MODERN WEB APPLICATIONS

UNIT-1

Introduction: MERN, MERN Components, Serverless Hello World Application, ES6, DOM, JSON, Installation.
React Basics: Introduction, Virtual DOM, Components in React, Tradeoffs, using JSX , React Project Structure, State, Component Communication, Oneway data flow, Rendering and Life Cycle method.

Q) Briefly explain MERN Components.

Any web application is made by using multiple technologies. The combination of these technologies is called a **"stack".**

MERN is an acronym that stands for MongoDB, ExpressJS, ReactJS, and NodeJS. These four technologies are used together to create web applications that are fast, efficient, and scalable.

MERN Components:

1. React

React anchors the MERN stack. React is an open-source JavaScript library maintained by Facebook that can be used for creating views rendered in HTML. You use React to render a view (the V in MVC).

1.1 Declarative

React views are declarative. A React component declares how the view looks like, given the data. When the data changes, the React library figures out how the new view looks, and just applies the changes between the old view and the new view. This makes the views consistent, predictable, easier to maintain, and simpler to understand.

1.2 Component-Based

The fundamental building block of React is a component, which maintains its own state and renders itself.

In React, all you do is build components. Then, you put components together to make another component that depicts a complete view or page.

A component encapsulates the state of data and the view, or how it is rendered. This makes writing and reasoning about the entire application easier, by splitting it into components and focusing on one thing at a time.

Components talk to each other by sharing state information in the form of read-only properties to their child components and by callbacks to their parent components.

1.3 No Templates

Many web application frameworks rely on templates to automate the task of creating repetitive HTML or DOM elements.

In React there is an intermediate language to represent a virtual DOM, and that is JSX, which is very similar to HTML. You can write pure JavaScript to create your virtual DOM if you prefer.

1.4 Isomorphic

React can be run on the server too. That's what isomorphic means: the same code can run on both server and the browser.

1.5 VDOM

React takes care of screen refreshing in case of updates in the page using its virtual DOM technology, in the form of a virtual representation, an in-memory data structure. React can compute the differences in the virtual DOM very efficiently, and can apply only these changes to the actual DOM.

2. Node.js

Node.js is JavaScript outside of a browser. The Node.js runtime runs JavaScript programs.

2.1 Node.js Modules

For Node.js, there is no HTML page that starts it all. In the absence of the enclosing HTML page, Node.js uses its own module system based on CommonJS to put together multiple JavaScript files.

Modules are like libraries. You can include the functionality of another JavaScript file by using the keyword require.

Node.js ships with a bunch of core modules compiled into the binary. These modules provide access to the operating system elements such as the file system, networking, input/output, etc. They also provide some utility functions that are commonly required by most programs.

2.2 Node.js and npm

npm is the default package manager for Node.js. You can use npm to install third-party libraries (packages) and also manage dependencies between them. The npm registry (www.npmjs.com) is a public repository of all modules published by people for the purpose of sharing. In fact, even though React is largely client-side code and can be included directly in your HTML as a script file, it is recommended instead that React is installed via npm. But, once installed as a package, we need something to put all the code together that can be included in the HTML so that the browser can get access to the code. For this, there are build tools such as browserify or webpack that can put together your own modules as well as third-party libraries in a bundle that can be included in the HTML.

2.3 Node.js Is Event Driven

Node.js has an asynchronous, event-driven, non-blocking input/output (I/O) model, as opposed to using threads to achieve multitasking.

It relies on callbacks to let you know that a pending task is completed. So, if you write a line of code to open a file, you supply it with a callback function to receive the results. On the next line, you continue to do other things that don't require the file handle. Event-driven programming is natural to Node.js due to the underlying language constructs such as closures.

Node.js achieves multitasking using an event loop. This is nothing but a queue of events that need to be processed and callbacks to be run on those events. In the above example, the file that is ready to be read is an event that will trigger the callback you supplied while opening it.

3. Express

Node.js is just a runtime environment that can run JavaScript. Express is the framework that simplifies the task of writing your server code.
Express is a web server framework meant for Node.js The Express framework lets you define routes, specifications of what to do when a HTTP request matching a certain pattern arrives.
Express parses request URL, headers, and parameters for you. On the response side, it has, as expected, all of the functionality required by web applications. This includes setting response codes, setting cookies, sending custom headers, etc.

4. MongoDB

4.1 NoSQL

NoSQL stands for "non-relational". NoSQL databases are not necessarily relational databases.

4.2 Document-Oriented

Compared to relational databases where data is stored in the form of relations, or tables, MongoDB is a document-oriented database. The unit of storage (comparable to a row) is a document, or an object, and multiple documents are stored in collections (comparable to a table). Every document in a collection has a unique identifier by which it can be accessed. The identifier is indexed automatically.

The downside is that the data is stored denormalized. This means that data is sometimes duplicated, requiring more storage space.

4.3 Schema-Less

Storing an object in a MongoDB database does not have to follow a prescribed schema. All documents in a collection need not have the same set of fields.

4.4 JavaScript Based

For relational databases, there is a query language called SQL. For MongoDB, the query language is based on JSON: you create, search for, make changes, and delete documents by specifying the operation in a JSON object.

Q) What is ES6? Explain the following each with an example.

a. Arrow Function
b. Rest & Spread Operators
c. Map Object
d. Set Object
e. Classes
f. Promises
g. Symbol
h. Modules
i. var,let,const
j. Destructuring

a. Arrow function:

Arrow functions allows a short syntax for writing function expressions. You don't need the function keyword, the return keyword, and the **curly brackets**.

Eg.

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

```
//ES6
const x = (x, y) => x * y;
```

arrow_function.js

```
const mult = (x, y) => { return x * y };
console.log(mult(2,5))
```

Output:

```
C:\Program Files\nodejs\node.exe .\arrow_function.js
10                                                    arrow_function.js:2
```

b. Rest Operator:

The rest operator is represented by three dots (...), and it is used to represent an indefinite number of arguments as an array. The rest operator is commonly used in function definitions, but it can also be used in other contexts like array destructuring.

Eg.

rest_op.js

```
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);
console.log("sum of numbers is ",x)
x = sum(4, 9, 16, 25, 29);
console.log("sum of numbers is ",x)
```

Output:

```
sum of numbers is  326
sum of numbers is  83
```

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

Eg.

spread_op.js

```
//Copying an array
let fruits = ['Apple','Orange','Banana'];
let newFruitArray = [...fruits];
console.log(newFruitArray);

//Concatenating arrays
let arr1 = ['A', 'B', 'C'];
let arr2 = ['X', 'Y', 'Z'];
let result = [...arr1, ...arr2];
console.log(result);
```

Output:

```
(3) ['Apple', 'Orange', 'Banana']
(6) ['A', 'B', 'C', 'X', 'Y', 'Z']
```

c. **Map Object:**
   A Map holds key-value pairs where the keys can be any datatype.
   A Map remembers the original insertion order of the keys.
   Map has a property that represents the size of the map.

Map Methods

Method	Description
new Map()	Creates a new Map object
set()	Sets the value for a key in a Map
get()	Gets the value for a key in a Map
clear()	Removes all the elements from a Map
delete()	Removes a Map element specified by a key

has()	Returns true if a key exists in a Map
forEach()	Invokes a callback for each key/value pair in a Map
entries()	Returns an iterator object with the [key, value] pairs in a Map
keys()	Returns an iterator object with the keys in a Map
values()	Returns an iterator object of the values in a Map

Property	Description
size	Returns the number of Map elements

Eg.

```
const map1 = new Map();

map1.set('a', 1);
map1.set('b', 2);
map1.set('c', 3);
console.log(map1.get('a')); // Expected output: 1

map1.set('a', 97);
console.log(map1.get('a')); // Expected output: 97

console.log(map1.size); // Expected output: 3

map1.delete('b');
console.log(map1.size); // Expected output: 2

console.log(map1.has('a')); // true
let iterator1 = map1.keys();
console.log(iterator1.next().value); //displays first key

iterator1 = map1.values();
console.log(iterator1.next().value);//displays first value
```

Output:

```
C:\Program Files\nodejs\node.exe .\map_methods.js
1
97
3
2
true
a
97
```

d. Set Object:

A JavaScript Set is a collection of unique values.
Each value can only occur once in a Set.
A Set can hold any value of any data type.

Set Methods

Method	Description
new Set()	Creates a new Set
add()	Adds a new element to the Set
delete()	Removes an element from a Set
has()	Returns true if a value exists
clear()	Removes all elements from a Set
forEach()	Invokes a callback for each element
values()	Returns an Iterator with all the values in a Set
keys()	Same as values()
entries()	Returns an Iterator with the [value,value] pairs from a Set
Property	**Description**
size	Returns the number elements in a Set

Eg.

```
const mySet1 = new Set()

mySet1.add(1)          // Set(1) { 1 }
mySet1.add(5)          // Set(2) { 1, 5 }
mySet1.add(5)          // Set(2) { 1, 5 }
mySet1.add('some text') // Set(3) { 1, 5, 'some text' }

const o = {a: 1, b: 2}
mySet1.add(o)
mySet1.add({a: 1, b: 2})   // o is referencing a different object, so this is okay

mySet1.has(1)              // true
mySet1.has(3)              // false, since 3 has not been added to the set
mySet1.has(5)              // true
mySet1.has(Math.sqrt(25))  // true
mySet1.has('Some Text'.toLowerCase()) // true
mySet1.has(o)       // true

mySet1.size         // 5

mySet1.delete(5)    // removes 5 from the set
mySet1.has(5)       // false, 5 has been removed

mySet1.size         // 4, since we just removed one value
```

```
mySet1.add(5)       // Set(5) { 1, 'some text', {...}, {...}, 5 } - a previously deleted item will be added as a new item, it will not retain its original position before deletion

console.log(mySet1) // Set(5) { 1, "some text", {...}, {...}, 5 }
```

e. Classes:

The ES6 JavaScript supports the Object-Oriented programming components. Classes allow developers to create blueprints for objects with specific properties and methods.
Eg.

Class_demo.js

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
 area(){
    let a = this.height * this.width
    return a;
  }
}
const r = new Rectangle(2,5);
console.log(r.area());
```

Output:
10

f. Promise:

Promise help to manage asynchronous operations in JavaScript. A promise represents a value that may not be available yet, but will be resolved at some point in the future.

A promise can be in one of three states:
Pending: The initial state, neither fulfilled nor rejected.
Fulfilled: Meaning that the operation completed successfully and the promise has a resulting value.
Rejected: Meaning that the operation failed and the promise has a reason for the failure.

Once a Promise is fulfilled or rejected, it will be immutable. The **Promise()** constructor takes two arguments that are **rejected** function and a **resolve** function. Based on the asynchronous operation, it returns either the first argument or second argument.

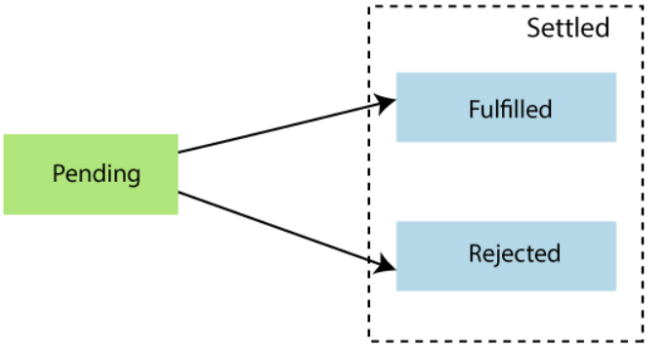


Fig. Promise States

Eg.

Promise1.js

```
 let myPromise = new Promise(function(myResolve, myReject) {
  let x = 1;
  if (x == 0) {
    myResolve("OK");
  } else {
    myReject("Error");
  }
});

myPromise.then(
  function(value) {console.log(value);},
  function(error) {console.log(error);}
);
```

Output:

```
Error
```

Promise2.js

```
//all()
const promise1 = Promise.resolve(3);
const promise2 = Promise.resolve(43);
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'yes');
  setTimeout(reject, 1000, 'no');
});

Promise.all([promise3, promise2, promise1]).then((values) => {
  console.log(values);
});
```

Output:

```
(3) ['yes', 43, 3]
```

g. Symbol:

Symbols are a new primitive data type in JavaScript that represent a unique identifier. They are created using the Symbol() function.
Symbols provide a way to create unique identifiers in JavaScript that can be used for a variety of purposes.
Symbols can also be used to create "private" properties or methods in objects, since they cannot be accessed using the object's keys.
Eg.
symbol_eg.js

```
const sym1 = Symbol();
const sym2 = Symbol("chp");
const sym3 = Symbol("chp");
console.log(sym3.valueOf())
console.log(Symbol("chp") === Symbol("chp"))

console.log(Symbol.for("praneeth") === Symbol.for("praneeth"));

const globalSym = Symbol.for('chp'); // Global symbol
console.log(Symbol.keyFor(globalSym));
```

Output:

```
C:\Program Files\nodejs\node.exe .\symbol_eg.js
Symbol(chp)
false
true
chp
```

symbol_obj.js

```
let person = {
    name: "chp",
    age: 20
};
let id = Symbol("id");
person[id] = 1;

for(let key in person) {
    console.log(key);
}
console.log(person);
```

Output:

```
C:\Program Files\nodejs\node.exe .\symbol_obj.js
name                                                    symbol_obj.js:9
age                                                     symbol_obj.js:9
{name: 'chp', age: 20, Symbol(id): 1}
```

In above program, the loop iterates over the name and age properties, but it does not iterate over the id property because symbols are not enumerable by default.

h. Modules:

In ES6, modules are a way to organize and encapsulate code into reusable, independent components.

Modules are defined using the export and import statements. The export keyword is used to make functions, objects, or values available from a module and to be used in other parts of the program. The import keyword is used to load the exported functionality from other modules into the current module.

In package.json we have to add the property "type": "module" to use modules.

Eg.

module.js

```
export let data = "praneeth";
export const add = (a,b) =>{
   return a+b;
};

export const mult = (a, b) => {
   return a * b;
 };
```

module_demo.js

```
import {data, add, mult} from
'./module.js';

console.log("Welcome ",data);
console.log(add(3, 4));
console.log(mult(3, 4));
```

Output:

```
Welcome  praneeth
7
12
```

i. **var,let,const**

var1.js

```
var a =10;
console.log(a)
{

  var a = 20
 console.log(a)
}
console.log(a)
```

Output:

10
20
20

the last display statement also be 20, because the inner declaration of a persists outside the block scope and has overwritten the original value of a declared at the beginning of the program since a is declared as var but not let.

let1.js

```
let a =10;
console.log(a)
{
  let a = 20
 console.log(a)
}
console.log(a)
```

Output:

10
20
10

This program declares two variables with the same name "a" but using different scopes due to the use of the let keyword.

Const is similar to let but can't modify the value of a variable.

const1.js

```
const a =10;
console.log(a)
{
  const a = 20
  console.log(a)
}
console.log(a)
```

Output:

10
20
10

const2.js

```
const a =10;
console.log(a)
{
  const a = 20
   const a =15
 console.log(a)
}
console.log(a)
```

Output:

```
SyntaxError: Identifier 'a' has
already been declared
```

j. **Destructuring:**

Destructuring is a feature that allows you to extract values from arrays or objects and assign them to variables in a more concise and readable way.

There are two types of destructuring: array destructuring and object destructuring.

Array Destructuring:

With array destructuring, you can assign array elements to variables by enclosing them in square brackets [].

Object Destructuring:

With object destructuring, you can assign object properties to variables by enclosing them in curly braces {}.

Eg.

```
//Array Destructuring
let fruits = ["Apple", "Banana"];
let [a, b] = fruits; // Array destructuring assignment
console.log(a, b);

//Object Destructuring
let person = {name: "Praneeth", age: 30};
let {name, age} = person; // Object destructuring assignment
console.log(name, age);
```

Output:

```
C:\Program Files\nodejs\node.exe .\destructuring_assignment.js
Apple Banana                          destructuring_assignment.js:4
Praneeth 30                           destructuring_assignment.js:9
```

Q) What is JSON? Explain how to convert JSON object to JS object and vice-versa.

JSON stands for JavaScript Object Notation, which is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is based on a subset of the JavaScript programming language and is often used to transmit data between a server and web application, as an alternative to XML.

Eg. JSON Data format:

```
{
  "name": "Ch Praneeth",
  "age": 30,
  "city": "Vijayawada",
  "email": "praneeth@ex.com",
  "phone": [
    {
      "type": "home",
      "number": "1234"
```

```
    },
    {
      "type": "work",
      "number": "5678"
    }
  ]
}
```

Eg.

json_eg.js

```
//json data to js object
let text = '{ "students" : ['+
'{ "firstName":"ch" , "lastName":"praneeth" },' +
'{ "firstName":"v" , "lastName":"rajesh" },' +
'{ "firstName":"P" , "lastName":"nageswarao" } ]}';
console.log("Before ",typeof text)
const obj = JSON.parse(text);
console.log("After ",typeof obj)
console.log(obj.students[1].lastName)


//js object to json data
const obj1 = {
   id: 2,
   customer: "ch praneeth",
   city: "vijayawada"
 };

 console.log(JSON.stringify(obj1));
```

Output:

```
C:\Program Files\nodejs\node.exe .\json_eg.js
Before  string                                          json_eg.js:6
After  object                                           json_eg.js:8
rajesh                                                  json_eg.js:9
{"id":2,"customer":"ch praneeth","city":"vijayawad …on_eg.js:18
a"}
```

Q) Explain about DOM.

The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. Essentially, it creates a tree-like structure of all the elements on a web page, making it possible to manipulate those elements using code.

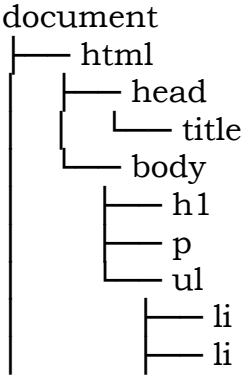
Eg.:

index.html:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Subjects</title>
    <script type = "text/javascript" src="index.js"></script>
  </head>
  <body>

    <h1>Welcome to </h1>
    <p>Modern Web Applications.</p>
    <ul>
      <li>React</li>
      <li>Express</li>
    </ul>
    <input type = "button" onclick = "change()" value = "Display">
  </body>
</html>
```

This code creates a basic web page with a title, heading, paragraph, and a list. When the web page is loaded, the browser creates a DOM tree that represents the page. The tree starts with the "document" object, which represents the entire page, and has child nodes that represent the different elements of the page.

DOM tree for above HTML code:



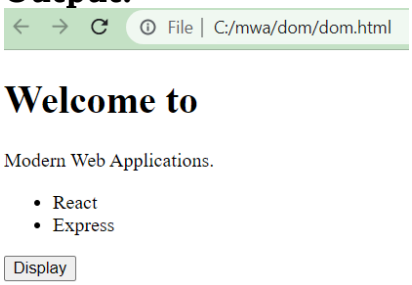
With the DOM, you can use JavaScript to access and manipulate any element on the page. For example, if you wanted to change the text of the heading to "Hello, world!", you could do it like this:

index.js

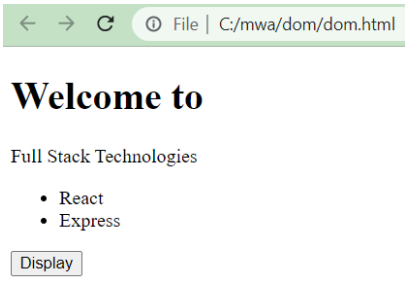
```
// Get the h1 element
var heading = document.getElementsByTagName("p")[0];
// Change the text of the heading
heading.textContent = "Full Stack Technologies";
```

This code uses the getElementsByTagName method to find the first h1 element on the page and then sets its textContent property to "Hello, world!". This would change the text of the heading on the page to "Hello, world!".

Output:

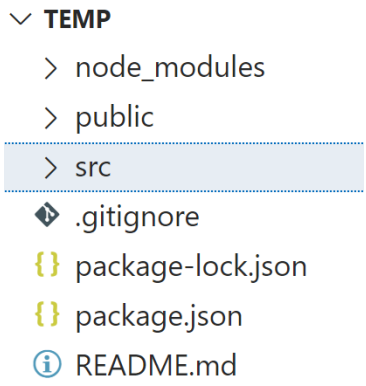


(a) Before Clicking on Button



(b) After Clicking on Button

Q) Explain React Project Structure.



Node_modules: This folder contains all the installed dependencies for the application.
public/: This folder contains the static assets and HTML template for the application.
index.html: The main HTML file that serves as the entry point for the application.
src/: This folder contains the source code of the application.
assets/: This folder contains all the static assets used in the application, such as images, fonts, and styles.
images/: This folder contains all the image files used in the application.
styles/: This folder contains all the CSS files used in the application.
components/: This folder contains all the reusable UI components of the application.
routes/: This folder consists of all routes of the application. It consists of private, protected, and all types of routes. Here we can even call our sub-route.

services/: This folder contains all the services used in the application, such as APIs, authentication, and analytics.
utils/: This folder contains all the utility functions used in the application.
helpers.js: A helper function implementation.
App.js: The root component of the application.
index.js: The entry point of the application that renders the root component.
package.json: The configuration file for the project that includes all the dependencies, scripts, and metadata.
README.md: The documentation file for the project.

Q) Explain how Virtual DOM works in React.

The Virtual DOM is a programming concept that is commonly used in web development like React. It is an abstraction of the real DOM (Document Object Model) and is a lightweight copy of the actual DOM, which is created and maintained in memory.

The Virtual DOM is used to increase the performance and efficiency of web applications. The idea behind it is that it is much faster to manipulate the Virtual DOM than the actual DOM.

When a change happens, React determines differences between the actual and in-memory DOMs. Then it performs an efficient update to the browser's DOM. This process is often referred to as a diff ("what changed?") and patch ("update only what changed") process as shown in below fig.

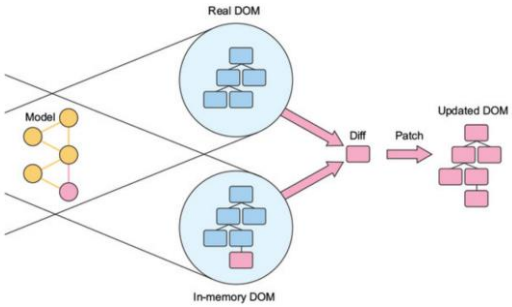


Fig. React's diffing and update procedure

Eg.

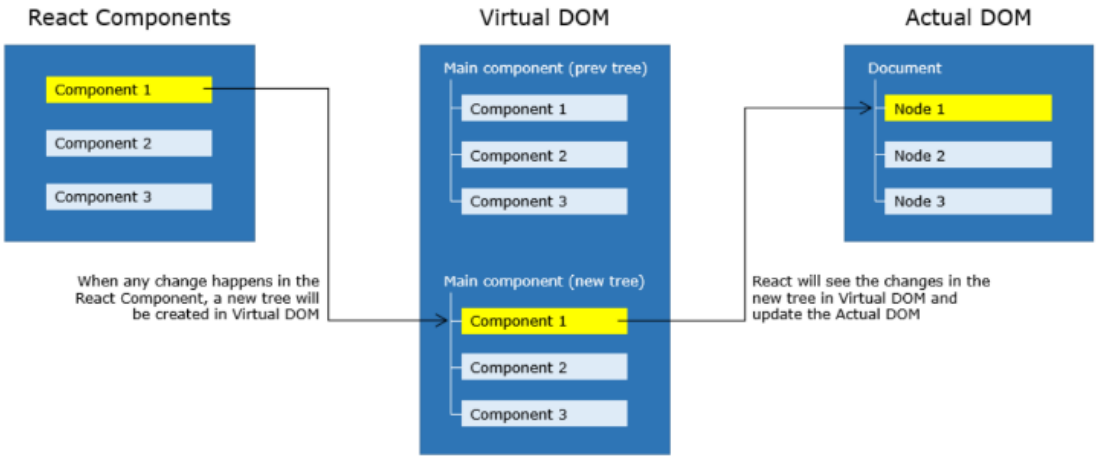


Fig. Working of Virtual DOM

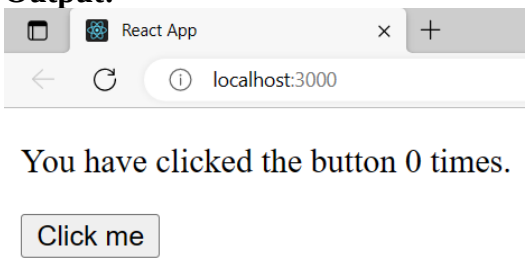
- Whenever any updates happens in the application, the virtual DOM gets modified. React computes the difference between the previous virtual tree and the current virtual tree.
- Based on these differences React will figure out how to make updates to the actual DOM efficiently.
- React does all the computations in its abstracted DOM and updates the DOM tree accordingly.
- Virtual DOM enhances performance and efficiency by minimizing expensive updates in the actual DOM
- Hence React is said to be a great performer because it manages a Virtual DOM.

Eg.
index.js

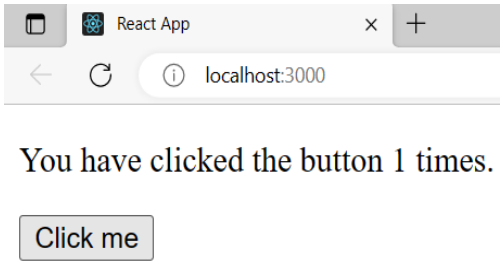
```
import React from 'react';
import ReactDOM from 'react-dom';

class ExampleComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  handleClick() {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>You have clicked the button {this.state.count} times.</p>
        <button onClick={() => this.handleClick()}>Click me</button>
      </div>
    );
  }
}

ReactDOM.render(
  <ExampleComponent />,
  document.getElementById('root')
);
```

In this example, we have a React component called ExampleComponent that maintains a count state. When the user clicks the button, the handleClick method is called, which updates the state with a new count. The render method returns a virtual DOM tree that reflects the new count, and React updates the actual DOM with the minimum set of changes necessary to achieve the desired result.

Output:



(a) Before Clicking the Button

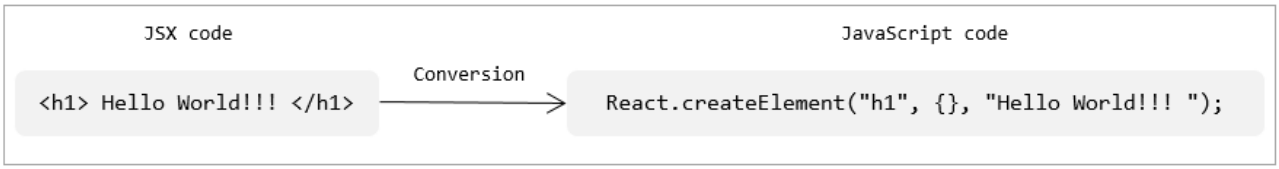


(b) After Clicking the Button

Q) Explain about JSX.

JSX is a special syntax introduced in ReactJS to write elements of components. It is syntactically identical to HTML and hence it can be easily read and written. Code written using JSX helps in visualizing the DOM structure easily.

As the browser does not understand JSX code, this gets converted to JavaScript using the plugins of the babel.
Conversion of JSX to JavaScript happens as shown below:



Advantages:

1. Improved readability: JSX can make code easier to read and understand, especially for developers who are familiar with HTML. It allows developers to write UI code in a more declarative and intuitive way.
2. Enhanced functionality: JSX enables developers to use the full power of JavaScript within their markup, such as adding conditional statements, loops, and functions directly into the markup.
3. Better performance: JSX can improve performance by reducing the number of function calls needed to update the DOM. With JSX, React can efficiently update only the parts of the DOM that have changed instead of re-rendering the entire page.
4. Code reuse: JSX makes it easy to create reusable components with a clean and concise syntax. This can lead to more efficient development and less code repetition.

HTML vs JSX:

1 Attributes and properties: In HTML, attributes are case-insensitive, while in JSX, attributes use camelCase syntax. For example, in HTML, we would use class="my-class", whereas in JSX, we would use className="my-class".

2 Self-closing tags: In HTML, some tags are self-closing, such as <img> and
. In JSX, all tags must be closed, even if they do not have any content, like .

3 Inline styles: In HTML, we use the style attribute to add styles to an element, and the value is a string containing CSS properties and values separated by a semicolon. In JSX, we use the style attribute as an object, where the keys are the CSS properties in camelCase syntax, and the values are their corresponding values in quotes.

4 Comments: In HTML, we use <!-- --> to add comments, while in JSX, we use {/* */}.

Q) What are React Fragments? Explain with an example.

React Fragments
By adhering to JSX syntax the tag can be used for grouping the elements and this introduces an extra and unnecessary node into the DOM. As a solution to this, React Fragments are introduced which is a common pattern in React for a component to return multiple elements. React Fragments allows to group a list of React elements without adding any extra node to the DOM.

```
function App() {
  return (
    <React.Fragment>
      <h3>ReactJS:</h3>
      <p> React is a JavaScript library for creating User Interfaces.</p>
    </React.Fragment>
  );
}
export default App;
```

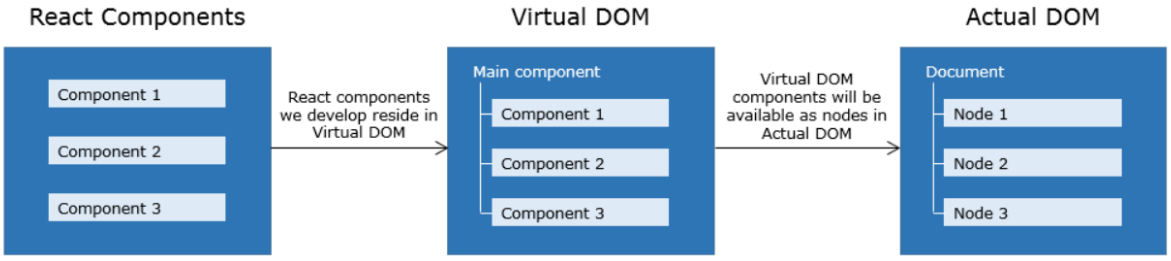
Empty Tags

You can use empty tags instead of React.Fragment

```
function App() {
  return (
    <>
      <h3>ReactJS:</h3>

      <p> React is a JavaScript library for creating User Interfaces.</p>
    </>
  );
}
```

Q) Explain about components in React.

A React component is the fundamental unit of any React application. They are capable of encapsulating data and view as a single unit. These components can work in conjunction with each other.
Components make code reusable, testing easy, and can take care of separation of concerns.
Here, in ReactJS, components reside in virtual DOM and these components will be available as nodes in actual DOM as shown below:



In React.js, there are two types of components: function components and class components.

1. **Function Components**

A function component is a JavaScript function that takes in an object of properties (props) as its argument and returns a React element. Function components are the simplest way to create components in React.

Eg.

Welcome.js

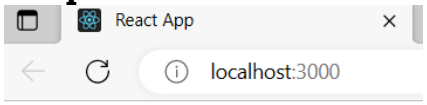
```
import React from 'react';

function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}
export default Welcome;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import Welcome from './Welcome.js';

ReactDOM.render(
  <Welcome name = "chp" />,
  document.getElementById('root')
);
```

This function component takes in a props object as a parameter and returns a JSX element that displays a greeting message with the name passed in as a prop.

Output:



Hello, chp!

2. Class Components

A class component is a JavaScript class that extends the React.Component class. Class components have more features than function components, such as state and lifecycle methods.

Eg.

Welcome.js

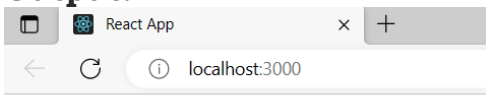
```
import React from 'react';

class Welcome extends
React.Component {
  render() {
    return <h1>Hello,
{this.props.name}!</h1>;
  }
}
export default Welcome;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-
dom';
import Welcome from
'./Welcome.js';

ReactDOM.render(
   <Welcome name = "praneeth" />,
  document.getElementById('root')
);
```

This class component defines a render() method that returns a JSX element that displays a greeting message with the name passed in as a prop. The this.props object is used to access the props passed to the component.

Output:



Hello, praneeth !

Q) What are state and props? Explain with suitable examples.

In a real-time application, components must deal with dynamic data. This data could be something internal to the component or may be the data that is passed from another component. To bind the data to the component, you need two JS objects i.e., state and props.

State:

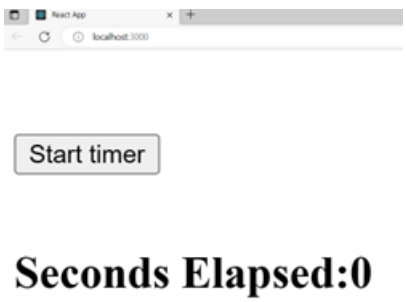
- States are mutable
- They are reserved for interactivity. The component's event handlers may update the state and trigger a UI update
- The state will be set with a default value when component mounts and will mutate in time based on user events generated

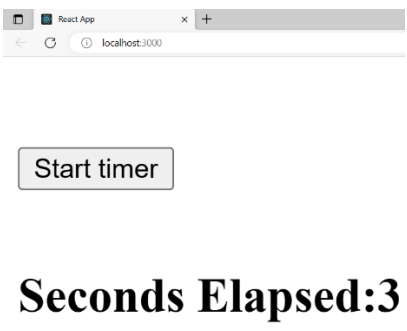
Eg.

index.js

```
import {createRoot} from 'react-dom/client';
import React from 'react';

class Timer extends React.Component {
  constructor() {
      super();
      this.state = {
          secondsElapsed: 0
      };
  }
  start = () => {
      this.setState({
          secondsElapsed: this.state.secondsElapsed + 1
      });
  }
  handleClick = (e) => {
      this.interval = setInterval(this.start, 1000);
  }
  render() {
      return ( <React.Fragment><br/><br/>
          <button onClick = {this.handleClick}>Start timer</button><br/><br/>
          <h2> Seconds Elapsed:
              {this.state.secondsElapsed}
          </h2>
          </React.Fragment>);
      }
  }

const root = createRoot(document.getElementById('root'));
root.render(<Timer />);
```

Output:





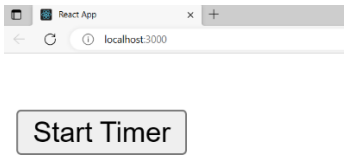
Props:

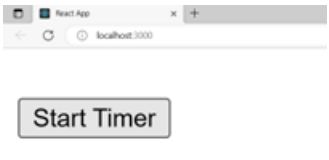
- Props are immutable
- The child component cannot modify the props. However, when parent component updates the data that is passed as props then the Child component gets updated props.
- Whenever props are updated the component gets re-rendered and displays the latest value in the UI.

Index.js

```
import {createRoot} from 'react-dom/client';
import React from 'react';
class Timer extends React.Component {
  constructor() {
  super();
  this.state = {
          secondsElapsed: 0
      };
  }
  start = () => {
      this.setState({
          secondsElapsed: this.state.secondsElapsed + 1
      });
  }
  handleClick = (e) => {
      this.interval = setInterval(this.start, 1000);
  }
  render() {
      return ( <React.Fragment><br/>
                  <button onClick = {this.handleClick}>
                      Start Timer
                  </button><br/><br/>
                  <Resultant new = {this.state.secondsElapsed}/>
              </React.Fragment>);
          }
}
class Resultant extends React.Component {
  render() {
          return ( <div>
                      <h3> Seconds Elapsed: {this.props.new} </h3>
                  </div>);
              }
}
const root = createRoot(document.getElementById('root'));
root.render(<Timer />);
```

Output:





State Vs Props.

Characteristic	State	Props
Mutable	Yes	No
Purpose	Used for mutable data that affects UI	Used for immutable data that is passed to a component
Set by	Defined and updated within a component.	Passed down from parent component as an attribute or property
Changes	Changes trigger component re-rendering	Changes don't trigger re-rendering unless they come from a parent

Q) Define Hook. Make a function component as a stateful component using hooks.

A "hook" is a special function that allows you to add stateful logic to function components. Hooks were introduced in React 16.8 and are used to manage state and lifecycle methods in function components.

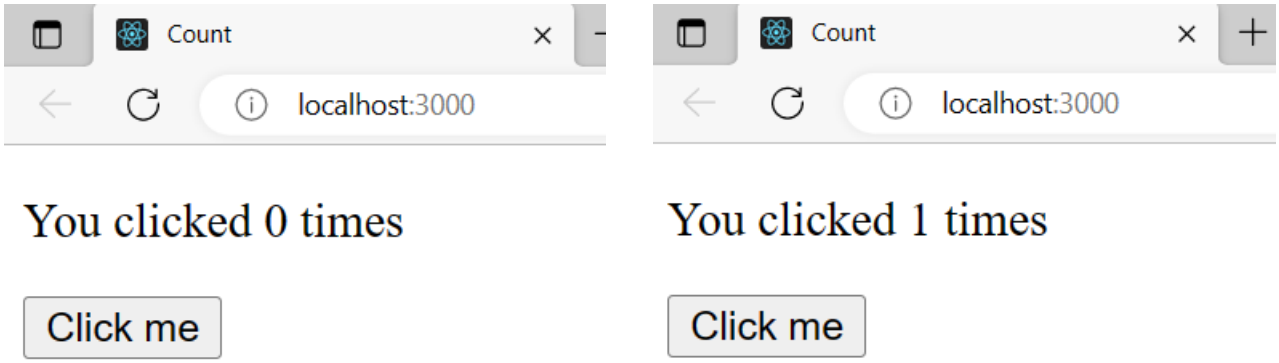
index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = 'Count';
  });
```

```
    return (
        <div>
          <p>You clicked {count} times</p>
          <button onClick={() => setCount(count + 1)}>
            Click me
          </button>
        </div>
      );

}

ReactDOM.render(
    <Example />,
    document.getElementById('root')
  );
```

The useState hook is used to define a state variable called count which we initialize with a value of 0.
The setCount function returned by the useState hook is used to update the value of count.
The useEffect hook is used to define a side effect that will run every time the component renders.

Output:



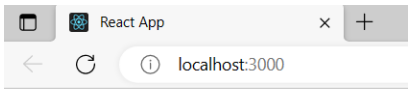
Q) Explain how component communication takes place in React (or) Explain composite component in React (or) Explain how to establish relation between components (Parent-Child).

Components can receive data from their parent components via props, which are essentially properties that are passed down from the parent component.

index.js

```
import React from "react";
import ReactDOM from 'react-dom';

function ParentComponent() {
  const name = "Praneeth";
  return (
    <div>
      <h1> Hello </h1>
      <ChildComponent name={name} />
    </div>

  );
}

function ChildComponent(props) {
  return <h2>Welcome, {props.name}!</h2>;
}

ReactDOM.render(
  <ParentComponent />,
  document.getElementById('root')
  );
```

Output:



Hello

Welcome, Praneeth!

Q) Explain data flow in React.

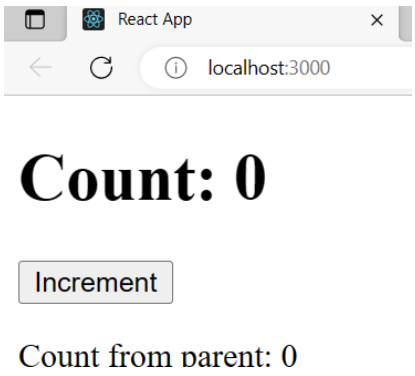
In react data flow is one-way from parent component to child component. The parent component passes data as props to its child component, and the child component can only read the data but cannot modify it. If the child component wants to change the data, it needs to send a request to the parent component, and the parent component will decide whether to modify the data and pass the updated data as props to the child component.

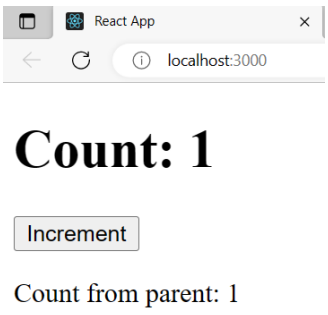
Eg.

index.js

```
import {React, useState} from "react";
import ReactDOM from 'react-dom/client';
function Parent() {
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <h1>Count: {count}</h1>
      <Child count={count} onIncrement={incrementCount} />
    </div>
  );
}

function Child(props) {
  return (
    <div>
      <button onClick={props.onIncrement}>Increment</button>
      <p>Count from parent: {props.count}</p>
    </div>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
      <Parent />
);
```

Output:





Here, the Child component can display the count value, but it cannot modify it directly. Instead, it calls the onIncrement() that is passed as a prop to the

Parent component when the user clicks the "Increment" button. The Parent component updates the count value using the setCount() and passes the updated value to the Child component as props.

Q) Explain about Mutable and Immutable states in React.

Mutable state:

In React, state is a data structure that holds information about the component that can change over time. When we talk about mutable and immutable states in React class components, we're referring to how the state is updated.

Mutable state refers to state that can be modified directly.

Eg. Mutable state:

```
state = {
  count: 0
};
```

We can modify the count value directly using the setState method like this:

```
this.setState({ count: this.state.count + 1 });
```

This changes the value of count in place, meaning that the original state object is modified.

On the other hand, immutable state refers to state that cannot be modified directly. Instead, we create a new object that represents the updated state. We do this by making a copy of the original state object, modifying the copy, and then setting the state to the new object.

Eg. immutable state:

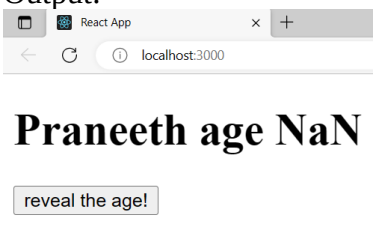
```
state = {
  person: {
    name: "Praneeth",
    age: 30
  }
};

handleClick = () => {
  const updatedPerson = {
    ...this.state.person,
    age: this.state.person.age + 1
  };

  this.setState({ person: updatedPerson });
};
```

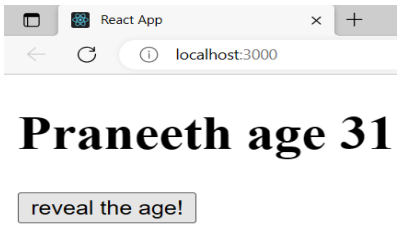
a) Write a program in React.js to illustrate Mutable state

Index.js

```
import React from "react";
import ReactDOM from 'react-dom';
class Secret extends React.Component{
constructor(props) {
super(props);
this.state = {
name: 'Praneeth',age:NaN
};
this.onButtonClick = this.onButtonClick.bind(this);
}
onButtonClick() {
this.setState(() => ({
age: 31
}));
}
render() {
return (
<div>
<h1>{this.state.name} age {this.state.age}</h1>
<button onClick={this.onButtonClick}>reveal the age!</button>
</div>
)
}
}

ReactDOM.render(
    <Secret />,
    document.getElementById('root')
    );
```

Output:





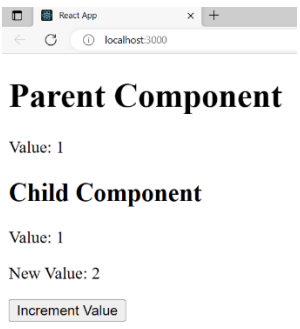
b) write a program in React.js to illustrate Immutable state using Props.

index.js

```
import React from "react";
import ReactDOM from 'react-dom';
```

```
function ChildComponent(props) {
  const newProps = {
    ...props,
    value: props.value + 1
  };

  return (
    <div>
      <h2>Child Component</h2>
      <p>Value: {props.value}</p>
      <p>New Value: {newProps.value}</p>
    </div>
  );
}


function ParentComponent() {
  const [value, setValue] = React.useState(0);

  return (
    <div>
      <h1>Parent Component</h1>
      <p>Value: {value}</p>
      <ChildComponent value={value} />
      <button onClick={() => setValue(value + 1)}>Increment Value</button>
    </div>
  );
}

ReactDOM.render(
  <ParentComponent />,
  document.getElementById('root')
  );
```

Output:



c) Write a program in React.js to illustrate immutable state.

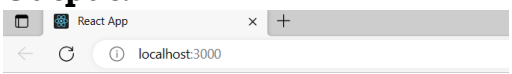
In general, the state in react is mutable. But by creating a new copy of the state and setting the state to new copy can be made state as immutable.

Eg.

index.js

```
import React,{ Component } from "react";
import ReactDOM from 'react-dom';

class Subjects extends Component {
constructor(props) {
    super(props);
    this.state = {
    sub: ["Data Structures", "DAA",
          "Java"],
    };
}
AddSub = () => {
    let newSubs = [...this.state.sub];
    newSubs.push(" DBMS ");
    newSubs.push(" Python ");
    this.setState({ sub: newSubs });
    console.log(" Subjects in the AddSub function ", this.state.sub);
};

render() {
    console.log(" Subjects in the render() ", this.state.sub);

    return (
    <div>
        <h2>Subjects to be covered for Interview:</h2>
        {this.state.sub.map((sub, idx) => (
        <div key={idx}>
            <h4> {sub}</h4>
        </div>
        ))}
        <button onClick={this.AddSub}>Add Subjects</button>
    </div>
    );
}
}

ReactDOM.render(
    <Subjects />,
    document.getElementById('root')
    );
```

Output:



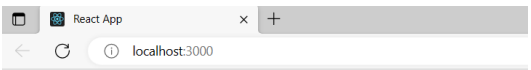
Subjects to be covered for Intervie

Data Structures

DAA

Java

Add Subjects



Subjects to be covered for Interview:

Data Structures

DAA

Java

DBMS

Python

Add Subjects

Q) Explain how to pass a method as props with an example.

FC.js

```
import React from 'react';
import ChildComponent from './SC';

function ParentComponent() {
  const handleClick = () => {
    alert('Button clicked!');
  }

  return (
    <div>
      <h1>Parent Component</h1>
      <ChildComponent handleClick={handleClick} />
    </div>
  );
}
export default ParentComponent;
```

Here, the ParentComponent component defines a function called handleClick. It then renders a ChildComponent component and passes this function to it as a prop called handleClick.

SC.js

```
import React from 'react';

function ChildComponent(props) {
  return (
    <div>
      <h2>Child Component</h2>
      <button onClick={props.handleClick}>Click me!</button>
    </div>
  );
}
export default ChildComponent;
```

In the ChildComponent component, we receive the handleClick function as a prop and use it as the onClick handler for a button.
When the button is clicked, the handleClick function in the ParentComponent component will be called, and the alert message "Button clicked!" will be displayed.

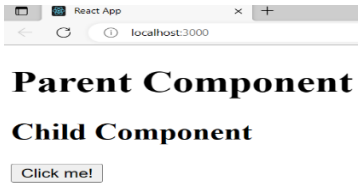
App.js

```
import ParentComponent from
"./Components/FC";

function App() {
  return (
    <div className="App">
      <ParentComponent />
    </div>
  );
}

export default App;
```

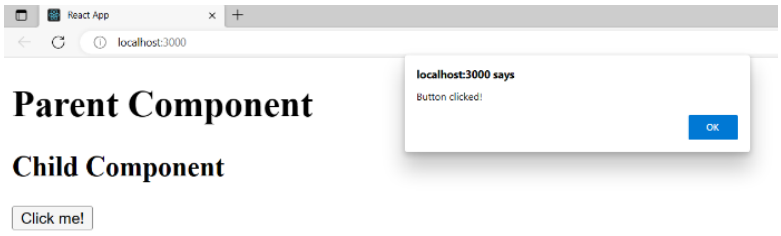
index.js

```
import React from 'react';
import ReactDOM from 'react-
dom/client';

import App from './App';

const root =
ReactDOM.createRoot(document.ge
tElementById('root'));
root.render(

    <App />

);
```

Output:



(a) Before Clicking on Button



(b) After Clicking on Button

Q) Explain how data flows in React

One-way data flow is a fundamental principle in React, which means that data flows in one direction, from the parent component to the child component. In other words, the parent component passes data as props to its child component, and the child component can only read the data but cannot modify it. If the child component wants to change the data, it needs to send a request to the parent component, and the parent component will decide whether to modify the data and pass the updated data as props to the child component.

Eg. index.js

```
import {React, useState} from "react";
import ReactDOM from 'react-dom/client';
```

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

  const incrementCount = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <h1>Count: {count}</h1>
      <Child count={count} onIncrement={incrementCount} />
    </div>
  );
}

function Child(props) {
  return (
    <div>
      <button onClick={props.onIncrement}>Increment</button>
      <p>Count from parent: {props.count}</p>
    </div>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
      <Parent />
);
```

In this example, the Parent component has a state variable count that is initialized to 0. It passes count and the incrementCount function as props to the Child component. The Child component can display the count value, but it cannot modify it directly. Instead, it calls the onIncrement function that is passed as a prop to the Parent component when the user clicks the "Increment" button. The Parent component updates the count value using the setCount function and passes the updated value to the Child component as props. The Child component then re-renders with the updated count value. This is an example of one-way data flow because the data only flows from the Parent component to the Child component, and the Child component cannot modify the data directly.

Q) Explain about defaultProps and PropTypes with suitable examples.

defaultProps is used to set default values for props in case they are not passed explicitly by the component. It is defined as a static property on the component class.

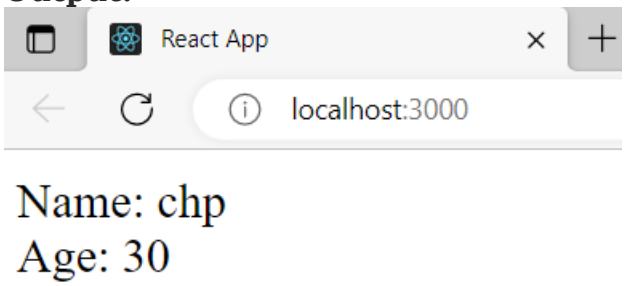
PropTypes is used to specify the expected type of props for a component. It is also defined as a static property on the component class. If the component passes props that do not match the expected types, a warning will be logged to the console in development mode.

Eg.

index.js

```
import React from "react";
import {createRoot} from 'react-dom/client';
import PropTypes from 'prop-types';

class PropsDemo extends React.Component {
  static defaultProps = {
    name: 'chp'
  };
  static propTypes = {
    name: PropTypes.string.isRequired,
    age: PropTypes.number
  };

  render() {
    return (
      <div>
        <div>Name: {this.props.name}</div>
        <div>Age: {this.props.age}</div>
      </div>
    );
  }
}

const root = createRoot(document.getElementById('root'););
root.render(<PropsDemo age={30} />);
```

Output:



In the above example,

if the component does not pass the name prop to PropsDemo, the default value of 'chp' will be used.

PropTypes are used to specify that the name prop is a required string and the age prop is an optional number.

Q) Explain indetail about Component Life Cycle in React.

React components have a life cycle of their own, and they go through a series of phases during their lifetime. These phases are divided into 3 main categories: Mounting, Updating, Unmounting.

1. Mounting: This phase occurs when a component is created and inserted into the DOM. During the Mounting phase, the following methods are called:

i. constructor(): This is the first method that is called when a component is created. It is used to initialize state and bind methods.
ii. componentWillMount(): It is called just before a component is mounted. IT is mainly used to set the initial state of a component, or to perform any actions that should be done before the component is mounted.
iii. render(): It returns a description of what the component should render.
iv. componentDidMount(): It is called after the component has been inserted into the DOM.

2. Updating: This phase occurs when a component is updated due to changes in props or state. During the Updating phase, the following methods are called:

i. shouldComponentUpdate(nextProps, nextState): This method is called before the component is updated. It should return a boolean value indicating whether the component should update or not. By default, it returns true.
ii. componentWillUpdate(nextProps, nextState): method is a lifecycle method in React that is called just before a component is re-rendered due to changes in its state or props. You can use the componentWillUpdate() method to perform any necessary actions before the component is updated.
iii. render(): This method is called to re-render the component with the updated state and/or props.
iv. componentDidUpdate(prevProps, prevState): This method is called after the component has been updated. This is where you can perform any side-effects, such as fetching new data.

3. Unmounting: This phase occurs when a component is removed from the DOM. During the Unmounting phase, the following method is called:
i. componentWillUnmount(): This method is called just before the component is removed from the DOM. This is where you can perform any cleanup, such as removing event listeners or cancelling any pending API requests.

Eg:
index.js:

```
import React from 'react';
import ReactDOM from 'react-dom';
```

```
class LifeCycle extends React.Component {
  constructor(props)
  {
      super(props);
      this.state = { who : "PVPSIT" };
      console.log("In constructor");
  }

  componentWillMount()
  {
      console.log("componentWillMount()");
  }

  componentDidMount()
  {
      console.log("componentDidMount()");
  }

  changeState()
  {
      this.setState({ who : "IT"});
  }

  shouldComponentUpdate(nextProps, nextState)
  {
      console.log("shouldComponentUpdate()");
      return true;
  }


 componentWillUpdate()
  {
      console.log("componentWillUpdate()");
  }

  componentDidUpdate()
  {
      console.log("componentDidUpdate()");
  }

  componentWillUnmount() {
    alert("The component named is about to be unmounted.");
    }

   render()
  {
      return (
          <div>
          <h1>Welcome to , { this.state.who }</h1>
          <h2>
          <button onClick={this.changeState.bind(this)}>Click Here!</button>

          </h2>
          </div>);
```

```
  }
}

ReactDOM.render(
  <Lifecycle />,
  document.getElementById('root')
  );
```

Output: