**React.js**

React is a JavaScript library for building user interfaces, developed by Facebook. Released in 2013.

Hello World From HTML:

<body>

    <div id="root">

        <h1>Hello World From HTML!</h1>

    </div>

</body>

Hello world from JS:

<body>

    <div id="root"></div>

    <script>

        const heading = document.createElement("h1"); //we want to create h1 tag in the div

        heading.innerHTML = "Hello World From JS!!";

        const root = document.getElementById("root"); //root, i.e., a block where the h1 tag will be created

        root.appendChild(heading); //now heading is appended to the root

    </script>

</body>

Hello World from React:

For react we need cdn(content delivery network) link. It is one of the many ways to use React inside our code. And using cdn is not the recommended way of doing it. (We will be using npn for React & React DOM (when we can have the react on our system as a package, then why to call it using cdn links and making it costly in terms of time? Also, the versions keep on changing, so we will have to modify the link as well, but when we use npm, package.json & pacakage-lock.json, keeps the version record for us))

A **CDN**, or Content Delivery Network, is a network of servers distributed across various locations worldwide. Its purpose is to deliver web content, such as images, videos, scripts, stylesheets, and other assets, to users more efficiently and reliably.

They are widely used by websites and web applications of all sizes to deliver content quickly and reliably to users around the world.

    <script crossorigin src="https://unpkg.com/react@18/umd/react.development.js"></script>

    <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>

Crossorigin: The crossorigin attribute is used in HTML to specify how the browser should handle requests for resources that are hosted on a different domain (cross-origin requests).

The crossorigin attribute can be added to certain HTML elements like <script>, <img>, <video>, <audio>, <link>, and <iframe>. It allows you to control how the browser handles the crossorigin requests for those resources.

Possible values for the crossorigin attribute:

1. anonymous: Requests will not include any credentials such as cookies or HTTP authentication.
2. use-credentials: Requests will include credentials such as cookies or HTTP authentication if the requesting origin is allowed to access the resource. This value allows for more secure requests but requires proper CORS (Cross-Origin Resource Sharing) configuration on the server which you are trying to access.

*Normal script elements pass minimal information to the [window.onerror](https://developer.mozilla.org/en-US/docs/Web/API/Window/error_event" \o "window.onerror) for scripts which do not pass the standard*[*CORS*](https://developer.mozilla.org/en-US/docs/Glossary/CORS)*checks. So use crossorigin, to pass the information which helps with CORS checks.*

*An "origin" in web development is defined by the combination of the following components of a URL:*

*Scheme: The protocol used, such as "http" or "https".*

*Domain: The hostname of the server, such as "example.com".*

*Port: The port number, if specified, such as ":8080".*

*The URL "https://www.example.com/page1" and "https://www.example.com/page2" have the same origin because they have the same scheme ("https"), domain ("*[*www.example.com*](http://www.example.com/)*"), and port (default port for HTTPS, which is 443).*

*The URL "https://api.example.com/data" and "https://www.example.com/page" have different origins because they have different domains ("api.example.com" vs. "*[*www.example.com*](http://www.example.com/)*"), even though they use the same scheme and default port for HTTPS.*

When your web page wants to get something like an image or a script from another website (which is a different "origin" or domain), the browser needs to decide if it's okay to do that. It's like asking, "Can I use this?"

The crossorigin attribute is like telling the browser how it should ask for permission. There are two options:

"anonymous" means the browser doesn't send any special information along with the request. It's like saying, "I'm just looking, not sharing any personal details."

"use-credentials" means the browser might send cookies or other authentication information if it has them. It's like saying, "Hey, I'm allowed to access this, here's my ID."

So, crossorigin helps control how the browser fetches resources from other places, keeping things secure while allowing your web page to use stuff from different websites.

<body>

    <div id="root">

        <h1>Hello</h1>

    </div>

    <script crossorigin src="https://unpkg.com/react@18/umd/react.development.js"></script>

    <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>

    <script>

        const heading = React.createElement("h1", {}, "Hello World from React!");

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

        root.render(heading);

    </script>

</body>

Here, the React code creates an h1 tag. It replaces all the contents present inside the id = “root”, i.e. the output of the above code is : **Hello World From React!**

And Not: **Hello**

**Hello World From React!**

Here, React.createElement() creates an object and not any tag, having all the details, like key, props (props has the information of the content which will go inside the tag and the attributes of the tag) etc

We have 2 cdn links here, one is React development and other is react dom development. React development is core of React. React DOM, is the react library which is used for DOM manipulation.

We have 2 different files, they are not clubbed to make one file, because react doesn’t only work on browsers, it also works in phone as react Native, and other places as well, so there are different functionalities which happen on phone and browsers. So, one file has all the core of React, and other file is different for different platforms.

Now, creating an external JS file.

<body>

    <div id="root"></div>

    <script crossorigin src="https://unpkg.com/react@18/umd/react.development.js"></script>

    <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>

    <script src="App.js"></script>

</body>

Out JS file should come after the cdn links, as we are using React, so React cdn links should be processed first.

App.js =>

const heading = React.createElement("h1", { id: "heading" }, "Hello World From React");

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

root.render(heading);

It will create an h1 tag(having id = heading) inside our div with id = root

/\*

<div id = "parent">

    <div id = "child1">

        <h1>I'm h1 tag</h1>

        <h2>I'm h2 tag<h2>

    </div>

</div>

\*/

const parent = React.createElement("div", { id: "parent" },

React.createElement("div",

{ id: "child1" },

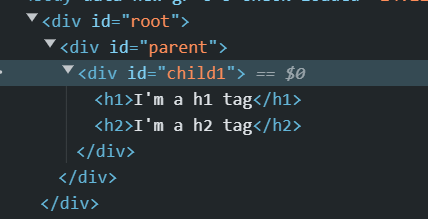
[React.createElement("h1", {}, "I'm a h1 tag"),

React.createElement("h2", {}, "I'm a h2 tag")]

));

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

root.render(parent);



All the divs are injected through react inside root div.

/\*

<div id = "parent">

    <div id = "child1">

        <h1>I'm h1 tag</h1>

        <h2>I'm h2 tag<h2>

    </div>

    <div id = "child2">

        <h1>I'm h1 tag</h1>

        <h2>I'm h2 tag<h2>

    </div>

</div>

\*/

const parent1 = React.createElement("div", { id: "parent" },

[React.createElement("div",

{ id: "child1" },

[React.createElement("h1", {}, "I'm a h1 tag"),

React.createElement("h2", {}, "I'm a h2 tag")]

), React.createElement("div",

{ id: "child2" },

[React.createElement("h1", {}, "I'm a h1 tag"),

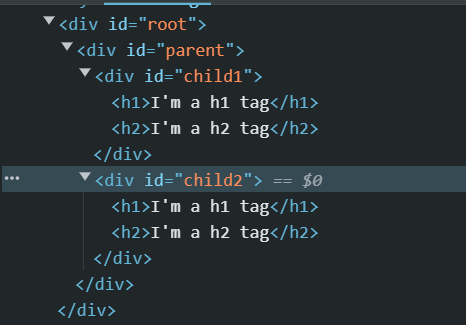
React.createElement("h2", {}, "I'm a h2 tag")]

)])

;

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

root.render(parent1);



**Assignment 1:**

**- What is `Emmet`?**

toolkit for web developers

provides shortcuts and abbreviations, known as Emmet abbreviations or snippets, to quickly generate code snippets and reduce typing time.

Ex. typing div and then pressing Tab expands into <div></div>

**- What is `CDN`? Why do we `use` it?**

A **CDN**, or Content Delivery Network, is a network of servers distributed across various locations worldwide.

Its purpose is to deliver web content, such as images, videos, scripts, stylesheets, and other assets, to users more efficiently and reliably.

They are widely used by websites and web applications of all sizes to deliver content quickly and reliably to users around the world.

**- Why is `React known as React`?**

React is named for its core concept of "reacting" to changes in data.

It builds user interfaces that dynamically "react" to changes, making them interactive and responsive.

Example, a counter, when you click a button to increase the count, React "reacts" by updating the displayed count on the screen without needing to refresh the entire page. This responsiveness is what makes React powerful for building modern web applications.

**- What is `crossorigin in script tag`?**

The crossorigin attribute is used in HTML to specify how the browser should handle requests for resources that are hosted on a different domain (cross-origin requests).

**- What is difference between `React and ReactDOM`?**

`React` is a JavaScript library for building User Interfaces whereas `ReactDOM` is also JavaScript library that allows `React to interact with the DOM`.

React plans out how the user interface should be structured (how our webpage will look and how it will behave when someone tries to interact with it), and ReactDOM takes those plans and brings them to life on the web so users can see and use your application. They work together to make web development easier and more efficient!

DOM stands for Document Object Model.

It is a tree like representation of all the html objects/elements (p, a, div)

It's a programming interface provided by web browsers that allows developers to interact with the structure and content of web pages using code.

the DOM converts the structure of an HTML document into a tree of objects, where each object represents a part of the document, like an element or a piece of text. This tree structure allows developers to manipulate, access, and modify different parts of the web page using languages like JavaScript.

**- What is difference between `react.development.js` and `react.production.js` files via CDN?**

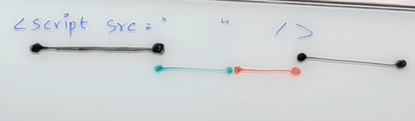
|  |  |  |
| --- | --- | --- |
|  | **react.development.js** | **react.production.js** |
| Purpose | Development purposes | Production, as this file is optimised for production use |
| Debugging Information | Includes additional debugging information, warnings, etc. | Doesn’t have debugging information and warnings |
| File Size | Larger due to included debugging code | Smaller due to removal of debugging code |
| Performance | May have slightly slower performance due to debugging code | Optimized for faster performance |
| Recommended Use | Development environments where debugging is essential | Live or production environments where performance is critical |

**Async vs Defer attribute in script tag**

These are used to control the loading and execution behaviour of external JavaScript files

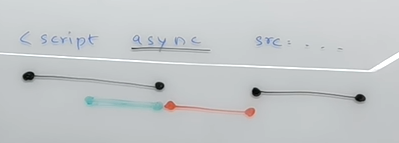
1. In normal scenario : <script src = ‘…’> </script>

Html parsing is done is by the browser, and as soon as the browser encounters the script tag, it stops the html parsing and it goes to the source mentioned in the script tag, fetches the data, and executes the data then and there, and then the html parsing continues.



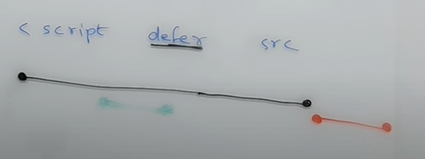
1. Using async : <script async src = ‘…’> </script>

Html parsing is being done, during that parsing browser encounters the script tag, then it goes to the mentioned source, but the html parsing is still being done in parallel (in background), once the browser fetches the data, it is executed, now at the time of execution, the html parsing stops, and once execution is completed, html parsing continues.



1. Using defer : <script defer src = ‘…’> </script>

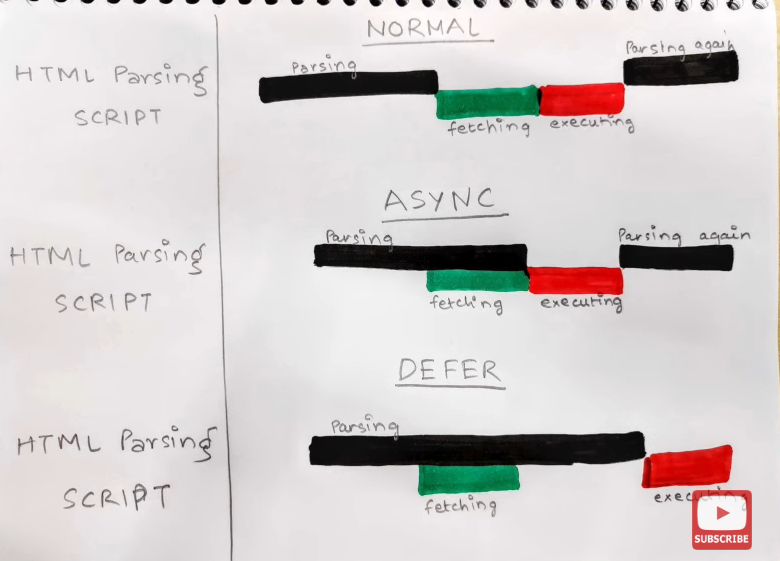
Html parsing is being done, during that parsing browser encounters the script tag, then it goes to the mentioned source, but the html parsing is still being done in parallel, the browser fetches the data, but it will be executed once the html parsing is done, once done, execution will take place

****

Use async when the script file does not depend on the other scripts for its execution and can be executed independently. (if there are multiple script tags having async)

Multiple scripts with the defer attribute are downloaded concurrently, but they are executed in the order they appear in the document.

Use defer when the order of execution matters (e.g., scripts that depend on other scripts).

****

**Difference between a `Library and Framework`?**

A library is a collection of pre-written code that provides specific functionality, which developers can use in their projects. On the other hand, a framework is a structured set of rules and conventions that provides a skeleton for building applications. It dictates the flow and structure of the code, and developers plug their code into the framework's predefined structure.

A `library` is a collection of packages that perform specific operations whereas a `framework` contains the basic flow and architecture of an application. The major difference between them is the complexity. Libraries contain a number of methods that a developer can just call whenever they write code. React js is library and Angular is Framework.

The `framework` provides the flow of a software application and tells the developer what it needs and calls the code provided by the developer as required. If a `library` is used, the application calls the code from the library.

**Npm**: it is a package manager.

It is a standard repository for all the packages.

It will manage all the packages that we will install on our system. Sometimes, these packages are also called as dependency.

Give command:

npm init ----to initialise npm

give answers to all the questions asked after this command.

package name: (assignment-3)

version: (1.0.0)

description: Assignmnet 3 questions

entry point: (app.js) index.html

test command: jest

git repository:

keywords: Assignment3,Nested header element, JSX, Component Composition

author: Raksha Sharma

license: (ISC)

About to write to C:\Users\asus\Downloads\HTML\_CSS\_GIT\_JS\React\react\Assignment 3\package.json:

{

"name": "assignment-3",

"version": "1.0.0",

"description": "Assignmnet 3 questions",

"main": "index.html",

"scripts": {

"test": "jest"

},

"keywords": [

"Assignment3",

"Nested",

"header",

"element",

"JSX",

"Component",

"Composition"

],

"author": "Raksha Sharma",

"license": "ISC"

}

Is this OK? (yes)

This will create package.json file

Package.json is a configuration for npm. It keeps the track of all the dependencies, all the packages that our code is using

A **bundler** is a tool used in web development to combine multiple separate files, typically written in languages like JavaScript, CSS, and HTML, into a single file or a few files that can be served to the browser. These files are optimized for production use on websites or web applications.

Popular bundlers include *webpack, Parcel, and Rollup.*

We will be using Parcel.

Two types of dependencies/package:

1. dev dependencies: it is generally required in development phase (npm install -D parcel)
2. normal dependencies: used in production as well

npm install -D parcel

this command will download the parcel from npm which is a repo of packages.

In package.json we have

  "devDependencies": {

    "parcel": "^2.12.0"

  }

There is caret sign in front of the version.

~version **“Approximately equivalent to version”**, will update you to all future patch versions, without incrementing the minor version. ~2.12.0 will use releases from 2.12.0 to <2.13.0

^version **“Compatible with version”**, will update you to all future minor/patch versions, without incrementing the major version. ^2.12.0 will use releases from 2.12.0 to <3.0.0.

In general, if you value stability and want to minimize the risk of introducing breaking changes, you may prefer the tilde version range. If you prioritize staying up-to-date with bug fixes and minor updates while avoiding major version upgrades, the caret version range may be a better fit.

One more file got created, after installing parcel. It is package-lock.json . It has a version of Parcel without ~(tilde) or ^(caret), i.e. it has an exact version details. It keeps track of all the versions of the dependencies and transitive dependies.

One folder got created, node\_modules, it has all the code of parcel (or dependencies in general). We have multiples files in the folder, other than Parcel, it is because Parcel has its own dependencies, and those dependencies have their own dependencies, and they have their dependencies… this is known as *transitive dependencies*. So, node\_modules is a collection of dependencies.

We don’t need to push this node\_modules folder to our git repository, as the package.json and package-lock.json can automatically create the node\_modules folder, as they have all the dependency details. So whatever you can regenerate, don’t put on github.

If I delete npm\_modules from my system, I can regenerate the folder again

To regenerate: npm install

To execute a package:

npx parcel index.html //for dev page

npx parcel build index.html //for Production page

after executing the parcel, our webpage will be hosted on a server : <http://localhost:1234>

this command says: execute npm package parcel with source as index.html

installing react & react dom:

npm install react

npm install react-dom

we have installed react & react dom with normal dependencies

Now we will remove the cdn links, and run: npx parcel index.html

But we don’t see nay ouput coming from React & an error on console that “React is not defined”

It is because we haven’t told our code how to understand react, so we need to add import React from “react”. First react is the keyword from React.createElement and all other React functions, and second react is the package which is present in the npm\_modules folder

import React from "react";

import ReactDOM from "react-dom/client";

But it throws error, that Browser scripts cannot have imports or exports

Because our browser is considering our javascript as ordinary javascript, but we are using a package, and importing it into our JS, so we need to tell the same to the browser.

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

Type = module causes the code to be treated as a JavaScript module.

Add this in html code.

I was getting an error “🚨 Build failed. Error: Expected content key de1e4a02ec63c4eb to exist”

For this I deleted .parcel-cache & dist folders, and again ran npx parcel index.html

 Parcel

- creating Dev Build

- Host webpage on Local Server

- Automatically Refreshes page => HMR = Hot Module Replacement (Parcel uses File Watching Algorithm - written in C++)

- Parcel caching things as well, hence faster builds

- Image optimization

- Minification of files

- Bundling

- Compress the files

- Consistent hashing

- Differential bundling => to support older browsers

- Code splitting

- Diagnostic => errors handling

- Tree Shaking - remove unused code

- Different Dev & production bundle

We need to tell what all browsers can support our project:

It needs to be configured in package.json. Add below code in package.json

  "browserslist":[

    "Last 2 Chrome version",

    "Last 3 Firefox version"]

<https://browserslist.dev/?q=bGFzdCAyIHZlcnNpb25z>

Can check the % of people using these versions

**Assignment 2**

**- What is `npm`?**

It is a package manager for JavaScript and NodeJS

It is a standard repository for all the packages.

It is used for installing, managing and sharing code packages and dependencies.

Its alternate is yarn.

npm init

`npm init -y` can be used to skip the setup step, `npm` takes care of it and creates the `package.json` json file automatically , but without configurations.

**- What is `Parcel/Webpack`? Why do we need it?**

Parcel/Webpack/Rollup are bundlers mostly used for Javacript & Typescript.

Help in minifying, cleaning, and making code compact for easier transfer during server requests and responses while the functionality of the code remains unchanged.

Minify Code: bundlers reduce the size of code by removing unnecessary characters like comments, new lines, and white spaces.

Organize Code: Bundlers organize multiple JavaScript files into a single bundle for efficient loading.

A JavaScript bundler can be used when your project becomes too large for a single file or when you're working with libraries that have multiple dependencies.

Installation commands:

npm install -D parcel

`-D` is used for development and as a development dependency.

- Parcel Commands:

- For development build:

npx parcel <entry\_point>

- For production build :

npx parcel build <entry\_point>

**- Why is `.parcel-cache`?**

`It stores information about your project when parcel builds it, so that when it rebuilds, it doesn't have to re-parse and re-analyze everything from scratch. It's a key reason why parcel can be so fast in development mode.

**- What is `npx`?**

`npx` stands for `Node Package eXecute`. It is simply an `npm` package runner. It allows developers to execute any Javascript Package available on the npm registry without even installing it. It comes bundled with npm (Node Package Manager) starting from version 5.2.

For example, if you want to run a command from a package named parcel, you can simply use npx parcel command in your terminal. npx will download the latest version of parcel from the npm registry, execute the specified command from that package, and then discard the downloaded package afterwards.

This on-the-fly execution makes it convenient for running occasional commands from packages that you don't need to keep installed in your project dependencies or globally on your system.

**- What is difference between `dependencies vs devDependencies`?**

`"dependencies"` : Packages required by your application in production.

`"devDependencies"` : Packages that are only needed for local development and testing.

**- What is Tree Shaking?**

`Tree shaking`, also known as `dead code elimination`, is the practice of `removing unused code in your production build`. It's important to ship as little code to your end-users as possible. By statically analyzing our source code, we can determine what's not being used and exclude it from our final bundle.

**- What is Hot Module Replacement?**

`Hot Module Replacement (HMR)` exchanges, adds, or removes modules while an application is running, without a full reload. This can significantly speed up development in a few ways: Retain application state which is lost during a full reload. Save valuable development time by only updating what's changed.

HMR saves you time by only updating the pieces you've changed.

When you're working on your website or web app, HMR lets you make changes to your code and see them instantly without needing to refresh the whole page. HMR swaps out the old code with the new code while the page is still running, so you can see the effects of your changes right away.

**- List down your favourite 5 superpowers of Parcel and describe any 3 of them in your own words.**

**Hot Module Replacement (HMR)**: When you're working on your website or web app, HMR lets you make changes to your code and see them instantly without needing to refresh the whole page. HMR swaps out the old code with the new code while the page is still running, so you can see the effects of your changes right away.

**Local Server**: A local server is like a mini-computer that runs on your own computer. It's not connected to the internet, so only you can access it. When you're building a website or web app, you can use a local server to see how your project looks and works without putting it online. With a local server, you can view your website in a web browser just like you would if it were live on the internet. This allows you to test your code, check for errors, and see how everything fits together before sharing it with the world. It's a handy tool for web developers that helps streamline the development process and catch any issues early on.

**File Watching Algorithm**: Parcel's file watching algorithm, written in C++, efficiently monitors changes to files in the project directory, ensuring that updates are detected quickly and accurately. This enhances the development experience by providing fast feedback on code changes.

**Image Optimization**: Automatically optimizes images during the build process, reducing file sizes without sacrificing quality. This improves page load times and overall performance of web applications.

**Tree Shaking**: Removes unused code from the final bundle, reducing its size and improving load times. This optimization ensures that only the necessary code is included in the production bundle, leading to a more efficient and lightweight application.

**- What is `.gitignore`? What should we add and not add into it?**

gitignore file tells Git which files to ignore when committing your project to the GitHub repository. gitignore is located in the root directory of your repo. / will ignore directories with the name.

In our code we shouldn't add the files, which we can re-generate in future e.g, `node\_modules`, `dist` etc.

**- What is the difference between `package.json` and `package-lock.json` files?**

Both of these files have the same format, and perform similar functions in the root of a project. The difference is that `package-lock. json` cannot be published, and it will be ignored if found in any place other than the root project.

The package. json is used for more than dependencies - like defining project properties, description, author & license information, scripts, etc. . It holds information like project details, who made it, what it's allowed to do, and importantly, what other tools the project needs to work correctly.

The package-lock. json is solely used to lock dependencies to a specific version number. It keeps track of the exact versions of all the tools the project needs. This helps ensure that everyone working on the project is using the same versions of those tools.

**- Why should I not modify `package-lock.json`?**

Editing the package-lock.json file manually is not recommended because it is a generated file, meaning it is automatically created based on the dependencies and their versions listed in the package.json file. Here's why you shouldn't edit it manually:

Generated File: The package-lock.json file is automatically generated by npm or yarn when you install or update dependencies in your project. It is meant to accurately reflect the current state of your project's dependencies based on the package.json file.

Exact Versions: package-lock.json contains specific version numbers for each dependency and its dependencies. These versions are determined based on the dependency tree and are crucial for ensuring that everyone working on the project uses the same versions of dependencies. Manual editing could disrupt this synchronization.

Risk of Errors: Manually editing package-lock.json can introduce errors or inconsistencies, leading to unexpected behavior or issues when installing or updating dependencies. It's easy to make mistakes, such as typos or incorrect version numbers, which can cause problems in your project.

When you first initialize a project with npm or yarn (npm init or yarn init), or when you install dependencies for the first time (npm install or yarn install), the package-lock.json file is generated.

**- What is `node\_modules`? Is it a good idea to push that on git?**

The `node\_modules` folder contains generated code. This is not code you've written and you should never make any updates to the files inside Node modules because there's a pretty good chance they'll get overwritten next time you install some modules.

It is better to not commit the `node\_modules` folder, and instead add it to your `.gitignore` file.

Here are all the reasons why you shouldn't commit it:

The node\_modules folder has a massive size (up to Gigabytes).

It is easy to recreate the node\_modules folder via packages.json

**- What is the `dist` folder?**

The `/dist` stands for distributable. The /dist folder is used to store files that are ready for deployment or distribution to users. It typically contains optimized, production-ready files that have been processed and compiled from the project's source code.

**- What is `browserlists`?**

It is a configuration or setting used in web development to specify which browsers and browser versions your project should support.

**- Script types in html(MDN Docs)**

async

crossorigin

defer

src

type

so, we were using npx parcel index.html to execute our parcel package.

But it can also be done using scripts, and it is the recommended way of doing it.

So, in package.json, we need to write below code in scripts:

 "scripts": {

    "start": "parcel index.html",

    "build" : "parcel buld index.html",

    "test": "jest"

  }

Start for development build and build for production.

In start we have written the command that needs to be executed for development build.

These scripts, start & build, are executed by running commands:

npm run start or npm start // for development

npm run build // for production. Please note, we cant run npm build, it is not a proper command

import React from "react";

import ReactDOM from "react-dom/client";

const heading = React.createElement("h1", {id:"heading"}, "Namaste Javascript"); //for creating an object

const root = ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

root.render(heading);

React.createElement() : it creates an object. Render takes this object and converts it into an HTML Element and push it to the browser. It will **replace** everything present inside the id “root”, with whatever is being rendered.

This syntax of React.createElement is too difficult to read when we have nested elements. So Facebook developers created a new syntax called as **JSX.**

JSX (JavaScript XML) is a syntax extension for JavaScript that allows you to write **HTML-like** code directly within JavaScript.

//JSX – HTML-like Syntax or XML Syntax

const JSXHeading = <h1>Namaster React in JSX</h1> //React element

const rootJSX= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

rootJSX.render(JSXHeading);

Please note: JSX is not HTML, its like HTML.

JSX is just a syntax.

JSXHeading creates a React object, same as that of React.createElement

const heading = React.createElement("h1", {id:"heading"}, "Namaste React"); //for creating an object

console.log(heading);

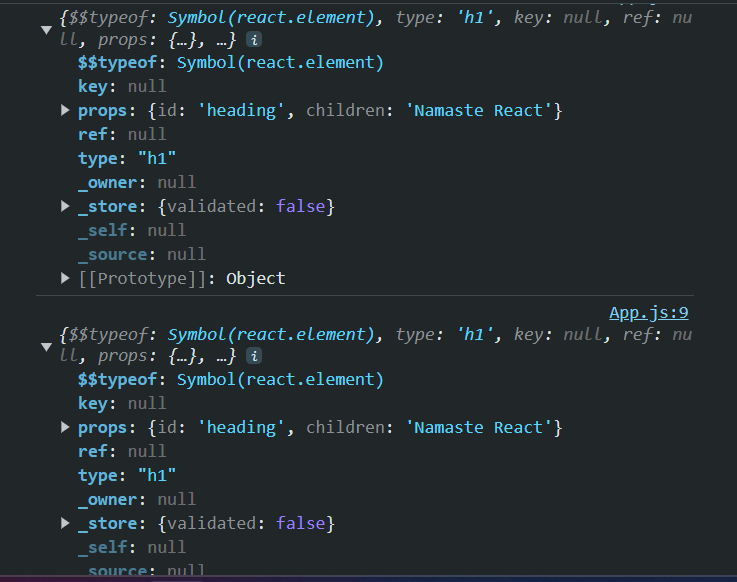
//JSX => React.createElement => React Element or JS Object => Rendering makes it a HTML Element

const JSXHeading = <h1 id = "heading">Namaster React in JSX</h1>;

console.log(heading);

const rootJSX= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

rootJSX.render(JSXHeading);



So, both creates React Objects, just syntax differs.

JSX is not a proper JS syntax, JS cant read and understand it. But parcel is making it understand and print the output on the webpage. So, what happens is, Parcel asks **Babel** (another package) to transpile (converts in the language understood by the JS Engine or React) the code before it reaches JS engine.

JSX transpiles the code to React.createElement

const heading = React.createElement("h1", {id:"heading"}, "Namaste React"); //for creating an object

console.log(heading);

//JSX => Babel transpiles it to React.createElement => React Element or JS Object => Rendering makes it a HTML Element

const JSXHeading = <h1 id = "heading" className = "heading">Namaster React in JSX</h1>;

console.log(heading);

const rootJSX= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

rootJSX.render(JSXHeading);

JSX is not HTML, we can see that from above example. In HTML we write class, to assign class attribute to any element, but in JSX, we use className.

**We have to use camel case to define the attributes in JSX.**

const JSXHeading = (<h1 id = "heading" className = "heading">

    Namaster React in JSX

    </h1>);

Use () when giving the html-like code in multiple lines.

**Everything is a component in React. For eg in a webpage, Search bar is a component, button is a component, header, title, input box, footer,etc everything is a component.**

Two types of components in React:

1. Class based component (old way of writing code) => use js classes
2. Functional component (currently used) => use js functions

A React component is a reusable building block or piece of UI (User Interface) that encapsulates logic and UI elements together.

The "init case" or "Title Case" convention is commonly used for naming React components, otherwise it throws error.

Functional Component:

It is a normal JS function, which returns some JSX(an html-like code). OR We can say that functional Component is a function which returns React element. (JSX gets transpiled to React.createEelemnt which creates a React element)

//React component

const HeadingComponent = () => {

    return <h1>This is a React Functional component</h1>

}

const fn = () => {

    return true;

}

const fn1 = () => true;

// both fn and fn1 are same, fn1 is shorthand

So, above functional component can also be written as:

//React functional component

const HeadingComponent = () => <h1>This is a React Functional component</h1>;

If there are multiple lines, wrap them inside parenthesis.

//React functional component

const HeadingComponent = () => (

<h1>This is a React Functional component</h1>

);

//React functional component

const HeadingComponent = () => (

    <div id = "container">

        <h1 className = "heading">This is a React Functional component</h1>

    </div>

);

This functional react component cant be rendered using previous render() syntax, as we use that syntax for React elements, not React components.

//React functional component

const HeadingComponent = () => (

    <div id = "container">

        <h1 className = "heading">This is a React Functional component</h1>

    </div>

);

const root= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

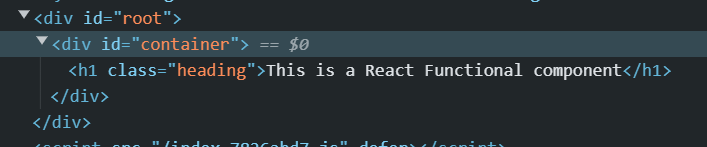
root.render(HeadingComponent); //CANT BE DONEEE!!

So, we need to write as below for react components:

root.render(<HeadingComponent/>);

Bu this syntax, Babel understands that there is a functional component known as HeadingComponent

We got this heading injected inside div having id as root



const TitleComponent = () =>(

    <h1 tabIndex = "5" className = "title">Namaste react Using JSX</h1>

);

const HeadingComponent = () => (

    <div id = "container">

        <TitleComponent/>

        <h1 className = "heading">This is a React Functional component</h1>

    </div>

);

const root= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

root.render(<HeadingComponent/>);

This is known as **React Component Composition.** Component inside another component.



const TitleComponent = () =>(

    <h1 tabIndex = "5" className = "title">Namaste react Using JSX</h1>

);

const Content = () => (

    <h2 className = "para">This is a React Functional component</h2>

)

const HeadingComponent = () => (

    <div id = "container">

        <TitleComponent/>

        <Content/>

    </div>

);

const root= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

root.render(<HeadingComponent/>);

We can use regular functions as well:

const HeadingComponent = function() {

    return(

    <div id = "container">

        <TitleComponent/> //Component inside component

        <Content/>

    </div>

)};

But we will be using arrow function, as it is industry standard.

You can put any valid [JavaScript expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions) inside the curly braces in JSX.

const count = 26;

const HeadingComponent = function() {

    return(

    <div id = "container">

        <TitleComponent/>

        {count}

        <Content/>

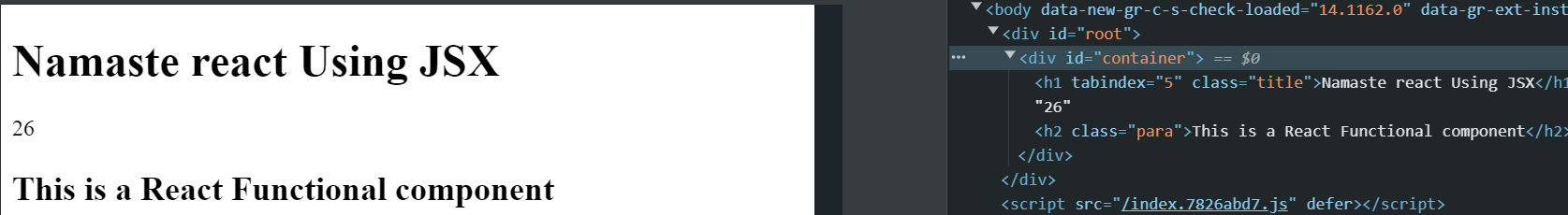
        {console.log("Hi")}

    </div>

)};

const root= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

root.render(<HeadingComponent/>);



const elem = <span>React JS</span>

const HeadingComponent = function() {

    return(

    <div id = "container">

        <TitleComponent/>

        {elem}

        <Content/>

        {console.log("Hi")}

    </div>

)};

const root= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

root.render(<HeadingComponent/>);



const elem = <span>React JS</span> //React element

const title = (               //React Element

    <h1 tabIndex = "5" className = "title">

        {elem} //react element inside React element

        Namaste react Using JSX

        </h1>

);

const Content = () => (         //Function React Component

    <h2 className = "para">This is a React Functional component</h2>

)

const HeadingComponent = function() {

    return(

    <div id = "container">

        {title} //element inside component

        <Content/>

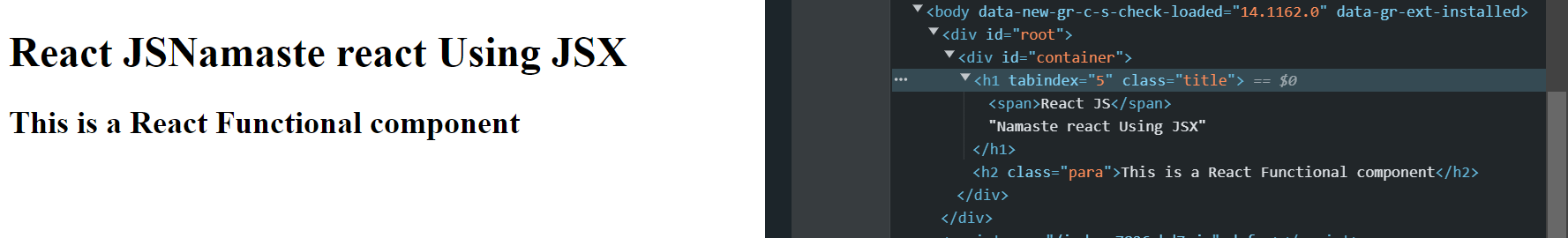
        {console.log("Hi")}

    </div>

)};

const root= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

root.render(<HeadingComponent/>);



const elem = <span>React JS</span> //React element

const Content = () => (         //Function React Component

    <h2 className = "para">This is a React Functional component</h2>

)

const title = (               //React Element

    <h1 tabIndex = "5" className = "title">

        {elem}

        Namaste react Using JSX

        </h1>

);

const container = (

    <div id = "container">

        {title}

        <Content/> //React component inside element

    </div>

)

const HeadingComponent = function() {

    return(

    {container}

)};

const root= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

root.render(container);



const elem = <span>React JS</span> //React element

const Content = () => (         //Function React Component

    <h2 className = "para">This is a React Functional component</h2>

)

const title = (               //React Element

    <h1 tabIndex = "5" className = "title">

        {elem}

        Namaste react Using JSX

        </h1>

);

const container = (

    <div id = "container">

        {title}

        <Content/> // Can also be written as : <Content></Content>

</div>

)

const HeadingComponent = function() {

    return(

        <div>

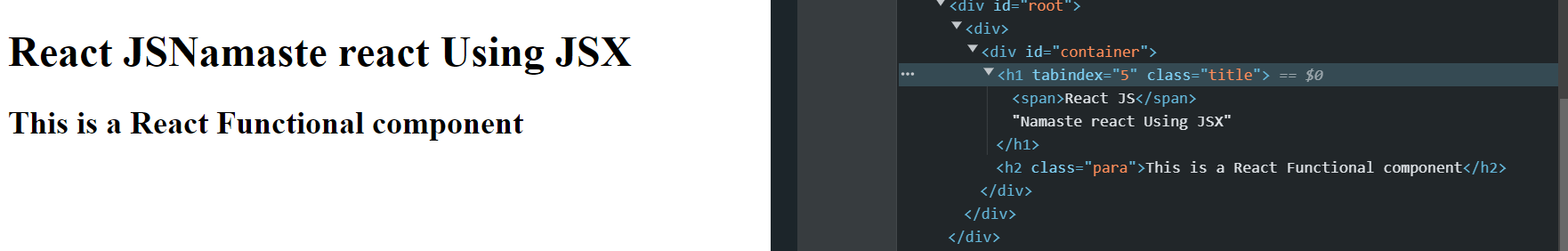
            {container}

        </div>

)};

const root= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

root.render(<HeadingComponent/>);



We can use React element inside react element, react component inside react component, react element inside react component, react component inside react element.

**We can call react function component any number of times**

const container = (

    <div id = "container">

        {title}

        <Content/>

        <Content></Content>

    </div>

)

We can call function components in JSX code, as function components are JS functions only.

import React from "react";

import ReactDOM from "react-dom/client";

const elem = <span>React JS</span> //React element

const Content = () => (         //Function React Component

    <h2 className = "para">This is a React Functional component</h2>

)

const title = (               //React Element

    <h1 tabIndex = "5" className = "title">

        {elem}

        Namaste react Using JSX

        </h1>

);

const container = (

    <div id = "container">

        {title}

        {Content()} //same thing as below

        <Content></Content> //same thing as below

        <Content/>

    </div>

)

const HeadingComponent = function() {

    return(

        <div>

            {container}

        </div>

)};

const root= ReactDOM.createRoot(document.getElementById("root")); //for displaying it on the browser

root.render(<HeadingComponent/>);

**Assignment 3:**

**- What is `JSX`?**

JSX, or JavaScript XML, is a syntax extension for JavaScript that allows us to write HTML-like code directly within JavaScript.

It's primarily associated with React, a popular JavaScript library for building user interfaces.

JSX makes it easier to describe the structure of UI components using familiar HTML tags and attributes, along with embedding JavaScript expressions for dynamic content and logic.

While JSX isn't understood by browsers directly, it's transpiled into regular JavaScript by tools like Babel before being served to the browser.

JSX is a powerful feature that enhances readability, maintainability, and expressiveness when building complex user interfaces in React.

**- Superpowers of `JSX`.**

1. Readability: JSX improves the readability of code by allowing developers to write UI components using HTML-like syntax, making it easier to understand the structure and layout of the user interface.

2. Embedding JavaScript: JSX allows JavaScript expressions to be embedded directly within the HTML-like syntax, enabling dynamic content and logic to be seamlessly integrated into UI components.

const container = (

    <div id = "container">

        {title}

        {Content()} //same thing as below & JS inside JSX

        <Content></Content> //same thing as below

        <Content/>

    </div>

)

3. Component Composition: JSX facilitates component composition, allowing developers to compose smaller, reusable components together to build more complex UIs. This promotes modularity and code reuse.

**- Role of type `attribute` in script tag? What `options can I use` there?**

The type attribute in the <script> tag specifies the script content. It tells the browser how to interpret and execute the script. The default value for the type attribute is "text/javascript", which indicates that the content of the <script> tag is JavaScript code.

However, in HTML5, the type attribute is optional for <script> tags, and if omitted, the browser assumes the content is JavaScript by default.

Here are some common options that can be used with the type attribute:

`text/javascript` : It is the basic standard of writing javascript code inside the `<script>` tag.

### Syntax

```

<script type="text/javascript"></script>

```

- `text/ecmascript` : this value indicates that the script is following the `EcmaScript` standards.

- `module`: This value tells the browser that the script is a module that can import or export other files or modules inside it.

- `text/babel` : This value indicates that the script is a babel type and required bable to transpile it.

- `text/typescript`: As the name suggest the script is written in `TypeScript`.

**- `{TitleComponent}` vs `{<TitleComponent/>}` vs `{<TitleComponent></TitleComponent>}` in `JSX`.**

- `{TitleComponent}`: This value describes the `TitleComponent` as a javascript expression or a variable.

The `{}` can embed a javascript expression or a variable inside it.

- `<TitleComponent/>` : This value represents a Component that is basically returning Some JSX value. In simple terms `TitleComponent` a function that is returning a JSX value.

A component is written inside the `{< />}` expression.

- `<TitleComponent></TitleComponent>` : `<TitleComponent />` and `<TitleComponent></TitleComponent>` are equivalent only when `< TitleComponent />` has no child components. The opening and closing tags are created to include the child components.

**### Example**

```

<TitleComponent>

    <FirstChildComponent />

    <SecondChildComponent />

    <ThirdChildComponent />

</TitleComponent>

```

**### if there are no child elements, then we can write**

```

<TitleComponent />

```

**Building an App:**

1. Decide a layout, make a layout on a paper, or wherever suitable.
2. Decide the components your App going to have
3. Start writing code for each component

const styleCard = {

    background: "#f0f0f0",

}

const RestroCard = () => {

    return (

            <div class="restro-card" style={styleCard}></div>

)

}

**OR**

const RestroCard = () => {

    return (

            <div class="restro-card" style={{

                background: "#f0f0f0",

            }}></div>

)

}

When we have repeating content, we use functional component with props and we pass **props** to the functional component as we pass parameter to any JS function

<RestroCard resName = "Meera Hotel" cuisine = "Indian, Chinese, Continental" rating = "4.3 stars" eta = "23 mins"/>

React will take all these props or properties and wrap them up inside an object and pass them to the functional component as a prop, which is an object.

We can pass any number of props to a functional component.

const RestroCard = (props) => {

    console.log(props);

    return (

            <div className="restro-card" style={{

                background: "#f0f0f0",

            }}>

                <img src="https://www.foodiesfeed.com/wp-content/uploads/2023/06/burger-with-melted-cheese.jpg" alt="restaurant logo" />

                <h3>{props.resName}</h3>

                <h4>{props.cuisine}</h4>

                <h5>{props.rating}</h5>

                <h5>{props.eta}</h5>

            </div>)

}

const BodyComp = () => {

    return (<div className="body">

        <div className="search-container">

            <input type="text" placeholder="Search"></input>

            <button>Search</button>

        </div>

        <div className="restro-container">

        <RestroCard resName = "Meera Hotel" cuisine = "Indian, Chinese, Continental" rating = "4.3 stars" eta = "23 mins"/>

        <RestroCard resName = "Lilac" cuisine = "Indian" rating = "4.7 stars" eta = "17 mins" />

        </div>

    </div>

    )

}

**OR**

const RestroCard = ({resName, cuisine, rating, eta}) => {

    return (

            <div className="restro-card" style={{

                background: "#f0f0f0",

            }}>

                <img src="https://www.foodiesfeed.com/wp-content/uploads/2023/06/burger-with-melted-cheese.jpg" alt="restaurant logo" />

                <h3>{resName}</h3>

                <h4>{cuisine}</h4>

                <h5>{rating}</h5>

                <h5>{eta}</h5>

            </div>)

}

const BodyComp = () => {

    return (<div className="body">

        <div className="search-container">

            <input type="text" placeholder="Search"></input>

            <button>Search</button>

        </div>

        <div className="restro-container">

        <RestroCard resName = "Meera Hotel" cuisine = "Indian, Chinese, Continental" rating = "4.3 stars" eta = "23 mins"/>

        <RestroCard resName = "Lilac" cuisine = "Indian" rating = "4.7 stars" eta = "17 mins" />

        </div>

    </div>

    )

}

**OR**

const RestroCard = (props) => {

    const {resName, cuisine, rating, eta} = props;

    return (

            <div className="restro-card" style={{

                background: "#f0f0f0",

            }}>

                <img src="https://www.foodiesfeed.com/wp-content/uploads/2023/06/burger-with-melted-cheese.jpg" alt="restaurant logo" />

                <h3>{resName}</h3>

                <h4>{cuisine}</h4>

                <h5>{rating}</h5>

                <h5>{eta}</h5>

            </div>)

}

const BodyComp = () => {

    return (<div className="body">

        <div className="search-container">

            <input type="text" placeholder="Search"></input>

            <button>Search</button>

        </div>

        <div className="restro-container">

        <RestroCard resName = "Meera Hotel" cuisine = "Indian, Chinese, Continental" rating = "4.3 stars" eta = "23 mins"/>

        <RestroCard resName = "Lilac" cuisine = "Indian" rating = "4.7 stars" eta = "17 mins" />

        </div>

    </div>

    )

}

**Config driven UI:**

Controlling the UI, how UI looks like, using data or config and the config comes from backend. For example, you want an offer carousal in your website, and offers vary with locations, so config driven UI will be helpful in such cases. We will have different carousals for different locations. For example, we want a background color of some div to be green in Mumbai, yellow in Delhi, or something like that, then we can use config driven UI.

**Deconstructing assignment in JS:**

 const {imgCode,

        altText,

        name,

        cuisines,

        avgRating,

        costForTwo,

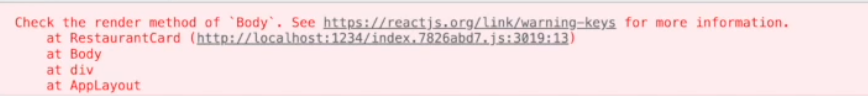
        eta

    } = resData?.data

In the given code snippet, it appears that resData is an object, and data is a property of resData.

* { imgCode, altText, name, cuisines, avgRating, costForTwo, eta }: This is object destructuring syntax. It allows you to extract multiple properties from an object and assign them to individual variables with the same names as the properties.
* resData?.data: This is optional chaining syntax (?.). It is used to safely access nested properties of an object. If resData is null or undefined, or if resData.data is null or undefined, the expression will short-circuit and return undefined instead of throwing an error.

So, the line of code is extracting the properties imgCode, altText, name, cuisines, avgRating, costForTwo, and eta from the data property of the resData object. If resData or resData.data is null or undefined, all these variables will be assigned undefined. Otherwise, they will be assigned the corresponding values from the data object.



<div className="restro-container">

            {

                resList.map((restaurant) => (<RestroCard key = {restaurant.data.id} resData = {restaurant}/>))

            }

Always use a **key** whenever using a looping method (loop, map)

For react all the RestroCards are same, so we need a key property to identify it uniquely.

when rendering a list of elements using the map() function, it's essential to provide each rendered element with a unique key prop. This key prop helps React efficiently update and re-render the list when items are added, removed, or rearranged.

Optimized Rendering: React uses a technique called "reconciliation" to efficiently update the DOM. When you modify a list by adding, removing, or reordering items, React needs a way to identify which elements have changed. By providing a unique key, React can easily identify which elements have been added, removed, or moved, and update the DOM accordingly without re-rendering the entire list.

**Assignment**

**- Is `JSX` mandatory for React?**

`JSX` is an Extension Syntax that allows writing HTML and Javascript together easily in React and is used to create React elements. These elements are then rendered to the React DOM. Each JSX element is just to make use of React easy and for calling React.createElement(component, props, …children) with less work. So, anything that is done with JSX can also be done with just plain JavaScript. So `JSX` is not mandatory but is used for writing better and clean code instead of writing code using `React.CreateElement`.

**- Is `ES6` mandatory for React?**

`ES6` is not mandatory for `React` but is highly recommendable. The latest projects created on React rely a lot on ES6. React uses ES6. ES6 stands for ECMAScript 6. ECMAScript was created to standardize JavaScript, and ES6 is the 6th version of ECMAScript, it was published in 2015.

ES6 is aligned with modern JavaScript development practices and is widely supported by modern browsers and development tools. Therefore, it's highly recommended to leverage ES6 features when building React applications.

**- How can I write `comments` in JSX?**

JSX comments are written as follows:

- `{/\* \*/}` - for single or multiline comments

{/\* A JSX comment \*/}

{/\*

Multi

line

JSX

comment

\*/}

**- What is `<React.Fragment></React.Fragment>` and `<></>`?**

<React.Fragment> and <>...</> are both ways to create a fragment in React. Fragments allow you to group multiple JSX elements without adding extra nodes to the DOM.

<React.Fragment></React.Fragment>:

<React.Fragment> is a built-in component provided by React for creating fragments.

It allows you to wrap multiple JSX elements without adding any additional DOM nodes (DOM nodes refer to the individual elements within the Document Object Model (DOM) tree. In web development, when a web page is loaded in a browser, its structure is represented as a hierarchical tree-like structure called the DOM. Each element in the HTML document, such as <div>, <p>, <span>, etc., is represented as a node in this tree.).

You can use it like a regular component, but it doesn't render any extra HTML elements.

It's especially useful when you need to return multiple elements from a component without wrapping them in a parent element.

`<></>` is the shorthand tag for `React.Fragment`. The only difference between them is that the shorthand version does not support the key attribute.

#### Example

```

return (

<React.Fragment>

<Header />

<Navigation />

<Main />

<Footer />

</React.Fragment>

);

return (

<>

<Header />

<Navigation />

<Main />

<Footer />

</>

);

```

**- What is `Reconciliation` in React?**

`Reconciliation` is the process through which React updates the Browser DOM and makes React work faster. React use a `diffing algorithm` so that component updates are predictable and faster. React would first calculate the difference between the real DOM and the copy of DOM (Virtual DOM) when there's an update of components.

React stores a copy of Browser DOM which is called `Virtual DOM`. When we make changes or add data, React creates a new Virtual DOM and compares it with the previous one. Comparison is done by `Diffing Algorithm`.

React compares the Virtual DOM with Real DOM. It finds out the changed nodes and updates only the changed nodes in Real DOM leaving the rest nodes as it is. This process is called Reconciliation.

(or)

Reconciliation is the process by which React updates the UI to reflect changes in the component state. The reconciliation algorithm is the set of rules that React uses to determine how to update the UI in the most efficient way possible. React uses a virtual DOM (Document Object Model) to update the UI.

{

In React, reconciliation refers to the process of updating the DOM to match the most recent state of a React component tree.

When you make changes to the state or props of a React component, React needs to determine what changes have occurred and update the rendered output accordingly. Reconciliation is the mechanism by which React efficiently computes and applies these changes to the DOM.

Key aspects of reconciliation in React include:

1. Virtual DOM Comparison: React uses a virtual DOM to represent the current state of the UI. When changes occur, React computes the difference (diff) between the previous virtual DOM and the new virtual DOM. This diffing algorithm is a crucial part of the reconciliation process.

2. Efficient Updates: React aims to minimize the number of DOM operations needed to update the UI. Instead of re-rendering the entire component tree, React only updates the parts of the virtual DOM that have changed. It then efficiently applies these changes to the actual DOM.

3. Keys: React uses keys to uniquely identify elements in a collection (e.g., a list of items). Keys help React determine which elements have been added, removed, or re-ordered, allowing it to update the DOM efficiently without re-rendering the entire list.

Overall, reconciliation is a critical part of React's rendering process that enables efficient updates to the UI in response to changes in state or props.

}

**- What is `React Fiber`?**

React Fiber is a concept of ReactJS that is used to render a system faster, smoother and smarter.

The Fiber reconciler, which became the default reconciler for React 16 and above, is a complete rewrite of React’s reconciliation algorithm to solve some long-standing issues in React.

**- Why do we need `keys` in React?**

A `key` is a special attribute you need to include when creating lists of elements in React. Keys are used in React to identify which items in the list are changed, updated, or deleted. In other words, we can say that keys are unique Identifier used to give an identity to the elements in the lists.

Keys should be given to the elements within the array to give the elements a stable identity.

#### Example

```

<li key={0}>1</li>

<li key={1}>2</li>

<li key={2}>3</li>

**- Can we use `index as keys` in React?**

Yes, you can use index as keys in React, but it's generally not recommended, especially when rendering lists of dynamically generated components.

Using index as keys can lead to issues in certain scenarios:

Unpredictable Behavior: If the order of items in the list can change (e.g., items are added, removed, or reordered), using index as keys may result in unpredictable behavior. React relies on keys to determine which items have changed, and using index alone does not provide a stable identity for each item.

Poor Performance: Using index as keys can negatively impact performance, especially when items are added or removed from the list. React needs to re-render components when their keys change, and using index as keys may cause unnecessary re-renders of components that haven't actually changed.

Accessibility: Keys are often used to maintain component state and accessibility attributes. Using index as keys may cause accessibility issues, as screen readers and other assistive technologies rely on stable and meaningful keys to identify and navigate elements.

**- What is `props in React`? Ways to.**

props stands for properties. Props are arguments passed into React components. props are used in React to pass data from one component to another (from a parent component to a child component(s)). They are useful when you want the flow of data in your app to be dynamic.

#### Example

```

function App() {

return (

<div className="App">

<Tool name="Chetan Nada" tool="Figma"/> // name and tool are props

</div>

)

}

```

**- What is `Config Driven UI`?**

Config-driven UI (User Interface) refers to the approach of designing and building user interfaces by utilizing configuration files or data structures rather than hardcoding UI elements and behaviors directly into the application code.

Controlling the UI, how UI looks like, using data or config and the config comes from backend. For example, you want an offer carousal in your website, and offers vary with locations, so config driven UI will be helpful in such cases. We will have different carousals for different locations. For example, we want a background color of some div to be green in Mumbai, yellow in Delhi, or something like that, then we can use config driven UI.

**- Difference between `Virtual DOM` and `Real DOM`?**

The Virtual DOM and Real DOM are two concepts used in web development to manage and update the user interface (UI) of web applications. Here's the difference between them:

1. **Virtual DOM:**

* The Virtual DOM is an abstraction of the HTML DOM (Document Object Model) created by React and other libraries/frameworks.
* It is a lightweight, in-memory representation of the actual DOM elements and their properties.
* When changes are made to the UI in React, the Virtual DOM is updated instead of directly modifying the Real DOM.
* After updating the Virtual DOM, React performs a process called "reconciliation" to compute the minimal set of changes needed to update the Real DOM efficiently.

The Virtual DOM allows React to optimize rendering performance by batching updates, avoiding unnecessary DOM operations, and minimizing browser reflows.

1. **Real DOM:**

The Real DOM is the actual HTML DOM tree generated by the browser based on the HTML markup of the web page.

* It represents the structure of the web page and contains all the elements, attributes, and text nodes that make up the UI.
* Any changes made to the Real DOM directly affect the visible content and layout of the web page.
* Manipulating the Real DOM can be costly in terms of performance, especially when dealing with large or complex UIs, as it requires the browser to repaint and reflow the entire page.
* The Real DOM updates are typically slower compared to the Virtual DOM updates, especially in cases where multiple changes need to be applied to the UI.

In summary, the Virtual DOM is an in-memory representation of the HTML DOM used by libraries like React to optimize UI updates and improve rendering performance, while the Real DOM is the actual browser-rendered DOM tree that represents the current state of the web page. The Virtual DOM allows for efficient batched updates and minimizes the performance overhead associated with directly manipulating the Real DOM.

**Industry Standards/ Good practices:**

1. Always keep component files separately in src folder
2. Create a separate file for different components, also give the name of the scripts same as that of component name
3. Don’t keep hardcoded data in the components file (example img urls), create a separate folder (“utils” or “config” or “common”)
4. Also, name all the constants in CAPITAL LETTERS

Two types of export & import:

1. Default export
2. Named export

“Export default” can only be written once in a file. For exporting multiple things from a same file, we use named export.

export const LOGO\_URL = "https://e7.pngegg.com/pngimages/637/877/png-clipart-rasht-online-food-ordering-restaurant-application-software-kitchen-food.png" ;

export const IMG\_LINK\_1 = "https://images.pexels.com/photos/";

export const IMG\_LINK\_2= "/pexels-photo-" ;

export const IMG\_LINK\_3 = ".jpeg?auto=compress&cs=tinysrgb&w=600";

Named export is imported as below:

import {IMG\_LINK\_1} from "../utils/constants";

import {IMG\_LINK\_2} from "../utils/constants";

import {IMG\_LINK\_3} from "../utils/constants";

This is also okay:

const LOGO\_URL = "https://e7.pngegg.com/pngimages/637/877/png-clipart-rasht-online-food-ordering-restaurant-application-software-kitchen-food.png" ;

export const IMG\_LINK\_1 = "https://images.pexels.com/photos/";

export const IMG\_LINK\_2= "/pexels-photo-" ;

export const IMG\_LINK\_3 = ".jpeg?auto=compress&cs=tinysrgb&w=600";

export default LOGO\_URL;

default exporting one component and named exporting the rest.

When we try to filter out the data, for example if we want restaurants having avgRating > 4 to be rendered on the webpage, then we use filter function, but filter function filters out the data, but UI doesn’t change accordingly. So we use hook functions. Hook functions are Normal JS utility functions developed by Facebook developers.

Two most important hooks:

1. useState() – to generate super powerful state variables in react 🡪 imported as named import : import {useState} from “react”;
2. useEffect()

state variable cant be modified directly by using = sign. We need to use a function to do that.

   const [restaurantList, setrestaurantList] = useState(resList);

setrestaurantList , it is the function which helps us in updating and rendering the restautantList

const [restaurantList, setrestaurantList] = useState(resList);

//Can also be written as below. This is called as array destructuring

const arr = useState(resList);

const [restaurantList, setrestaurantList] = arr;

//Can also be written as below. This is called as array destructuring

const arr = useState(resList);

const restaurantList = arr[0];

const setrestaurantList = arr[1];

State variable binds the UI with data, that is, if data is changed, the UI will also get updated.

**Whenever a state variable updates, react re-renders the component.**

**Why React is fast?**

1. React has virtual DOM
2. React uses Diffing algorithm which is very efficient
3. It can do efficient DOM manipulation