**Introduction :**

**Open-source JavaScript library for building user interfaces.**

Components based architecture.

JavaScript: ‘this’ keyword, filter, map and reduce.

ES6-let & const, arrow functions, template literals, default parameters, object literals, rest and spread operators and destructuring assignments.

ES6-ECMA Script: Modern JS

**Difference between var, let and const:**

Scope update & redeclared access if initialized later

Var functional yes yes(undefined)

Let block only updated no(error)

Const block no no(error)

**Arrow Function(Fat Arrow):**

It is new feature of the ES6 standard which allows us to write function in simple way.

Syntax: let function\_name =(arg1,arg2,..,argN)=>{statement(s)}

Eg. let x=(a,b)=> a\*b;

**Template Literals:**

Use backticks(` `) instead of “ ” to write statements in console log.

To access variable use ${var\_name}.

Example:

let f\_name=”Vrishabh”;

let l\_name=”Vyawahare”;

console.log(“My name is ”+f\_name+”.My last name is ”+l\_name+”.”); //ES5

console.log(`My name is ${f\_name}.My last name is ${l\_name}.`); //ES6

**Template Literals(strings) Method:**

Method name should be in camelCase and value is case sensitive.

console.log(`${f\_name}`.startsWith(V)); //true

console.log(`${f\_name}`.endsWith(h)); //true

console.log(`${f\_name}`.includes(Vrish)); //true

console.log(`${f\_name} `.repeat (2)); // Vrishabh Vrishabh

To simplify we can declare var as,

var first\_name=`${f\_name}` ;

console.log(first\_name.endsWith(h));

**Destructuring assignments:**

JavaScript expression that unpacks values from arrays, or properties from objects into distinct variables.

const lang = [ ‘React’, ’HTML’, ’CSS’, ’JS’ ] ;

let [fav1, fav2, fav3, fav4] = lang //destructuring array

let [fav1, , , fav4]=lang

const Data={

name : ’Vrishabh’,

age : 18

}

let { name : myname, age}= Data; //destructuring obj

**Default Parameter:**

We can initialize function parameter with default value.

When an argument is omitted or undefined the default is used.

Example:

Function mul (a,b=1){

return a\*b;

}

mul( 5,2 ); //10

mul( 5 ); //5 as default value for b is 1.

**Rest Parameters:**

Allows us to represent an indefinite number of an arguments as an array.

We can use three dots (… ) as prefix before function’s last parameters as rest operator.

Example:

function data( a,b,…num){

console.log(num); //[ 1,2,3,4,5,6 ]

console.log(…num); // 3,4,5,6

}

data( 1,2,3,4,5,6 );

**Spread Operator:**

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

Example:

const set1 = [1, 2, 3];

const set2 = [4, 5, 6];

const set3 = [...set1, ...set2]; //[1,2,3,4,5,6]

const Vehicle = { brand: 'Ford', model: 'Mustang', color: 'red'}

const NewVehicle = { type: 'car', year: 2021, color: 'yellow'}

const ComboVehicle = {...Vehicle, ...NewVehicle}

output: { brand: 'Ford', model: 'Mustang', color: 'yellow', type: 'car', year: 2021}

Note: unmatched properties were combined, but matched properties were overwritten by the last object that was passed.

**What is Axios?**

It is a HTTP client library which is used to make requests to an API, return data from the API, and then do things with that data in our React application.

***npm install axios*** => command to install axios.

Download **Node JS** version and **VS code**.

Open terminal and run command : **npx create-react-app project-name**

**Code .** //run project in VS code

It provides you to create single page React applications.

**NPM:**

Node Package manager is warehouse of Code packages. Once we installed Node.js then automatically NPM installed. Npm is the node package manager that contains many JavaScript libraries, including React.

**Node version : node -v NPM Version : npm -v**

**Folder Structure:**

**package.json:** metadata of react app. version details.

**node\_modules:** contain all dependencies.

**DOM**: Document Object Model is a platform that allows scripts to dynamically access and update the content, structure and style of a document.

**ReactDOM**: It is part/package of React library used to render components and elements on the web.

**render( ) method**: It render React components into an HTML DOM tree.

Element Container

ReactDOM.render(<h1> HELLO WORLD </h1>,document.getElementById(“root”));

What to show? Where to show?

**JSX**: JavaScript Extension or XML.

Import React from ‘react’; //To use JSX

Multiple JSX element must be wrapped in enclosing tag such as <div>…</div> or [ <h1>,<p> ].

In JSX, we can write expression inside { }. Expression can be variable, property or JS exp.

**React.Fragment**: allows you to wrap multiple JSX elements without adding an extra node to DOM.

We can use <> </> bracket to wrap JSX elements.

**Date & Time method :**

Const date=new Date( ).toLocaletDateString( );

Const time=new Date( ).toLocalettimeString( );

**Component:** Code inside .js file which describe the part of UI.

It is reusable and can be nested into another file.

Building blocks of react app. It divides react app into smaller pieces of code.

Two types:

1) Functional components

2) Class components

1. Functional components:

JavaScript function which returns JSX code.

Stateless components.

Example:

function user( ){

return(

<h1> HELLO VRISHABH </h1>

)

}

export default user;

1. Class components:

To track the state of components and use lifecycle methods. To use properties available in React in your class component you must inherit the Component class.

Example:

import React,{Component} from 'react';

class ClassEx extends Component

{

render(){

return(

<h2>React JavaScript</h2>

)

}

}

export default ClassEx;

**CSS Style:**

To use CSS style in react, we have to import .css file in .js file.

import “./filename.css”;

\*{ margin : 0; padding : 0; box-sizing : border-box; }

text-transform : capitalize|uppercase|lowercase; => control capitalization of text.

box-sizing : border-box; allows to include the padding and border in an element's total width and height.

margin :auto; => to horizontally center the element within its container.

display: flex; =>child element automatically align like row with auto width and auto height.

justify-content: center; => adjust items to center

Google font: <https://fonts.google.com/>

Embedded link code inside index.html head tag.

<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Sofia">

Internal CSS:  
Use style attribute as Obj. use css property as key value pair.

Style={{ color: ‘black’, textAlign:’center’}}

**Styled components:**

Styled-components is a library that allows you to write CSS in JS while building custom components in React JS. Styled-components generates unique class names for your styles.

***npm install styled-components;***

***import styled from 'styled-components';***

for <h1> *Styled Components* </h1> tag styling:

we will create our custom component called ***Heading*** and use it instead of the <h1> tag with custom styling and wrap the style in backticks.

*const Heading = styled.h1`*

*color: red;*

*font-size: 4rem;*

*<* Heading *>Styled Components</* Heading *>*

*`*

**Nested styled :**

If ***className*** is used for tag then use ***.*** ***className*** to access.

const Mycolor=’red’ ***//to access global CSS prop use ${ Mycolor }***

const Wrapper = styled.div`

h1{ color:${ Mycolor }; }

button{

padding: 4px 10px

***&:*** hover{

Color : red;

}

**span**{ ..}

} `

<Wrapper> <h1> ……..</h1> <button > <span>…..</span></button> <Wrapper>

**Props in styled components:**

Use attribute in custom element like<Button bg=”red”>click Me <Button>

background-color: $ { (props) => props.bg }

To access prop use callback function.

**Import/Export:**

While exporting there should be max one default.

While exporting many var, function or components use {name1,name2,name3 };

While importing default we can use different name.

Importing more than one var, components, functions use { }. We can also use \* As any name.

**Props:**

Props stand for properties.

Props are like function arguments in js and attributes in HTML.

**Array in JavaScript:**

Store multiple value in one variable.

Index start with 0.

Example:

Var name=[ ‘Ro’, ’King’, ’Mahi’ ]; OR

Var name2=new array( ‘Ro’, ’King’, ’Mahi’ );

**Map(** **)** **Method**:

Map( ) creates a new array from calling a function for each array element.

It doesn’t change the original array.

Syntax:

array.map(function(currentValue, index, arr), thisValue)

Example:

const numbers = [65, 44, 12, 4];

const newArr = numbers.map((val) => { return val \* val })

**conditional rendering:**

Use Ternary Operator.

{ Condition ? exp (true) : exp(false) }

**Emoji:**

Install **:emojisense** extension.

Setting => Extension => **:emojisense** =>Languages => Edit Setting.json file(11th line).

Use : create Emoji.

**React Auto Suggestion:**

"emmet.includeLanguages": {"javascript": "javascriptreact"},

Add above part in Setting => open Setting JSON(right top side)

**React Event:**

React can perform actions based on user events.

React has the same events as HTML: click, change, mouseover etc.

React events are written in camelCase and inside curly braces.

Example: onClick={shoot}

**Form <form>:**

HTML form has default behavior of browsing to a new page when the user submits the form.

In HTML, form data is usually handled by the DOM.

In React, form data is usually handled by the components.

In React, value attribute is treated as props and props are read-only therefore we have to use defaultValue or onChange attribute. defaultValue attribute unable to manage the value.

onChange attribute has event object. event.target return the input field.

Name attribute used to recognize input field which has changes.

When the data is handled by the components, all the data is stored in the component state.

We can use the useState Hook to keep track of each inputs value.

You can control changes by adding event handlers in the onChange attribute of input tag.

onChange={ (e) => setName(e.target.value) }

Submit action is controlled by adding an event handler in the onSubmit attribute for the <form>.

Form tag has default behavior which automatically refresh the page. HTML form has default behavior of browsing to a new page when the user submits the form.

To prevent that we use **obj.preventDefault( )** method.

UseRef hook is used for uncontrolled components.

**State Lifting up:**

**Pass data from child component to parent component.**

**Child ---> parent**

Create function in parent component. call function in child component with data as parameter.

**Material ui/icons:**

npm install @material-ui/icons => run command on cmd OR **npm install react-icons**

import { IconName} from "react-icons/bi";

<IconName />

**React-Toastify:**

allows you to add notifications to your app.

Type: success, warn, error, info.

theme: "colored"

position : "top - center"

autoClose:2000

***npm install --save react-toastify*** => install toastify command on terminal

import { ToastContainer, toast } from 'react-toastify';

import 'react-toastify/dist/ReactToastify.css';

function App(){

const notify = () => { toast.success("Wow so easy!",{position : "top-center"}) }

return (

<div>

<button onClick={notify}>Notify!</button>

<ToastContainer /> or <ToastContainer position=”top-center”/>

</div>

); }

**React Hooks:**

Hooks are functions that allow to use state and other React features such as lifecycle methods. Because of this, class components are generally no longer needed.

**Rules of Hook:**

Write inside functional component.

It should use at top level of functional component.

Component name must be PascalCase.

Import hook or directly write React.Hook

Don’t call Hook inside loops, condition.

**useState Hook:**

The React useState Hook allows us to manage state in a function component.

***Import {useState} from ‘react’;*** => import useState Hook

Const [curValue, updatedValue]=useState(initialValue);

updatedValue is a function that updates the state.

initialValue stored in curValue.

setInterval(Update,1000); => used as callback function. first para is function to be call and second is time period after which function should call.

To use Array with useState hook, defined initial value as array name and use map() to access it’s val.

To update Object while using useState hook, we use spread operator (…).

***When we use onClick, we can’t directly pass a function with an argument. Therefore, we use callback function.***

***Example: onClick ={ ( ) =>function\_name ( parameter) }***

**Logical && and || operator:**

Const [ demo, setDemo] =useState( “ ” );

{ demo || “ Vrishabh”} => if left true then return Left part else Right part.

{ demo && “ Vrishabh”} => if left true then return Right part else Left part.

***id:new Date( ).getTime ().toString( )*** object key-value pair is used to create unique id for map( ).

**Context** **API:**

Provides a way to pass data through the component tree without passing prop to every level.

**Create a Context:**

**Import {createContext} from ‘react’;**

**const MyContext = createContext( );**

**createContext( ):** create an object that hold the data you want to share across your application.

**Provider** component accepts a "value" prop that holds the shared data, and any component that is a child of the Provider component can access that shared data.

Child component must be wrapped in <context.Provider>.

<MyContext.Provider value={ “ Vrishabh ” }>

<ChildComponent />

</MyContext.Provider>

**Consumer** component can be used to access the context data.

Consumer should import Context object.

Consumer return function.so between consumer tag there should be callback function.

<firstname.Consumer>

{ ( fname ) => { return ( <h1>My name is {fname} </h1> ) } }

</firstname.Consumer>

**useContext** hook enables you to access the context data without using Consumer component explicitly.

**Import {useContext} from 'react';**

**const fname=useContext(Context);**

**UseEffect Hook :**

The effect hook lets you perform side effect in functional components.

side effects are fetching data, directly updating the DOM, and timers.

It always accepts a function.

useEffect(<function>, <dependency>)

useEffect(( ) => {

// Your effect

return ( ) => { // Cleanup };

}, [ ]);

Effects run after every render cycle.

[ ] represent dependency. If it is empty array then it calls only one time after page refresh.

useEffect automatically call when page load first time.

We can put condition inside to run effect conditionally.

***window.screen.width*** specify width of window screen.

***window.addEventListner(“resize”, function)*** this event catches the resize and call the function.

***window.innerwidth*** catch the current width.

***Cleanup function:*** The cleanup can prevent memory leaks and remove unwanted things. Some use-cases for this are:

Clean up subscriptions, modals, Remove event listeners, Clear timeouts.

**useRef Hook :**

It can be used to store a mutable value that does not trigger a re-render when updated.

It can be used to access a DOM element directly.

**const reference = useRef(initialValue)**

It returns reference ***object having special ‘current’ property***.

Assign the reference to ‘ ***ref*** ’ attribute of the element .

example: return <input ***ref={reference }*** />;

The reference must be updated either inside a useEffect() callback or inside handlers (event handlers, timer handlers, etc).

You can’t get a ref to a custom component.

You can read or write refs from event handlers or effects instead.

Do not write or read **ref.current** during rendering.

Changing a ref does not trigger a re-render, so refs are not appropriate for storing information you want to display on the screen. Use **state** for that instead.

**forwardRef:**

By using forwardRef, you can pass a reference from a parent component to a child component, even if that child component is wrapped inside another component. This enables the parent component to interact with the child’s DOM element or instance directly.

prop only works on HTML elements, not on React components.

When we want to pass down a reference to a React component, we need to tell React which HTML element it should reference, as there can be more than one in our component. That’s where forwardRef becomes useful. It allows us to specify which exact HTML element we want to reference.

import React, { forwardRef }from 'react'; // import forwardRef.

const Child=( props ,ref ) { //define Child component with arrow function.

return (

<> < input type ='text' ref={ ref } /> </>

);}

export default forwardRef (Child); //wrap child component in forwardRef

or use input element as callback function of React.forwardRef

**useMemo hook:**

The React useMemo Hook returns a memoized value.

Memoized value does not need to be recalculated. It stores each computation and return the same value when required again.

It is very useful in optimizing the performance of a React component by eliminating repeating heavy computations and avoid the unnecessary re-rendering of components.

***const Result = useMemo(function, dependencies);***

const factorial = useMemo(() => factorialOf(number), [number]);

useMemo hook takes two arguments: the calculation function and an array of dependencies.

The useMemo Hook only runs when one of its dependency’s updates.

If the dependencies don't change during the next renderings, then useMemo() doesn't invoke function, but returns the memoized value.

**useCallback hook:**

useCallback hook will return a memoized instance of the callback that only changes if one of the dependencies has changed.

useCallback help to prevent unnecessary re-renders as it returns the memoized function that can be passed as props to the children’s components.

When you define a function inside a component, it is recreated on every render, even if the component’s state or props have not changed.

The React.memo serves as a wrapper for a component and returns a memoized output of that component, which prevents the component or sub-components from unnecessary re-rendering.

const handler = useCallback(() => {

console.log("You clicked ");

}, [dep] );

**useReducer Hook :**

It acts as an alternate hook to the useState hook to manage complex state in your application**.**

***const [state, dispatch] = useReducer(Reducer, initialState);***

**Reducers** are pure functions that take in a state and action and return a new state.

The pure function always returns the same output if the same arguments are passed in and doesn’t produce any side-effects.

***function reducer(state, action) {***

***if (action.type===”reset”) {***

***return { count };***

***}***

***}***

The action to be executed is specified in reducer function. The reducer function will return the updated state.

**The dispatch function** accepts an object that represents the type of action we want to execute when it is called. It sends the type of action to the reducer function which updating the state.

Dispatch function call the reducer function. It trigger the type of action.

<button onClick={ ( ) => dispatch( { type: 'reset' } ) } > Reset </button>

**type** stands as the identifier of the dispatched action.

**React Router:**

***npm install react-router-dom*** =>install cmd

***import { BrowserRouter, Route, Routes} from 'react-router-dom';***

Everything that gets rendered will need to go inside the **<BrowserRouter>** element.

// index.js

ReactDOM.render(

<BrowserRouter>

<App />

</BrowserRouter>,

document.getElementById('root')

)

**<Routes >** element ensure that only one component is rendered at a time.

"Router may have only one child element" warning? If not used. It renders route exclusively.

**<Route>** are the links between the components and should be placed inside the < Routes > tags.

To tell the <Route> tags which component to load, simply add a **path attribute** and the name of the component you want to load with **component** **attribute**.

<Routes >

<Route path="/" element={Home} exact />

<Route path="/about" element={About} />

</Routes >

***Path= ” / ” represents a default page***.

We add **exact** to the Route tag else it loads the first one to match the route.

<Route path=’/\*’ component={Error} /> => for incorrect URL.

<Route path=’/\*’ component={<Navigate to=’/’>} /> => for incorrect url move to default page.

Use **<Outlet** **/>** tag for nested Routing in parent element.

**useLocation Hook:**

The useLocation hook returns the location object that contains information about the current URL such as path names, key, hash and other details related to the current route.

*import { useLocation } from "react-router-dom"*

*const location = useLocation ( )*

*{ location.pathname }*

**useHistory Hook:**

The useHistory hook allows us to track the history of changes.

*npm install --save react-router-dom*

*import { useHistory } from "react-router-dom"*

*const history = useHistory( )* //Declare a variable

*history.push("/dashboard")* // this object to redirect the user to another page.

*history.goback( )* // go to a previous page.

**useNavigate:**

updated version of useHistory with less code.

let navigate = useNavigate( )

navigate('/aboutpage');

navigate(-1) //GoBack

**useParams Hook:**

Implementing dynamic routing with useParams is simple and straightforward. Dynamic routing allows rendering your React components conditionally.

useParams is a hook that allows you to have access to dynamic parameters in the URL

The useParams Hook returns an object of key/value pairs from the current URL's dynamic parameters.

If the params are matched to the <Route path>, the Hook will be able to access these values.

<Route path="/products/:id" element={<Product/>} />

Import { useParams } from ‘react-router-dom’;

const { id } = useParams( );

**Lifting State up:**

Share data from child to parents.

Create function in parent and pass as prop in child components then call function with parameter as data in child components.

**Redux:**

npm install --save redux react-redux

Redux is a state management library that you can use with any JS library or framework like React, Angular, or Vue.

In a large application, Redux store the state at a central location and share it among the different components.

Store: hold state of application.

Action: Describe changes in state of application.

Reducer : carried out the state transition depending on actions.

It allows React components to read data from a Redux Store, and dispatch Actions to the Store to update data.

**Actions** are simple JavaScript objects that have a type property, describing the type of action and the entire information being sent to the store.

Actions are created using action creators which are the normal functions that return actions.

const IncAction = (num) => {

return {

type: "INCREMENT",

payload: num

}

a **Type** property to indicate the type of action to be carried out.

a **payload** object that contains the information that should be used to change the state.

In Redux, **reducers** are functions (pure) that take the current state of the application and an action and then return a new state.

const initialState = 0;

function reducer(state=initialState, action) {

switch (action.type) {

case 'INCREMENT':

return state + action.payload;

case 'DECREMENT':

return state - action.payload;

default: return state;

}

}

If there are more than one reducer function then use **combineReducer( )** to combine function.

***export const rootReducer = combineReducer({key1: reducer,});***

A **store** is a JavaScript object which can hold the application’s state and provide a few helper methods to access the state, dispatch actions. There is only one store.

getState( ) = give access to state it holds.

Dispatch()= allow to update the state.

Subscribe()=register listeners.

To connect react-redux with application we must import {provider} and wrap all component inside it. Then pass store as props to connect store with react -redux.

**useSelector:** The useSelector hook allows access to the state stored in a Redux store.

import { useSelector } from 'react-redux';

It is a replacement for the traditional *mapStateToProps* function used in class-based components.

const counter = useSelector(state => state.counter); //It return selector function

**useDispatch:**

It returns a reference to the dispatch function, which can be used to dispatch actions to the store.

const dispatch=useDispatch();

**combineReducers**:

combine multiple reducers together, Redux provides the combineReducers() method.

import { combineReducers } from 'redux';

const allReducers = combineReducers ( { reducer1,reducer2 } );

**Higher-order components (HOCs):**

HOCs are a powerful feature of the React library. They allow you to reuse component logic across multiple components.

In React, a higher-order component is a function that takes a component as an argument and returns a new component that wraps the original component.

const HOCComp=(Wrapper)=> {

function InnerHOC(props){

//LOGIC to be written here

return(

<div><Wrapper Increment={Increment} count={count}{…props}/></div>

)

}

return InnerHOC;

}

export default HOCComp;

Wrap component within HOCComp where we import it.

{…props} used to pass the original props as original components wrap inside new enhanced HOC components.

**Data Fetch :**

API stands for “Application Programming Interface,”

whenever the client requests the data, the API makes a GET request to the server and sends that back to the application for display.

Representational State Transfer API which consists of HTTP methods to fetch data from the server and display it in the application.

GET: fetch the data from a server.

POST: post the data to a server.

DELETE: delete the data from a server.

PUT: update or modify the data from a server.

Import the useState( ) hook and set to hold data.

We will create a callback function that will store the user’s data and then use the useEffect() hook to make the function run every time the page loads.

The fetch() method in JavaScript is used to request data from a server. The request can be of any type of API that returns the data in JSON or XML. The fetch() method requires one parameter, the URL to request, and returns a promise.

***const [data, setData] = useState([ ])***

***const FetchInfo = ( ) => {***

***return fetch(url)***

***.then((res) => res.json( ))***

***.then((dt) => setData(dt))***

***}***

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

**Async/Await:**

Asynchronous: non-blocking way of working. single thread allocated to handle multiple requests.

Synchronous means blocking the way of working.

A promise is an assurance that we will do something in the future. It will be executed when the execution time comes, or it will be rejected. Promises allow you to perform asynchronous operations.

A promise is used to handle the asynchronous output of an executed operation.

To use promises we have used .**then( )** which is handling the result of a promise and called when a promise is resolved.

The “await” keyword is used to ensure that all promises are synchronized which means they wait for each other to get executed. With await we can eliminate the use of callbacks. Async returns a promise and await prepend while calling a promise.

***const fetchInfo =***

***async ( ) => {***

***const response = await axios.get(url);***

***setData(response.data);***

***}***

**DateTimeRangePicker:**

***npm install @wojtekmaj/react-datetimerange-picker;***

*import DateTimeRangePicker from '@wojtekmaj/react-datetimerange-picker';*

*import '@wojtekmaj/react-datetimerange-picker/dist/DateTimeRangePicker.css';*

*import DateTimePicker from 'react-datetime-picker'; // for DateTime.*

*import 'react-datetime-picker/dist/DateTimePicker.css';*

*import 'react-calendar/dist/Calendar.css';*

*import 'react-clock/dist/Clock.css';*

rangeDivider rangeDivider={'To'} Divider between datetime inputs.

autofocus true Automatically focuses the input on mount.

disableClock true When set to true, will remove the clock.

format yyyy-mm-dd

maxDate & minDate maxDate={new Date(2023,8,1)}

monthPlaceholder monthPlaceholder={" "}

**react DatePicker:**

***import DatePicker from "react-datepicker";***

***import "react-datepicker/dist/react-datepicker.css";***

***const [value, onChange] = useState(new Date());***

***selected={value} // required***

***onChange={onChange} // required***

***excludeDates***={ new Date( ) } : exclude current date

***showIcon*** true show calendar Icon.

***autofocus*** true focus on input box.

***dateFormat yyyy/MM/dd hh:mm aa***

***isClearable*** trueclear icon to clear input.

***placeholderText={"Enter a Date"}***

***shouldCloseOnSelect*** true If true then on Select date Calendar close.

***showTimeSelect :*** h:mm aa If true it shows time to select.

***excludeDates***={ new Date( ) } exclude the date from select

***Input time*** :

timeInputLabel={ ‘ Time: ’ }

showTimeInput ={ ‘ true ‘ }

***Today Button*** which onclick select today Date:

todayButton={ ‘ Today ‘ }

***Month & Year Dropdown:***

showMonthDropdown

showYearDropdown

dropdownMode="select"

**filterDat**e={ ( date ) => {

return date.getDay( ) !== 0 && date.getDay( ) !== 6;

}}

Disable Saturday and Sunday.

Set ***Date Range Picker***:

const [startDate, endDate] = value; // value is useState declare const.

startDate={startDate}

endDate={endDate}

**Environment variables:**

Allows you to store sensitive data, such as API keys, database credentials, and other confidential information, outside your codebase.

By storing this sensitive information in environment variables, you prevent it from being exposed in your code or version control system. This is particularly important when your code is publicly accessible, as in open-source projects.

In your project's root directory, create a file called .env. This file will hold all your data.

Make sure to add .env to your .gitignore file to prevent it from being tracked. environment variables.

Adding Variables to the .env File: REACT\_APP\_API\_URL=https://akamobi.com

Accessing Environment Variables: const apiurl = process.env.REACT\_APP\_API\_URL;

**Bootstrap:**

***npm install react-bootstrap bootstrap***

***import 'bootstrap/dist/css/bootstrap.min.css';*** import in app.js or index.js file.

Bootstrap is the most popular HTML, CSS and JavaScript framework for developing a responsive and mobile friendly website***.***

import Button from 'react-bootstrap/Button'; OR

import { Button } from 'react-bootstrap';

**Alert:**

variant='primary' | 'secondary' | 'success' | 'danger' | 'warning' | 'info' | 'dark' | 'light'

dismissible : add close symbol.

<Alert.Heading>: to add heading.

**Badges:**

Used to show labeling component and small count.

<Badge bg="secondary" pill text=’ danger’>New</Badge>

bg=The visual style of the badge

pill = modifier class to make badges more rounded.

Text=Sets badge text color

**Button group:** Group a series of buttons together on a single line

<ButtonGroup size="lg" vertical>

<Button>Left</Button>

<Button>Right</Button>

</ButtonGroup>

<ButtonToolbar>:Combine sets of <ButtonGroup>

size : 'sm' | 'lg' Sets the size for all Buttons in the group.

vertical :Make the set of Buttons appear vertically stacked.

**<Button** variant=’primary’ size=’md’>

Variant: type of button

Variant=’outline-primary’ variants with no background color.

active Manually set the visual state of the button to :active.

disabled Disables the Button, preventing mouse events.

href Providing a href will render an <a> element, styled as a button.

<**Table** striped bordered hover>

bordered: Adds borders on all sides of the table and cells.

borderless: Removes all borders on the table and cells, including table header.

variant: dark light text on dark backgrounds.

hover: Enable a hover state.

size="sm" to make tables compact by cutting cell padding in half.

**<ListGroup>**

<ListGroup.Item>Item1</ListGroup.Item>

<ListGroup.Item>Item2 </ListGroup.Item>

</ListGroup>

variant="flush" removes outer borders and rounded corners to render list.

numbered : numbered list group items.

**Dropdowns**: displaying lists of links.

<DropdownButton>

<Dropdown.Item href="#/action-1">Action</Dropdown.Item>

</DropdownButton >

Title=”Title name”

size="lg"|”sm”

<Dropdown.Divider /> Separate groups of related menu items with a divider.

**Cards:**

<Card style={{ width: '18rem' }}>

<Card.Img src="holder.js/100px180" />

<Card.Body>

<Card.Title>Card Title</Card.Title>

<Card.Text>

Some quick example text to build on the card title.

</Card.Text>

<Card.Link href="#">Card Link</Card.Link>

<Button variant="primary">Go somewhere</Button>

</Card.Body>

</Card style>

<Card.Subtitle>: subtitle of cards.

<Card.Link href="#">Card Link</Card.Link> => create link

<Card.Header> = create header like list item.

<Card.Footer > = Create footer

variant="top"|”bottom” used with img tag to set image at the top or bottom.

border/bg="secondary" add background and border

Navbar:

<Navbar.Brand>: branding component must write inside <Container>.

<Nav>

<Nav.Link href="#home"> Home </Nav.Link> // create link

</Nav>

className=”justify-content-start/end/center” => set element at left, right, center.

className="flex-column" =>vertical nav

variant="tabs" |”underline”| ”pill” => The visual variant of the nav items.

If you want each Nav Item to be the same size use **justify**.

data-bs-theme="dark" => dark theme.

**Form:**

The <FormControl> component directly renders the <input> or other specified component.

<Form>

<Form.Control type="search placeholder="Search" /> //search bar

</Form>

disabled= disabled input

size: input box size

type="file" : input file

type=”color” : color input

<Form.Text> : text

**<Form.Select** size="lg">

<option>Large select</option>

</Form.Select>

**<Form.Check** type=”checkbox|radio|switch”/> **:** switch

**<FloatingLabel** label=”..”/> : placeholder is required on each <Form.Control>

**as="textarea"** set text area box

**Create fake API:**

Install JSON Server

**npm install -g json-server**

**json-server --watch db.json**

**.forEach()**

**This iterates over every element in an array with the same code, but does not change or mutate the array, and it returns undefined.**

**.map()**

**This method transforms an array by applying a function to all of its elements, and then building a new array from the returned values.**

**.reduce()**

**This method executes a provided function for each value of the array (from left to right).**

**.filter()**

**This checks every single element in an array to see whether it meets certain criteria as specified in the filter method, and then it returns a new array with the elements that match the criteria.**

Two distinct objects are never equal for either strict(===) or abstract(==) comparisons.

An expression comparing objects is only true if the operands reference the same object, functions are objects in JavaScript.