REACTJS NOTES

PART-1

www.linkedin.com/in/ashrayakk/

→ React is a library.

→ Main difference b/w framework & library:

Library takes minimum effort to put it cinto our code.

H.W

1) What is Emmet ?

→ designed to speedup the process of writing and editing code by providing a set of shortcuts that can be quickly expandable to full code blocks.

Let my index.html looks like :

```
<head>
    <title> Namaste All </title>
</head>
<body>
    <div id = "root">
        <h1> Namaste Everyone ! </h1>
    </div>
</body>
```

To do the above code in javascript:

```
<body>
    <div id = "root"></div>
</body>
<script>
    Const heading = document.createElement ("hi");
heading.innerHTML = "Namaste Javascript";
    const root = document.getElementById ("root");
    root.appendChild (heading);
</script>
```

To do the above code in ReactJS

H.W :- What is CDN?

Content Delivery Network (CDN) is a n/w of servers that delivers content to users. Both React and ReactDOM are available over a CDN.

H.W - What is Cross Origin attribute?

[CORS (Cross Origin Resource Sharing) is an HTTP-header based mechanism that allows a server to indicate any cross origins (domain, scheme or port) other than it's own from which a browser should permit loading resources.]

A):- Cross Origin attribute provide support for CORS

If we serve React from a CDN, keep crossorigin attribute set :-

```
<script crossorigin src="..." ></script>
```

This indicates that the script should be loaded from a different origin.

index.html

```
<body>
    <div id="root" ></div>
</body>
<script crossorigin
    src="https://...react@18..." >
</script>
<script crossorigin
    src="https://...react-dom@18/..." >
</script>
```

CDN Links

React Superpowers came from here

```
<script>
  const heading = React.createElement(
    "h1", { }, "Namaste Everyone");
  const root = ReactDOM.createRoot(
    document.getElementById("root"));
  root.render(heading);
</script>
</body>
```

Q). What is the { } denotes in above code?.

```
<div id="root">
    <h1 id="title">Namaste Everyone</h1>
</div>
```

This should come inside { }

Whatever I'm passing inside { }, will goes as tag attributes of h1.

React will overwrite everything inside "root" and replaces with whatever given inside render

* Do the below HTML code using React.

```
<div id="container>
    <h1> Heading 1 </h1>
    <h2> Heading 2 </h2>
</div>
```

To build a structure like this using React

```
<script crossorigin src="https://react@18..."/>
<script crossorigin src="https://react-dom..."/>
<script>
  const heading = create React.createElement(
    "h1",
      { id: "title",
      },
    "Heading 1"
  );
  const heading2 = React.createElement(
    "h2", { id: "title2", }, "Heading 2");
```

```
const container = React.createElement(
  "div",
  { id: "container" },
  [heading, heading2]
);

const root = ReactDOM.createRoot(
    document.getElementById("root")
);

root.render(container);
```

"BUT, JSX makes life more easy".

H.W

Why React is known as 'react'?

→ because it was designed to help developers "react" to changes in the state of an application, by efficiently rendering and updating the UI in response to those changes.

[klatch async & defer YT video].

Day 2 - Igniting our App

We will build our own 'create_React_app'

Code A :-

```
const heading = React.createElement(
    "h1",
    { id: "title",
    }
    "Heading 1"
);
```

* This code is exactly Similar to :-

Code B :- `<h1 id = "title" > Heading 1 </h1>`

ie, Code A will produce Code B in our DOM

If I want to create a div having 2 children h1 and h2, I'll do it like this :-

```
const container = React.createElement(
    "div", // createElement takes first argument as the "tag"
    { id :"container", // Second argument is the attributes
    }
    [heading1, heading2] // third argument is the children.
);
```

The above code will be like this ↓ inside DOM

```
<div id="container">
    <h1 id="title">Heading 1 </h1>
    <h2 id="title">Heading 2 </h2>
```

Inside second argument, we can write anything.

```
{
    id = "container"
    hello = "world"
}
```

These are called 'Props'

PROPS can be anything"

To make an app production ready, we should :-

(1) minify our file (remove our console logs, bundle things up)

(2) need a server to run things.

Even though we can load our "App.js", we can't get optimised version.

"Minify → Optimization → Clean console → and Bundle".

In React, to get external functionalities, we use "BUNDLERS" :-

a) Webpack is a bundler
b) vite
c) parcel

} These are alternatives

In create-react-app, the bundler used is "webpack".

Most bundlers do the same job.

Bundlers are packages. If we want to use a package in our code, we have to use a 'package manager'.

We use a 'package manager' known as npm or yarn.

npm :- No Problem Man

'npm' doesn't mean node package manager but everything else.

npm init (create a package.json file)

We use npm because we want a lot of packages in our project / react app.

[npm init -y] will skip lot of options.

npm install -D parcel

↳ dev dependency - beuz we want it in our developer machine....

↳ parcel is one of the dependency

Then, we'll get package-lock.json.

[Caret & tidle sign → read.]

our project will automatically update if we use caret sign. (^)

```
"devDependencies" : {
        "parcel" : "^2.8.2"
}
```

↳ caret

package-lock.json

will tell you which exact version of the library you are using.

Common issue

?) "Something is working on my localhost / mymachine but not works in production". Why?

A) · package-lock file tells the exact version of the library we are using. Suppose, for dev dependency we're using 2.8.4 version & is right in package-lock.json file.

But in 'package.json' file, it will be like "2.8.2"

'package-lock' file is an ^ file that locks the version

"node_modules"

→ which gets installed is like a database for the npm.

→ This is where the Super powers comes from.

→ Our app has dependency on 'parcel'

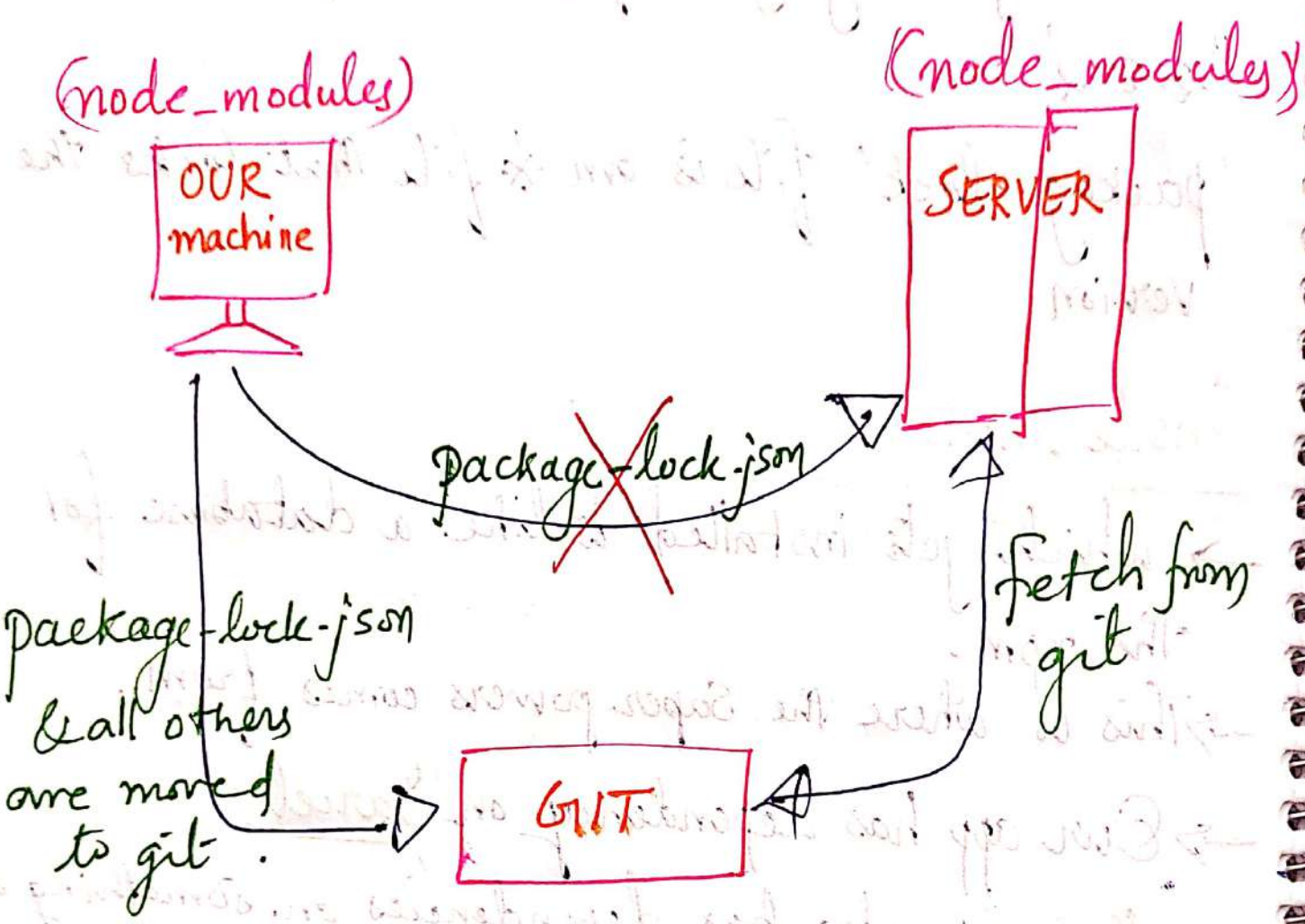
→ 'parcel' also has dependencies on something else.

All these dependencies/superpowers are in 'node_modules'.

?) We don't put 'node_modules' into git, why?

→ Because, our package-lock.json file have sufficient information to recreate node_modules.

→ package-lock.json file keep & maintain the version of everything in node_modules.



→ 'node_modules' in our machine can be regenerated in server using the package-lock.json file.

Previously, we used CDN links to get react into our app. This is not a good way...
We need to keep react in our node-modules.

```
npm install react
```

To ignite our app.

`npx parcel index.html`

- `npx` ▷ means 'execute using npm'
- `parcel` ▷ is the entry point

Click 'Enter'

Then, a miniserver is created for us:
like localhost:1234
Parcel given a server to us.
"Parcel" ignited our app.

As we removed CDN links, we don't have react in our app.
So, we want to import it into our app.
For that we use the keyword "import"

Never touch 'node_modules' & 'package-lock.json'

Normal js browser don't know "import".
So, it shows error.

```
import React from "react";
import ReactDOM from "react-dom/client";
```

As we got an error, we have to specify to the browser that 'we are not using a normal script tag, but a module'.

<script type="module" src="App.js"> </script>

We cannot import & export scripts inside a tag. Modules can import and export.

Note:

* Hot Module Reload (HMR)
  → means that parcel will keep a track of all the files which you're updating.

* How HMR works?
  → There is File Watcher Algorithms (written in C++). It keeps track of all the files which are changing realtime & it tells the server to reload.
  → These all are done by "PARCEL".

* There'll be a folder called .parcel-cache which will be there automatically.
In our project, parcel needs some space. So, it creates .parcel-cache.

* 'dist' folder keeps the files minified for us.

When we run command:

```
npx parcel index.html
```

This will creates a faster development version of our project & serves it on the server.

When I tell parcel to make a production build:

```
npx parcel build index.html
```

It creates a lot of things, minify your file. And "parcel will build all the production files to the dist folder"

* What takes a lot of time to load in a website?

A):- Media - Images

Parcel does image optimization also.

* Parcel also does 'Caching while development'.

* Parcel also takes care of your older version of browser

Compatible with older version of 'browsers'.

Sometimes we need to test our app on https becoz something only works on https.
Parcel gives us a functionality that we can just build our app on https. on dev machine.

npx parcel index.html --https

* Should put the .parcel-cache in gitignore

↳ because, anything which can be auto-generated should be put inside gitignore.

* Parcel ~~does~~ uses:

Consistent Hashing Algorithms

* Parcel is 'Zero Config'.

Parcel features on a glance:

-> HMR - Hot Module Reloading
-> File Watcher algorithm - C++
-> Bundling
-> minify code
-> Cleaning our code
-> Dev and production build
-> Super fast build algorithm
-> Image Optimization
-> Caching while development
-> Compression
-> Compactible with older browser version
-> HTTPS on dev
-> port Number
-> Consistent Hashing Algorithm
-> Zero Config
-> Tree Shaking

⁂ TRANSITIVE DEPENDECIES

We have our package manager which handles and takes care of our transitive dependencies of our code.

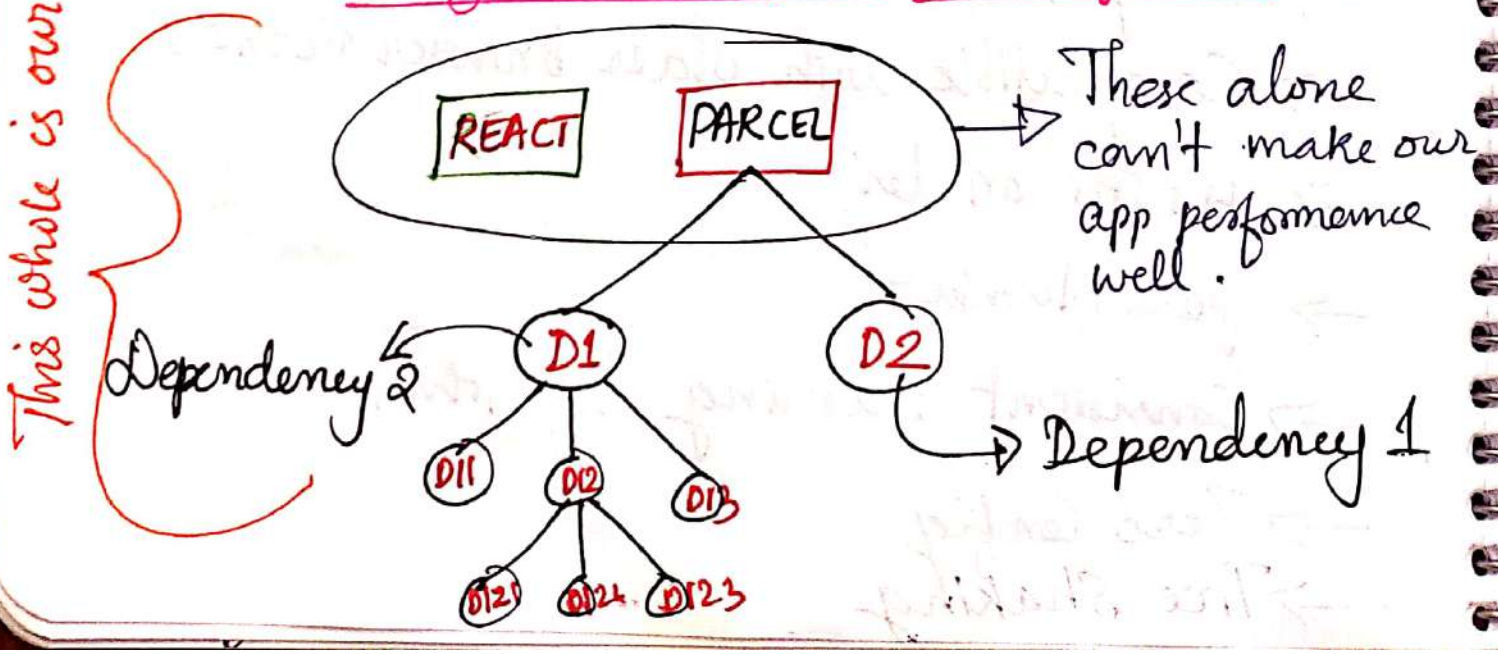
If you've to ~~use~~ build a production ready ~~build~~ app which uses all optimizations (like minify, cleaning bundling, compression, consistent hashing & etc) ~~then~~, we need to do all these.

But we can't do this ~~do~~ alone.

We need some dependencies on it.

Those dependencies are also dependent on many other dependencies.

Fig :- Our App (Dependency tree)



Q) How do I make our app compactible with older browsers?

A): There is a package called 'browserslist' & parcel automatically gives it to us.

Browserslist makes our code compactible for a lot of browsers

go to 'browserslist.dev'

In package.json file, do :-

```
"browserslist": [
  "last 2 versions"
]
```

→ Support 74%

feeded with some configurations

Parcel means my app will make sure that my app works in last 2 versions of all the browsers available.

If you don't care about other browsers, except chrome

```
"browserslist": [
  "last 2 Chrome versions"
]
```

→ Support 16%.

FINALLY,

"Parcel is a beast"..

→ Tree Shaking

→ parcel has this superpower

→ means removing unwanted code.

Eg:- Suppose your App is importing a library which has a lot of functions (say, 20 helper funths). Then, all those 20 fns will come into your code. But in my App, I may want to use only 1 or 2 out of it.

Here, PARCEL will ignore all the unused code

create-react-app : uses 'webpack' along with 'babel'

Q) How can you build a performant-web-scalable app?

→ There are so many things that react optimises for us and parcel (bundlers) gives us.

→ Our whole application is a combination of all these things.

Pollyfill

→ to make older browsers understand our new code, the new code is converted into a older code which browser can understand called pollyfill.

→ babel do this conversion automatically.

Eg:- ES6 is the newer version of javascript.

If I'm working on 1999 browser, my browser will not understand what is this const, new Promise etc

So, there is a replacement ~~for~~ code for these functionalities which is compatible with older version of browsers.

→ So, this is what happen when we write "browserslist" → our code is converted to older one.

Babel

→ is a javascript package / library used to convert code written in newer versions of JS (ECMAScript 2015, 2016, 2017 etc) into code that can be run in older JS engines.

To run our app, command is :-

```
npx parcel index.html
```

We always don't have to write this command.

Generally, we build a script inside package.json which runs this command in an easy way.

package.json

```
"scripts": {
    "start" : "parcel index.html",
    "test" : "jest"
}
```

So, to run the project, I've to use :-

```
npm run start
```

Shortcut :-

```
npm start
```

"start" script execute this command.

Build command

```
npx parcel build index.html
```

```
"scripts": {
    "build" : "parcel build index.html"
}
```

Note :-

`npx = npm run`

So, `npm run build` will build a project.

Console logs are not removed automatically by parcel. You have to configure your projects to remove it.

→ There is a package which helps to remove console logs :-

`babel-plugin-transform-remove-console`

→ Before ~~After~~ installing this package, create a folder called `.babelrc` } for configuration

→ And include :-

```
{
  "plugins": [["transform-remove-console",
    { "exclude": ["error", "warn"] }]]
}
```

→ Then, build : `npm run build`
to see that all console.logs are removed.

Render – means updating something in the DOM.

Reconciliation

→ helps to make React applications fast and efficient by minimizing the amount of work that needs to be done to update the changes.

→ So, you don't have to worry about what changes on every update.

Eg:-

```
<ul>
    <li> first </li>
    <li> second </li>
</ul>
```

} siblings.

When adding an element at the end of the children: The tree works well

```
<ul>
    <li> first </li>
    <li> second </li>
    <li> third </li>
</ul>
```

- render() function as creating a tree of React elements.
- On the next state or props update, render() fn will return a different tree of React elements.

Whenever react is updating the DOM, for eg :-

```
<ul>
    <li> Duke </li>
    <li> Villanova </li>
</ul>
```

Now, I introduced one child over the top, then react will have to do lot of efforts; react will have to re-render everything. That means, [react will have to change the whole DOM tree.]

```
<ul>
    <li> Connecticut </li>
    <li> Duke </li>
    <li> Villanova </li>
</ul>
```

As react has to re-render everything, it will not give you good performance.

In large-scale application, it is far too expensive.

SOLUTION – Introduction of Keys

→ React supports 'key' attribute.

→ When children have keys, React uses the key to match # children in the original tree with children in subsequent tree. Thus, making tree-conversion efficient

```
<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

Thus, react has to do very less work.

So, always use keys whenever you have multiple children.

CreateElement

→ React.createElement() is creating an object.

→ This object is converted into HTML code and puts it upon DOM.

If you want to build a big HTML structure, then using 'createElement()' is not a good solution.
So, there comes introduction of JSX.

JSX

When facebook created React, the major concept behind bringing react was "that we want to write a lot of HTML using Javascript" because JS is very performant.

```
import {createElement as ce} from "react";
Const heading = React.ce(
  "h1",
  { id : "title",
    key : "h1"
  }
  " Hello World "
);
```

Instead of writting all these :-

```
const heading =
  <h1> Hello World </h1>
```

This is JSX

※ JSX is not 'HTML inside javascript'

⟹ JSX has 'HTML-like' syntax

This is a valid javascript code :-

```
const heading = (
    <h1 id = "title" key = "h1">
        Hellowoorld
    </h1>
);
```

JSX expression

React keeps track of 'key'.

H.W :- 1) What is diff b/w HTML & JSX?

Our browser cannot understand JSX.

'Babel' understands this code.

HW

2) What are different usage of JSX ?

3) How to create image tags inside JSX ?

⟹ ~~JSX uses creat~~

→ JSX uses React.createElement behind the Scenes.

⇒ JSX ⇒ React.createElement ⇒ Object ⇒ HTML (DOM)

→ Babel converts JSX to React.createElement()

(Read Babel's Documentation)

→ JSX is created to empower React.

Advantages of JSX

- → Developer experience
- → Syntactical Sugar
- → Readability
- → Less code
- → maintainability
- → No Repeatition.

Babel comes along with parcel.

COMPONENT

'Everything is a component in React'.

React Components

—▷ 2 types :-

a) Functional component - NEW WAY of writing code.

b) Class Based component - OLD WAY

—▷ Functional Component

- is nothing but a javascript function.
- is a normal JS fn which returns some piece of react elements (here, JSX).

Eg :-

```
Const HeaderComponent = () => {
    return <h1> HelloWorld </h1>;
};
```

- For any component,
  Name starts with capital letter.
  (If is not mandatory, but it's a convention)

to render functional component, write :-

<Header Component />

React element

```
const heading = (
  <h1 id="title"
      key="h1">
    Hello World
  </h1>
);
```

converting it into arrow fn will make functional component.

Functional Component

```
const heading = () => {
  return (
    <h1 id="title"
        key="h1">
      Hello World
    </h1> );
};
```

React element is finally an object

Functional component is finally a function.

The Next Amazing thing

(1)

*
```
const Title = () => {
  <h1>Hello World</h1>
}
```

} is a functional component

* const HeaderComponent = () => {

```
    return (
      <div>
        <Title />
        <h2> Nomaste React </h2>
        <h2> Hai all </h2>
      </div>
    );
  };
```

instead of this you can write { Title() }

This is 'component composition'

This is a normal javascript function !!

---

② * const title = (

```
  <h1> Helloworld </h1>
);
```

} is a normal variable

* const HeaderComponent = () => {

```
    return (
      <div>
        {title}
        <h2>Nomaste React </h2>
        <h2> Hai all </h2>
      </div>
    );
};
```

NB:

* Whenever you write JSX, you can write any piece of javascript code between parantheis {}. It will work.

* JSX is very secure.
  JSX makes sure your app is safe.
  It does sanitization.

```
const data = api.getData();

const HeaderComponent = () => {
  return (
    <div>
      {data}
      <h2> Hello World </h2>
    </div>
  );
};
```

→ Write anything inside {}, JSX will sanitize the code.

Component Composition

If I have to use a component inside a component. Then, it is called component composition / Composing components.

"Building a Food-ordering App".

* Is JSX mandatory ? ⇒ NO
* Is Typescript mandatory ? ⇒ NO
* Is ES6 mandatory ? ⇒ NO

3 ways of component composition :-

1. {Title()}
2. <Title/> ⟶ used generally.
3. <Title></Title>

BUILDING OUR APP...

Name : Food Villa

"Whenever you are writing code, do planning."

Our app look like this 

Food Villa

Nav-bar / Header

LOGO

Home About Support Cart

Body

Search

Image

Name of Restaurant
Cuisines
Star Rating

Image

Name of Restaurant
Cuisines
Star Rating

Image

Name of Restaurant
Cuisines
Star Rating

Image

Name of Restaurant
Cuisines
Star Rating

Footer

@2022 Foodvilla Copyright

So, the app layout should have :-

* Header
* Body
* Footer.

```
Const AppLayout = () => {
  return (
    - Header
        - Logo
        - Nav Items (on right side)
        - Cart
    - Body
        - Searchbar
        - RestaurantList
            - RestaurantCard
                - Image
                - Name
                - Rating
                - Cusines
    - Footer
        - links
        - Copyrights
  )}
```

App.js

HEADER COMPONENT

For building header component :-

```
✡ Const Title = () => {
    <a href="/">
      <img
        alt="food_villa_logo"
        src="https://someurl"
      /></a>
  };

✡ Const HeaderComponent = () => {
    return (
      <div className="header">
        <Title/>
        <div className="nav-items">
          <ul>
            <li>Home</li>
            <li>About</li>
            <li>Contact</li>
            <li>Cart</li>
          </ul>
        </div>
      </div>
    );
  };
```

Note :-

※ Jsx expressions must have one parent element.

React.Fragment

→ is a component which is exported by 'React'.
[import React from "react";]

→ grouped a list of children without adding extra nodes to the DOM.

Eg :- import React from 'react';

```
const AppLayout = () => {
    return (
        <React.Fragment>
            <Header/>
            <Body/>
            <Footer/>
        </React.Fragment>
    );
};
```

→ Shorthand syntax <> </> is used instead of <React.Fragment> </React.Fragment>

* But, you can't pass styles to empty brackets.

Giving style inside React

To give inline styles in react, do :-

FIRST METHOD ↓

Eg :-

```
* const styleObj = {
      backgroundColor : "red",
  };
* const jsx = (
      <div style = {styleObj}>
          <h1> Hello </h1>
          <h2> World </h2>
      </div>
  );
```

styleObj is Normal JS object

inside this parantheis, you can write any piece of JS code.

→ In React, style is given using javascript objects

→ Alternative way :-

* const jsx = (

```
<div style = {{
    backgroundColor : "yellow";
}}>
<h1> Hello </h1>
<h2> World </h2>
</div>
);
```

SECOND METHOD

→ give class name to the div (or whatever tag) and write CSS inside css file.

* const jsx = (

```
<div classname = "jsx" >
<h1> Hello </h1>
<h2> World </h2>
</div> );
```

* css file

```
.jsx {
    backgroundColor : "blue";
}
```

THIRD METHOD ⬇

→ using external library like 'Tailwind css', 'Bootstrap', 'MaterialUI', etc

---

H.W → Can I use a 'React.Fragment' inside my 'React.Fragment' ?

A):- Yes, you can nest 'React.Fragment' components inside other 'React.Fragment' components.

Eg:-

```
import React from 'react';
const jsx = (
  <>
    <ChildA />
    <>
      <ChildB />
      <ChildC />
    </>
    <ChildD />
  </>
);
}
```

Child B & C are siblings & will be grouped together without adding an extra node to the DOM.

BODY COMPONENT

While building restaurant cards, we need some data for this card. Ways :-

- Using hard coded data,
- integrate with api

Hard coded data - code will be like ↓

```
* const RestaurantCard = () => {
    return (
      <div classname = "card" >
        <img src = " https://some url" >
        <h2> Burger King </h2>
        <h3> Burgers, American </h3>
        <h4> 4.2 stars </h4>
      </div>
    ); };

* const Body = () => {
    return (
      <div>
        <RestaurantCard />
      </div>
    );
  };
```

The name, image and all other datas shown in the above card won't be same always. So, it should be dynamic.

As we are using JSX, we can do javascript inside HTML.

Making data dynamic

* const burgerking = {

    name: "Burger King",

    image: "https://some url",

    cusines: ["Burger", "American"],

    rating: "4.2",

};

In real world data does not comes like this

* const RestaurantCard = () => {

    return (

            <h2>{burgerking.name}</h2>

            <h3>{burgerking.cusines.join(",")}</h3>

            <h4>{burgerking.rating}</h4>

    );

};

In real world, data is not like this. There are a number of restaurants.

To render many restaurants :-

(i) have many restaurant cards by doing :

```
Const Body = () => {
  return (
    <div>
      <RestaurantCard />
      <Restaurant Card />
      <Restaurant Card />
      <Restaurant Card />
    </div>
  );
};
```

(ii) Make all these cards dynamic.

In real world, data doesn't comes like above one [burgerking] having a single object.

It comes as "array of objects".

One of the obj would be that 'burgerking'. Similarly, there would be other objects as well.

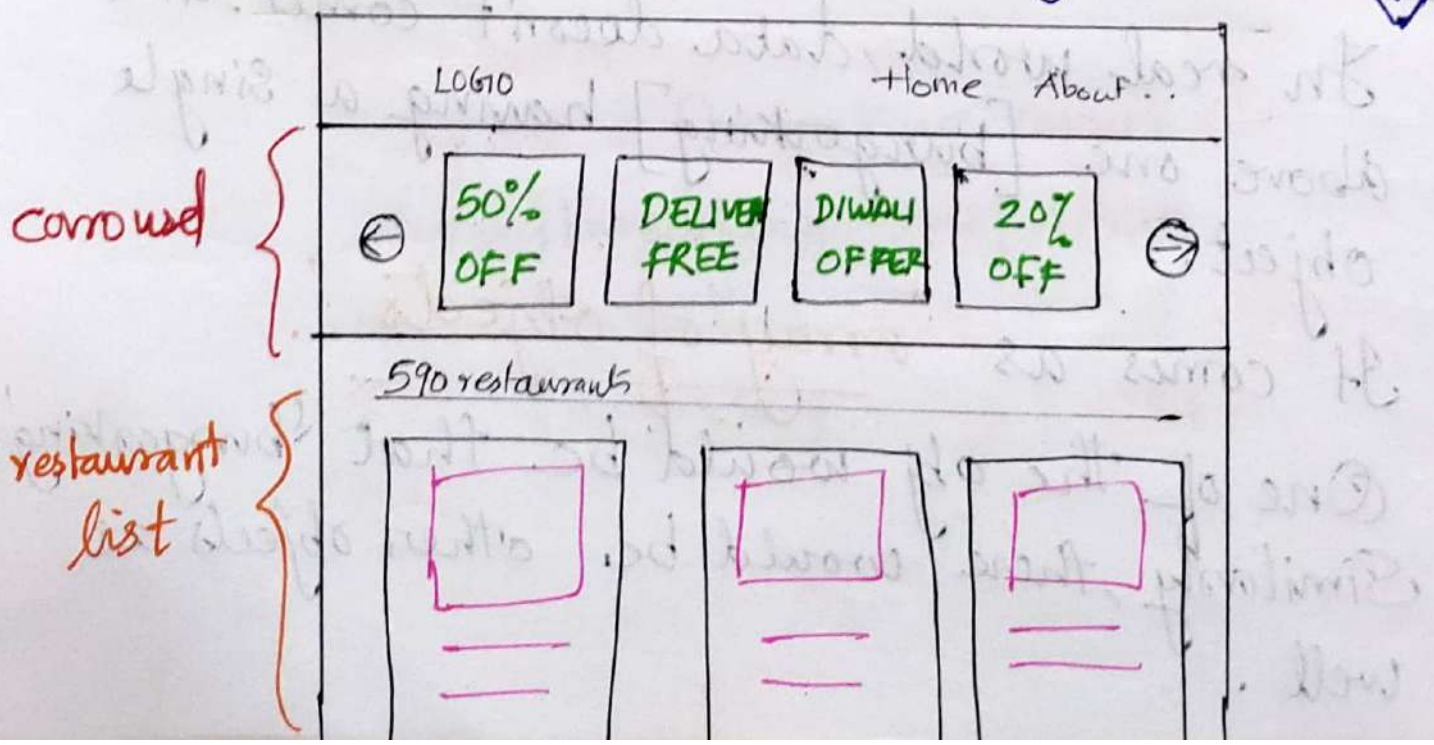
CONFIG DRIVEN UI

All the UI (let say, Swiggy homepage) is driven by a config which is send by backend (api)

Eg:- Swiggy should work in Kolkata, Dehradun, Delhi and whatever location it is, we should have different website info on each location.
For that, we control our front-end using 'config'. That is why it is known as Config driven UI.
All these are controlled by backend.

* How we design Config driven UI?

Suppose, our UI is something like this ↓



If our 'config' is coming from backend, and my data will come like:

```
const config = [
  {
    type : "carousel",
    cards : [
      {
        offerName : "50% OFF"
      },
      {
        offerName : "No Delivery Charge"
      }
    ]
  },
  {
    type : "restaurants",
    cards : [
      {
        name : "Burgerking",
        image : "https://some url",
        cusines : ["Burger", "American"],
        rating : "4.2",
      },
```

These offers are different for different locations

If there are no offers for a particular location, our backend will not send us this object or send empty list of cards

Only backend will change offers & website will render accordingly

```
{
    name : "KFC",
    image = "https://some url",
    cusines : ["Burger", "American"],
    rating : "4.2",
}
]]]
```

* Take data from swiggy website and build your own — watch from 01:45:37

Optional chaining

→ allows us to access an object's properties without having to check if the object or its properties exist.

→ represented by '?'.

→ new feature introduced in javascript ES2020.

So, after taking real data from swiggy, our 'Restaurant Card' function looks like this:

```
const RestaurantCard = () => {
  return (
    <div className="card">
      <img src={"https://cloudinaryurl" +
        restaurantList[1].data?.cloudinaryImgId
      } />
      <h2>{restaurantList[0].data?.name}</h2>
      <h3>{restaurantList[0].data?.cuisines.join(", ")}</h3>
      <h4>{restList[0].data?.time} minutes</h4>
    </div>
  );
};
```

Now, all my cards will show same data. I need to transfer dynamic data. We are having hard coded data right now. So, I'll make like :-

my first <RestaurantCard/> card should come from first object & second card from second object

So, the BODY will look like below :

```
* const Body = () => {
    return (
      <div className="restaurant-list">
        <RestaurantCard restaurant={restaurantList[0]} />
        <Restaurant card restaurant={restaurantList[1]} />
        <Restaurant card restaurant={restaurantList[2]} />
        <Restaurant card restaurant={restaurantList[3]} />
        <RestaurantCard restaurant={restaurantList[4]} />
      </div>
    );};
```

Whatever you pass in here is known as PROPS
(here, restaurant is the PROPS)

PROPS

→ Shorthand for properties

→ "passed props" means I'm passing some data or class. or properties into my functional component.

→ restaurant = {restaurantList[3]}

This means react wraps up all these properties into a variable known as 'props'. I can call it anything.

→ So, our RestaurantCard function will now look like ↓

```
* const RestaurantCard = (props) => {
    return (
      <div className="card">
      <img src={"https://some url" +
          props.restaurant.data?.cloudinaryImgId}
      />
      <h2>{props.restaurant.data?.name}</h2>
      <h3>{props.restaurant.data?.cuisines.join(",")}</h3>
      <h4>{props.restaurant.data?.time}minutes</h4>
      </div>
    );
};
```

→ OBJECT DESTRUCTURING

```
(props) => ({restaurant})
(restaurant.data?.name)
```

We can destructure our ({restaurant}) also ↓

* const {name, cuisines, cloudinaryImg, time} = restaurant.data;

then,
{restaurant.data?.name} ⟹ {name}

If I don't want to destructure like above. but want to destructure everything on the flyer Then it will look like ↓

* const RestaurantCard = ({
    name, cuisines, cloudinaryImgId, time
  }) => {

```
return (
  <div classname ="card">
  <img src = { "https://some url" +
     cloudinaryImgId }
  />
<h2> {name} </h2>
<h3> {cuisines.join(", ")}</h3>
<h4> {time} minutes </h4>
</div> ); };
```

→ Then, I have to pass my individual props in Body :-

```
* const Body = () => {
    return (
    <div classname = "restaurant-list">
    <RestaurantCard
      name = {restaurantList[0].data.name}
      Cuisines = {restaurantList[0].data.cuisines}
    />
    <RestaurantCard
      name = {restaurantList[1].data.name}
      cuisines = {restaurantList[1].data.cuisines}
    />
</div> ); };
```

This has all the props like name, cusines, time etc & I may want to pass all of these props. So, instead of writing each prop individually. I will do :-

```
{...restaurantList[1].data}
```

Spread Operator

→ denoted by three dots (...)

→ It allows an iterable (an object that can be looped over, such as an array or a string) to be expanded in places where multiple elements are expected.

Eg :- For above code for Body(), multiple elements are to be written for every <RestaurantCard/>

So, instead of that we can write

{...restaurantList[0].data}

So, my Body() fn will now look like ⇩

```
* const Body = () => {
    return (
      <div className="restaurant-List">
        <RestaurantCard {...restaurantList[0].data} />
        <RestaurantCard {...restaurantList[1].data} />
        <RestaurantCard {...restaurantList[2].data} />
      </div>
    );};
```

→ But if I have a 100 restaurant, writing code like above is not a good way.
So, we can run a loop over this. But in functional programming, we don't use for loop.
We use `.map`.

`.map` is the best way to do it.

[* Watch map, filter & redux video on YT.]

→ HW :- difference between map & foreach.

→ So, using map, my code looks like ↓

```
* const Body = () => {
    return (
      <div className="restaurant-list">
        {restaurantList.map((restaurant) => {
          return (
            <RestaurantCard {...restaurant.data}/>
          );
        })}
      </div>
    ); };
```

→ restaurantList — is my array of objects.

restaurantList.map( ) —— means :-

This will map my array and I will pass a function (callback function). This callback fn takes each object. ie, (restaurant).

So, for each object in that array, I want ~~to~~ my function to return some piece of JSX. (That JSX is my <RestaurantCard/>

So, I will spread my (restaurant) object in card..

→ Everything that we build is a CONFIG DRIVEN UI.

→ After running, React will give a warning that "Each child in a list should have a unique "key" prop.

<RestaurantCard {...restaurant.data}

key={restaurant.data.id}

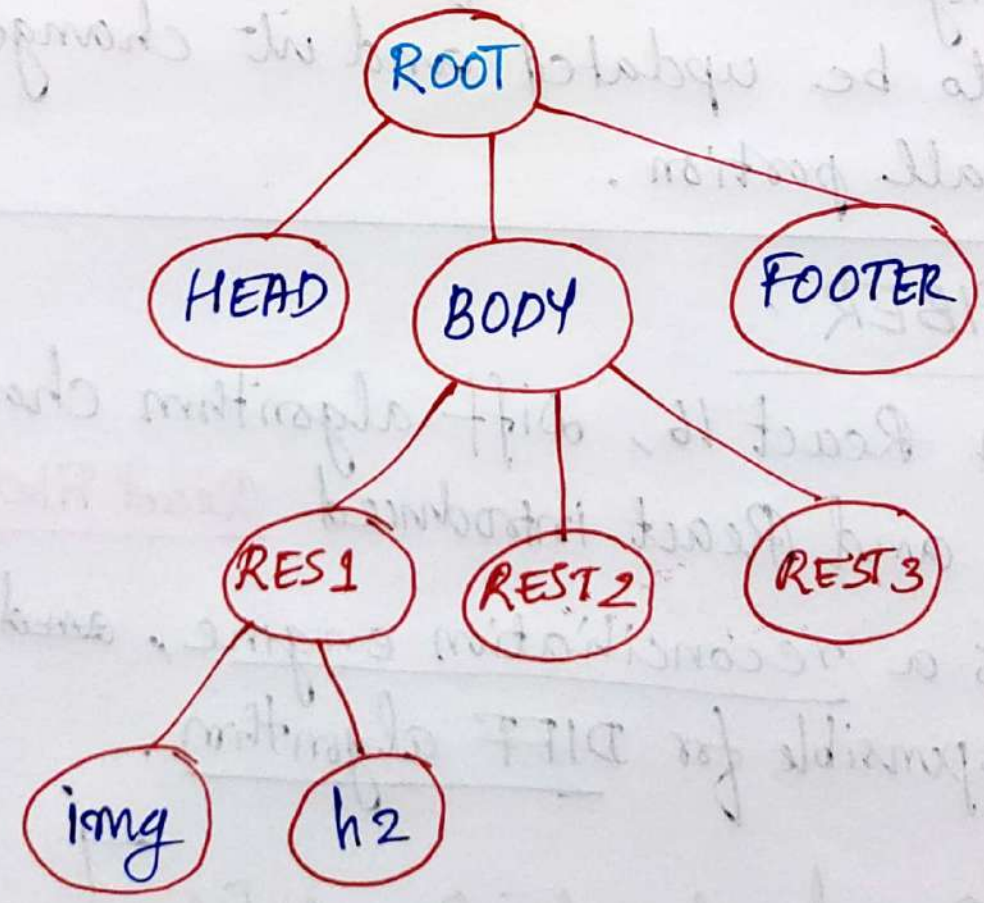
/>

VIRTUAL DOM

→ Let the structure of our DOM look like ↓

```
<>
    <head>
    <body>
        <Rest 1>
        <Rest 2> <img .. />
        <Rest 3>
    </body>
    <head> <footer />
</>
```

→ We keep a representation DOM with us, which is known as virtual DOM.



→ We need Virtual DOM for Reconciliation

* → Reconciliation is an algorithm that React uses to diff one tree from other. It uses Diff Algorithm and in determining what needs to change and what does not in UI.

[To find out DIFFERENCE between one tree (Actual DOM) and other (VIRTUAL DOM)]

→ Diff Algorithm then finds out what needs to be updated and it change only that small portion.

REACT FIBER

→ In React 16, Diff algorithm changed a little and React introduced React Fiber

→ It's a reconciliation engine, ~~and~~ which is responsible for DIFF algorithm.

[Read about React fiber]

React File Structure

* React doesn't have opinions on how you put files into folders
* Some devs group their files by features
* While moving files into folders, we need to export it so that we can import it wherever necessary.

There are 2 ways of exporting :-

(i) export default

→ This is the default way.

→ This means you want to export only one value.

A module is a self contained unit that can expose assets to other modules using export, and acquire assets from other modules using import

→ There can be only one default export.

→ Default export is the value that will be imported from the module, if we use the simple import statement :-

```
import Title from "./components/Title";
```

'Title' is the name that will be given locally to that variable assigned to contain the value, and it doesn't have to be named like the origin export.

(ii) Named exports

Eg :- `export const Title = () => { }`

→ This is a named export with a name 'Title'

→ It can be imported like :-

```
import {Title} from "./components/Header";
```

→ If we created a new file which have 2 components and I want to export both of these components.

(i) I can wrap these components into single object and can export.

(ii) Or I can export it seperately

→ If Header.tsx have 2 components :-
Header & Title.
Then, either we can ~~each~~ export each using names,
or wrap it into single component and use default export. So, import {Title, Header} from "./components/Header"; will work.
Or we can use :-

```
import * as Obj from "./components/Header";
```

Then, we can use
'Obj.Title' in our code

Header.js

```
export const Title = () => {..}
~~export~~ ~~const~~
const Header = () => {..}
export Default Header
```

App.js

```
import Header,
{Title} from
"./comp/Header"
```

When you are using the component in same file, you don't have to export.

CONFIG FILE

- Create a config file in your project.
- I put all the hardcoded things into my config file. (config.js)
- Also called as constants file (constants.js)
- Config file ~~should~~ will be like :-

```
const IMG_CON_URL = "https://some url";
```

- make this as named export

```
export const IMG_CON_URL = "https//";
```

- Then import it like :

```
import { IMG_CON_URL } from "./config";
```

- Put all hardcoded data. So, put restaurantList also in config file.

Building Search Functionality

→ Search bar - inside Body.

→
```
const Body = () => {
  return (
    <>
      <div className="search-container">
        <input
          type="text"
          className="search-input"
          placeholder="Search"
          value=""
        />
        <button classname="search-btn">
          Search
        </button>
    </>
  );
};
```

→ We've got search input and search button with us. But if I try to write inside my input box, it's not working (because it's controlled by React.

[If I write the same code inside my HTML file, it will work.

ONE-WAY DATA BINDING IN REACT

* Const Body = () => {

```
const SearchTxt = "KFC";
return (
  <>
  <div>
   <input type="text"
          placeholder="Search"
          value={searchTxt}
  </>      />
  );};
```

I have a variable 'SearchTxt' and if I put that here, Then, the value 'KFC' will go inside my input box.

→ I'm not able to edit the value "KFC" because it is a hardcoded value.

→ To change the value in the input box, we need to modify the variable SearchTxt

But in input box, If we write something, it won't change SearchTxt.

This is called One-way data binding.

* How will I change the value of SearchTxt ?

Ans) :- Write an onChange method

onChange = {(e) => onChangeInput}

Create an onChangeInput function - It takes a function (which is a call back fn) which have a e event.

So, whenever input is changed, this function will be called.

→ If you need to maintain a variable that changes itself, then you need to maintain a 'React-kinda' variable.

→ React Variable.

It's like a state variable.

* [Every component in React maintains a state. So, you can put some variables on to that state.]

[Every time you have to create a local variable, you use state in it.]

→ In react, If I want to create a local variable like SearchTxt, I will create it using useState Hook

useState Hook

→ New way of creating variables

```
Const [searchTxt] = useState();
```

If we have to create local variables in react, you need to use state variables.

→ State variables are created using **useState hook**

What is Hooks ?

- Hooks are normal functions.
- I get useState hook from 'react' library. (imported using named export)

What is the function of useState ?

- It's to create state variables..

```
const [SearchTxt] = useState();
```

(Second element is a set function to update the variable)

This function returns an array

The first element of this array is variable name.

'SearchTxt' is a local state variable.

→ To give a default value to my useState variable, do this ↓

```
Const [searchTxt] = useState("KFC");
```

→ This is how we create a local state variable in react.

In javascript, we create a variable searchTxt having a value "KFC" like this ↓

```
const searchTxt = "KFC";
```

→ In react, to modify the variable 'searchTxt' I have to use function.

useState() gives us that function.

Let us call that function 'setSearchTxt()'

```
const [searchTxt, setSearchTxt] = useState("KFC");
```

```
onChange = {(e) => {
    setSearchTxt("e.target.value");
}}
```

e → from this event property I can read whatever I'm typing

```
* const Body = () => {
  const [searchTxt, setSearchTxt] = useState("KFC");
  return (
    <>
    <div classname = "search-container">
    <input type = "text"
           className = "search-input"
           placeholder = "Search"
           value = {searchTxt}
           onchange = {(e) => {
             setSearchTxt(e.target.value);
           }}
    />
    <button className = "search-btn">
      Search - {searchTxt}
    </button>
    </>
  );};
```

TWO-WAY BINDING

Here, I'm reading as well as writing searchTxt. ~~Both~~

※ We have local variables. Why do we need state variables?

A):- Becoz, react has no idea what's happening to your local variables. So, react won't re-render any updates happening on that variable. Everytime, the variable wants to be in sync with the UI. For that, we need to use State variables.

React keeps track of State variable.

※ Whenever my variable is updated, my whole 'Body' (here) component re-renders. ie, React destroy the 'Body' component and create it again. Reconciliation (Diff algm) is happening behind the scenes.

※ But it just re-render that updated portion. It is very quick.

[Interesting Section → 01:43:00]

Search Functionality

① We need to filter the data.

→ data here is restaurantList.

→ Create a function filterData();

② Update restaurantList when we filter the data.

→ I cannot update it directly,
we've to make a State variable for this.

```
Const [restaurants, setRestaurants] = useState(
                                      restaurantList);
```

③ Suppose I updated my restaurant with filterdata,
then I must modify this local variable,
restaurant with the filter data.

FilterData function

```
filterData (searchTxt, restaurants)
```

searchTxt → my input. This the text we have to search in

restaurants → I'll search my searchTxt inside restaurants and filter data.

```
Const data = filterData(searchTxt, restaurants);
  setRestaurant(data);
```

Filter Algorithm (normal js).

```
function filterData(searchTxt, restaurants) {
  return (
    restaurants.filter((restaurant) =>
      restaurant.data.name.includes(searchTxt)
    );
}
```

```
<div className = "restaurant-List">
 {restaurants.map((restaurant) => {
    return (
      <RestaurantCard {...restaurant.data}
        key = {restaurant.data.id} />
    );
 })};
</div>
```

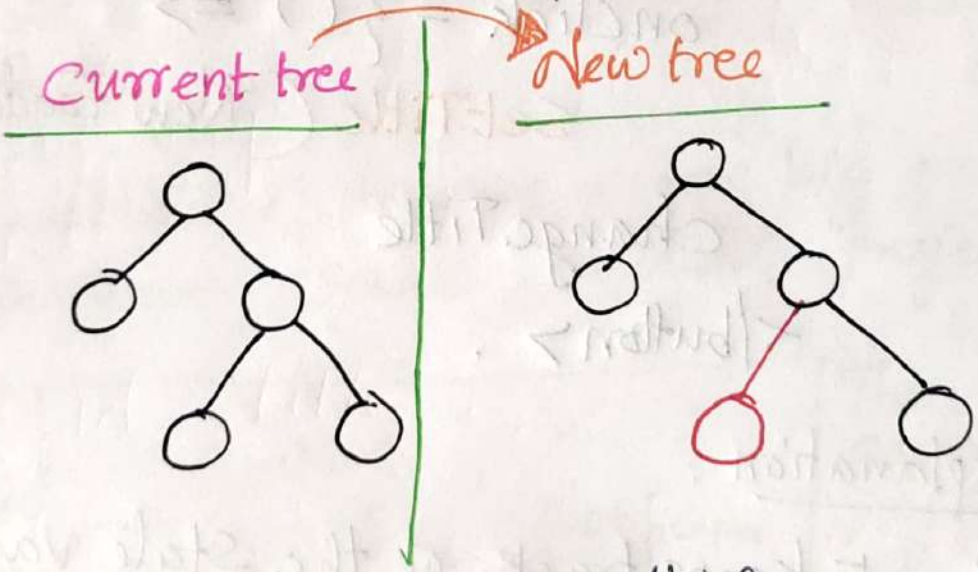
After one search, the state got updated and again when we search keywords, it won't work.

Why React is fast ?

→ Because it has Virtual DOM, Reconciliation, Diff algorithm.

→ In Diff algorithm, current tree is compared with the new tree and the difference is reflected on the DOM.

→ React Fibre is the updated reconciliation algm.



→ React is fast because of its fast DOM manipulation.

→ Diff algm 'detects' what exactly got changed in the page & it'll just change that while re-rendering the whole tree.

useState Hook

React gives a state variable and a function that update state variable.

Eg:- `const [title, setTitle] = useState("hi")`

initial value.

```
return (
<div>
  <h1> {title} </h1>
  <button
    onClick = {() =>
      setTitle("New Food App")} >
    changeTitle
  </button> .
```

Explanation :

React keeps track of the state variable title, once it is created.

On click of the button 'changetitle', the title get updated from 'hi' to "New food app"

The whole UI will re-render and it will update the UI quickly.

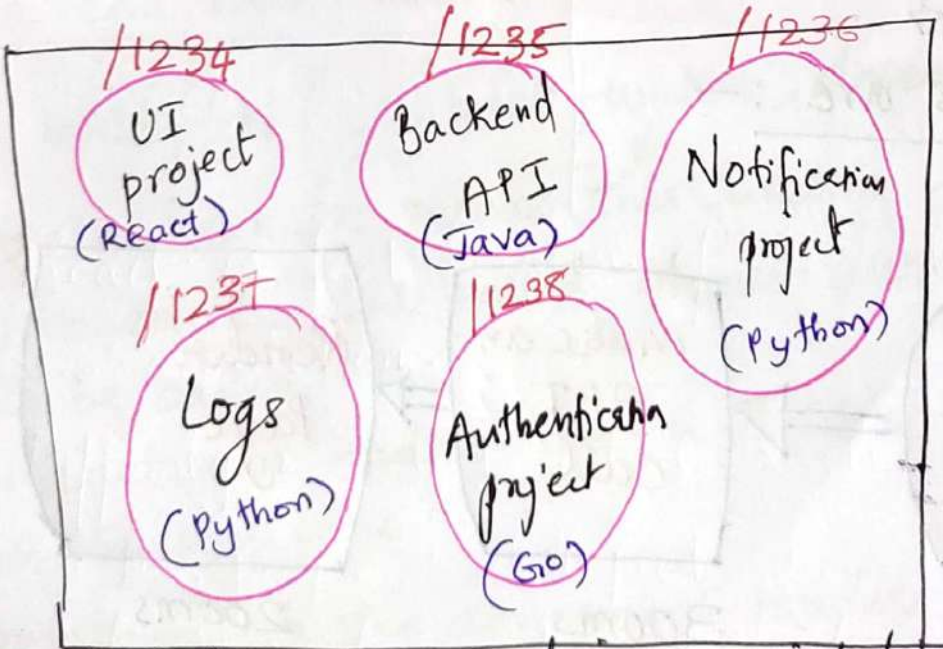
Whenever a state variable is changed, React re-freshes # or re-render the whole component. And react do this super fastly.

MICROSERVICES

In older days, there used to be a single big application. So, everything like APIs, SMS, Notification, UI, JSP pages etc used to be in same project. Suppose if we have to change one button, we used to deploy this whole project / app/" . It was such a mess. This architecture was known as monolith

SEPERATION OF CONCERN

But now, instead of having a one big project, we used to have small different projects.



There are seperate projects. Here, seperation of concern or single responsibility is there.

The tools & languages used in a project depends on the usecase. All these projects are deployed in different ports but same domain name.

How to make an API call ?

→ Javascript gives us fetch() function

→ Where do we call this api in our component ?

On every ~~UI~~ state change or any time my UI is updated, this would get re-render if we called api just after or before the usestate declaration. This is not a good way.

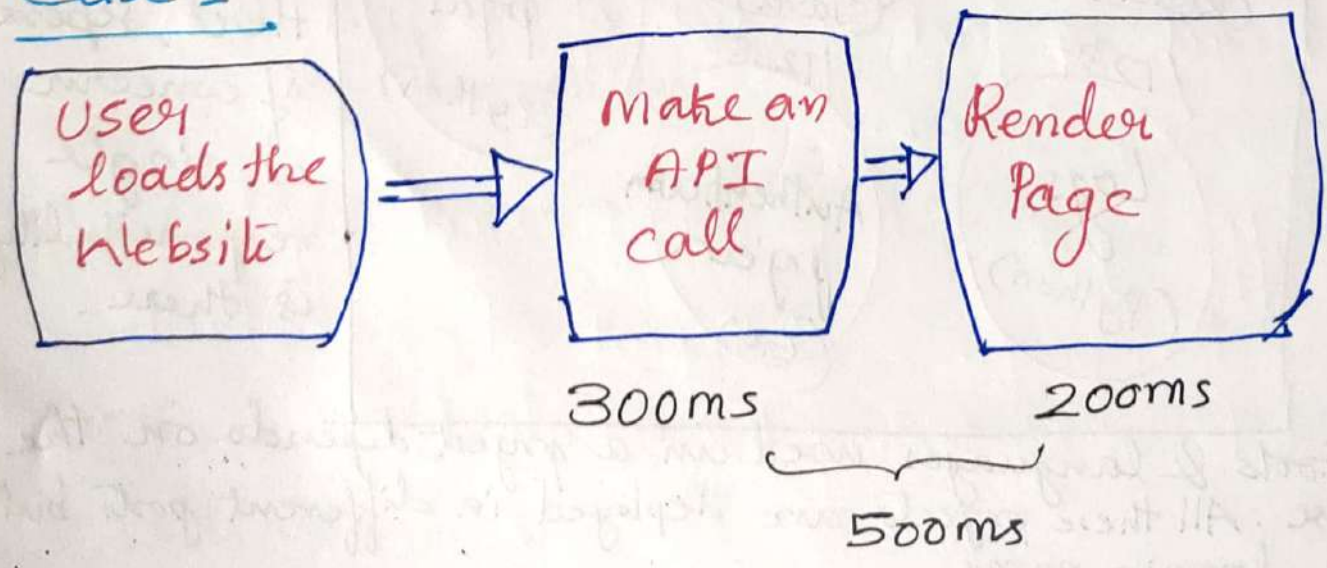
Best Way to call an api ?

It should be like, as in when our 'Body' component loads, it used to call an API and fill the data.

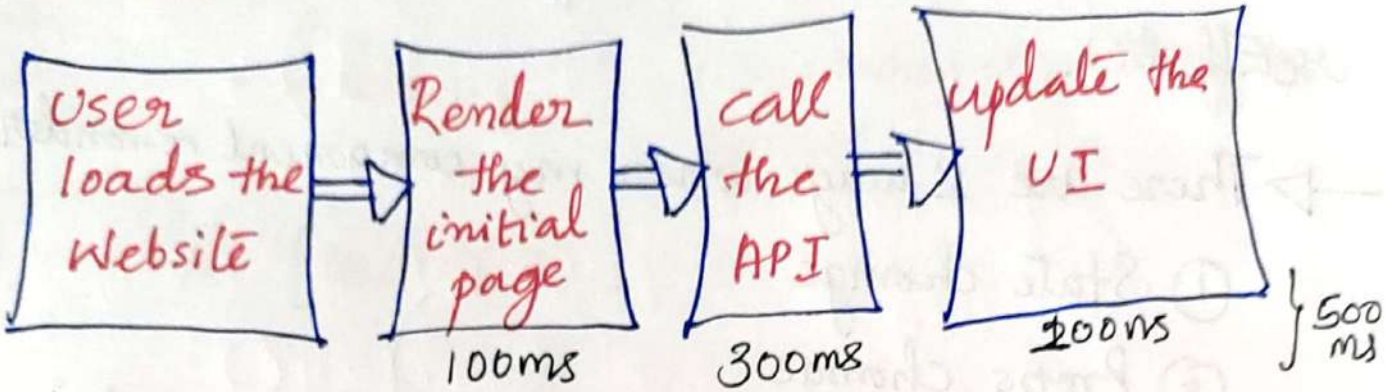
There is different ways to handle this in react :-

2 Ways are :-

Case 1



Case 2



Case 2 is the good way. All this will happens very quickly. To make this happens, react gives us a hook: useEffect

useEffect Hook

→ We get this from react library.

→ useEffect is a function. We call this function by passing another function to it which is a call back function.

useEffect (callback function , dependency array)

→ Call back fn means "this function" is not called immediately but called whenever useEffect wanted to be called.
React will make sure that it is called in a specific time.

→ Whenever the component renders & re-renders & re-renders, first of all, the code of the component will be called & after every render,

it will call the callback fn that pass inside useEffect().

→ There are 2 ways when my component re-render :

① State change

② Props change

→ useEffect will be called on every re-render, which is a bad way. If we don't want to call it after every re-render, pass in a dependency array into it.

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

[ ] → dependency array

() => { } → call back function

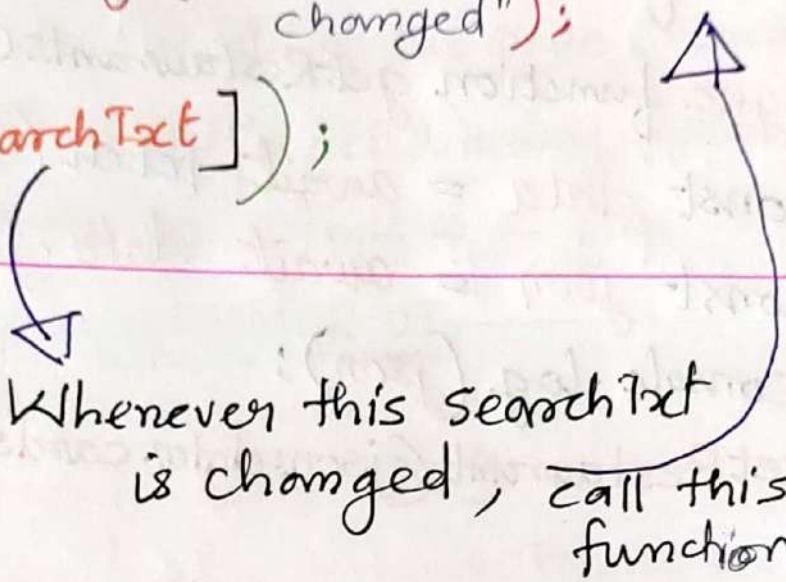
→ Role of Dependency array :-

Eg: → const [searchTxt, setSearchTxt] = useState("")

→ useEffect(() => {
    console.log("render")
}, [searchTxt]);

Suppose, I want to call this useEffect only when the 'searchTxt' changes, then I have to pass that 'searchTxt' in the array.

```
useEffect(() => {
    console.log("call this when dependency is
                 changed");
}, [searchTxt]);
```



Whenever this searchTxt is changed, call this callback function

Suppose, if I don't want my 'call-back' fn to be dependent on anything. It will be called just once.

And also It will be called after render.

So, useEffect will be called just once and after initial render if there is empty dependency array.

Fetching real data

API call happens asynchronously.

Let's create a async function.

Async-await is the most preferred way.

* useEffect (() => {
getRestaurants();}, []);

* async function getRestaurants() {
const data = await fetch("https/swiggyurl");
const json = await data.json();
console.log(json);
setRestaurant(json?.data?.cards[2]?.data?.data?.cards)
};

There are security parameters & browser will block us from calling that swiggy api.
Browser won't let us to call swiggy from our localhost.
To modify this, there is a plugin CORS.

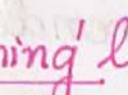
[Allow CORS: Access-Control-Allow-Origin]
Add this plugin to chrome
This plugin lets you bypass the CORS error

[Watch video - CORS - YT]

The old data will be rendered first for a few seconds and then new data comes.

If we removed that old data, page shows an ugly UI. So, initial screen to get rendered should be a loader / shimmer UI. That is a basic skelton.

SHIMMER UI

There was a research done and earlier people used to saw 'Spinning' loaders  at first and then suddenly every restaurants come up.
This is a bad user experience. ✗.
Human brain don't like to view so many fluctuations in the UI, according to psychology.
Psychologist figured out that, instead of Spinners empty boxes should be shown. It is a better UI experience then for the users ✓
As suddenly changes are not happening, our eyes won't hurt
This is known as SHIMMER UI

Shimmer UI resembles actual UI, so users will understand how quickly the web or mobile app will load even before the content has shown up.
Every big company is following this.

CONDITIONAL RENDERING

→ Same as conditions (if operator or the conditional operator) work in Javascript.

→ Your components will often need to display different things depending on different conditions.

→ In React,
you can conditionally render JSX using javascript syntax like :-

- if statement
- && operator
- ? : operator

→ Here, we need to render either a shimmer UI or a normal (data) UI.

Pseudocode :-

1. if (restaurant) is empty ⇒ Load Shimmer UI
2. if (restaurant) has data ⇒ load actual UI

① Make a Shimmer UI component

<Shimmer/>

▷ Before 'return', I can write some conditions.
ie, conditional rendering.

```
return (restaurants.length == 0) ? <Shimmer/> : (
  <>
    ...
  </>
);
```

After one search, the state got updated and again when we search keywords, it won't work. This is not working because 'restaurant' gets filtered out. Every time, we should not update this 'restaurant', so we will have to maintain 2 variables :-

Var 1 ⇒ List of all restaurant

Var 2 ⇒ List of filtered restaurand.

Whenever I'm filtering, I should filter from 'all restaurant'.

So, maintain 2 copies of the restaurant.

I show 'filtered restaurant' in my UI.

```
const [allRestaurants, setAllRestaurant] = useState([]);
const [filteredRestaurants, setFilteredRestaurants] = useState([]);
```

So, when I make an API call, I will populate my allrestaurants :-

```
setAllRestaurants(json?.data?.cards[2]?.data?.
    data?.cards);
```

→ So, for the first time, make :-

setAllRestaurants (json?.data?.cards[2].data?.data?.cards);

setFilteredRestaurants (json?.data?.cards[2].data?.data?.cards);

At first time, make filteredrestaurants as all.

→ In Search button, on clicking it, I'm passing 'restaurants' before.

Now, I have to search from my 'allRestaurants'

→ I want to show my Shimmer when my 'allRestaurants' length is 0.

Shimmer is only shown when we have no data

```
<button onClick={() => {
    const data = filterData(searchText, allRestaurants);
    setFilteredRestaurants(data);
}}
return allRestaurants?.length === 0 ? (
    <Shimmer/>) : ( <></> )
```

* How to avoid Rendering component ?

→ 1 way :- use optional chaining (allRest?.length===0)

→ 2nd way :- If allRestaurant is not there, return null. (means, It won't get rendered). or can return some JSX

→ EARLY RETURN

```
if(!allRestaurants) return null;
```

* Suppose if my filteredRestaurants is empty,

```
if (filteredRestaurants?.length===0) return (
  <h1> No matches </h1>; );
```

HW :-

1. Check for all Restaurants & show Shimmer
2. Then show wheather filteredRest are not there or there.
3. I should see "No matches" only when my filteredRest length is zero.

Watch Recording for :-

→ Case conversion :- 02:45:00

→ Login-Logout toggle : 02:47:00

→ Read about JS Expression in Docs.

Chap 07 - Finding the Path

Date : 15/Jan/2023

Ashraya KK
@ashrayaa

Note :

- Never create a component, inside a component
- You can compose the component
- Never write a useState inside an if..else. ~~loop~~ or for loop.
- useState is a hook that react gives to create local state variable inside a functional component. So, never use useState outside a functional component (it won't make sense).
- We can use more than one useEffect, according to the usecase.
- To store the images locally, create a folder 'assets' & images can be kept inside it.

H.W :- Why CDN is a greatplace to host images ?

↳ CDN is faster. It caches the image. It returns very fast and have 100% uptime. It optimises (imgs are already optimized when we put into CDN)

Eg :- Swiggy uses CDN.

SHIMMERUI

```
const Shimmer = () => {
    return (
        <div classname = "restaurant-List">
        { Array (10). fill (" "). map ((e, index)
            => ( <div classname = "shimmer card"
                key = {index} >
            </div>) )}
        </div>
    ); };
export default shimmer;
```

NB:

- When you're building an app, be concious about what packages are you using.
- You should use a big packages when you've complex things to do

Eg:- If you have to build a lot of big and lengthy forms in your app, in which each and every input box have their own RegEx check, pattern check, validation etc.

So, instead of building all the components on your own, we can use a package, (npm package), a library) known as :-

FORMIK. It's a open source library. You can use also React-Hook-Form instead of FORMIK.

ROUTING

- Finding the path of different pages of our app.
- For that we use a library or npm package called React Router.
- Install this package

```
npm i react-router-dom
```

- Make an 'About.js & Contact.js' pages.

To click the 'About' on homepage and to move to the About page, we first have to create a routing configuration.

- import {createBrowserRouter} from "react-router-dom";

This is a function we get from react-router-dom. It will help us create routing.

→ To create the routing, do :-

```
const appRouter = createBrowserRouter([
  {
    path : "/",
    element : <AppLayout/>,
  },
  {
    path : "/about",
    element : <About/>
  },
]);
```

Here, I will define where My app will get loaded with "/path".

This is where we create Routing configuration

[→ There are multiple Routers. Read Doc]

"createBrowserRouter" is the most recommended route for all React Router web projects.

→ This alone won't work, we need to provide this appRouter to our app.

For that there is a component RouterProvider which is coming from react-router dom. Import it.

→ And render this RouterProvider in your App.js.

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

→ 'react-router-dom' is a powerful library which ~~shows~~ gives us a better UI for showing us this error pages.

Create an error page of your own "Error.js" and pass this component to our router config.

```
{
  path: "/",
  element: <AppLayout />,
  errorElement: <Error />,
}
```

If there is an error in the path, it will load the error element.

→ To show more information about the error in the errorpage, 'react-router-dom' gives us {useRouteError}. Import this in Error.js.

```
import {useRouteError} from "react-router-dom"
```

→ { useRouteError } is a hook : which won't allow that red-colour error to come in consoles. It catch all the routing errors and we can show those error to the user.

Error.js

```
import { useRouteError } from "react-router-dom";

const Error = () => {
    Const err = useRouteError();
    const {
    return (
      <div>
        <h1>OOps! Something went wrong</h1>
        <h2>
          {err.status + ":" + err.statusText}
        </h2>
      </div>
    );
};

export default Error;
```

Problem with anchor tag

It will reload the entire page when it is clicked. It disrupts user experience and can result in a slower page load time. This cause problems for Single-page applications (SPAs)

Single Page Applications (SPA)

- React apps are SPAs
- Having SPAs will not reload, It will not make network call when we are changing pages
- Loads a single HTML page and dynamically update the page in response to user interaction without reloading the entire page.
- This approach allows for faster navigation and a more seamless user experience.

Two types of Routing

1. Client-side routing
2. Server-side routing

Server-Side Routing

- all our pages come from server.
- make a network call, get the html, js, css and loads the whole page.

Client-side Routing

→ dynamically update content of SPA in response to change in URL.

→ don't do full page reload.

→ 'React-router-dom' gives {Link}.

```
import {Link} from "react-router-dom";

<Link to = "/about" >
    <li> About </li>
</Link>
```

→ To keep Header & Footer stick on to every page, change the routing config.
ie, to make the about page children of <AppLayout /> do:

```
Const appRouter = createBrowserRouter([
  {
    path: "/"
    element : <AppLayout/>,
    errorelement : <Error/>
    children : [
      { path : "/about",
        element : <About/>,
      }, ] }, ]);
```

```
→ Const AppLayout = () => {
    return (
      <>
        <Header />
        <Outlet />
        <Footer />
      </>
    );
};
```

→ React-router-dom gives acess to Outlet.
This outlet will be filled by the configuration.
So, all the children will go inside Outlet.
acording to the route.

DYNAMIC ROUTING

→ process of rendering components in response to a change in the application's URL.

→ If the route to restaurant menu page is like

```
{
  path : "/restaurant/:id"
  element : <RestMenu />
}
```

id is dynamic (it can be anything).

→ To read the 'id' passed in URL,
'react-router-dom' gives us { useParams }.
It's the routing parameters.

<RestMenu/>

```
→ import {useParams} from "react-router-dom";

const RestMenu = () => {
  const params = useParams();
  const {id} = params;
  return (
    <div>
      <h1>Restaurant id : 1234</h1>   ← {id}
      <h2>Namaste</h2>
    </div>
  );
};
export default RestMenu;
```

→ We will be having the id inside params.

Making an API call in Restaurant Detail Page :

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

```
async function getRestaurantInfo() {
  const data = fetch("https://....." + id);
  const json = await data.json();
}
```

→ <RestMenu/> component, at last will be like 👇

*

```
import {useEffect, useState} from "react";
import { useParams } from "react-router-dom";
import {IMG_CDN_URL} from "../constants";
import Shimmer from "./Shimmer";

const RestaurantMenu = () => {
    const {resId} = useParams();
    const [restaurant, setRestaurant] = useState(null);
    useEffect(() => {
        getRestaurantInfo();
    }, []);
    async function getRestaurantInfo() {
        const data = await fetch(
          "https://www.url=" + resId );
        const json = await data.json();
        setRestaurant(json.data);
    }
```

```
return !restaurant ? (<Shimmer/>) : (
  <div>
    <h1> Restaurant id : {resId} </h1>
     <h2> {restaurant.name} </h2>
     <h3> {restaurant.area} </h3>
      <h3> {restaurant.city} </h3>
      <h3> {restaurant.avgRating} </h3>
    </div>
    <div>
     <h1> Menu </h1>
     <ul>
      {Object.values(
        restaurant?.menu?.items).
        .map((item) => (
          <li key={item.id}> {item.name} </li>
        ))}
     </ul>
    </div>
  );
};

export default RestaurantMenu;
```

CLASS BASED COMPONENTS

- It was less maintainable, have more code and a little messy.
- They are no longer used.
- Developers can do almost everything using functional components now.
- Functional component, at the eod is just normal js functions. Similary ↓
- Class based components are just normal js class.

CLASS BASED COMPONENT	FUNCTIONAL COMPONENT

`import React from "react";`
`class Profile extends React.component{`
`render(){`
`return <h1>Profile</h1>`
`}}`
`export default Profile;`
	`const Profile =()=>{`
	`return (`
	`<h1>Profile</h1>`
	`);`
	`}`
	`export default Profile`

ProfileClass.js	ProfileFunc.js.

React.component come from here.

You've to tell react that this is a class component not normal javascript class

name of the component

Keyword class

You can't create a class component without render method

```
import React from "react";

class Profile extends React.component {

  render(){ => return some JSX

    return <h1> Profile </h1>;
  }}

export default Profile;
```

Whatever you return from this will be injected to the DOM.

`render(){}` is the only mandatory method for class based components.

PROPS IN CLASS COMPONENTS

In About.js, Suppose we called class component "ProfileClass.js" as : ⇩ and if we pass props inside it :-

```
<ProfileClass name={"Ashraya"} />
```

So, in class component, we have 'this' keyword.

So, in 'ProfileClass.js' ⇩

```
<h2> Name : {this.props.name} </h2>
```

<ProfileClass name={"Ashraya"} xyz={"abc"}/>

Even if there are many props, react will collect all these props and attach with keyword 'this'. And I can use the props like :

```
<h2> Name : {this.props.name} </h2>
<h2> xyz : {this.props.xyz} </h2>
```

STATE IN CLASS COMPONENT

```
class Profile extends React.component {
    constructor (props); {
        super (props);
        this.state = {
            count : 0,
            count2 : 0,
        };
    }
```

super (props) ; → This is necessary because it passes the received props to the constructor of the base component class, allowing the component access it's properties.

constructor (props); { ... } → creates & initialize the component's state.

Constructor function is called only once, when the component is first created & initialised, making it an ideal place to set the initial state of the component using this.state = {..}.

- Whenever we load a class, constructor is called.
- To use the state we created :

```
render() {
  return (
    <h2> Count : {this.state.count} </h2>
    <h2> Count2: {this.state.count2} </h2>
  ); };
```

- setCount fn

We do not mutate state directly

```
render() {
  return (
    <h2> Count : {this.state.count} </h2>
    <button
      onClick = {() => {
        this.setState ({
          Count : 1, count2 = 2,
        })
      }) > SetCount </button>
```

React Lifecycle

→ first constructor is called

→ Then, component is rendered

React component lifecycle refers to the series of methods that get executed at different stages of a component's existence in React application.

The lifecycle methods can be divided into 3 phases :→

1. Mounting Phase
2. Updating Phase
3. Unmounting Phase

Mounting Phase

→ These methods are called when an instance of a component is being created & inserted into the DOM :

- constructor()
- static getDerivedStateFromProps()
- render()
- componentDidMount() → This method is called immediately after the first render of of a component. It is executed only once during the lifecycle of a component.

Updating Phase :

→ These methods are called when a component is updated in response to changes in its props or state :-

- **shouldComponentUpdate ()** -

  This method is called before a component is updated. It returns a boolean value indicating whether the component should be updated or not.

- **componentWillUpdate ()** -

  This method is called just before a component is updated. It is only executed if shouldComponentUpdate() is true

- **render ()** - Used to rended the component after it has been updated.

- **componentDidUpdate ()** - This method is called immedietly after a component is updated. It is only executed if ShouldComponentUpdate() is true.

Unmounting Phase

→ The phase where component is being removed from the DOM.

- componentWillUnmount() :- This method is called just before a component is removed from the DOM.

Best Place to make API call in class Components

→ componentDidMount () { . . }

This is because during mounting phase.

1) Constructor ( ) → is called

then, 2) render ( ) → is called.

atlast, 3) componentDidMount() → is called

Eg:- Class Apidatafetchexample extends React.Compment {

state = { data: [], loading: true, error: null };

componentDidMount () {

this.fetchData (); }

fetchData = () ⇒ {

fetch ("https://api.example/data")

```
.then(response => response.json())
.then(data => this.setstate({data, loading:false}));
. catch(data
. then(
. catch(error => this.setState({error,
                                loading: false}));
};
```

CORE - BASIC OF CLASS COMPONENT

About.js (Parent Component)

```
class About extends Component{
 constructor(props){
   super(props);
console.log("Parent-constructor");
 }
componentDidMount(){
console.log("Parent-componentDidMount");
 }
render(){
 console.log("Parent-render");
 return(
 <Profile />
 );}}
```

Profile.js (Child Component)

```
class Profile extends Component{
 constructor(props){
   super(props);
console.log("Child-constructor");
 }
componentDidMount(){
console.log("Child-componentDidMount");
 }
render(){
 console.log("Child-render");
 return(
    <h1>Profile class</h1>
 );
}}
```

—▷ In above eg.

<About /> is the Parent component

<Profile /> is the child component of <About />

—▷ In what order the above code will execute?

① Parent - Constructor

② Parent - render (In render() of About it sees <Profile /> and it will trigged the lifecycle method of this childeren component)

③ Child - constructor

④ Child - render

⑤ Child - componentDidMount

⑥ Parent - componentDidMount.

Another Case

If <About/> component have 2 children :→
First child & Second child.

Let's see how it will be executed :-

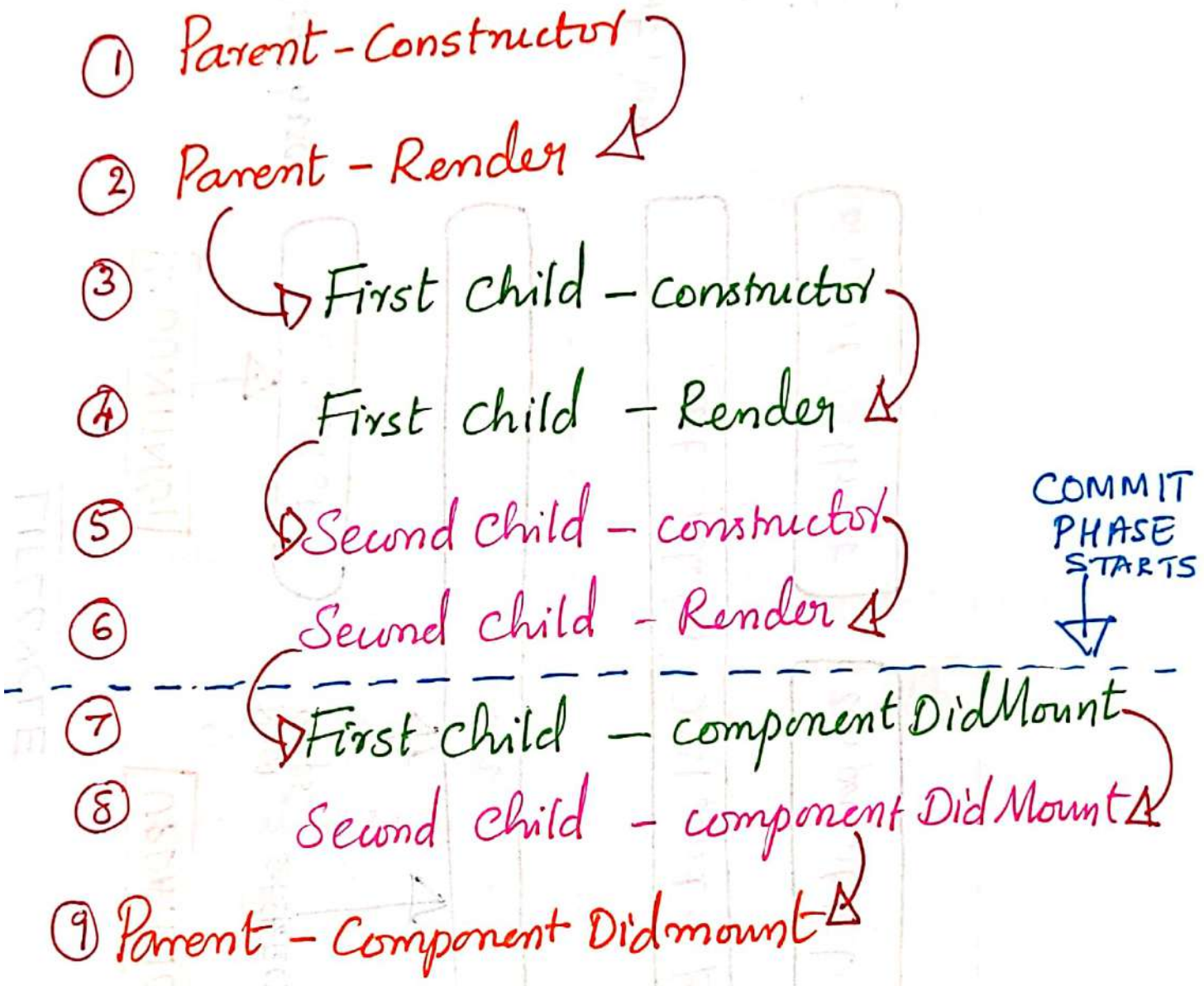
About.js

```
class About extends Component {
    constructor (props) {
        Super (props);
    console.log ("Parent-constructor");
    }
    componentDidMount () {
        console.log ("Parent- ~~render~~ componentDidMount");
    }
    render () {
        console.log ("Parent-render");
    return ( <>
    <Profile name = {"First child"} />
    <Profile name = {"Second child"} />
    </> ); }}
```

Profile class .js

```
class Profile extends React.component {
    constructor (props) {
        Super (props);
        this.state = {
            count : 0,
        };
    console.log ("Child - constructor " + this.props.name);
}
componentDidMount () {
    console.log ("Child - componentDidMount" +
                this.props.name);
}
render () {
    console.log ("Child - render" + this.props.name)
    return (
        <h1> Profile Class </h1>
    )
};
```

Order of Execution :-



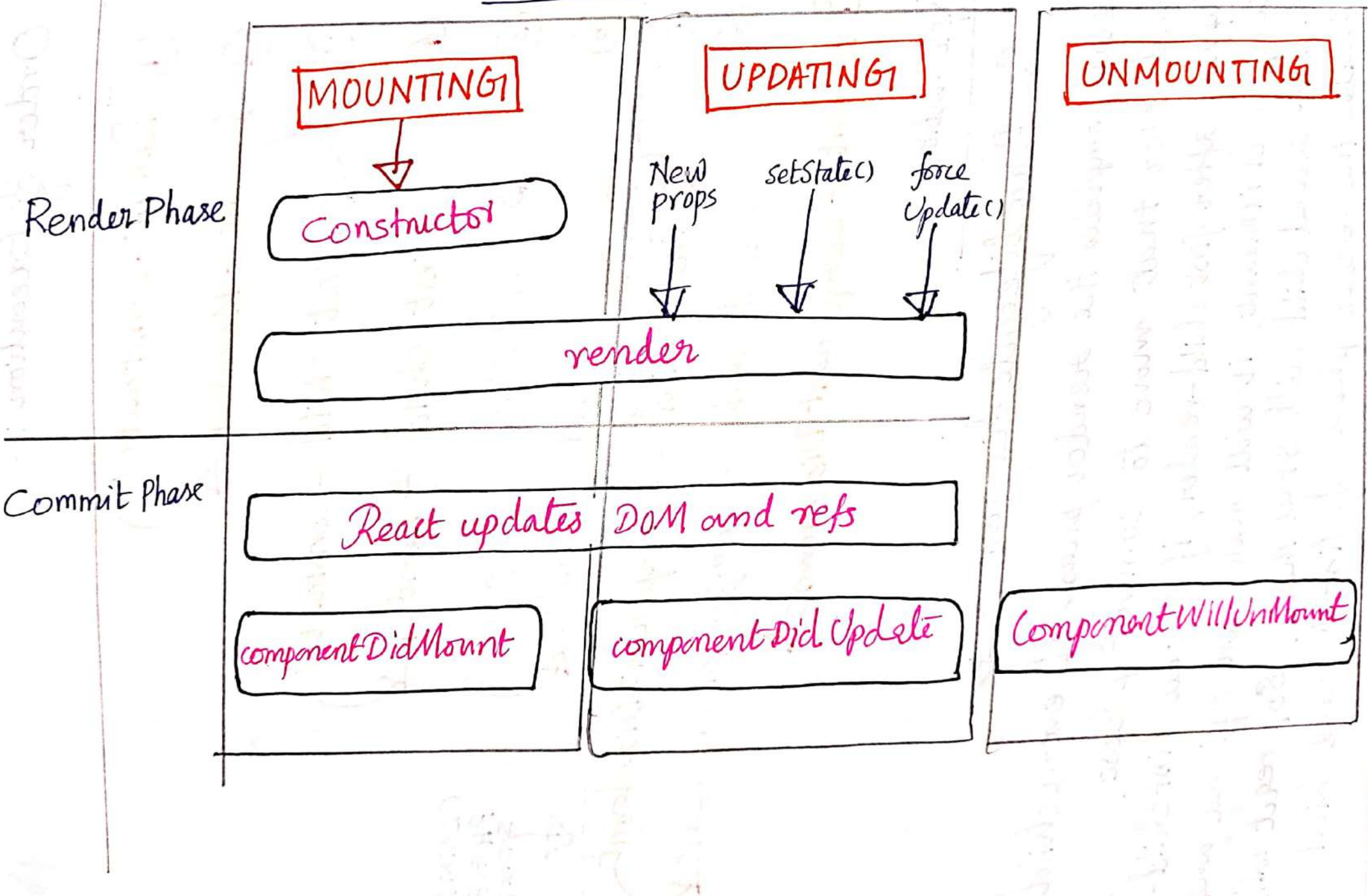
Explanation

See React lifecycle diagram ⟹

React completes the render phase of every child and after that move to commit phase.

Suppose after first child-render, if I call firstchild-componentDidMount, it will make an api call and my ~~fir~~ second child will stay there. So, react will batch the render phase of first & second child

LIFECYCLE



React do rendering in 2 phases :-

1. Render Phase
2. Commit Phase

First of all, react finishes the RENDER PHASE.

Render Phase : → is fast

→ includes constructor and render method.

Commit Phase

→ Phase where react modifies the DOM.

→ componentDidMount is called after the initial render has finished.

→ Commit Phase is slow.

Making an API Call

→ Let's use github user api.

→ make an api call in the child component

Profileclass.js :-

* class Profile extends React.component {

```
constructor(props) {
    super(props);
        this.state = {
            userInfo : {
                name : " ",
                location : " ",
            }};
    console.log("Child - Constructor");
}

async componentDidMount() {
    const data = await fetch("https://api-github..
                                .com/users/Ashrayaa");
    const json = await data.json();
    console.log(json);
    this.setstate({
        userInfo : json,
    });
    console.log("Child - componentDidMount");
}
```

```
render() {
  console.log("Child - render");
  return (
    <h1> Name : {this.state.userInfo?.name} </h1>
    <img src={this.state.userInfo?.avatar_url} </h1>
    <h2> Location: {this.state.userInfo.location}
                                          </h2>
```

Sequence of method called in above code

I have parent 'About.js' inside one child 'ProfileClass.js'.

1. Parent - Constructor
2. Parent - Render
3. Child - constructor
4. Child - render
5. ~~API call~~
5. Parent - componentDidMount } is called before making api call.

This is because React finishes render cycle first and then it goes to commit cycle. As Child - componentDidmount, will take some time for the data to load, Parent - componentDidMount is called before. So, hence this Sequence.

① Parent - constructor

② Parent render

③ Child constructor

④ child render

⑤ DOM is updated

⑥ json is logged in console

⑦ Parent - componentDidMount

⑧ Child - componentDidMount

It is called before but is been put into the wait cycle. Because we are using async.

⑨ Child - render

* setState trigger next render. It will trigger reconciliation process. So, the child will be rendered once again when we have the data.

This re-render cycle is known as "UPDATING"

* ComponentDidMount is called after first render.
* ComponentDidUpdate is called after every next render
* Before the component is unmounted from the DOM, ComponentWillUnmount will be called.

NB: Never compare React Lifecyclemethod with Functional components
In modern react code, they removed the concepts of lifecycle method.

ComponentDidUpdate

→ is called after every subsequent render.
→ In functional component, we use dependency array in useEffect which indicates when should useEffect should be called.

Eg:-
```
useEffect(() => {
    // API call
}, [count, count2]);
```

This means that whenever the count and count2 gets updated, UseEffect gets executed.

Earlier, in class-based components, this is done like below:- Which is hectic!

```
componentDidUpdate (prevProps, prevState) {
  if (
    this.state.count !== prevState.count ||
    this.state.count2 !== prevState.count2)
  {
    //API call.
    //code
  }
}
```