React Notes

#Basics:

* Emmet:

It basically generates some code for us inside VS Code.

Q. Create Hello World Program using only HTML

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <div id="root">
      <h1>Hello World! using only HTML</h1>
    </div>
  </body>
</html>
```

Q. Create Hello World Program using Javascript

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>

  <body>
    <div id="root"></div>
  </body>
  <script>
    const heading = document.createElement("h1");
    heading.innerHTML = "Hellow World! From Javascript";

    const divElement = document.getElementById("root");

    divElement.appendChild(heading);
  </script>
</html>

```


TanmayVaidya

Q. Create Hello World Program using React

```html
<body>
  <div id="root"></div>

  <script
    crossorigin
    src="https://unpkg.com/react@18/umd/react.development.js"
  ></script>
  <!--This file i.e. react.development.js contains the React Code which is plain JS-->

  <script
    crossorigin
    src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
  ></script>
  <!--This file i.e. react-dom.development.js is useful for DOM operations or use to modify the DOM-->

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

Content Delivery Networks (CDN) Links

* Crossorigin:

The crossorigin attribute in the script tag enables Cross-Origin Resource Sharing (CORS) for loading external JavaScript files from a different origin than the hosting web page. This allows the script to access resources from the server hosting the script, such as making HTTP requests or accessing data.

Q. What is {} denotes in above code?

```html
<div id="root">
  <h1 id="title">Hello World!</h1>
</div>
```

This (id='title'), classes, etc should come under {}. Whenever I'm passing inside {}, will go as tag attributes of h1.

* Note:

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

Q. Do the below HTML code in React.

```html
<div id="parent">
  <div id="child1">
    <h1>Heading 1</h1>
    <h2>Heading 2</h2>
  </div>
  <div id="child2">
    <h1>Heading 1</h1>
    <h2>Heading 2</h2>
  </div>
</div>
```

To build the structure like this in React

```jsx
const parent = React.createElement("div", { id: "parent" }, [
  React.createElement("div", { id: "child1" }, [
    React.createElement("h1", { id: "heading_1" }, "Heading 1"),
    React.createElement("h2", { id: "heading_2" }, "Heading 2"),
  ]),
  React.createElement("div", { id: "child2" }, [
    React.createElement("h1", { id: "heading_1" }, "Heading 1"),
    React.createElement("h2", { id: "heading_2" }, "Heading 2"),
  ]),
]);

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

root.render(parent);

```

Q. To make our app Production ready what should we do?

1. Minify our file (Remove console logs, bundle things up)
2. Need a server to run things

Minify —> Optimization —>Clean console —> Bundle

* Bundlers:

In React, to get external functionalities, we use Bundlers. It packages you app properly so it can be shipped to production.

Examples:

1. Webpack
2. vite
3. Parcel

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

* Package Manager:

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:

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

`npm init` (Creates a package.json file)

`npm install -D parcel` → Parcel is one of the dependency

↳ Dev dependency: because we want it in our development machine

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

* package-json:

It is a configuration for npm

* Caret sign(^):

If we use caret sign, then it will automatically update to the minor versions and patch versions.

* Tilde sign(~):

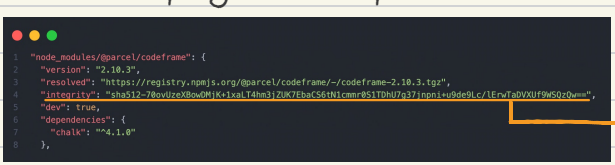
If we use tilde sign, then it will automatically update only to the patch versions.

* package-lock.json:

Will tell you which exact version of the library we are using.

Q. Common Issue: It is working on my local, how it break on production?

- To avoid this, package-lock.json keeps an hash (in integrity field) to verify that whatever is in my dev machine, is the same version deployed on the production.



Hash for the particular module

* node_modules:

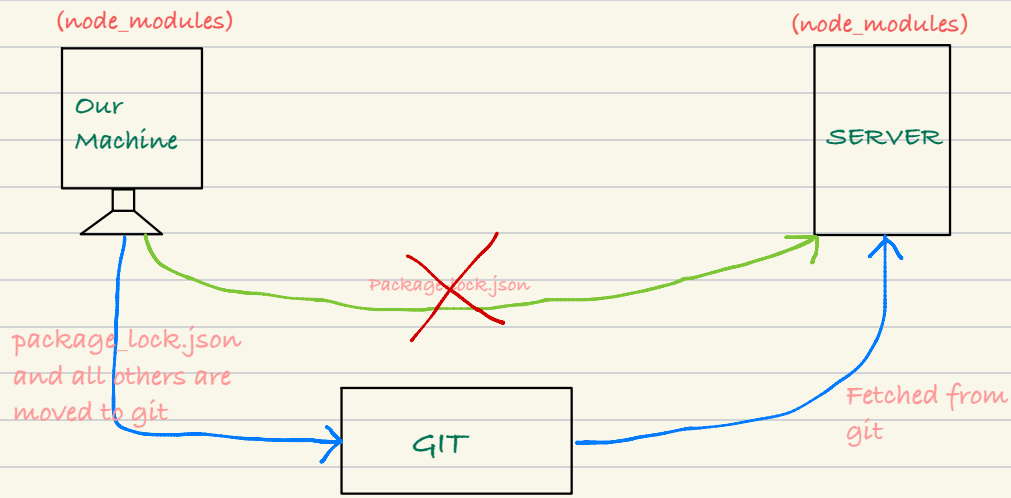
- Which gets installed is like a database for the npm.
- This is where super powers comes from.
- Our app has dependency on parcel
- parcel also has dependencies on something else.
- Every dependency in node_module will have its package.json

All these dependencies / superpowers are in node_modules

Q. We don't put the 'node_modules' into git. Why?

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

package-lock.json file keeps and maintains the version of everything in the node_modules.

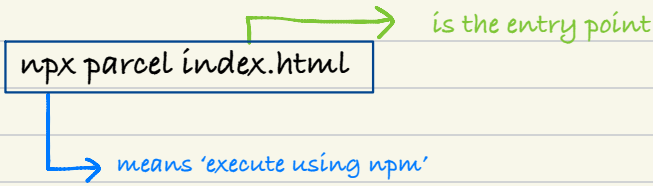


'node_modules' in our machine can be regenerated in server using the package-lock.json file.

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:



Then, a mini server is created for us:
Like localhost:1234

Parcel given a server to us.
"Parcel" ignited our app.

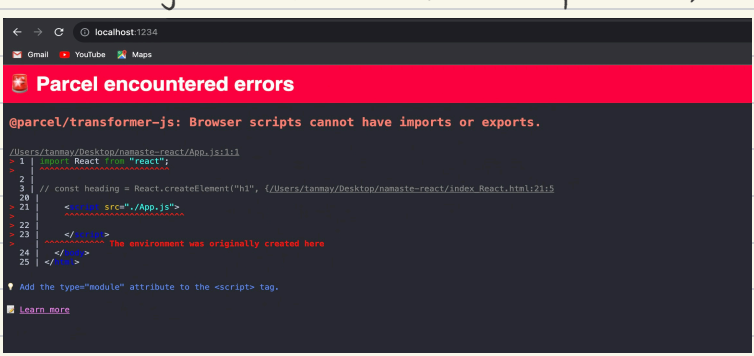
> Never touch 'node_modules' and 'package-lock.json'

* Import:

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

Common error:

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



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

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

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

* Hot Module Replacement (HMR):

- means that parcel will keep a track of all the files which you are updating .

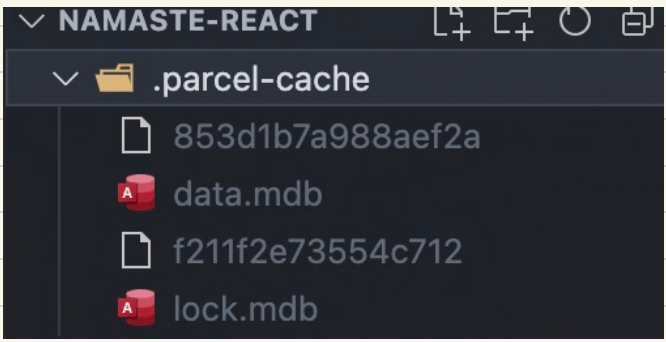
* How HMR Works?

- There is File Watcher Algorithm (written in C++). It keeps track of all the files which are changing realtime and it tells the server to reload.
- These are all done by PARCEL

* .parcel-cache:

- Parcel caches code all the time.
- When we run the application, a build is created which takes some time in ms.
- If we make any code changes and save the application, another build will be triggered which might take even less time than the previous build.
- This reduction of time is due to parcel cache.
- Parcel immediately loads the code from the cache every time there is a subsequent build.

- On the very first build parcel creates a folder .parcel-cache where it stores the caches in binary codeformat.
- Parcel gives faster build, faster developer experience because of caching.



* dist:
- It keeps the files minified for us.
- When bundler builds the app, the build goes into a folder called dist.
- The `/dist` folder contains the minimized and optimised version of the source code.
- Along with the minified code, the /dist folder also comprises of all the compiled modules that may or may not be used with other systems.
- When we run command:

```
npx parcel index.html
```

Tanmay Vaidya

- This will create a faster development version of our project and 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 the parcel will build all the production files to the dist folder.

Q. What takes a lot of time to load in a website?

Media, Images

* Parcel uses:

1. It does image optimisation
2. Parcel also does 'Caching while development'
3. Parcel also takes care of your older versions of browser
4. Sometimes we need to test our app on https, because 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
```

5. Parcel uses 'Consistent Hashing Algorithms'
6. Parcel is 'zero config'

*

Should put .parcel-cache in .gitignore

Because, anything which can be regenerated should be put inside gitignore

* Parcel features in a glance:

- Hot Module Replacement (HMR)
- File Watcher Algorithm - C++
- Bundling
- Minify Code
- Cleaning our code
- Dev and production build
- Super fast build algorithm
- Image Optimization
- Caching while development
- Compression
- Compatible with older browser versions
- Https on dev
- Image Optimization
- Port No
- Consistency Hashing Algorithm
- Zero Config
- Tree Shaking

* Transitive Dependencies:

- We have our package manager which takes care of our transitive dependencies of our code.
- If we've to build a production ready app which uses all optimisations (like minify, bundling, compression, etc), we need to do all these.
- But we can't do this alone, we need some dependencies on it. Those dependencies are also dependent on other dependencies.

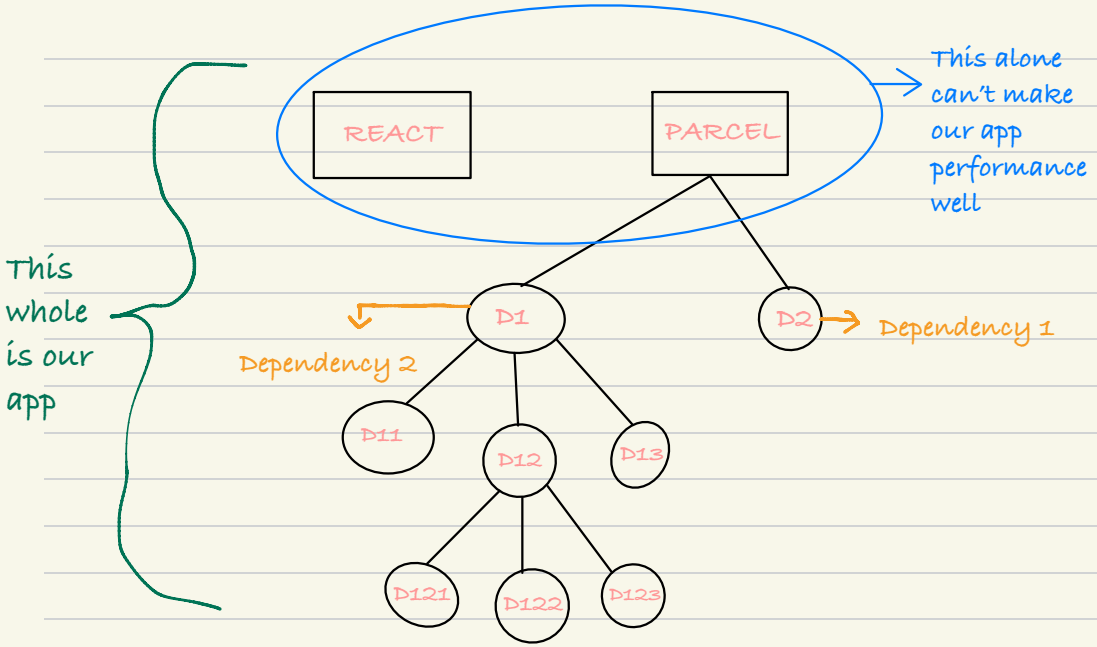


Fig: Our App (Dependency Tree)

Q. How do I make my app compatible with older browsers?

- There is a package called **browserslist** and parcel automatically gives it to us.

* Browserslist:

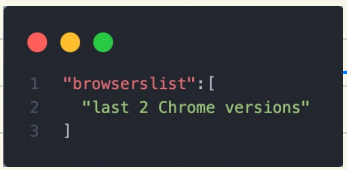
- It makes our code compatible for a lot of browsers.
- In package.json file do:

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

→ Support 74% of all the browers

Means my parcel 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:



→ Support 16%

* Tree Shaking:

- Parcel has this superpower.
- means removing unwanted code.

Example:

- Suppose your app is importing a library which has a lot of functions (say 20). Then, all those 20 functions 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:

- It uses web pack along with babel

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

- There are so many things that react optimises for us and parcels (bundlers) gives us.
- Our whole application is a combination of all these things.

Tanmayvaidya

* Polyfill:

- 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.

Example:

a. ES6 is the newer version of JavaSript. If I'm working on 1999 browser, my browser will not understand what is this const, new Promise, etc.

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

- So this is what happens when we write browserslist, our code is converted to older one.

* Babel:

- It 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.

* Scripts in package.json:

Run Command:

- To npx 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": {
  "test": "jest",
  "start":"parcel index.html"
},
```

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

`npm run start` or `npm start`

- start script will execute this command.

Build Command:

- To build our app, command is:

`npx parcel build index.html`

Package.json:

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

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

`npm run build`

* Note:

npx = npm run

* Remove console logs automatically:

- Console logs are not removed automatically by parcel. You have to configure you projects to remove it.
- There is a package to remove console logs.

Babel-plugin-transform-remove-console

For configuration

- Before installing the package, create a folder .babelrc
- 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 minimising the amount of work that needs to be done to update the changes.
- So, you don't have to worry about changes on every update.

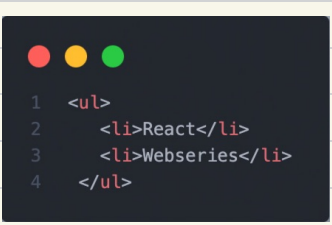
Example:

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

* render()

- Creates a tree of React elements.
- On the next state or props update, render() function will return a different tree of React Elements.
- Whenever react is updating the DOM,

Example:



Tanmayvaidya

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

```
<ul>
  <li>Namaste</li>
  <li>React</li>
  <li>Webseries</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 the children in the original tree with children in subsequent tree.
- Thus, making tree-conversion efficient.

```
<ul>
  <li key="2014">Namaste</li>
  <li key="2015">React</li>
  <li key="2016">Webseries</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

* JavaScript XML (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.

```
const heading = React.createElement("h1", {
    id:"title", key:"h1"
}, "Hello World!");

```

Instead of writing all the above code

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

This is JSX

- JSX is not 'HTML inside JavaScript'
- JSX has 'HTML like syntax'
- React keeps track of keys

- Our browser cannot understand JSX. Babel understands this code.
- JSX uses React.createElement() behind the scenes.

JSX => React.createElement() => object => HTML DOM

- Babel converts JSX to React.createElement()
- JSX is created to empower React.

* Advantages of JSX:
  - Developer Experience
  - Syntactical Sugar
  - Readibilty
  - Less code
  - Maintainability
  - No Repitition

Babel comes along with Parcel

Tanmayvaidya

* Components:

- Everything is a component in React.

* React Components:

2 types

- Functional Components - NEW way
- Class Based Components - OLD way

* Functional Components:

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

Example:

```
const HeadingComponent = () => {
  return (
    <div id="container">
      <h1>Namaste React</h1>
    </div>
  );
};
```

- For any component, Name starts with capital letter. (It is not mandatory but it's a convention)
- To render functional component write <HeaderComponent />

React Element:

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

→ React Element is an object

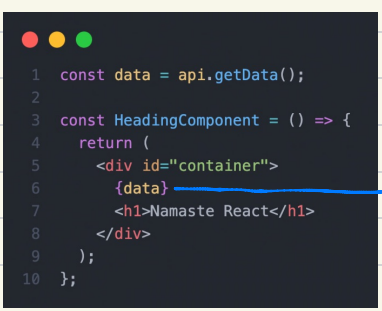
Functional Component:

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

Functional Component is a function

Important points:

- Whenever you write jsx, you can write any piece of javascript code between Paranthesis {}. It will work.
- JSX is very secure.
- JSX makes sure your app is safe.
- It does samitization.



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

const HeadingComponent = () => {
  return (
    <div id="container">
      {data}
      <h1>Namaste React</h1>
    </div>
  );
};
```

Write anything inside {}, Jsx will sanitise the code

* Component Composition:

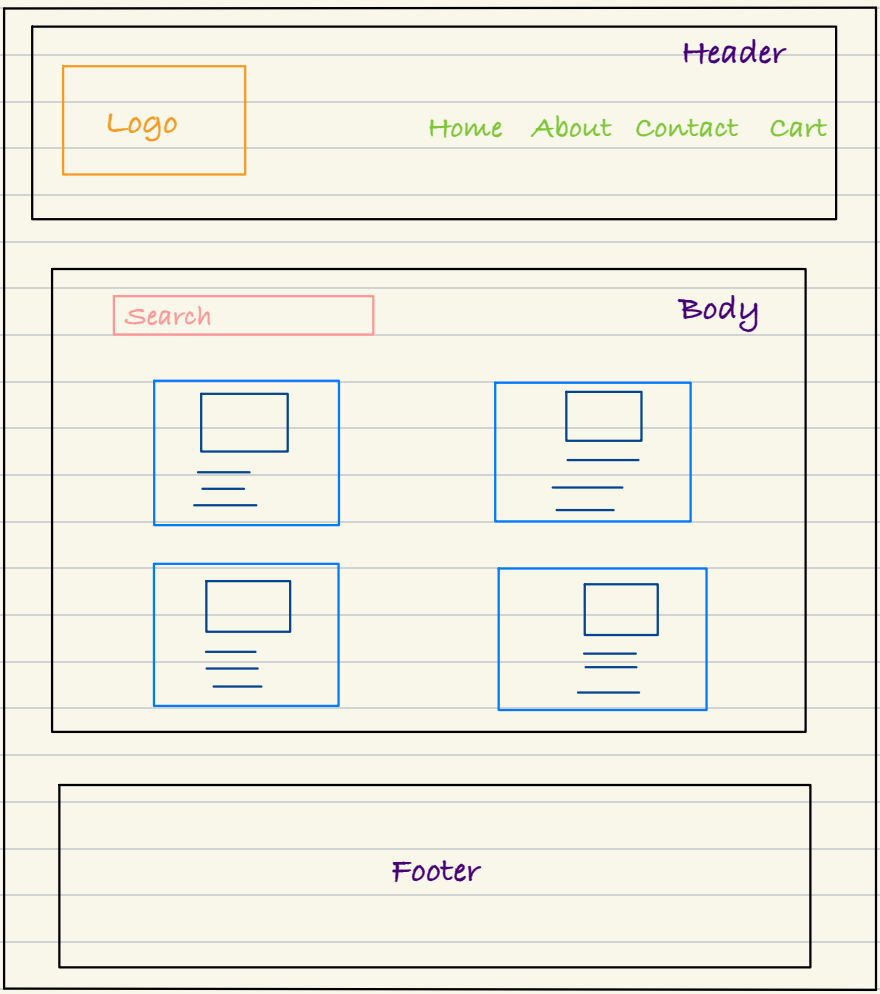
- If I have to use a component inside a component. Then it is called component composition / composing components.

3 ways of component composition:

- {Title()}
- <Title /> (used generally)
- <Title></Title>

#Building our First App (Food Delivery App):

- Basic Layout of App



- So the App layout should have:
  a. Header
  b. Body
  c. Footer

```
const AppLayout =()=>{
  return(
    - Header
      - Logo
      - Nav Items
    - Body
      - Search
      - RestaurantContainer
      - RestaurantCard
        - Img
        - Rating
        - cuisine
        - Delivery time
      -Footer
        - Copyright
        - Links
        - Address
        - Contact
  )
}
```

* Header Component:

```jsx
const Header = () => {
  return (
    <div className="flex justify-between border-b-2 border-black ">
      <div className="items-center flex">
        <img
          src={logo}
          className=" w-36 h-24 items-center cursor-pointer ml-8"
        />
      </div>

      <div>
        <ul className="flex p-4 m-4 justify-between">
          <li className="m-4 cursor-pointer">Home</li>
          <li className="m-4 cursor-pointer">About us</li>
          <li className="m-4 cursor-pointer">Contact Us</li>
          <li className="m-4 cursor-pointer">Cart</li>
        </ul>
      </div>
    </div>
  );
};
```

Note:

JSX expressions must have one parent element.

Tanmayvaidya

* React Fragment:

- It is a component which is exported by 'React'.
  [import React from "react"]
- Groups list of children without adding extra nodes to the DOM.

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

- Shorthand syntax <> </> is used generally instead of <React.Fragment> </React.Fragment>
- But, you cannot pass styles to empty brackets.

Styling inside React:

- Using Javascript Object.
- giving class name to the respective tag and write the css inside .css file.
- using external libraries like 'Tailwind CSS', 'Bootstrap', 'Material UI', etc.

Q. Can I use 'React.Fragment ' inside my 'React.Fragment'?

Yes

Child B and C are siblings and will be grouped together without adding an extra node to dom

```jsx
const Test = () => {
  <>
    <ChildA />
    <>
      <ChildB />
      <ChildC />
    </>
    <ChildD />
  </>;
};
```

Note:

In real world, data coming from api comes as "Array of Objects"

* Config Driven UI:

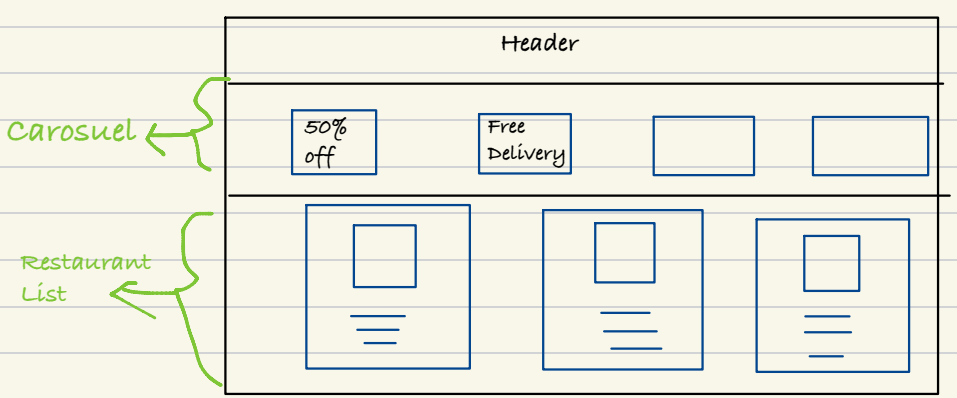
- All the UI (let say, Swiggy homepage) is driven by a config which is send by backend (api).

Example:

- Swiggy operates in various locations, including Kolkata and Delhi, and displays website information tailored to each specific location.
- This is achieved through a Config-Driven UI, where the backend controls the front-end configuration, allowing for targeted content delivery based on the user's location.
- This system ensures that users see relevant information, such as restaurants, promotions, and details specific to their location, resulting in a personalized and convenient experience.

How we design Config Driven UI:

- Suppose our UI is something like this:



- If my config is coming from backend, my data will come like:

```
const config = [
  {
    type: "carosuel",
    cards: [
      {
        offerName: "50% off",
      },
      {
        offerName: "Free Delivery",
      },
    ],
  },
  {
    type: "restaurants",
    cards: [
      {
        name: "Burger King",
        image: "https://someUrl",
        cuisines: ["Burger", "American"],
        rating: "4.2",
      },
      {
        offerName: "KFC",
        image: "https://someUrl",
        cuisines: ["Burger", "American"],
        rating: "4.2",
      },
    ],
  },
];
```

These offers are different for different locations

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

Only backend will change offers and website will render accordingly

* Optional Chaining:

- Allows us to access an object's properties without having to check if the object or its properties exist.
- represented by ? operator.
- Introduced in JavaScript ES 2020.

Example:

```
restaurantList[0].data?.name
```

Tanmayvaidya

* Props:

- shorthand for properties
- When we say "pass props," we mean we're transferring data or specific features into our React components. Just like passing information to a friend, we're sending data to our components to define their behavior and characteristics.
- resData = {restaurantList[0]}

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

```jsx
const RestaurantCard = (props) => {

  return (
    <div className="res-card m-4 bg-slate-200 w-52 h-auto cursor-pointer hover:border-black hover:border-2">
      <div className="pl-4 pt-4 pb-4">
        <h3 className=" font-semibold">{props.resData.name}</h3>
        <h3>{props.resData.cuisines.join(", ")}</h3>
        <h3>{props.resData.avgRating} stars</h3>
        <h3>{props.resData.costForTwo}</h3>
        <h3>{props.resData.sla.deliveryTime} minutes</h3>
      </div>
    </div>
  );
};
```

* Object Destructuring:

- It unpacks specific properties from an object into individual variables, streamlining data access and making code cleaner.

```jsx
const RestaurantCard = (props) => {
  const { resData } = props;

  const { name, cuisines, avgRating, costForTwo, sla, cloudinaryImageId } =
    resData?.info;
```

* Spread Operator (...):

- It unpacks iterables like arrays and objects, spreading their elements wherever it's used.

Example:

- Combine arrays: [...a, ...b] joins a and b.
- Add elements: [...a, "new"] puts "new" at the start of a.
- Pass function arguments: func(...a) expands a as individual args.

* Virtual DOM:

- The Virtual DOM is React's secret weapon! Imagine it as a lightweight, in-memory snapshot of your webpage's UI.
- When changes happen, React updates this snapshot first, then efficiently figures out the minimal edits needed to bring the real DOM in sync.
- We need Virtual DOM for Reconciliation.

* Reconciliation:

- It takes the updated Virtual DOM and figures out the most efficient way to bring the real DOM in line.
- It uses Diff Algorithm and it determines 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:

- React Fiber is a complete rewrite of React's internal reconciliation engine, built for speed and flexibility.
- It is responsible for Diff Algorithm.
- So, Reconciliation is the "what" of updating the DOM, while React Fiber is the "how" it's done in a more efficient and flexible way.

* React File Structure:

- React itself doesn't dictate a specific file structure, but common best practices help keep your code organized and maintainable.

* Export:

- There are 2 types of exports commonly used:

1. Named Export:

- Exports multiple values with custom names, imported with specific names

(e.g., `export const add = (a, b) => a + b;
import { add } from "./math";`).

2. Default Export:

- Exports only one value, imported with name

(e.g., ` Hello = () => { ... };
export default Hello;
import Hello from "./greetings";`).

TanmayVaidya

* Config File:

- A config.js file acts as the central hub for configuring different aspects of your application or project.
- It stores and manages essential settings like API keys, environment variables, theme preferences, and other configurations needed for your code to run properly.

Example:

- Config file will look like this and it should also have named export.

```
export const CDN_URL = 'https://someurl';
```

- Then import will look like:

```
import {CDN_URL} from "./config"
```

- This improves maintainability, flexibility, and collaboration, especially when working with teams or sharing configurations across environments.

* Building Search functionality:

```jsx
const Body = () => {

  return  (
    <div className="body">
      <div className="m-4 filter flex">
        <div className="search">
          <input
            className="boder-black border-2 py-1 px-4 mx-4"
            type="text"
            value=" "
          ></input>
          <button
            className="bg-orange-500 hover:bg-orange-700 text-white font-bold py-2 px-4 rounded mr-5"
          >
            Search
          </button>
        </div>
      </div>
    </div>
  );
};

export default Body;
```

- We've got search input and search button with us.
- But is I try to write inside my input box, its not working (because it is controlled by react).
- Same code if written in HTML file would work.

* One-way Data Binding in React:

```jsx
const Body = () => {
  const searchText = "KFC";

  return (
    <div className="body">
      <div className="m-4 filter flex">
        <div className="search">
          <input
            className="boder-black border-2 py-1 px-4 mx-4"
            type="text"
            value={searchText}
          ></input>
          <button className="bg-orange-500 hover:bg-orange-700 text-white font-bold py-2 px-4 rounded mr-5">
            Search
          </button>
        </div>
      </div>
    </div>
  );
};

export default Body;
```

I have a variable 'searchText' and if I put that here

Then the value "KFC" will go inside my input box

- I will not be able to edit the "KFC" value because it is hardcoded.
- To change the value in the input box, we need to modify the variable "searchText".
- But if I write anything in input box, it won't change the 'searchText'.
- This is called One-Way Data Binding.
- In Simple words, it can be explained as:

> "changes in the data automatically update the UI, but changes in the UI do not automatically update the data."

Q. How will I change the value of 'searchText'?

- Using onChange method.

onChange = {(e)=>onChangeInput (or e.target.value)}

- onChange method takes a function (which is a call back function) having an e event.
- So whenever an 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 is like a state variable

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

- Everytime, you have to create a local variable, you can use state in it.
- In react, if I want to create a local variable like 'searchText', I will create it using useState Hook.

* Hooks:

- Hooks are normal functions.
- React Hooks are like tools you can "hook into" React features without needing a class component.
- Hooks bring the power of classes to function components, making your React code simpler and more flexible.

* useState Hook:

- Its used to create state variables.

```
const [searchText] = useState()
```

a. This function returns an array.
b. The first element of this array is variable name
c. Second Element is a set function to update the variable

- searchText is a local state variable.
- To give a default value to my useState variable we do this:

TanmayVaidya

```
const [searchText] = useState("KFC")
```

- In React, to modify the variable 'searchText', I have to use a function.
- useState() gives us that function.
- Let us call that function as setSearchText()

```
const [searchText, setSearchText] = useState("");
```

- So our onChange function will be now:

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

From this event property I can read what I'm typing

- So our Body Component will now updated as:

```
const Body = () => {
  const [searchText, setSearchText] = useState("");

  return (
    <div className="body">
      <div className="m-4 filter flex">
        <div className="search">
          <input
            className="boder-black border-2 py-1 px-4 mx-4"
            type="text"
            value={searchText}
            onChange={(e) => {
              setSearchText(e.target.value);
            }}
          ></input>
          <button className="bg-orange-500 hover:bg-orange-700 text-white font-bold py-2 px-4 rounded mr-5">
            Search
          </button>
        </div>
      </div>
    </div>
  );
};
```

* Two way Data Binding:

- Two-way data binding allows bidirectional data flow, meaning that changes in the UI automatically update the component's state, and changes in the state automatically update the UI.
- So above here is Two way binding as I am reading as well as writing searchText.

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

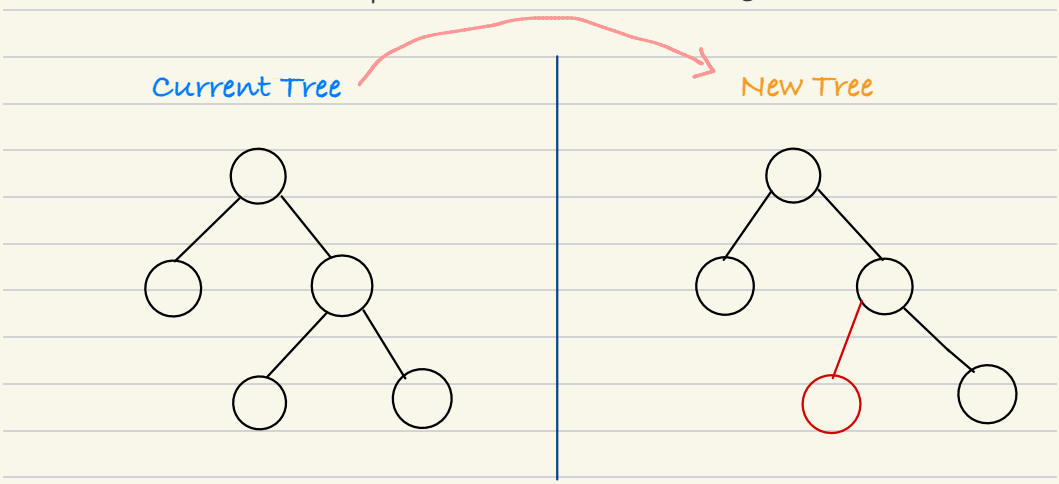
- Because, react has no idea what's happening to you local variables.
- So react, won't re-render on any updates happening on that variable.
- We need every time, the variable needs to be in sync with the UI. So for that we need to use State variables.
- React keeps a track of state variables.

Q. So what does React do by keeping track of them?

- Whenever, my state variable is updated, my whole component (here Body) re-renders i.e. React destroys the 'Body' Component and create it again.
- Reconciliation (Diff. Algorithm) is happening behind the scenes.

Q. 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 algorithm.



- React is fast because of its fast DOM Manipulation.
- Diff algorithm detects what exactly got changed in the page and it will just change that while re-rendering the whole tree.

Q. Explain useState Hook in short with example.

- React gives a state variable and a function that updates state variable.

```
const [title, setTitle] = useState("hi");

return (
  <div className="body">
    <h1>{title}</h1>
    <button
      onClick={() => {
        setTitle("Swiggy");
      }}
    >
      Click Here!
    </button>
  </div>
);
};
```

- React keeps the track of state variable title, once it is created.
- On click of the button 'Click Here!', the title gets updated from 'hi' to 'Swiggy'.
- The whole UI will re-render and it will update the UI quickly.

Note:

- Whenever state variable updates, react triggers a reconciliation cycle (re-renders the component).

Tanmayvaidya

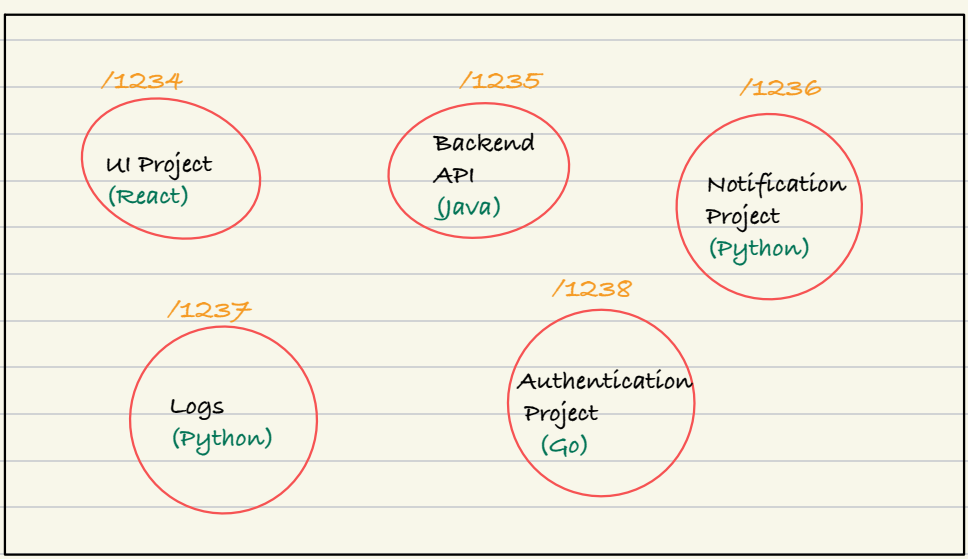
#Microservices:

* Monolith:

- In older days, there used to be a single big application.
- So everything like Apis, SMS, Notifications, UI, JSP pages etc used to be in the same project.
- Suppose if we have to change one button, we used to deploy this whole project / application. It was such a mess.
- This architecture was known a Monolith.

* Separation of Concern:

- But now, instead of having a one big project, we have small different projects.
- Here, separation of concern or single responsibility principle is there.
- The tools and languages used in a projects depends on the usecase .
- All these projects are deployed in different ports but same domain name.



Q. How to make an API call?

- JavaScript gives us fetch() function.

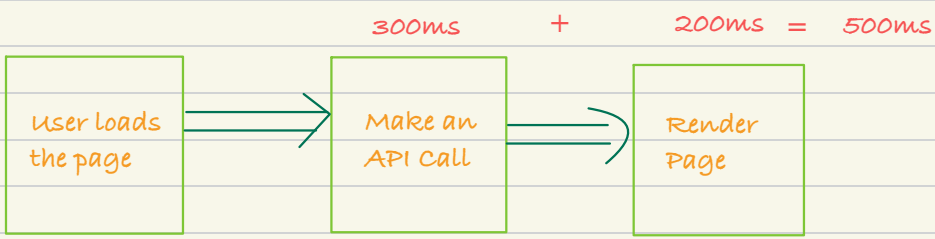
Q. Can we call API anywhere in our component?

- We can't just call API anywhere within our component. Because every time our states and props change, every time UI is updated, component re-renders.
- Every time our component re-renders, API gets called which is a big performance hit. This is not a good way.

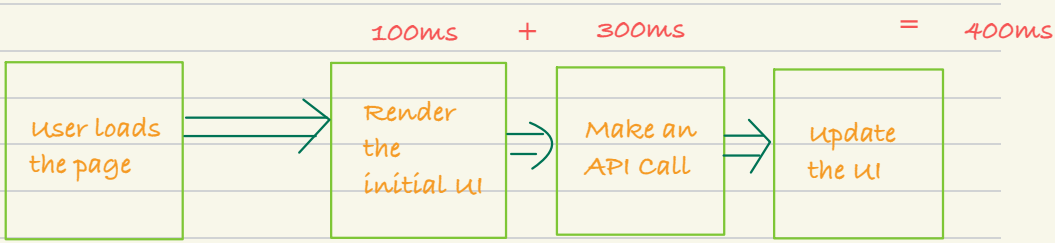
* Best way to call an API:

- There are two ways we can call API:

Case 1:



Case 2:



- In way 1, page will be available to us in (300 + 200 = 500) ms (milli seconds).
- In way 2, Page will be available to us in 100ms which is lesser than the first approach.
- So, Way 2 is a good way of calling API because of the user Experience.

- What we want is on initial page load we want our API to be called just once.
- To make this functionality happen React gives us access to the second most important hook known as useEffect hook.

* useEffect Hook:
- useEffect is a hook given by react which runs after a component renders and re-renders in DOM provided the hook must be placed inside that component.
- useEffect hook expects two parameters. First parameter is a callback function and second one is a dependency array.

useEffect(callback function, dependency array)

- The callback function will not be called immediately, but it gets called when useEffect wants it to be called.
- When a component renders or re-renders, useEffect calls its callback function which is the default behaviour of useEffect hook.
- We can control this action by providing a dependency array. If dependency array is empty useEffect calls its callback just once which is after component's initial render but when the component re-renders useEffect never calls its call back.
- For every change in states and props component re-renders.

- Once we provide a state variable to the dependency array, useEffect calls its call back function after component's initial render and subsequent renders in response to the state variable change.

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

- We make API call inside the callback function of useEffect and provide an empty dependency array to the hook so that API will be called just once.

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

- The dependency array is optional. If we don't provide the dependency array useEffect will not be dependent on anything. So, every time component is rendered useEffect would be called.

```
useEffect(()=>{})
```

* Role of Dependency array with example:

i.

```
const [searchText, setSearchText] = useState("");

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

- Suppose, I want to call this useEffect only when the searchText changes, then I have to pass the searchText in the dependency array.

ii.

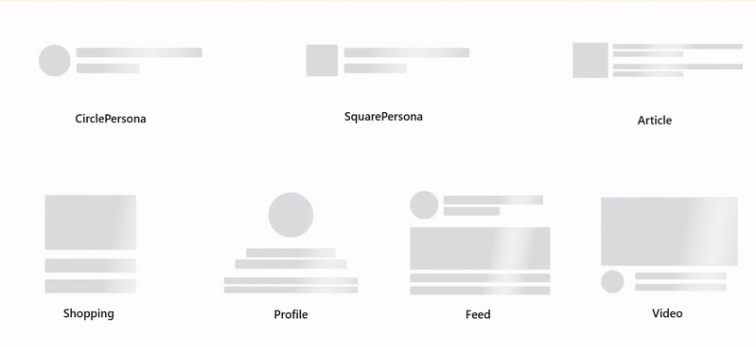
```
useEffect(() => {
  fetchData();
}, []);

const fetchData = async () => {
  const data = await fetch(
    "https://www.swiggy.com/dapi/restaurants/list/v5?lat=18.5204303&lng=73.8567437&is-seo-homepage-enabled=true&page_type=DESKTOP_WEB_LISTING"
  );

  const json = await data.json();

  setListOfRestaurants(
    json?.data?.cards[1]?.card?.card?.gridElements?.infoWithStyle?.restaurants
  );
  setFilteredRestaurants(
    json?.data?.cards[1]?.card?.card?.gridElements?.infoWithStyle?.restaurants
  );
};
```

- In this code, API will be called once after the component renders for the first time because in useEffect hook, dependency array is empty.

- The old data would be rendered for a first 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 basic loader / shimmer ui. This is a basic skeleton.

* 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 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 condition operator) work in the JavaScript.
- Rendering components based on a given condition check is known as Conditional Rendering.
- In React, you can conditionally render JSX using Javascript syntax like:
    - if statement
    - & & operator
    - ? : operator

Tanmayvaidya

Example:

- In below example we are showing shimmer to the UI when restaurant list data is not available.
- When the component is rendering, listofRestaurants state variable is initialised with empty array [] and as per conditional rendering logic we are showing shimmer in the UI.
- When listofRestaurants state is changed Body component re-renders and as per conditional rendering logic, the listofRestaurants state variable will have data this time which displays list of restaurants in the UI.

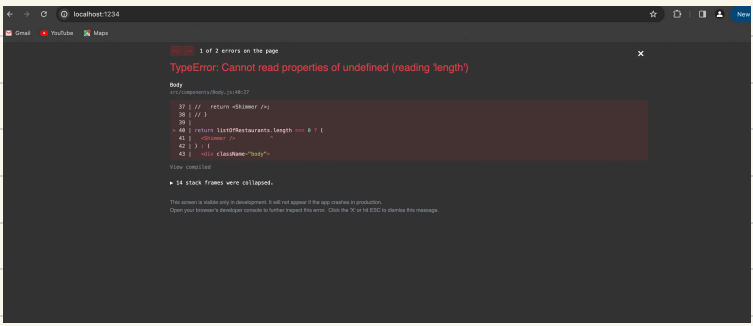
Pseudo Code:

- if (restaurant) is empty => Load Shimmer UI
- if (restaurant) has data => Load Actual UI

```
return listOfRestaurants?.length === 0 ? <Shimmer /> : <></>;
```

Q. How to avoid rendering a component when data is not available?

- When listOfRestaurants is undefined and we can't get access to undefined length property. Thus, in UI we get below error



- There are two ways to avoid such errors:

i. Optional Chaining:

```
return listOfRestaurants?.length === 0 ? <Shimmer /> : <></>;
```

ii. Early Return:

- Before rendering the shimmer component, we are doing early return so that if listOfRestaurants is not available or undefined, return null or a piece of valid JSX.

```
if (!listOfRestaurants) {
  return null;
}

return listOfRestaurants.length === 0 ? <Shimmer /> : <></>;
```

→ Early Return

- Suppose if my filteredRestaurants is empty,

```
if(filteredRestaurants?.length ===0){
  <h1>No Matches!!</h1>
}
```

TanmayVaidya

#Important Notes about React:

i. Never create a component inside a component:

- Because If we create a component inside a component, every time parent component renders or re-renders, the child component will be created which is not memory efficient.

ii. Never create useState variables inside if else block:

- Because If we define UseState variables inside if-else block and if the block does not get executed React will not be able to know whether the state variable really exist or not.
- React will not be able to trace that variable and React does not like inconsistency. React should know what your component is doing.

iii. Never create useState variables inside a for loop:

- Because If we do, it will create state variables as many times as the number of iteration. One state variable was needed, many were created.

iv. Never create useState variables outside of a function component:

- Because if we do, state variables will not be a part of that component.

v. We can use more than one useEffect, according to the use case.

vi. To store images locally, create a folder assets and images can be kept inside it.

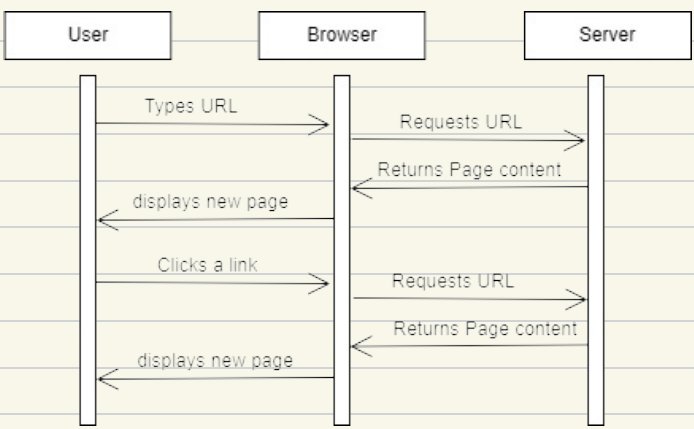
TanmayVaidya

#Routing:

- There are two types of Routing:
  - Server Side Routing
  - Client Side Routing

* Server Side Routing:

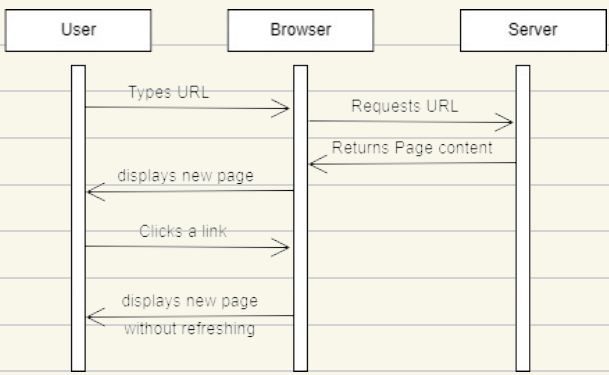
- Server-side routing is the traditional approach to handling routes in web applications.
- The process involves requesting a new page from the server and providing it to the user every time a link is clicked.
- One major issue with server-side routing is that every request result in a full-page refresh, which is not performance efficient in most cases.



TanmayVaidya

* Client Side Routing:

- Client-side routing involves JavaScript which handles the routing process internally.
- In client-side routing, a request to the server is prevented when a user clicks a link, hence no page refresh even when the URL changes.
- Because all of the available data and components are already present in the local machine.



* Single Page Application (SPA):

- In older days, when we used to navigate from pages to pages network calls were made, data were loaded from the server and then displayed in UI.
- Basically, we were reloading the entire page on every network call.
- But with SPA, our application does not reload for every action that user takes.
- SPA does not make network calls, when we navigate from pages to pages or some portion of the pages.
- It 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 experience

Q. Why CDN is a great place to host images?

- CDN is faster
- It caches the images
- It returns very fast and has 100% uptime
- It optimizes (imgs are already optimized when we put into CDN)
- Big applications like Swiggy, Zomato uses CDN to host images.

* Note:

- When you are building an app, be conscious about what packages are you using.
- You should use big packages when you have complex things to do.

Example:

- 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 Reg. Exp. check, pattern check, validation etc.
- So, instead of building all components on your own, we can use a npm package (or library) known as Formik.
- You can use React-Hook-Form also instead of Formik.

* Formik:

- Formik simplifies building forms in React.
- Forget managing state, validation, and submission manually.
- With Formik, you just define your form inputs and it handles the rest: keeping track of values, showing errors, and submitting data smoothly.
- Think of it as your friendly form-building assistant!

* Routing in React:

- 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 and 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"

- * createBrowserRoute:
  - This is a function that we get from react-router-dom.
  - createBrowserRouter function takes an array of router configuration objects of which each object has their own configuration set.
  - We provide path and element to each configuration object so that when we navigate to the path, component or element associated with that path will be rendered in UI.
  - createBrowserRouter creates a router for us.
  - There are multiple Routers in react. (Can read about them in docs).
  - createBrowserRoute is the most recommended route for all the React Router web projects.

- To create routing, do:

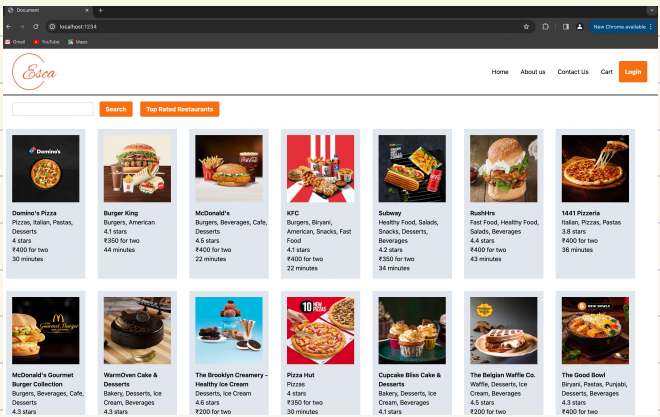
```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout />,
  },
  {
    path: "/about",
    element: <About />,
  },
  {
    path: "/contact",
    element: <Contact />,
  },
]);
```

- Creating a router will not be enough, we need to provide this router to our application so that our app can use it for routing.
- So we need to provide this appRouter to our app.
- For this to happen RouterProvider is the component that we import from react-router-dom.
- Now, instead of rendering a component inside render function, we provide RouterProvider component with a prop routing configuration array as an argument to render function which will render component as per the route configuration.

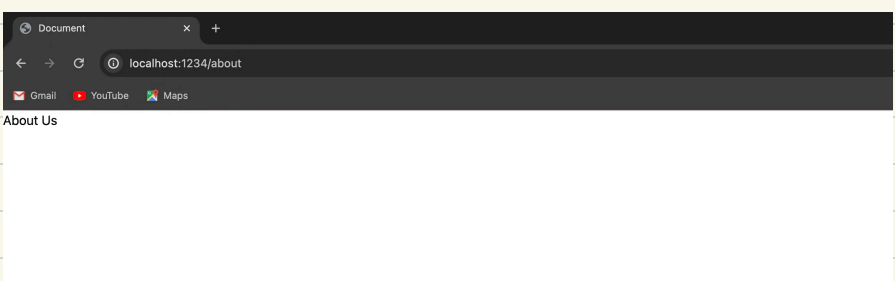
```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout />,
  },
  {
    path: "/about",
    element: <About />,
  },
  {
    path: "/contact",
    element: <Contact />,
  },
]);

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

Rendering RouterProvider

- Now if we navigate to the base URL http://localhost:1234/ we get AppLayout component page



- If we navigate to about page URL http://localhost:1234/about we get about component page

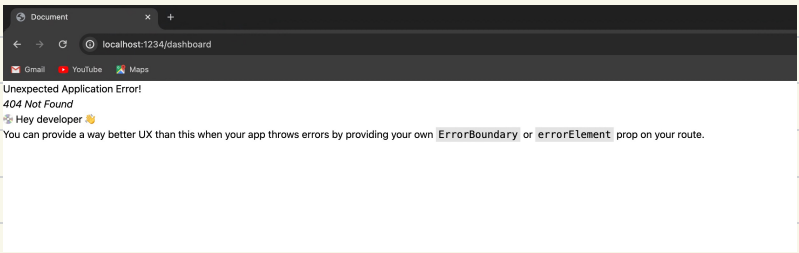


- This way we can achieve routing in react.

Tanmay Vaidya

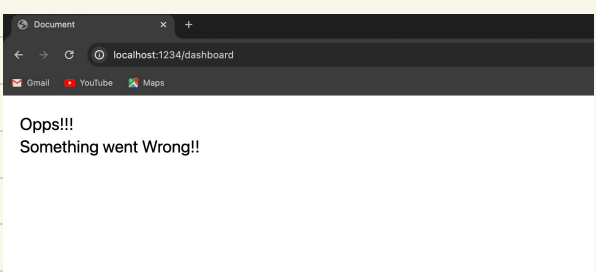
Q. What happens when user navigates to a URL that does not exist?

- react-router-dom is a powerful Library that also handles the error in case we navigate to a route that does not even exist.
- We get a default 404 error page.



- However, we can show our custom error page.
- To display it on the UI, we add one more attribute, errorElement, to the route configuration object for which the provided path doesn't match the URL entered in the UI.
- This errorElement calls the error component whenever an invalid URL is provided.

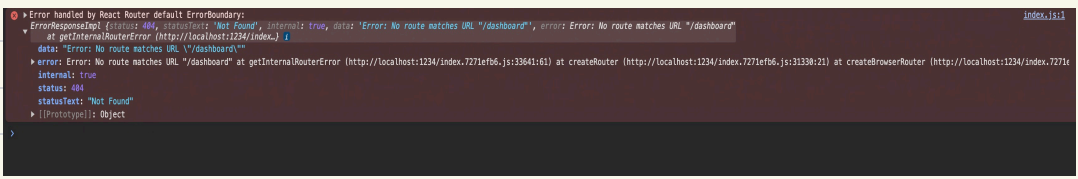
```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout />,
    errorElement: <Error />,
  },
  {
    path: "/about",
    element: <About />,
  },
  {
    path: "/contact",
    element: <Contact />,
  },
]);
```



TanmayVaidya

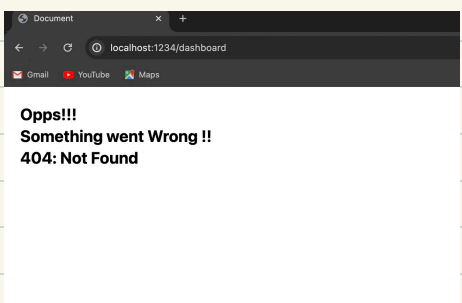
Q. How do we show more information about the error in case user redirects to an Invalid URL?

- react-router-dom provides a hook useRouteError which returns an error object which states what types of error did we encounter.
- It is a hook which won't allow this red colour error to come in consoles.

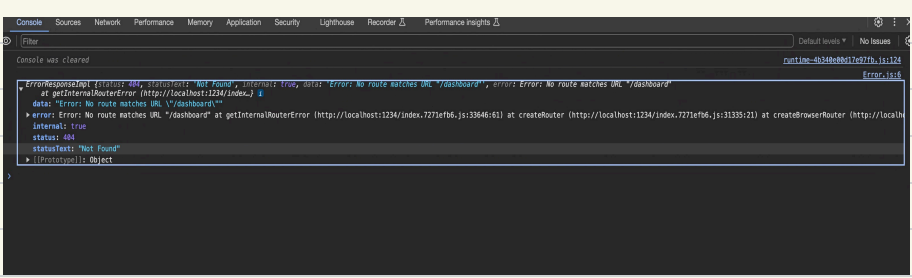


- It catches all the routing errors and we can show those error to the user.

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

const Error = () => {
  const err = useRouteError();

  console.log(err);

  return (
    <>
      <div className="m-6">
        <h1 className="text-2xl font-bold">Opps!!!</h1>
        <h1 className="text-2xl font-bold">Something went Wrong !!</h1>
        <h1 className="text-2xl font-bold">
          {err.status}: {err.statusText}
        </h1>
      </div>
    </>
  );
};

export default Error;
```



- On the console:



Q. What are problems with Anchor Tag?

- Anchor tag leads to Server-Side routing, because in anchor tag we have an attribute href where we provide the URL and when we click on the link, a call is made to the URL hosted in a server and our page reloads until we get the data from the server.
- This is not a good user experience
- To overcome this we use Links.

* Links:

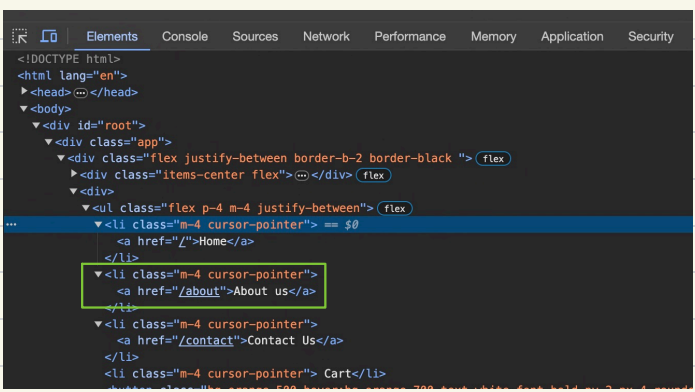
- To achieve client-side routing, we use Links in React.
- We should enclose navigation elements inside a Link tag given by react-router-dom.
- In Link tab we define an attribute named to where we set a path for the link and to decides where to navigate when the link is clicked.

```jsx
import { useState } from "react";
import logo from "../../assets/logo.png";
import { Link } from "react-router-dom";

const Header = () => {
  const [btnName, setBtnName] = useState("Login");

  return (
    <div classNameName="flex justify-between border-b-2 border-black ">
      <div classNameName="items-center flex">
        <img
          src={logo}
          classNameName=" w-36 h-24 items-center cursor-pointer ml-8"
        />
      </div>

      <div>
        <ul classNameName="flex p-4 m-4 justify-between">
          <li classNameName="m-4 cursor-pointer">
            <Link to="/">Home</Link>
          </li>
          <li classNameName="m-4 cursor-pointer">
            <Link to="/about">About us</Link>
          </li>
          <li classNameName="m-4 cursor-pointer">
            <Link to="/contact">Contact Us</Link>
          </li>
          <li classNameName="m-4 cursor-pointer"> Cart</li>
          <button
            classNameName="bg-orange-500 hover:bg-orange-700 text-white font-bold py-2 px-4 rounded"
            onClick={() => {
              btnName === "Login" ? setBtnName("Logout") : setBtnName("Login");
            }}
          >
            {btnName}
          </button>
        </ul>
      </div>
```

- Now if we click on About, it will lead me to about component page.

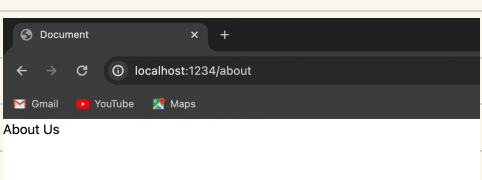
At this moment If we inspect the page, we see anchor tag instead of Link tag. Why?

- Because the Link tag uses anchor tag behind the scene and this is done by react-router-dom library.
- To make the Link tag understandable by browser react-router-dom keeps a track of all the Link tags and converts them to anchor tag which browser understands.



* Nested Routing:

- Nested routing enables us placing multiple children components within the parent component and performs routing on child components without reloading the entire page.
- Right now, if we notice about page component, we can't see the header and footer component loaded. But we want them to be loaded for about component and for other page components.



- In below screen shot, if my path is "/" , I want my "Body" component to be inside header and footer component.
- If my path is "/about", I want my "About" component to be inside header and footer component and so on.

```jsx
const AppLayout = () => {
  return (
    <div className="app">
      <Header />
      <Body />
      <About />
      <Contact />
      <Footer />
    </div>
  );
};
```

- We want our header and footer component fixed and display different components inside header and footer on page navigation.
- To make it happen we will have to make About, Body and Contact component, children of AppLayout component.
- So we need to change the routing config
- Also, we provide an Outlet in our Layout where we want our component to be loaded while navigated.

```jsx
const AppLayout = () => {
  return (
    <div className="app">
      <Header />
      <Outlet />
      <Footer />
    </div>
  );
};

const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout />,
    children: [
      {
        path: "/",
        element: <Body />,
      },
      {
        path: "/about",
        element: <About />,
      },
      {
        path: "/contact",
        element: <Contact />,
      },
    ],
    errorElement: <Error />,
  },
]);

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

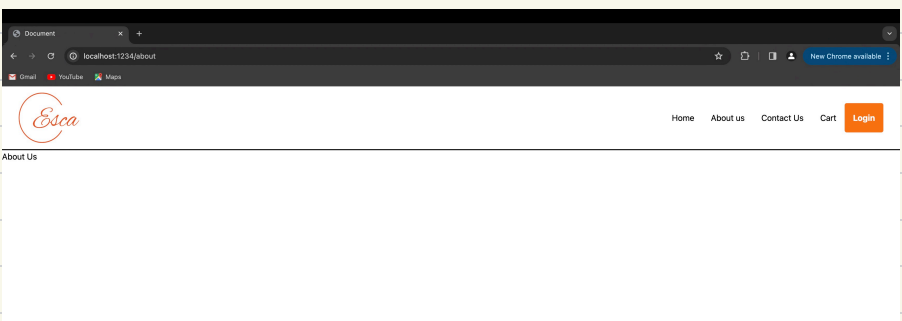
Outlet

Children

* Outlet:

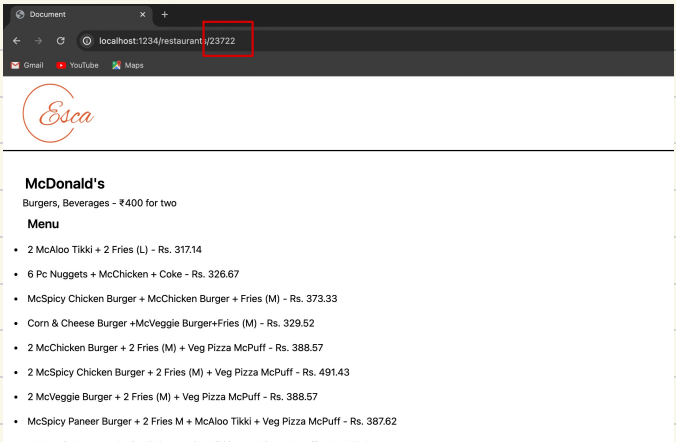
- Outlet is a component that react-router-dom provides to plug in children components into layout component during page navigation.
- Outlet is a place holder for children components configured in route configuration objects.
- When we navigate from page to page react triggers reconciliation which updates only the updated portion in the dom which is the outlet.

Now, About component page navigation Looks like:



* Dynamic Routing:

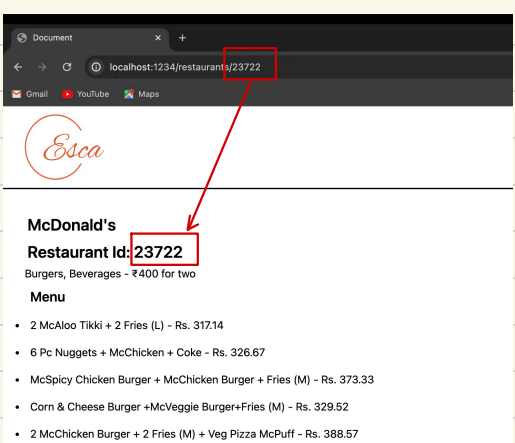
- Dynamic routing is the time at which routing take place.
- In Dynamic routing, routes are initialised dynamically.
- It is a process of rendering components in response to a change in application's URL.
- In our case, restaurant card is clicked and route for that card is generated.



- Behind the scene, when we click on a restaurant card from restaurant list, Id of that restaurant will be attached to the URL and this will be read by restaurant Menu component.
- Restaurant Menu component uses this ID for making API calls and displaying the required data in UI.

* useParams:

- useParam is one of the hooks given by react-router-dom library.
- We can use this hook to retrieve route parameters from a route URL as use it in the component we navigated to.
- useParam enables us to read data from the route URL.



Code Implementation for Dynamic Routing and useParams:

Body.js:

```jsx
<div className="res-container flex flex-wrap">
  {filteredRestaurants?.map((restaurant) => (
    <Link
      className="res-container flex flex-wrap"
      key={restaurant.info.id}
      to={"/restaurants/" + restaurant.info.id}
    >
      <RestaurantCard resData={restaurant} />
    </Link>
  ))}
</div>
```

RestaurantMenu.js:

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

const RestaurantMenu = () => {
  const [resInfo, setResInfo] = useState(null);

  const { resId } = useParams();

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

  const fetchMenu = async () => {
    const data = await fetch(MENU_API + resId);

    const json = await data.json();

    console.log("Menu Json: ", json);

    setResInfo(json.data);
  };

  if (resInfo === null) {
    return <Shimmer />;
  }

  const { name, cuisines, costForTwoMessage } =
    resInfo?.cards[2]?.card?.card?.info;

  const { itemCards } =
    resInfo?.cards[4]?.groupedCard?.cardGroupMap?.REGULAR?.cards[2]?.card?.card;

  console.log(itemCards);

  return (
    <>
      <div className="menu p-8">
        <h1 className=" text-2xl p-1 font-semibold">{name}</h1>
        <h1 className=" text-2xl p-1 font-semibold">Restaurant Id: {resId}</h1>
        <h2>
          {cuisines.join(", ")} - {costForTwoMessage}
        </h2>
        <h2 className="text-xl p-2 font-semibold">Menu</h2>
        <ul className=" list-item list-disc">
          {itemCards?.map((item) => (
            <li className=" list-item p-2" key={item.card.info.id}>
              {item.card.info.name} (" - Rs. ")
              {item.card.info.price / 100 || item.card.info.defaultPrice / 100}
            </li>
          ))}
        </ul>
      </div>
    </>
  );
};

export default RestaurantMenu;
```

RestaurantMenu.js:

```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout />,
    children: [
      {
        path: "/",
        element: <Body />,
      },
      {
        path: "/about",
        element: <About />,
      },
      {
        path: "/contact",
        element: <Contact />,
      },
      {
        path: "/restaurants/:resId",
        element: <RestaurantMenu />,
      },
    ],
    errorElement: <Error />,
  },
]);
```

Q. What would happen if we do console.log(useState())?

- If we do console.log(useState()), we get an array [undefined, function] where first item in an array is state is undefined and the second item in an array is setState function is bound dispatchSetState.

Tanmayvaidya

#Custom Hooks:

Q. Why to use hook in React when we have normal JavaScript function?

- Because in hooks we can define state variables so that react can keep track of these variables and triggers reconciliation on every state change.
- When it comes to a normal JavaScript function we can't define state variables inside the function so that React cannot trigger reconciliation process.

Q. Why should we build our own hook? Why custom hook?

- Reusabilty
- Readability
- Modularity / Separation of Concerns
- Testability / Maintainability

- **Modularity:**
  - Each react component should have a single responsibility which is to render the data in UI.
  - We should separate functional logics away from the component and offloads this responsibility to a custom hook or a helper function and use them whenever required.
  - This is called modularity meaning breaking down the code into smaller pieces, each having their own responsibility.

- **Testability:**
  - We break down larger piece of code into smaller chunks that we store in utilities files which enables us to write more test cases.

* Single Resource Principle (SRP):

- In React, the Single Responsibility Principle (SRP) suggests that each component should have a single responsibility or concern.
- This means that components should ideally do one thing well, promoting modularity, reusability, and ease of maintenance.
- For example, a button component should handle only button-related tasks, while a form component should focus solely on form functionality.
- SRP helps keep code organized and facilitates debugging and refactoring as the application scales.

Q. Why custom hook? Explain with example.

- Assuming we have a component that makes an API call to fetch some data and once the data is available, component renders the data in UI.
- Two tasks are being performed by the component, API call and data rendering.
- We have written the code in such a way that it is breaking Modularity.
- We should offload API call responsibility to a custom hook so that our component will focus only on one specific task which is data rendering in UI.

Example:

- We have written API call logic inside Restaurant Menu component.
- As per SRP, Restaurant Menu component should only render data in UI.
- Restaurant Menu component should not worry about how the data is fetched from the API.

- We will try to extract this logic.
- We will create a custom hook that will help us to get the restaurant details.

Standard Practice:

> Create a hook with "use" name infront of it

Great place to keep re-usable files:

- Make a folder utils (Its utility, helper, common, etc any name)
- Inside it create .js file for custom hooks just like we create for our components.

- Create a Custom Hook useRestaurantMenu.
- Create a new file names as useRestaurantMenu.js inside utils folder
- Now we separate the fetching logic out from the Restaurant Menu component and place it inside a custom hook in useRestaurantMenu.js.
- So that when Restaurant Menu component needs this module functionality, it gets it from the import.

```jsx
import { useState, useEffect } from "react";
import { MENU_API } from "../utils/constants";

const useRestaurantMenu = (resId) => {
  const [resInfo, setResInfo] = useState(null);

  //fetching logic

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

  fetchData = async () => {
    const data = await fetch(MENU_API + resId);
    const json = await data.json();

    setResInfo(json.data);
  };

  return resInfo;
};

export default useRestaurantMenu;
```

useRestaurantMenu.js

- Meanwhile in RestaurantMenu, it would be imported and used as:

```jsx
import Shimmer from "./Shimmer";
import { useParams } from "react-router-dom";
import useRestaurantMenu from "../utils/useRestaurantMenu";

const RestaurantMenu = () => {
  const { resId } = useParams();

  const resInfo = useRestaurantMenu(resId);
```

RestaurantMenu.js

Tanmayvaidya

* Building Online and Offline feature:

- If the user has no internet connection then it should show You are Offline check your internet connection else it should show the actual data.

- Create a Custom Hook useOnlineStatus.
- Create a new file names as useOnlineStatus.js inside utils folder

```js
import { useEffect, useState } from "react";

const useOnlineStatus = () => {
  const [onlineStatus, setOnlineStatus] = useState(true);

  useEffect(() => {
    addEventListener("offline", (event) => {
      setOnlineStatus(false);
    });

    addEventListener("online", (event) => {
      setOnlineStatus(true);
    });
  }, []);

  return onlineStatus;
};

export default useOnlineStatus;
```

useOnlineStatus.js

namaste-react - Body.js

```jsx
const onlineStatus = useOnlineStatus();

if (!onlineStatus) {
  return (
    <h1 className="font-bold text-3xl p-8">
      No Internet!! Please check your Internet Connection
    </h1>
  );
}
```

Body.js

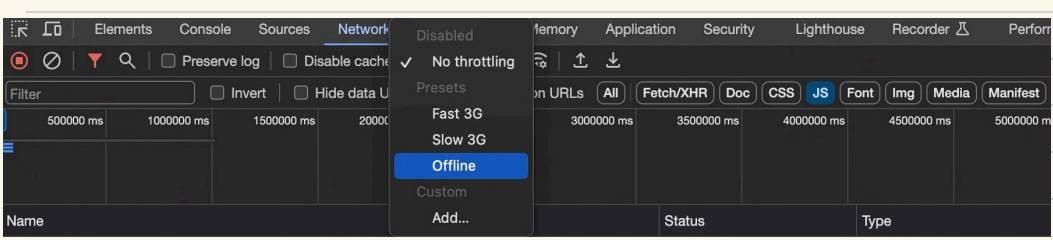
Explanation:

- In JavaScript, window object provides a method called addEventListener where we attach events.
- In our case we are attaching online and offline events to addEventListener method, also we are providing a callback function as a second argument to addEventListener which gets called when user gets online or offline.
- We want this event to be called just once for both online and offline. That's why we have put them inside useEffect hook.

Important:

- When we navigate from component to component, online and offline event listeners should have been removed but they still exist.
- So whenever, eventlistner is added, we should clean it up.
- Because, whenever you are going offline and getting back online, eventlistener is created only once because we have empty dependency array.
- It is always a good practice to remove event listeners on components navigation otherwise browser will keep hold to these events which is a big performance hit.

Simulating offline online behaviour in chrome browse:



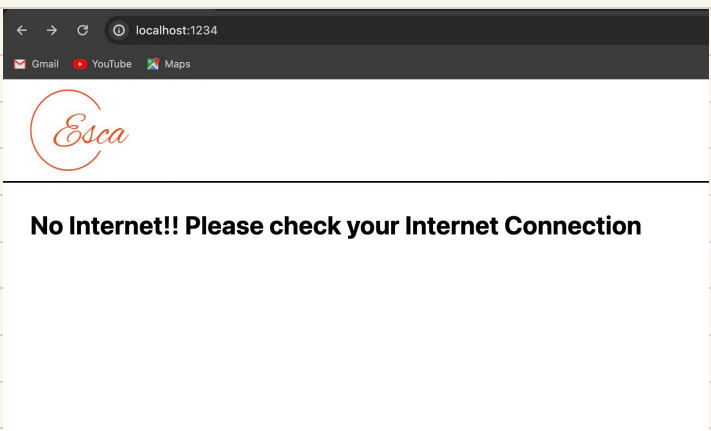
TanmayVaidya

Offline: offline mode

No Throttling: Online mode (Full Speed)

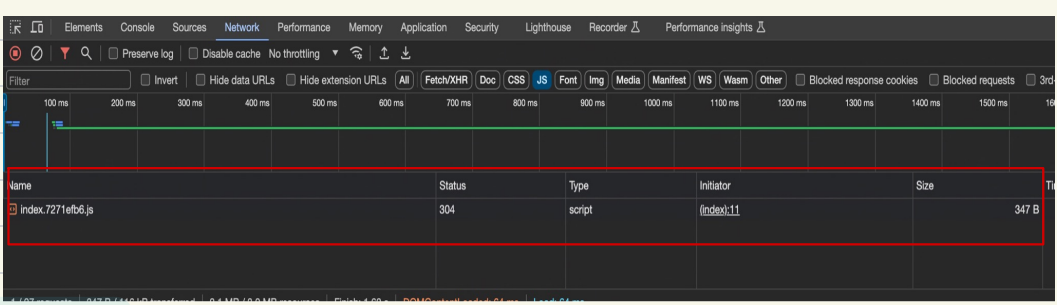
Fast 3G / Slow 3G: Network Speeds

- Now on clicking on offline we get to see this:



* Optimization:

- In our application parcel created only one JavaScript file where all of our code is bundled and minified together.



- The size of the index file bundle is less in production build..

- In large scale application like MakeMyTrip, there are thousands of components and if we bundle them all together in a single index.js file the application will become very slow.
- Because when the page loads for the first time, all components from the bundle will be loaded at once.
- So, to build a large scale production ready application, we should do Chunking.

* Chunking:

- It is also called as:
  - Code Splitting
  - Lazy Loading
  - Dynamic Bundling
  - On Demand Loading
  - Dynamic import

- If we build a large-scale production ready application, there will be multiple components and bundling them all together in a single JS file will take so much time to render in UI.
- Therefore, we should break the entire production ready code into smaller chunks.
- This concept is known as Chunking or code splitting.

Example:

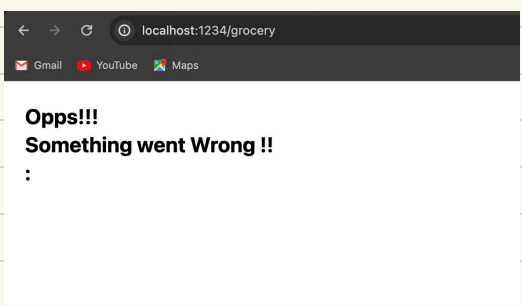
- Create a Grocery Component.
- Now in App.js instead of importing in the regular way, do this:

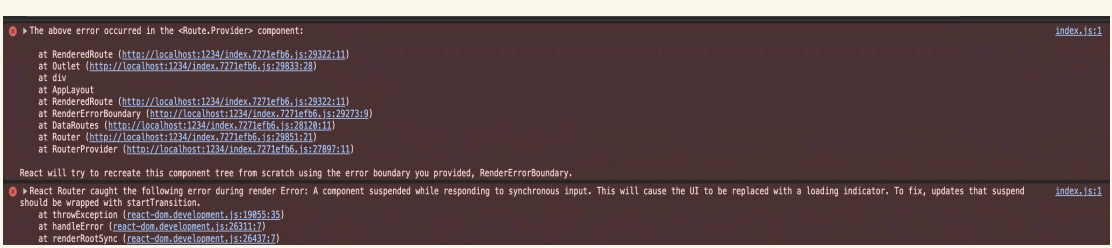
```
const Grocery = lazy(() => import("./components/Grocery"));
```

- So, now index.js file in dist folder won't have the code of Grocery.
- It is created as a separate file while loading.
- This is called On-Demand Loading.

- When you are loading your component on demand, react tries to suspend it.

- So, when Grocery is loaded for the first time, we see an error message on screen.





- This is because, Grocery file took 27ms to get loaded. But react tries to render it before it gets loaded. Hence the error.
- Solution for this is Suspense.

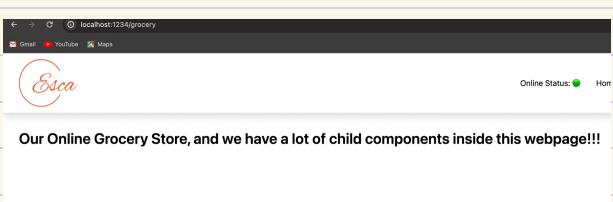
* Suspense:

- It suspends the lazy loaded components until bundler for the component is available in UI

- We can wrap Grocery inside Suspense

```
{
  path: "/grocery",
  element: (
    <Suspense>
      <Grocery />
    </Suspense>
  ),
},
```

- Now React knows Grocery Component will be lazy loaded
- So it waits for the promise to get resolved.
- Import () returns a promise and react waits for the promise to get resolved when we wrap a component inside a suspense tag.
- Now when we navigate to Grocery page, we see the contents of it because this time contents are coming from the Grocery component from loaded bundle.

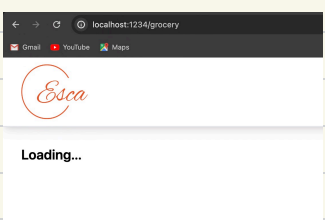


* Fallback:

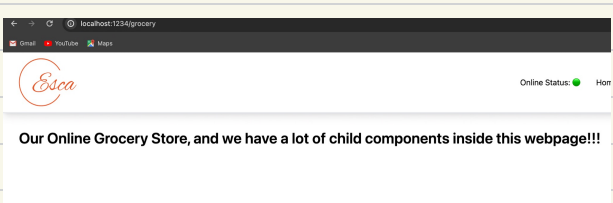
- When bundle is not loaded react suspends the component for a while until the bundle is available.
- During this time for a better user experience we should display a shimmer UI on screen.
- So, suspense component accepts a prop called fallback where we call shimmer UI component.
- We can also write JSX in fallback attribute within {}.

```
{
  path: "/grocery",
  element: (
    <Suspense
      fallback={
        <h1 className="font-semibold text-2xl m-8">Loading...</h1>
      }
    >
      <Grocery />
    </Suspense>
  ),
},
```

- So Now if we navigate to the Grocery page from Home page.



- When bundle is loaded:



Note:

Never ever dynamically load your component inside other component.

Why:

- Because if we do, for ever state or prop change, parent component will be rendered.
- As a result, lazy loading of the child component takes place on every render. So, every time the bundle will be loaded.

Tanmayvaidya

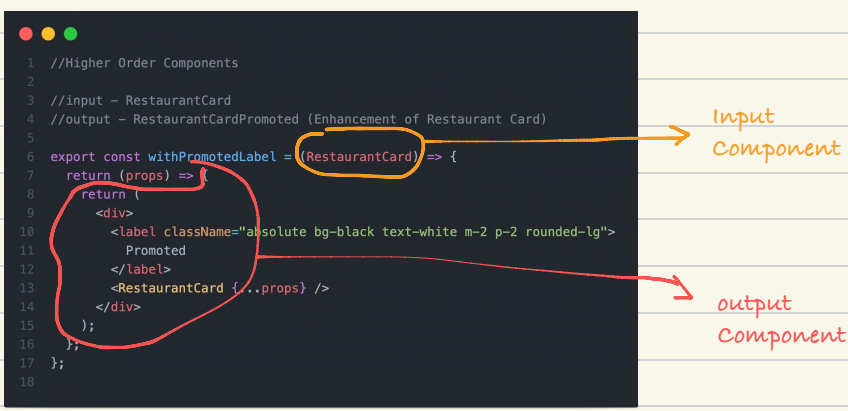
#Playing with Data:

* Higher Order Components:

- Higher Order Components (HOCs) in React are functions that take a component and return a new, enhanced component.
- They enable code reuse by adding functionality to components, promoting composability and separation of concerns.

Example:

- In Swiggy Bangalore, out of many restaurants, there are few promoted restaurants present.
- The only difference between normal and promoted restaurant is that it has promoted label on the UI Card.



RestaurantCard.js

- Created above HigherOrderComponent

```
const RestaurantCardPromoted = withPromotedLabel(RestaurantCard);
```

Body.js

- Now, as withPromotedLabel is a HOC, we have passed the component RestaurantCard in it, and it will return us a new component RestaurantCardPromoted which has a promoted label inside it.

- Now, if is promoted is true then we would be returning the HOC else the normal Component

```
<div className="res-container flex flex-wrap">
  {filteredRestaurants?.map((restaurant) => (
    <Link
      className="res-container flex flex-wrap"
      key={restaurant.info.id}
      to={"/restaurants/" + restaurant.info.id}
    >
      {restaurant?.info?.promoted ? (
        <RestaurantCardPromoted resData={restaurant} />
      ) : (
        <RestaurantCard resData={restaurant} />
      )}
    </Link>
  ))}
</div>
```

Body.js

Creating Accordions:

- Now, lets display all the categories along with their items into accordions.
- Create a RestaurantCategory.js file for displaying the catergories

- From RestaurantMenu.js, we will pass the categories data to the newly created component.

```jsx
import Shimmer from "./Shimmer";
import { useParams } from "react-router-dom";
import useRestaurantMenu from "../utils/useRestaurantMenu";
import RestaurantCategory from "./RestaurantCategory";

const RestaurantMenu = () => {
  const { resId } = useParams();

  const resInfo = useRestaurantMenu(resId);

  if (resInfo === null) {
    return <Shimmer />;
  }

  const { name, cuisines, costForTwoMessage } =
    resInfo?.cards[0]?.card?.card?.info;

  const categories =
    resInfo?.cards[2]?.groupedCard?.cardGroupMap?.REGULAR?.cards.filter(
      (c) =>
        c.card?.card?.["@type"] ===
        "type.googleapis.com/swiggy.presentation.food.v2.ItemCategory"
    );

  return (
    <>
      <div className="menu p-2 text-center">
        <h1 className=" text-2xl font-bold my-6">{name}</h1>
        <h2 className="text-lg font-bold">
          {cuisines.join(", ")} - {costForTwoMessage}
        </h2>
        {/* Categories Accordians */}
        {categories?.map((category) => (
          <RestaurantCategory
            key={category?.card?.card?.title}
            data={category?.card?.card}
          />
        ))}
      </div>
    </>
  );
};

export default RestaurantMenu;
```

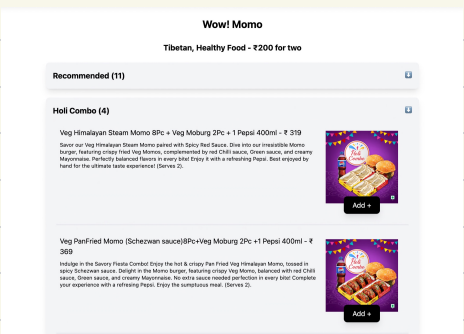
RestaurantMenu.js

- In RestaurantCategory.js we would write logic for displaying the categories heading, i.e. accordions heading

```jsx
import { useState } from "react";
import ItemList from "./ItemList";

const RestaurantCategory = ({ data }) => {
  //console.log(data);

  const [showItems, setShowItems] = useState(false);

  const handleClick = () => {
    setShowItems(!showItems);
  };

  return (
    <div>
      <div className=" w-6/12 mx-auto my-5 bg-gray-100 shadow-lg p-4 rounded-lg ">
        <div
          className="flex justify-between cursor-pointer"
          onClick={handleClick}
        >
          <span className="font-bold text-lg">
            {data?.title} ({data?.itemCards?.length})
          </span>
          <span>⬇️</span>
        </div>
        {showItems && <ItemList items={data?.itemCards} />}
      </div>
    </div>
  );
};

export default RestaurantCategory;
```

RestaurantCategory.js

- Here if my showItems is true, then only show the content inside it.
- So, when showItems is true, accordion would be open, and if false then accordion would be closed.
- Inside ItemList, the content of the list is shown.

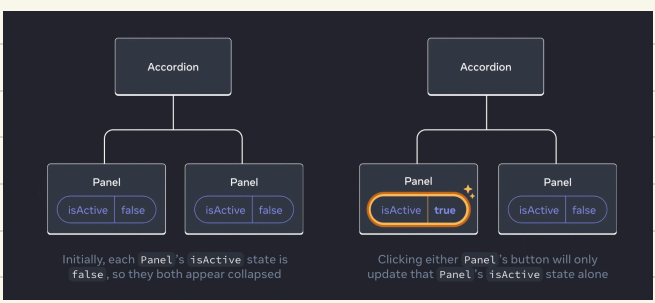


Tanmayvaidya

- Here the power of show, hide is given to RestaurantCategory.
- So each accordion will have its own state / power to control the show, hide. So at a time all the accordions can also be opened.
- So as RestaurantCategory has its own state, i.e., its parent (RestaurantMenu) does not have any control over the states of RestaurantCategory.
- Hence RestaurantCategory is an Uncontrolled Component.

* Uncontrolled Components:

- Uncontrolled components in React allow child components to manage their own state internally through the DOM, without direct control or state management by the parent component.

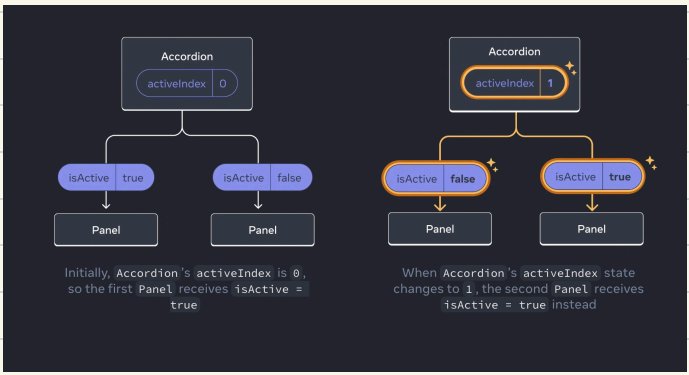


- For eg, here each panel has its own state as isActive and it also not controlled by their parent Accordion.

- Now I don't want to give the power of show/hide to the RestaurantCategory, but need to give it to its parent RestaurantMenu. (I need only one accordion to be open at a time)
- So for parent to control the child components state, the child's component state should be lifted up to the parent component.
- So now, RestaurantCategory will be Controlled Component.

* Lifting State Up:

- It involves moving the state management from child components to their common ancestor, typically a parent component.



- For eg, here each panel state is managed by the parent component (Accordion).

* Controlled Component:

- Controlled components in React are those where the parent component manages the state of form elements.
- The parent passes down the current values as props to child components, which in turn notify the parent of any changes via event handlers.

- Now, for controlling the RestaurantCard from RestaurantMenu, we would do the necessary modifications.

```
1  const [showIndex, setShowIndex] = useState(null);
```

RestaurantMenu.js

```jsx
{categories?.map((category, index) => (
  // Controlled Component
  <RestaurantCategory
    key={category?.card?.card?.title}
    data={category?.card?.card}
    showItems={index === showIndex ? true : false}
    setShowIndex={() => setShowIndex(index)}
  />
))}
```

RestaurantMenu.js

```jsx
const RestaurantCategory = ({ data, showItems, setShowIndex }) => {
  //console.log(data);

  const handleClick = () => {
    // setShowItems(!showItems);
    setShowIndex();
  };

```

RestaurantCategory.js

- Now show/hide power is controlled by RestaurantMenu

Q. There is a small bug in above code. What is it?

- Here, the bug is once if we open an accordion, we can't close the same accordion by clicking on it.
- It is because our condition for showItem was

  index === showIndex ? true : false

- So each time it will check the index and return true.

Q. How to fix the above bug?

```jsx
{categories?.map((category, index) => (
  // Controlled Component
  <RestaurantCategory
    key={category?.card?.card?.title}
    data={category?.card?.card}
    showItems={index === showIndex}
    setShowIndex={() =>
      setShowIndex(index === showIndex ? null : index)
    }
  />
))}
```

RestaurantMenu.js

```jsx
const RestaurantCategory = ({ data, showItems, setShowIndex }) => {
  //console.log(data);

  const handleClick = () => {
    // setShowItems(!showItems);
    setShowIndex((prevIndex) => (prevIndex === showItems ? null : showItems));
  };

```

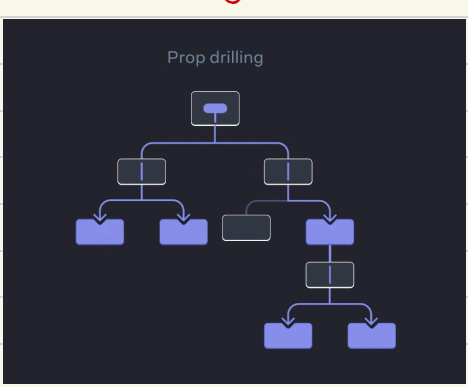
RestaurantCategory.js

TanmayVaidya

- In the RestaurantMenu Component, we added a showIndex state variable to keep the rack of the currently expanded category index. It's initialised with null since no category is open by default.
- When mapping through the categories, we use the index to determine whether the category should be expanded or collapsed based on whether it's index matches the showIndex.
- We pass the showIndex state and a function setShowIndex to the RestaurantCategory Component.
- The setShowIndex function is used to update the showIndex state when a category is clicked. If the category index matches the showIndex, we set it to null to collapse it.

* Props Drilling:

- Props drilling refers to the process of passing props down through multiple levels of nested components in a React application.
- It occurs when data needs to be transferred from a parent component to a deeply nested child component.
- While it's a common practice in React, it can lead to code that's harder to maintain as the application grows in complexity.
- This is because props may end up being passed through intermediate components that don't actually use them, adding unnecessary complexity.

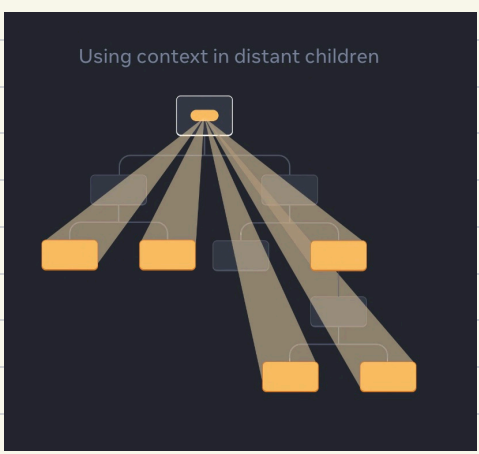


- Alternative solutions like context API (React Context) or state management libraries (like Redux) can help mitigate this issue by providing a centralized way to manage and access state across components, reducing the need for props drilling and making the codebase more scalable and maintainable.

* React Context:

- React Context is commonly used when you have data that needs to be accessed by many components at different levels of nesting, such as themes, user authentication, localization, or global state management.

- It helps to avoid prop drilling and makes your code cleaner and more maintainable by providing a way to share data across components without having to explicitly pass props down through every level of the component tree.



Tanmayvaidya

Example:

- We would create a data that should be accessible anywhere from the app i.e. loggedInUser.
- So now we would create a context named as UserContext inside utils in UserContext.js file.
- React provides a utility function called createContext. It takes in the default value of our context and helps to make our data available throughout our app

```
import { createContext } from "react";

const UserContext = createContext({
  //pieces of information
  loggedInUser: "Default User",
});

export default UserContext;

```

UserContext.js

- Now, we have successfully created an user data which we can globally use and exported it. We have to now access this data from the necessary components.
- Now, we want to access this in our header component.
- To access it, i.e. to use the context, React provides us another hook called useContext.

Tanmayvaidya

```jsx
import { useContext, useState } from "react";
import logo from "../../assets/logo.png";
import { Link } from "react-router-dom";
import useOnlineStatus from "../utils/useOnlineStatus";
import UserContext from "../utils/UserContext";

const Header = () => {
  const [btnName, setBtnName] = useState("Login");

  const onlineStatus = useOnlineStatus();

  const data = useContext(UserContext);
  console.log(data);
```

```
Console was cleared
▼ {loggedInUser: 'Default User'} i
    loggedInUser: "Default User"
  ▶ [[Prototype]]: Object
>
```

Header.js

- We can extract the loggedInUser:

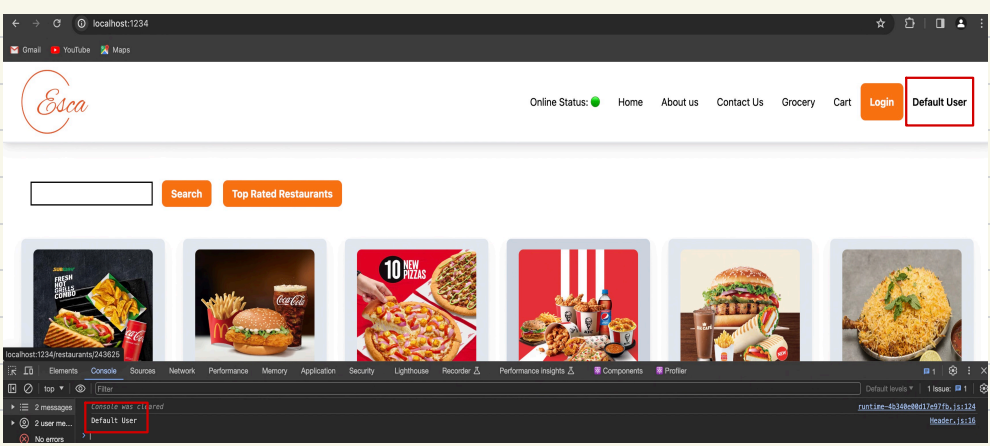
```jsx
const { loggedInUser } = useContext(UserContext);
console.log(loggedInUser);
```

Header.js

- Now, We need to display it in header

```jsx
<li className="m-4 font-semibold cursor-pointer">{loggedInUser}</li>
```

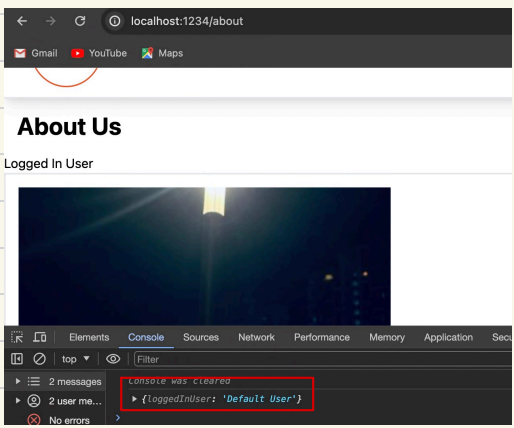
Header.js



* Using useContext in Class Based Components:

- We know that in CBC, we do not have any concept of hook.
- So, in CBC we first import the context that was created (here UserContext) and use it as a component.
- This component can accept a JSX piece of code, within which we can write a call function where the function parameter is actually the context data.

```jsx
import UserContext from "../utils/UserContext";
import User from "./User";
import UserClass from "./UserClass";
import React from "react";

class About extends React.Component {
  constructor(props) {
    super(props);

    // console.log("Parent Constructor");
  }

  componentDidMount() {
    // console.log("Parent Did Mount");
  }

  render() {
    // console.log("Parent Render");
    return (
      <>
        <h1 className="p-4 font-bold text-3xl">About Us</h1>

        <div>
          Logged In User
          <UserContext.Consumer>
            {(data) => {
              console.log(data);
            }}
          </UserContext.Consumer>
        </div>
        <UserClass name={"Child1"} location={"Maharashtra"} />
      </>
```



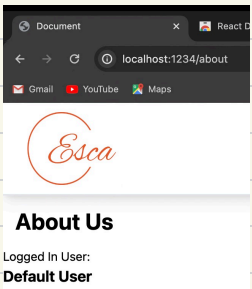
About.js

- UserContext.Consumer component in React is used to consume the context data.
- It allows components to access the context value without using the useContext() hook.

- We can extract loggedInUser:

```jsx
<div>
  Logged In User:
  <UserContext.Consumer>
    {({ loggedInUser }) => (
      <h1 className="text-xl font-bold">{loggedInUser}</h1>
    )}
  </UserContext.Consumer>
</div>
```

About.js



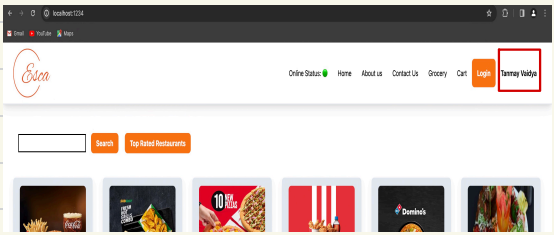
- Till now we are always using the dummy user data . But in an actual app, we first need to authenticate the user and then override the dummy data with the logged in user details.
- Remember that we have already hardcorded a loggedInUser data inside the App component (because we will making an API call to authenticate the user from that component's useEffect() hook).

```jsx
//authentication
useEffect(() => {
  //Make an API call and send username and password
  //dummy data
  const data = {
    name: "Tanmay Vaidya",
  };

  //Update username
  setUserName(data.name);
}, []);
```

App.js

- Like we called the UserContext as a component in the CBC i.e. the About Component and used UserContext.Consumer to get the context data, here also we will a very similar UserContext.Provider to provide values to the context data.

```jsx
const AppLayout = () => {
  const [userName, setUserName] = useState();

  //authentication
  useEffect(() => {
    //Make an API call and send username and password
    //dummy data

    const data = {
      name: "Tanmay Vaidya",
    };

    //Update username
    setUserName(data.name);
  }, []);

  return (
    <UserContext.Provider value={{ loggedInUser: userName }}>
      <div className="app">
        <Header />
        <Outlet />
      </div>
    </UserContext.Provider>
  );
};
```



App.js

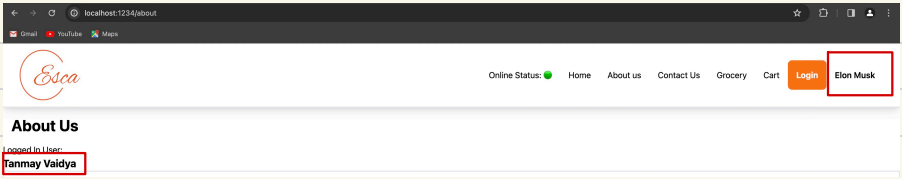
- Now, here we have mentioned all the components i.e. Header, and Outlet inside the Provider tag. This helps to update the context data for all the components.
- If any of the component be used outside the provider, that component would have been served the default value of the context.

Q. Can we have nested Providers with different values?

- Yes

```jsx
return (
  <UserContext.Provider value={{ loggedInUser: userName }}>
    <div className="app">
      <UserContext.Provider value={{ loggedInUser: "Elon Musk" }}>
        <Header />
      </UserContext.Provider>
      <Outlet />
    </div>
  </UserContext.Provider>
);
```

App.js



- Suppose we will have an input box beside the Top Rated button with a value as the user name. Now, when we change that value, the user details should also be changed. (This is a useless functionality, but good for conceptualisation).

- We know that the user detail (not the dummy one) is a state variable and we can only change a state variable with a set function.
- Also, we have to make this set function accessible throughout our app. So to do this, we need to pass this set function in our value props too (of the UserContext.Provider component of the App component).

- Then in the Body component, we will make an input box with the value as the context data's username and we will also attach a onChange function to change the user details as per the input box's value.

App.js

```jsx
const AppLayout = () => {
  const [userName, setUserName] = useState();

  //authentication
  useEffect(() => {
    //Make an API call and send username and password
    //dummy data

    const data = {
      name: "Tanmay Vaidya",
    };

    //Update username
    setUserName(data.name);
  }, []);

  return (
    <UserContext.Provider value={{ loggedInUser: userName, setUserName }}>
      <div className="app">
        <Header />
        <Outlet />
      </div>
    </UserContext.Provider>
  );
};
```

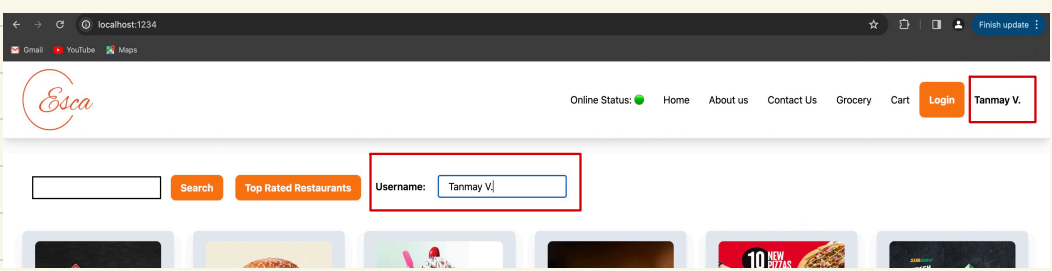
- Also in Body Component, we would extract the loggedInUser and setUserName.

```
const { loggedInUser, setUserName } = useContext(UserContext);
```

```
<div className="mx-6">
  <label className="font-semibold">Username: </label>
  <input
    className="border-solid border-black border-2 py-1 px-4 mx-4 border-r-2"
    type="text"
    value={loggedInUser}
    onChange={(e) => {
      setUserName(e.target.value);
    }}
  ></input>
</div>
```

Body.js

- So now on UI it would work as:



Tanmayvaidya

Tanmayvaidya

#Redux:

* Redux:

- Redux is a State management tool for UI applications most commonly used in library such as React JS and frameworks such as angular and view etc.
- It allows us to store all our application's data in one place known as store.
- This makes it easier to build complex real world applications and keep the state organized.

* Advantages of Redux:

- Provides Centralised state management system.
- Prevents Props Drilling.
- Preferable over context API / React Context in large scale application.
- Provides Performance Optimisations.

Q. How Redux provides Performance Optimisations ?

- In react whenever the state or prop gets changed the component tree using the state or prop gets re-rendered.
- But in Redux when the data is changed inside store, a shallow copy of that data is created as a result re-render is less likely.

* Disadvantages of Redux:

- A huge learning curve for new comers.
- Not suitable for small applications.
- Configuring Redux store is too much complex.
- Redux requires lots of boiler plate code making the code messy and unreadable.
- Debugging is difficult.

* Redux ToolKit (RTK):

- RTK is a library that makes it easier to work with Redux in a React application.
- It enables us to write redux code in a concise way.
- RTK abstracts the basic redux code preventing unnecessary boiler plate code which enables developer like us to write clean concise redux code.

* Redux Store:

- Redux store is a big JS object.
- Redux store is basically a central place to store data needed by any components in react app.
- We can have multiple react context to store multiple data. But we have a single redux store for global data management.

Q. But is it a good practice to keep all the data inside the whole big object ?

- Yes it is absolutely fine.
- But so that our redux store become very big and clumsy, we have something known as Slicing in our redux store.

* Slices:

- Slices are logical separations or portions within the Redux Store.
- Each slice maintains a mapping between actions and reducer functions so that when the action is dispatched reducer of that action gets called.
- In short slices not only stores data portion wise but also it manages/updates these data.

Tanmayvaidya

Q. What slices we can have in our app?

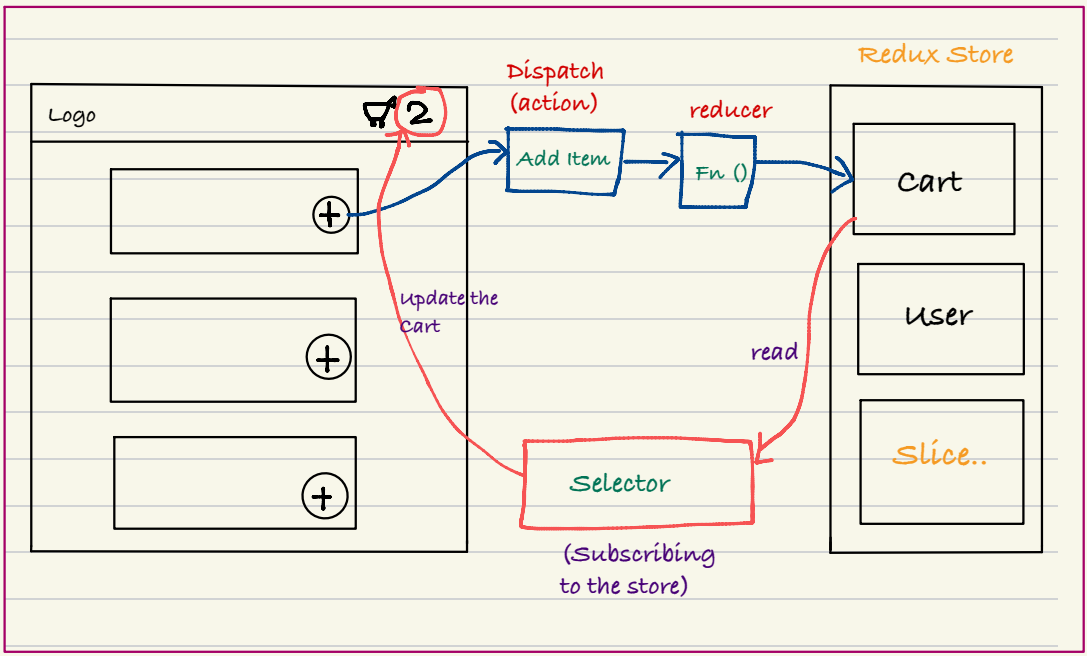
- We can have
  - User Slice
  - Authentication Slice
  - Theme Slice
  - Cart Slice

* Redux ToolKit Architecture:

- The way E commerce application works is, in E-cart system we can add items to the cart and then proceed for a check out.
- Our component cannot directly modify the store.
- Suppose, if we click on the '+' button, we cannot directly modify the store or cannot modify the 'cart'
- So what can we do?
  - We have to dispatch an action
- Here for example, action be of 'add item'.
- When we will click on 'add item', it means we are dispatching an action.
- So what will this action do?
  - It will call a function (normal JS function)
- And this function will modify our cart.

Q. Why we cannot directly modify the store?

- Directly modifying the store in state management libraries like Redux is discouraged because it violates principles of immutability.
- Modifying the store directly can lead to unpredictable behavior, make debugging difficult, and bypass important tools and optimizations provided by the library.

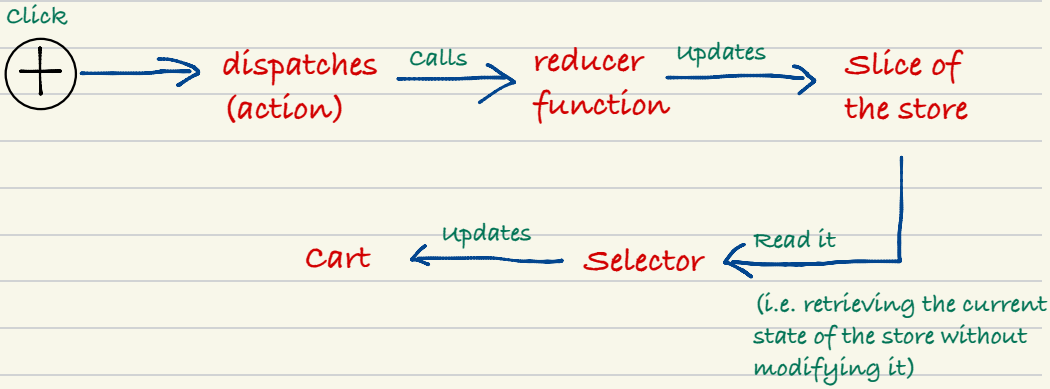


- When we click on the '+' button, we dispatch an action which calls a function known as reducer function, which modify or update the slice (here cart) of the store.

- If we want to read cart, we have to call selector.
- The selector will give us the information of the store and will modify our react component.

- For more clear understanding, below is the basic flow:

Tanmayvaidya



- Selector means we are selecting the slice out of the store or we are selecting the portion of the store.
- This selector is a hook known as useSelector().

- There is one more jargon, that is when we use selector it is known as subscribing to the store

Subscribing the store —Means→ Reading the store

- So, basically it is like async with the store.
- Whenever my store will modify, cart will be automatically modify in the UI.
- It means that component is subscribed by using Selectors.
- Whole flow in one line :

When we click on the + button, it dispatches an action which calls a reducer function, which modifies or update the slice of the store, and because cart is subscribed to the store using a selector, it automatically gets updated.

* Setup Redux in React:

- We need to install 2 libraries:

1. `npm i @reduxjs/toolkit`

   a. This library provides functions using which we can create Redux store, Slices.
   b. This library provides the core functionality of Redux.

2. `npm i react-redux`

   a. This library acts as a bridge between React app and Redux.

* Create Redux Store:

- Now, we will create our store.
- For creating store, RTK (Redux Tool Kit) provides an API configureStore which creates Redux store.
- This configureStore will contain Slices.

```
import { configureStore } from "@reduxjs/toolkit";

//Creating Redux Store
const appStore = configureStore({

});

export default appStore;
```

appStore.js

* Providing Redux Store to Application:

- Now, this store is different and our app is also different.
- React-redux library provides a component Provider which acts as a bridge between redux store and react application.
- So we will need an Provider to connect our store with the app.
- To establish this bridge, we wrap the application root components within the Provider component and to make the store available to our application we pass store as a prop inside Provider Component.
- We can put this Provider for the whole app or for certain components also.

App.js

```
import { Provider } from "react-redux";
import appStore from "./utils/appStore";

const Grocery = lazy(() => import("./components/Grocery"));

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

const AppLayout = () => {
  const [userName, setUserName] = useState();

  //authentication
  useEffect(() => {
    //Make an API call and send username and password
    //dummy data
    const data = {
      name: "Tanmay Vaidya",
    };

    //Update username
    setUserName(data.name);
  }, []);

  return (
    <Provider store={appStore}>
      <UserContext.Provider value={{ loggedInUser: userName, setUserName }}>
        <div className="app">
          <Header />
          <Outlet />
        </div>
      </UserContext.Provider>
    </Provider>
  );
};
```

* Creating Slices inside Redux Store:

- RTK provides an API createSlice which creates Slices for us.

```js
import { createSlice } from "@reduxjs/toolkit";

const cartSlice = createSlice({
  name: "cart",
  initialState: {
    items: [],
  },
  reducers: {
    addItem: (state, action) => {
      state.items.push(action.payload);
    },
    removeItem: (state) => {
      state.items.pop();
    },
    clearCart: (state) => {
      state.items.length = 0;
    },
  },
});

export const { addItem, removeItem, clearCart } = cartSlice.actions;

export default cartSlice.reducer;
```

Mutating the state (Directly modifying the state)

name:

- It represents name of the slice which is cart.

initialState:

- It represents an object where we set initial state of the slice.
- In our case we have set items to an empty array because initially our cart will have zero items in it.

reducers:

- It represent an object where we map actions with reducer functions
- For eg. addItem is a reducer fn()which takes 2 parameters state and action, and it modifies the state based on the action.

cartSlice.reducer:

- It represents an object which wraps up all the reducer functions and export them as in one big reducer object.

cartSlice.actions:

- In line number 21 we are de-structuring all the actions from cartSlice.actions and exporting them at once.

Q. What are reducer functions?

- Reducer functions are normal JavaScript function which get called when associated action is dispatched.
- In reducer function we provide state management logic which updates the state based on the triggered action.
- Reducer function takes in two parameters state and action where state is the current state and action is a plain JavaScript object having payload received from UI.
- When no action is dispatched state variable will have initial state of the cart slice (items = []) and when action is triggered state (items) will be updated

* Adding Slices to Redux Store:

- configureStore receives an object where we add and configure all our slices.
- The object has a key named reducer with a value a new object which contains mapping between slice name and the actual imported slice.
- For different slices, we add their configuration as a key value pair inside reducer property.

```
import { configureStore } from "@reduxjs/toolkit";
import cartSlice from "./cartSlice";

//Creating Redux Store
const appStore = configureStore({
  reducer: {
    cart: cartSlice,
  },
});

export default appStore;
```

- Here this whole big reducer is an app reducer, and this reducer contains small reducers for each slice, i.e. for each slice there is a reducer and we will just add all the reducers inside it.

* Subscribing to Redux Store:

- React application subscribe to the redux store using useSelector hook.
- useSelector comes from react-redux library because useSelector acts as a bridge between React and Redux store.

- To check functionality of subscribing, we would provide dummy data to the items in the initial state.

```
import { createSlice } from "@reduxjs/toolkit";

const cartSlice = createSlice({
  name: "cart",
  initialState: {
    items: ["burger", "pizza"],
  },
  reducers: {
```

- Now, we will subscribe using the useSelector()

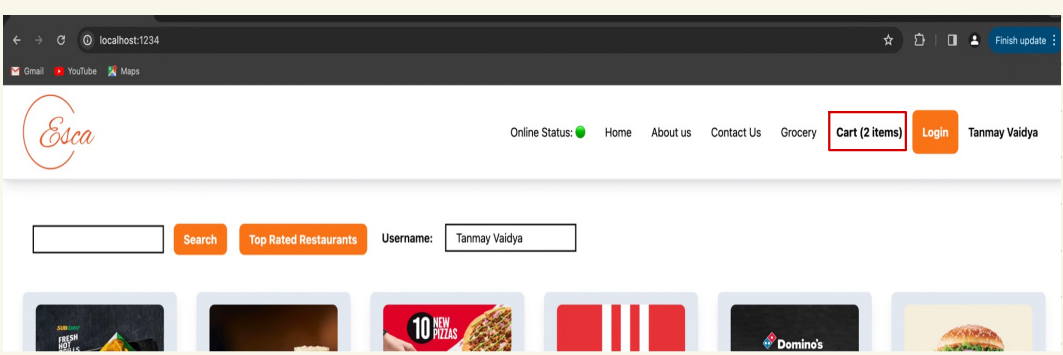
```
//Subscribing to the store using a Selector
const cartItems = useSelector((store) => store.cart.items);
```

It will give access to whole store

It will give access to cart store that we need.

- Now, we will display this data on UI:

```
<li className="m-4 font-bold cursor-pointer">
  Cart ({cartItems.length} items)
</li>
```



Tanmayvaidya

* Dispatching an Action:

- React-redux library provides a hook called useDispatch that returns a function and the function takes an action with an action payload.
- This function is responsible for dispatching the action.

- Lets understand with example, onClick of Add Button we need to dispatch and item

```jsx
<button
   className="p-2 flex items-center justify-center w-1/2 rounded-lg bg-black text-white shadow-lg"
   onClick={handleAddItem}
 >
   Add +
 </button>
```

```jsx
import { useDispatch } from "react-redux";
import { addItem } from "../utils/cartSlice";
import { CDN_URL } from "../utils/constants";

const ItemList = ({ items }) => {
  // console.log(items);

  const dispatch = useDispatch();

  const handleAddItem = () => {
    //Dispatch an Action
    dispatch(addItem("pizza"));
  };
```

Whatever is passed here will go inside reducer fn(), that too inside a payload [action.payload]

```jsx
const cartSlice = createSlice({
  name: "cart",
  initialState: {
    items: [],
  },
  reducers: {
    addItem: (state, action) => {
      //mutating the states
      state.items.push(action.payload);
    },
    removeItem: (state) => {
      state.items.pop();
    },
    clearCart: (state) => {
      state.items.length = 0;
    },
  },
});
```

This is now pizza

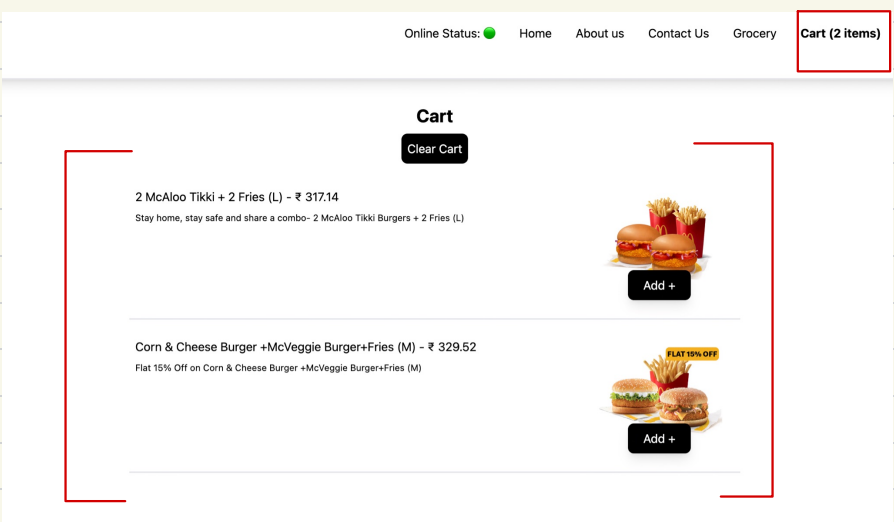
- Now lets add specific item.

```
const ItemList = ({ items }) => {

  const dispatch = useDispatch();

  const handleAddItem = (item) => {
    //Dispatch an Action
    dispatch(addItem(item));
  };

  return (
    <div>
      {items?.map((item) => (
        <div
          key={item?.card?.info?.id}
          className="p-2 m-2 border-gray-200 border-b-2 text-left flex"
        >
          <div className="w-9/12">
            <div className="py-2">
              <span>{item?.card?.info?.name}</span>
              <span>
                {" "}
                - ₹{" "}
                {item?.card?.info?.price
                  ? item?.card?.info?.price / 100
                  : item?.card?.info?.defaultPrice / 100}
              </span>
            </div>
            <p className="text-xs">{item?.card?.info?.description}</p>
          </div>
          <div className="w-3/12 p-4">
            <img
              src={CDN_URL + item?.card?.info?.imageId}
              className="w-full h-auto"
            />
            <div className="flex justify-center -mt-4">
              <button
                className="p-2 flex items-center justify-center w-1/2 rounded-lg bg-black text-white shadow-lg"
                onClick={() => handleAddItem(item)}
              >
                Add +
              </button>
            </div>
```

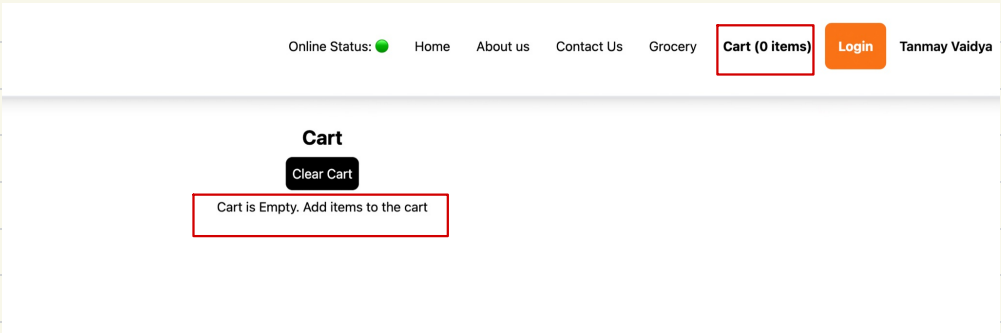
- Let's Create a Cart Component and display the added items into the cart.

Tanmayvaidya

```jsx
const Cart = () => {
  const cartItems = useSelector((store) => store.cart.items);

  const dispatch = useDispatch();
  const handleClearCart = () => {
    dispatch(clearCart());
  };

  return (
    <div className=" text-center m-4 p-4">
      <h1 className="text-2xl font-bold">Cart</h1>
      <div className="w-6/12 m-auto">
        <button
          className="p-2 m-2 bg-black text-white rounded-lg"
          onClick={handleClearCart}
        >
          Clear Cart
        </button>
        {cartItems.length === 0 && (
          <h1>Cart is Empty. Add items to the cart</h1>
        )}
        <ItemList items={cartItems} />
      </div>
    </div>
  );
};

export default Cart;
```



- Now as we have created a dispatch action for clearCart, we will see how it works. So now on clicking on Clear Cart we will see:



* Important Points:

- Whenever you use selector, make sure you subscribe to the right portion of the store

Example:

- Max Efficiency:

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

- Less Efficiency:

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

- In Vanilla Redux (old redux), we don't mutate states. We create a new state and modify the new state and return it. But in new Redux (RTK), we directly modify the states.

- Redux Toolkit uses Immer behind the scenes.

#More Hooks:

* useMemo():
  - It is a hook in React that helps you optimize performance by memoizing the result of expensive computations.
  - It is a React Hook that lets you cache the result of a calculation between re-renders.

Syntax:

```
const cachedValue = useMemo(calculateValue, dependencies)
```

calculatedValue:

- It is the function whose result is memoized.

dependencies:

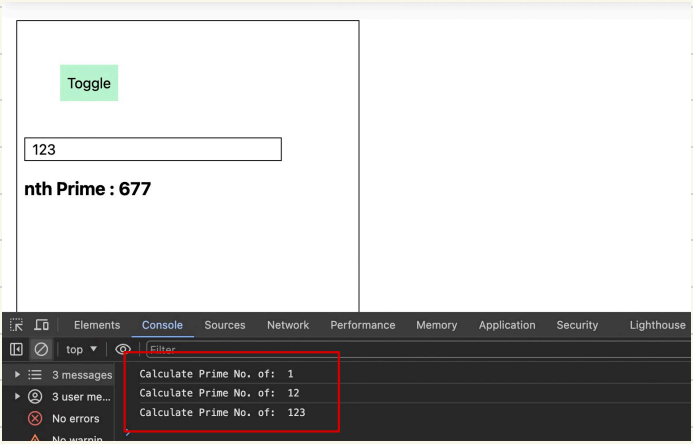
- They are the values that, when changed, trigger recalculation of calculateValue

Example:

- Without useMemo:
  - The provided code allows users to calculate prime numbers by entering a value into the input field. Additionally, it provides functionality to toggle between light and dark modes with a button click.

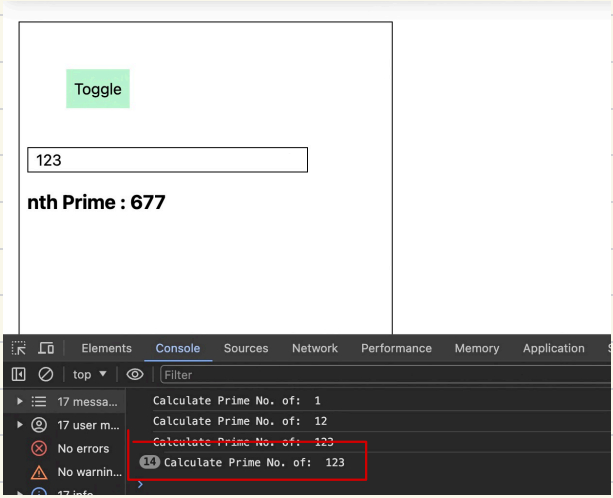
```jsx
import React, { useState, useMemo } from "react";
import { findPrime } from "../utils/helper";

const Demo = () => {
  const [text, setText] = useState(0);
  const [isDarkTheme, setIsDarkTheme] = useState(false);

  const prime = () => {
    console.log("Calculate Prime No. of: ", text);
    return findPrime(text);
  };

  return (
    <div
      className={
        "m-4 p-2 w-96 h-96 border border-black " +
        (isDarkTheme && "bg-gray-900 text-white")
      }
    >
      <div>
        <button
          className="m-10 p-2 bg-green-200"
          onClick={() => setIsDarkTheme(!isDarkTheme)}
        >
          Toggle
        </button>
      </div>
      <div>
        <input
          className="border border-black w-72 px-2"
          type="number"
          value={text}
          onChange={(e) => setText(e.target.value)}
        />
      </div>
      <div>
        <h1 className="mt-4 font-bold text-xl">nth Prime : {prime()}</h1>
      </div>
    </div>
  );
};

export default Demo;
```

- So as toggling and prime no are not related. So it was expected that change on one will not affect the other.
- But the output is different.



Before Clicking on toggle button

- Now we will click on toggle. (I'm clicking on toggle many times to show the real difference.



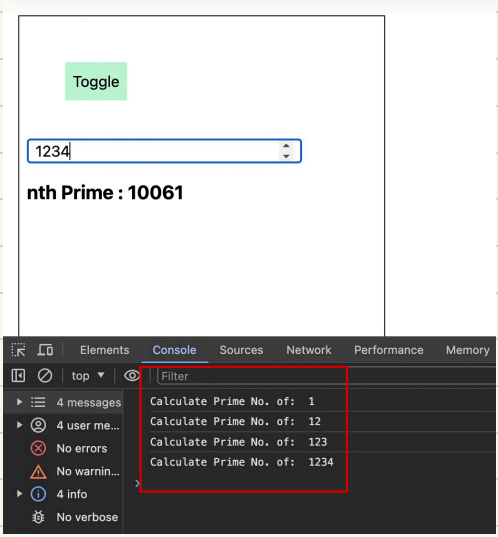
After Clicking on toggle button for multiple times

- As you can see above that even our prime calculation is not dependent on the toggle button, it is still re-rendering which is a major performance issue for heavy computation.
- The reason behind this is the reconciliation process, as whenever the state changes, it will re-render the whole component. Here due to toggle state, it re-renders the component .

- With useMemo:

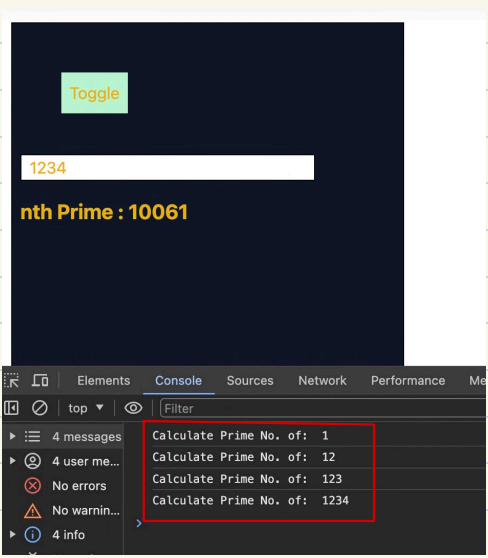
```jsx
import React, { useState, useMemo } from "react";
import { findPrime } from "../utils/helper";

const Demo = () => {
  const [text, setText] = useState(0);
  const [isDarkTheme, setIsDarkTheme] = useState(false);

  const prime = useMemo(() => {
    console.log("Calculate Prime No. of: ", text);
    return findPrime(text);
  }, [text]);

  return (
    <div
      className={
        "m-4 p-2 w-96 h-96 border border-black " +
        (isDarkTheme && "bg-gray-900 text-yellow-500  ")
      }
    >
      <div>
        <button
          className="m-10 p-2 bg-green-200"
          onClick={() => setIsDarkTheme(!isDarkTheme)}
        >
          Toggle
        </button>
      </div>
      <div>
        <input
          className="border border-black w-72 px-2"
          type="number"
          value={text}
          onChange={(e) => setText(e.target.value)}
        />
      </div>
      <div>
        <h1 className="mt-4 font-bold text-xl ">nth Prime : {prime} </h1>
      </div>
    </div>
  );
};

export default Demo;
```

- Without clicking on toggle, just entering into input:



Before Clicking on toggle button

- Now we will click on toggle. (I'm clicking on toggle many times to show the real difference.



After Clicking on toggle button for multiple times

- Now, even after state change of toggle and re-rendering of component, prime is not calculated.
- It would be calculated only we we change prime input, so we wrote [text] in the dependencies.
- Here we are memoizing / caching the heavy operations which help in preformance gain.

* useCallback():

- It is a hook in React that helps optimize performance by memoizing a callback function.
- It is a React Hook that lets you cache a function definition between re-renders

Syntax:

```
const cachedFn = useCallback(fn, dependencies)
```

fn:

- It is the function to be memoized.

dependencies:

- They are the values that, when changed, trigger the recreation of the memoized function.

Tanmayvaidya

Q. Comparison between useMemo and useCallback

- **useMemo:**
  - Primarily used for memoizing expensive computations and returning a cached value.
  - Used when you want to memoize the result of a computation and use it elsewhere in your component.

- **useCallback:**
  - Primarily used for memoizing functions to avoid unnecessary re-creations.
  - Used when you want to memoize a function to prevent unnecessary re-renders in child components that depend on that function.

Tanmayvaidya

Tanmayvaidya

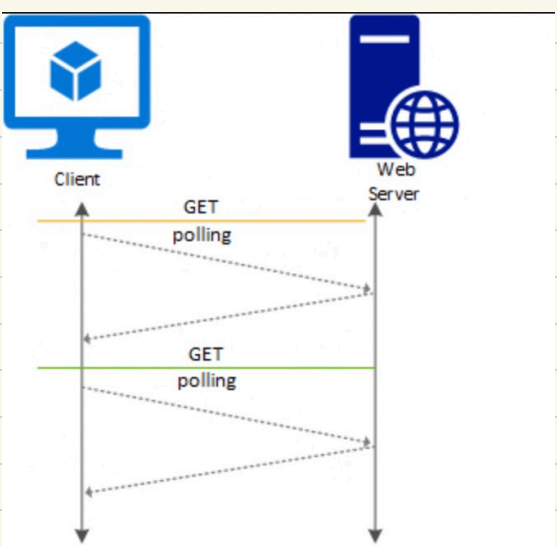
#Few Important Concepts:

* API Polling:

- API polling, also known as polling, is a technique used in software development to retrieve updated data from a server at regular intervals.
- It involves sending periodic requests to an API endpoint to check for new or updated information.

Working of API Polling:

- The client (such as a web browser or a mobile app) sends a request to the server's API endpoint.
- The server processes the request and sends back a response with the requested data.
- After a certain period of time, the client sends another request to the server, repeating the process to fetch any new or updated data.



Usage:

- API polling is commonly used in scenarios where real-time updates are not necessary or practical.
- Weather updates, Youtube Live Chats, etc are the examples where API Polling is used.

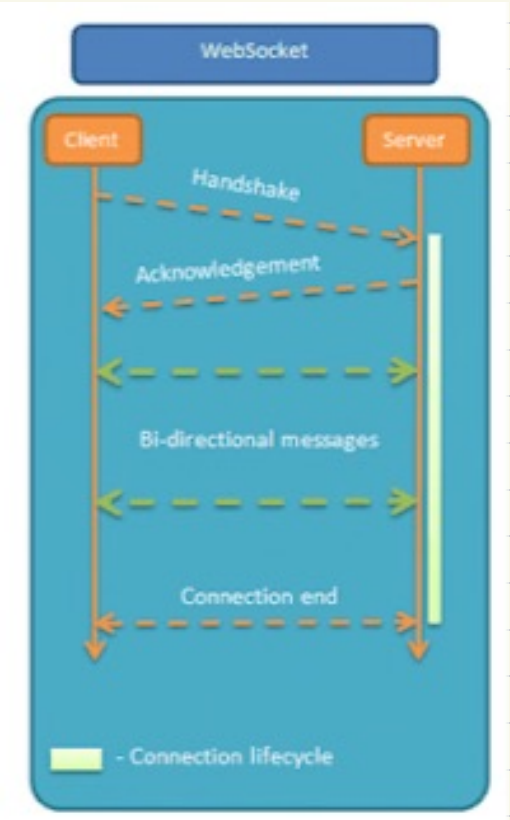
* WebSocket:

- WebSocket is a communication protocol that provides full-duplex communication channels over a single, long-lived connection between a client and a server.
- Unlike traditional HTTP requests, which are unidirectional and stateless, WebSocket allows for bidirectional communication, enabling real-time data transfer between client and server.

Working of WebSocket:

- WebSocket connection initiation involves an HTTP handshake, indicating the client's intention to upgrade to the WebSocket protocol.
- Upon successful handshake, the application-layer protocol is upgraded from HTTP to WebSockets, utilizing the existing TCP connection.
- HTTP is then excluded from further communication, and both client and server can exchange data using the WebSocket protocol.
- This establishes a persistent, full-duplex channel, enabling bidirectional communication without reliance on HTTP.

Tanmayvaidya



Usage:

- WebSocket is particularly useful for real-time applications where low latency, bidirectional communication, and efficient data transfer are essential.
- Chat applications, Online gaming, Stock trading platforms, etc are the examples where WebSocket is used.

* Browser Tab Isolation:

- Each tab in a web browser operates within its own isolated environment, including its own memory allocation.
- This isolation ensures that the activities and resources utilized within one tab do not directly affect the performance or stability of other tabs.
- Overall, the browser's ability to provide memory and resource isolation for each tab contributes to a more secure, stable, and efficient browsing experience, ensuring that activities in one tab do not negatively impact the performance or stability of other tabs.

Tanmayvaidya