HTML

Hyper Text Markup Language

HTML is the code that is used to structure a web page and its content. The components used to design the structure of websites are called HTML tags.

First HTML File

index.html

It is the default name for a website's homepage.

HTML Tag

A container for some content or other HTML tags.

This is a paragraph

Basic HTML Page

```
<!DOCTYPE html>          Tells browser you are using HTML5
<html>          Root of an HTML document
 <head>          Container for metadata
  <title>My First Page</title>          Page title
 </head>
 <body>          Contains all data rendered by the browser
  <p>Hello world</p>          Paragraph tag
 </body>
</html>
```

Quick Points

- HTML tag is parent of head & body tag.
- Most HTML elements have opening & closing tags with content in between.
- Some tags have no content in between, e.g.,
.
- We can use inspect element/view page source to edit HTML.

Comments in HTML

This is part of the code that should not be parsed.

<!-- This is an HTML Comment -->

HTML is NOT case sensitive

 = <P>

<html> = <HTML>

<head> = <HEAD>

<body> = <BODY>

HTML Attributes

Attributes are used to add more information to the tag.

<html lang="en"> lang - attribute en – value

Heading Tag

Used to display headings in HTML.

<h1> (most important)

<h2>

<h3>

<h4>

<h5>

<h6> (least important)

Paragraph Tag

Used to add paragraphs in HTML.

This is a sample paragraph

Anchor Tag

Used to add links to your page.

Google

Image Tag

Used to add images to your page.

<img src="/image.png" alt="Random Image">

Br Tag

Used to add next line (line breaks) to your page.

Bold, Italic & Underline Tags

Used to highlight text on your page.

Bold **Bold**

Italic *Italic*

<u>Underline</u> <u>Underline</u>

Big & Small Tags

Used to display big & small text on your page.

<big>Big</big>

<small>Small</small>

Hr Tag

Used to display a horizontal ruler, used to separate content.

<hr>

Subscript & Superscript Tag

Used to display subscript and superscript text.

_{subscript}

H₂O

^{superscript}

3² =9

Pre Tag

Used to display text as it is (without ignoring spaces & next line).

<pre>

This

is a sample

text.

</pre>

Page Layout Techniques

Using semantic tags for layout

<header>

<main>

<footer>

Section Tag

For a section on your page.

```
<section class="intro">
  <h2>Introduction</h2>
  <p>This is the introduction section.</p>
</section>
```

Article Tag

For an article on your page.

<article> This is my Article </>

Aside Tag

For content aside from the main content (e.g., ads).

<aside> </aside>

Revisiting Anchor Tag

For opening a link in a new tab:

Google

Clickable picture:

<img src="link">

Revisiting Image Tag

Set height and width:

<img src="link" height="50px">

<img src="link" width="50px">

Div Tag

Div is a container used for other HTML elements. It is a block element (takes full width).

```
<body>
  <div class="header">
    <h1>Header</h1>
  </div>
  <div class="main">
    <h2>Main Content</h2>
    <p>This is the main content area. You can put your main content here.</p>
  </div>
    <p>Footer</p>
  </div>
</body>
```

List of Div Tags

<address>
<article>
<aside>
<blockquote>
<canvas>
<dd>

<dl>
<dt>
<fieldset>
<figcaption>
<figure>
<footer>
<form>
<h1>-<h6>
<header>
<hr>

<main>
<nav>
<noscript>

<p>
<pre>
<section>
<table>
<tfoot>

<video>

Span Tag

Span is also a container used for other HTML elements. It is an inline element (takes width as per size).

List of Span Tags

<a>		<q>
<abbr>	<i>	<samp>
<acronym>	<img>	<script>
<b>	<input>	<select>
<bdo>	<kbd>	<small>
<big>	<label>	
 	<map>	
<button>	<object>	<sub>
<cite>	<tt>	<sup>
<code>	<var>	<textarea>
<dfn>	<output>	<time>

List in HTML

Lists are used to represent real-life list data (unordered and ordered).

Unordered list

```
<ul>
 <li>Apple</li>
 <li>Mango</li>
</ul>
```

Ordered list

```
<ol>
 <li>Apple</li>
 <li>Mango</li>
</ol>
```

Tables in HTML

Tables are used to represent real-life table data.

Table row: <tr></tr>

Table data: <td></td>

Table header: <th></th>

```
<table>
 <caption>Student Data</caption>
  <tr>
   <th>Name</th>
   <th>Roll No</th>
  </tr>
  <tr>
   <td>Gokul</td>
   <td>2403</td>
  </tr>
  <tr>
   <td>Nitesh</td>
   <td>2404</td>
  </tr>
</table>
```

Student Data

Name	Roll No
Gokul	2403
Nitesh	2404

Colspan Attribute

colspan=”n”

Used to create cells which spans over multiple columns.

```
<table>
    <caption>Student Data</caption>
    <thead>
      <tr>
        <th>Name</th>
        <th>Roll No</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>Gokul</td>
        <td>2403</td>
      </tr>
      <tr>
        <td>Nitesh</td>
        <td>2404</td>
      </tr>
      <tr>
        <td colspan="2">Additional Information</td>
      </tr>
    </tbody>
  </table>
```

Student Data

Name	Roll No
Gokul	2403
Nitesh	2404
Additional Information	

Form in HTML

Forms are used to collect data from the user (e.g., sign-up/login/help requests/contact me).

<form> Form content </form>

Action in Form

Action attribute is used to define what action needs to be performed when a form is submitted.

```
<form action="/action.php">
```

Form Element

Input

```
<input type="text" placeholder="Enter Name">
```

Password Input

```
<input type="password" name="password" placeholder="Enter your password">
```

Radio Button

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

Label

The <label> tag is used to define labels for <input> elements.

```
<label for="username">Username:</label>
<input type="text" id="username" name="username">
```

Class & Id

```
<div id="id1" class="group1"></div>
<div id="id2" class="group1"></div>
```

Checkbox

```
<input type="checkbox" value="class X" name="class" id="id1">
<label for="id1"></label>
<input type="checkbox" value="class X" name="class" id="id2">
<label for="id2"></label>
```

Textarea

```
<textarea name="feedback" id="feedback" placeholder="Please add
Feedback"></textarea>
```

Select

```
<select name="city" id="city">
  <option value="Delhi">Delhi</option>
  <option value="Mumbai">Mumbai</option>
  <option value="Bangalore">Bangalore</option>
</select>
```

Iframe Tag

Website inside website:

```
<iframe src="link"></iframe>
```

Video Tag

```
<video src="myVid.mp4" controls height="300" width="400" loop autoplay></video>
```

Audio Tag

```
<audio src="path/to/audiofile.mp3" controls autoplay loop muted preload="auto">
</audio>
```

Submit Button

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

Reset Button

```
<input type="reset" value="Reset">
```

Button

```
<input type="button" value="Click Me">
```

File Input

```
<input type="file" name="fileToUpload">
```

Email Input

```
<input type="email" name="email" placeholder="Enter your email">
```

Number Input

```
<input type="number" name="quantity" min="1" max="5">
```

Date Input

```
<input type="date" name="birthday">
```

Datetime-local Input

```
<label for="meeting">Meeting date and time:</label>
<input type="datetime-local" id="meeting" name="meeting">
```

Textarea Tag

The <textarea> tag is used to create a multi-line text input.

```
<textarea name="message" rows="4" cols="50" placeholder="Enter your message"></textarea>
```

CSS

- Cascading Style Sheet
- makeup
- not a programming language, but a styling language
- But for styling, there should be some content, and that's why we studied HTML before CSS

Basic Syntax

```
h1 {
 color: red;     Property :Value;
}
```

Semicolon shows that one property has ended & it is important to put this even though it won't incur error.

Including Style

Inline

Writing style directly inline on each element

```
<h1 style="color: red"> CDAC Bengaluru </h1>CDAC
```

Internal

Style is added using the <style> element in the same document

```
<style>
h1 {
 color : red;
}
 </style>
```

External Stylesheet

- Writing CSS in a separate document & linking it with HTML file
- An inline style will override external and internal styles

Color Property

- Used to set the color of the foreground

```
color: red;
color: pink;
color: blue;
color: green;
```

Background Color Property

- Used to set the color of the background

```
background-color: red;
background-color: pink;
background-color: blue;
background-color: green;
```

Color Systems

RGB

- We don't have to think of colors on our own, we can just use color picker tools online or search online

```
color: rgb(255, 0, 0);
color: rgb(0, 255, 0);
```

Color Systems

Hex (Hexadecimal)

- Use Google color picker to get this #code

```
color : #ff0000;

color : #00ff00;
```

Selectors

```
* { }      →Universal Selector

h1 { }    →Element Selector

#myId { }        →Id Selector

.myClass { }    →Class Selector
```

Text Properties

Text Alignment

- Text alignment doesn't mean align according to the page; i.e. right does not mean on the page's right side but the parent's right side
- In CSS3, latest CSS -> start and end are introduced for language support like Arabic

```
text-align: left / right / center
```

Text Decoration

- Also add style, wavy, dotted or color like red
- Can also set to none for hyperlinks

```
text-decoration: underline / overline / line-through
```

Font Weight

Font weight is to show how dark or light our text is; it can be names or in terms of numbers

```
font-weight: normal / bold / bolder / lighter
font-weight: 100-900
```

Font Family

We can write multiple families as a fallback mechanism

```
font-family: arial
font-family: arial, roboto          →If 1st not available 2nd will apply
```

Units in CSS

Absolute

- Pixels (px)
- 96px = 1 inch
- cm, mm, inch & others are also there but pixels are the most used

```
font-size: 2px;
```

Line Height

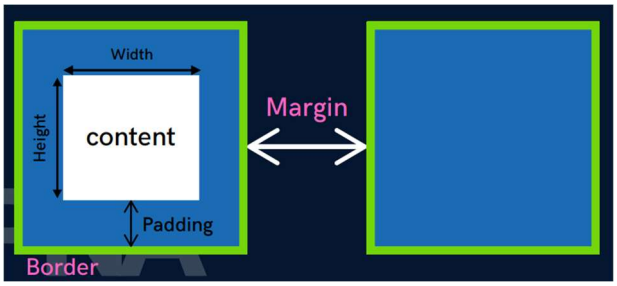
```
line-height: 2px
line-height: 3
line-height: normal
```

Text Transform

```
text-transform: uppercase / lowercase / capitalize / none
```

Box Model in CSS

- Height
- Width
- Border
- Padding
- Margin



Height

By default, it sets the content area height of the element.

```
div {
  height: 50px;
}
```

Width

```
div {
  width: 50px;
}
```

Border

Used to set an element's border

```
border-width: 2px;
border-style: solid / dotted / dashed;
border-color: black;
```

Shorthand

```
border: 2px solid black;
```

Border Radius

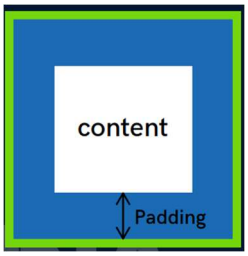
Used to round the corners of an element's outer border edge.

```
border-radius: 10px;

border-radius: 50%;  →circle
```

Padding

- padding-left
- padding-right
- padding-top
- padding-bottom



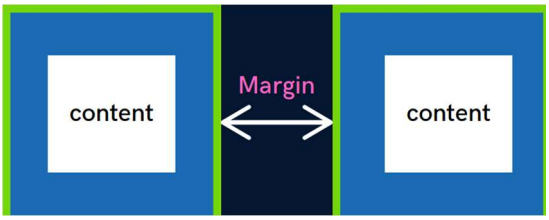
Shorthand

```
padding: 50px;

padding: 1px 2px 3px 4px;     →top | right | bottom | left -> clockwise
```

Margin

- margin-right
- margin-left
- margin-top
- margin-bottom



Shorthand

```
margin: 50px;

margin: 1px 2px 3px 4px;      → top | right | bottom | left -> clockwise
```

Display Property

- inline - Takes only the space required by the element. (no margin/ padding)
- block - Takes full space available in width.
- inline-block - Similar to inline but we can set margin & padding.
- none - To remove element from document flow.

```
display: inline / block / inline-block / none;
```

Visibility

- Note: When visibility is set to none, space for the element is reserved. But for display set to none, no space is reserved or blocked for the element.

```
visibility: hidden;
```

Alpha Channel

RGBA

Opacity (0 to 1)

```
color: rgba(255, 0, 0, 0.5);

color: rgba(255, 0, 0, 1);
```

Units in CSS

Relative

- %
- em→ Relative to the font size of the element
- rem→ Relative to the root element font size
- More like vh, vw etc

Em

- Font size of child will be half of parent for 0.5em.
- For padding & margin it's relative to the same element's font size.

Rem (Root Em)

- Font size of child will be half of parent for 0.5em.
- For padding & margin it's relative to the same element's font size.

Others

- vh: relative to 1% viewport height
- vw: relative to 1% viewport width

Percentage (%)

- It is often used to define a size as relative to an element's parent object.

```
width: 33%;

margin-left: 50%;
```

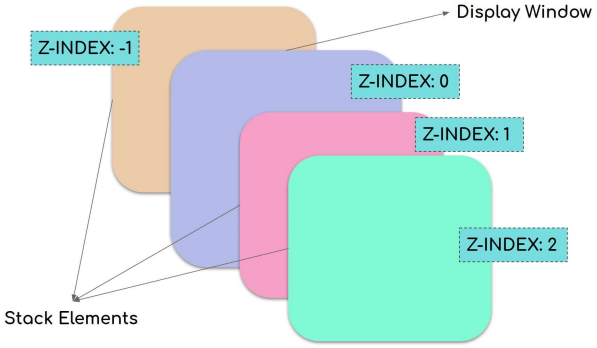
Position

```
position: static / relative / absolute / fixed;
```

- static - default position (The top, right, bottom, left, and z-index properties have no effect)
- relative - element is relative to itself. (The top, right, bottom, left, and z-index will work)
- absolute - positioned relative to its closest positioned ancestor. (removed from the flow)
- fixed - positioned relative to browser. (removed from flow)
- sticky - positioned based on user's scroll position

z-index

- It decides the stack level of elements.
- Overlapping elements with a larger z-index cover those with a smaller one.




```
z-index: auto (0);

z-index: 1 / 2 / ...;

z-index: -1 / -2 / ...;
```

Background Image

- Used to set an image as background

```
background-image: url("image.jpeg");
```

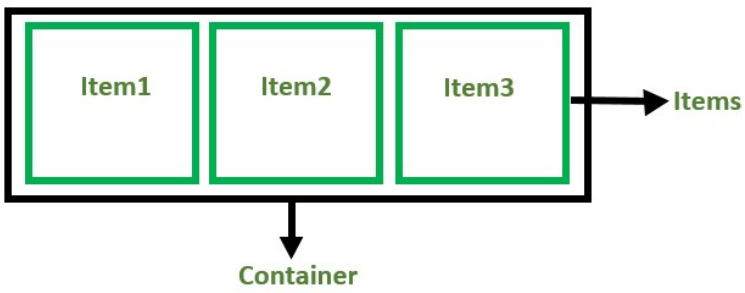
Background Size

- cover = fits with no empty space
- contain - fits with image fully visible
- auto = original size

```
background-size: cover / contain / auto;
```

Flexbox (Flexible Box Layout)

- A one-dimensional layout method for arranging items in rows or columns.



Flexbox Direction

- Flex-direction: row; (default)
- flex-direction: row-reverse;
- flex-direction: column;
- flex-direction: column-reverse;

Flex Properties for Flex Container

- justify-content: Alignment along the main axis.
- Options: flex-start, flex-end, center, space-between, space-around, space-evenly.

- align-items: Alignment along the cross axis.
- Options: flex-start, flex-end, center, baseline, stretch.

- flex-wrap: Controls whether flex items wrap onto multiple lines.
- Options: nowrap, wrap, wrap-reverse.

- align-content: Alignment of space between and around the content along the cross-axis.
- Options: flex-start, flex-end, center, space-between, space-around, stretch

Flex Properties for Flex Item

- align-self: Alignment of individual items along the cross axis.
- Options: auto, flex-start, flex-end, center, baseline, stretch.
- flex-grow: How much a flex item will grow relative to the rest if space is available.
- flex-shrink: How much a flex item will shrink relative to the rest if space is available.

Media Queries

Purpose: Help create responsive websites by applying styles based on the viewport size and orientation.

Example:

```
@media (max-width: 600px) {
  div {
    background-color: red;
  }
}
```

Combine conditions with logical operators:

```
@media (min-width: 200px) and (max-width: 400px) {
   div {
      background-color: green;
   }
}
```

Transitions

- Transitions enable defining the transition between two states of an element.
- Useful for adding animation to elements.

Properties

- transition-property: The property you want to transition (e.g., font-size, width).
- transition-duration: Duration of the transition (e.g., 2s, 4ms).
- transition-timing-function: How the transition should be applied (e.g., ease-in, ease-out, linear, steps).
- transition-delay: Delay before the transition starts (e.g., 2s, 4ms).

Example

```
transition: font-size 2s ease-in-out 0.2s;
```

Css Transform

- Used to apply 2D and 3D transformations to an element.
- Common transformations:
  - transform: rotate(45deg); - Rotates the element by 45 degrees.
  - transform: scale(2); - Scales the element by a factor of 2.
  - transform: translate(20px); - Moves the element 20px to the right.
  - transform: skew(30deg); - Skews the element by 30 degrees.

- Transformations can apply to both x and y axes separately:
  - transform: scaleX(0.5);
  - transform: scaleY(0.5);
  - transform: translateX(20px);
  - transform: translateY(20px);

Animation

- Keyframes allow defining the changes in styles at various points in the animation
- Example:

```
@keyframes myName {
 from { font-size: 20px; }
 to { font-size: 40px; }
}
```

Animation Properties:

- animation-name: The name of the keyframe.
- animation-duration: Duration of the animation.
- animation-timing-function: The timing function of the animation.
- animation-delay: Delay before the animation starts.
- animation-iteration-count: Number of times the animation should repeat.
- animation-direction: Direction of the animation.

Animation Shorthand

- Combines multiple animation properties into a single declaration.
- Example:

```
animation: myName 2s ease-in-out 0.2s infinite alternate;
```

Percentage (%) in Animation

- This allows for more granular control over the animation compared to using only from and to. Here's how you can use percentages in keyframes:

```
@keyframes myName {
    0% { font-size: 20px; }
    50% { font-size: 30px; }
    100% { font-size: 40px; }
}
```

- Using percentages allows for defining intermediate steps in the animation, giving more control over the timing and progression of the animation effects.

JAVASCRIPT

What is JavaScript?

- Use console in any browser or just open one of your projects and use that
- The code is not permanent, we can use the up & down arrow to bring back old code

JavaScript (JS) is a programming language. We use it to give instructions to the computer.

Our 1st JS Code

```
console.log("CDAC Bengaluru");
```

console.log is used to log (print) a message to the console.

Variables in JS

- Variables are containers for data
- Variable names are case sensitive; “a” & “A” are different.
- Only letters, digits, underscore (_) and $ are allowed (not even space).
- Only a letter, underscore (_) or $ should be the 1st character.
- Reserved words cannot be variable names.

let, const & var

- var: Variable can be re-declared & updated. A global scope variable.
- let: Variable cannot be re-declared but can be updated. A block scope variable.
- const: Variable cannot be re-declared or updated. A block scope variable.

Data Types in JS

- Primitive Types: Number, String, Boolean, Undefined, Null, BigInt, Symbol

Comments in JS

- Part of Code which is not executed

```
1  //This is a single line comment
2
3  /* This is a multi-line
4     comment. */
```

Operators in JS

- Arithmetic Operators: Used to perform some operation on data
  - Modulus
  - Exponentiation
  - +, -, *, /
  - Increment
  - Decrement
- Assignment Operators
  - =, +=, -=, *=, %=, **=
- Comparison Operators
  - == Equal to
  - != Not equal to
  - === Equal to & type
  - !== Not equal to & type
    - , >=, <, <=
- Logical Operators
  - && Logical AND
  - || Logical OR
  - ! Logical NOT

Conditional Statements

- if Statement
- if-else Statement
- else-if Statement

Ternary Operators

- condition ? true output : false output

Loops in JS

- Loops are used to execute a piece of code again & again

for Loop

```
for (let i = 1; i <= 5; i++) {
   console.log("CDAC");
}
```

Infinite Loop

- A Loop that never ends

while Loop

```
while (condition) {
   // do some work
}
```

do-while Loop

```
do {
   // do some work
} while (condition);
```

for-of Loop

```
for (let val of strVar) {
   //do some work
}
```

for-in Loop

```
for (let key in objVar) {
   //do some work
}
```

Strings in JS

- String is a sequence of characters used to represent text
  - Create String
  - String Length
  - String Indices

```
let str = “CDAC Bengaluru“;

str.length;

str[0], str[1], str[2];
```

Template Literals in JS

- A way to have embedded expressions in strings

```
`this is a template literal`
```

String Interpolation

- To create strings by doing substitution of placeholders

```
`string text ${expression} string text`
```

String Methods in JS

- These are built-in functions to manipulate a string

```
str.toUpperCase();

str.toLowerCase();

str.trim(); // removes whitespaces

str1.concat(str2); // joins str2 with str1

str.replace(searchVal, newVal);

str.charAt(idx);

str.slice(start, end?); // returns part of string
```

Arrays in JS

- Collections of items
    - Example Arrays:

```
let heroes = ["ironman", "hulk", "thor", "batman"];

let marks = [96, 75, 48, 83, 66];

let info = ["rahul", 86, "Delhi"];
```

Array Indices

- Accessing elements:

```
arr[0], arr[1], arr[2], ...
```

Array Methods

- **Push**: Adds to end.
- **Pop**: Deletes from end & returns the element.
- **toString**: Converts array to string.
- **Concat**: Joins multiple arrays & returns result.
- **Unshift**: Adds to start.
- **Shift**: Deletes from start & returns the element.
- **Slice**: Returns a piece of the array.

```
slice(startIdx, endIdx);
```

- **Splice**: Changes original array (add, remove, replace).

```
splice(startIdx, delCount, newEl1, ...);
```

Functions in JS

- **Definition**: Block of code that performs a specific task, can be invoked whenever needed.
- **Syntax**:

```
function functionName(param1, param2, ...) {
   // do some work
}
functionName(); // Function call
```

Arrow Functions:

Compact way of writing a function

```
const functionName = (param1, param2, ...) => {
   // do some work
}
const sum = (a, b) => {
   return a + b;
}
```

forEach Loop in Arrays

- Syntax:

```
arr.forEach(callbackFunction);
```

Example

```
arr.forEach((val) => {
   console.log(val);
});
```

- **Map**

```
let newArr = arr.map((val) => {
   return val * 2;
});
```

Filter

```
let newArr = arr.filter((val) => {
    return val % 2 === 0;
});
```

Reduce:

```
let result = arr.reduce((acc, val) => {
   return acc + val;
}, 0);
```

The 3 Musketeers of Web Dev

- **HTML**: Structure
- **CSS**: Style
- **JS**: Logic

Starter Code

- **Connecting HTML with JS**: <script> tag
- **Connecting HTML with CSS**: <style> tag

Window Object

- Represents an open window in a browser.
- Global object with many properties & methods.

DOM (Document Object Model)

- Created by the browser when a web page is loaded.
- Allows manipulation of HTML elements.

DOM Manipulation

- **Selecting Elements**:

```
document.getElementById("myId");

document.getElementsByClassName("myClass");

document.getElementsByTagName("p");

document.querySelector("#myId / .myClass / tag"); // returns first element

document.querySelectorAll("#myId / .myClass / tag"); // returns a NodeList
```

 Properties:

- tagName: returns tag for element nodes
- innerText: returns the text content of the element and all its children
- innerHTML: returns the plain text or HTML content in the element
- textContent: return textual content even for hidden elements

 Attributes:

- getAttribute(attr) //to get the attribute value
- setAttribute(attr, value) // to set the attribute value

 Style:

- node.style

 Inserting Elements:

```
node.append(el); // adds at the end of node (inside)

node.prepend(el); // adds at the start of node (inside)

node.before(el); // adds before the node (outside)

node.after(el); // adds after the node (outside)

node.remove(); // removes the node
```

Events in JS

- **The change in the state of an object is known as an Event.**
- Fired to notify code of "interesting changes" that may affect code execution.
- **Types of Events**:
    - Mouse events (click, double click, etc.)
    - Keyboard events (keypress, keyup, keydown)
    - Form events (submit, etc.)
    - Print event & many more

Event Handling in JS

- Syntax

```
node.event = () => {
  // handle here
}
btn.onclick = () => {
  console.log("btn was clicked");
}
```

Event Object

- A special object with details about the event.
- All event handlers have access to the Event Object's properties and methods.
- Example

```
node.event = (e) => {

  // handle here

  console.log(e.target, e.type, e.clientX, e.clientY);

}
```

Event Listeners

- Adding and removing event listeners

```
node.addEventListener(event, callback);

node.removeEventListener(event, callback);
```

Prototypes in JS

- A javascript object is an entity having state and behavior (properties and method).
- JS objects have a special property called prototype.
- We can set prototype using __proto__
- If object & prototype have the same method, the object's method will be used.

Classes in JS

Class is a program-code template for creating objects.

Those objects will have some state (variables) & some behaviour (functions) inside it.

```
class MyClass {
   constructor() { ... }
   myMethod() { ... }
}

let myObj = new MyClass();
```

Constructor method:

- Automatically invoked by new.
- Initializes object.

Inheritance in JS

- Passing down properties & methods from parent to child class.

```
class Parent { ... }
class Child extends Parent { ... }
```

Method Overriding:

- If Child & Parent have the same method, the child's method will be used.

Super Keyword

- Used to call the constructor of the parent class

```
super.parentMethod(args);
super(args); // calls Parent's constructor
```

Error Handling

- Try-catch block

```
try {
   // normal code
} catch (err) {
   // handling error
}
```

Sync in JS

- **Synchronous execution**: Runs in a particular sequence, each instruction waits for the previous one to complete.
- **Asynchronous execution**: Executes the next instructions immediately and doesn't block the flow.

Callbacks

- Function passed as an argument to another function.

Callback Hell

- Nested callbacks forming a pyramid structure.
- Difficult to understand & manage.

Promises

- For eventual completion of a task.
- It is solution to callback hell.

```
let promise = new Promise((resolve, reject) => { ... });


// States
// Pending: result is undefined
// Resolved: result is a value (fulfilled)
// Rejected: result is an error object
```

Handlers

```
promise.then((res) => { ... });

promise.catch((err) => { ... });
```

Async-Await

- **Async function**: Always returns a promise.

```
async function myFunc() { ... }
```

Await: Pauses execution until the promise is settled.

IIFE: Immediately Invoked Function Expression

- An Immediately Invoked Function Expression (IIFE) is a JavaScript function that runs as soon as it is defined.
- It's a design pattern also known as a self-executing anonymous function and is used to create a private scope to avoid polluting the global scope.

```
(function() {
  // This code runs immediately
  var message = "Hello, World!";
  console.log(message);
})();
```

In this example:

- The function is defined within parentheses `()`.
- It is immediately invoked by the trailing `()`.

You can also pass parameters to an IIFE:

```
(function(name) {
   var message = "Hello, " + name + "!";
   console.log(message);
})("Gokul");
```

In this example, the IIFE takes a parameter `name` and prints a greeting message using the provided name.

Javascript ES6

- ES6, short for ECMAScript 6, is a version of the JavaScript language specification that was finalized in 2015.
- It brought significant enhancements and new features to JavaScript, making the language more powerful, expressive, and easier to work with.

New Features in ES6

The let keyword
The const keyword
Arrow Functions
The {a,b} = Operator
The [a,b] = Operator
The ... Operator
For/of
Map Objects
Set Objects
Classes
Promises
Symbol
Default Parameters
Function Rest Parameter
String.includes()
String.startsWith()
String.endsWith()

Array entries()
Array.from()
Array keys()
Array find()
Array findIndex()
Math.trunc
Math.sign
Math.cbrt
Math.log2
Math.log10
Number.EPSILON
Number.MIN_SAFE_INTEGER
Number.MAX_SAFE_INTEGER
Number.isInteger()
Number.isSafeInteger()
New Global Methods
JavaScript Modules

let and **const** were introduced as new variable declaration keywords, alongside the existing **var**

var:

- var is function-scoped.
- It means that a variable declared with var is accessible throughout the entire function in which it is defined.
- It allows redeclaration and reassignment.
- Variables declared with var are hoisted to the top of their scope during runtime.
- Example:

```
var x = 10;
if (true) {
  var y = 20;
  console.log(x); // Output: 10
}
console.log(y); // Output: 20
```

let

- let is block-scoped, meaning it's only accessible within the block it's declared in

(typically denoted by curly braces {}).

- It allows reassignment but not redeclaration within the same scope.
- Variables declared with let are not hoisted.

- Example

```
let a = 30;
if (true) {
  let b = 40;
  console.log(a); // Output: 30
  console.log(b); // Output: 40
}
console.log(a); // Output: 30
// console.log(b); // This will throw an error
```

const:

- const is also block-scoped like let.
- It requires an initializer and cannot be reassigned.
- However, for objects and arrays, the properties or elements of the object/array can still be modified.
- const are not hoisted.

Example

```
const PI = 3.14159;
console.log(PI); // Output: 3.14159
// PI = 3.14; // This will throw an error
```

Arrow Functions

- Arrow functions are a concise way to write functions in JavaScript, introduced in ECMAScript 6 (ES6).
- They provide a more compact syntax compared to traditional function expressions and offer some benefits in terms of lexical scoping and this binding

Syntax: Arrow functions have a concise syntax using the => arrow token.

```
// Syntax:
// (parameters) => { statements }
// or
// parameter => expression
// Examples:
let add = (a, b) => {
   return a + b;
};
let square = x => x * x;
```

```
// ES5
var x = function(x, y) {
  return x * y;
}
```

No this Binding:

- Arrow functions do not have their own this context.
- Instead, they inherit this from the enclosing lexical context.

```
function Person(name) {
   this.name = name;
   this.sayHello = () => {
      console.log(`Hello, my name is ${this.name}`);
   };
}
let person = new Person('John');
person.sayHello(); // Output: Hello, my name is John
```

Implicit Return:

- Arrow functions allow implicit return if the function body consists of a single expression.

```
let multiply = (a, b) => a * b;
```

Object Destructuring

- Object destructuring is a feature introduced in ECMAScript 6 (ES6) that allows you to extract multiple properties from an object and assign them to variables in a more concise way.
- It provides a shorthand syntax to access and unpack values from objects

```
// Object
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30
};
// Destructuring assignment
const { firstName, lastName, age } = person;
console.log(firstName); // Output: John
console.log(lastName); // Output: Doe
console.log(age); // Output: 30
```

Variable Assignment:

- You can use object destructuring to assign variables with the same name as the object's properties.
- The variables are assigned the corresponding values from the object.
- You can also rename the variables during destructuring.

```
const { firstName: fName, lastName: lName } = person;
console.log(fName); // Output: John
console.log(lName); // Output: Doe
```

Default Values:

- You can provide default values during destructuring in case a property is undefined.

```
const { city = 'Unknown' } = person;
console.log(city); // Output: Unknown
```

Nested Object Destructuring:

- You can destructure nested objects by specifying the property path.

```
const user = {
  name: 'John',
  age: 30,
  address: {
    city: 'New York',
    country: 'USA'
  }
};
const { name, address: { city, country } } = user;
console.log(name); // Output: John
console.log(city); // Output: New York
console.log(country); // Output: USA
```

Rest Properties:

- You can use the rest syntax (...) to gather remaining properties into a new object.

```
const { firstName, ...rest } = person;
console.log(firstName); // Output: John
console.log(rest); // Output: { lastName: 'Doe', age: 30 }
```

Array Destructuring

Destructuring assignment makes it easy to assign array values and object properties to variables

```
// Create an Array
const fruits = ["Banana", "Orange", "Apple", "Mango"];

// Destructuring Assignment
let [fruit1, fruit2] = fruits;

console.log(fruit1); // Output: Banana

console.log(fruit2); // Output: Orange
```

The Spread (...) Operator

- The ... operator expands an iterable (like an array) into more elements:

```
const x1 = [1, 2, 3];

const x2 = [4, 5, 6];

const x3 = [7, 8, 9];

// Merge arrays using the spread operator

const year = [...x1, ...x2, ...x3];

console.log(year); // Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
const numbers = [23, 55, 21, 87, 56];

let maxValue = Math.max(...numbers);

console.log(maxValue); // Output: 87
```

for/of

- The for...of loop is a feature introduced in ECMAScript 6 (ES6) that provides a convenient way to iterate over iterable objects such as arrays, strings, maps, sets, and other iterable objects.

- Unlike the traditional for loop, for...of loop abstracts away the details of indexing and provides a simpler syntax for iterating over the elements of an iterable.

Iterating Arrays:

```
const numbers = [1, 2, 3, 4, 5];
for (const number of numbers) {
  console.log(number);
}
// Output:
// 1
// 2
// 3
// 4
// 5
```

Iterating Strings:

```
const message = "Hello";
for (const char of message) {
  console.log(char);
}
// Output:
// H
// e
// l
// l
// o
```

Iterating Maps:

```
const map = new Map();
map.set('a', 1);
map.set('b', 2);
map.set('c', 3);
for (const [key, value] of map) {
   console.log(` ${key} = ${value}` );
}
// Output:
// a = 1
// b = 2
// c = 3
```

```
const fruits = new Map([
 ["apples", 500],
 ["bananas", 300],
 ["oranges", 200]
]);

let numb = fruits.get("apples");

console.log(numb); // Output: 500
```

Iterating Sets:

```
const set = new Set([1, 2, 3, 4, 5]);
for (const number of set) {
   console.log(number);
}
// Output:
// 1
// 2
// 3
// 4
// 5
```

```
// Create a Set
const letters = new Set();
// Add some values to the Set
letters.add("a");
letters.add("b");
letters.add("c");
// Display the contents of the Set
console.log(letters); // Output: Set { 'a', 'b', 'c' }
```

Classes

- In JavaScript, classes were introduced in ECMAScript 6 (ES6) to provide a more convenient and structured way to define objects and their behavior, similar to classes in other object-oriented programming languages like Java and Python.

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  // Instance method
  sayHello() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
  // Static method
  static greet() {
    console.log("Hello from the Person class!");
  }
}
//Creating Objects
const person1 = new Person('Alice', 30);
const person2 = new Person('Bob', 25);
//Instance method
person1.sayHello(); // Output: Hello, my name is Alice and I am 30 years old.
person2.sayHello(); // Output: Hello, my name is Bob and I am 25 years old.
//static method
Person.greet(); // Output: Hello from the Person class!
```

- Classes support inheritance through the extends keyword.
- Child classes can inherit properties and methods from parent classes.

```
class Student extends Person {
  constructor(name, age, grade) {
    super(name, age); // Call the parent class constructor
    this.grade = grade;
  }
  // Override parent class method
  sayHello() {
    console.log(`Hello, my name is ${this.name}, I am ${this.age} years old, and I am in grade ${this.grade}.`);
  }
}
```

Promises

- JavaScript Promises are objects used for asynchronous operations.
- They represent a future value or an eventual completion (or failure) of an asynchronous operation and allow you to handle the result or error once the operation finishes.
- Promises provide a cleaner and more readable way to work with asynchronous code compared to traditional callback-based approaches.

Creating a Promise:

- To create a Promise, you instantiate a new Promise object and pass a function (usually referred to as the executor function) as an argument.
- The executor function takes two parameters: resolve and reject, which are functions used to fulfill or reject the Promise.

```
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation
  setTimeout(() => {
    const randomNumber = Math.random();
    if (randomNumber > 0.5) {
      resolve(randomNumber); // Fulfill the Promise with a value
    } else {
      reject(new Error('Random number is too small')); // Reject the Promise with an error
    }
  }, 1000);
});
```

Consuming a Promise:

- Once a Promise is created, you can consume its result using the then and catch methods.
- The then method is called when the Promise is fulfilled, and the catch method is called when the Promise is rejected.

```
myPromise
  .then(result => {
    console.log('Success:', result);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Chaining Promises:

- Multiple asynchronous operations can be chained by returning a Promise from within a then handler.
- This allows you to create sequences of asynchronous tasks.

```
myPromise
  .then(result => {
    console.log('First Promise fulfilled:', result);
    return result * 2; // Return a new Promise
  })
  .then(result => {
    console.log('Second Promise fulfilled:', result);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Promise.all:

- The Promise.all method is used to handle multiple Promises concurrently.
- It takes an array of Promises as input and returns a single Promise that resolves when all of the input Promises have resolved, or rejects if any of the input Promises reject.

```
const promise1 = fetch('https://api.example.com/data1');
const promise2 = fetch('https://api.example.com/data2');
Promise.all([promise1, promise2])
  .then(responses => {
    // Handle responses
  })
  .catch(error => {
    // Handle error
  });
```

The Symbol Type

- A JavaScript Symbol is a primitive data type just like Number, String, or Boolean.
- It represents a unique "hidden" identifier that no other code can accidentally access.
- For instance, if different coders want to add a person.id property to a person object belonging to a third-party code, they could mix each others values.

```
const person = {
 firstName: "John",
 lastName: "Doe",
 age: 50,
 eyeColor: "blue"
};

let id = Symbol('id');
person[id] = 140353;
// Now person[id] = 140353
// but person.id is still undefined
```

Note:

Symbols are always unique.

If you create two symbols with the same description they will have different values:

Symbol("id") == Symbol("id"); // false

Default Parameter Values

ES6 allows function parameters to have default values.

```
function myFunction(x, y = 10) {
  // y is 10 if not passed or undefined
  return x + y;
}
myFunction(5); // will return 15
```

```
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

String.includes()

- The includes() method returns true if a string contains a specified value, otherwise false:

```
let text = "Hello world, welcome to the universe.";
text.includes("world")    // Returns true
```

String.startsWith()

- The startsWith() method returns true if a string begins with a specified value, otherwise false:

```
let text = "Hello world, welcome to the universe.";

text.startsWith("Hello")   // Returns true
```

String.endsWith()

- The endsWith() method returns true if a string ends with a specified value, otherwise false:

```
var text = "John Doe";
text.endsWith("Doe")    // Returns true
```

Array entries()

Example

Create an Array Iterator, and then iterate over the key/value pairs:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const f = fruits.entries();

for (let x of f) {
  document.getElementById("demo").innerHTML += x;
}

//Output:

//[0, "Banana"]
//[1, "Orange"]
//[2, "Apple"]
//[3, "Mango"]
```

The entries() method does not change the original array.

Array.from()

The Array.from() method returns an Array object from any object with a length property or any iterable object.

Example

Create an Array from a String:

```
Array.from("ABCDEFG")   // Returns [A,B,C,D,E,F,G]
```

Array keys()

The keys() method returns an Array Iterator object with the keys of an array.

Example

Create an Array Iterator object, containing the keys of the array:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const keys = fruits.keys();

let text = "";
for (let x of keys) {
  text += x + "<br>";
}
```

Array find()

The find() method returns the value of the first array element that passes a test function.

This example finds (returns the value of) the first element that is larger than 18:

Example

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Note that the function takes 3 arguments:

The item value

The item index

The array itself

Array findIndex()

The findIndex() method returns the index of the first array element that passes a test function.

This example finds the index of the first element that is larger than 18:

Example

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.findIndex(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

ES6 added the following methods to the Math object:

Math.trunc()

Math.sign()

Math.cbrt()

Math.log2()

Math.log10()

The Math.trunc() Method

Math.trunc(x) returns the integer part of x:

Example

```
Math.trunc(4.9);   // returns 4
Math.trunc(4.7);   // returns 4
Math.trunc(4.4);   // returns 4
Math.trunc(4.2);   // returns 4
Math.trunc(-4.2);   // returns -4
```

The Math.sign() Method

Math.sign(x) returns if x is negative, null or positive:

Example

```
Math.sign(-4);   // returns -1
Math.sign(0);   // returns 0
Math.sign(4);   // returns 1
```

Node.js

- Node.js is an open source server environment.
- Node.js allows you to run JavaScript on the server.

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

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World!');
}).listen(8080);
```

Why Node.js

- Traditional server-side languages like PHP or ASP follow a sequential process for handling file requests.
- They send the request to the file system, wait for it to open and read the file, and then return the content to the client.
- During this waiting period, the server can't handle other tasks.
- Node.js, on the other hand, takes an asynchronous approach.
- It also sends the file request to the file system but doesn't wait for it to complete.
- Instead, Node.js continues with other tasks, making it highly efficient.
- When the file system finishes reading the file, Node.js retrieves the content and returns it to the client.
- This asynchronous, non-blocking behavior allows Node.js to handle multiple requests simultaneously and efficiently utilize system resources.

Introduction to Asynchronous Programming and Callbacks in Node.js

Asynchronous Programming:

- Asynchronous programming allows Node.js to handle multiple operations concurrently, making efficient use of resources and enabling non-blocking I/O operations.

- In Node.js, many operations like reading files, making HTTP requests, and interacting with databases are inherently asynchronous.
- Asynchronous programming is essential for building scalable and performant Node.js applications, especially for handling high-concurrency scenarios.

Callbacks:

- Callbacks are functions passed as arguments to other functions and are executed later when a specific operation completes.
- In Node.js, callbacks are a common mechanism for handling asynchronous operations.
- Callback functions typically have an error-first argument convention, where the first argument is an error object (if an error occurred), and subsequent arguments represent the results of the operation.
- **Reading a File Asynchronously:**

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

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

- Asynchronous programming in Node.js enables efficient handling of I/O-bound operations.
- Callbacks are a common mechanism for handling asynchronous operations, following an error-first convention.
- Callback hell can be mitigated by using named functions, promises, or async/await syntax for better code readability and maintainability.

Event Loop and Timers in Node.js

Event Loop:

- The Event Loop is a crucial part of Node.js responsible for handling asynchronous operations and I/O events.
- It continuously checks for tasks in the event queue and processes them one by one, allowing Node.js to handle multiple operations concurrently.
- The Event Loop ensures non-blocking behavior, making Node.js highly scalable and efficient for I/O-bound applications.

Timers:

- Timers in Node.js allow scheduling code execution at specific times or after certain intervals.
- Node.js provides several timer functions, including setTimeout, setInterval, and setImmediate, for scheduling code execution asynchronously.

```
console.log('Start');

setTimeout(() => {

  console.log('Delayed execution');

}, 2000);

console.log('End');
```

- The Event Loop is responsible for handling asynchronous operations and I/O events in Node.js.
- Timers like setTimeout, setInterval, and setImmediate allow scheduling code execution asynchronously.
- Understanding the Event Loop and timers is essential for building efficient and scalable Node.js applications.

Node.js Modules

- Modules are similar to JavaScript Libraries.
- Set of functions needed to included in the program

To include a module use the keyword: require()

var http = require('http');

The application has access to http module and can create a server.

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

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('Hello World!');
}).listen(8080);
```

Creating module and using it outside the program

```
exports.myDateTime = function () {
  return Date();
};
```

Use the exports keyword to make properties and methods available outside the module file.

Save the code above in a file called "myfirstmodule.js"

```
var http = require('http');
var dt = require('./myfirstmodule');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write("The date and time are currently: " + dt.myDateTime());
  res.end();
}).listen(8080);
```

HTTP Module – Node.js

Creating HTTP Servers:

- The HTTP module in Node.js allows you to create HTTP servers that can listen for incoming requests and respond to them.

- You can create both HTTP and HTTPS servers using the http and https modules respectively.

Making HTTP Requests:

- In addition to creating servers, the HTTP module enables you to make outgoing HTTP requests to other servers or APIs.

Event-Driven Architecture:

- The HTTP module follows an event-driven architecture, where you define event handlers for different stages of the HTTP request-response cycle.

Creating an HTTP Server:

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

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!\n');
});

const PORT = 3000;

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

- The Node.js HTTP module enables you to create HTTP servers and make HTTP requests.
- HTTP servers created with Node.js follow an event-driven architecture, allowing you to define event handlers for different stages of the request-response cycle.
- Understanding how to create HTTP servers and make HTTP requests is essential for building web applications and APIs with Node.js.

npm package manager

npm is the standard package manager for Node.js.

npm manages downloads of dependencies of the project

Installing all dependencies

If a project has a package.json file, by running

npm install

- It will install everything the project needs, in the node_modules folder, creating it if it's not existing already

Installing a single package

Install a specific package by running

```
npm install <package-name>
```

Updating packages

Updating is also made easy, by running

```
npm update
npm update <package-name> //single package to update
npm install <package-name>@<version>
npm run
npm start
npm run watch
npm run dev
npm run prod
```

- The package.json file serves as the core of the Node.js system.
- It acts as the manifest file for every Node.js project, encapsulating crucial metadata.
- Understanding, learning, and effectively working with Node.js hinge upon comprehending the package.json file.
- It encompasses essential information about the project, guiding its development, dependencies, and scripts.
- The package.json file contains the metadata information.

- This metadata information in **package.json** file can be categorized into below categories.
- **Identifying metadata properties:** It basically consist of the properties to identify the module/project such as the name of the project, current version of the module, license, author of the project, description about the project etc.
- **Functional metadata properties:** As the name suggests, it consists of the functional values/properties of the project/module such as the entry/starting point of the module, dependencies in project, scripts being used, repository links of Node project etc.

Create a package.json file

1. **npm init**
2. **Writing directly to file**

```
{
 "name": "my-node-project",
 "version": "1.0.0",
 "description": "A simple Node.js project",
 "main": "index.js",
 "scripts": {
  "start": "node index.js",
  "test": "jest"
 },
 "dependencies": {
  "express": "^4.17.1",
  "lodash": "^4.17.21"
 },
 "devDependencies": {
  "jest": "^27.0.0"
 },
 "author": "John Doe",
 "license": "MIT"
}
```

File System

- Node.js as a File Server
- The Node.js file system module allows you to work with the file system on your computer.
- To include the File System module, use the require() method:

var fs = require('fs');

Common use for the File System module:

1. Read files
2. Create files
3. Update files
4. Delete files
5. Rename files

Read Files

The fs.readFile() method is used to read files on your computer.

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  fs.readFile('demofile1.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(8080);
```

Create Files

The File System module has methods for creating new files:

fs.appendFile()

fs.open()

fs.writeFile()

The **fs.appendFile()** method appends specified content to a file. If the file does not exist, the file will be created:

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

fs.open('mynewfile2.txt', 'w', function (err, file) {
  if (err) throw err;
  console.log('Saved!');
});
```

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

fs.writeFile('mynewfile3.txt', 'Hello content!', function (err) {
  if (err) throw err;
  console.log('Saved!');
});
```

Update Files

The File System module has methods for updating files:

fs.appendFile()

fs.writeFile()

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

fs.appendFile('mynewfile1.txt', ' This is my text.', function (err) {
  if (err) throw err;
  console.log('Updated!');
});
```

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

fs.writeFile('mynewfile3.txt', 'This is my text', function (err) {
  if (err) throw err;
  console.log('Replaced!');
});
```

Delete Files

To delete a file with the File System module, use the **fs.unlink()** method.

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

fs.unlink('mynewfile2.txt', function (err) {
  if (err) throw err;
  console.log('File deleted!');
});
```

Rename Files

To rename a file with the File System module, use the **fs.rename()** method.

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

fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err) {
  if (err) throw err;
  console.log('File Renamed!');
});
```

Node.js - Express Framework

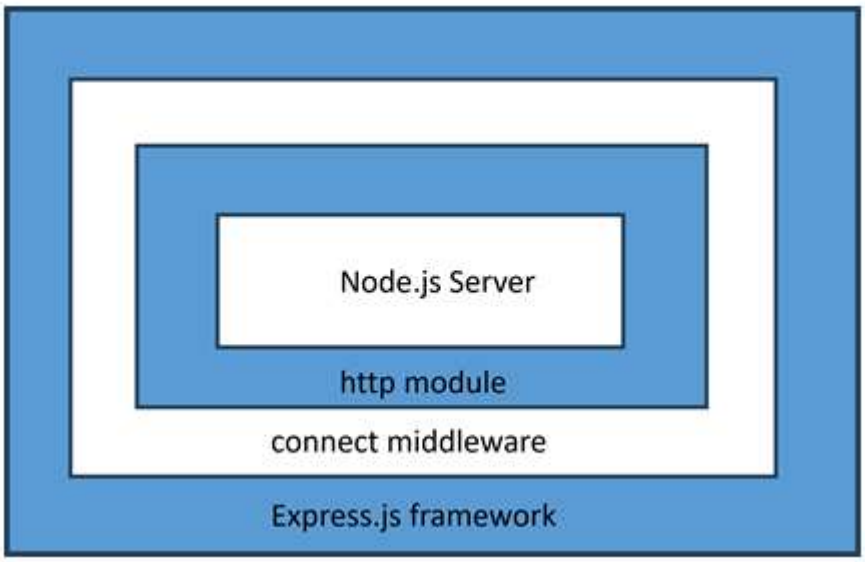
Introduction:

- Express.js is a minimalist web framework for Node.js that provides a robust set of features for building web applications and APIs.
- It simplifies the process of creating routes, handling requests and responses, and managing middleware.

Key Features:

1. **Routing**: Define routes to handle different HTTP requests and URL paths.
2. **Middleware**: Use middleware functions to process requests before they reach route handlers.
3. **Template Engines**: Integrate with template engines like EJS, Handlebars, or Pug to generate dynamic HTML content.
4. **Static File Serving**: Serve static assets such as CSS, JavaScript, and images.
5. **Error Handling**: Implement error-handling middleware to catch and handle errors gracefully.

6. **HTTP Helpers**: Use built-in HTTP helper methods for common tasks like sending responses, setting headers, etc.



Installing Express

The Express.js package is available on npm package repository.

```
D:\expressApp> npm init
D:\expressApp> npm install express --save
```

Application object

- An object of the top level express class denotes the application object. It is instantiated by the following statement –

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

- The Application object handles important tasks such as handling HTTP requests, rendering HTML views, and configuring middleware etc.
- The app.listen() method creates the Node.js web server at the specified host and port. It encapsulates the createServer() method in http module of Node.js API.

```
app.listen(port, callback);
```

Basic Routing

- The app object handles HTTP requests GET, POST, PUT and DELETE with app.get(), app.post(), app.put() and app.delete() method respectively.
- The HTTP request and HTTP response objects are provided as arguments to these methods by the NodeJS server.
- The first parameter to these methods is a string that represents the endpoint of the URL.
- These methods are asynchronous, and invoke a callback by passing the request and response objects.

GET method

```
app.get('/', function (req, res) {
   res.send('Hello World');
})
```

- **Request Object** – The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- **Response Object** – The response object represents the HTTP response that an Express app sends when it gets an HTTP request. The send() method of the response object formulates the server's response to the client.
- The response object also has a **sendFile()** method that sends the contents of a given file as the response.

```
res.sendFile(path)
```

Basic Express Server

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
  res.send('Hello, World!');
});
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Route Parameters

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID: ${userId}`);
});
```

Middleware Example

```
// Custom middleware function
const logger = (req, res, next) => {
  console.log(`Request URL: ${req.url}`);
  next(); // Call next middleware in the chain
};
app.use(logger);
```

Template Engine Integration (EJS)

```
// Set view engine
app.set('view engine', 'ejs');
// Render EJS template
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.render('user', { userId });
});
```

Static File Serving

```
// Serve static files from the 'public' directory
app.use(express.static('public'));
```

Error Handling Middleware

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

Routing with Express Router

```
// Create a router instance

const router = express.Router();

// Define routes

router.get('/users', (req, res) => {

  res.send('List of users');

});

// Mount the router on a specific path

app.use('/api', router);
```

- Express.js is a powerful web framework for Node.js, providing features for routing, middleware, template engines, static file serving, and more.
- It simplifies the process of building web applications and APIs by providing a clean and intuitive API.
- By understanding and leveraging the features of Express.js, you can develop scalable, efficient, and maintainable Node.js web applications.

Save the following HTML script as index.html in the root folder of the express app.

```
<html>

<body>

<h2 style="text-align: center;">Hello World</h2>

</body>

</html>
```

Change the index.js file to the code below –

```
var express = require('express');
var app = express();
var path = require('path');
app.get('/', function (req, res) { res.sendFile(path.join(__dirname,"index.html"));
})
var server = app.listen(5000, function () {
console.log("Express App running at http://127.0.0.1:5000/");
})
```

Run the above program and visit http://localhost:5000/, the browser shows Hello World message as below:

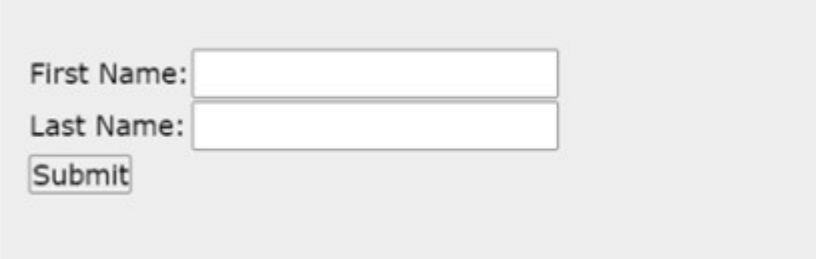
Let us use sendFile() method to display a HTML form in the index.html file.

```
<html>
<body>
<form action = "/process_get" method = "GET">
First Name: <input type = "text" name = "first_name"> <br>
Last Name: <input type = "text" name = "last_name"> <br>
<input type = "submit" value = "Submit">
</form>
</body>
</html>
```

The above form submits the data to /process_get endpoint, with GET method. Hence we need to provide a app.get() method that gets called when the form is submitted.

```
app.get('/process_get', function (req, res) {
// Prepare output in JSON format
response = { first_name:req.query.first_name, last_name:req.query.last_name };
console.log(response);
res.end(JSON.stringify(response));
 })
```

The form data is included in the request object. This method retrieves the data from request.query array, and sends it as a response to the client.



POST method

```
<html>
<body>
<form action = "/process_POST" method = "POST">
First Name: <input type = "text" name = "first_name"> <br>
Last Name: <input type = "text" name = "last_name"> <br>
<input type = "submit" value = "Submit">
</form>
</body>
</html>
```

To handle the POST data, we need to install the body-parser package from npm.

Use the following command.

npm install body-parser –save

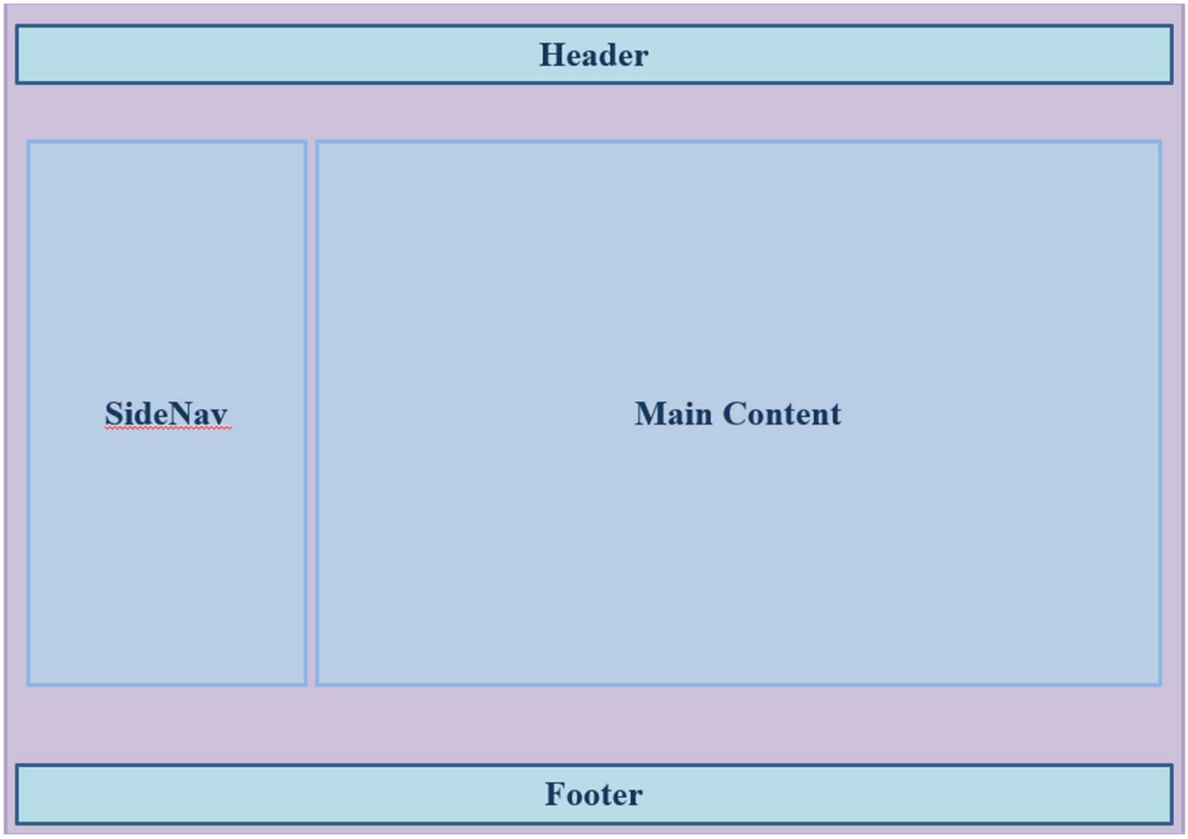
```
var express = require('express'); var app = express();
var path = require('path');
var bodyParser = require('body-parser');
// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })
app.use(express.static('public'));
app.get('/', function (req, res) { res.sendFile(path.join(__dirname,"index.html"));
})
app.get('/process_get', function (req, res) {
// Prepare output in JSON format
response = { first_name:req.query.first_name, last_name:req.query.last_name
};
console.log(response);
res.end(JSON.stringify(response));
})
app.post("/process_post", )
var server = app.listen(5000, function () {
console.log("Express App running at http://127.0.0.1:5000/");
})
```

React

Introduction

- React is Java Library
- Mainly used for designing UI
- Developed by Facebook

Component



React Features

- Reusable code
- React is declarative
- React.lib actually build
- Rendering, updating- automatic
- React native-mobile application
- ES6 is used

React Commands

Open teriminal in VS Code or node Terminal

npm (npx) create-react-app APP_NAME

cd APP_NAME

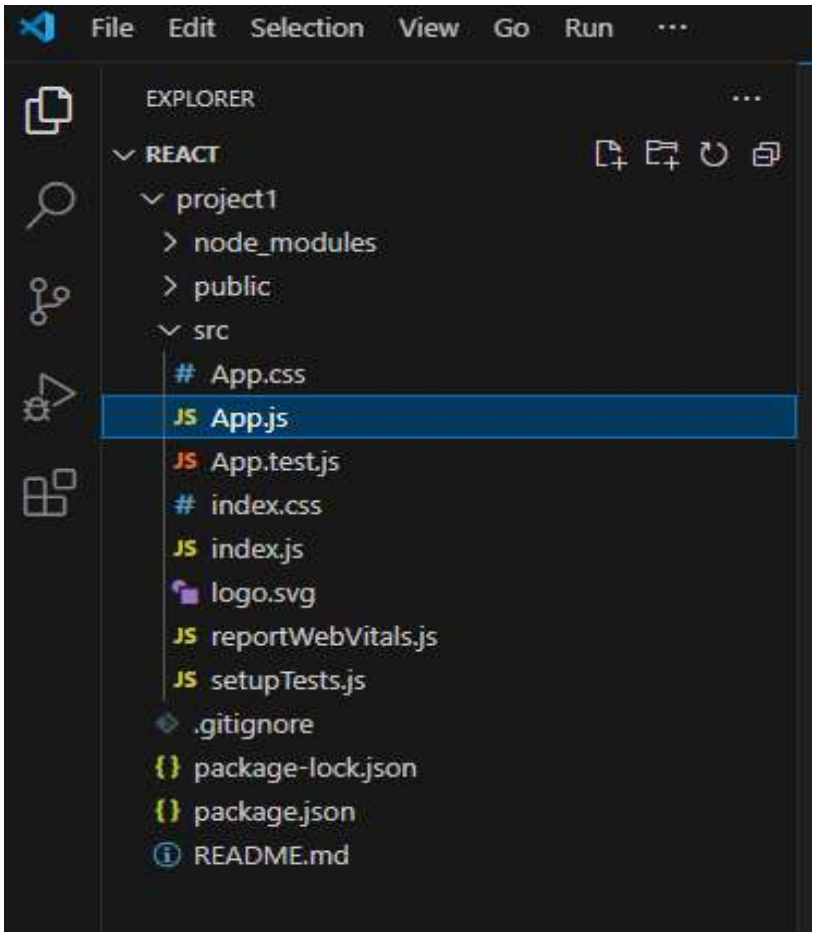
npm start

Npx create-react-app appname

npx – npm package runner

npm create-react-app –g
Create-react-app app name

Folder Structure



Folder Structure

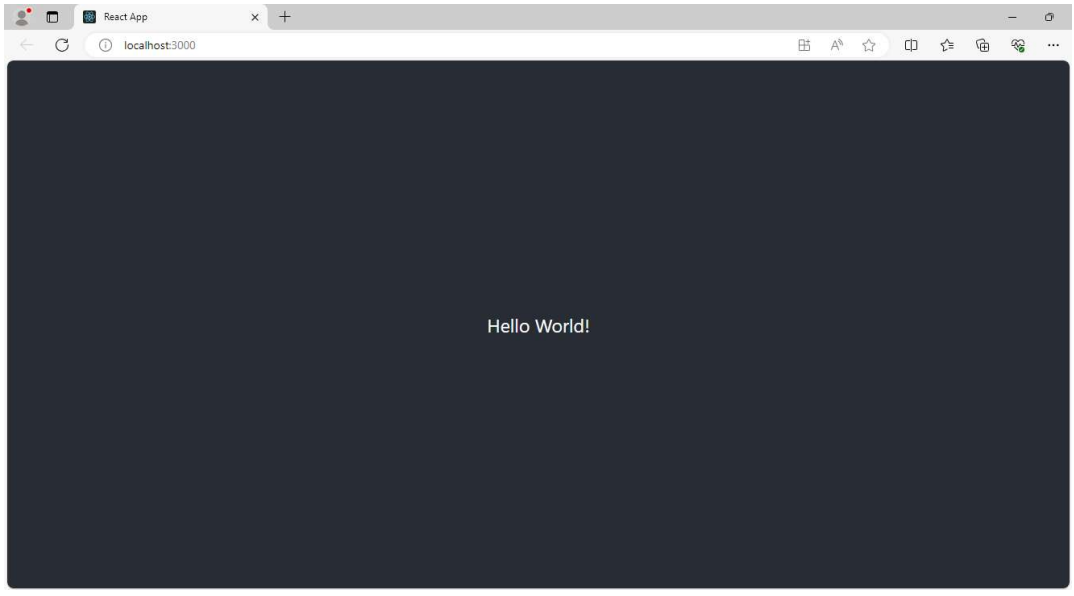
- my-react-app/
  - node_modules/ // Installed dependencies
  - public/ // Public assets and HTML template
    - index.html // Main HTML file
    - favicon.ico // Favicon
  - src/ // Source code
    - index.js // Entry point for React app
    - App.js // Main component
    - App.css // Styles for App component
    - components/ // Reusable components
    - assets/ // Static assets (images, fonts, etc.)

    ...
  - package.json // Project configuration and dependencies
  - README.md // Project documentation

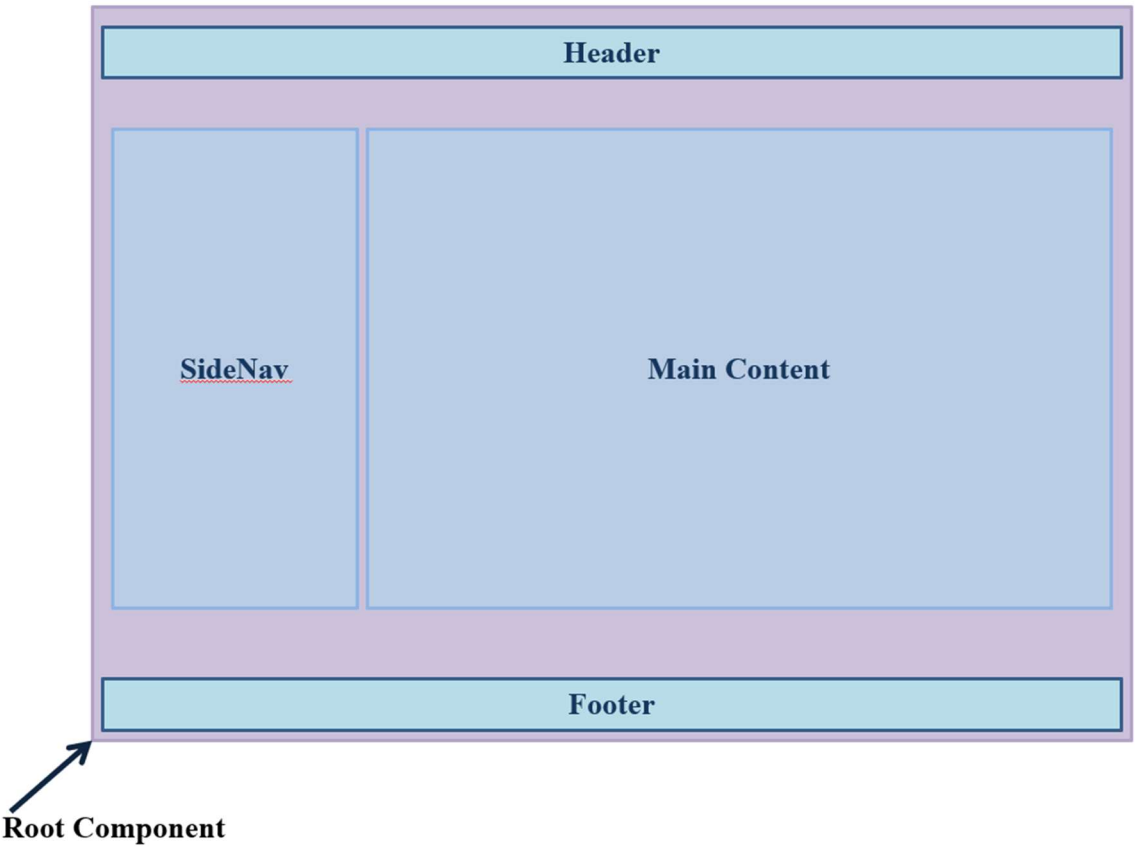
Hello World program

```
//App.js
import './App.css';

function App() {
 return (
  <div className="App">
   <header className="App-header">
    <p>
     Hello World!
    </p>
   </header>
  </div>
 );
}
export default App;
```



Component



- In React, components are the building blocks of the UI.
- They are reusable, self-contained units that manage their own state and can be composed to build more complex user interfaces.
- There are two main types of components in React:
    - **Function components**
    - **Class components**.

Function Components

- Function components are the simplest form of a React component.
- They are JavaScript functions that accept props (short for properties) as an argument and return React elements describing what should appear on the screen.

Class Components

- Class components are JavaScript classes that extend from React.Component.
- They have additional features such as local state and lifecycle methods.

Functional Component

```
//Greeting.js
import React from 'react';
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
export default Greeting;
```

Class Component

```
//Message.js
import React, { Component } from 'react';
class Message extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
export default Message;
```

```
// App.js
import React from 'react';
import Greeting from './Components/Greeting';
import Counter from './Components/Counter';
import Message from './Components/Message';
function App() {
 return (
  <div>
   <Greeting name="Alice" />
   <Message name="Bob"/>
  </div>
 );
}

export default App;
```

- Components are reusable
- Sidenav can be used for both left and right
- Comps can contain other comps
- Components are written in javascript js file
- AppComponent in App.js

```
<div>
   <Greeting name="Alice" />
   <Greeting name="Ken" />
   <Message name="Bob"/>
   <Message name="Ben"/>
  </div>
```

Hello, Alice!

Hello, Ken!

Hello, Bob

Hello, Ben

Render Function in React Class Components

- In React class components, the render method is a required method that returns the JSX (JavaScript XML) markup to be rendered to the DOM.
- It is where the UI of component is defined based on its current state and props.

```
render() {
return (
<div>
<h1>Hello, World!</h1>
<p>This is a simple React component.</p> </div>
 );
}
```

State in React

- Why state ?
- States are mutable data that belong to a component and can influence its rendering.
- **Functional Components:**
- **useState Hook**: Functional components use the useState Hook to manage state.
- **Class Components:**
- **this.state**: Class components use this.state to manage state.

Initializing State:

- In functional components, state is initialized using the useState Hook, while in class components, it's initialized in the constructor using this.state.

Updating State:

- State can be updated using setState in class components and the setter function returned by useState Hook in functional components.

Immutability:

- State should be updated immutably. Direct mutation of state in React is not recommended.

Async Updates:

- State updates may be asynchronous in React. Always use the functional form of setState or pass a function to setState when the new state is computed based on the previous state.

Avoid excessive use of stateful components, as excessive state can lead to decreased performance and increased complexity.

Functional Component

```
//Counter.js
import React, { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0); // Initializing state variable
  const incrementCount = () => {
    setCount(count + 1); // Updating state variable
  };
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}
export default Counter;
```

Class Component

```
//Counter.js
import React, { Component } from 'react';
class Counter extends Component {
 constructor(props) {
  super(props);
  this.state = { count: 0 }; // Initializing state in constructor
 }
 incrementCount = () => {
  this.setState({ count: this.state.count + 1 }); // Updating state
 };
 render() {
  return (
   <div>
    <p>Count: {this.state.count}</p>
    <button onClick={this.incrementCount}>Increment</button>
   </div>
  );
 }
}
export default Counter;
```

Events

- React events are written in camelCase
- syntax: **onClick, onSubmit**.
- React event handlers are written inside curly braces:

<button onClick={handleIncrement}>Count!</button>

HTML:

<button onclick="count()">Clcik!</button>

```
//ClickEvent.js
import React from 'react';

function ClickEvent(props) {
  const click=()=>{
    alert("Clicked by " + props.name);
  }
  return (
    <div>
      <button onClick={click}>Click the button</button>
    </div>
  );
}

export default ClickEvent;
```

```
//ClickEventsWithPara.js
import React from 'react';

function ClickEventsWithPara() {
  const click=(n)=>{
    alert(n);
  }
  return (
    <button onClick={()=>click('clicked')}>click here</button>
  );
}
export default ClickEventsWithPara;
```

```
//App.js
function App() {
 return (
  <div className="App">
   <ClickEvent name="Ben" />
   <ClickEventsWithPara/>
  </div>
 );
}
export default App;
```

Conditional Rendering

- Conditional rendering is a powerful feature in React that allows to display different components or content based on certain conditions.

Using the && Operator:

The logical **AND (&&)** operator can be used for conditional rendering in React.

It works by short-circuiting and only rendering the component or element on the right side if the condition on the left side is true.

{condition && <Component />}

```
function ShoppingCart(props) {
 const items = props.items;
 return (
  <div>
   {items.length > 0 && (
    <h2>You have {items.length} items in your cart.</h2>
   )}
  </div>
 );
}
```

Using the Ternary Operator:

- The ternary operator **(condition ? trueValue : falseValue)** is another way to conditionally render content in React.
- It evaluates the condition and renders the true value if the condition is true, otherwise, it renders the false value.
- **{condition ? <TrueComponent /> : <FalseComponent />}**

```
function Greeting(props) {
 const isLoggedIn = props.isLoggedIn;
 return (
  <div>
   {isLoggedIn ? (
    <h1>Welcome back!</h1>
   ) : (
    <h1>Please log in.</h1>
   )}
  </div>
 );
}
```

Using If-Else Statements (in JavaScript):

- Traditional JavaScript if-else statements for conditional rendering can be used, but they must be used within the body of a function component or method.

```
function Greeting(props) {
 if (props.isLoggedIn) {
  return <h1>Welcome back!</h1>;
 } else {
  return <h1>Please log in.</h1>;
 }
}
```

Lists in React:

- In React, you often need to render lists of elements dynamically.
- You can use the map() function to iterate over an array of data and generate a list of React elements based on that data.

```
const elements = data.map((item) => (
  <ElementComponent key={uniqueKey} prop1={item.prop1} prop2={item.prop2} />
));
```

- data: An array of data you want to render as a list.
- map((item) => ...): Iterates over each item in the array.
- <ElementComponent>: The React component you want to render for each item in the list.
- key={uniqueKey}: The key prop is required for each element in the list.
- It helps React identify which items have changed, are added, or are removed. It should be a unique identifier for each item in the list.
- prop1={item.prop1} prop2={item.prop2}: Passes props to the ElementComponent based on the data in each item.

Keys in React

- Keys are special attributes that provide a unique identity to each element in a list.
- They help React identify which items have changed, are added, or are removed.
- Keys should be stable, predictable, and unique among siblings.
- **<Element key={uniqueKey} />**

- key: The special attribute used to assign a unique identifier to each element in the list.

- uniqueKey:It's often based on the data being rendered, such as an ID from a database or an index in the array.
- Avoid using array indexes as keys if the order of items may change, as it can negatively impact performance and cause issues with component state.

```
function TodoList({ todos }) {
  return (
    <ul>
      {todos.map((todo) => (
        <li key={todo.id}>{todo.text}</li>
      ))}
    </ul>
  );
}
const todos = [
  { id: 1, text: 'Learn React' },
  { id: 2, text: 'Build a todo app' },
  { id: 3, text: 'Deploy to production' }
];
function App() {
  return <TodoList todos={todos} />;
}
```

Forms & Input

Handling Forms

- Handling forms is about how you handle the data when it changes value or gets submitted.
- In HTML, form data is usually handled by the DOM.
- In React, form data is usually handled by the components.
- When the data is handled by the components, all the data is stored in the component state.
- Can control changes by adding event handlers in the onChange attribute.
- Can use the useState Hook to keep track of each inputs value and provide a "single source of truth" for the entire application.

React Router

- Create React App doesn't include page routing.
- React Router is the most popular solution.

Add React Router

- To add React Router in your application, run this in the terminal from the root directory of the application:

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

Router Program

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch, Link } from 'react-router-dom';
const Home = () => (
 <div>
  <h2>Home</h2>
  <p>Welcome to the homepage!</p>
 </div>
);
const About = () => (
 <div>
  <h2>About</h2>
  <p>This is the about page.</p>
 </div>
);
const Contact = () => (
 <div>
  <h2>Contact</h2>
  <p>Contact us at contact@example.com</p>
 </div>
);
```

```
const App = () => (
 <Router>
  <div>
   <nav>
    <ul>
     <li><Link to="/">Home</Link></li>
     <li><Link to="/about">About</Link></li>
     <li><Link to="/contact">Contact</Link></li>
    </ul>
   </nav>
   <Switch>
    <Route path="/about">
     <About />
    </Route>
    <Route path="/contact">
     <Contact />
    </Route>
    <Route path="/">
     <Home />
    </Route>
   </Switch>
  </div>
 </Router>
);
export default App;
```

Redux

- Redux is a predictable state container for JavaScript applications, primarily used with frameworks like React for managing application state in a centralized manner.
- It helps to write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

Key Concepts in Redux:

- **Store**: The central piece of Redux is the store, which holds the application's state. You can think of it as a big JavaScript object where all of your application's state resides.
- **Actions**: Actions are payloads of information that send data from your application to your Redux store. They are plain JavaScript objects and must have a type property indicating the type of action being performed.

```
const increment = () => ({
  type: 'INCREMENT'
});
const decrement = () => ({
  type: 'DECREMENT'
});
const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
};
```

Reducers:

- Reducers specify how the application's state changes in response to actions sent to the store.
- A reducer is a pure function that takes the previous state and an action, and returns the next state.
- It's important that reducers are pure functions without side effects, meaning they should not mutate the state directly but return a new state object. Example reducer:

Store Methods: The store has several important methods:

- getState(): Retrieves the current state of the Redux store.
- dispatch(action): Dispatches an action to the Redux store. This is how you trigger state changes.
- subscribe(listener): Adds a change listener that will be called any time an action is dispatched, and some part of the state tree may have changed. You can use this to update your UI in response to state changes.

Let's create a simple Redux counter application using React to demonstrate the concepts:

Setup Redux and React:

First, ensure you have Redux and React installed:

```
npm install redux react-redux
```

Implement Redux Store and Reducer:

Create a file counterReducer.js:

```
const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
};
export default counterReducer;
```

Create Redux Store and Connect to React:

In App.js:

```
import React from 'react';
import { createStore } from 'redux';
import { Provider, useDispatch, useSelector } from 'react-redux';
import counterReducer from './counterReducer';
const store = createStore(counterReducer);
const App = () => {
 return (
  <Provider store={store}>
   <Counter />
  </Provider>
 );
};
const Counter = () => {
 const count = useSelector(state => state);
 const dispatch = useDispatch();
 return (
  <div>
   <h1>Counter</h1>
   <p>Count: {count}</p>
   <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
   <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
  </div>
 );
};
export default App;
```

Explanation:

Store Creation: createStore(counterReducer) creates a Redux store with our counterReducer as its reducer.

Provider: <Provider store={store}> makes the Redux store available to any nested components that have been wrapped in the connect() function.

useSelector: Hook from React-Redux to extract data from the Redux store state, in this case, state => state simply returns the current count.

useDispatch: Hook from React-Redux to dispatch actions to the Redux store.

Steps to follow:

1. Open VS code, Mysql Workbench
2. Open Terminal in VS Code(Desktop) -> mkdir Form_App
3. cd From_App
4. mkdir frontend -> cd frontend npx creat-react-app .
5. Open mysql workbench: create schema Db-Name;
6. Use Db-Name;
7. cd.. And enter the Application directory in the VS Code
8. mkdir backend -> cd backend
9. npm init –y
10. npm install express body-parser cors mysql
11. Inside backend create file server.js

backend/server.js

```
//Importing the dependencies
const express = require('express');
const bodyParser = require('body-parser');
const mysql = require ('mysql');
const cors = require('cors');
//Instantiating the express and PORT number
const app = express();
const PORT =  3000;
//using cors and bodyparser.json
app.use(cors());
app.use(bodyParser.json());
```

```
//Getting the db credentials
const dB = mysql.createConnection({
host: 'localhost',
username:'reactuser',
password: 'reactuser10',
database: 'formapp'
});
dB.connect((err, conn)=>{
        if(err) throw err;
        console.log("Db Connected");
});
app.post('/api/submit',(req, res)=>{
        const {fname, lname, pnum. gen}= req.body;
        console.log(req.body);
const sql = 'Insert into candidate(fname, lname, pnum, gen) values(?,?,?,?)';
dB.query(sql,[fname, lname, pnum, gen],(err, result)=>{
        if(err) throw err;
        result.send("User Added to the DB");
}) //Query Ends here
})//app.post ends here


app.listen(PORT, ()=>{
        console.log(`Server Running at ${PORT}` )
})
```

12. Enter to frontend -> src folder

13. mkdir component -> cd component -> ApplicationForm.js

14. npx install axios

```
frontend/src/component/ApplicationForm.js
import React, {useState} from 'react';
import axios from 'axios';
function ApplicationForm(){
        const [fname, setFname] = useState('');
        const [lname, setLname] = useState('');
        const [pnum, setPnum] = useState('');
        const [gen, setGen] = useState('');
const handleSubmit = async (e)=>{
                e.preventDefault();
        try{
        await   axios.post('http://localhost:3000/api/submit',{fname, lname, pnum,
gen});
        alert("Uploaded Sucessfully");
        setFname('');
        setLname('');
        setPnum('');
        setGen('');
        }
        catch(err){
                console.error(err);
                alert("failed");
        }
}
```

```
return (
<form onSubmit={handleSubmit}>
<div>
<label> Fname: </label>
<input type="text" value= {fname} onChange= {(e)=>setFname(e.target.value)} />
</div>
<div>
<label> Lname: </label>
<input type="text" value= {lname} onChange= {(e)=>setLname(e.target.value)} />
</div>
<div>
<label> Phone Number: </label>
<input type="number" value= {pnum} onChange= {(e)=>setPnum(e.target.value)} />
</div>
<div>
<label> Gender: </label>
<select value={gen} onChange= {(e)=>setGen(e.target.value)}
<option value="">Select Gender </option>
<option value="Male">Male</option>
<option value="Female">Female </option>
<option value="Others">Others</option>
</select>
</div>
<button type="submit"> Submit</button>
</form>
)}//Function ends here
)
```

MySQL Workbench

```
create database formapp;
use formapp;

create table candidate(
id int auto_increment primary key,
fname varchar(255) not null,
lname varchar(255) not null,
pnum int not null,
gen varchar(10) not null
);

create user 'reactuser'@'localhost' identified by 'reactuser10';
grant all privileges on formapp.* to 'reactuser'@'localhost';
flush privileges;
alter user 'reactuser'@'localhost' identified with mysql_native_password by 'reactuser10';

select * from candidate;
```