React intro.

- React is a javascript library.
- JS is a Component - Based front-end library.
- JS is a model-view Controller architecture.
- The virtual Dom is a Concept used by react.

What is dom?

- Document-object-model is a representation of the structure of web pages.
- Virtual Copy. of real dom.
- first memory is update to virtual dom.
- Diffing: when virtual dom is Compare with real dom.

Development Environment.

- Thunder Client extension = postman.
- ES7 - snippets.
- Bracket pair Colorizer.
- Auto-rename tag.
- Prettier.

- Node js.
- Create react App

Jsx:
- we can write html using jsx in react application.
- install react router dom to use routes to move to next page.
- { } → jsx element
- class is replaced with classname.
- for is replaced with htmlFor.
- tabIndex.
- Only one tag is returned.

Components:
- Reusable Components, that contain piece of code.

function vs class:
- Before hooks: (Before 16.8 version)
  - function didnot manage state while class did (this.state)
  - function didnot have life cycle while class did (ComponentDidMount)
  - function is used for simple ui. While class is used for complex logics.

- After Hooks (version 16.8 to 18.2)
  - function use use-state while class use this.state for state management.
  - function use use effect
  - function after hooks is more concise while class is more verbose.
  - HOCs - Higher-order Components.
    - return a component with in a component.

Rendering:
- virtual dom is created by calling the render method or by return statement.
- virtual dom to real-dom.
  - Called reaching / reconciliation

Conditional rendering & CSS:
- Conditional rendering (javascript)
  - if statement
  - ternary operator
  - logical operator (&&)
  - Switch statement

- Rendering lists.
  - using javascript (map())

Advanced Concepts / intermediate Concepts.

- Intermediate Concepts:

Component life cycle:

- it refers to the sequence of events like created or rendering into the dom to update or delete or removed from the dom.

Life cycle Method in class Components:

The life cycle can be broken into three main phases:

1) Mounting: Component is created or inserting into dom.
2) Updating: Component's state or prop change and re-render.
3) Unmounting: Component is removed from dom.

Mounting:

- Constructor(): Initialized the state/Component
- getDerivedStateFromProps(): Sync state with Prop
- render(): renders the Component.
- componentDidMount(): runs after the Component has been mounted.

Updating:

- getDerivedstateFromProps(): sync Prop with state during update.
- shouldComponentUpdate(): Determine Component re-renders or not.
- render(): re-renders JSX whenever state or Prop change.
- getSnapshotBeforeUpdate(): Capture prev dom related data.
- componentDidUpdate(): runs after the Component has been updated.

Unmounting:

- componentWillUnmount(): runs just Before the Component is unmounted & destroy.

useEffect hook:

- It allows you to perform side effects such as data fetching or manually changing the dom. It acts as combination of life cycle.
- useEffect runs after every render by default.

State Management:

1. useState(): it allow to add state to function.
   - It return two elements a current state & a function to update current state.
2. useReducer(): it takes a reducer function & a initial state
   - It return a current state and a dispatch function.

Event handling:

1). Event Naming Convension.
   CamelCase / {handleClick}.

2). Passing function as event handlers.
   alert("clicked me");

3). Inline Event handler.
   with in JSX element.

4). Event objects.
   handleClick = (event) => {.

5). Preventing default Behaviour.
   event.preventDefault();
example: form submit will refresh the page, so prevent it we will use it., So we can use event Customly.

6). Binding Event handlers.
- Binding in Constructor: (Private)
  this.handleClick = this.handleClick.bind
- Binding using class field. (Public) (this);
  handleClick = () => {

7). Multiple Event handle.
   onMouseEnter / onMouseLeave.

Passing arguments to Event handler..
handleClick(1, event) / .bind(1, event)

Forms:

- Controlled forms: Uses setState or use state.
  - value & onchange manage form inputs
  - Use where data manipulated continously as the user types
- Uncontrolled forms: uses refs to directly access the dom. (useref)
  - Form data is handle by dom.
  - only need to grab the final value as submission.

- Advanced Concepts:

Context API:

- When to use:
  - Global state Management: A global state which accessed by multiple components at different level. like: User authentication, Theme provider (light / dark)
  - Avoiding prop drilling: Passing props all different layers of components. known as prop drilling.

- Sharing function & Methods: sharing callback's or event handlers.
like: toggle a modal.
- Managing Complex State:
- Performance optimization: avoid unnecessary re-renders.

- Where to use:
- Root Component: wrap your entire application, it will provide the context at all levels.
like: Theme provider, Auth providers.
- Specific Subtree: at a specific level.
- Resuable Components: like button at any level.

- How to use:
- Create the Auth Context.
- Create the provider Context.
- Create/wrap the application.
- Use Context in Component 1 or many.
- Update Main Entry file (index.js)

React Router:

- Setting up:
  - Install react router
    npm install react-router-dom.
  - Set up the router in app.js
    for the application.

Higher-order Components: (Hocs)

- Hoc: hoc can take a component as an argument & return a new component
- Why use Hocs:
  - Code reusability.
  - Separation of Concern: wrapped component should focus on its only logic.
  - Conditional re-rendering
- When use Hocs:
  - Data fetching.
  - Authentication.
  - Form handling
  - Logging & Monitoring.

- How to use Hocs:
  - Create the Hoc.
  - Create the login page which navigate to protected Components.
  - Create the protected wrap Components.
  - Wrap the protected Components.
  - Call wrap protected Components to header.

Error boundaries:
- It is a react Component that allow to handle javascript error in graceful way.
- Implement error boundaries use two lifecycle method.
- getDerivedState from Error(error)
- ComponentDidCatch(error, info)

- How to use error boundaries.
  - Create error boundary Component
  - Create a Component which throw error.
  - Wrap the Buggy Components.

- handling error:
  - Error boundaries
  - Async function (try/catch)
  - Display error message
  - logging error.

```
import * as Sentry from
'@sentry/react';
Sentry.captureException(error);
```

- Modern React features:

Hooks:

- When to use Custom hooks:
  - When multiple Components share similar state logic.
    Then, Create a hooks and use it in different Components.
- Use Context - Define in Context:
  - Like you want user-name on different pages.
- Use Memo:
  - When use useMemo:
    Ex: in e-Commerce site you want to filter products on based of price. Then use

useMemo because, it only runs when filter Criteria changes or product list changes.

- How to use it:

Call the use memo:

import { useMemo, useState } from 'react'

- Create a function.
- Use useState to manage state variables

Const [filters, setFilters] = useState({
Category: '',
PriceRange: '', });

- Implement use Memo:

Const memorized value = useMemo(() =>
{
--- Calculation.
}, [dependencies]);

- Use Call back:

- When use Call back:

Ex: if a user filter a Category Price above 2000 So, a new instance is Created. so if you filter again as brand, file, a new instance is Created again (it happen in useMemo), but, if you use Callback with memo only one instance is Created.

- Callback help in to avoid a function re-renders.

Concurrent Mode / React-window:-

- how react-window is so responsive
  - By virtualizing the list
  - Instead of rendering the entire 100,000 item list. it only renders the item visible in the viewport plus a small buffer above & below.

```
import {fixedSizeList as List} from
                    'react-window';
npm install react-window
npm install react-virtualized-
                    auto-sizer
```

State Management Libraries:

- Redux
  - it is a state management library which can store all app important data into one central store.
  - Why use it:
    - to avoid prop drilling / passing props down down many levels
    - keep state predictable.
    - Makes debugging easier.
    - Example: we login & created a state name 'login state' it store login data globally then we create a 'cart state' to store all add to cart data, know we have add to cart info of x user, if the user is offline and we does not need to call backend every time.
  - Core Concepts:
    - Store: Central place where state is kept.
    - Action: An action object that says what happened.

- Reducer: A pure function that updates the state based on actions.
- Dispatch: Sends an action to reducer.
- Subscribe: lets you listen to state changes (React automatically does this)

- How Redux interacts with Backend.
  - Backend: real source of truth.
  - Redux: Temporary storage / Cache on frontend for fast UI updates.
  - flow: log in → redux update with state instantly → Backend validates credentials in bg.
    add items to Cart → redux update Cart instantly → Backend saves Cart.

- How it works.
  - we have a Counter reducer which can work as reducer.
  - then we have plain redux which can give objects & dispatch to reducer.

- Then we have a store.js which can store the state & then we pass this store into index & use into Main file.

- Plain Redux / Thunk redux
  - Plain redux only pass objects
  - Thunk redux pass functions like API's etc. with the help of 'apply middleware'

- Redux Toolkit (RTK)
  - State management both (sync + async) → plain or Thunk both.
  - less code, less boilerplate.
  - official recommended by redux team.

- Use redux is large app easy to debug when state logic is complex use redux.

- Mobx
  - Another state management library completely different from redux.
  - In redux their is only one single store but in Mobx Multiple stores of class.
  - Mobx is mutable will redux is immutable.
  - In redux we manually update but Mobx automatically handles it.
  - Mobx is easy, very low boilerplate., used in small & medium apps.

- Zustand
  - It is small, fast and Scalable state management libraries
  - Uses a minimal API just a 'Create' function to make store.
  - No reducer, no boilerplate. no actions file
  - Zustand feels like hooks + Context without re-render issues.

- Single store = good for small apps.
Multiple store = Scale for large apps.
Selectors, async, middleware =
full production ready features.
- Devtools: a middleware
that connects your Zustand store
to redux devtools browser
extension.
- install redux devtool extension.
ctrl+shift+j / Redux tab.
- Persist: a middleware that
stores your Zustand store
in local storage., keep
page/state even after page
refresh., Saves in local storage.
ctrl+shift+j / ~~redux~~
application → local storage
↳ api-storage.

- React Query: (Server-Side state management)
- It helps you fetch, Cache
synchronize & update data
from APIs with minimal
code.

- Why react Query over useeffect + fetch:
  - first why useeffect.
    - it's built into react.
    - Good for simple requests
    - works fine when you just need to call fetch().
  - What react Query can do on top of useeffect:
    - with useeffect you have to:
      - Manually manage loading, error state
      - write logic for retries and stale data.
      - handle caching & refetching yourself.
    - react Query:
      - loading, error, retry auto
      - Caches data for instant access.
      - Makes pagination, infinite scroll, sync effortless.

- Portals:
  - Render a react Component outside the main dom hierarchy example model.
    - My real life Scenario I Create a model a edit model when keyboard open the model Css's from bg because it does not used portals.
  - How to use it:

```
{ showPortal &&
    ReactDom.createPortal(
      <div>
        ...
      </div>,
      document.body
    )
}
```

  - Where to use it:
    - use for models.
    - use for toast, multiple buttons which you can show on screen with no stop/ scroll view restriction.

- Fragments:
  - one of the fundamental concepts
    <> , </>
  we need to use this to return multiple calls because react components returns one.

- Refs & forward refs.
  - ref give us a direct handle to a dom elements.
    example: focus a field like user does not type email field and press enter auto focus of email.
  - How to use it:
    inputRef.current.focus();

- Strict Mode
  - Again fundamental development concepts it handles all check automatically.
    <React.StrictMode>

- Profiler API:
  - It helps you to measure performance.
  - It tracks render time, detect un-necessary renders integrate with DevTools.

- Render Prop?
  - Render prop is a pattern where a Component takes a function as a prop & calls it to render UI.

- Suspense / use Transition.
  - Lazy loading -> Suspense.