Basics of React

What is React? (Why use React?)

1. **React** is a JavaScript library for building dynamic and interactive UIs efficiently.
2. It follows a **component-based architecture**, making code reusable and modular.
3. **Virtual DOM** allows faster updates by minimizing actual DOM manipulations.
4. Maintained by **Facebook**, it has strong community support and ecosystem.
5. React can be combined with other libraries (Redux, Router) for full application development.

How React Works (Virtual DOM Concept)

1. React maintains a **Virtual DOM**, a copy of the real DOM in memory.
2. When a change happens, React updates the Virtual DOM first.
3. Then it compares (**diffs**) the new Virtual DOM with the old one.
4. It updates only the changed parts on the real DOM for **better performance**.
5. This process is called **reconciliation**.

JSX (JavaScript XML)

1. JSX is a **syntax extension** for JavaScript that lets you write HTML-like code.
2. JSX gets transpiled into React.createElement calls during build time.
3. You can embed JavaScript expressions inside JSX using {}.
4. JSX must have **one parent element** (use Fragment if needed).
5. Helps create **clearer and cleaner UI code**.

Components (Functional vs Class Components)

1. **Functional Components** are simple JavaScript functions returning JSX.
2. **Class Components** are ES6 classes that extend React.Component and manage lifecycle methods.
3. Functional components can now manage state and side effects using **hooks**.
4. Class components use this.state and this.setState().
5. Modern React recommends using **functional components** with hooks.

Props (Passing Data to Components)

1. **Props** are used to pass data from parent to child components.
2. Props are **immutable** inside the receiving component.
3. You can pass any data type: strings, numbers, functions, arrays, objects.
4. Props improve **reusability** and **customization** of components.
5. Default values can be set using defaultProps.

State (Managing Local Component State)

1. State holds dynamic data that influences component rendering.
2. In functional components, use the useState hook.
3. Updating state causes a **re-render** of the component.
4. State should not be mutated directly — always use setters (setState, setXyz).
5. Good state management improves app predictability and debugging.

Event Handling in React

1. Event handlers are added directly as **props** (e.g., onClick, onChange).
2. React events are **synthetic events** wrapping native events for cross-browser consistency.
3. Use arrow functions or bind handlers for accessing this in class components.
4. Prevent default behavior using e.preventDefault().
5. Events can be passed down and handled inside child components too.

Conditional Rendering

1. Display different UI elements based on conditions (if, ternary, &&).
2. Simple ternary:

```
Ex :-
{loggedIn ? <Dashboard /> : <Login />}
```

3. Use short-circuit && for rendering small pieces:

```
Ex :-
{messages.length > 0 && <MessageList />}
```

4. You can create helper functions for complex conditions.
5. Conditional rendering improves user experience by showing only what's necessary.

Lists and Keys (Rendering Lists)

1. Render lists dynamically using map().
2. Always provide a unique key prop to each list item for efficient rendering.
3. Keys should ideally be unique IDs, not array indexes.
4. Example:-

```
items.map(item => <li key={item.id}>{item.name}</li>)
```

5. Helps React optimize re-rendering and avoid bugs.

Forms and Controlled Components

1. In controlled components, form inputs are tied to state variables.
2. onChange event handlers update the state as the user types.
3. Provides better control over form behavior and validation.
4. Example:

```
<input value={email} onChange={(e) => setEmail(e.target.value)} />
```

5. Helps create forms that behave predictably and validate easily.

Lifting State Up

1. When multiple components need the same state, move it to their **closest common parent**.
2. Parent component controls the state and passes it down via props.
3. Ensures that all components stay in sync.
4. Makes data flow unidirectional and predictable.
5. Helps avoid duplicate state and inconsistencies.

Basic Styling in React

1. You can use normal CSS files and import them into your components.
2. **Inline styles** are written as objects:

```
<div style={{ backgroundColor: 'red' }}></div>
```

3. CSS-in-JS libraries like Styled Components allow scoped styling.
4. CSS Modules provide **locally scoped class names**.
5. Tailwind CSS is another way for utility-first, responsive designs.

Intermediate React

React Router (Navigation between pages)

1. **React Router** is the standard library for routing in React applications.
2. It lets you create **single-page applications (SPAs)** with multiple views.
3. Core components: **`<BrowserRouter>`, `<Routes>`, `<Route>`, `<Link>`**, and **`<Navigate>`.**
4. Example:

```
<BrowserRouter>
  <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
  </Routes>
</BrowserRouter>
```

5. It supports dynamic routes, route parameters **(`/user/:id`)**, and nested routing.

Context API (State management without Redux)

1. **Context API** allows sharing global data without passing props manually through every component.
2. Create a context using **`React.createContext().`**
3. Provide a context using **`<Context.Provider>`** and consume it with **`useContext()`** hook.
4. Useful for theme management, user authentication, language localization.
5. Suitable for small-to-medium state management needs — for larger apps, prefer Redux.

React Hooks

useState

1. **useState** hook is used to create and manage local state in functional components.
2. Syntax: **`const [count, setCount] = useState(0);`**
3. When the state changes, the component re-renders automatically.
4. You can update primitive types (numbers, strings) and complex types (arrays, objects).

useEffect

1. **useEffect** handles side effects like API calls, subscriptions, or manually changing the DOM.
2. Syntax:

```
jsx
CopyEdit
useEffect(() => { fetchData(); }, []);
```

3. Runs after the first render and after every update unless dependency array is given.
4. Cleanup functions inside **`useEffect`** help avoid memory leaks.

useContext

1. **useContext** allows you to consume a context created by **`React.createContext().`**
2. It avoids "prop drilling" (passing props deeply down the component tree).
3. Syntax: **`const user = useContext(UserContext);`**
4. Context changes cause all components using it to re-render.

useRef

1. **useRef** provides a way to directly reference a DOM element or persist a mutable value.
2. Unlike state, changing a ref doesn't cause a re-render.
3. Useful for focusing inputs, managing animations, timers, etc.
4. Example:

```
Jsx
const inputRef = useRef(null);
```

useMemo

1. **useMemo** memoizes a **computed value** between renders to improve performance.
2. Syntax:

```
jsx
const result = useMemo(() => expensiveFunction(num), [num]);
```

3. Avoids recalculating expensive operations unless dependencies change.
4. Overuse can add unnecessary complexity — use wisely!

---

useCallback

1. **useCallback** memoizes a **function** between renders.
2. Useful to prevent unnecessary function re-creation, especially in child components.
3. Syntax:

```
Jsx
const memoizedFn = useCallback(() => doSomething(a, b), [a, b]);
```

4. Works together with **`React.memo`** for optimizing performance.

---

useReducer

1. **useReducer** is an alternative to useState for complex state logic.
2. Similar to Redux: state management via **actions** and **reducers**.
3. Syntax:

```
Jsx
const [state, dispatch] = useReducer(reducerFn, initialState);
```

4. Very useful when dealing with **multi-step forms, wizards, or nested states**.

Custom Hooks (Building your own hooks)

1. Custom Hooks let you extract and reuse logic between components.
2. A custom hook is just a JavaScript function that uses built-in hooks.
3. Naming convention: always start with `use` (like **`useAuth`, `useForm`, `useFetch`**).
4. Improves **code readability**, **reusability**, and **maintainability**.
5. Example:

```
Jsx
function useCounter() {
  const [count, setCount] = useState(0);
  return { count, increment: () => setCount(count + 1) };
}
```

Higher Order Components (HOC)

1. A **HOC** is a function that takes a component and returns a new component with additional props or functionality.
2. Common use cases: authentication guards, logging, role-based UI.
3. Syntax example:

```
jsx
function withLogger(WrappedComponent) {
  return (props) => {
    console.log('Rendering', WrappedComponent.name);
    return <WrappedComponent {...props} />;
  };
}
```

4. HOCs promote **code reuse** but can lead to **wrapper hell** if overused.

Render Props

1. A **Render Prop** is a technique for sharing code using a prop whose value is a function.
2. The component's child is a function that receives data and returns UI.
3. Example:

```
jsx
<DataProvider render={(data) => <Chart data={data} />} />
```

4. Flexible alternative to HOCs but can cause deeply nested code if not managed properly.

Error Boundaries

1. **Error Boundaries** catch JavaScript errors anywhere in the component tree and display fallback UI.
2. Only class components can be error boundaries (yet).
3. Implement **`componentDidCatch(error, info)`** and **`static getDerivedStateFromError()`**.
4. Useful for catching production bugs and preventing app crashes.

Fragments and Strict Mode

Fragments

1. **Fragments** let you group a list of children without adding extra nodes.
2. Syntax:

```
jsx
<>
 <ChildA />
  <ChildB />
</>
```

Strict Mode

1. **StrictMode** is a tool for highlighting potential problems in an application.
2. It activates additional checks and warnings for its children.
3. Wrap parts of your app with **`<React.StrictMode>`** in development mode.

Portals (Rendering outside the DOM tree)

1. **Portals** let you render children into a DOM node that exists outside the parent component hierarchy.
2. Useful for modals, popups, and tooltips.
3. Syntax:

```
jsx
ReactDOM.createPortal(child, document.getElementById('modal-root'));
```

4. Portals help in managing UI layers correctly.

React Forms Libraries (Formik, React Hook Form)

1. **Formik**: Focuses on form state management and validation.
2. **React Hook Form**: Lightweight form management with less boilerplate.
3. Both libraries handle field validation, errors, touched fields, and form submissions.
4. Reduce complexity for large, nested, dynamic forms.

State Management

Redux Basics

1. **Redux** is a predictable state container for JavaScript apps, mostly used with React.
2. Centralizes the app's state in a **single global store**.
3. Data flow follows a **unidirectional pattern**: View → Action → Reducer → Store → View.
4. Works great for complex applications where many components need access to shared state.
5. **Three core concepts**: Actions, Reducers, Store.

Actions

1. **Actions** are plain JavaScript objects describing **what happened**.
2. Must have a `type` property (and optional payload).
3. Example:

```
javascript
{ type: 'ADD_TODO', payload: { text: 'Learn Redux' } }
```

4. Actions are dispatched to the store to trigger a state change.
5. Follow a consistent action naming convention (`FETCH_USER_SUCCESS`, etc.) for maintainability.

Reducers

1. **Reducers** are pure functions that take the current state and an action, then return a new state.
2. They handle different action types via `switch` or `if` statements.

3. Reducers must be **pure** — no API calls or side effects inside them.
4. Example:

```
function counterReducer(state = { count: 0 }, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    default:
      return state;
  }
}
```

5. You can **combine multiple reducers** using **`combineReducers`**.

Store

1. **Store** holds the application state and provides methods to access and update it.
2. Created using **`createStore(reducer).`**
3. Provides methods like **`store.dispatch(action),`** **`store.getState(),`** and **`store.subscribe(listener).`**
4. You usually provide the store to React using **`<Provider store={store}>.`**
5. Only one store per app (mostly).

Dispatch

1. **dispatch** is the method used to send actions to the store.
2. Example:

```
dispatch({ type: 'ADD_TODO', payload: { text: 'Learn Redux' } });
```

3. Dispatch triggers the reducers to process the action.
4. You can also dispatch **async actions** with middlewares like Thunk or Saga.

Connect (for Class Components)

1. **connect()** is a higher-order function from **`react-redux`** that connects class components to the store.
2. It maps state and dispatch to props using **`mapStateToProps`** and **`mapDispatchToProps`**.
3. Example:

```
connect(mapStateToProps, mapDispatchToProps)(MyComponent)
```

4. With modern React (Hooks), `connect` is less common — replaced with **`useSelector`** and **`useDispatch`**.

Redux with Hooks (useSelector, useDispatch)

---

useSelector

1. **useSelector** allows functional components to read values from the Redux store.
2. Syntax:

```
const user = useSelector((state) => state.user);
```

3. Re-renders component only if selected data changes.
4. Avoid selecting the entire state to improve performance.

---

useDispatch

1. **useDispatch** gives access to the store's dispatch method inside functional components.
2. Syntax:

```
const dispatch = useDispatch();
dispatch({ type: 'LOGOUT_USER' });
```

3. Used together with `useSelector` for managing Redux in functional components.
4. Encourages a cleaner and simpler component structure.

Redux Middleware (Thunk, Saga)

1. **Middleware** sits between dispatching an action and the reducer receiving it.
2. **Thunk**: allows action creators to return a function (useful for async operations like API calls).

```
function fetchUser() {
  return (dispatch) => {
    axios.get('/user').then(response => dispatch({ type: 'SET_USER',
payload: response.data }));
  };
}
```

3. **Saga**: uses generator functions (`function*`) to handle more complex async workflows elegantly.
4. Middleware adds flexibility like logging, crash reporting, authentication handling.

Redux Toolkit (RTK) — Modern Redux

Why Redux Toolkit?

1. RTK is the **official, recommended way** to write Redux logic.
2. Solves "boilerplate" problems of vanilla Redux (less code, better patterns).
3. Provides pre-built helpers for store configuration, reducers, and actions.
4. Built-in support for async operations with `createAsyncThunk`.

Setting up Redux Toolkit

1. Install:

   ```
   # npm install @reduxjs/toolkit react-redux
   ```

2. Create a slice using `createSlice()`.
3. Configure store with `configureStore()`.
4. Use `Provider` to wrap the app and pass the store.

---

configureStore

1. `configureStore` automatically sets up the Redux DevTools and adds good defaults.
2. Combines reducers and adds middleware like Redux Thunk by default.
3. Example:

   ```
   const store = configureStore({ reducer: { counter: counterReducer } });
   ```

---

createSlice

1. `createSlice` generates **action creators** and **action types** automatically.
2. You define state, reducers, and actions inside the slice.
3. Example:

```
const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1 },
    decrement: (state) => { state.value -= 1 }
  }
});
```

4. Highly reduces code and makes it more maintainable.

---

createAsyncThunk

1. `createAsyncThunk` simplifies **async operations** (like API requests) in Redux.
2. Automatically generates pending, fulfilled, and rejected action types.
3. Example:

```
const fetchUsers = createAsyncThunk('users/fetch', async () => {
  const response = await axios.get('/users');
  return response.data;
});
```

4. Makes handling API states (loading, success, error) very easy.

RTK Query (for APIs)

1. RTK Query is an advanced API-fetching tool included in Redux Toolkit.
2. Helps **automatically cache, fetch, refetch** data without writing boilerplate API calls.
3. Supports caching, pagination, and optimistic updates out of the box.
4. Reduces the need for **`axios`, `fetch`, `createAsyncThunk`**, and separate slices for data fetching.

Entity Adapter (for normalized state)

1. Redux Toolkit provides **Entity Adapter** utilities to manage collections of data.
2. Automatically provides methods like **`addOne`, `removeOne`, `updateOne`**.
3. Keeps the state **normalized** (like **`{ ids: [], entities: {} }`**), improving performance.
4. Useful for lists of users, posts, products, etc.

Best Practices in Redux Toolkit

1. Keep slice files **small and focused** (one slice per feature).
2. Prefer RTK Query for API integration instead of custom thunks.
3. Normalize nested state with Entity Adapter when needed.
4. Avoid unnecessary selectors inside components — use memoized selectors if required.
5. Always handle loading, error, and success states properly.

API Integration

Fetch API / Axios

1. **Fetch API** is a built-in JavaScript API to make HTTP requests, while **Axios** is an external library that makes it easier and has more features.
2. Fetch example:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data));
```

3. Axios example:

```
axios.get('https://api.example.com/data')
  .then(response => console.log(response.data));
```

4. Axios automatically handles **JSON parsing**, **error handling**, and supports **interceptors**.
5. For larger React projects, **Axios** is preferred for better features and ease of use.

CRUD Operations (Create, Read, Update, Delete)

1. CRUD operations represent basic API interactions:
   - **Create** → POST request
   - **Read** → GET request
   - **Update** → PUT/PATCH request
   - **Delete** → DELETE request
2. Example (Axios):

```
// Create
axios.post('/api/items', { name: 'Item Name' });

// Read
axios.get('/api/items');

// Update
axios.put('/api/items/1', { name: 'Updated Name' });

// Delete
axios.delete('/api/items/1');
```

3. Handle **loading**, **success**, and **error** states when doing API calls.
4. Always wrap API calls inside **`try-catch`** blocks when using **`async-await`**.

Handling Loading, Success, and Error States

1. **Loading state**: Show a spinner or loader while the API request is in progress.
2. **Success state**: Update UI or show a success message when data is fetched or action is completed.
3. **Error state**: Show an error alert if the request fails.
4. Typical pattern:

```
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);
const [data, setData] = useState([]);

const fetchData = async () => {
  setLoading(true);
  try {
    const response = await axios.get('/api/data');
    setData(response.data);
  } catch (err) {
    setError(err.message);
  } finally {
    setLoading(false);
  }
};
```

5. **User experience** improves a lot if loading and error states are properly handled.

Token-based Authentication (Bearer Token)

1. APIs often require authentication using **Bearer Tokens** (JWTs or OAuth tokens).
2. Send tokens in the Authorization header:

```
axios.get('/api/protected', {
  headers: { Authorization: `Bearer ${token}` }
});
```

3. Tokens are usually stored in **localStorage** or **cookies** after login.
4. You can create an **Axios instance** to automatically attach tokens to every request:

```
const api = axios.create({
  baseURL: 'https://api.example.com',
  headers: {
    Authorization: `Bearer ${localStorage.getItem('token')}`
  }
});
```

5. Always **refresh tokens** securely if they expire (use Refresh Tokens flow).