Module 9 - Introduction to React.js

THEORY EXERCISE

Question 1: What is React.js? How is it different from other JavaScript frameworks and libraries?

- React.js is a **JavaScript library** for building user interfaces (UI).
- Made by Meta (Facebook), it helps you make parts of a web page (called *components*) that you can reuse.
- It's mostly used to build **interactive** web apps, especially single-page applications (where things update without reloading the page).
- **Library vs Framework**: React is often called a library (not a full framework) because it focuses just on the UI. You choose other libraries for routing, state management, etc.
- **Flexibility**: Because React is unopinionated, you have freedom to pick your tools (e.g. React Router, Redux) rather than being forced into a "React way" for everything.
- **Data Binding**: React uses **one-way (unidirectional) data flow**, which makes data changes more predictable and easier to debug, compared to frameworks like Angular that use two-way binding.
- **Learning Curve**: Because React is just a UI library, its core API surface is relatively smaller compared to full frameworks like Angular, making it easier to learn in some respects.

Question 2: Explain the core principles of React such as the virtual DOM and component-based architecture.

- **Component-Based**: You break your UI into small pieces (components). Each component can manage its own data (state) and logic, and you can reuse them.
- **Virtual DOM**: React keeps a lightweight copy of the web page in memory (virtual DOM). When something changes, it figures out the smallest change needed and updates only that part in the real page.
- **Declarative UI**: You write *what you want the UI to look like* for a given state, and React ensures the real page matches that.
- **One-way Data Flow**: Data flows from parent components to child components, making things more predictable.

Question 3: What are the advantages of using React.js in web development?

Easy to learn and use

Reusable component

Good performance

Fast rendering and virtual DOM

Question 4: What is JSX in React.js? Why is it used?

- **JSX** (JavaScript XML) is a syntax extension for JavaScript that looks like HTML but lives in JavaScript.
- It's used in React to *describe the UI structure* in a more readable, declarative way.

- While React doesn’t require JSX, most developers use it because it makes writing and visualizing components easier.

Question 5: How is JSX different from regular JavaScript? Can you write JavaScript inside JSX?

- JSX is **not just JavaScript** — it's a special syntax that gets transformed (by tools like Babel) into React.createElement(...) calls.
- Yes, you can write JavaScript **inside JSX**, but only *expressions*, not statements.
- For example, you can put variables, arithmetic (2 + 2), function calls, or object properties inside JSX using curly braces.

Question 6: Discuss the Importance of using curly braces {} in JSX expressions

- Curly braces in JSX are like a **window into JavaScript** — whatever is inside them is treated as JavaScript, not a string.
- Using them, you can embed dynamic values (variables), run functions, or evaluate expressions right in your markup.
- When you want to pass a JS object (for example for style), you sometimes use **double curly braces**: the outer ones open JSX JS-evaluation, and the inner ones define the object.

Question 7: What are components in React? Explain the difference between functional components and class components.

- **Components** in React are reusable building blocks for the UI. Each component encapsulates its own structure (JSX), logic, and optionally state.
- **Functional components**: These are plain JavaScript functions that take props as input and return a React element (JSX).
- **Class components**: These are ES6 classes that extend React. Component. They must define a render() method which returns JSX.
- **Key differences**:
    - **State & lifecycle**: Class components have built-in support for state and lifecycle methods.
    - **Simplicity**: Functional components are simpler, especially with hooks now allowing them to use state and effects.
    - **Props access**: In function components, props are passed as function arguments. In class components, you access them via this.props.

Question 8: How do you pass data to a component using props?

- You pass data from a parent component to a child component by giving **props** as attributes: e.g., <Greeting name="Amit" />.
- In the child component:
    - For a **functional component**, you receive the props in the function argument: function Greeting(props) { … }.

- For a **class component**, you access props with this.props: this.props.name (inside methods like render).

Question 9: What is the role of render () in class components?

- In a **class component**, the render () method is **mandatory**. React calls render () to know what UI (JSX) to display.
- Render () returns the React elements (JSX) that describe what should be shown on the screen.
- Every time the component's state or props change, React may call render () again to update the UI accordingly.

Question 10: What are props in React.js? How are props different from state?

- **Props** (short for "properties") are data passed from a parent component to a child component.
- Props are **read-only** inside the child component — the child should not modify them.
- **State**, by contrast, is data that is **local** to a component and can change over time.
- State is **mutable** — you can update it (in class components) using this.setState() or (in functional components) using hooks.
- Summary comparison:

Feature	Props	State
Source	Passed from parent	Managed inside component
Mutability	Immutable	Mutable (via setState)
Purpose	Configuration/data input	Tracks internal component data

Question 11: Explain the concept of state in React and how it is used to manage component data.

- **State** is a special object in React components that holds data that may change over time.
- In class components, you typically initialize state in the constructor:
- constructor(props) {
- super(props);
- this. State = { count: 0 };
- }
- When the state changes, React **re-renders** the component (or at least the part that depends on the state) so the UI updates to reflect the new state.

- State is used for dynamic data: user inputs, counters, toggles, API response data — anything that changes in response to interaction or events.

Question 12: Why is this. Set State () used in class components, and how does it work?

- this.set State () is the method provided by React to **update the component's state** in class components.
- When you call setState(), React **schedules an update** to the state (it doesn't always happen immediately) and then re-renders the component.
- The new state you pass to setState() is **merged** with the existing state (i.e., you don't have to replace the full state object).
- Because updates are **asynchronous**, relying on this.state immediately after calling setState() might not give you the updated value.

Question 12: How are events handled in React compared to vanilla JavaScript? Explain the concept of synthetic events.

- In React, you don't directly use addEventListener on DOM nodes. Instead, you attach event handlers via **JSX attributes**, e.g., <button onClick={…}>.
- React uses a **Synthetic Event** system: it wraps the native browser events into its own "SyntheticEvent" object.

- The synthetic event system ensures that events behave consistently across different browsers.
- Also, React “pools” synthetic events (for performance), so the event object may be reused.

Question 13: What are some common event handlers in React.js? Provide examples of onClick, onChange , and onSubmit.

Some of the most commonly used event handlers in React are:

- **onClick** — triggered when a user clicks a button (or other clickable element).
- function MyButton() {
- function handleClick() {
- alert("Button clicked!");
- }
- return <button onClick={handleClick}>Click me</button>;
- }
- **onChange** — used for form inputs; fires when the value of an input changes.
- function MyInput() {
- function handleChange(event) {
- console.log(event.target.value);

- }
- return <input type="text" onChange={handleChange} />;
- }
- **onSubmit** — used on HTML <form> elements; triggers when the form is submitted.
- class MyForm extends React.Component {
- handleSubmit = (event) => {
- event.preventDefault();
- console.log("Form submitted!");
- };
- render() {
- return (
- <form onSubmit={this.handleSubmit}>
- <button type="submit">Submit</button>
- </form>
-);
- }
- }

Question 14: Why do you need to bind event handlers in class components?

- In ES6 classes, methods are **not bound by default**. If you pass a class method directly as an event handler, this inside that method will be undefined (or not what you expect) when it's invoked.
- Binding ensures that inside your handler, this correctly refers to the component instance, so you can access this.state, this.props, or call this.setState().
- Common ways to bind:
    - **Constructor binding**:
    - constructor(props) {
    - super(props);
    - this.handleClick = this.handleClick.bind(this);
    - }
    - **Class fields / arrow functions**: Define the handler as an arrow function so it uses the lexical this:
    - handleClick = () => {
    - console.log(this);
    - }

Question 15: What is conditional rendering in React? How can you conditionally render elements in a React component?

- **Conditional rendering** means showing different UI (or no UI) depending on some condition (state, props, etc.).

- In React, since JSX lets you embed JavaScript, you can use standard JS logic (like if statements) or expressions inside your component to decide what to return/render.
- You can also conditionally render “nothing” by returning null.

Question 16: Explain how if-else, ternary operators, and && (logical AND) are used in JSX for conditional rendering.

Here are the three common patterns:

1. **if-else**
    - You can use a normal if … else in your component (outside of JSX) to choose which JSX to return.
    - Example:
    - function Greeting({ isLoggedIn }) {
    - if (isLoggedIn) {
    - return <h1>Welcome back!</h1>;
    - } else {
    - return <h1>Please log in.</h1>;
    - }
    - }
2. **Ternary operator (condition ? expr1 : expr2)**
    - This is a shorthand for if-else and is very common inside JSX.
    - Example:

- return (
-
- {isLoggedIn
- ? <h1>Welcome back!</h1>
- : <h1>Please log in.</h1>}
-
-);

3. **Logical AND (&&)**
    - Use condition && expression when you want to render something only if the condition is true (and render nothing if it's false).
    - Example:
    - return (
    -
    - {hasNotifications && You have new notifications! }
    -
    -);

Question 17: How do you render a list of items in React? Why is it important to use keys when rendering lists?

- To render a list in React, you typically use JavaScript's map() inside JSX. For example:

- function NumberList({ numbers }) {
- return (
-
- {numbers.map((n) => (
- <li key={n.id}>{n.value}
-))}
-
-);
- }
- **Keys** are used to give each list item a stable, unique identity. React uses these keys when reconciling (updating) the list, so it knows which items changed, were added, or removed.
- Without proper keys, React's updates can be inefficient or even buggy, because it may not correctly match items between renders.

Question 18: What are keys in React, and what happens if you do not provide a unique key?

- **Keys** are special props (usually a string or number) you pass to elements in a list so React can uniquely identify each item.
- If you don't provide a unique key (or you use a non-stable key, like the array index), React may reuse or reorder DOM elements incorrectly, which can lead to performance issues or bugs (e.g., wrong component state being preserved).

- React will also warn you in the console: “Each child in a list should have a unique ‘key’ prop.”

Question 19: How do you handle forms in React? Explain the concept of controlled components.

- In React, **forms** are typically handled by keeping the form data (like input values) in the component’s **state**, and updating that state as the user types or makes changes.
- A **controlled component** is a form element (like <input>, <textarea>, or <select>) whose value is controlled by React: you set its value prop to a state variable, and you handle changes via an onChange event to update that state.
- Because React state is the “single source of truth,” controlled components make the form data predictable, easy to validate in real time, and easier to manipulate (for example, enabling or disabling submit buttons).

Example (functional component):

```
function MyForm() {
  const [name, setName] = useState("");

  function handleChange(event) {
    setName(event.target.value);
  }
```

```
  function handleSubmit(event) {
    event.preventDefault();
    alert("Submitted name: " + name);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" value={name} onChange={handleChange} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}
```

Question 20: What is the difference between controlled and uncontrolled components in React?

- **Controlled components**:
    - The input's value is stored in React **state**, not in the DOM.

    - On each change (e.g. a keystroke), an event handler (like onChange) updates the React state.
    - This gives you great control: you can do **real-time validation**, conditionally enable/disable fields, or transform user input as they type.
    - The component is predictable because state is the "single source of truth."
- **Uncontrolled components**:
    - The input's value is handled by the **DOM itself**, not by React.
    - To read the value, you use a **ref** (useRef or createRef) to access the DOM node when needed (for example, on form submit).
    - This can be simpler to implement for basic forms, or when you don't need to validate on every keystroke.
    - But it gives you less control and is less "React-like" for managing form data.

Que-21 What are lifecycle methods in React class components – and the phases of a component's lifecycle?

- **Lifecycle methods** are special methods in React class components that let you run code at specific times during a component's "life" — for example, when it is created, updated, or removed.
- React components go through **three main phases**:

- **Mounting** — when the component is being inserted into the DOM for the first time
- constructor(props): called first; used to set up initial state and bind methods.
- static getDerivedStateFromProps(props, state): called just before rendering; lets you update state based on props.
- render(): the only required method — returns the JSX to render.
- componentDidMount(): called after the component has been rendered to the DOM; useful for side-effects like data fetching.
- **Updating** — when the component's props or state change, causing a re-render.
- static getDerivedStateFromProps(props, state): again called before render, when props or state change.
- shouldComponentUpdate(nextProps, nextState): lets you decide whether React should re-render or skip the update (for performance).
- render(): re-render UI based on new state/props.
- getSnapshotBeforeUpdate(prevProps, prevState): called just before the changes from render are applied to the DOM; you can capture info (like scroll position).
- componentDidUpdate(prevProps, prevState, snapshot): called after the component has updated; good for side-effects that depend on previous state/props.

- **Unmounting** — when the component is being removed from the DOM.
- componentWillUnmount(): called right before the component is removed. Use it to clean up resources (e.g. cancel timers, remove event listeners).

Question 22: Explain the purpose of componentDidMount(), componentDidUpdate(), and componentWillUnmount().

1. **componentDidMount()**
    - This method runs **immediately after** the component is first rendered and mounted to the DOM.
    - Use it for initialization tasks that need the DOM to be ready: e.g., fetching data from APIs, setting up subscriptions, or interacting with third-party libraries.
    - If you call this.setState() inside it, React will re-render the component — but since the update happens before the browser paints, the user doesn't see an intermediate state.
2. **componentDidUpdate(prevProps, prevState, snapshot)**
    - This runs **right after** the component's updates (i.e., after render) whenever its props or state change.
    - It's useful for handling side-effects based on the updated data: for example, making network requests **only when certain props or state change**.

- If you've used getSnapshotBeforeUpdate(), the “snapshot” value it returns is passed to componentDidUpdate() as a third argument, letting you act using that information (for example, scroll position).
- **Be careful**: if you call setState() inside componentDidUpdate(), you need to wrap it in a condition (checking prevProps or prevState) — otherwise you risk an **infinite loop**.

3. **componentWillUnmount()**
    - This method is called **just before** the component is removed (unmounted) from the DOM.
    - You use it to **clean up** any side-effects started in componentDidMount() (or elsewhere): for example, clearing timers, invalidating network requests, removing event listeners, or unsubscribing from services.
    - You should **not** call setState() here, because once unmounted the component won’t be re-rendered.

22. What are React Hooks? How do useState() and useEffect() work in functional components?

What are React Hooks?

- Hooks are special functions introduced in **React 16.8** that let you “hook into” React features (like state and lifecycle) from **functional components**, without needing class-based components.

- They make it possible to share and reuse logic between components through **custom hooks**.
- There are some rules: you can only call Hooks at the top level (not inside loops or conditionals) and only from React function components or custom hooks.

useState() Hook:

- useState lets a functional component have **local state**.
- Syntax:
- const [state, setState] = useState(initialValue);
    - state is the current value.
    - setState is a function to update that state.
- When you call setState(...), React schedules a **re-render** of that component with the updated state.
- You can use multiple useState calls in a single component to manage independent pieces of state.

useEffect() Hook:

- useEffect is for **side effects** — operations that interact with external systems, like fetching data, subscriptions, or modifying the DOM.
- Its signature is:
- useEffect(() => {
- // effect logic
-

- return () => {
- // cleanup logic (optional)
- };
- }, [dependencies]);
- The first argument is a function that contains the effect. The second argument is an **array of dependencies**.
    - If the dependency array is empty ([]), the effect runs only **once**, after the first render (component mount).
    - If you provide dependencies, the effect will re-run whenever any of them change.
- The effect function can optionally **return a cleanup function**, which React runs when the component unmounts or before re-running the effect (if dependencies change).
- useEffect basically replaces class lifecycle methods like componentDidMount, componentDidUpdate, and

23. What problems did hooks solve in React development? Why are hooks considered an important addition to React?

Hooks solved several key limitations and pain points in React:

1. **Complexity of Class Components**
    - Before Hooks, to use state or lifecycle features, you had to write class components, which involve boilerplate (constructor, this, binding) and can be confusing. Hooks let

you use state and side effects in *functions*, making code simpler and more straightforward.

- They avoid the complexity of this keyword in classes.

2. **Reusability of Stateful Logic**
    - With classes, sharing logic between components often required patterns like Higher-Order Components (HOCs) or render props, which can make code nested and hard to follow. nickhuemmer.com
    - Hooks allow for **custom hooks**: you can extract and reuse stateful logic (e.g., data fetching, form management) easily.
3. **Separation of Concerns**
    - Class lifecycle methods often mix unrelated logic (e.g., data fetching, subscription cleanup, state updates) all in one method like componentDidMount or componentDidUpdate. Hooks let you separate concerns more cleanly: each useEffect can handle a specific side effect.
    - This leads to more modular, testable code.
4. **Better Performance and Readability**
    - Hooks like useMemo and useCallback help with memoization and optimizing performance. (More on that later.)
    - Functional components with Hooks are often more readable and leaner than class-based ones.
5. **Functional Programming Style**

- Hooks encourage a more functional programming approach, where state and behavior are co-located and side effect logic is explicit. This aligns with modern JS patterns.

24. What is useReducer? How do we use it in a React app?What is useReducer?

- useReducer is a Hook that provides a **reducer-based** way to manage state, similar to Redux but scoped to a component.
- It's especially useful when your state logic is **complex**, involves multiple sub-values, or when the next state depends on the previous one.
- It returns [state, dispatch]:
  - state: current state
  - dispatch: a function to send actions to the reducer

How to use it:

1. Define a **reducer function**:
2. function reducer(state, action) {
3. switch (action.type) {
4. case 'increment':
5. return { count: state.count + 1 };
6. case 'decrement':
7. return { count: state.count - 1 };

8. default:
9. return state;
10. }
11. }
12. Use the hook in your component:
13. const [state, dispatch] = useReducer(reducer, { count: 0 });
    - The second argument is the **initial state**.
14. Dispatch actions to change state:
15. <button onClick={() => dispatch({ type: 'increment' })}>+</button>
16. <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
17. Count: {state.count}
    - dispatch passes the action to the reducer, which computes the new state.
18. (Optional) *Lazy initialization*: useReducer also takes a third argument, init, which is a function for **lazy initialization** of state.

Why use it vs useState:

- For simple state (like a single number or boolean), useState is enough.
- But for more structured state (objects, nested data) or when multiple related pieces of state need coordinated updates, useReducer provides a cleaner and more predictable pattern.

- Also, because the reducer is a pure function, the logic can be tested independently of React components.

25. What is the purpose of useCallback & useMemo Hooks?useCallback

- It returns a **memoized version of a function** — i.e., it caches the function definition so that it doesn't get recreated on every render.
- Signature:
- const memoizedFn = useCallback(() => {
- // callback logic
- }, [dependencies]);
- Useful when:
    - You're passing a callback to a child component (especially a memoized child) and you don't want it to re-render because the function reference changes each render.
    - You use the function inside a useEffect's dependency array, and you don't want the effect to run just because the function was recreated.

useMemo

- useMemo returns a **memoized value** — i.e., the result of a function is cached and recomputed only when its dependencies change.

- Signature:
- const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
- Useful when:
    - You have **expensive calculations** (loops, heavy computation) that you don't want to run every render.
    - You declare an object or array inside a component that is used in dependency arrays (e.g., useEffect) or passed to children — by memoizing it, you ensure **referential equality** and avoid unnecessary rerenders.

26. What's the Difference between useCallback & useMemo Hooks?

Here are the main differences:

Hook	What is Memoized	When to Use
useCallback	The **function definition** itself	When you want a stable function reference (e.g., to pass to child components, or as a dependency in useEffect)
useMemo	The **result/value** of a computation	When you have an expensive computation or want to memoize objects/arrays to maintain referential equality

27. What is useRef? How does it work in a React app?What is useRef?

- useRef is a Hook that returns a **mutable ref object** whose .current property persists for the *entire lifetime* of the component.
- Unlike state, updating a .current value of a ref **does not trigger a re-render**.

How useRef works / how to use it:

- Basic usage:
- const myRef = useRef(initialValue);
- Common use-cases:
    1. **DOM reference**: You can attach ref to a JSX element to imperatively access that DOM node:
    2. <input ref={myRef} />
    3. // Later
    4. myRef.current.focus();
    5. **Storing mutable values**: Use useRef to keep any mutable value (e.g., timers, previous state) that you don't want to cause re-renders.
    6. **Persisting values across renders**: Because .current persists, you can use useRef to store things like previous prop or state values, but without the overhead of state updates.

Why use it:

- When you need **direct access to DOM elements** (e.g., focus, measure size) in functional components.
- When you want to store **mutable data** that doesn't cause re-render (unlike state).
- For use in timers, subscriptions, or when implementing certain custom hooks.

Question 28: What is React Router? How does it handle routing in single-page applications (SPAs)?

What is React Router?

- **React Router** is a popular library (typically react-router-dom for web) that adds routing capabilities to React apps.
- Since React by itself doesn't have built-in routing, React Router provides a declarative way to define "routes" (URL paths) and map them to React components.
- It enables **client-side routing**, meaning navigation is handled inside the browser (without full-page reloads), giving a smooth, app-like experience.

How React Router handles routing in SPAs:

1. **URL Synchronization**: React Router listens to browser URL changes (using the History API) and keeps the UI in sync with the current URL.
2. **Matching Routes**: You define <Route>s which specify a path and which component to render when the URL matches that path.

3. **Navigation**: Components like <Link> let users navigate between routes without triggering a full page refresh — they change the URL and React Router handles rendering the right component.
4. **History Management**: Using the browser's native History API (e.g., pushState, popstate), the router changes the URL and React reacts to these changes, rendering components accordingly.
5. **Nested Routes**: React Router supports nested routing, so you can build complex UI hierarchies (routes within routes).
6. **Efficient Rendering**: Since routing is on the client, only the required components re-render — there's no full page reload, which is much faster and feels more seamless.

Question 29: Difference between BrowserRouter, Route, Link, and Switch

Here's a breakdown of each:

Component	Purpose / Role in React Router
BrowserRouter	This is a **router implementation** that uses the HTML5 History API to keep your UI in sync with the URL. It wraps your entire React app (or at least the part that does routing) so that routing context is available.
Route	Defines a mapping between a URL path and a React component. When the path matches the current URL, React Router renders the associated component. In Route, you can specify props like path and what to render (in v6, via element).

Component	Purpose / Role in React Router
Link	This is used to **navigate** between different routes without reloading the page. It renders as an <a> tag under the hood but prevents full page reload by handling navigation in JavaScript. You use the to prop to specify where you want to navigate.
Switch	(React Router v5) Switch renders **only the first Route child** that matches the URL. This is useful when you have multiple possible routes but want to render just one. In v6, Switch is replaced by Routes (which has a somewhat different behavior / matching logic).

Putting it all together: Example

Here's a very simple example (React Router v5 style) to illustrate how these work together:

```
import React from 'react';

import { BrowserRouter, Switch, Route, Link } from 'react-router-dom';


function Home() {

  return <h1>Home Page</h1>;

}
```

```
function About() {
  return <h1>About Page</h1>;
}

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link>{" | "}
        <Link to="/about">About</Link>
      </nav>

      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
      </Switch>
    </BrowserRouter );
}
```

Question 30: What do you mean by RESTful web services?

- **REST** stands for **Representational State Transfer** — it's an architectural style for designing networked applications.
- **RESTful web services** are web services (APIs) that conform to the principles of REST.
- Key constraints / principles of RESTful services include:
    1. **Resource-based URIs**: Each resource (like a “user”, “post”) is identified by a URI.
    2. **Statelessness**: Every request from client to server must contain all information needed — the server does not store client-specific session.
    3. **Uniform interface**: The API uses standard HTTP methods (GET, POST, PUT, DELETE) to perform CRUD operations.
    4. **Self-descriptive messages**: The request and response should contain enough information (e.g., via headers, media types) so that the client and server can understand how to process them.
    5. **(Optional) Hypermedia as the engine of application state (HATEOAS)**: In a truly hypermedia-driven REST API, the responses include links to other related resources.
- Benefits of RESTful web services: lightweight, scalable, cacheable, platform/language agnostic, easy to understand and use.

Question 31: What is JSON Server? How do we use it in React?What is JSON Server?

- JSON Server is a Node.js tool that lets you **quickly spin up a mock REST API** using just a JSON file (usually db.json).
- It supports standard RESTful operations out-of-the-box: GET, POST, PUT/PATCH, DELETE.
- It’s primarily used for development, prototyping, or testing; it's *not* recommended for production usage.

- **How to use JSON Server in a React project:**

1. **Install JSON Server**:
2. npm install -g json-server

(or use npx json-server … if you don’t want to install globally)

3. **Set up a db.json file**: Create a JSON file that represents your “database” — e.g.:
4. {
5. "posts": [
6. { "id": 1, "title": "Hello World" },
7. { "id": 2, "title": "JSON Server Rocks" }
8.],
9. "users": [
10. { "id": 1, "name": "Alice" },
11. { "id": 2, "name": "Bob" }
12.]
13. }

14. **Run JSON Server**:

15. json-server --watch db.json --port 5000

This will start a REST API server at http://localhost:5000 that exposes endpoints like /posts and /users.

16. **Use it from React**: In your React app, you can make fetch or axios requests to this mock server just like any real API. (See next question.)

Question 32: How do you fetch data from a JSON-Server API in React? Explain the role of fetch() or axios() in making API requests.

- **Fetching data from JSON Server:**
    - Since JSON Server exposes REST endpoints, you can call them from React using standard HTTP request libraries / APIs.
    - For example, using **fetch()**:
    - useEffect(() => {
    - fetch('http://localhost:5000/posts')
    - .then(response => {
    - if (!response.ok) {
    - throw new Error('Network response was not ok');
    - }
    - return response.json();
    - })

```
    .then(data => setPosts(data))
    .catch(error => {
      console.error("Fetching error:", error);
      // handle error state
    });
}, []);
```

Or using **axios**:

```
import axios from 'axios';

useEffect(() => {
  axios.get('http://localhost:5000/posts')
    .then(res => setPosts(res.data))
    .catch(err => {
      console.error("Axios fetch error:", err);
      // handle error
    });
}, []);
```

- You can also do POST / PUT / DELETE using similar fetch or axios calls, because JSON Server supports CRUD.

- **Role of fetch() / axios() in making API requests:**
  - These are *HTTP client tools* (in-browser or Node-side) that let your React application talk to APIs.

- fetch() is a built-in browser API for making network requests; it returns a promise and is quite low-level (you need to parse JSON, check status, etc.).
- axios is a popular third-party library that simplifies HTTP requests: it has built-in JSON parsing, better error handling, request cancellation, interceptors, etc.
- Both are used to **send requests** (GET, POST, DELETE ...) to the server and **receive responses**, which you then integrate into your React state and UI.

Question 33: What is Firebase? What features does Firebase offer?

- **What is Firebase?**
  - Firebase is a Backend-as-a-Service (BaaS) platform by Google. It provides a set of tools and services to help developers build web and mobile applications without managing all backend infrastructure.
- **Key Firebase Features:**

1. **Realtime Database**: A NoSQL database where data is synchronized in real time across clients.

2. **Cloud Firestore**: A modern, scalable NoSQL document database. It supports real-time listeners, offline support, and complex queries.

3. **Authentication**: Firebase provides easy-to-use auth (sign-up / sign-in) using email/password, or providers like Google, Facebook, GitHub, etc.

4. **Hosting**: Host static web apps (HTML, JS, CSS) with SSL, custom domains, and global CDN.

5. **Cloud Functions**: Serverless functions that run in response to events (database writes, HTTP triggers, authentication, etc.).

6. **Cloud Storage**: For storing user-generated content like images, videos, files.

7. **Analytics / Crashlytics / Performance Monitoring**: Built-in tools to understand app usage, monitor crashes, track performance.

8. **Security Rules**: Firebase lets you define security rules to control who can read/write data in the database or storage.

9. **Offline Support**: For databases like Firestore and Realtime DB, clients can keep working even if offline; data synchronizes once connection restores.

Question 34 : Discuss the importance of handling errors and loading states when working with APIs in React

Handling **errors** and **loading states** is crucial in any React app that communicates with APIs, for several reasons:

1. **User Experience (UX)**
    - Showing a **loading indicator** (spinner, skeleton UI) while data is being fetched avoids a blank screen and gives feedback that something is happening.
    - If an API request fails, gracefully showing **error messages** (e.g., “Failed to fetch data. Please try again.”) helps users understand what went wrong.

2. **Robustness and Reliability**
    - Network requests can fail (server down, connectivity issues, timeout). Without error handling, your app might crash or behave unpredictably.
    - Handling different error cases (4xx, 5xx, network errors) ensures your UI doesn’t break and can recover or retry.
3. **State Management**
    - You'll often maintain local state for data, isLoading, and error in a component (or via a hook / context / reducer). This helps in cleanly representing the 3 states: “not yet fetched”, “loading”, “error / success”.
    - Example:
    - const [data, setData] = useState(null);
    - const [isLoading, setIsLoading] = useState(false);
    - const [error, setError] = useState(null);
    -
    - useEffect(() => {
    - setIsLoading(true);
    - fetch(url)
    - .then(res => {
    - if (!res.ok) throw new Error('Server error');
    - return res.json();
    - })

```
    .then(json => {
      setData(json);
      setError(null);
    })
    .catch(err => setError(err.message))
    .finally(() => setIsLoading(false));
  }, [url]);
```

4. **Performance & Debugging**
    - If you track errors, you can log them (locally or to a service) and debug issues proactively.
    - By knowing when and how often load / error states occur, you can optimize (e.g., caching, retries, backoff).
5. **Handling Partial UI**
    - In cases where only a part of your component depends on API data, you can render a placeholder / skeleton for that part, while other parts of the UI are already visible.
    - If error happens, you can fallback to a retry button or fallback UI without breaking the entire application.

Question 35: What is the Context API in React? How is it used to manage global state across multiple components?

The React Context API is a built-in feature that lets you share data ("global state") across many components without passing props down manually at every level.

- It's useful when some data (like user info, theme, language, etc.) needs to be accessible by many components, potentially nested deeply — so you avoid "prop drilling."

  - **Create a context** using `createContext()`.

  const MyContext = React.createContext(defaultValue);

- **Wrap components with a `Provider`** to supply the data.

```
<MyContext.Provider value={someValue}>
  <App />
</MyContext.Provider>
```

Question 36: Explain how createContext() and useContext() are used in React for sharing state.

1. createContext() — creating a Context

- You start by calling createContext(defaultValue) to create a context object. The argument defaultValue is used when a component consumes the context but there's no matching Provider above it in the tree.
- Example:

```
import { createContext } from 'react';
```

const MyContext = createContext(defaultValue);

useContext() — consuming the context value in a component

- In a function (functional) component, you can use the useContext hook to access the context value:

```
import { useContext } from 'react';


const value = useContext(MyContext);
```

- value will be the current context value from the nearest matching Provider above in the component tree.
- If there's no Provider above, value will default to the defaultValue given to createContext().

Question 37:What is Redux, and why is it used in React applications? Explain the core concepts of actions, reducers, and the store.

Redux is an open-source JavaScript library that acts as a predictable container for application state. It can be (and often is) used with React to manage global state across an application.

Store: Redux maintains a single store — one object that holds the entire application state. This "global store" is the central place for state, instead of having many fragmented states in various components.

Actions: To change state in Redux, you don't mutate it directly. Instead you dispatch "actions" — plain JavaScript objects that describe what happened (e.g. user clicked a button, data fetched,

etc.). An action must have a type field (and can include extra data, payload).

Reducers: Reducers are pure functions that take the current state and an action, and return a new state based on that action. They define how state should update in response to actions — but without directly mutating existing state, rather returning a new updated state object.

Question 37: How does Recoil simplify state management in React compared to Redux?

Recoil uses *atoms* (small independent pieces of state) and *selectors* (for derived or computed state), and lets you read or update state using React-style hooks like useRecoilState.

There's **no need** to write actions, action types, reducers, or a big global store — which means much **less boilerplate**.

Components only re-render if they actually use the changed atom — so updates are more **efficient and localized**.

What Redux gives you that Recoil trades off

- Redux offers a **centralized, predictable flow** (dispatch → reducers → store), which can help when you want strict control over how state changes happen.
- For very large or complex state logic, with side-effects or complex business rules, Redux's structure and ecosystem (middleware, dev-tools) is often advantageous.