

Read Notes:

- + Bundler: 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]

Diff between (v) cache & (v) tide in package.json?

- v - automatically updates to newer versions
- v - strict on the versioning

Package-lock: tells the exact version that we use in our app (without cache/tide)
[locks the version in production & maintains the integrity of dependencies 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 separate 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 Dependency: When 1 library requires another library (should be listed in package.json) { then the other one requires another } [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] React fibre is renamed as the batch 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]

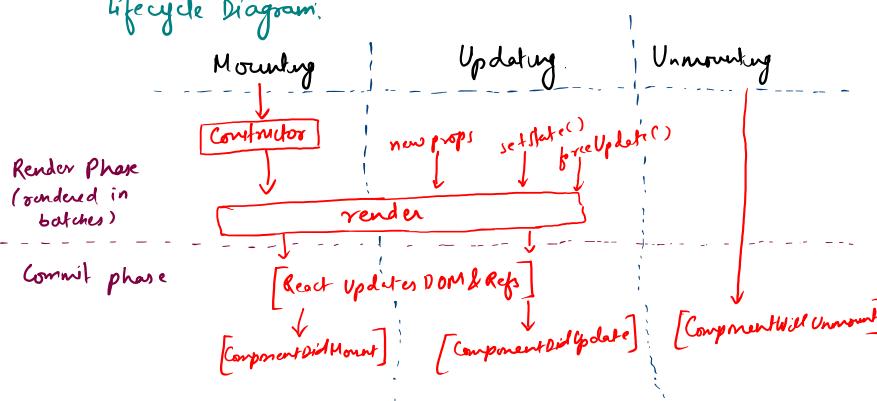
class based components

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: [ ] ,  
}
```

* State vs. props:

↳ State: Maintain a local variable that has a scope within a component.

↳ Prop: Value passed from one component to another.

* Prop drilling: Passing props from one component to another which is not a direct child.

Eg:
Prop drilling
not efficient
component

```
App (user)
  ↳ <Body user={user}/>
    ↳ <RestaurantContainer user={user}/>
      ↳ <RestaurantCard user={user}/>
```

* Lifting state up: Parent controls the states of child components.

[Read Commit]: March -13: fixed 'wide' show bug]

* React Context API:

↳ Central space to manage data.

↳ createContext & useContext anywhere.

↳ Context vs State & Props?

↳ Context is a central data
State & props are bound to components

↳ In class-based components, we have to use it on a component:

Wage
in our component
<UserContext.Consumer>
{ (value) => console.log(value) }

* We can override default value in

<UserContext.Provider>

↑
our context
name (varies)

* <React.Provider> & <React.Consumer> are very powerful.

Redux: Let's Build Our Store

Q) Why do we need Redux?
→ To handle data inside large app.

Q) Context vs. Redux?
→ Instead of making different contexts, we can create a Redux store.

Q) Alternatives:
→ NgRx, MobX

Q) Cons of Redux:
① Tough to setup/learn

* Redux: old way, Redux-Toolkit: New/Optimized way.

* Redux Architecture:

→ A separate entity to handle data (just like context, it can be accessible from anywhere). Unlike Context, there can be only one (1) store in the whole application.

→ Used for putting:
① theme-based data
② authentication-based data
and so on.

* Slices (logical separation of our store):

⇒ We create 'slices' of our store. Each store can have different 'slices'. There can be different kinds of slices:
① user slice ② cart slice
③ theme slice and so on.

* Our components cannot directly modify a 'slice':

⇒ We will have to 'dispatch' an 'action' for the same.

Diagram:



Eg Flow:

- ① Click on add to cart.
- ② we dispatch an action ('addItem' action)
- ③ action calls a function [called as 'reducer']
- ④ function modifies the cart.

Conclusion:

When we addToCart, it dispatches the action('addItem') which calls a reducer function, which then updates the cart slice.

* To read the cart:

⇒ We have to use something called as 'selector' (diagram above). Using selector is also called as 'subscribing' to the store.

Diagram:
④ → dispatcher(action) → reducer function] (updates)



* Installation:
① npm i @reduxjs/toolkit
core redux

② npm i react-redux
↑ bridge between react & redux.

* [PTO]

create slice! [code]

```

const cartSlice = createSlice({
    name: 'cart',
    initial State: {
        items: []
    },
    reducers: {
        addItem: (state, action) => {
            state.items.push(action.payload);
        }
    }
});

export const addItems = cartSlice.actions;
export default cartSlice.reducer;

```

name of the slice

initial value of the cart

initial state of the application.

addItem is the action that gets dispatched

[reducer function]

[data that will be passed when the action is dispatched.]

* Add slice to the store:

```

const store = configureStore({
    reducer: {
        cart: cartSlice,
    },
});
export default store;

```

adding a slice to a store.

* Subscribing to our store:

```

const cartItems = useSelector(store => store.cart.items)

```

Slice Name.

* For dispatching actions:

- ⇒ ① use Dispatch Hook,
- ② dispatch(addItem("Grapes"));

Testing (is. Time for the test)

★ jest and react-testing-library
[uses jest behind the scenes]

Q.) Why do we do testing?

=> One component can affect a lot of things, therefore, it is possible that the changes in dependent parts of the app. can introduce a bug. It should not be the case that adding new code breaks the existing code.

Q.) Test driven development: Very slow, but extremely efficient.

Q.) Types of testing: 1.) Manual Testing.

2.) Automation Testing (Selenium)

3.) E2E Testing (End-to-End: Cypress)

Importantly for dev.s

{ 1.) Unit Testing

5.) Integration Testing.

★ React - Testing - Library:

=> A part of DOM testing library.

↳ React - Testing - Library uses Jest behind the scenes:

=> Jest is a JS framework to focus on simplicity.

Steps:

1) Setting up React - Testing - Library :

- ① npm i --save-dev @testing-library/react
- ② npm i -D jest

2) Configure jest : npx jest --init

3) Installed 'jest-environment-jsdom'

4.) Create first test (inside --tests--)

5.) Configure babel [babel-jest]

6.) add configuration in .babelrc/babel.config.js file.

Testing:

* Render() function: [Tests a component inside a jsdom container]

↳ If code does not understand JSX, add
["@babel/preset-react", {"runtime": "es2015"}]

* Mocking Objects:

⇒ Create 'mock' folder,

⇒ Create mock file & export it.

⇒ In jest.config.js file, add config to moduleNameMapper object.

Eg. for Images:

```
moduleNameMapper: {  
  "....(png|svg|jpg)": "path of mock  
  file created"
```

Testing :

- * "import {StaticRouter} from "react-router-dom/server";"
↳ works without browser so it can be used
for testing.