

Read Notes:

- + Bundlers: Webpack, Vite **Parcel** [Read about this]
 - used in (create-react-app)
 - using this in our code!

Installation (parcel): npm install -D parcel
same as [- save-dev]

Difference between (A) carm & (B) tide in package.json?

- A - automatically updates to newer versions
- B - strict on the versioning

Package-lock: tells the exact version that we use in our app (without carm/tide)

- [Locks the version in production]
- Maintains the integrity (using hash)

node-modules: database for npm.

* Run app using parcel: npx parcel index.html.
Parcel does HMR: Hot module Reload

- ↳ How? File Watcher Algorithm > C++

What does Parcel do more?

- ① Bundling * Also does HMR & hot module replacement
 - ② Minifying code
 - ③ Cleaning code
 - ④ Creating Dev & Production Builds (has superficial build stage)
 - ⑤ Takes care of Image Optimization
 - ⑥ Caching while development. → consistent hashing algo:
 - ⑦ Compression
 - ⑧ Compatible with older browser versions
 - ⑨ HTTPS on dev
 - ⑩ PORT number
- * Tree Shaking - removes unwanted code
- * generate build file = parcel build index.html

* Transitive Dependencies: When 1 library requires another library
↳ then the other one requires another
↳ (should be in package-lock.json) [e.g. A → B ← C ← D]

* browserslist: []

- ↳ for compatibility in browser
- (Ref. browserslist-dev)

↳ build command: npx parcel build index.html

↳ Why do we use JSX? → because normally everything will be mixed up (if we use React.createElement) everywhere.

↳ JSX → JavaScript XML

Study about key reconciliation (in ReactDOM) (why do we need keys?)

* How JSF does sanitization? (for CSS attack)

* Config driven UI: data is shown as per Backend configuration only

* Optional Chaining: This operator returns undefined when an object is accessed which returns null/undefined instead of throwing error

* Virtual DOM: Representation of actual DOM, which uses reconciliation: (exactly like git diff)

* Reconciliation: Algorithm that React uses to differentiate one tree from other & finds out what needs to be updated (therefore key reconciliation is required).

* [After React 16] Read fibre is announced as the total key reconciliation engine, which is responsible for diff (reconciliation)

* Default export vs. Named Export

- export default name;
- export const name;

* Every component in React maintains a 'state':

- ↳ Every time we have to make changes to our local variable, we use 'state'.

* One way & Two way data binding:

- ↳ View to Component Data Binding: When using inputs
- ↳ Component to View data binding when rendering variables

* State variables are used to be in sync with the UI

* useEffect() parameters: callback function & dependency array
Syntax: useEffect(() => {}, []);

* Component Re-renders only two times:

- ① Either state changes
- ② OR props changes

PTO ⇒ Next Page!!

} Read about this

Naive React (Basic Notes)

1) `useEffect`: → is called 'after' initial render by default

> Not passing the array renders the component everytime state/props are changed.

Eg: `useEffect(() => {});`

> Passing single array renders the component only once (first time).

Eg: `useEffect(() => {}, []);`

> Passing array with values renders the component everytime the value is changed.

Eg: `useEffect(() => {}, [searchText]);`

React Router Dom (npm i react-router-dom)

> Create a routing configuration:

① `createBrowserRouter (array: list of paths)`

> Use `RouterProvider`:

`root.render(<RouterProvider router={appRouter}>)`

> `useRouteError` (import { useRouteError } from ...)

Usage: `const error = useRouteError();`
error data → contains error info.

> `Link` tag (import { Link } from 'react-router-dom';)

→ Syntax: `<Link to="/"> About </Link>`
↑ gets converted to `<a>` tag
at the end.

> `Outlet` tag:

→ Helps in changing the components with the help of nested routes.

→ all children of a nested route go into `Outlet`!

Class-based Components

* Creation Syntax:

```
class Profile extends React.Component {  
    render() {  
        //  
    }  
}
```

* Most important part of a class-based component:
↳ render() method

* React gives access to state in class components through 'this' keyword: 'this.state'

Eg:
`this.state = {
 count: 0,
};`

* Create multiple state variables:

↳ `this.state = {
 count1: 0,
 count2: 0,
 count3: 1
};`

React maintains one single object for states.

* For setState in class-based components:

`this.setState({count1: 1});`

* useEffect in class-based components:

↳ `ComponentDidMount() {
 //
}`

↑
Lifecycle Methods

[PTO]

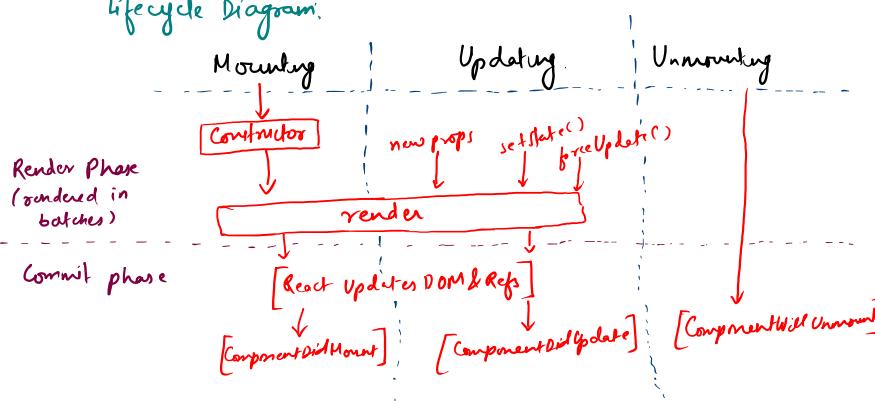
class based components

Lifecycle Methods: render()
componentDidMount()
ComponentDidUpdate()
& so on.

* Read about React Lifecycle Methods diagram [Render & Commit phase]

↳ React does rendering in batches, all the child components render first & then move towards componentDidMount().

lifecycle Diagram:



* ComponentDidMount can be made async but useEffect cannot. Why?

↳ useEffect does not expect the callback function to use a promise.

* why do we use ComponentWillUnmount?

↳ to cleanup unwanted code.

eg: setTimeout/setInterval in
componentDidMount will continue
to run even if we navigate
to another route

Custom Hooks:

- * Modularity helps in:
 - ① Maintainable,
 - ② Readable,
 - ③ Testable code
- Reasons why
we use custom
Hooks

- * Custom Hooks work same as normal functions except the fact that they do not return JSX.
- * Always clean up event listeners in useEffect cleanup
- * Lazy Loading / chunking / splitting / Dynamic Bundling
 - > Implement Lazy Loading:
 - lazy() function from react-library
 - ↳ named export.

Syntax:

↳ const Instantant = lazy(() => import('./components/Instantant'));

* Suspense Component:

- Lazy Loading Components can take a bit of time, so we have suspense component to render it a bit late.
 - [<suspense> <Instantant/> </suspense>]

↳ Suspense takes a prop: fallback (which also renders a component until the suspense awakes)

* [P.T.O.]

★ Lecture 12: Tailwind CSS [Jo dikhta hai wo bikta hai]

→ Using a library gives us consistency

- Example of Libraries:
- Material UI
 - Ant UI
 - Chakra UI

(Read about pros & cons of each)

- Ways of writing CSS:
- Normal CSS
 - SCSS (SASS)
 - Inline CSS
 - ^{Component} Libraries (MUI, Material, etc)
 - styled components

→ Tailwind CSS:

- CSS Framework
- CSS on the go
- reusable
- less bundle size
- flexible UI (customizations)
- Good documentation.

To init tailwind:

> npx tailwindcss init [creates a tailwind.config.js file]

tailwind.config.js:

```
module.exports = {  
  content: [ ],  
  theme: { extend: {} , } ,  
  plugins: [ ] ,  
}
```

