UNIT - 2 WEB - D WITH MERN NOTES

UNIT 2: REACTJS

1. INTRODUCTION TO REACTJS (Handwritten Notes pg 1, Printed Notes pg 2, 6, 15)

- **Definition (Printed Notes pg 2):** ReactJS is a JavaScript library for building user interfaces (UIs) or UI components. It is used to create reusable UI components and was created by Facebook.
- It allows developers to create large web applications that can change data, without reloading the page.
- Main purpose is to be fast, scalable, and simple.
- It works only on user interfaces in the application. This corresponds to the view in the MVC (Model-View-Controller) template.
- Can be used with a combination of other JavaScript libraries or frameworks, like Angular.
- **Why use React instead of other frameworks, like Angular? (Printed Notes pg 6)**
  1. **Easy creation of dynamic applications:** React makes it easier to create dynamic web applications because it provides less coding and more functionality compared to JavaScript applications which can get complex quickly.
  2. **Improved performance:** React uses a **virtual DOM** (Document Object Model), which makes web applications perform faster. The virtual DOM compares its previous state and updates only those components in the real DOM whose states have changed, rather than updating all components like conventional web applications.
  3. **Reusable components:** Components are the building blocks of any React application. A single app usually consists of multiple components. These components have their own logic and controls and can be reused throughout the application, dramatically reducing development time.
  4. **Unidirectional data flow:** React follows a unidirectional data flow. This means that when designing a React app, child components are often nested within parent components. Since data flows in a single direction, it becomes easier to debug errors and know where the problem occurs.
  5. **Dedicated tools for easy debugging:** Facebook has released a Chrome extension for debugging React applications, making the process faster and easier.
- **Key Features of React (Printed Notes pg 2):**
  1. **JSX (JavaScript XML):** A syntax extension to JavaScript. It is used with React to describe what the user interface should look like. By using JSX, we can write HTML-like structures in the same file that contains JavaScript code.
  2. **Components:** The building blocks of any React application. A single app usually consists of multiple components. React splits the user interface into independent, reusable parts that can be processed separately. [PYQ Q4(a) asks about components and differences between class/functional components]

3. **Virtual DOM:** React keeps a lightweight representation of the real DOM in memory, known as the virtual DOM. When the state of an object changes, the virtual DOM changes only that object in the real DOM, rather than updating all the objects. This leads to faster updates and better performance. [PYQ Q1(b) asks to differentiate Shadow DOM and Virtual DOM, though Shadow DOM is not explicitly React, Virtual DOM is key here]
4. **One-way data-binding (Unidirectional Data Flow):** React's one-way data binding keeps everything modular and fast. A unidirectional data flow means that when designing a React app, child components are often nested within parent components, and data flows downwards (from parent to child via props).
5. **High performance:** React updates only those components that have changed, rather than updating all components at once. This results in much faster web applications.

- **Advantages summarized (Handwritten Notes pg 2):** Reusable Components, Open Source, Efficient and fast, Works in browser, Large Community.

2. GETTING STARTED WITH REACT APP / HOW TO CREATE A REACT APP (Handwritten Notes pg 1, Printed Notes pg 7-8)

- **Prerequisites:**
  1. **Install NodeJS (Printed Notes pg 7):** Node.js is needed because `npm` (Node Package Manager) is used to install the React library and other dependencies. `npm` contains many JavaScript libraries, including React.
     - Check Node.js version: `node -v` (Handwritten Notes pg 1)
- **Steps to Create a React App:**
  1. **Install `create-react-app` package (Printed Notes pg 8):** This is a command-line tool that sets up a new React project with a good default configuration.

     ```
     npm install -g create-react-app  # Installs globally (older method)
     # OR, using npx (recommended, no global install needed)
     npx create-react-app my-react-app
     ```

     (Handwritten Notes pg 1 shows `npm create vite@4.1.0` which is an alternative, faster build tool. Then `Project name: react-app`, `Select a framework: React`, `Select a variant: Javascript`. This is a Vite-specific setup). For `create-react-app`:

     ```
     npx create-react-app project-name
     ```

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

     ```
     cd project-name
     ```

     (Handwritten Notes pg 1: `cd react-app`)
  3. **Install dependencies (if using Vite and not automatically done):**
     (Handwritten Notes pg 1: `npm install` or `npm i`)
  4. **Install a text editor (Printed Notes pg 8):** Like VS Code or Sublime Text.

5. **Start the development server:**

   ```
   npm start
   ```

   (Handwritten Notes pg 1: `npm run dev` for Vite projects)

- **Project Structure (Simplified from Handwritten Notes pg 1 for a `create-react-app` like structure):**
  - `node_modules/`: Contains all project dependencies.
  - `public/`: Contains static assets like `index.html`, images, etc.
    - `index.html`: The main HTML page where the React app is mounted (usually has a `<div id="root"></div>`).
  - `src/`: Contains the JavaScript code for the React application.
    - `App.css`: CSS styles for the `App` component.
    - `App.js` (or `App.jsx`): The main application component.
    - `index.css`: Global CSS styles.
    - `index.js` (or `index.jsx`): The entry point of the React application, renders the `App` component into the DOM.
      (Handwritten Notes pg 1, `main.jsx` for Vite):

      ```
      import React from 'react'
      import ReactDOM from 'react-dom/client' // Note: client for React 18+
      import App from './App' // or App.jsx
      import './index.css'

      ReactDOM.createRoot(document.getElementById('root')).render(
        <React.StrictMode>
          <App />
        </React.StrictMode>
      )
      ```

    - `Message.jsx` (Handwritten Notes pg 1 - example component):

      ```
      // Message.jsx
      function Message() {
        return <h1>Helloworld this is Message</h1>;
      }
      export default Message;
      ```

      And used in `App.jsx`:

      ```
      // App.jsx
      import Message from './Message'; // Assuming Message.jsx is in the
      ```

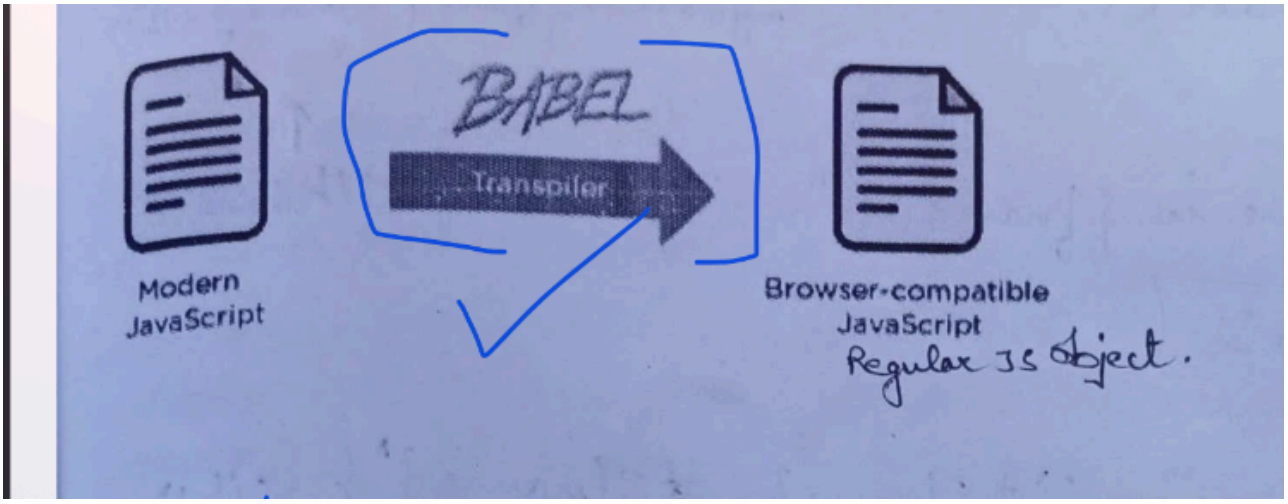
```
same folder

function App() {
  return (
    <div>
      <Message />
    </div>
  );
}
export default App;
```

- `.gitignore`: Specifies intentionally untracked files that Git should ignore.
- `package.json`: Contains project metadata, scripts (like `start`, `build`), and lists dependencies.
- `package-lock.json` (or `yarn.lock`): Records exact versions of dependencies.

3. TEMPLATING USING JSX (Printed Notes pg 2, 3)

- **Definition (Printed Notes pg 2):** JSX stands for JavaScript XML. It is a syntax extension to JavaScript. It is used with React to describe what the user interface should look like. By using JSX, we can write HTML-like structures (or React elements) in the same file that contains JavaScript code.
- It is not valid JavaScript, so it needs to be transpiled into regular JavaScript calls (e.g., `React.createElement()`) by a transpiler like Babel before it can be understood by browsers.
- **Can web browsers read JSX directly? (Printed Notes pg 4):**
  - No, web browsers cannot read JSX directly because they are built to read regular JS objects, and JSX is not a regular JavaScript object.
  - For a web browser to read a JSX file, the file needs to be transformed (transpiled) into a regular JavaScript object. This is typically done using **Babel**.



  - (Diagram on Printed Notes pg 4: Modern JavaScript (JSX) -> BABEL Transpiler -> Browser-compatible JavaScript)

- **Embedding Expressions in JSX:** JavaScript expressions can be embedded within JSX using curly braces `{}`.
  (Printed Notes pg 2, diagram "JSX Expression in React"):

```
// JavaScript
// HTML {JS}
render() {
  const name = "React Developer";
  const sum = 10 + 20;
  return (
    <div>
      <h1>Hello, {name}!</h1> {/* Variable expression */}
      <p>10 + 20 = {sum}</p> {/* Arithmetic expression */}
      <p>My lucky number is {Math.random() * 10}</p> {/* Function call
expression */}
    </div>
  );
}
```

  (Handwritten Notes pg 5, "Template Literals" - this is a JS feature but used here in a React context which is actually JSX. This shows how JS expressions are embedded.)

```
// const fname = "Kritika";
// const lname = "Kalihar";
// ReactDOM.render(
//   <>  {/* React Fragment */}
//     <h1>My name is {fname + " " + lname}</h1>
//     {/* or <h1>My name is {`${fname} ${lname}`}</h1> using template
literals within the expression */}
//     <p>my lucky number is {5+5}</p>
//   </>,
//   document.getElementById("root")
// );
// O/P: My name is Kritika Kalihar. my lucky number is 10.
```

- **JSX Attributes:**
  - HTML attributes are written in camelCase in JSX (e.g., `className` instead of `class`, `htmlFor` instead of `for`).
  - Attribute values can be strings (in quotes) or JavaScript expressions (in curly braces).

```
const myClass = "container";
const element = <div className={myClass} id="main-div">Content</div>;
```

- **JSX represents Objects:** JSX expressions become regular JavaScript function calls that evaluate to JavaScript objects (React elements).

`React.createElement(component, props, ...children)`
For example, `<MyButton color="blue" shadowSize={2}>Click Me</MyButton>` might transpile to something like:
`React.createElement(MyButton, {color: 'blue', shadowSize: 2}, 'Click Me')`

- **JSX requires a single root element:** When returning JSX from a component, you must return a single root element. If you need multiple elements at the top level, wrap them in a container like a `<div>` or a React Fragment (`<> ... </>` or `<React.Fragment> ... </React.Fragment>`).

4. CLASSES USING JSX (CLASS COMPONENTS) (Printed Notes pg 7, 15, 17, 22) [PYQ Q4(a) difference between class and functional components]

- **Definition:** Class components are one way to define components in React. They are ES6 classes that extend `React.Component`.
- They must have a `render()` method that returns JSX (what should be displayed on the screen).
- They can hold and manage their own state (see State and Props section).
- They have access to lifecycle methods (see Lifecycle section).
- **ES5 vs ES6 Component Syntax (Printed Notes pg 7):**
  - **ES5 (using `React.createClass` - now deprecated):**

```
// ES5
var MyComponentES5 = React.createClass({
  render: function() {
    return (
      <h3>Hello Simplilearn (ES5)</h3>
    );
  }
});
```

  - **ES6 (using ES6 classes - standard way):**

```
// ES6
class MyComponentES6 extends React.Component {
  render() {
    return (
      <h3>Hello Simplilearn (ES6)</h3>
    );
  }
}
```

- **Example of a Class Component (Printed Notes pg 17, simplified):**

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

```
class Greeting extends React.Component { // or class Greeting extends
Component
  render() {
    return <h1>Welcome to {this.props.name}</h1>; // Accessing props
  }
}
export default Greeting;
```

(Handwritten Notes pg 18, "Class Component Profile.jsx" and its usage in "App.jsx" provides a good example structure)

5. COMPONENTS (Printed Notes pg 2, 6, 14, 15, 22) [PYQ Q4(a) Components in React JS]

- **Definition (Printed Notes pg 2, 15):** Components are the building blocks of any React application. A single app usually consists of multiple components. A component is essentially a piece of the user interface. React splits the UI into independent, reusable parts that can be processed separately.
- Components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen (Printed Notes pg 14).
- **Why use components?**
  - **Reusability:** Write once, use many times.
  - **Maintainability:** Easier to manage and update smaller, independent pieces of code.
  - **Readability:** Code becomes more organized and understandable.
- **Types of Components (Printed Notes pg 15, 22):**
  1. **Functional Components (Stateless Functional Components):**
     - **Definition:** Simpler way to write components. They are JavaScript functions that accept props as an argument and return JSX.
     - Typically used for presenting UI that doesn't manage its own state or have lifecycle methods (though Hooks like `useState` and `useEffect` now allow functional components to manage state and side effects).
     - They are also called "stateless components" (historically, before Hooks) because they primarily derive data from props.
     - **Example (Printed Notes pg 17, Handwritten Notes pg 16 "Functional component"):**

```
// Functional Component
function Greeting(props) {
  return <h1>Welcome to {props.name}</h1>;
}
// Or using arrow function
// const Greeting = (props) => <h1>Welcome to {props.name}</h1>;

export default Greeting;
```

(Handwritten Notes pg 16 `Profile.jsx` (functional) receiving props and rendering it)

2. **Class Components (Stateful Class Components):**
    - **Definition:** ES6 classes that extend `React.Component`.
    - Can hold and manage their own state using `this.state`.
    - Have access to lifecycle methods (e.g., `componentDidMount`, `componentDidUpdate`).
    - Must have a `render()` method that returns JSX.
    - Called "stateful components" as they can manage internal state.
    - **Example (Printed Notes pg 17, Handwritten Notes pg 18 "Class component Profile.jsx"):**

```jsx
import React from 'react';

class UserProfile extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      location: "New York"
    };
  }

  render() {
    return (
      <div>
        <h1>Hello, {this.props.name} from {this.state.location}!
</h1>
      </div>
    );
  }
}
export default UserProfile;
```

- **Differences between Class and Functional Components (Printed Notes pg 22):** [PYQ Q4(a)]

Feature	Class Components	Functional Components
State	Can hold or manage state (using `this.state`).	Cannot hold or manage state (traditionally; now can with `useState` Hook).
Simplicity	More complex as compared to stateless functional components.	Simple and easy to understand (especially for presentational UI).
Lifecycle methods	Can work with all lifecycle methods.	Does not work with lifecycle methods (traditionally; now can with `useEffect` Hook for side effects).
`this` keyword	Uses `this` to access props and state.	Does not use `this`. Props are passed as arguments.

Feature	Class Components	Functional Components
Syntax	ES6 class extending `React.Component`.	JavaScript function.
Reusability	Can be reused.	Can be reused.

    - **Note:** With the introduction of React Hooks, functional components can now manage state, lifecycle, and other React features that were previously only available in class components, making them more powerful and often preferred.

- **Component Naming:** Component names must always start with a capital letter (Printed Notes pg 14). React treats components starting with lowercase letters as DOM tags (e.g., `<div />` vs `<Welcome />`).
- **Rendering a Component:**

```
const element = <Welcome name="Sara" />; // Using a functional or class component
ReactDOM.render(element, document.getElementById('root'));
```

- **Composing Components:** Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail.

```
function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

- **`render()` method (Printed Notes pg 17):** [PYQ (implied) Q4(a) as part of class components]
    - **Use:** It is required for each class component to have a `render()` function. This function returns the HTML (or more accurately, JSX that becomes HTML) which is to be displayed in the component.
    - If you need to render more than one element from `render()`, all elements must be wrapped inside one parent tag (e.g., `<div>`, `<form>`, or `<React.Fragment>`).
- **Comments in React (JSX) (Printed Notes pg 12):**
    - Comments in JSX are written using curly braces `{}` and JavaScript comment syntax inside.

    1. **Single-line comments (within JSX):**

```
render() {
  return (
```

```
    <div>
      {/* This is a single-line comment in JSX */}
      <h1>Hello</h1>
    </div>
  );
}
```

2. **Multi-line comments (within JSX):**

```
render() {
  return (
    <div>
      {/*
        This is a
        multi-line comment
        in JSX
      */}
      <p>World</p>
    </div>
  );
}
```

(Printed notes pg 12 shows a Python comment example, but for React/JSX, the above is correct).

- **Embedding Two or More Components into One (Printed Notes pg 21):** This is done by simply including the component tags within the JSX of a parent component.

```
class Simple extends React.Component {
  render() {
    return (<h1>Simplilearn</h1>);
  }
}

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello</h1>
        <Simple /> {/* Embedding Simple component */}
      </div>
    );
  }
}
ReactDOM.render(<App />, document.getElementById('index'));
```

6. STATE AND PROPS (Printed Notes pg 17, 18, 19, 20, 21) [PYQ Q5(a) Explain State and Props in React JS. Give an example to update the state of component.]

- **Props (Properties) (Printed Notes pg 14, 20):**
  - **Definition:** Props (short for properties) are arguments passed into React components. They are a way to pass data from a parent component to a child component.
  - Props are read-only; a component must never modify its own props. This makes components pure (given the same inputs, they always render the same output).
  - Passed to components similar to how arguments are passed in a function or attributes in HTML.
  - **How to pass props (Printed Notes pg 20, Handwritten Notes pg 16):**
    Parent Component:

```jsx
// App.js (Parent)
import Profile from './Profile';

function App() {
  return (
    <div>
      {/* Passing string prop */}
      <Profile text="Hello Props" />
      {/* Passing object prop (Handwritten Notes pg 16) */}
      <Profile data={{ name: 'Peter', detail: 'Profile data' }} />
    </div>
  );
}
```

    Child Component (receiving props):

```jsx
// Profile.js (Child - Functional Component)
function Profile(props) {
  return (
    <div>
      <h1>{props.text}</h1>
      {props.data && ( // Check if data prop exists
        <h2>Name: {props.data.name}, Detail: {props.data.detail}</h2>
      )}
    </div>
  );
}
// For Class Component (Handwritten Notes pg 18 "Class component
Profile.jsx")
// class Profile extends React.Component {
//   render() {
```

```
//     return <h1>{this.props.text}</h1>;
//     // return <h1>{this.props.data.name}</h1>;
//   }
// }
```

- **State (Printed Notes pg 19):**
  - **Definition:** State is a built-in React object that is used to contain data or information about the component. A component's state can change over time, typically in response to user actions or network responses. Whenever the state of a component changes, the component re-renders.
  - State is private and fully controlled by the component. It is not accessible to any other component unless explicitly passed down as props.
  - Used primarily in Class Components (though Functional Components can use state via the `useState` Hook).
  - **Implementing State in Class Components (Printed Notes pg 19):**
    1. **Initialize state in the constructor:**

```
class Clock extends React.Component {
  constructor(props) {
    super(props); // Always call super(props) in constructor
    this.state = { date: new Date(), message: "Welcome to
Simplilearn" };
  }
  // ...
}
```

(Handwritten Notes pg 18 "State -> Profile.jsx (component)" initializing state)

    2. **Access state:** Use `this.state.propertyName`.

```
render() {
  return <h1>{this.state.message}</h1>;
}
```

  - **Updating State (Printed Notes pg 19):** [PYQ Q5(a) Give an example to update the state of component.]
    - Use the `this.setState()` method. **Never modify `this.state` directly.**
    - `setState()` schedules an update to a component's state object. When state changes, the component responds by re-rendering.
    - `setState()` is asynchronous. React may batch multiple `setState()` calls for performance.
    - **Example (Printed Notes pg 19, 20, Handwritten Notes pg 19):**

```
// Profile.jsx (Class Component with state update)
// (Handwritten Notes pg 19)
```

```
class Profile extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'Peter',
      email: 'peter@test.com'
    };
    // Binding is necessary to make `this` work in the callback for
older JS syntax
    // or if not using arrow functions for event handlers.
    this.updateState = this.updateState.bind(this);
  }

  updateState() {
    this.setState({
      name: 'Bruce',
      email: 'bruce@test.com'
    });
  }

  render() {
    return (
      <div>
        <h1>Hello {this.state.name}</h1>
        <p>Email: {this.state.email}</p>
        <button onClick={this.updateState}>Update State</button>
        {/* Alternatively, using an arrow function in the onClick
handler:
        <button onClick={() => this.setState({ name: 'Bruce'
})}>Update Name</button>
        */}
      </div>
    );
  }
}
// Output before click: Hello Peter, Email: peter@test.com
// Output after click: Hello Bruce, Email: bruce@test.com
```

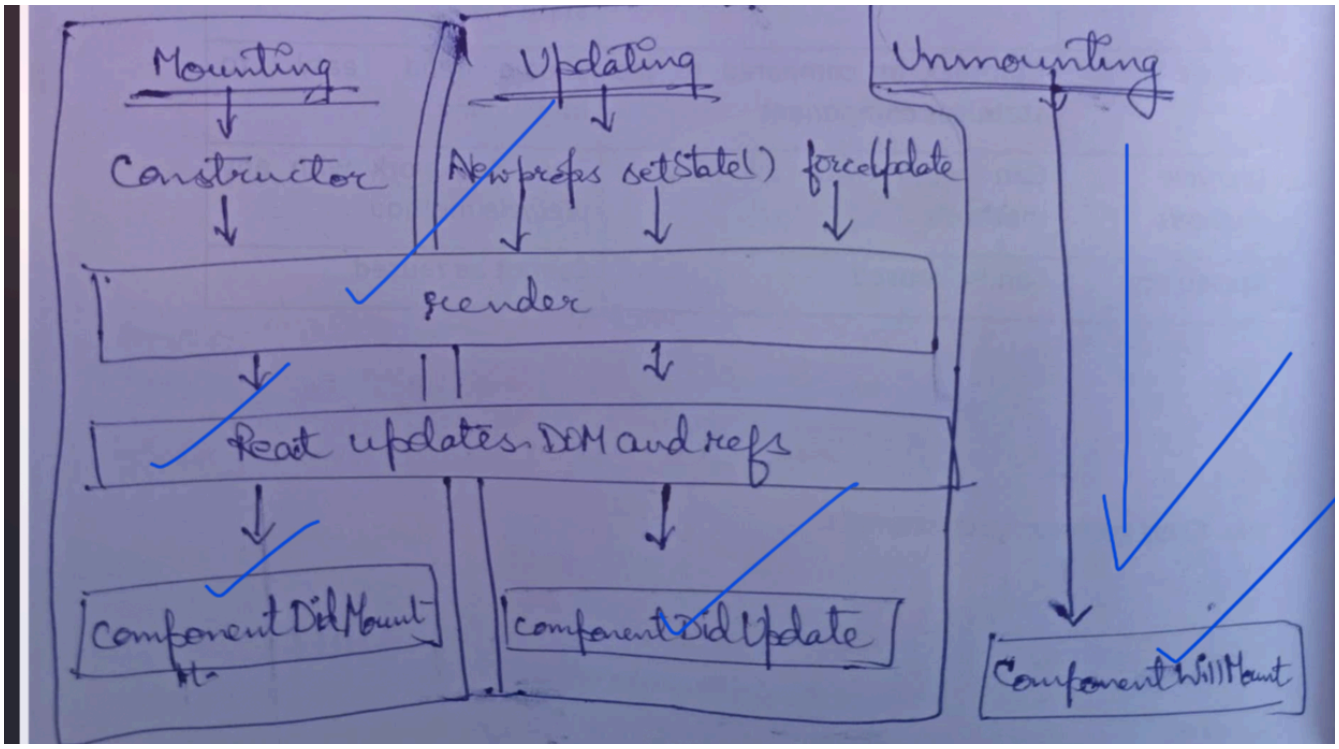
(Printed Notes pg 20 `App` component shows `this.buttonPress.bind(this)` and `this.setState` in `buttonPress` method).

- **Differences between State and Props (Printed Notes pg 21):** [PYQ Q5(a)]

Feature	State	Props
Use	Holds information about the component's internal data that can change.	Allows passing data from one component to another (parent to child).
Mutability	Is mutable (can be changed using `setState()`).	Are immutable (read-only within the component receiving them).
Read-Only	Can be changed by the component itself.	Are read-only by the child component.
Child Components Access	Child components cannot directly access or modify parent's state.	Child component can access props passed by its parent.
Stateless Components	Functional components traditionally cannot have state (use `useState` Hook now).	Functional components can receive and use props.
Ownership	Owned and managed by the component itself.	Passed from parent; owned by the parent.

7. LIFECYCLE OF COMPONENTS (Printed Notes pg 23, 24) [PYQ (related to Q5(a) indirectly as state changes trigger lifecycle methods)]

- **Definition:** Each React component has a lifecycle which you can monitor and manipulate during its three main phases: Mounting, Updating, and Unmounting. Lifecycle methods are special methods that get called automatically at particular points in a component's life.
- Primarily applicable to Class Components (Functional Components use the `useEffect` Hook to achieve similar functionality).
- **Phases and Key Lifecycle Methods (Printed Notes pg 23 shows a diagram, pg 24 lists methods):**



1. **Mounting (Component is being created and inserted into the DOM):**
    - Order of execution:
        1. `constructor(props)`:
            - Called before the component is mounted.
            - Used for initializing state (`this.state = ...`) and binding event handlers.
            - Must call `super(props)` first if it's a subclass.
        2. `static getDerivedStateFromProps(props, state)`: (Less common)
            - Called right before `render()`, both on initial mount and on subsequent updates.
            - Used when the state depends on changes in props over time.
            - Should return an object to update the state, or `null` to update nothing.
        3. `render()`:
            - **Required method.**
            - Examines `this.props` and `this.state` and returns one of the following: React elements (JSX), Arrays and fragments, Portals, String and numbers, Booleans or `null` (renders nothing).
            - Should be a pure function of props and state.
        4. `componentDidMount()`: [PYQ (common knowledge)]
            - Invoked immediately after a component is mounted (inserted into the tree/DOM).
            - Good place for network requests (e.g., fetch data from an API), DOM manipulations (if necessary), or setting up subscriptions.
            - `setState()` can be called here (will trigger an extra rendering, but it will happen before the browser updates the screen).

2. **Updating (Component is being re-rendered as a result of changes to either its props or state):**
    - Order of execution:
        1. `static getDerivedStateFromProps(props, state)`: (Called again)
        2. `shouldComponentUpdate(nextProps, nextState)`:
            - Invoked before rendering when new props or state are being received. Defaults to `true`.
            - Returns a boolean value indicating whether React should continue with the rendering or skip it (for performance optimization).
            - Not commonly used for deep equality checks; `React.PureComponent` or manual checks are alternatives.
        3. `render()`: (Called again if `shouldComponentUpdate` returns `true`)
        4. `getSnapshotBeforeUpdate(prevProps, prevState)`: (Less common)
            - Called right before the most recently rendered output is committed to the DOM.
            - Enables your component to capture some information from the DOM (e.g., scroll position) before it is potentially changed.
            - Any value returned by this will be passed as the third parameter to `componentDidUpdate()`.
        5. `componentDidUpdate(prevProps, prevState, snapshot)`:
            - Invoked immediately after updating occurs (not called for the initial render).
            - Good place for operating on the DOM when the component has been updated, or for network requests based on prop changes.
            - Can call `setState()` here, but it must be wrapped in a condition (e.g., comparing `prevProps` with `this.props`) to avoid an infinite loop.
3. **Unmounting (Component is being removed from the DOM):**
    - Order of execution:
        1. `componentWillUnmount()`: [PYQ (common knowledge)]
            - Invoked immediately before a component is unmounted and destroyed.
            - Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in `componentDidMount()`.

- **Older/Deprecated Lifecycle Methods (Not listed in the syllabus provided's image of React topics, but good to be aware they existed):**
    - `componentWillMount()` (Deprecated)
    - `componentWillReceiveProps(nextProps)` (Deprecated)
    - `componentWillUpdate(nextProps, nextState)` (Deprecated)

- **Error Handling Lifecycle Methods (See Error Handling section):**
  - `static getDerivedStateFromError(error)`
  - `componentDidCatch(error, info)`

8. RENDERING LISTS (Printed Notes pg 10, 11)

- **Definition:** Rendering a collection of items (like an array) as a list of elements in React.
- This is done by iterating over the array and returning a React element for each item. The JavaScript `map()` array method is commonly used for this.
- **How lists work in React (Printed Notes pg 11):**
  - We create lists in React similar to how we do in regular JavaScript. Lists display data in an ordered (or unordered) format.
  - The traversal of lists is typically done using the `map()` function.
- **Using the `map()` function:**
  - The `map()` function creates a new array by calling a provided function on every element in the calling array.
  - In React, you can use `map()` to take an array of data and transform it into an array of JSX elements.

```
function NumberList(props) {
  const numbers = props.numbers; // e.g., [1, 2, 3, 4, 5]
  const listItems = numbers.map((number) =>
    // Each list item needs a unique "key" prop (see below)
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbersArray = [1, 2, 3, 4, 5];
// ReactDOM.render(
//   <NumberList numbers={numbersArray} />,
//   document.getElementById('root')
// );
```

(Printed Notes pg 11 shows an example with `const names = ['Kohli', 'Saif', ...];` and then mapping `names` to `<li>` elements).

- **Keys in Lists (Printed Notes pg 11):** [PYQ (common interview/React knowledge)]

- **Definition:** Keys are special string attributes you need to include when creating lists of elements.
- **Why use keys?**
  1. **Unique Identifier:** A key is a unique identifier that helps React identify which items have changed, been added, or been removed from the list.
  2. **Efficient Updates:** Keys help React to efficiently update the UI. When items in a list are reordered, added, or removed, React uses keys to match children in the original tree with children in the subsequent tree. Without keys, React might have to re-render more components than necessary, or might lose component state.
  3. **Determining Re-renders:** It also helps determine which components need to be re-rendered instead of re-rendering all components every time. This increases performance, as only the updated components are re-rendered.
- **Choosing a Key:**
  - Keys should be **stable, predictable, and unique** among its siblings.
  - Often, data from your backend will have a unique ID which is perfect for a key.
  - Using the **array index as a key is generally discouraged** if the order of items may change, as it can lead to issues with component state and performance. It's acceptable if the list is static and items are never reordered.
- **Example with keys (from Printed Notes pg 11):**

```
const names = ['Kohli', 'Saif', 'Arun', 'Aamir', 'Arif'];

const ListOfNames = () => {
  const listItems = names.map((name, index) => // Using name as key, assuming names are unique
    <li key={name}>
      {name}
    </li>
  );
  // If names weren't unique, you might use index (with caution) or a unique ID from data:
  // const listItems = data.map((item) => <li key={item.id}>{item.text}</li>);

  return (
    <ul>{listItems}</ul>
  );
};
```

9. PORTALS

- **Definition:** Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.
- Normally, when you return an element from a component's `render` method, it's mounted into the DOM as a child of the nearest parent DOM node. Portals allow you to break out of this containment.
- **Use Cases:**
  - Modals, dialogs, tooltips, loaders, or any UI element that needs to appear on top of the rest of the application, often directly under `<body>` or a specific portal `<div>`.
  - Managing `z-index`, `overflow`, and event bubbling can be simpler when elements are rendered in a separate DOM tree.
- **Syntax:**

```
ReactDOM.createPortal(child, container)
```

  - `child`: Any renderable React child, such as an element, string, or fragment.
  - `container`: A DOM element that exists in the HTML structure (e.g., `document.getElementById('portal-root')`).
- **Example:**
  - In your `public/index.html`:

```
<div id="root"></div>
<div id="modal-root"></div> {/* The container for the portal */}
```

  - Your React component:

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

class Modal extends React.Component {
  render() {
    // React does not create a new div. It renders the children into `domNode`.
    // `domNode` is any valid DOM node, regardless of its location in the DOM.
    return ReactDOM.createPortal(
      this.props.children, // The content to render (e.g., the modal UI)
      document.getElementById('modal-root')
    );
  }
}

class App extends React.Component {
```

```
  constructor(props) {
    super(props);
    this.state = { showModal: false };
    this.handleShow = this.handleShow.bind(this);
    this.handleHide = this.handleHide.bind(this);
  }

  handleShow() {
    this.setState({ showModal: true });
  }

  handleHide() {
    this.setState({ showModal: false });
  }

  render() {
    const modal = this.state.showModal ? (
      <Modal>
        <div className="modal-content">
          This is a modal!
          <button onClick={this.handleHide}>Hide modal</button>
        </div>
      </Modal>
    ) : null;

    return (
      <div className="app">
        This is the main app content.
        <button onClick={this.handleShow}>Show modal</button>
        {modal}
      </div>
    );
  }
}

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

- **Event Bubbling through Portals:** Even though a portal can be anywhere in the DOM tree, it behaves like a normal React child in every other way, including event bubbling. An event fired from inside a portal will propagate to ancestors in the containing React tree, even if those ancestors are not ancestors in the DOM tree.

10. ERROR HANDLING (ERROR BOUNDARIES)

- **Definition:** Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.
- They catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.
- Error boundaries do **not** catch errors for:
  - Event handlers (use regular `try...catch` here).
  - Asynchronous code (e.g., `setTimeout` or `requestAnimationFrame` callbacks).
  - Server-side rendering.
  - Errors thrown in the error boundary itself (rather than its children).
- **How to create an Error Boundary:**
  - A class component becomes an error boundary if it defines one or both of these lifecycle methods:
    1. `static getDerivedStateFromError(error)`: This method is called during the "render" phase, so side-effects are not permitted. It should return an object to update state, which will then cause the fallback UI to be rendered.
    2. `componentDidCatch(error, errorInfo)`: This method is called during the "commit" phase, so side-effects are allowed. It is used for things like logging the error to an error reporting service.
- **Example:**

```
import React from 'react';

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false, error: null, errorInfo: null };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true, error: error };
  }

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    console.error("Uncaught error:", error, errorInfo);
    this.setState({ errorInfo: errorInfo });
  }
```

```
  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return (
        <div>
          <h1>Something went wrong.</h1>
          <details style={{ whiteSpace: 'pre-wrap' }}>
            {this.state.error && this.state.error.toString()}
            <br />
            {this.state.errorInfo && this.state.errorInfo.componentStack}
          </details>
        </div>
      );
    }
    // Normally, just render children
    return this.props.children;
  }
}

// Example of a component that might throw an error
class BuggyCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { counter: 0 };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(({counter}) => ({
      counter: counter + 1
    }));
  }

  render() {
    if (this.state.counter === 3) {
      // Simulate a JS error
      throw new Error('I crashed!');
    }
    return <h1 onClick={this.handleClick}>{this.state.counter}</h1>;
  }
}

// Usage
```

```
// function App() {
//   return (
//     <div>
//       <p>Click the counter three times to trigger an error.</p>
//       <hr />
//       <ErrorBoundary>
//         <p>These two counters are inside the same error boundary.</p>
//         <BuggyCounter />
//         <BuggyCounter />
//       </ErrorBoundary>
//       <hr />
//       <p>These two counters are each inside of their own error
boundary.</p>
//       <ErrorBoundary><BuggyCounter /></ErrorBoundary>
//       <ErrorBoundary><BuggyCounter /></ErrorBoundary>
//     </div>
//   );
// }
```

- **Placement:** The granularity of error boundaries is up to you. You might wrap top-level route components to display a generic error message for the entire page, or wrap individual widgets in an error boundary to protect the rest of the application from crashing.

11. **ROUTERS (REACT ROUTER) (Printed Notes pg 27, 28)**

- **Definition (Printed Notes pg 27):** React Router is a standard routing library built on top of React, which is used to create routes in a React application for navigating between different views or components. It enables building single-page applications (SPAs) with navigation without a full page refresh.
- **Why do we need React Router? (Printed Notes pg 27):**
  - It maintains consistent structure and behavior and is used to develop single-page web applications.
  - Enables multiple views in a single application by defining multiple routes in the React application.
- **React routing vs. Conventional routing (Printed Notes pg 27):**

Feature	React Routing (SPA)	Conventional Routing (Multi-Page Application)
Page Structure	Single HTML page (typically `index.html`).	Each view is often a new HTML file.
Navigation	User navigates multiple views within the same file/DOM.	User navigates multiple files for each view.
Page Refresh	Page does not refresh (only relevant parts update).	Page refreshes every time the user navigates.

Feature	React Routing (SPA)	Conventional Routing (Multi-Page Application)
Performance	Generally improved performance and faster transitions.	Slower performance due to full page reloads.

- **Key Components of React Router (v5/v6 - syntax varies slightly):**
  - `<BrowserRouter>` (or `<HashRouter>`): Wraps your entire application to enable routing. `BrowserRouter` uses the HTML5 history API, while `HashRouter` uses the URL hash (`#`).
  - `<Routes>` (v6) / `<Switch>` (v5): Renders the first child `<Route>` or `<Redirect>` that matches the current location.
  - `<Route>`: Defines a mapping between a URL path and a component to render.
    - `path`: The URL path to match (e.g., `/`, `/about`, `/users/:id`).
    - `element` (v6) / `component` or `render` (v5): The component to render when the path matches.
  - `<Link>` (or `<NavLink>`): Used to create navigation links, similar to HTML `<a>` tags, but handles navigation within the React app without a full page reload.
- **Implementing React Routing (Printed Notes pg 28 shows a v5-like example):**
  - **Install:** `npm install react-router-dom`
  - **Example (Conceptual, adapted for v6 simplicity):**

```
// App.js
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link } from 'react-
router-dom';

// Define some components for different pages
const HomePage = () => <h1>Home Page</h1>;
const AboutPage = () => <h1>About Us</h1>;
const ContactPage = () => <h1>Contact Us</h1>;
const NotFoundPage = () => <h1>404 - Page Not Found</h1>;

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li><Link to="/">Home</Link></li>
            <li><Link to="/about">About</Link></li>
            <li><Link to="/contact">Contact</Link></li>
```

```
          </ul>
        </nav>
        <hr />
        <h1>React Router Example</h1> {/* (From Printed Notes pg 28)
*/}

        <Routes>
          <Route path="/" element={<HomePage />} /> {/* (App component
in printed notes example) */}
          <Route path="/about" element={<AboutPage />} />
          <Route path="/contact" element={<ContactPage />} />
          <Route path="*" element={<NotFoundPage />} /> {/* Catch-all
for 404 */}
        </Routes>
      </div>
    </Router>
  );
}
export default App;

// In your index.js:
// import ReactDOM from 'react-dom/client';
// import App from './App';
// const root = ReactDOM.createRoot(document.getElementById('root'));
// root.render(<App />);
```

(The example on Printed Notes pg 28 uses `component={App}` which is v5 syntax; v6 uses `element={<App />}`).

12. REDUX (Printed Notes pg 24, 25, 27)

- **Definition (Printed Notes pg 24):** Redux is an open-source, JavaScript library used to manage the application state. React often uses Redux to build the user interface. It is a predictable state container for JavaScript applications and is used for the entire application's state management.
- Redux helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.
- **Core Concepts/Components of Redux (Printed Notes pg 25):**
  1. **Store:**
     - **Definition:** A single JavaScript object that holds the entire state of your application.
     - It's the single source of truth.
     - The only way to change the state is by dispatching actions.
     - Created using `createStore(reducer)`.
  2. **Action:**

- **Definition:** Plain JavaScript objects that represent an "intention" to change the state. They are the source information for the store.
- Actions must have a `type` property (usually a string constant) that indicates the type of action being performed.
- They can optionally have a `payload` property containing data.
- Example: `{ type: 'ADD_TODO', text: 'Learn Redux' }`

3. **Reducer:**
   - **Definition:** Pure functions that specify how the application's state changes in response to actions sent to the store.
   - A reducer takes the previous state and an action as arguments and returns the next state.
   - `(previousState, action) => newState`
   - Must be pure: no side effects, no API calls, no mutations of input arguments. Always return a new state object.
   - Multiple reducers can be combined using `combineReducers()`.
4. **Dispatch:**
   - The method `store.dispatch(action)` is used to send an action to the Redux store. This is the only way to trigger a state change.
5. **Subscribe:**
   - The method `store.subscribe(listener)` allows UI components (or other parts of the app) to listen for state changes in the store. The `listener` function will be called whenever an action is dispatched and the state might have changed.

- **Data Flow in Redux (Unidirectional):**
  1. User interacts with the UI (e.g., clicks a button).
  2. UI dispatches an **Action** (e.g., `{ type: 'INCREMENT' }`).
  3. The **Store** calls the **Reducer** with the current state and the dispatched action.
  4. The Reducer calculates the new state based on the action and returns it.
  5. The Store saves the new state returned by the reducer.
  6. The Store notifies all subscribed UI components (listeners) about the state change.
  7. Subscribed UI components re-render based on the new state.
  - (Diagram on Printed Notes pg 25: Action -> Dispatcher (part of Store) -> Store -> View. View triggers new Action).
- **Redux vs. Flux (Printed Notes pg 27):**

SN	Redux	Flux
1.	Redux is an open-source JavaScript library used to manage application State.	Flux is an application architecture (pattern) by Facebook, not a framework or library itself.

SN	Redux	Flux
2.	Store's state is immutable (reducers return new state).	Store's state can be mutable (though immutability is often encouraged).
3.	Can only have a single-store for the entire application.	Can have multiple stores.
4.	Uses the concept of a reducer.	Uses the concept of a dispatcher (central hub for actions).

13. REDUX SAGA

- **Definition:** Redux Saga is a middleware for Redux that aims to make application side effects (i.e., asynchronous things like data fetching and impure things like accessing the browser cache) easier to manage, more efficient to execute, easy to test, and better at handling failures.
- It uses ES6 Generator functions to make asynchronous flows look like standard synchronous JavaScript code (easier to read and write).
- **Why use Redux Saga?**
  - In Redux, reducers must be pure functions, meaning they cannot have side effects. Asynchronous API calls are side effects.
  - Redux Saga provides a dedicated place to manage these side effects, keeping your action creators and reducers clean.
- **Core Concepts:**
  - **Sagas:** Generator functions (`function* mySaga() {}`).
  - **Effects:** Plain JavaScript objects that describe the side effect to be performed. Sagas yield these effect objects to the Redux Saga middleware, which then executes them and resumes the Saga.
    - `call(fn, ...args)`: Calls a function. If it returns a Promise, Saga pauses until the Promise resolves.
    - `put(action)`: Dispatches an action to the Redux store.
    - `take(pattern)`: Pauses until an action matching the pattern is dispatched.
    - `takeEvery(pattern, saga, ...args)`: Spawns a new saga task on each action matching the pattern.
    - `takeLatest(pattern, saga, ...args)`: Spawns a new saga task, but if a previous task for the same pattern is still running, it will be cancelled.
    - `select(selector, ...args)`: Gets a piece of state from the Redux store.
- **Example Flow (Conceptual):**
  1. A React component dispatches an action like `{ type: 'FETCH_USER_REQUEST', userId: '123' }`.
  2. A "watcher" Saga (using `takeEvery` or `takeLatest`) is listening for `FETCH_USER_REQUEST`.

3. When the action is dispatched, the watcher Saga calls a "worker" Saga.
4. The worker Saga might use `call` to make an API request: `const user = yield call(Api.fetchUser, action.userId)`.
5. If the API call is successful, the worker Saga dispatches a success action: `yield put({ type: 'FETCH_USER_SUCCESS', user: user })`.
6. If the API call fails, it dispatches a failure action: `yield put({ type: 'FETCH_USER_FAILURE', error: error.message })`.
7. Reducers handle `FETCH_USER_SUCCESS` or `FETCH_USER_FAILURE` to update the store state.

- **Setup:**
  1. Install: `npm install redux-saga`
  2. Create Saga middleware: `const sagaMiddleware = createSagaMiddleware()`
  3. Apply middleware to the Redux store: `const store = createStore(rootReducer, applyMiddleware(sagaMiddleware))`
  4. Run your root saga: `sagaMiddleware.run(rootSaga)`

14. IMMUTABLE.JS

- **Definition:** Immutable.js is a library created by Facebook that provides several persistent immutable data structures including: `List`, `Stack`, `Map`, `OrderedMap`, `Set`, `OrderedSet`, and `Record`.
- **Immutability:** An immutable object is an object whose state cannot be modified after it is created. When you want to "change" an immutable object, you get a new object with the new values, while the original object remains unchanged.
- **Why use Immutable.js with React/Redux?**
  1. **Performance Optimization:**
     - React components often re-render if their props or state change. Comparing complex mutable objects can be expensive.
     - With immutable objects, change detection is very fast: if the reference to an object is the same, the object hasn't changed. This works well with React's `shouldComponentUpdate` or `React.PureComponent` for optimizing re-renders.
     - Redux reducers must be pure and return new state objects. Immutable.js makes it easier to manage state updates immutably.
  2. **Change Tracking & Undo/Redo:** Since previous versions of the data are preserved, implementing features like undo/redo becomes simpler.
  3. **Predictability:** Easier to reason about state changes because data structures don't change unexpectedly.
- **Key Data Structures and Operations:**
  - **`Map` (like JavaScript Object):**

```
import { Map } from 'immutable';
let map1 = Map({ a: 1, b: 2, c: 3 });
let map2 = map1.set('b', 50); // map1 is still {a:1, b:2, c:3}
                              // map2 is Map { "a": 1, "b": 50, "c": 3
}
console.log(map1.get('b')); // 2
console.log(map2.get('b')); // 50
```

- **`List` (like JavaScript Array):**

```
import { List } from 'immutable';
let list1 = List([1, 2, 3]);
let list2 = list1.push(4);    // list1 is still List [ 1, 2, 3 ]
                              // list2 is List [ 1, 2, 3, 4 ]
let list3 = list2.set(0, 100); // list3 is List [ 100, 2, 3, 4 ]
console.log(list1.get(0));    // 1
```

- **Considerations:**
  - **Learning Curve:** Introduces a new API for data manipulation.
  - **Interoperability:** Need to convert to/from regular JavaScript objects/arrays when interacting with other libraries (`toJS()`, `fromJS()`).
  - **Bundle Size:** Adds to the application's bundle size.
  - While powerful, sometimes simpler immutability patterns in plain JavaScript (like using spread syntax for objects/arrays) might be sufficient.

15. SERVICE SIDE RENDERING (SSR) WITH REACT

- **Definition:** Server-Side Rendering (SSR) is the ability of an application to render requested web pages on the server instead of rendering them in the browser (Client-Side Rendering - CSR, which is the default for typical React SPAs).
- With SSR, the server sends a fully rendered HTML page to the browser. The browser can then display the content immediately. After the initial HTML is loaded, client-side JavaScript (React) can "hydrate" the static HTML, making it interactive.
- **How it works (Simplified):**
  1. User requests a page from the server.
  2. The server (often a Node.js environment) runs the React application code.
  3. React components are rendered to an HTML string on the server (e.g., using `ReactDOMServer.renderToString()`).
  4. The server sends this fully formed HTML document as a response to the browser.
  5. The browser displays the HTML content quickly (Fast Time to Content - TTC).
  6. The browser then downloads the client-side JavaScript bundle.
  7. React "hydrates" the existing server-rendered HTML, attaching event handlers and making the page interactive (Time to Interactive - TTI).

- **Benefits of SSR:**
    1. **Improved SEO:** Search engine crawlers can better understand and index content because they receive fully rendered HTML pages, rather than a nearly empty HTML shell that requires JavaScript to populate.
    2. **Faster Perceived Performance (Faster Time to Content):** Users see content sooner because the browser doesn't have to wait for all the JavaScript to download and execute before displaying the page.
    3. **Better User Experience on Slow Connections/Devices:** Users with slow internet or less powerful devices can see content more quickly.
- **Challenges/Drawbacks of SSR:**
    1. **Increased Server Load:** Rendering on the server consumes server resources.
    2. **More Complex Setup:** SSR typically requires a more complex build process and server configuration (e.g., using Node.js with Express).
    3. **Slower Time to First Byte (TTFB):** The server needs time to render the page, so the initial response might be slightly delayed compared to just sending a static HTML shell.
    4. **Full Page Reloads (if not an SPA after initial load):** If not implemented carefully within an SPA architecture, navigation might still cause full page reloads after the initial SSR.
- **Frameworks for SSR with React:**
    - **Next.js:** A popular React framework that provides SSR, Static Site Generation (SSG), and other features out-of-the-box.
    - **Remix:** Another full-stack web framework that leverages web fundamentals and server-side rendering.
    - Custom solutions using Node.js/Express and `ReactDOMServer`.

16. **UNIT TESTING (IN REACT)** [PYQ Q5(b)(iii) Unit Testing]

- **Definition:** Unit testing is a software testing method by which individual units of source code (components, functions, modules) are tested in isolation to determine if they are fit for use.
- In React, this often means testing individual components to ensure they render correctly for given props and state, and that their event handlers and logic work as expected.
- **Why Unit Test React Components?**
    - **Ensure Correctness:** Verify that components behave as intended.
    - **Prevent Regressions:** Catch bugs early when making changes to the codebase.
    - **Improve Code Quality:** Writing testable code often leads to better-designed components.
    - **Facilitate Refactoring:** Provides confidence when refactoring code.
    - **Documentation:** Tests can serve as a form of documentation for how components are supposed to work.
- **Tools for Unit Testing React:**

1. **Jest:** A popular JavaScript testing framework developed by Facebook. It's often used as the test runner and provides assertion libraries and mocking capabilities. `create-react-app` comes with Jest pre-configured.
2. **React Testing Library (RTL):** A library that provides utilities to test React components in a way that resembles how users interact with them. It encourages testing behavior rather than implementation details. It focuses on querying the DOM based on accessibility attributes and user-visible text.
3. **Enzyme (Older, less recommended now):** A JavaScript testing utility for React that makes it easier to assert, manipulate, and traverse your React Components' output. Provides shallow rendering, full DOM rendering, and static rendering.

- **What to Test in a React Component:**
  - **Rendering:** Does the component render correctly for different props and states? Does it render child components?
  - **User Interaction:** Do event handlers (clicks, input changes) work correctly? Does the state update as expected?
  - **Conditional Logic:** Does the component render different UI based on conditions?
  - **Props:** Does the component handle various props correctly?
- **Example using Jest and React Testing Library (Conceptual):**
  - Component to test (`Greeting.js`):

```
// Greeting.js
import React, { useState } from 'react';

function Greeting({ initialName = "Guest" }) {
  const [name, setName] = useState(initialName);
  const [inputValue, setInputValue] = useState("");

  const handleChangeName = () => {
    if (inputValue.trim() !== "") {
      setName(inputValue);
      setInputValue("");
    }
  };

  return (
    <div>
      <h1>Hello, {name}!</h1>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
```

```
        placeholder="Enter a name"
      />
      <button onClick={handleChangeName}>Change Name</button>
    </div>
  );
}
export default Greeting;
```

- Test file (`Greeting.test.js`):

```
// Greeting.test.js
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import '@testing-library/jest-dom'; // for extended matchers
import Greeting from './Greeting';

describe('Greeting Component', () => {
  test('renders with default initial name', () => {
    render(<Greeting />);
    expect(screen.getByText('Hello, Guest!')).toBeInTheDocument();
  });

  test('renders with provided initial name', () => {
    render(<Greeting initialName="Alice" />);
    expect(screen.getByText('Hello, Alice!')).toBeInTheDocument();
  });

  test('changes name when button is clicked with input value', () => {
    render(<Greeting />);
    const inputElement = screen.getByPlaceholderText('Enter a name');
    const buttonElement = screen.getByText('Change Name');

    fireEvent.change(inputElement, { target: { value: 'Bob' } });
    fireEvent.click(buttonElement);

    expect(screen.getByText('Hello, Bob!')).toBeInTheDocument();
    expect(inputElement.value).toBe(''); // Input should be cleared
  });

  test('does not change name if input is empty when button is clicked',
() => {
    render(<Greeting initialName="Charlie" />);
    const buttonElement = screen.getByText('Change Name');
```

```
    fireEvent.click(buttonElement); // Input is empty

    expect(screen.getByText('Hello, Charlie!')).toBeInTheDocument(); //
Name should not change
  });
});
```

17. WEBPACK

- **Definition:** Webpack is a powerful, open-source static module bundler for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph which maps every module your project needs and generates one or more bundles.
- **Core Concept: Module Bundler**
  - In modern web development, applications are often split into many files (modules) for better organization (e.g., JavaScript files, CSS files, images, fonts).
  - Browsers, especially older ones, might not efficiently handle loading many individual files or might not understand certain module formats (like ES6 modules directly without `<script type="module">`).
  - Webpack takes these modules with their dependencies and packages them into a smaller number of optimized static assets (bundles, usually JavaScript files) that can be efficiently loaded by a browser.
- **Key Features and Benefits:**
  1. **Dependency Graph:** Webpack starts from entry points you define, recursively builds a dependency graph that includes every module your application needs.
  2. **Loaders:** Webpack only understands JavaScript and JSON files by default. Loaders allow webpack to process other types of files (e.g., CSS, SASS, TypeScript, Babel for ES6+, images, fonts) and convert them into valid modules that can be added to your dependency graph and ultimately to your bundles.
     - `babel-loader`: Transpiles ES6+ JavaScript to ES5.
     - `css-loader`: Interprets `@import` and `url()` like `import/require()` and will resolve them.
     - `style-loader`: Injects CSS into the DOM via `<style>` tags.
     - `file-loader`, `url-loader`: For handling images and fonts.
  3. **Plugins:** Plugins can perform a wider range of tasks like bundle optimization, asset management, and environment variable injection. They hook into webpack's build process.
     - `HtmlWebpackPlugin`: Simplifies creation of HTML files to serve your webpack bundles.
     - `MiniCssExtractPlugin`: Extracts CSS into separate files instead of injecting it into JavaScript.
     - `TerserWebpackPlugin`: Minifies JavaScript.

4. **Code Splitting:** Allows you to split your code into various bundles which can then be loaded on demand or in parallel. This can significantly improve initial load time.
5. **Tree Shaking:** A form of dead code elimination. It removes unused exports from your bundles, reducing their size.
6. **Development Server (`webpack-dev-server`):** Provides a live reloading development server.
7. **Hot Module Replacement (HMR):** Exchanges, adds, or removes modules while an application is running, without a full reload, improving development experience.

- **Basic Configuration (`webpack.config.js`):**

```
// webpack.config.js (simplified example)
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  mode: 'development', // or 'production'
  entry: './src/index.js', // The main entry point of your application
  output: {
    filename: 'bundle.js', // The name of the output bundle
    path: path.resolve(__dirname, 'dist'), // The output directory
    clean: true, // Clean the output directory before each build
  },
  module: {
    rules: [
      {
        test: /\.js$/, // Apply this rule to .js files
        exclude: /node_modules/, // Don't apply to files in node_modules
        use: {
          loader: 'babel-loader', // Use babel-loader for these files
        },
      },
      {
        test: /\.css$/, // Apply this rule to .css files
        use: ['style-loader', 'css-loader'], // Process with css-loader
then style-loader
      },
      {
        test: /\.(png|svg|jpg|jpeg|gif)$/i,
        type: 'asset/resource', // For image assets
      },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
```

```
      template: './src/index.html', // Use this HTML file as a template
    }),
  ],
  devServer: {
    static: './dist', // Serve content from the dist directory
    hot: true, // Enable Hot Module Replacement
  },
};
```

- **Usage:** `create-react-app` and Vite handle webpack (or similar bundler like Rollup for Vite) configuration internally, abstracting much of this complexity away from the developer for typical projects. However, understanding webpack is beneficial for customizing build processes or working on more complex setups.
