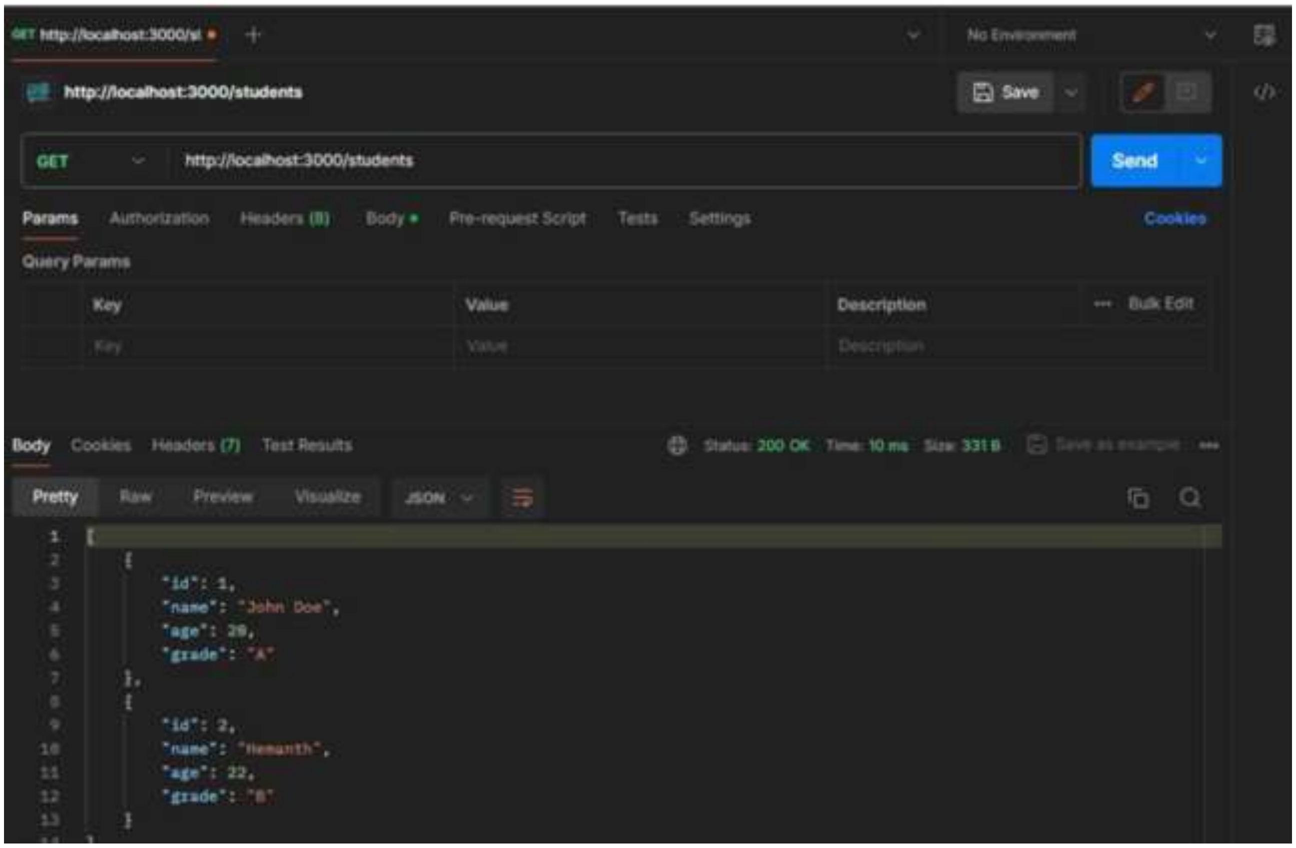
```
{
"name":"Hemanth",
"age":22,
"grade":"6"
}
```

Status: 201 Created Time: 35 ms Size: 246 B

```
{
    "id": 2
}
```

GET: #all Students

- Set the request type to GET.
- Enter the URL: *http://localhost:3000/students.*
- Click on the **"Send"** button
- You should receive a response with details of all students in your SQLite database.

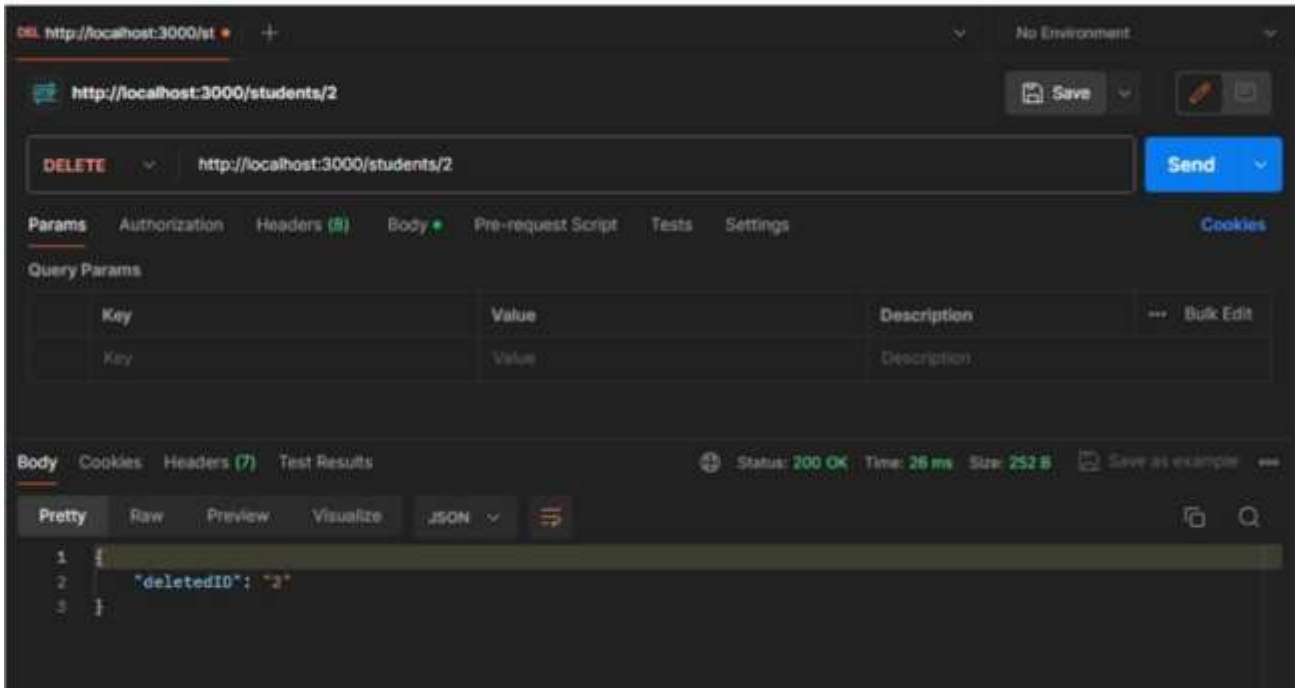


GET http://localhost:3000/students

```
[
    {
        "id": 1,
        "name": "John Doe",
        "age": 20,
        "grade": "A"
    },
    {
        "id": 2,
        "name": "Hemanth",
        "age": 22,
        "grade": "B"
    }
]
```

DELETE:

- Set the request type to DELETE.
- Enter the URL for the student you want to delete (replace: id with an actual student ID): *http://localhost:3000/students/:id*
- Place instead of ID which replace with number that is ID to be deleted.
- Then click **send**



http://localhost:3000/students/2

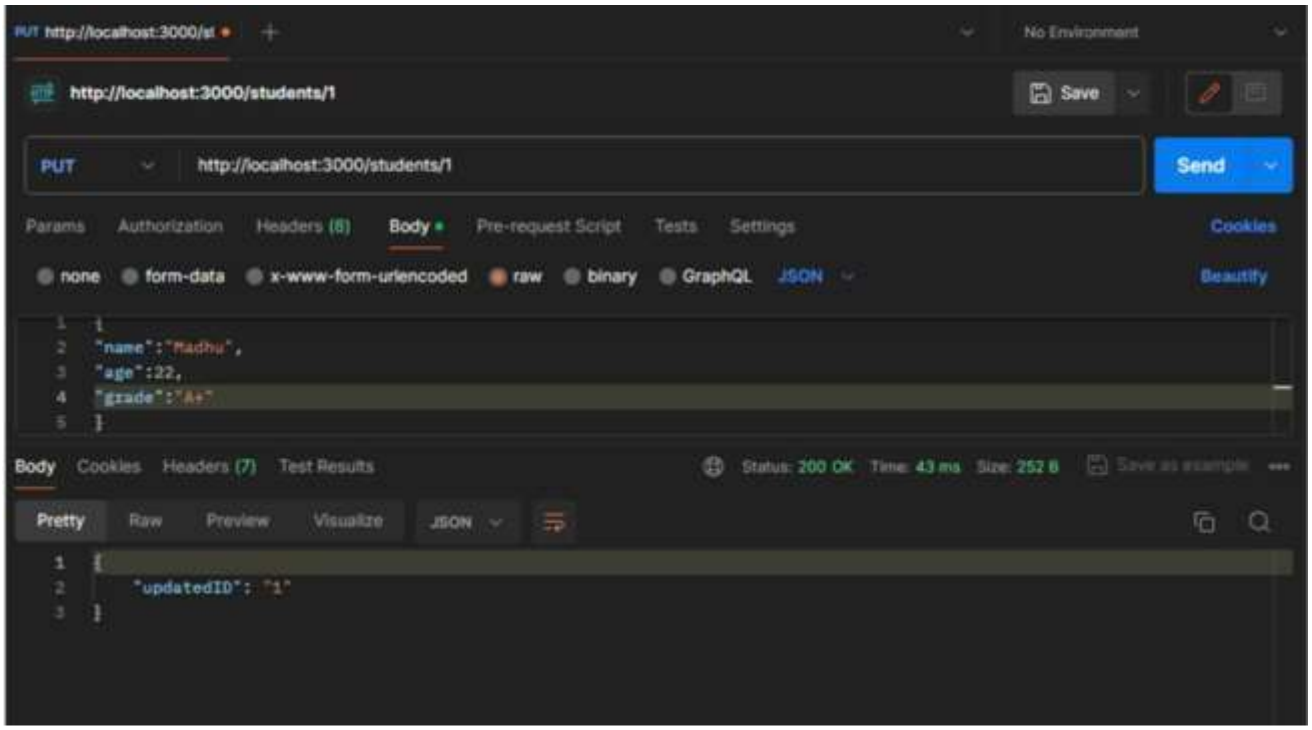
DELETE http://localhost:3000/students/2

Status: 200 OK Time: 26 ms Size: 252 B

```
{
    "deletedID": "2"
}
```

PUT:

- Set the request type to PUT.
- Enter the URL for the student you want to delete (replace: id with an actual student ID): *http://localhost:3000/students/:id*
- Go to the **"Body"** tab.
- Select raw and set the body to JSON format



PUT http://localhost:3000/students/1

```
{
"name":"Madhu",
"age":22,
"grade":"A+"
}
```

Status: 200 OK Time: 43 ms Size: 252 B

```
{
    "updatedID": "1"
}
```

<span style="color:red">**11. For the above application create authorized end points using JWT (JSON Web Token).**</span>

Step 1: Set up your project
If you haven't already set up a Node.js project, create one by running:
mkdir my-jwt-app
cd my-jwt-app
npm init –y

Then, install the required dependencies:

npm install express jsonwebtoken bcryptjs dotenv
Step 2: Create a JWT utility file (jwtUtils.js)
First, create a utility file to handle JWT generation and verification.

```
// jwtUtils.js
const jwt = require('jsonwebtoken');

// Secret key for JWT signing (store this securely, e.g., in environment variables)
const JWT_SECRET_KEY = process.env.JWT_SECRET_KEY || 'your-secret-key';

// Function to generate a JWT token
function generateToken(userId) {
const payload = { userId };
  const options = { expiresIn: '1h' }; // Set the expiration time for the token

  return jwt.sign(payload, JWT_SECRET_KEY, options);
}
```

```
// Function to verify a JWT token
function verifyToken(token) {
try {
    const decoded = jwt.verify(token, JWT_SECRET_KEY);
    return decoded; // Returns the decoded payload if the token is valid
  } catch (err) {
    return null; // Return null if the token is invalid
  }
}

module.exports = { generateToken, verifyToken };
```

Step 3: Create an Express server (server.js)

Next, create your main server file where you define routes for login, registering users, and protecting routes using JWT authorization.

javascript
Copy

```
// server.js
const express = require('express');
const bcrypt = require('bcryptjs');
const jwtUtils = require('./jwtUtils');
const dotenv = require('dotenv');

// Initialize dotenv for environment variables
dotenv.config();

// Create an express app
const app = express();

// Middleware to parse JSON request bodies
app.use(express.json());

// Dummy users database (for the sake of the example, you should use a real database)
const users = [];

// Endpoint to register a new user
app.post('/register', async (req, res) => {
const { username, password } = req.body;

  // Check if the user already exists
  const userExists = users.some(user => user.username === username);
  if (userExists) {
    return res.status(400).json({ message: 'User already exists' });
  }

  // Hash the password
  const hashedPassword = await bcrypt.hash(password, 10);
  // Save the user to the dummy database
  const newUser = { username, password: hashedPassword };
  users.push(newUser);

  return res.status(201).json({ message: 'User registered successfully' });
});

// Endpoint to login a user and generate a JWT
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  // Find the user in the database
  const user = users.find(user => user.username === username);
  if (!user) {
    return res.status(401).json({ message: 'Invalid credentials' });
  }

  // Compare the password with the hashed one in the database
  const passwordMatch = await bcrypt.compare(password, user.password);
  if (!passwordMatch) {
    return res.status(401).json({ message: 'Invalid credentials' });
  }

  // Generate a JWT token
  const token = jwtUtils.generateToken(user.username);

  return res.json({ message: 'Login successful', token });
});

// Middleware to check if the request has a valid JWT token
function authenticateJWT(req, res, next) {
  const token = req.header('Authorization') && req.header('Authorization').replace('Bearer
', '');

  if (!token) {
    return res.status(403).json({ message: 'No token provided' });
  }

  const decoded = jwtUtils.verifyToken(token);

  if (!decoded) {
    return res.status(403).json({ message: 'Invalid or expired token' });
  }

  // Add the decoded user info to the request object
  req.user = decoded;

  next();
```

```
}

// Protected endpoint (requires valid JWT)
app.get('/protected', authenticateJWT, (req, res) => {
  return res.json({ message: `Hello ${req.user.userId}, this is a protected route.` });
});

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Step 4: Environment Variables (.env)

In the .env file, store your secret key for signing JWTs. This is a crucial part of security, so do not expose this key publicly.

Create a .env file at the root of your project:

```
JWT_SECRET_KEY=your-secret-key
```

Note: Replace your-secret-key with a strong, random string.

Step 5: Test the Application

Now, you can start the server by running:

node server.js

The server will start running on http://localhost:3000.

1. Register a new user

POST request to http://localhost:3000/register

Body:

```
{
  "username": "john_doe",
  "password": "password123"
}
```

2. Login the user to get a JWT token

POST request to http://localhost:3000/login

Body:

```
{
  "username": "john_doe",
  "password": "password123"
}
```

Response:

```
{
  "message": "Login successful",
  "token":  "your-jwt-token"
}
```

3. Access a protected route using the JWT token

GET request to http://localhost:3000/protected

Header:

```
Authorization: Bearer <your-jwt-token>
```

Response:

```
{
  "message": "Hello john_doe, this is a protected route."
}
```

12. Create a react application for the student management system having registration, login, contact, about pages and implement routing to navigate through these pages.

Step 1: Create a New React Application

If you haven't already created a React app, use `create-react-app` to quickly set up the project.

```
npx create-react-app student-management
  cd student-management
```

Step 2: Install React Router

React Router will allow us to implement routing for navigation between different pages.

```
npm install react-router-dom
```

Step 3: Set Up the Project Structure

Here's how the project structure might look like after creating the necessary components:

src/ components/ About.js Contact.js Login.js Register.js App.js index.js

Step 4: Create the Components for Each Page

Now, let's create the React components for each page (`About.js`, `Contact.js`, `Login.js`, `Register.js`).

About.js

```
import React from 'react';


const About = () => {

  return (

    <div>
```

```
      <h2>About Student Management System</h2>
      <p>This is a simple student management system to manage student records.</p>
    </div>
  );
};
export default About;
```

contact.js

```
import React from 'react';

const Contact = () => {
  return (
    <div>
      <h2>Contact Us</h2>
      <p>If you have any questions, feel free to reach out to us at: contact@school.com</p>
    </div>
  );
};
export default Contact;
```

login.js

```
import React, { useState } from 'react';
const Login = () => {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
const handleLogin = (e) => {
  e.preventDefault();
  // Add login logic (e.g., authentication API call)
  console.log('Login details:', email, password);
};
return (
  <div>
    <h2>Login</h2>
    <form onSubmit={handleLogin}>
      <input
        type="email"
        placeholder="Email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        required
      />
      <input
        type="password"
        placeholder="Password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        required
      />
      <button type="submit">Login</button>
      </form>
    </div>
  );
};
export default Login;
```

register.js

```
import React, { useState } from 'react';
const Register = () => {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const handleRegister = (e) => {
  e.preventDefault();
    // Add registration logic (e.g., API call to register the user)
    console.log('Registration details:', name, email, password);
  };
  return (
    <div>
      <h2>Register</h2>
      <form onSubmit={handleRegister}>
        <input
          type="text"
          placeholder="Full Name"
          value={name}
          onChange={(e) => setName(e.target.value)}
          required
        />
        <input
          type="email"
          placeholder="Email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
          required
        />
        <input
          type="password"
          placeholder="Password"
          value={password}
          onChange={(e) => setPassword(e.target.value)}
          required
        />
        <button type="submit">Register</button>
      </form>
    </div>
  );
};
export default Register;
```

Step 5: Set Up Routing in App.js

Now that you have the components, let’s set up routing in the `App.js` file to navigate between these pages.

App.js

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch, Link } from 'react-router-dom';

import About from './components/About';

import Contact from './components/Contact';

import Login from './components/Login';

import Register from './components/Register';

const App = () => {

  return (

    <Router>

      <div>

        <nav>

          <ul>

            <li><Link to="/">Home</Link></li>

            <li><Link to="/about">About</Link></li>

            <li><Link to="/contact">Contact</Link></li>

            <li><Link to="/login">Login</Link></li>

            <li><Link to="/register">Register</Link></li>

          </ul>

        </nav>



        <Switch>
```

```
          <Route path="/" exact>
            <h2>Welcome to the Student Management System</h2>
          </Route>
          <Route path="/about" component={About} />
          <Route path="/contact" component={Contact} />
          <Route path="/login" component={Login} />
          <Route path="/register" component={Register} />
        </Switch>
      </div>
    </Router>
  );
};
export default App;
```

App.js

```
body {
  font-family: Arial, sans-serif;
  padding: 20px;
  background-color: #f9f9f9;
}


nav ul {
  list-style-type: none;
  padding: 0;
}
```

```
nav ul li {
  display: inline;
  margin-right: 10px;
}

nav ul li a {
  text-decoration: none;
  color: #333;
  font-weight: bold;
}

h2 {
  color: #333;
}

form {
  display: flex;
  flex-direction: column;
  max-width: 400px;
  margin: 0 auto;
}

input {
  padding: 10px;
  margin-bottom: 10px;
  border: 1px solid #ccc;
  border-radius: 5px;
}

button {
  padding: 10px;
  background-color: #4CAF50;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

button:hover {
  background-color: #45a049;
}
```

Step 7: Test the Application

Start the application by running:

bash

Copy

```
npm start
```

Navigate to http://localhost:3000 in your browser.

13. Create a service in react that fetches the weather information from openweathermap.org and the display the current and historical weather information using graphical representation using chart.js

Algorithm:

1. Visit the OpenWeatherMap website (https://openweathermap.org/) and click on “Sign Up” or “Log In” to create an account or log into your existing account.
2. Once logged in, navigate to your account dashboard.
3. From the dashboard, locate my API Keys section and click on “Create Key” or “API Keys” to generate a new API key.
4. Provide a name for your API key (e.g., “WeatherApp”) and click on the “Generate” or “Create” button.
5. Your API key will be generated and displayed on the screen. Make sure to copy it as we will need it later.

Step 2: Set up a new React project

1. Open your terminal or command prompt.
2. Run the following command to create a new React project

```
npm create-react-app  weather-app
```

3. Once the project is created, navigate into the project directory:

```
cd weather-app
```

Install required packages.

In the project directory, install the necessary packages by executing the following command:

```
npm install axios
```

We will use the Axios library to make HTTP requests to the OpenWeatherMap API.

Step 4: Create a Weather component.

1. Inside the “src” directory, create a new file called “Weather.js” and open it in your code editor.
2. Add the following code to define a functional component named Weather:

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';

const Weather = () => {
  const [city, setCity] = useState('');
  const [weatherData, setWeatherData] = useState(null);

  const fetchData = async () => {
    try {
      const response = await axios.get(

`https://api.openweathermap.org/data/2.5/weather?q=${city}&units=metric&appid={YOUR_API_K
EY}`
      );
      setWeatherData(response.data);
      console.log(response.data); //You can see all the weather data in console log
    } catch (error) {
      console.error(error);
    }
  };
  useEffect(() => {
    fetchData();
  }, []);

  const handleInputChange = (e) => {
    setCity(e.target.value);
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    fetchData();
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          placeholder="Enter city name"
          value={city}
          onChange={handleInputChange}
        />
        <button type="submit">Get Weather</button>
      </form>
      {weatherData ? (
        <>
          <h2>{weatherData.name}</h2>
          <p>Temperature: {weatherData.main.temp}°C</p>
          <p>Description: {weatherData.weather[0].description}</p>
          <p>Feels like : {weatherData.main.feels_like}°C</p>
          <p>Humidity : {weatherData.main.humidity}%</p>
          <p>Pressure : {weatherData.main.pressure}</p>
          <p>Wind Speed : {weatherData.wind.speed}m/s</p>
        </>
      ) : (
        <p>Loading weather data...</p>
      )}
    </div>
  );
};
export default Weather;
```

Replace `{YOUR_API_KEY}` in the API URL with the API key you generated from OpenWeatherMap.

Step 5: Connect the Weather component to your app.

1. Open the “App.js” file in the “src” directory.
2. Replace the existing code with the following code:

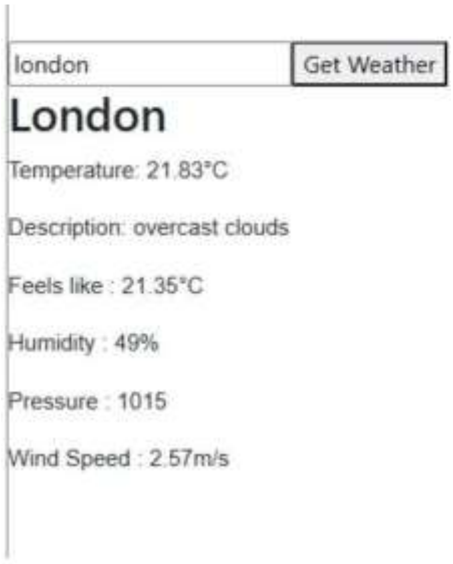
Step 5: Connect the Weather component to your app.

1. Open the “App.js” file in the “src” directory.

2. Replace the existing code with the following code:

```
import React from 'react';
import Weather from './Weather';
const App = () => {
  return (
    <div>
      <h1>Weather Forecast App</h1>
      <Weather />
    </div>
  );
};
export default App;
```

OUTPUT:



```
London
Temperature: 21.83°C
Description: overcast clouds
Feels like : 21.35°C
Humidity : 49%
Pressure : 1015
Wind Speed : 2.57m/s
```

When searched a city

14. Create a TODO application in react with necessary components and deploy it into github.

Step 1: Set Up Your Project

1. **Create a new React app**:

   Open your terminal (Command Prompt/Terminal) and run the following command to create a new React app called todo-app:

   ```
   npx create-react-app todo-app
   ```

This will set up the React development environment.

2. **Navigate to the project folder:**

Change to the newly created `todo-app` directory:

```
cd todo-app
```

Step 2: Build the Components

1. App.js - The main app component

- Open src/App.js and replace the default content with the following code:

```
import React, { useState } from 'react';
import TodoList from './components/TodoList';
import TodoForm from './components/TodoForm';

function App() {
  const [todos, setTodos] = useState([]);

  // Function to add a new todo
  const addTodo = (text) => {
    const newTodo = { id: Date.now(), text: text, completed: false };
    setTodos([...todos, newTodo]);
  };

  // Function to delete a todo
  const deleteTodo = (id) => {
    setTodos(todos.filter((todo) => todo.id !== id));
  };

  // Function to toggle the completion status of a todo
  const toggleComplete = (id) => {
    setTodos(
      todos.map((todo) =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      )
    );
  };

  return (
    <div className="App">
      <h1>Todo App</h1>
      <TodoForm addTodo={addTodo} />
      <TodoList
        todos={todos}
        deleteTodo={deleteTodo}
        toggleComplete={toggleComplete}
      />
    </div>
  );
}

export default App;
```

2. TodoForm.js - Form to add a new TODO

- In the src/components directory, create a new file called TodoForm.js and add the following code:

```
import React, { useState } from 'react';

const TodoForm = ({ addTodo }) => {
  const [input, setInput] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (input) {
      addTodo(input);
      setInput('');
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Add a new todo"
      />
      <button type="submit">Add Todo</button>
    </form>
  );
};

export default TodoForm;
```

3. TodoList.js - Displays the list of TODOs

In the src/components directory, create a new file called TodoList.js and add the following code:

```
import React from 'react';
import TodoItem from './TodoItem';
const TodoList = ({ todos, deleteTodo, toggleComplete }) => {
  return (
    <ul>
      {todos.map((todo) => (
        <TodoItem
          key={todo.id}
          todo={todo}
          deleteTodo={deleteTodo}
          toggleComplete={toggleComplete}
        />
      ))}
    </ul>
  );
};


export default TodoList;
```

4. TodoItem.js - Displays individual TODO items

In the src/components directory, create a new file called TodoItem.js and add the following code:

```
import React from 'react';
```

```
const TodoItem = ({ todo, deleteTodo, toggleComplete }) => {
  return (
    <li>
      <input
        type="checkbox"
        checked={todo.completed}
        onChange={() => toggleComplete(todo.id)}
      />
      <span
        style={{
          textDecoration: todo.completed ? 'line-through' : 'none',
        }}
      >
        {todo.text}
      </span>
      <button onClick={() => deleteTodo(todo.id)}>Delete</button>
    </li>
  );
};
export default TodoItem;
```

Step 3: Add Some Basic Styles

Open the src/App.css file and add the following styles to make the app look better:

```
.App {
  text-align: center;
  font-family: Arial, sans-serif;
}
form {
  margin-bottom: 20px;
}
input {
  padding: 8px;
  margin-right: 10px;
  width: 200px;
}
button {
  padding: 8px;
  background-color: #4CAF50;
  color: white;
  border: none;
  cursor: pointer;
}
```

```
button:hover {
  background-color: #45a049;
}
ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin: 10px 0;
}
span {
  flex: 1;
  margin-left: 10px;
}
```

Step 4: Run the Application Locally

To see the app in action locally, run the following command in your terminal:

```
npm start
```

This should open the app in your browser at http://localhost:3000.

Step 5: Push to GitHub

1. Create a new GitHub repository

Go to GitHub and create a new repository (e.g., todo-app).

2. Initialize a Git repository locally

In your terminal, navigate to the todo-app folder and run the following commands:

```
git init

git add .

git commit -m "Initial commit"
```

3. Link your local repo to GitHub

Add the GitHub repository URL as a remote (replace <your-repository-url> with your actual GitHub repository URL):

```
git remote add origin <your-repository-url>
```

4. Push your code to GitHub

Now push the code to GitHub:

```
git push -u origin master
```

Step 6: Deploy the Application to GitHub Pages

1. Install gh-pages package

To deploy the app to GitHub Pages, install the gh-pages package:

```
npm install gh-pages --save-dev
```

2. Update package.json

Open the package.json file and add the following line to specify the homepage URL for GitHub Pages:

```
"homepage":  "https://<username>.github.io/<repo-name>"
```

Replace <username> with your GitHub username and <repo-name> with your repository name.

Then, in the scripts section of package.json, add these two deploy-related scripts:

```
"scripts": {
  "predeploy": "npm run build",
  "deploy": "gh-pages -d build"
}
```

3. Deploy the app

Now, run the following command to deploy your app to GitHub Pages:

```
npm run deploy
```

Step 7: View the Deployed Application

After the deployment is complete, go to the URL specified in the homepage field of package.json. The app should now be live at:

https://<username>.github.io/<repo-name>