React Notes:

* Bundler: Webpack, Vite, parcel [Read about this]
  - used in (create-react-app)
  - using this in current course!

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

Diff between (^) caret & (~) tilde in package.json?
^ - automatically updates to newer versions
~ - strict on the versioning.

Package-lock: tells the exact version that we use in our app (without caret/tilde)
[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
2. Minifying code
3. Cleaning code
4. Creating Dev & Production Builds (has superfast build also)
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

* Transitive Dependency: When 1 library requires another library & then the other one requires another one [Eg: A ← B ← C ← D]
  (handled by Package-lock.json)

* browserslist: []
  ↳ for compatibility in browsers
  (Ref: browserslist.dev)

↳ build command: npx parcel build index.html

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

↳ JSX → JavaScript XML

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

* How JSX does sanitization? (for CSS attacks)

* 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 (DOM) from other & finds out what needs to be updated (Hence key reconciliation is required).

* [After React 16] React Fiber is announced as the latest 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

} Read about this

PTO ⇒ Next Page!!

Namaste - React Course 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 childrens of a nested route go into Outlet!

[P.T.O]

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

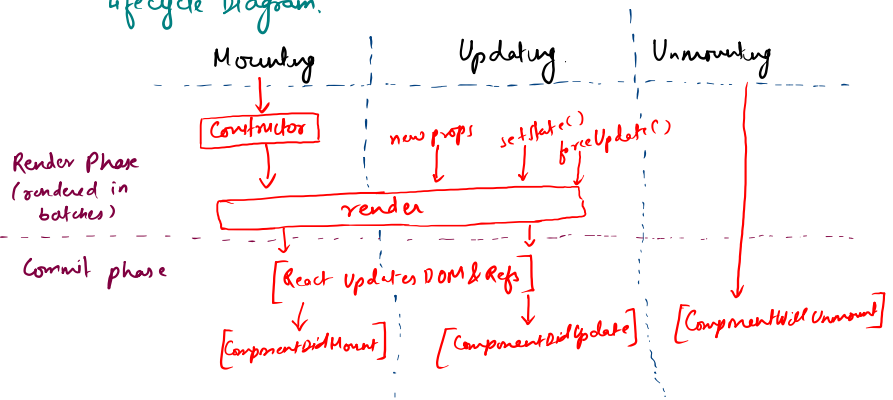
[PTO]

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 be 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 eventlisteners in useEffect cleanup

* Lazy Loading / chunking / splitting / Dynamic Bundling

> Implement Lazy loading:

lazy() function from react-library
↑ named export.

Syntax:
↳ const Instamart = lazy(() => import('./components/Instamart'));

* Suspense Component:

-> Lazy Loading Components can take a bit of time, so we have suspense component to render it a bit late:

[<suspense> <Instamart /> </suspense>]

↳ Suspense takes a props: 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 and cons of each)

→ Ways of writing CSS:
- Normal CSS
- SCSS (SASS)
- Inline CSS
- Component libraries (MUI, Mantine, 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:
App (user)
↳ <Body user={user}/>
↳ <RestaurantContainer user={user}/>
↳ <RestaurantCard user={user}/>

Prop drilling ↑ not efficient

★ Lifting State Up: Parent component controls the states of child components

[Read Commit: March -13: fixed hide/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 binded to components

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

Usage in class component:
<UserContext.Consumer>
{(value) => console.log(value)}
</UserContext.Consumer>

★ 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 apps

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

Q) Alternatives:
→ NGRX, MOBX

Q) Cons of Redux: (i) Tough to setup/learn

★ Redux: Old way, Redux-Toolkit: Newly opinionated 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: (i) theme-based data
(ii) authentication-based data
and so on.

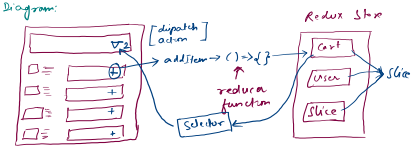
★) slices (logical separation of our store):

⇒ We create 'slices' of our store. Each store can have different 'slices'. There can be different kinds of slices: (i) user slice (ii) cart slice
(iii) theme slice and so on.

★ Our components cannot directly modify a 'slice':

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

Diagram:



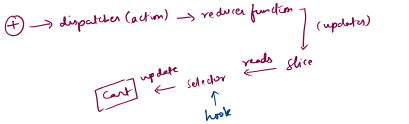
Eg Flow:
(i) Click on add to cart.
(ii) We dispatch an action ('addItem' action)
(iii) action calls a function [called as 'reducer']
(iv) function modifies the cart.

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:



★ Installation: (i) npm i @reduxjs/toolkit
↑ core redux

(ii) npm i react-redux
↑ bridge between react & redux

★ [PTO]

create Slice! [code]

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

export const { addItem } = cartSlice.actions;
export default cartSlice.reducer;
```

name of the slice

initial value of the cart

addItem is the action that gets dispatched

the initial state of the application.

[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)
```

SliceName.

★ For dispatching actions:

⇒ ① use Dispatch Hook,

② dispatch(addItem("Grapes"));