1

What is HTTP? Explain the structure of HTTP request message.

What is HTTP?

HTTP (HyperText Transfer Protocol) is the protocol used for transferring data over the web. It defines how messages are formatted and transmitted between clients (browsers) and servers. HTTP operates over the **TCP/IP protocol** and uses a request-response model.

Structure of HTTP Request Message

An HTTP request consists of:

1. **Request Line**:
   - Contains the **HTTP Method** (GET, POST, etc.), **Request URI** (path to the resource), and **HTTP Version** (e.g., HTTP/1.1).

   Example:

   ```
   GET /index.html HTTP/1.1
   ```

2. **Headers**:
   - Key-value pairs providing additional information, like **Host**, **User-Agent**, **Accept**, etc.

   Example:

   ```
   Host: www.example.com
   User-Agent: Mozilla/5.0
   Accept: text/html
   ```

3. **Empty Line**:
   - A blank line separating headers from the body.
4. **Body (Optional)**:
   - Contains data sent to the server, used with methods like **POST** or **PUT**.

   Example:

```
name=JohnDoe&email=john@example.com
```

Complete HTTP Request Example:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

2

1.b. Create a Placement form with the following controls:
a. A text box to collect the student's name and SRN
b. A multiline input field to collect college name and address
c. A student should give choice to tick their department from a list of departments i. CSE
ii. ECE
iii. MECH
d. A collection of three radio buttons for Semester option that are labelled as follows:
i. IV
ii. V
iii.VI
e. Submit and reset Button

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Placement Form</title>
</head>
<body>

<h2>Placement Form</h2>

<form action="submit_form.php" method="post">

    <!-- Student Name and SRN -->
    <label for="name">Student Name:</label>
    <input type="text" id="name" name="name" placeholder="Enter your name"
required>
```

```
    <br><br>

    <label for="srn">SRN:</label>
    <input type="text" id="srn" name="srn" placeholder="Enter SRN" required>
    <br><br>

    <!-- College Name and Address -->
    <label for="college">College Name and Address:</label><br>
    <textarea id="college" name="college" rows="4" cols="50"
placeholder="Enter your college name and address" required></textarea>
    <br><br>

    <!-- Department Selection (Checkboxes) -->
    <label>Department:</label><br>
    <input type="checkbox" id="cse" name="department" value="CSE">
    <label for="cse">CSE</label><br>
    <input type="checkbox" id="ece" name="department" value="ECE">
    <label for="ece">ECE</label><br>
    <input type="checkbox" id="mech" name="department" value="MECH">
    <label for="mech">MECH</label>
    <br><br>

    <!-- Semester Selection -->
    <label>Semester:</label><br>
    <input type="radio" id="iv" name="semester" value="IV" required>
    <label for="iv">IV</label>
    <input type="radio" id="v" name="semester" value="V">
    <label for="v">V</label>
    <input type="radio" id="vi" name="semester" value="VI">
    <label for="vi">VI</label>
    <br><br>

    <!-- Submit and Reset Buttons -->
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">

</form>

</body>
</html>
```

Explanation of the Controls:

1. **Student Name and SRN**:

- Text input fields to collect the student's name and SRN (Student Registration Number). Both fields are required.

2. **College Name and Address**:
   - A multiline input field (`<textarea>`) to collect the college name and address. This allows the user to enter multiple lines of text.
3. **Department Selection (Checkboxes)**:
   - The departments (CSE, ECE, MECH) are provided as checkboxes, allowing students to select one or more departments.
4. **Semester Selection (Radio Buttons)**:
   - A group of radio buttons for selecting the semester (IV, V, VI). Radio buttons are used because only one semester can be chosen.
5. **Submit and Reset Buttons**:
   - A submit button to submit the form data and a reset button to clear all the form fields.

How It Works:

- The user fills in their name, SRN, college details, selects departments using checkboxes, and chooses a semester via radio buttons.
- Upon clicking **Submit**, the form will be submitted to `submit_form.php` for processing (you would need a back-end to handle the submission).
- The **Reset** button clears all the input fields.

3

1.c. Describe the structure of HTTP Request and Response message using an example.

Structure of HTTP Request and Response Messages

HTTP (HyperText Transfer Protocol) facilitates communication between clients (typically browsers) and servers. It works by sending **requests** from the client to the server and receiving **responses**. Both the **request message** and **response message** follow a specific structure.

1. HTTP Request Message Structure

An HTTP request is sent by the client (browser) to the server to request a resource (e.g., a web page or file). It consists of the following components:

a. Request Line

- **HTTP Method**: Defines the action the client wants to perform on the resource.
  - Example: `GET`, `POST`, `PUT`, `DELETE`, etc.

- **Request URI**: Specifies the resource being requested (can include query parameters).
  - Example: `/home`, `/index.html`, `/search?q=term`
- **HTTP Version**: Specifies the version of the HTTP protocol being used.
  - Example: `HTTP/1.1`, `HTTP/2`

Example of Request Line:

```
GET /index.html HTTP/1.1
```

b. Headers

Headers provide additional information about the request. They are sent as **key-value pairs** and can include:

- **Host**: Specifies the domain of the server (required in HTTP/1.1).
- **User-Agent**: Identifies the client (browser) making the request.
- **Accept**: Specifies the type of content the client can handle (e.g., HTML, JSON).
- **Content-Type**: Specifies the type of content in the request body (used in `POST` or `PUT` requests).
- **Authorization**: Contains credentials for HTTP authentication.

Example of Headers:

```
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36
Accept: text/html
```

c. Empty Line

An empty line separates the headers from the body (if there is one). It marks the end of the headers section.

d. Body (Optional)

The body contains the data sent by the client (e.g., form data, file upload). This is typically present in `POST`, `PUT`, or `PATCH` requests.

Example Body:

```
name=JohnDoe&email=john@example.com
```

Example of a Complete HTTP Request:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36
Accept: text/html
```

2. HTTP Response Message Structure

An HTTP response is sent by the server to the client after processing the request. It consists of the following components:

a. Status Line

- **HTTP Version**: The version of HTTP being used.
- **Status Code**: A 3-digit code indicating the result of the server's processing.
  - Example: `200 OK`, `404 Not Found`, `500 Internal Server Error`.
- **Status Message**: A brief description of the status code.

Example of Status Line:

```
HTTP/1.1 200 OK
```

b. Headers

Similar to request headers, response headers provide metadata about the response, such as:

- **Content-Type**: Specifies the type of the response body (e.g., `text/html`, `application/json`).
- **Content-Length**: Indicates the length of the response body.
- **Date**: The date and time the response was sent.
- **Server**: Information about the web server software handling the request.

Example of Response Headers:

```
Content-Type: text/html; charset=UTF-8
Content-Length: 1234
Date: Sat, 15 Dec 2024 10:00:00 GMT
Server: Apache/2.4.41 (Unix)
```

c. Empty Line

Just like the request message, an empty line separates the headers from the body.

d. Body (Optional)

The body contains the actual data returned by the server (e.g., HTML content, images, JSON). If the request is for a webpage, the body contains the HTML code.

Example Body (HTML content):

```
<html>
  <head>
    <title>Example Page</title>
  </head>
  <body>
    <h1>Welcome to Example.com</h1>
  </body>
</html>
```

Example of a Complete HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 1234
Date: Sat, 15 Dec 2024 10:00:00 GMT
Server: Apache/2.4.41 (Unix)

<html>
  <head>
    <title>Example Page</title>
  </head>
  <body>
    <h1>Welcome to Example.com</h1>
  </body>
</html>
```

Summary of the Structure:

- **HTTP Request**:
    1. **Request Line**: Method, URI, and version.
    2. **Headers**: Additional metadata.
    3. **Empty Line**: Separates headers and body.
    4. **Body (Optional)**: Data sent by the client.
- **HTTP Response**:
    1. **Status Line**: Version, status code, and message.
    2. **Headers**: Metadata about the response.
    3. **Empty Line**: Separates headers and body.
    4. **Body (Optional)**: Data returned by the server.

4

1.d. With a neat diagram, explain the CSS Box model and its significance

CSS Box Model

The **CSS Box Model** is a fundamental concept in web design that defines how elements are structured and how their dimensions are calculated on a webpage. It consists of several parts that control the spacing, padding, borders, and the actual content of an element.

Parts of the CSS Box Model:

1. **Content**: This is where the actual content of the element (text, images, etc.) is displayed. Its dimensions (width and height) are defined by the element's content.
2. **Padding**: This is the space between the content and the border. Padding is used to create space inside the element, around the content.
3. **Border**: The border wraps around the padding (if any) and the content. It can be styled with width, color, and type (solid, dashed, etc.).
4. **Margin**: This is the outermost space around the element. It separates the element from other elements on the page. It does not have any background color and is transparent.

Diagram of the CSS Box Model

Here’s a visual representation of the box model:

```
+----------------------+
|        Margin        |
```

```
|  +--------------------+ |
|  |      Border        | |
|  |  +--------------+  | |
|  |  |   Padding    |  | |
|  |  |  +--------+  |  | |
|  |  |  | Content |  |  | |
|  |  |  +--------+  |  | |
|  |  |  +--------------+  | |
|  +--------------------+ |
+-------------------------+
```

Explanation of Each Part:

1. **Content**:
   - This is where the actual content of the element is placed. For example, text, images, or other types of media. The width and height of the content area are specified by the `width` and `height` properties in CSS.
2. **Padding**:
   - Padding adds space inside the element between the content and the border. You can specify padding on all four sides of the element, or for specific sides (top, right, bottom, left).
   - **CSS Example**:

     ```css
     padding: 10px;  /* Uniform padding on all sides */
     padding-top: 5px;  /* Specific padding on the top */
     ```

3. **Border**:
   - A border surrounds the padding (if present) and content area. It has a width, color, and style (such as solid, dashed, etc.). The border's width is added to the overall size of the element.
   - **CSS Example**:

     ```css
     border: 2px solid black;  /* Creates a solid black border */
     ```

4. **Margin**:
   - Margin is the outermost space, providing distance between the element and other surrounding elements. Margins are transparent and do not have a background color.
   - **CSS Example**:

```
margin: 20px;  /* Adds 20px of space outside the element */
```

Significance of the CSS Box Model:

- **Layout Control**: The box model allows you to control how elements are sized and spaced on a page. Understanding the box model helps ensure that your layouts appear as expected, with proper spacing and alignment.
- **Total Element Size**: The width and height properties define the content area, but the total size of an element also includes padding, borders, and margins. Knowing how each of these components contributes to the element's total size is critical for building responsive and well-spaced layouts.
- **Overflow Management**: By managing padding, borders, and margins, you can control how content overflows from its container and ensure it fits within the available space.
- **Responsive Design**: The box model helps in creating flexible layouts that can adapt to different screen sizes by adjusting padding, borders, and margins dynamically.

Example of Box Model in CSS:

```
.box {
  width: 200px;          /* Content area width */
  height: 150px;         /* Content area height */
  padding: 20px;         /* Space between content and border */
  border: 5px solid red; /* Border around the element */
  margin: 10px;          /* Space outside the element */
}
```

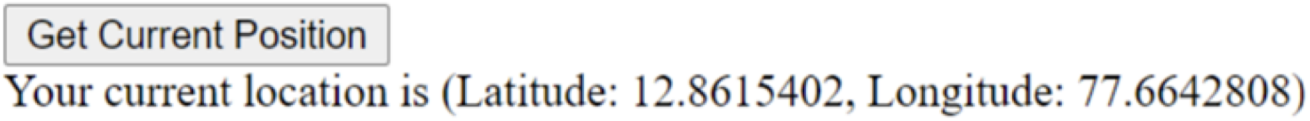
Conclusion:

The **CSS Box Model** is essential for building structured and visually consistent web pages. It defines how space is managed within and around elements, providing developers with the tools to control layout, spacing, and alignment. By understanding and manipulating the box model, web designers can create more intuitive, aesthetically pleasing, and responsive user interfaces.

5

. What is Geo Location? Explain the two methods of navigator.geolocation object?
Write a program to display your current location on the browser when the button “Get Current

Position” is clicked as shown below.



ans:

- Enables your web application to obtain the geographical position of your website visitors
- The user has to accept to share their location,
- The Geolocation API is work with the navigator.geolocation object.
- Accessed via JavaScript, through the **navigator.geolocation** object.

- navigator.geolocation object allows you to access geo location through two primary functions:
  - getCurrentPosition()
    - Returns the location of the visitor as a one-time snapshot
  - watchPosition()
    - returns the location of the visitor every time the location changes

Syntax : getCurrentPosition(success, error, options)

- Both functions take the following parameters:
  - Success callback function
  - Error callback function (optional)
  - Geo location options object (optional)
  - If the getCurrentPosition() method is successful, it returns a coordinates object to the function specified in the parameter (success)

HTML5 – Geo Location
Introduction

The Geolocation API uses three methods of Geolocation interface which are given following:

Methods	Description
getCurrentPosition()	It identifies the device or the user's current location and returns a position object with data.
watchPosition()	Return a value whenever the device location changes.
clearWatch()	It cancels the previous watchPosition() call

HTML5 – Geo Location
Code Example

```
navigator.geolocation.getCurrentPosition(showPosition);
function showPosition(position) {
    x.innerHTML = "Latitude: " + position.coords.latitude +
    "<br>Longitude: " + position.coords.longitude;
  }

navigator.geolocation.getCurrentPosition( function(position) {
    console.log("your position is: " + position.coords.latitude + ", " +
    position.coords.longitude);
} );
```

* getCurrentPosition can be replaced with watchPosition if you need to receive updates to location

```
<html>
    <head>
        <title> Geo Location</title>
        <script>
            function getpos(){
                navigator.geolocation.getCurrentPosition(show,error);
            }
            function show(position){
                ps=position;
                console.log("Current Position"+ps.coords.latitude+" "+ps.coords.longitude);
            }
           function error(error){
                e=error;
                console.log(e);
            }
        </script>
    </head>

    <body>
        <button onclick="getpos()"> Get Current Position</button>
    </body>
</html>
```

```
<html>
        <head>
                <title>My first HTML document</title>


                <title> GeoLocation</title>
                <style>

                </style>
                <script type="text/javascript">
                function getpos(){
                        navigator.geolocation.watchPosition(show,error);
                        }
                function show(position){
                        ps=position;
                        document.write("Current Position:"+position.coords.latitude+ "
"+position.coords.longitude);
                        }
                function error(error) {
                        e=error;
                                                }

                </script>

        </head>
        <body>
        <button onclick="getpos()"> Get Current Position</button>

        </body>



</html>
```

6

What is AJAX? Explain in detail any 3 XHR object properties.

What is AJAX?

AJAX (Asynchronous JavaScript and XML) is a set of web development techniques that allow web pages to communicate with servers asynchronously (in the background) without needing to reload the entire page. It allows for more dynamic and responsive web applications, enabling content to be updated, retrieved, or sent to a server without the need for a full page reload.

AJAX typically uses the **XMLHttpRequest (XHR)** object or the newer **Fetch API** to send and receive data from the server. Though originally designed to work with XML, AJAX now commonly works with **JSON**, **HTML**, and other formats.

Key Benefits of AJAX:

- **Asynchronous communication**: The page can continue to function while the data is being retrieved from the server.
- **Improved User Experience**: Reduces waiting time, as only parts of the web page need to be updated.
- **Reduced Server Load**: Only relevant data is sent/received, which can help reduce server load.

Three Key XHR Object Properties:

1. `readyState`:

    - This property represents the current state of the `XMLHttpRequest` (XHR) object. It holds an integer value from 0 to 4, indicating the progress of the request.

`readyState`	**Description**
`0`	Request not initialized.
`1`	Server connection established.
`2`	Request received.
`3`	Processing request.
`4`	Request complete and response is ready.

    Example:

```
if (xhr.readyState === 4) {
    console.log('Request completed');
}
```

    The `readyState` property is typically used in combination with the `onreadystatechange` event to determine the status of the request and handle the response once the request is complete.

---

2. `status`:

- The `status` property returns the HTTP status code from the server's response to the request. This is an integer value that helps determine if the request was successful or if there was an error.

Common HTTP status codes:

- `200`: OK (Request successful)
- `404`: Not Found (Requested resource not found)
- `500`: Internal Server Error (Server issue)

Example:

```
if (xhr.status === 200) {
    console.log('Request was successful');
} else {
    console.log('Request failed with status: ' + xhr.status);
}
```

This property is useful for checking if the request was completed successfully or if there were issues during the request/response cycle.

---

3. **`responseText`**:

- The `responseText` property contains the response data as a string (usually used for plain text or HTML content). It holds the text returned by the server after the request is processed.

If the server returns data in a different format (e.g., JSON), you may need to parse it using `JSON.parse()`.

Example:

```
xhr.onreadystatechange = function() {
    if (xhr.readyState === 4 && xhr.status === 200) {
        console.log(xhr.responseText); // Logs the response data
    }
};
```

Note: If the response is in JSON format, you would use `xhr.responseText` in combination with `JSON.parse()` to convert the string into a JavaScript object.

```
let responseObject = JSON.parse(xhr.responseText);
```

Example of Using XHR Object

```
var xhr = new XMLHttpRequest(); // Create a new XHR object

// Define the request
xhr.open("GET", "https://api.example.com/data", true); // Request type: GET,
URL: api.example.com/data

// Set up the event listener for the 'readystatechange' event
xhr.onreadystatechange = function() {
    if (xhr.readyState === 4) { // Check if the request is complete
        if (xhr.status === 200) { // Check if the status is OK
            console.log(xhr.responseText); // Print the response text
(usually JSON or HTML)
        } else {
            console.log("Request failed with status: " + xhr.status);
        }
    }
};

// Send the request
xhr.send();
```

Summary:

- **AJAX** allows for asynchronous web requests, providing a more dynamic and responsive web experience.
- The `XMLHttpRequest` **(XHR)** object is used to send and receive requests asynchronously.
- Key properties of the XHR object include:
  - `readyState`: The current state of the request.
  - `status`: The HTTP status code from the server response.
  - `responseText`: The text returned from the server in response to the request.

7

2.c. Give the difference between JSON and XML.
Represent the details of three 6th sem students both in JSON and XML. Consider the attributes id, name, branch, CGPA

Difference Between JSON and XML:

Feature	JSON (JavaScript Object Notation)	XML (Extensible Markup Language)
Format	Lightweight data-interchange format, easy for humans and machines to read.	A markup language that uses tags to describe data.
Syntax	Uses a key-value pair format, similar to JavaScript objects.	Uses tags to define elements and attributes.
Data Structure	Supports simple structures like objects (key-value pairs) and arrays (lists).	Supports complex hierarchical structures through nested tags.
Readability	More readable and concise.	Less readable and more verbose due to tags.
Data Types	Can store strings, numbers, arrays, booleans, and null.	Can store only text data. Additional data types must be manually defined.
Data Validation	Limited data validation is supported (through JSON Schema).	Supports data validation (through DTD, XML Schema).
Namespaces	No concept of namespaces.	Supports namespaces to avoid element name conflicts.
Support for Comments	No support for comments.	Supports comments using `<!-- comment -->`.
File Size	Generally smaller due to less overhead.	Can be larger because of the additional markup.
Parsing	Easier to parse, especially in JavaScript using `JSON.parse()`.	Requires more complex parsing and libraries (e.g., DOM, SAX).

Representation of Three 6th Semester Students in JSON and XML

1. JSON Representation:

```
[
  {
```

```
    "id": "101",
    "name": "John Doe",
    "branch": "CSE",
    "CGPA": "8.5"
  },
  {
    "id": "102",
    "name": "Jane Smith",
    "branch": "ECE",
    "CGPA": "9.1"
  },
  {
    "id": "103",
    "name": "Robert Brown",
    "branch": "MECH",
    "CGPA": "7.8"
  }
]
```

2. XML Representation:

```
<students>
  <student>
    <id>101</id>
    <name>John Doe</name>
    <branch>CSE</branch>
    <CGPA>8.5</CGPA>
  </student>
  <student>
    <id>102</id>
    <name>Jane Smith</name>
    <branch>ECE</branch>
    <CGPA>9.1</CGPA>
  </student>
  <student>
    <id>103</id>
    <name>Robert Brown</name>
    <branch>MECH</branch>
    <CGPA>7.8</CGPA>
  </student>
</students>
```

Summary:

- **JSON** is a simpler and more lightweight format for representing data with a clear structure using key-value pairs.
- **XML** is more verbose and uses tags to represent the structure of the data, supporting complex hierarchical relationships.
- Both formats can represent the same data but differ in structure and syntax, with JSON being more concise and XML offering greater flexibility in terms of validation and metadata.

8

3.a. Create a simple server (use http request GET method) that will allow you to display an HTML file on the browser given the pathname. Set default pathname to index.html. Handle errors by displaying file not found message

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

// Create the server
const server = http.createServer((req, res) => {
  // Check if the request method is GET
  if (req.method === 'GET') {
    // Get the pathname from the URL or default to 'index.html'
    let filePath = req.url === '/' ? '/index.html' : req.url;

    // Set the full path to the file
    const fullPath = path.join(__dirname, filePath);

    // Check if the file exists
    fs.readFile(fullPath, (err, data) => {
      if (err) {
        // If file is not found, send a 404 response
        res.statusCode = 404;
        res.setHeader('Content-Type', 'text/plain');
        res.end('File not found');
      } else {
        // If file is found, serve it with correct content type
        res.statusCode = 200;
        res.setHeader('Content-Type', 'text/html');
        res.end(data);
      }
    });
  } else {
    // Handle any non-GET requests (optional, since we are only concerned
```

```
with GET here)
    res.statusCode = 405; // Method Not Allowed
    res.setHeader('Content-Type', 'text/plain');
    res.end('Only GET requests are allowed');
  }
});

// Set the server to listen on port 3000
server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

Explanation:

1. **HTTP Method**: The server only handles `GET` requests. If a non-`GET` request is received, it sends a `405 Method Not Allowed` response.
2. **Default File**: When the request URL is `/`, it serves `index.html` as the default file.
3. **File Not Found**: If the requested file is not found, it returns a `404 File not found` error.
4. **Server Listening**: The server listens on port `3000`.

9

3.b. With a neat diagram explain component life cycle.

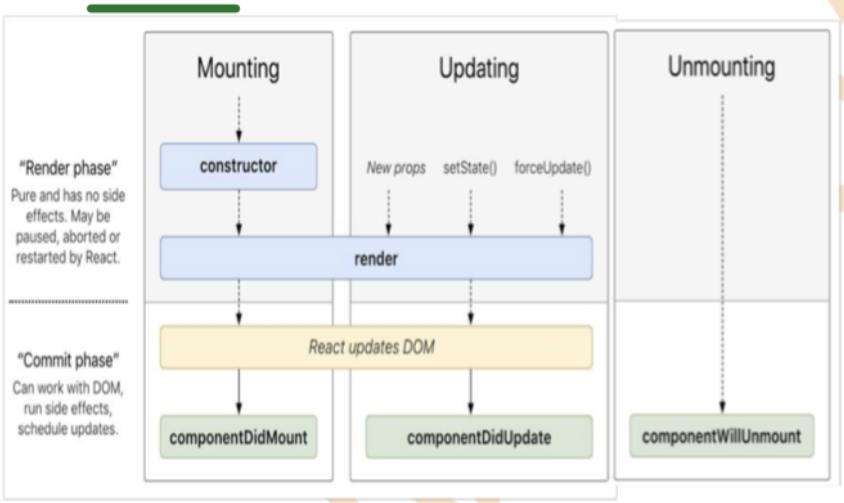
Life Cycle Methods

The series of events that happen from the starting of a React component to its ending. Every component in React should go through the following lifecycle of events.

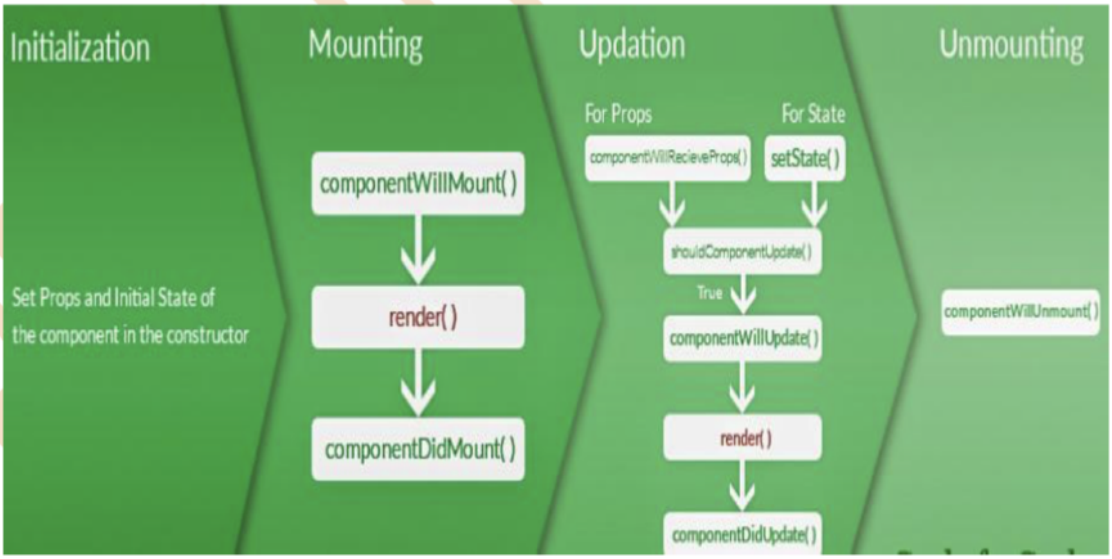
Mounting - Birth of the Component

Updating- Growing of component

Unmounting- End of the component



Functions defined in every phase is more clear with the below diagram



Functions/Methods in detail

1. constructor() : Premounting

This function is called before the component is mounted. Implementation requires

calling of super() so that we can execute the constructor function that is inherited from React.Component while adding our own functionality. Supports Initializing the state and binding our component

```
        constructor() {
  super()
  this.state = {
    key: "value"
  }
}
```

It is possible to use the constructor to set an initial state that is dependent upon props. Otherwise, this.props it will be undefined in the constructor, which can lead to a major error in the application.

```
constructor(props) {
  super(props);
  this.state = {
    color: props.initialColor
  };
}
```

2. componentWillMount()

This is called only once in the component lifecycle, immediately before the component is rendered. Executed before rendering, on both the server and the client side. Suppose you want to keep the time and date of when the component was created in your component state, you could set this up in componentWillMount.

```
componentWillMount() {
  this.setState({ startDateTime: new Date(Date.now())});
}
```

3. render()

Most useful life cycle method as it is the only method that is required. Handles the rendering of component while accessing this.state and this.props

4. **componentDidMount()**

Function is called once only, but immediately after the render() method has taken place. That means that the HTML for the React component has been rendered into the DOM and can be accessed if necessary. This method is used to perform any DOM

manipulation of data-fetching that the component might need.

The best place to initiate API calls in order to fetch data from remote servers. Use setState which will cause another rendering but It will happen before the browser updates the UI. This is to ensure that the user won't see the intermediate state. AJAX requests and DOM or state updates should occur here. Also used for integration with other JavaScript frameworks like Node.js and any functions with late execution such as **setTimeout** or **setInterval**

5. componentWillReceiveProps()

Allows us to match the incoming props against our current props and make logical. We get our current props by calling this.props and the new value is the nextProps argument passed to the method. It is invoked as soon as the props are updated before another render method is called.

6. **shouldComponentUpdate()**

Allows a component to exit the Update life cycle if there's no reason to use a replacement render. It may be a no-op that returns true. Means while updating the component, we'll re-render.

7. componentWillUpdate()

Called just before the rendering

8. componentDidUpdate()

Is invoked immediately after updating occurs. Not called for the initial render. Will not be invoked if **shouldcomponentUpdate() returns false**.

9. **componentWillUnmount()**

The last function to be called immediately before the component is removed from the DOM. It is generally used to perform clean-up for any DOM-elements or timers created in componentWillMount.

What is the significance of key property? Describe with an example

Introduction to Keys

A key is a unique identifier which helps to identify which items have changed, added, or removed. Useful when we dynamically created components or when users alter the lists. The best way to pick a key is to choose a string that uniquely identifies the items in the list. Keys used within arrays should be unique among their siblings. However, they **don't need to be globally unique.** Also helps in efficiently updating the DOM.

Coding example 6: Consider an array containing n elements in it. Display these elements using n bullet items in an unordered list

```
<script type = "text/babel">
        const arr = ["book1","book2","book3", ","book4"]
```



```
        function Booklist(props)
                {       return (<ul> <li >{props.books[0]}</li>
                                <li >{props.books[1]}</li>
                                <li >{props.books[2]}</li>
                                <li >{props.books[3]}</li>
                        </ul>)
                }
        ReactDOM.render(<Booklist books = {arr}/>,document.getElementById("root"))
</script>
```

Observation: As and when the number of elements changes in the array, the code runs into trouble. Refer to the below code to have this dynamism.

Coding example 7:

```
 <script type = "text/babel">
        function Booklist(props)
        {
                const book_lists= props.books
                //console.log(book_lists)
                const b = book_lists.map((book,index) => <li key = {index}
>{book}</li>)
                return <ul>{b}</ul>
        }
        ReactDOM.render(<Booklist books = {arr}/>,document.getElementById("root"))
</script>
```

11

4.a. What is RESTful API? Explain any 4 design specification/constraints of REST API.

REST API's:

REST (short for representational state transfer) is an architectural pattern for application programming interfaces (APIs).

Resource Based

- The APIs are resource based (as opposed to action based). APIs are formed by a combination of resources and actions.
- Resources are accessed based on a Uniform Resource Identifier (URI), also known as an *endpoint*. Resources are nouns (not verbs).
- Resources can also form a hierarchy. For example, the collection of orders of a customer is identified by /customers/1234/orders, and an order of that customer is identified by /customers/1234/orders/43.

HTTP Methods as Actions

- To access and manipulate the resources, you use HTTP methods. While resources were nouns, the HTTP methods are verbs that operate on them. They map to CRUD (Create, Read, Update, Delete) operations on the resource. Tables 5-1 shows commonly used mapping of CRUD operations to HTTP methods and resources.

Table 5-1. *CRUD Mapping for Collections*

Operation	HTTP Method	Resource	Example	Remarks
Read – List	GET	Collection	GET /customers	Lists objects (additional query string can be used to filter)
Read	GET	Object	GET /customers/1234	Returns a single object (query string may be used to filter fields)
Create	POST	Collection	POST /customers	Creates an object, and the object is supplied in the body.
Update	PUT	Object	PUT /customers/1234	Replaces the object with the object supplied in the body.
Update	PATCH	Object	PATCH /customers/1234	Modifies some attributes of the object, specification in the body.
Delete	DELETE	Object	DELETE /customers/1234	Deletes the object

- Two important concepts about the HTTP methods are *safety* and *idempotency* of the methods.
- A safe method is one whose results can be cached. Thus, GET, HEAD, and OPTIONS are safe methods; you can call them any number of times and get the same results.
- An idempotent method is one that has the same effect when called multiple times. Note that it's not the same result; instead it's the effect on the resource.
- A safe method is always idempotent, but not the other way round.
- Only PUT and DELETE are idempotent, whereas POST and PATCH are not.
- If you use PUT multiple times, you continue to replace the same resource with the *same* new contents, thus the outcome is the same for each attempt. DELETE, if seen as "let the resource not exist,".
- PATCH and PUT are different, even though both can be used to update a resource.
- PATCH is used to modify a resource by adding to an array in the resource. PUT is used to completely replace the resource.

What is a RESTful API?

RESTful API (Representational State Transfer) is an architectural style for designing networked applications. It is based on a set of principles and constraints that allow for scalable and stateless communication between client and server, often using HTTP. RESTful APIs are widely used in web services due to their simplicity, scalability, and ease of integration with web applications.

A **RESTful API** allows clients (such as web browsers, mobile apps, or other services) to interact with servers by sending requests and receiving responses. The communication between client and server typically happens over HTTP, and the data is usually represented in JSON or XML formats.

Four Design Specifications/Constraints of REST API

RESTful APIs follow several constraints to ensure simplicity and scalability. Here are four important constraints:

1. **Statelessness**:

    - **Explanation**: In REST, each request from the client to the server must contain all the information the server needs to fulfill the request. The server does not store any client context between requests. Each request is independent, and the server treats each request as if it is being made for the first time.
    - **Significance**: This ensures scalability because the server doesn't need to remember previous interactions. It reduces the server's load and makes it easier to scale the application horizontally.

    Example: A client might request information about a product with the following GET request:

    ```
    GET /products/12345
    ```

    The request contains all the information necessary for the server to process it, without needing to remember any previous interactions.
2. **Client-Server Architecture**:

    - **Explanation**: REST is based on the separation of concerns between the client and the server. The client is responsible for the user interface and user experience, while

the server handles data storage and business logic. The server and client communicate through a stateless protocol like HTTP.
- **Significance**: This separation allows both the client and server to be developed and scaled independently. The client can be built with different technologies (e.g., web browsers, mobile apps), while the server remains unchanged.

Example: The client (e.g., a web browser) sends a request to the server for data, and the server responds with the requested data (e.g., JSON or HTML).

3. **Cacheability**:

- **Explanation**: In REST, responses from the server can be explicitly marked as cacheable or non-cacheable. If a response is cacheable, the client or intermediary servers (such as proxies or CDNs) can store the response data for reuse in future requests, reducing server load and improving performance.
- **Significance**: Cacheability enhances the performance and scalability of applications by reducing the need for repeated requests for the same data. It also improves the overall user experience by reducing latency.

Example: A response for fetching user data could include headers like `Cache-Control: max-age=3600` to indicate that the response can be cached for one hour.

4. **Uniform Interface**:

- **Explanation**: A key principle of REST is the uniformity of the interface. This means that the same set of HTTP methods (GET, POST, PUT, DELETE, etc.) are used consistently across different resources. The structure of the API should be standardized, making it easy for developers to understand and use.
- **Significance**: The uniform interface simplifies the learning curve for developers and improves the maintainability of the system. It allows for predictable and consistent interactions between the client and the server.

Example: A RESTful API for managing resources might follow standard HTTP methods:

- `GET /users`: Retrieve a list of users
- `GET /users/{id}`: Retrieve a specific user by ID
- `POST /users`: Create a new user
- `PUT /users/{id}`: Update an existing user
- `DELETE /users/{id}`: Delete a user

Summary of Key RESTful API Constraints:

1. **Statelessness**: Each request is independent and contains all the necessary information for processing.
2. **Client-Server Architecture**: Separates client concerns (UI) and server concerns (data and logic).
3. **Cacheability**: Responses can be cached to improve performance.
4. **Uniform Interface**: A consistent set of HTTP methods and resource URLs for a predictable interaction.

12

4.b. Write server-side script in JavaScript to route requests for GET and POST requests for flight details. The details are stored in the mongodb database in the following format.
{from:"BLR", to:"DEL", dept:"12:25", arrv:"14:25", flnum:"6E-2428"}
The server script should support the following routes:
- GET /flights – return details of all flights
- GET /flights/:from/:to – return details of flight between specific airports
- POST /flights – save details of a flight and return a success or error message

```
// Import required modules
const express = require('express');
const { MongoClient } = require('mongodb');
const bodyParser = require('body-parser');

// Initialize the app
const app = express();
const port = 3000;

// MongoDB URI
const uri = 'mongodb://localhost:27017';

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

// MongoDB client
let db;

// Connect to MongoDB
MongoClient.connect(uri)
  .then(client => {
    db = client.db('flightDB'); // Select the database
    console.log('Connected to MongoDB');
```

```
  })
  .catch(err => {
    console.error('Failed to connect to MongoDB', err);
  });

// GET all flights
app.get('/flights', async (req, res) => {
  try {
    const flights = await db.collection('flights').find().toArray();
    res.status(200).json(flights);
  } catch (err) {
    res.status(500).json({ message: 'Error fetching flights', error: err });
  }
});

// GET flight details between specific airports
app.get('/flights/:from/:to', async (req, res) => {
  const { from, to } = req.params;
  try {
    const flight = await db.collection('flights').findOne({ from: from, to:
to });
    if (flight) {
      res.status(200).json(flight);
    } else {
      res.status(404).json({ message: 'No flight found between the specified
airports' });
    }
  } catch (err) {
    res.status(500).json({ message: 'Error fetching flight details', error:
err });
  }
});

// POST flight details
app.post('/flights', async (req, res) => {
  const { from, to, dept, arrv, flnum } = req.body;

  // Validate required fields
  if (!from || !to || !dept || !arrv || !flnum) {
    return res.status(400).json({ message: 'All fields are required' });
  }

  try {
    const newFlight = { from, to, dept, arrv, flnum };
    const result = await db.collection('flights').insertOne(newFlight);
    res.status(201).json({ message: 'Flight details saved successfully',
```

```
flight: result.ops[0] });
  } catch (err) {
    res.status(500).json({ message: 'Error saving flight details', error:
err });
  }
});

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

13

4.c. Explain the role of express middleware function. What are the different types of middleware an express application can use?

Role of Express Middleware Function

In **Express.js**, **middleware functions** are functions that are executed during the **request-response cycle**. Middleware has access to the **request object** (`req`), the **response object** (`res`), and the **next middleware function** in the application's request-response cycle.

The primary roles of middleware in Express are:

1. **Modify Request/Response Objects**: Middleware can modify incoming request data or outgoing response data before reaching the route handler.
2. **Execute Code**: Middleware can execute code for tasks like logging, authentication, or data parsing.
3. **Terminate Requests**: Middleware can send a response and terminate the request-response cycle (e.g., error handling).
4. **Call Next Middleware**: Middleware can pass control to the next middleware in the stack using the `next()` function.

Types of Middleware in Express Applications

Express provides the flexibility to use various types of middleware. These are:

1. **Application-Level Middleware**

2. **Router-Level Middleware**
3. **Built-in Middleware**
4. **Third-Party Middleware**
5. **Error-Handling Middleware**

---

1. Application-Level Middleware

Application-level middleware is bound to an Express application instance using `app.use()` or `app.METHOD()`.

- Example: **Logging Middleware**

```
const express = require('express');
const app = express();

// Application-level middleware
app.use((req, res, next) => {
  console.log(`Request Method: ${req.method}, URL: ${req.url}`);
  next(); // Pass control to the next middleware
});

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

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

---

2. Router-Level Middleware

Router-level middleware is bound to an instance of `express.Router()` and works like application-level middleware but is specific to routes.

- Example:

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

```
// Middleware specific to this router
router.use((req, res, next) => {
  console.log('Router-level middleware triggered');
  next();
});

router.get('/home', (req, res) => {
  res.send('Welcome to Home Page');
});

const app = express();
app.use('/api', router);

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

---

3. Built-in Middleware

Express provides several built-in middleware functions for common tasks like serving static files and parsing incoming requests.

- **Examples**:
  - `express.json()` : Parses incoming JSON data.
  - `express.urlencoded()` : Parses URL-encoded data.
  - `express.static()` : Serves static files like images, CSS, and JavaScript.
- Example:

```
const express = require('express');
const app = express();

// Built-in middleware to parse JSON and serve static files
app.use(express.json());
app.use(express.static('public'));

app.post('/data', (req, res) => {
  res.send(`Received data: ${JSON.stringify(req.body)}`);
});

app.listen(3000, () => {
```

```
  console.log('Server running on port 3000');
});
```

---

4. Third-Party Middleware

Third-party middleware can be installed via `npm` to extend Express.js functionality. Examples include `morgan` for logging, `cors` for handling Cross-Origin Resource Sharing, and `body-parser` for parsing bodies.

- **Example** using `morgan` for logging requests:

```
const express = require('express');
const morgan = require('morgan');

const app = express();

// Third-party middleware
app.use(morgan('combined')); // Logs HTTP requests

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

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

---

5. Error-Handling Middleware

Error-handling middleware is used to catch and process errors during the request-response cycle. It takes **four arguments**: `(err, req, res, next)`.

- **Example**:

```
const express = require('express');
const app = express();

// Route handler
app.get('/', (req, res) => {
```

```
  throw new Error('Something went wrong!');
});

// Error-handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Internal Server Error');
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

Summary Table: Types of Middleware

Middleware Type	Description
Application-Level	Defined at the application level using `app.use()` or `app.METHOD()`.
Router-Level	Scoped to a specific router instance using `express.Router()`.
Built-in Middleware	Middleware provided by Express (e.g., `express.json()`, `express.static()`).
Third-Party Middleware	Middleware installed via npm (e.g., `morgan`, `cors`, `body-parser`).
Error-Handling Middleware	Catches and processes errors; defined with four arguments `(err, req, res, next)`.

Significance of Middleware

Middleware enhances the functionality of an Express application by:

- Managing tasks like **logging**, **authentication**, **data parsing**, and **error handling**.
- Decoupling the **application logic** into smaller, manageable pieces.
- Allowing for code reusability and modularity.

14

4.d. Write a program to upload a file to Node.js server using express file upload library.

```
const express = require("express");
const fileupload = require("express-fileupload");
const fs=require("fs");//file system

var app=express();


app.use(fileupload())

app.get('/',(req,res)=>
{
        res.sendFile(__dirname+'/index.html')
})

app.post('/',(req,res)=>
{
        if(req.files)
                console.log(req.files);

//select file and display
        var file=req.files.file;
        var filename=file.name;
        console.log(filename);

//select and move(upload)
        file.mv('./files/'+ filename,function(err)
        {
                if(err)
                {
                        res.send(err);
                }else
                {
                        res.send("File " + file.name + "uploaded ");
                }
        })
})

app.listen(3000,function(){
```

```
        console.log("Server up at 3000")
});
```

15

1.a. What is a protocol? Differentiate between HTTP and HTTPs

1.a. What is a Protocol?

A **protocol** is a set of rules or standards that govern how data is transmitted and received over a network. It ensures proper communication between devices or systems by defining formats, rules, and procedures.

Difference Between HTTP and HTTPS

Aspect	HTTP (HyperText Transfer Protocol)	HTTPS (HyperText Transfer Protocol Secure)
Definition	A protocol for transferring data over the web.	A secure version of HTTP that encrypts data.
Security	Data is **not encrypted** (plain text).	Data is **encrypted** using SSL/TLS protocols.
Port Number	Uses **Port 80** by default.	Uses **Port 443** by default.
Data Safety	Vulnerable to attacks like sniffing or tampering.	Protects data from eavesdropping or tampering.
Encryption	No encryption mechanism.	Encrypts data with SSL/TLS encryption.
Speed	Faster, as no encryption overhead.	Slightly slower due to encryption process.
Usage	Suitable for non-sensitive data transfer.	Preferred for secure transactions (e.g., banking, login pages).
URL Prefix	Starts with `http://`.	Starts with `https://`.
Certificate	Does not require an SSL certificate.	Requires an SSL/TLS certificate.

Summary:

- **HTTP**: Standard for communication without security.
- **HTTPS**: Secure version of HTTP, encrypting data for safety. It is essential for handling sensitive information.

16

1.b. Explain the usage of checkbox and radio button in forms with example.

1.b. Usage of Checkbox and Radio Button in Forms

Checkboxes and radio buttons are used in HTML forms to allow users to select options. However, their behavior is different:

1. **Checkbox**: Allows **multiple selections**.
2. **Radio Button**: Allows **only one selection** within a group.

---

1. Checkbox

- **Usage**: Used when a user can select multiple options from a set of choices.
- **Behavior**: Each checkbox acts independently; users can tick or untick multiple options.

Example of Checkbox:

```
<h3>Select Your Hobbies</h3>
<form>
  <input type="checkbox" id="reading" name="hobby" value="Reading">
  <label for="reading">Reading</label><br>

  <input type="checkbox" id="traveling" name="hobby" value="Traveling">
  <label for="traveling">Traveling</label><br>

  <input type="checkbox" id="music" name="hobby" value="Music">
  <label for="music">Music</label><br>
```

```
  <input type="submit" value="Submit">
</form>
```

Explanation:

- `type="checkbox"` defines checkboxes.
- Multiple checkboxes can be selected simultaneously.
- The `value` attribute stores the selected option for submission.

---

2. Radio Button

- **Usage**: Used when the user must select **only one option** from a set of mutually exclusive choices.
- **Behavior**: Selecting one option automatically deselects the others within the same group (same `name` attribute).

Example of Radio Button:

```
<h3>Select Your Gender</h3>
<form>
  <input type="radio" id="male" name="gender" value="Male">
  <label for="male">Male</label><br>

  <input type="radio" id="female" name="gender" value="Female">
  <label for="female">Female</label><br>

  <input type="radio" id="other" name="gender" value="Other">
  <label for="other">Other</label><br>

  <input type="submit" value="Submit">
</form>
```

Explanation:

- `type="radio"` defines radio buttons.
- All radio buttons in the same group share the **same** `name` (e.g., `name="gender"`).
- Only one radio button in the group can be selected at a time.

Key Differences:

Aspect	Checkbox	Radio Button
Selection	Allows multiple options to be chosen.	Allows only one option to be chosen.
Grouping	Independent checkboxes.	Grouped by the same `name` attribute.
Example Use	Select hobbies, preferences, etc.	Select gender, payment method, etc.

Combined Example:

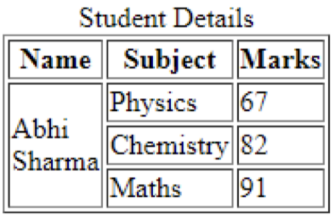
```
<h3>Survey Form</h3>
<form>
  <!-- Checkboxes -->
  <h4>Select Languages You Know:</h4>
  <input type="checkbox" id="english" name="language" value="English">
  <label for="english">English</label><br>

  <input type="checkbox" id="spanish" name="language" value="Spanish">
  <label for="spanish">Spanish</label><br>

  <!-- Radio Buttons -->
  <h4>Select Your Age Group:</h4>
  <input type="radio" id="below18" name="age" value="Below 18">
  <label for="below18">Below 18</label><br>

  <input type="radio" id="above18" name="age" value="18 and Above">
  <label for="above18">18 and Above</label><br>

  <input type="submit" value="Submit">
</form>
```

This form uses **checkboxes** to select multiple languages and **radio buttons** to select only one age group.

17

1.c. Create a table as shown in figure below with borders and caption as “Student Details”

(6.0 Marks)



Student Details

Name	Subject	Marks
Abhi Sharma	Physics	67
	Chemistry	82
	Maths	91

To create a table like the one shown in the image with borders and a caption "Student Details," you can use HTML. Below is the detailed code:

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Student Details</title>
    <style>
        table {
            border-collapse: collapse;
            width: 50%;
            text-align: center;
        }
        th, td {
            border: 1px solid black;
            padding: 8px;
        }
        caption {
            font-weight: bold;
            margin: 10px;
        }
    </style>
</head>
<body>
    <table>
        <caption>Student Details</caption>
        <tr>
            <th>Name</th>
            <th>Subject</th>
            <th>Marks</th>
        </tr>
        <tr>
            <td rowspan="3">Abhi Sharma</td>
```

```
            <td>Physics</td>
            <td>67</td>
        </tr>
        <tr>
            <td>Chemistry</td>
            <td>82</td>
        </tr>
        <tr>
            <td>Maths</td>
            <td>91</td>
        </tr>
    </table>
</body>
</html>
```

Key Points in the Code:

1. **Bordered Table**: The `border-collapse: collapse;` ensures that the table has clean borders.
2. **Caption**: The `<caption>` tag is used to add the title "Student Details."
3. **Rowspan**: The `rowspan="3"` merges the "Name" column for all subjects (Physics, Chemistry, and Maths).
4. **Styling**: Inline CSS styles for borders, padding, and alignment make the table visually neat.

18

1.d.
Mention the steps required to add a hyperlink to an image

To make an image clickable as a hyperlink in HTML, you need to wrap the `<img>` tag with an `<a>` (anchor) tag.

Steps:

1. **Use the `<a>` tag** to define the hyperlink.
    - The `href` attribute specifies the URL or link destination.
2. **Insert the `<img>` tag** inside the `<a>` tag.

- The `<img>` tag is used to display the image.
- The `src` attribute specifies the image source (URL or file path).

3. **Optional attributes** can be added to the `<img>` tag:
   - `alt`: Specifies alternate text for the image.
   - `width` and `height`: Set the size of the image.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Image Hyperlink Example</title>
</head>
<body>
    <h3>Click on the Image to Visit Google</h3>

    <!-- Anchor tag wrapping the image -->
    <a href="https://www.google.com" target="_blank">
        <img src="google-logo.png" alt="Google Logo" width="200"
height="100">
    </a>
</body>
</html>
```

Explanation:

1. `<a href="https://www.google.com">`:
   - Wraps the image and specifies the hyperlink to `https://www.google.com`.
   - `target="_blank"` ensures the link opens in a new tab.
2. `<img src="google-logo.png" alt="Google Logo">`:
   - Displays the image from the file `google-logo.png`.
   - `alt` provides alternate text if the image fails to load.
3. **Output**:
   - The image appears on the webpage.
   - When clicked, it redirects to the specified URL.

Summary:

To add a hyperlink to an image:

- Use `<a href="URL">` to define the link.
- Place the `<img>` tag inside the `<a>` tag.
- Customize the image with `src`, `alt`, `width`, and `height` attributes.

19

2.a. What is Event bubbling in DOM ?

Event Bubbling in DOM

Event bubbling is a type of event propagation in the **Document Object Model (DOM)** where an event triggered on a child element **bubbles up** (or propagates) to its parent elements in the hierarchy, all the way up to the root element.

In simpler terms:
When an event occurs on a **child element**, it **first triggers** on the child and then bubbles up to its **ancestor elements** (like parent, grandparent, etc.).

How Event Bubbling Works

1. The event starts from the **target element** (the element where the event occurred).
2. It moves upward through the **DOM tree** to the root `<html>` element.
3. Any event listeners attached to the ancestor elements are also triggered if the event propagates.

Example of Event Bubbling

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Event Bubbling Example</title>
    <script>
```

```
        // Parent Event Listener
        document.getElementById("parent").addEventListener("click",
function() {
            alert("Parent element clicked!");
        });

        // Child Event Listener
        document.getElementById("child").addEventListener("click",
function() {
            alert("Child element clicked!");
        });
    </script>
</head>
<body>
    <!-- Parent Element -->
    <div id="parent" style="padding: 30px; background-color: lightblue;">
        Parent Element
        <!-- Child Element -->
        <button id="child">Click Me</button>
    </div>
</body>
</html>
```

Explanation:

1. **Child Button Clicked**:
   When you click the `<button>` (child element):
   - The child's event listener (`Child element clicked!`) executes first.
   - Then, the parent's event listener (`Parent element clicked!`) executes **because of bubbling**.
2. **Event Propagation**:
   The event propagates from the child `<button>` to the parent `<div>`.

Event Flow: Capturing vs. Bubbling

- In **event bubbling** (default behavior), events propagate **upwards** from child to parent.
- In **event capturing**, events propagate **downwards** from parent to child.
  - Capturing can be enabled by passing `true` as the third parameter in `addEventListener`.

```
element.addEventListener("click", handler, true); // Capturing phase
```

Key Points:

- Event bubbling is the default propagation behavior in the DOM.
- Events bubble from the **child** element to its **parent** elements.
- You can **stop bubbling** using `event.stopPropagation()`.

```
event.stopPropagation(); // Prevents the event from bubbling up
```

Significance of Event Bubbling

1. Allows event delegation: You can attach a single event listener to a parent element and handle events for its child elements.
2. Efficient handling of dynamically added elements.

20

Give an example for built in object in JavaScript.

Built-in Object in JavaScript

JavaScript provides several **built-in objects** to simplify programming. These objects offer useful functionalities and properties for common tasks like working with strings, numbers, dates, arrays, and more.

Example: `Math` Object

The `Math` object in JavaScript is a built-in object that provides mathematical constants and functions. You do not need to create it, as it is automatically available.

Example Code:

```
// Using Math built-in object
let number = -8.5;

console.log("Absolute Value:", Math.abs(number));       // Returns 8.5
console.log("Rounded Value:", Math.round(number));      // Returns -9
console.log("Square Root of 16:", Math.sqrt(16));       // Returns 4
console.log("Power (2^3):", Math.pow(2, 3));            // Returns 8
console.log("Random Number (0 to 1):", Math.random());  // Returns a random
number
```

Output (Example):

```
Absolute Value: 8.5
Rounded Value: -9
Square Root of 16: 4
Power (2^3): 8
Random Number (0 to 1): 0.5634
```

Key Points about Math Object:

1. **Static**: You don't create an instance of `Math` (e.g., `new Math()`).
2. **Functions and Constants**: It includes helpful functions like `abs`, `round`, `sqrt`, `pow`, and constants like `Math.PI`.

Other **built-in objects** include:

- `Date` (e.g., for handling dates and times)
- `String` (e.g., for working with text)
- `Array` (e.g., for handling lists of values)
- `Number` (e.g., for numerical operations)

Each built-in object comes with its own properties and methods that simplify common tasks.

21

Explain JQuery selectors.

jQuery Selectors

jQuery selectors are used to select and manipulate HTML elements. These selectors allow developers to target elements using **CSS-like syntax** and perform actions on them.

Types of jQuery Selectors

Here are the most commonly used types of selectors in jQuery:

1. **Element Selector (`$("tag")`)**
   Selects all HTML elements of a specific type (tag).
   Example:

   ```
   $("p").css("color", "blue"); // Selects all <p> tags and changes text color to blue.
   ```

2. **ID Selector (`$("#id")`)**
   Selects a specific element by its `id`.
   Example:

   ```
   $("#header").hide(); // Hides the element with id "header".
   ```

3. **Class Selector (`$(".class")`)**
   Selects all elements with a specific `class`.
   Example:

   ```
   $(".highlight").css("background-color", "yellow"); // Highlights all elements with "highlight" class.
   ```

4. **Universal Selector (`$("*")`)**
   Selects **all elements** in the document.

Example:

```
$("*").css("margin", "0"); // Removes margin from all elements.
```

5. **Attribute Selector (`$("[attribute]")`)**
   Selects elements with a specific attribute.
   Example:

```
$("[type='text']").val("Hello!"); // Selects input fields of type "text"
and sets value.
```

6. **Pseudo-Selectors**
   - `:first` and `:last`: Select the first or last element.
     Example:

```
$("li:first").css("color", "red"); // Selects the first <li> and
turns text red.
```

   - `:even` and `:odd`: Select even or odd indexed elements.
     Example:

```
$("tr:odd").css("background-color", "#f2f2f2"); // Zebra stripes
table rows.
```

   - `:input`: Selects all input elements (`<input>`, `<textarea>`, etc.).
     Example:

```
$(":input").css("border", "1px solid red");
```

7. **Child Selector**
   Targets children of an element.
   - **Direct Child Selector (`$("parent > child")`)**
     Example:

```
$("div > p").css("font-weight", "bold"); // Targets direct <p>
children of <div>.
```

   - **Descendant Selector (`$("ancestor descendant")`)**
     Example:

```
$("ul li").css("color", "green"); // Selects all <li> inside <ul>.
```

Example Program:

```
<!DOCTYPE html>
<html>
<head>
  <title>jQuery Selectors Example</title>
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script>
    $(document).ready(function () {
      $("p").css("color", "blue"); // Element selector
      $("#unique").css("font-size", "20px"); // ID selector
      $(".highlight").css("background-color", "yellow"); // Class selector
      $("[type='text']").val("Hello jQuery!"); // Attribute selector
      $("li:first").css("color", "red"); // Pseudo-selector
    });
  </script>
</head>
<body>
  <h1>jQuery Selectors</h1>
  <p>This is a paragraph.</p>
  <p id="unique">This paragraph has an ID.</p>
  <p class="highlight">This paragraph has a class.</p>
  <input type="text" placeholder="Type here">
  <ul>
    <li>First item</li>
    <li>Second item</li>
  </ul>
</body>
</html>
```

Output Explanation

1. All `<p>` tags are turned blue.
2. The `<p>` with ID `unique` has a larger font size.
3. The `<p>` with class `highlight` has a yellow background.

4. The input field gets pre-filled with "Hello jQuery!".
5. The first list item (`<li>`) is styled red.

Significance of jQuery Selectors

- Simplifies DOM manipulation using familiar CSS syntax.
- Provides efficient targeting of elements.
- Enables dynamic styling and behavior of elements based on conditions or events.

22

d. Explain following HTML New tags with an example: 1. Audio and Video tag
2. Progress Bar

1. Audio and Video Tags

HTML5 introduced the `<audio>` and `<video>` tags to embed multimedia content like audio and video files directly into web pages without requiring external plugins like Flash.

---

a. `<audio>` Tag

- Used to embed audio files such as `.mp3`, `.ogg`, or `.wav`.
- Attributes:
  - `controls`: Adds playback controls (play, pause, volume).
  - `autoplay`: Automatically starts playing the audio.
  - `loop`: Repeats the audio.
  - `src`: Specifies the audio file URL (or can use `<source>` tag).

Example of `<audio>` Tag:

```
<!DOCTYPE html>
<html>
<head>
  <title>Audio Example</title>
</head>
<body>
```

```
  <h2>HTML5 Audio Example</h2>
  <audio controls>
    <source src="audiofile.mp3" type="audio/mpeg">
    Your browser does not support the audio tag.
  </audio>
</body>
</html>
```

- **Output:** An audio player with play/pause, volume, and seek controls.
- If the browser doesn't support the `<audio>` tag, the fallback message ("Your browser does not support the audio tag.") will be displayed.

---

b. `<video>` Tag

- Used to embed video files such as `.mp4`, `.ogg`, or `.webm`.
- Attributes:
  - `controls`: Adds video controls (play, pause, volume, fullscreen, etc.).
  - `autoplay`: Starts playing the video automatically.
  - `loop`: Repeats the video.
  - `poster`: Specifies an image to show before the video starts.
  - `muted`: Starts the video with muted audio.

Example of `<video>` Tag:

```
<!DOCTYPE html>
<html>
<head>
  <title>Video Example</title>
</head>
<body>
  <h2>HTML5 Video Example</h2>
  <video width="320" height="240" controls poster="thumbnail.jpg">
    <source src="video.mp4" type="video/mp4">
    Your browser does not support the video tag.
  </video>
</body>
</html>
```

- **Output:** A video player with playback controls. A poster image will be shown before the video starts.

- Fallback message will display if the browser doesn't support `<video>`.

---

2. Progress Bar

The **`<progress>` tag** is used to represent the completion progress of a task (e.g., file download, loading progress, etc.).

Attributes of `<progress>`:

- `value`: The current progress value (a number).
- `max`: The maximum value of the progress bar (default is `1`).

Example of `<progress>` Tag:

```
<!DOCTYPE html>
<html>
<head>
  <title>Progress Bar Example</title>
</head>
<body>
  <h2>HTML5 Progress Bar Example</h2>
  <label for="fileUpload">File Upload Progress:</label>
  <progress id="fileUpload" value="70" max="100"></progress>
  <p>70% completed</p>
</body>
</html>
```

- **Output:** A progress bar that visually represents 70% completion.
- The `value` attribute specifies the progress (current state), and the `max` attribute sets the total.

Significance of These Tags

1. **Audio and Video:**
   - Allows embedding multimedia content without third-party plugins.
   - Provides better performance and cross-browser support.

- Supports modern attributes like autoplay, loop, and controls for a seamless experience.

2. **Progress Bar:**
   - Helps to display the state of progress visually.
   - Useful for loading indicators, file upload progress, or task completion tracking.
   - Simple and easy to implement in forms and dynamic web applications.

23

Give an example for complex components in React.
A **complex React component** involves multiple smaller components working together and handling various logic or state. Here's a brief breakdown of the example **Todo List** app:

Key Components:

1. **TodoItem**: Displays individual todo items, with a checkbox to mark them as completed and a button to delete them.
2. **TodoInput**: A form to add new todos, taking input from the user.
3. **TodoList**: The main component managing the list of todos, including adding, deleting, and toggling completion.

Features:

- **State Management**: Uses React's `useState` to manage the todo list and individual item states.
- **Event Handling**: Handles events like adding, deleting, and toggling todos.
- **Component Composition**: Combines smaller components (`TodoItem` and `TodoInput`) within a larger component (`TodoList`).

This structure allows for a complex user interface where different components manage their parts but work together within the main `TodoList`.

24

Define Refs wrt React with an example.
Ref provides a way to access DOM nodes or React elements created in the render method. It is an attribute which makes it possible to store a reference to particular DOM nodes

or React elements.

According to React.js documentation some of the best cases for using refs are:

➢ managing focus

➢ text selection

➢ media playback

➢ triggering animations

➢ integrating with third-party DOM libraries

Usually props are the way for parent components to interact with their children. However, in some cases you might need to modify a child without re-rendering it with new props. That’s

exactly when refs attribute comes to use.

```
<div id="root"></div>
        <script type = "text/babel">
                class My_component extends React.Component
                {       constructor()
                        {       super(); this.myref = React.createRef()                 }
                        render()
                        {       alert("in render")
                                return (
                                        <input type = "text" ref = {this.myref} />
                                        <button onClick = {this.increment}>+</button>
                                        </div>
                                )       }
                        increment=()=>
                        {       this.myref.current.value++;                     }
                        decrement=()=>
                        {       this.myref.current.value--;                     }
                }
```



```
                ReactDOM.render(<My_component/>, document.getElementById("root"))
        </script>
```

25

Write a note on stateless components

Stateful vs Stateless Components

Stateful Components	Stateless Components
Also known as container or smart components.	Also known as presentational or dumb components.
Have a state	Do not have a state
Can render both props and state	Can render only props
Props and state are rendered like *{this.props.name}* and *{this.state.name}* respectively.	Props are displayed like *{props.name}*
A stateful component is always a *class* component.	A Stateless component can be either a functional or class component.

26

Create a class component which will render a H1 and paragraph element apply different inline style for H1 and P element created using java script objects (key value pair).

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

class StyledComponent extends Component {
  render() {
    // JavaScript objects defining styles
    const h1Style = {
      color: 'blue',
      fontSize: '36px',
      textAlign: 'center',
      fontWeight: 'bold'
    };

    const pStyle = {
      color: 'gray',
      fontSize: '18px',
```

```
      textAlign: 'left',
      margin: '20px'
    };

    return (
      <div>
        <h1 style={h1Style}>This is a Heading</h1>
        <p style={pStyle}>This is a paragraph with different inline styles
applied using JavaScript objects.</p>
      </div>
    );
  }
}

export default StyledComponent;
```

Explanation:

1. **Class Component**: This example uses a React class component that extends `React.Component`.
2. **Inline Styles**: Inline styles are applied through JavaScript objects. Each style is defined as a key-value pair, where the key is the CSS property in camelCase (e.g., `fontSize`, `textAlign`) and the value is the corresponding CSS value (e.g., `'36px'`, `'blue'`).
3. **Applying Styles**: The styles are applied to the `<h1>` and `<p>` elements using the `style` attribute in JSX. The value of `style` is a reference to the JavaScript objects `h1Style` and `pStyle`.

Output:

- The `<h1>` element will appear with blue text, 36px font size, centered, and bold.
- The `<p>` element will appear with gray text, 18px font size, aligned to the left, and a 20px margin.

27

Write a code snippet that reads from a file “test.txt” and writes the first 20 characters to the file “test20.txt” (Hint: You can use string methods). The code must appropriately handle the errors also.

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

// Read the contents of 'test.txt'
fs.readFile('test.txt', 'utf8', (err, data) => {
  if (err) {
    // Handle error if file reading fails
    console.error('Error reading file:', err);
    return;
  }

  // Extract the first 20 characters from the file content
  const first20Chars = data.slice(0, 20);

  // Write the first 20 characters to 'test20.txt'
  fs.writeFile('test20.txt', first20Chars, (err) => {
    if (err) {
      // Handle error if file writing fails
      console.error('Error writing file:', err);
    } else {
      console.log('Successfully written the first 20 characters to
test20.txt');
    }
  });
});
```

Explanation:

1. **Reading the file**: We use `fs.readFile()` to read the contents of `test.txt`. The `'utf8'` encoding ensures that the file content is read as a string.
2. **Error Handling**: If the `readFile` operation encounters an error (e.g., file not found), it will log the error message.
3. **Extracting the First 20 Characters**: We use the `slice(0, 20)` method to extract the first 20 characters from the file content.
4. **Writing to the New File**: We use `fs.writeFile()` to write the extracted characters to `test20.txt`. If the write operation fails, an error message is logged. If it succeeds, a success message is displayed.

Error Handling:

- If `test.txt` does not exist or there are issues reading it, the error will be logged.
- Similarly, any issues during the write operation will also be caught and reported.

Make sure that `test.txt` exists in the same directory as the script, or you can provide the full path to the file.

28

What are the different kinds of events that can be fired by streams?
In Node.js, streams are objects that enable reading or writing data in a continuous manner. Streams emit various events during their lifecycle to handle the data processing and error handling. Here are the common types of events that can be fired by streams:

1. `data`

- This event is emitted when there is data available to be consumed from a stream. It is typically used in readable streams.
- **Example**:

```
const fs = require('fs');
const readableStream = fs.createReadStream('example.txt');

readableStream.on('data', (chunk) => {
  console.log('Received chunk:', chunk);
});
```

2. `end`

- This event is emitted when there is no more data to read from a stream. It signals that the stream has finished sending data.
- **Example**:

```
readableStream.on('end', () => {
  console.log('No more data to read.');
});
```

3. `error`

- This event is emitted when an error occurs in the stream, such as failure in reading or writing data, or in the underlying system.
- **Example**:

```
readableStream.on('error', (err) => {
  console.error('An error occurred:', err);
});
```

4. `close`

- This event is emitted when the stream and the underlying resource (like a file or a network socket) have been closed. It may be triggered after the `end` event if there are no other operations to perform.
- **Example**:

```
readableStream.on('close', () => {
  console.log('Stream closed.');
});
```

5. `finish`

- This event is emitted on writable streams when all the data has been successfully written to the destination (e.g., a file or network).
- **Example**:

```
const writableStream = fs.createWriteStream('output.txt');

writableStream.on('finish', () => {
  console.log('All data written to the file.');
});

writableStream.write('Hello, world!');
writableStream.end();
```

6. `pipe`

- This event is emitted when data is being piped from a readable stream to a writable stream. The `pipe()` method is used to simplify connecting streams, and it triggers the `pipe` event internally.
- **Example**:

```
readableStream.pipe(writableStream);
```

7. `drain`

- This event is emitted on a writable stream when the internal buffer is drained, and it is safe to write more data to the stream.
- It is commonly used in situations where you need to wait until the stream is ready to accept more data (e.g., writing large amounts of data).
- **Example**:

```
writableStream.on('drain', () => {
  console.log('Stream buffer is drained, you can write more data.');
});
```

Summary:

- `data`: Fired when data is available for reading.
- `end`: Fired when the stream has no more data.
- `error`: Fired when an error occurs in the stream.
- `close`: Fired when the stream is closed.
- `finish`: Fired when data has been written to the writable stream.
- `pipe`: Fired when data is piped between streams.
- `drain`: Fired when the writable stream's internal buffer is emptied and ready for more data.

These events allow developers to manage and interact with streams effectively, handling data flow, errors, and stream lifecycle in an asynchronous manner.

29

4.c. Differentiate between update and save query in MongoDB.
In MongoDB, both the `update` and `save` operations are used to modify data, but they behave differently and are used in different scenarios. Here's a detailed comparison between the two:

1. `update()`:

- **Purpose**: The `update()` method is used to modify an existing document or documents in the collection that match the query criteria.
- **Behavior**:
  - By default, it updates only the first document that matches the query.
  - If you want to update multiple documents, you can pass `{ multi: true }` as an option.
  - The update operation only modifies specific fields that are provided in the update object.
  - You can use the `upsert` option to insert a new document if no document matches the query.
- **Syntax**:

```
db.collection.update(query, update, options)
```

  - `query`: The criteria to find the document to update.
  - `update`: The modification to apply to the document.
  - `options`: Optional settings like `upsert` or `multi`.
- **Example**:

```
db.users.update(
  { name: "Alice" }, // Find the document with name "Alice"
  { $set: { age: 30 } }, // Update the age to 30
  { upsert: false } // Do not insert if not found
);
```

2. `save()`:

- **Purpose**: The `save()` method is used to insert a new document or update an existing document. It behaves like an upsert operation, meaning:
  - If the document has an `_id`, it updates the existing document with that `_id`.
  - If the document does not have an `_id`, it inserts a new document.
- **Behavior**:
  - If a document with the same `_id` already exists, it is updated with the provided data.
  - If the document does not have an `_id`, a new document is inserted into the collection.
  - The `save()` method overwrites the entire document, unlike the `update()` method, which can update only specified fields.
- **Syntax**:

```
db.collection.save(document)
```

  - `document`: The document to be inserted or updated.
- **Example**:

```
db.users.save({ _id: 1, name: "Alice", age: 30 }); // This will update
or insert a document with _id: 1
```

Key Differences:

Feature	`update()`	`save()`
Purpose	Modifies existing document(s).	Inserts or updates a document.
Update Fields	Updates specific fields in a document.	Replaces the entire document with the new one.
Upsert Behavior	Can perform upsert with the `upsert` option.	Automatically performs upsert if `_id` is provided.
Multiple Document Update	Can update multiple documents using `{ multi: true }`.	Only updates the document with the same `_id`.
Default Operation	Updates the matching document(s).	Replaces the document if it exists, or inserts a new one.
Flexibility	More control over the update operation (e.g., `$set`, `$inc`).	Less flexible, as it replaces the entire document.

When to Use:

- **Use** `update()` when you need to:
  - Modify specific fields in a document without replacing the entire document.
  - Update multiple documents matching a query.
  - Optionally perform an upsert if no matching document is found.
- **Use** `save()` when:
  - You want to either insert a new document or update an existing one, and you are fine with replacing the entire document.
  - You are dealing with documents that have an `_id` and want to update them based on that identifier.

In general, `update()` provides more control and flexibility, whereas `save()` is simpler and behaves more like an upsert operation.

30

4.d. Create a MongoDB database Company that has a collection Employee (name, ssn, age, salary etc). Write a Server code to Insert the documents and retrieve the same.

```
const http = require('http');
const { MongoClient } = require('mongodb');

// MongoDB URI - Replace with your MongoDB URI
const url = 'mongodb://localhost:27017';
const dbName = 'Company';

const client = new MongoClient(url, { useNewUrlParser: true,
useUnifiedTopology: true });

const server = http.createServer(async (req, res) => {
    // Connect to MongoDB
    await client.connect();
    console.log('Connected to MongoDB');

    const db = client.db(dbName);
    const collection = db.collection('Employee');

    // Handle the POST request to insert data
    if (req.method === 'POST' && req.url === '/insert') {
        let body = '';
        req.on('data', chunk => {
            body += chunk;
        });

        req.on('end', async () => {
            const employee = JSON.parse(body);

            // Insert employee document into the Employee collection
            try {
                const result = await collection.insertOne(employee);
                res.statusCode = 200;
                res.setHeader('Content-Type', 'application/json');
                res.end(JSON.stringify({ message: 'Employee inserted
successfully', result }));
            } catch (error) {
                res.statusCode = 500;
                res.end(JSON.stringify({ message: 'Error inserting
employee', error }));
```

```
            }
        });
    }

    // Handle the GET request to retrieve all employee data
    else if (req.method === 'GET' && req.url === '/employees') {
        try {
            const employees = await collection.find({}).toArray();
            res.statusCode = 200;
            res.setHeader('Content-Type', 'application/json');
            res.end(JSON.stringify(employees));
        } catch (error) {
            res.statusCode = 500;
            res.end(JSON.stringify({ message: 'Error retrieving employees',
error }));
        }
    } else {
        res.statusCode = 404;
        res.end('Not Found');
    }
});

server.listen(3000, () => {
    console.log('Server running at http://localhost:3000/');
});
```

Explanation:

1. **MongoDB Connection**: The code connects to a local MongoDB server running at `mongodb://localhost:27017` and accesses the `Company` database.
2. **POST Request (`/insert`)**:
   - When a POST request is sent to `/insert`, the server reads the data (employee details) from the request body.
   - The employee data is expected to be in JSON format.
   - The data is inserted into the `Employee` collection using `insertOne()`.
3. **GET Request (`/employees`)**:
   - When a GET request is sent to `/employees`, the server fetches all the employee documents from the `Employee` collection using `find({})` and converts them into an array using `toArray()`.
4. **Response**: After each operation (inserting or retrieving data), the server responds with either a success message or an error message.

Example of an Employee Object (JSON format):

To insert data, you would need to send a POST request with the following JSON structure:

```
{
  "name": "John Doe",
  "ssn": "123-45-6789",
  "age": 30,
  "salary": 50000
}
```

Output Example:

After sending the GET request, you should get a response with all employee details stored in the `Employee` collection in the database, such as:

```
[
  {
    "_id": "60d9e0e7e7a1f7281c3a74c4",
    "name": "John Doe",
    "ssn": "123-45-6789",
    "age": 30,
    "salary": 50000
  }
]
```

Notes:

- The server uses `insertOne()` to insert a single employee.
- For multiple insertions, you can use `insertMany()`.
- The MongoDB URI (`mongodb://localhost:27017`) is for a local database. If you are using a cloud service (like MongoDB Atlas), you'll need to replace it with the appropriate connection string.

31

How error handling is achieved in ExpressJS
Error handling in Express.js is achieved through middleware functions that handle both synchronous and asynchronous errors. Express provides a robust mechanism to catch errors and handle them properly. Here's an overview of how error handling works in Express.js:

1. Basic Error Handling in Routes

For normal route handlers, you can handle errors by using `try...catch` blocks or by passing an error to the next middleware.

Example:

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  // Example of normal error handling
  try {
    let result = someFunctionThatMightThrowError();
    res.send(result);
  } catch (error) {
    // Send error response
    res.status(500).send('Internal Server Error');
  }
});
```

2. Using `next()` to Pass Errors

In Express, you can use the `next()` function to pass errors to the next error-handling middleware. If an error is passed to `next()`, Express will skip the remaining middleware and pass the error to the error-handling middleware.

Example:

```
app.get('/example', (req, res, next) => {
  try {
    // Some code that could throw an error
    throw new Error('Something went wrong!');
  } catch (error) {
    next(error);  // Pass the error to the next error-handling middleware
  }
});
```

3. Error Handling Middleware

Express allows you to define custom error-handling middleware that is specifically designed to catch and handle errors. This middleware should have 4 parameters:

- `(err, req, res, next)` – The `err` parameter is automatically populated with the error that was passed to `next()`.

Error-handling middleware functions must be defined **after** all other route and middleware handlers in the Express app.

Example of Error Handling Middleware:

```
// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack); // Log the error stack
  res.status(500).send('Something went wrong!');
});
```

In this example, when an error occurs in any route or middleware, the error is passed to the error-handling middleware, where you can log it and send an appropriate response to the client.

4. Asynchronous Error Handling (with Promises/Async-Await)

When dealing with asynchronous code (such as database queries, file reading, etc.), errors can be caught using `try...catch` blocks or by passing errors to `next()` in case of promises or async functions.

Example using async/await:

```
app.get('/asyncExample', async (req, res, next) => {
  try {
    let result = await someAsyncFunctionThatMightFail();
    res.send(result);
  } catch (error) {
    next(error);  // Pass the error to the error-handling middleware
  }
});
```

Alternatively, for functions that return promises:

```
app.get('/promiseExample', (req, res, next) => {
  somePromiseFunctionThatMightFail()
    .then(result => res.send(result))
    .catch(next); // Pass the error to the error-handling middleware
});
```

5. Handling Specific HTTP Status Codes

For different types of errors, you can customize the status code before sending the response. Express allows you to set the status code before sending the response using `res.status()`.

Example:

```
app.get('/notfound', (req, res, next) => {
  res.status(404).send('Page Not Found');
});

app.get('/unauthorized', (req, res, next) => {
  res.status(401).send('Unauthorized');
});
```

6. Handling 404 Errors (Not Found)

For routes that don't exist, you can add a middleware at the end of your route definitions to handle 404 errors.

Example:

```
app.use((req, res, next) => {
  res.status(404).send('Not Found');
});
```

This middleware will be executed if no other route matches the incoming request, ensuring that a 404 error is returned to the client.

7. Custom Error Classes

You can create custom error classes to handle different types of errors in a more structured way. For example, you can create an `AppError` class that extends the built-in `Error` class to handle specific error types, like validation errors.

Example of Custom Error Class:

```
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
  }
}

// Usage
```

```
app.get('/customError', (req, res, next) => {
  next(new AppError('Invalid input', 400));
});

app.use((err, req, res, next) => {
  if (err instanceof AppError) {
    return res.status(err.statusCode).send(err.message);
  }
  res.status(500).send('Something went wrong!');
});
```

Summary:

- **Error Handling Middleware**: Express allows you to define middleware that handles errors globally using `(err, req, res, next)`.
- **Use of `next()`**: Pass errors to the next middleware using `next(err)` to avoid duplication of error handling.
- **Asynchronous Error Handling**: Use `try...catch` or `catch(next)` in async routes to handle errors.
- **404 and Other Status Codes**: Handle 404 errors and other status codes using dedicated middleware.

This error handling mechanism ensures that your Express application can respond gracefully to unexpected conditions and provide useful feedback to users.

31

Explain how routing can be achieved in ExpressJS with the help of GET method

In Express.js, routing with the `GET` method allows you to handle client requests for data from a server. :

1. **Define Routes**: Use `app.get('/path', callback)` to define a route for a specific path. The callback receives the request (`req`) and response (`res`) objects.
2. **Dynamic Parameters**: You can define dynamic routes using `:param`, allowing you to capture values from the URL (e.g., `/user/:id`).
3. **Query Parameters**: Use `req.query` to access query parameters in the URL (e.g., `/search?term=nodejs`).
4. **Response Types**: You can send responses in various formats using `res.send()`, `res.json()`, or `res.render()`.

5. **Multiple Routes**: You can define multiple `GET` routes for different paths and handle different types of requests.
6. **Error Handling**: Errors can be handled with middleware by passing errors to the next function (`next(error)`).

In summary, `GET` routes are essential for retrieving data and can handle dynamic paths, query parameters, and errors in a streamlined way.

---

year 18

1.a. How to create tables using HTML? Write HTML code to design the table shown below.
(This part depends on the specific table shown in the question; since it is not visible in the uploaded text, I will provide a detailed example based on common requirements.)

Here's a detailed example of creating a table with rows, headers, and data:

```
<!DOCTYPE html>
<html>
<head>
    <title>Table Example</title>
</head>
<body>
    <table border="1" cellpadding="5" cellspacing="0">
        <caption>Example Table</caption>
        <tr>
            <th>Column 1</th>
            <th>Column 2</th>
            <th>Column 3</th>
        </tr>
        <tr>
            <td>Row 1, Col 1</td>
            <td>Row 1, Col 2</td>
            <td>Row 1, Col 3</td>
        </tr>
        <tr>
            <td>Row 2, Col 1</td>
            <td>Row 2, Col 2</td>
```

```
            <td>Row 2, Col 3</td>
        </tr>
    </table>
</body>
</html>
```

1.b. What are HTTP requests and HTTP responses? What are the 4 types of HTTP request methods?

1. **HTTP Request**: A message sent by the client to request resources or perform actions on a server. Key components include:
   - Request line: Defines method, resource URL, and protocol version.
   - Headers: Provide additional information (e.g., `Host`, `Content-Type`).
   - Body: Contains data for methods like `POST`.
2. **HTTP Response**: A message sent by the server as a reply to the client's request. Key components include:
   - Status line: Includes the protocol, status code, and status message (e.g., `200 OK`).
   - Headers: Provide metadata about the response.
   - Body: Contains the requested resource or an error message.

4 Types of HTTP Request Methods:

- `GET`: Retrieve information from the server.
- `POST`: Submit data to the server.
- `PUT`: Update or create resources.
- `DELETE`: Remove resources.

1.c. Using a neat diagram, explain the CSS box model.

The CSS box model represents the structure of an HTML element. Each element is composed of:

1. **Content**: The area where the text or image appears.
2. **Padding**: Space between the content and the border.
3. **Border**: Surrounds the padding and content.
4. **Margin**: Space between the element and its surroundings.

Here’s a visual representation of the CSS box model:

```
+------------------------------+
|            Margin            |
+------------------------------+
|           Border             |
+------------------------------+
|           Padding            |
+------------------------------+
|          Content             |
+------------------------------+
```

Example CSS Code:

```css
div {
    width: 100px;
    height: 100px;
    padding: 10px;
    border: 5px solid black;
    margin: 15px;
}
```

2.a. Write short notes on:

1. **Offline Browsing**:
   Offline browsing allows users to access web pages without an internet connection by saving a local copy of the webpage using features like *Service Workers*.
2. **Web Workers**:
   JavaScript API enabling background thread execution for tasks like computations, ensuring the main thread remains responsive.
3. **Hoisting**:
   A behavior in JavaScript where variable and function declarations are moved to the top of their scope during execution. For example:

   ```javascript
   console.log(a); // undefined
   var a = 5;
   ```

2.b. Write CSS rules for the following:

1. **Change the text color of a `div` with ID `test` from black to blue when hovered:**

```
#test:hover {
    color: blue;
}
```

2. **Embedded style for a paragraph with class `copper` to have red font color:**

```
<style>
    p.copper {
        color: red;
    }
</style>
```

3. **Inline style for a paragraph element to set font color to blue:**

```
<p style="color: blue;">I like to study</p>
```

2.c. Purpose of the `let` keyword, and differences between `let` and `var`:

- `let` declares variables with block-level scope and prevents redeclaration.
- `var` declares variables with function-level scope and allows redeclaration.

Examples:

```
// let example:
if (true) {
    let x = 10;
    console.log(x); // 10
}
// console.log(x); // Error: x is not defined

// var example:
if (true) {
    var y = 10;
    console.log(y); // 10
```

```
}
console.log(y); // 10
```

3.a. Display a message with hover popup:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hover Example</title>
    <script>
        function showPopup() {
            alert("Welcome to my web page");
        }
    </script>
</head>
<body>
    <p onmouseover="showPopup()">Welcome!!</p>
</body>
</html>
```

3.b. What are events and event handlers? List a few event handlers with examples.

1. **Events**: Actions or occurrences that happen in the browser, such as user interactions (e.g., clicks, key presses, or mouse movements).
2. **Event Handlers**: Functions that handle these events and define how the program should respond.

Examples of Event Handlers:

- `onclick`: Triggers when an element is clicked.
- `onmouseover`: Triggers when the mouse pointer hovers over an element.
- `onkeydown`: Triggers when a key is pressed.

Example Code:

```
<!DOCTYPE html>
<html>
<head>
    <title>Event Handlers</title>
```

```
    <script>
        function sayHello() {
            alert("Hello, World!");
        }
    </script>
</head>
<body>
    <button onclick="sayHello()">Click Me</button>
</body>
</html>
```

3.c. What is the use of `preventDefault()` and `stopPropagation()`? Explain with code.

1. **`preventDefault()`**: Prevents the default behavior of an event.
   Example: Stopping a form from submitting.

```
<!DOCTYPE html>
<html>
<head>
    <title>preventDefault Example</title>
    <script>
        function stopForm(event) {
            event.preventDefault();
            alert("Form submission prevented!");
        }
    </script>
</head>
<body>
    <form onsubmit="stopForm(event)">
        <input type="text" placeholder="Enter something">
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

2. **`stopPropagation()`**: Stops the event from bubbling up to parent elements.
   Example: Preventing a `div`'s click handler from triggering when a button inside it is clicked.

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>stopPropagation Example</title>
    <script>
        function parentClick() {
            alert("Parent Clicked!");
        }
        function childClick(event) {
            event.stopPropagation();
            alert("Child Clicked!");
        }
    </script>
</head>
<body>
    <div onclick="parentClick()" style="padding: 50px; background-color:
lightblue;">
        Parent Div
        <button onclick="childClick(event)">Child Button</button>
    </div>
</body>
</html>
```

4.a. What is a web server? How does it work?

1. **Web Server**: A software or hardware that serves web content (like HTML pages) to clients (browsers) over the internet using HTTP/HTTPS protocols.
2. **How It Works**:
   - The client sends an HTTP request to the web server.
   - The server processes the request and retrieves the required resources.
   - It sends back an HTTP response with the requested data (HTML, CSS, JS, etc.) or error messages.

4.b. With respect to Apache configuration, explain the following:

1. **Listen**: Specifies the IP address and port the server listens to for incoming connections. Example: `Listen 80`.
2. **Document Root**: The directory where the server looks for files to serve. Example: `/var/www/html`.
3. **HTTP Authentication**: Provides access control by requiring a username and password for protected areas of the server. Configured in `.htaccess` or `httpd.conf`.

4.c. How can we achieve basic file-based authentication in Apache?

1. Create a `.htpasswd` file:

```
htpasswd -c /path/to/.htpasswd username
```

2. Add configuration to `.htaccess`:

```
AuthType Basic
AuthName "Restricted Area"
AuthUserFile /path/to/.htpasswd
Require valid-user
```

5.a. Difference between `echo` and `print`. Which function to use in PHP for file operations?

1. **Difference between `echo` and `print`:**
   - `echo` can output multiple strings and does not return a value.
   - `print` can only output one string and returns `1`.
2. **File Operations in PHP**: Use the `fopen()` function to open a file. Example:

```php
$file = fopen("example.txt", "w");
fwrite($file, "Hello, File!");
fclose($file);
```

5.b. Modes of opening a file using `fopen`:

- `'r'`: Open for reading.
- `'w'`: Open for writing (overwrites file).
- `'a'`: Open for appending.
- `'r+'`: Open for reading and writing.
- `'w+'`: Open for reading and writing (overwrites file).
- `'a+'`: Open for reading and writing (appends data).

5.c. What is AJAX? What are different ways to deal with asynchronous code?

1. **AJAX (Asynchronous JavaScript and XML):** A technique for creating dynamic and asynchronous web applications by exchanging data with a server without reloading the page.
2. **Dealing with Asynchronous Code:**
   - **Callbacks**: Functions called after an asynchronous operation.
   - **Promises**: Handle asynchronous operations with `.then()` and `.catch()`.
   - **Async/Await**: Simplifies working with Promises. Example:

```
async function fetchData() {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);
}
```

---

year 2021

1.a. What are the 3 ways to include CSS styling in CSS? Explain with examples.

1. **Inline CSS**: Directly applied to an HTML element using the `style` attribute.

```
<p style="color:blue; font-size:14px;">This is a blue text.</p>
```

2. **Internal CSS**: Defined within a `<style>` tag in the `<head>` section of the HTML file.

```
<head>
    <style>
        p {
            color: green;
            font-size: 16px;
        }
```

```
    </style>
</head>
```

3. **External CSS**: Written in a separate `.css` file and linked to the HTML file using the `<link>` tag.

```
<head>
    <link rel="stylesheet" href="styles.css">
</head>
```

`styles.css`:

```
p {
    color: red;
    font-size: 18px;
}
```

1.b. Write the code to get the following output using tables.

Please provide the exact layout of the table, or let me know if you want a general example of creating a structured table with rows and columns.

1.c. Write code to get the following output.

Please specify the desired output for me to write the corresponding code.

Let's address the other questions from the document in detail.

2.a. What are the different ways to get an element from DOM? Explain any 3 with an example.

1. **`getElementById`**: Retrieves an element by its unique `id`.

```
<div id="example">Hello, World!</div>
<script>
```

```
    const element = document.getElementById("example");
    console.log(element.textContent); // Output: Hello, World!
</script>
```

2. `getElementsByClassName` : Retrieves all elements with a specific class name.

```
<div class="example">Item 1</div>
<div class="example">Item 2</div>
<script>
    const elements = document.getElementsByClassName("example");
    console.log(elements[0].textContent); // Output: Item 1
</script>
```

3. `querySelector` : Retrieves the first element that matches a specified CSS selector.

```
<div class="container">
    <p class="example">Hello!</p>
</div>
<script>
    const element = document.querySelector(".container .example");
    console.log(element.textContent); // Output: Hello!
</script>
```

2.b. Write a JavaScript program to calculate multiplication and division of two numbers.

```
function calculateOperations(num1, num2) {
    const multiplication = num1 * num2;
    const division = num1 / num2;
    console.log(`Multiplication: ${multiplication}`);
    console.log(`Division: ${division}`);
}

calculateOperations(10, 2);
```

Output:

```
Multiplication: 20
```

```
Division: 5
```

2.c. Write jQuery code to get the following output.

Requirement: When the user clicks the "Hide" button, text should hide, and when they click "Show," text should display.

```
<div id="text">This is a sample text.</div>
<button id="hide">Hide</button>
<button id="show">Show</button>

<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script>
    $(document).ready(function () {
        $("#hide").click(function () {
            $("#text").hide();
        });
        $("#show").click(function () {
            $("#text").show();
        });
    });
</script>
```

2.d. Write the difference between XML and JSON.

Feature	XML	JSON
Format	Markup language	Data-interchange format
Readability	More verbose and harder to read	Compact and easy to read
Data Types	Only strings	Supports numbers, strings, etc.
Usage	Used for document structure	Used for lightweight data

3.a. Explain the following methods of React Component Lifecycle.

1. `render()` : Renders the JSX to the DOM. Called during the initial mount and whenever the component updates.
2. `componentDidMount()` : Executes after the component is added to the DOM. Ideal for API calls or initializing subscriptions.
3. `componentDidUpdate(prevProps, prevState)` : Executes after a component updates. Useful for handling changes based on state or props.
4. `componentWillUnmount()` : Executes before the component is removed from the DOM. Used to clean up resources like event listeners or timers.

3.b. Write the difference between Stateless and Stateful Components in React.

Aspect	Stateless Components	Stateful Components
Definition	Function components	Class or functional with state
State	No internal state	Manages state
Reusability	Highly reusable	Less reusable due to state

3.c. Write a code to get output in a browser using React map and keys.

```
import React from 'react';

const ItemList = () => {
    const items = ['Apple', 'Banana', 'Cherry'];
    return (
        <ul>
            {items.map((item, index) => (
                <li key={index}>{item}</li>
            ))}
        </ul>
    );
};

export default ItemList;
```

3.d. Write a React class component `Article1` with a state to change color.

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

class Article1 extends Component {
    state = {
        color: 'red',
    };

    changeColor = () => {
        this.setState({ color: 'blue' });
    };

    render() {
        return (
            <div>
                <p style={{ color: this.state.color }}>This is an article.
</p>
                <button onClick={this.changeColor}>Change Color</button>
            </div>
        );
    }
}

export default Article1;
```

4.a. Write in brief the important features of Node.js.

1. **Event-Driven Architecture**: Handles requests asynchronously.
2. **Non-Blocking I/O**: Ensures scalability by not waiting for I/O operations.
3. **Single Programming Language**: JavaScript for both server and client.

4.b. Buffers in Node.js with code.

Buffers are used to handle binary data.

```
const buffer = Buffer.alloc(10); // Create a buffer of size 10
buffer.write("Hello"); // Write data
console.log(buffer.toString()); // Read data
```

4.c. Write a `calc.js` module and include it in Node.js.

calc.js:

```
exports.add = (a, b) => a + b;
exports.subtract = (a, b) => a - b;
exports.multiply = (a, b) => a * b;
```

main.js:

```
const calc = require('./calc.js');

console.log('Addition:', calc.add(10, 5));
console.log('Subtraction:', calc.subtract(10, 5));
console.log('Multiplication:', calc.multiply(10, 5));
```

4.d. Code to connect to MongoDB and insert documents.

```
const { MongoClient } = require('mongodb');

const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri);

async function run() {
    try {
        const database = client.db("Groceries");
        const collection = database.collection("Fruits");

        const fruits = [
            { name: "Apple", price: 100 },
            { name: "Banana", price: 50 },
            { name: "Cherry", price: 150 },
            { name: "Mango", price: 200 },
            { name: "Grapes", price: 120 },
```

```
        ];

        const result = await collection.insertMany(fruits);
        console.log("Documents inserted:", result.insertedCount);
    } catch (err) {
        console.error("Error:", err);
    } finally {
        await client.close();
    }
}

run();
```

1.a. What is HTTP? Explain the structure of an HTTP request message.

HTTP (Hypertext Transfer Protocol) is a protocol used to communicate between a client and a server. It operates in a request-response model where the client sends a request, and the server returns a response.

Structure of an HTTP Request Message:

1. **Request Line**:
   - Specifies the HTTP method (e.g., GET, POST), the URL, and the HTTP version.
   - Example: `GET /index.html HTTP/1.1`
2. **Headers**:
   - Provide additional information about the request (e.g., content type, user agent).
   - Example:

     ```
     Host: www.example.com
     User-Agent: Mozilla/5.0
     ```

3. **Blank Line**:
   - Separates headers from the body.
4. **Body** (Optional):
   - Contains data sent by the client (e.g., form data).

1.b. With a neat diagram, explain the CSS Box Model and its significance.

The CSS **Box Model** is a structure describing how elements are displayed in a web page. It includes:

1. **Content**: The area where the text or other content resides.
2. **Padding**: Space between the content and the border.
3. **Border**: The edge of the element.
4. **Margin**: Space between the element's border and neighboring elements.

Significance:

- Helps understand how elements interact in terms of spacing, alignment, and layout.

Diagram:

```
[ Margin ]
[  Border  ]
[ Padding ]
[ Content ]
```

1.c. Write code to get the following output using lists and table.

Please share the expected output image or describe it so that I can generate the corresponding code.

1.d. Write code to get the following output.

Again, provide the exact output requirements for a tailored solution.

2.a. Write briefly about the two methods of the geolocation object.

1. `getCurrentPosition(successCallback, errorCallback, options)`:
   - Fetches the current geographical position of the user.
   - Example:

```
navigator.geolocation.getCurrentPosition(
    (position) => console.log(position),
    (error) => console.error(error)
);
```

2. `watchPosition(successCallback, errorCallback, options)`:
   - Continuously monitors position changes.
   - Example:

```
const watchId = navigator.geolocation.watchPosition(
    (position) => console.log(position),
    (error) => console.error(error)
);
```

2.b. What is a web worker? Methods/events supported by web workers.

Web Worker:

- A JavaScript feature enabling background thread execution, preventing the main thread from blocking.

Methods/Events:

1. **Worker Methods**:
   - `postMessage(data)`: Sends data to the worker.
   - `terminate()`: Stops the worker.
2. **Worker Events**:
   - `onmessage`: Triggered when the worker receives a message.
   - `onerror`: Handles errors in the worker.

Example:
Main.js:

```
const worker = new Worker("worker.js");
worker.postMessage("Start working!");
worker.onmessage = (e) => console.log(e.data);
```

worker.js:

```
onmessage = (e) => {
    postMessage("Worker says: " + e.data);
};
```

2.c. Write a script to display texts and change their color on click.

```
<h1>Popular Games</h1>
<p id="cod">Call of Duty</p>
<p id="ac">Assassins Creed</p>
<p>PUBG</p>

<script>
    document.getElementById("cod").onclick = function () {
        this.style.color = "red";
    };
    document.getElementById("ac").onclick = function () {
        this.style.color = "red";
    };
</script>
```

2.d. Write code to get the following output using HTML Canvas.

2.d. Write a code to get the following output in browser using html canvas (6.0 Marks)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Canvas Triangle</title>
</head>
<body>
    <!-- Create a canvas element -->
    <canvas id="myCanvas" width="500" height="500"></canvas>

    <script>
        // Get the canvas element
        const canvas = document.getElementById("myCanvas");
        const ctx = canvas.getContext("2d");

        // Set fill color to black
        ctx.fillStyle = "black";

        // Draw a right-angled triangle
        ctx.beginPath();
        ctx.moveTo(50, 50); // Starting point
        ctx.lineTo(50, 200); // Bottom left
        ctx.lineTo(200, 200); // Bottom right
        ctx.closePath(); // Connect back to the start
        ctx.fill(); // Fill the triangle with black color
    </script>
</body>
</html>
```

How it works:

1. **Canvas Initialization**:
   - `<canvas>` creates a drawing area of 500x500 pixels.
   - `getContext("2d")` sets up a 2D drawing environment.
2. **Triangle Drawing**:
   - `moveTo()` sets the starting point (top of the triangle).
   - `lineTo()` draws lines to define the other two points of the triangle.
   - `closePath()` connects the last point to the starting point.
3. **Coloring**:
   - `fillStyle = "black"` sets the fill color.
   - `fill()` fills the triangle.

Output:

- A black right-angled triangle with the hypotenuse along the diagonal, similar to the one in the image.

3.a. React class component to change text color.

```
import React, { Component } from "react";

class Article1 extends Component {
    state = {
        color: "red",
    };

    changeColor = () => {
        this.setState({ color: "blue" });
    };

    render() {
        return (
            <div>
                <p style={{ color: this.state.color }}>This is a red text.
</p>
                <button onClick={this.changeColor}>Change Color</button>
            </div>
```

```
        );
    }
}

export default Article1;
```

3.b. What is the significance of the key property in React?

Key Property:

- Used to uniquely identify elements in a list, helping React efficiently update and re-render only the changed elements.

Example:

```
const items = ["Apple", "Banana", "Cherry"];
const listItems = items.map((item, index) => <li key={index}>{item}</li>);
```

3.c. Class component with different inline styles.

```
import React, { Component } from "react";

class StyledComponent extends Component {
    render() {
        const headerStyle = { color: "blue", fontSize: "24px" };
        const paragraphStyle = { color: "green", fontStyle: "italic" };

        return (
            <div>
                <h1 style={headerStyle}>This is a header</h1>
                <p style={paragraphStyle}>This is a paragraph</p>
            </div>
        );
    }
}

export default StyledComponent;
```

3.d. Explain any 4 components in the React lifecycle.

1. **Mounting**:
   - `componentDidMount` : Executes after component is added to the DOM.
2. **Updating**:
   - `componentDidUpdate` : Runs after a component updates due to props or state changes.
3. **Unmounting**:
   - `componentWillUnmount` : Cleans up resources when the component is removed.
4. **Error Handling**:
   - `componentDidCatch` : Handles errors during rendering.
     Here are detailed answers to the remaining questions from the document:

4.a. What are buffers in Node.js? Write a program to create a buffer, write data to the buffer, and read data from the buffer.

Buffers:

- Buffers are used in Node.js to handle binary data directly. They are particularly useful when working with streams, file I/O, or networking.

Example:

```
// Create a buffer
const buffer = Buffer.alloc(10);

// Write data to the buffer
buffer.write("Hello");

// Read data from the buffer
console.log(buffer.toString()); // Output: Hello
```

4.b. MongoDB Queries for the "course" collection.

1. List all documents:

```
db.course.find();
```

2. List documents with `code = "UE20CS204"`:

```
db.course.find({ code: "UE20CS204" });
```

3. List the first document with `credits = "4"`:

```
db.course.findOne({ credits: "4" });
```

4.c. Node.js server code to write data into `input.txt` and read it asynchronously.

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

// Write data to file
fs.writeFile('input.txt', 'This is a test message.', (err) => {
    if (err) {
        return console.error('Error writing to file:', err);
    }
    console.log('Data written successfully.');

    // Read data from file asynchronously
    fs.readFile('input.txt', 'utf8', (err, data) => {
        if (err) {
            return console.error('Error reading from file:', err);
        }
        console.log('File content:', data);
    });
});
```

4.d. Connect to MongoDB and insert documents into the "Fruits" collection.

```
const { MongoClient } = require('mongodb');

const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri);
```

```
async function run() {
    try {
        const db = client.db("Groceries");
        const collection = db.collection("Fruits");

        // Documents to insert
        const fruits = [
            { name: "Apple", price: 100 },
            { name: "Banana", price: 50 },
            { name: "Cherry", price: 150 },
            { name: "Mango", price: 200 },
            { name: "Grapes", price: 120 },
        ];

        const result = await collection.insertMany(fruits);
        console.log(`${result.insertedCount} documents inserted.`);
    } catch (error) {
        console.error("Error:", error);
    } finally {
        await client.close();
    }
}

run();
```

5.a. What is a RESTful API? Explain any 4 design specifications/constraints of REST.

RESTful API:

- A RESTful API follows REST (Representational State Transfer) principles, allowing stateless communication between client and server.

4 Design Specifications:

1. **Statelessness**:
   - Each request from a client contains all the information needed to process it.
2. **Client-Server Architecture**:
   - Separation of concerns between client (user interface) and server (data storage/processing).
3. **Uniform Interface**:

- Resources are accessed using standard HTTP methods (GET, POST, PUT, DELETE).

4. **Cacheability**:
   - Responses can be cached to improve performance.

5.b. Code to collect form data using a Pug template and retrieve it in Express.

Pug Template (`form.pug`):

```
doctype html
html
  head
    title Library Form
  body
    form(action="/submit" method="POST")
      label(for="name") Name:
      input(type="text" name="name" id="name")
      br
      label(for="sem") Semester:
      input(type="text" name="sem" id="sem")
      br
      label(for="branch") Branch:
      input(type="text" name="branch" id="branch")
      br
      button(type="submit") Submit
```

Express Code:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.set('view engine', 'pug');
app.use(bodyParser.urlencoded({ extended: true }));

app.get('/', (req, res) => {
    res.render('form');
});

app.post('/submit', (req, res) => {
```

```
    const { name, sem, branch } = req.body;
    res.send(`Form submitted! Name: ${name}, Semester: ${sem}, Branch:
${branch}`);
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

5.c. Structuring an Express application with multiple routes.

Approach:

1. **Use Separate Router Files**:
    - Create separate files for `players`, `match`, and `teams` routes.

Example Structure:

```
/routes
    players.js
    match.js
    teams.js
app.js
```

players.js:

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

router.get('/', (req, res) => res.send('All Players'));
router.post('/', (req, res) => res.send('Add a Player'));
router.get('/:id', (req, res) => res.send(`Player ID: ${req.params.id}`));

module.exports = router;
```

app.js:

```
const express = require('express');
const playersRouter = require('./routes/players');
const app = express();

app.use('/players', playersRouter);
```

```
app.listen(3000, () => console.log('Server running on port 3000'));
```

5.d. Express server with `/pes/:branch` and `/pes/:branch/:id` routes.

```
const express = require('express');
const app = express();

app.get('/pes/:branch', (req, res) => {
    res.send(`Branch name is ${req.params.branch}`);
});

app.get('/pes/:branch/:id', (req, res) => {
    const { branch, id } = req.params;
    res.send(`ID: ${id} and name: ${branch}`);
});

app.listen(3000, () => console.log('Server running on port 3000'));
```
