Syllabus
========
-ReactJs features (VirtualDOM , ReConciliation)
-Local Environemnt Setup (Create-react-app , vite)
-JSX
-Class Components
-Functional Components
-React Object
-Fragment
-Component Styling
-Conditional rendering
-Lists & keys
-Props : de-structuring,requiring props, proptypes,default props
-State
-Pure Component
-Memo Component
-HigherOrder Component
-Events : Synthetic Event
-Lifecycle Hooks
-Form
-Http - Axios
-Interceptors
-Routing
-REDUX (State Management)
-Unit Testing (JEST)
-EsLint

React Local Setup
=================
1. download Nodejs and install
   https://nodejs.org/en/download/

2. check if nodejs is installed? (open cmd and run the below command)
   node -v

3. check if NPM is installed? (NPM-Node Package Manager)
   npm -v

4. go to the folder where you want to create project(d:/sanjay/react)& run the below command
        npx create-react-app project1 (npx-node package executer)

5. go to the created project folder(project1) and start your React application
   cd project1
   npm start

6. A new browser window will pop up with your newly created React App!
   open a browser tab and type 'http://localhost:3000/' if browser doesn't open automatically.

   set PORT=3001 && npm start

React Project with Typescript
==========
-Create React App supports TypeScript out of the box.
-To create a new project with TypeScript support, run the below command
npx create-react-app my-app --template typescript
-To add TypeScript to an existing Create-React-App project, install the below things
npm install --save typescript @types/node @types/react @types/react-dom @types/jest

create-react-app
==========
-It is a React application boilerplate generator created by Facebook.
-This CLI tool installs React,ReactDOM & other libraries required for a react project.
-It provides a development environment configured for ease-of-use with minimal setup.
-It creates a frontend build pipeline. Under the hood, it uses Babel and webpack.

NPX
===
-NPX : node package executer.
-Its a Package runner/executer tool.
-It can execute any package that you want from the npm registry without even installing that package.
ex: npx create-react-app my-app

React app with Vite
==========
-Vite.js is a build tool and development server that is designed to optimize the development experience for modern web applications.
-It includes built-in support for TypeScript, CSS preprocessors.

1. Create a vite project
    npm create vite@latest
        (OR)
    npm create vite@latest my-app -- --template react
        (OR)
    npm create vite@latest my-app -- --template react-ts

2. Go to the crated Project, install , and serve the application
    cd my-app
    npm install
    npm run dev

What React is
==========
-React is a JavaScript library for building user interfaces.
-It is an open-source, component based library.
-It is created & maintained by Facebook.
-React is used to build single page applications.

-React allows us to create reusable UI components.
-ReactJS uses virtual DOM based mechanism to fill in data (views) in HTML DOM.

React is NOT a framework
==========
-React is a library and not a framework.
-The difference between a library and a framework is that a library only helps us in one aspect whereas a framework helps us in many aspects.
Let's take an example:
-React is a library because it only takes care of our UI.
-Angular, on the other hand, is a framework because it handles much more than the UI (It handles Dependency Injection, CSS encapsulation, httpClient, Form validation, routing etc.)

Framework	Library
-group of libraries to make your work easier	-performs specific, well-defined operations
-provides ready to use tools,standards templates for fast application development	-provides reusable functions for our code
-Collection of libraries & APIs	-collection of helper functions,objects
-cann't be easily replaceable	-can be easily replaceable by another
-angular,vue	-jQuery,ReactJs,lodash,moment
-Hospital with full of doctors	-A doctor who specializes in one

React	Angular
1. Library-2013	1. Framework-2009
2. Light-weight	2. Heavy
3. JSX + Javascript	3. HTML + Typescript
4. Uni-Directional	4. Two-way
5. Virtual DOM	5. Regular DOM
6. Axios	6. HttpClientModule
7. No	7. Dependency Injection
8. No	8. Form Validation
9. extra libraries needed	9. No additional libraries
10. UI heavy	10. Functionality Heavy

React Features
==========
-Light weight
-JSX
-Components (easy to build, easy to extend,reusable,loosly coupled)
-Oneway Data Binding (watchers will not be there for bindings)

-Virtual DOM
-Easy to learn because of its simple Design
-Performance

DOM (Document Object Model)
==========
-DOM is a tree-like structure representing the HTML of a web page
-Allows JavaScript to interact with and modify the page's content, structure, and style.

Why virtual DOM?
==========
-Frequent DOM manipulations are expensive and performance heavy.
-Every time the DOM changes, browser would need to recalculate the CSS,
run layout and repaint the web page.
-we need a way to minimize the time it takes to repaint the screen.
This is where the Virtual DOM comes in.
-React uses virtual DOM to enhance its performance.

What Virtual DOM is
==========
-A virtual DOM is a lightweight JavaScript object which is just a copy of the
real DOM.
-It is a node tree that lists the elements, their attributes and content as
Objects and their properties.
-Virtual DOM is a JavaScript object that mirrors the structure of the real DOM.
-A virtual DOM is the DOM where a representation of the UI is kept in memory and synced with the DOM.
-React never reads from real DOM, only writes to it.

Virtual DOM Benefits:
==========
Improved Performance: By reducing the number of direct DOM manipulations, the Virtual DOM significantly speeds up the update process.
Optimized Updates: React intelligently determines the most efficient way to update the real DOM, minimizing costly reflows and repaints.
Simplified Development: The Virtual DOM makes it easier to work with complex UIs by abstracting away the complexities of direct DOM manipulation.

How does React Work? (Virtual DOM)
==========
Initial Render: When page loads, React creates a virtual DOM tree based on the initial JSX or HTML code.
State Changes: When the application's state changes (user input, data updates), Generates a new Virtual DOM tree based on updated data.
Diffing: React compares the new Virtual DOM with the previous Virtual DOM, It identifies what exactly changed
Reconciliation: Based on diffing results, React calculates the minimal number of real DOM operations needed.
Real DOM Update: Only the necessary changes are applied to the real DOM, resulting faster updates.

What is the difference between Shadow DOM and Virtual DOM?
==========
-Shadow DOM is a browser technology for scoping variables and CSS in web components.
-Virtual DOM is a concept implemented by React on top of browser APIs.

ReactDOM
========
-ReactDOM is the glue between React and the DOM.
-React creates a virtual representation of your User Interface (what we call a Virtual DOM) and then ReactDOM is the library that efficiently updates the DOM based on the Virtual DOM.
-The reason why the Virtual DOM exists is to figure out which parts of the UI need to be updated and then batch these changes together.
-ReactDOM receives those instructions from React and then efficiently updates the DOM.

Web Browser Workflow
==========
Parsing HTML to construct DOM tree --> Render Tree construction --> Layout of the Render tree -->
-->painting the render tree.

React Reconciliation
==========
-Reconciliation is the process through which React updates the DOM.
(syncing the Virtual DOM to the actual DOM)
-Reconciliation is the mechanism that tracks changes in a component state and renders
the updated state to the screen.
-It's a step that happens between the render() function being called and the displaying of elements on the screen. This entire process is called reconciliation.

Stack Reconciler < React-16
==========
-Synchronous
-works like a stack
-cann't be interrupted

Fiber Reconciler
==========
-Fiber is the new reconciliation engine or re-implementation of core algorithm in React v16.
-React Fiber reconciler makes it possible to divide the work into multiple units of work(incremental rendering).
-It sets the priority of each work, and makes it possible to pause, reuse, and abort the unit of work.
-Fiber is Asynchronous.
-reconciliation and rendering to the DOM weren't separated, and React couldn't pause
its traversal to jump to other renders in between. This often resulted in lagging inputs.
-Fiber allows the reconciliation and rendering to the DOM to be split into two separate phases:
Phase 1: render(processing)

-React creates a list of all changes to be rendered in the UI
-Once the list is fully computed, React will schedule these changes to be executed in the next phase.
-React doesn't make any actual changes in this phase.
Phase 2: Commit

-React tells the DOM to render the changes that was created in the previous phase.
-the Reconciliation phase can be interrupted, the Commit phase cannot.

React Project - Folder Structure
==========
node_modules/ : Provides npm packages to the entire workspace. Workspace-wide node_modules dependencies are visible to all projects.
public/ : Only files inside the `public` folder can be referenced from the HTML
src/ : Source files for the root-level application project.
.gitignore : Specifies intentionally untracked files that Git should ignore.
package.json : Configures npm package dependencies that are available to all projects in the workspace.
package-lock.json : Provides version information for all packages installed into node_modules by the npm client.
README.md : Introductory documentation for the application.

React Project Flow
==========
1. index.html --> (container to inject component)
2. index.js --> root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />)
where what
3. App.js --> AppComponent Code

JSX
===
-JSX (JavaScript Syntax Extension)is a special syntax for React that makes it easier to represent the UI.
-JSX is used to describe the structure & content of a react component.
-JSX allows us to add the elements in DOM without using createElement() or appendChild() methods.
-JSX looks similar to HTML but it is not HTML.
-JSX code we write gets transformed into React.createElement().
-JSX is not part of browser. we need a tool(Babel)(a JavaScript compiler)
to transform it to valid JavaScript.
-JSX doesn't support void tags.
<img> invalid
<img></img> valid
<img /> valid
-Since JSX is closer to JavaScript than to HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.
-ex: class becomes className in JSX, and tabindex becomes tabIndex.

React Without JSX
==========
-Code written with JSX will be converted to use React.createElement().
-we don't have to use React,createElement() when we use JSX.
Syntax : React.createElement(type,[props],[...children])
example: React.createElement(div , {name:'myDiv'},[<p> ,])
-The first argument is the type of element we're creating
-The second argument is props object.
-The last argument is the children of that element.

ex: This is a div
-Babel converts the above JSX line to the below code

```
return React.createElement("div", {
   class: "test"
 }, "This is a div");
```

React Element
==========
-A React element is a JavaScript object with specific properties and methods that React assigns and uses internally.
-React elements are the instructions for how the browser DOM get created.
-When we use ReactDOM library React elements are getting changed into DOM elements.
-However, when we use React Native, React elements are getting changed into native UI elements of Android or iOS.
-We create React elements using a function called createElement().
-createElement() method is part of the Top-Level React API, and we use it to create React elements.
-This method takes three parameters:
    a. The first argument defines type of element to create. (h1/p/div)
    b. The second argument defines properties or attributes of the element.
    c. The third argument represents the element's children, any nodes or simple text that are inserted between the opening and closing tag.

```
const hello = React.createElement(
    "H1",
    {id: "msg", className: "title"},
    "Hello React Element"
);
```

-document.createElement() returns a DOM element (for example a div or an h1).
Whereas React.createElement() returns an object that represents the DOM element.

Module Systems
==========
1. CommonJS

```
module.exports = {member1,member2};
const member1 = require('Library/file name');
```

2. ECMASCRIPT

```
export member1;
export default member2;
import DefaultMember , {Namedmember} from 'file'
```

imports & exports
==========
-Default import:
 import DefaultMember from 'src/my_lib';
-Named imports:
 import { name1, name2 } from 'src/my_lib';
-Combining a default import with named imports
 import DefaultMember, { name1, name2 } from 'src/my_lib';

Named Export vs Default Export
==========
-Only one default export is allowed per file,where as multiple named exports are allowed per file.
-Named exports are useful to export several values.
-A default export can be a function, a class, an object(cann't be variables).
 This value is to be considered as the “main” exported value since it will be the simplest to import
-The name of imported module has to be the same as the name of the exported module for named exports.
-The naming of import is completely independent in default export and we can use any name we like.
ex: import MyReact, { MyComponent } from "react";
 correct wrong-namedExport

The React object
==========
-When we import React,we get a React object that contains methods and properties.
-React exposes its current version through the version property, here's how we can read that.
 import React from "react"
 console.log(React.version); //"16.9.0" / 18.2.0
 <h2>React Version is {React.version}</h2>
-in cmd run the below command
 npm view react version

React Emmet (react snippets - plugin)
==========
https://marketplace.visualstudio.com/items?itemName=rodrigovallades.es7-react-js-snippets

IMR - import React from 'react';
IMRD - import ReactDOM from 'react-dom'
IMRC - import React, { Component } from 'react'
IMPT - import PropTypes from 'prop-types'
RCC - React class component
RCE - React class Export component
RFC - React Functional Component
RFCE - React Functional Export Component
RMC - React Function Memo component
RCONST - constructor with super
RPC - React Class Pure Component
RPCE - React Class Pure Export Component
RAFC - React Arrow Function Component
RAFCE - React Arrow Function export Component
REN - render() { return( ) }
SST - this.setState({ })

extensions
==========
1. React Snippet
2. ESLint
3. prettier
4. code spell checker
5. gitlens
6. vscode-icons
7. Thunder Client

Browser extension
=================
json viewer

React Plugins
=============
1. React Developer Tools
2. React-sight
3. Redux DevTools
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()

React.StrictMode
================
-StrictMode is a tool for highlighting potential problems in a react application.
-It activates additional checks and warnings for its descendants(child elements).
-Strict mode checks are run in development mode only,they do not impact the production build.
-Strict Mode helps with the below things:
        Identifying components with unsafe lifecycles (componentWillMount)
        Warning about legacy string ref API usage
        Warning about deprecated findDOMNode() usage
        Detecting unexpected side effects
        Detecting legacy context API
-ex: import React, { StrictMode } from "react";
  <StrictMode>
            <App />
  </StrictMode>
Note: StrictMode renders components twice (on dev but not production) in order to
   detect any problems with our code and warn us about them.

Component
=========
-Components are the most basic UI building block of a React application.
-Each Component is responsible for outputting a small,reusable piece of HTML.
-A Component Represents a part of the User Interface.

-Components are Re-Usable and can be nested inside other component.
-A React application contains a tree of components.
-React components let you split the UI into independent, reusable pieces,
and think about each piece in isolation.

2 types Of Component
==========
1. Functional Component (stateless/presentational/dumb)
2. Class Component (statefull)

-The simplest way to define a component in React is to write a JavaScript function:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

-we can also use the ES6 class syntax to write components

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

-A functional component is just a plain JavaScript function which accepts props
as an argument and returns a React element.
-Functional Components are faster and much simpler than Class Components.
-Functional components are very useful in React, especially when you want to isolate state management from the component. That's why they are often called stateless components.

-React class components can be defined by extending React.Component or React.PureComponent.
-React component can be defined as an ES6 class that extends the base React.Component class.
-a React class component must define a render method that specifies how the component renders to the DOM.
-The render method returns React nodes, which can be defined using JSX syntax as HTML-like tags
-React requires that the first letter of a component class be capitalized.
This is required because based on capitalization JSX can tell the difference between
an HTML tag and a component instance. If the first letter of a name is capitalized,
then JSX realizes it's a component instance; if not, then it's an HTML tag.

Functional Component	Class Component
1. No 'this' keyword	1. More feature rich
2. solution without state	2. Maintain own private data- state
3. Mainly for UI	3. Complex UI Logic
4. Stateless/dumb/Presentational	4. Provide Life cycle hooks

When to use a Class Component over a Function Component?
==========
-If the component needs state or lifecycle methods then use class component
otherwise use function component.
-However, from React 16.8 with the addition of Hooks, we can use state , lifecycle methods and other features that were only available in class component in function component also.

-So, it is always recommended to use Function components, unless you need a React functionality whose Function component equivalent is not present yet, like Error Boundaries.

How to Use Bootstrap in a React application
==========
1. install bootstrap
  npm i bootstrap@3.3.7 (particular version)
      OR
  npm i bootstrap   (latest version)
  npm i bootstrap-icons

2. use 'bootstrap.min.css' in index.js file
  import "bootstrap/dist/css/bootstrap.min.css";
  import "bootstrap/dist/js/bootstrap.bundle.min.js";
  import 'bootstrap-icons/font/bootstrap-icons.css';

How to Use Bootstrap in a React application using CDN Link
==========
-Bootstrap can be used in a react application just by adding the below CDN link in 'index.html'
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6DOitfPri4tjfHxaWutUpFmBp4vmVor" crossorigin="anonymous">

How to use Bootstrap icons
==========

<i className="bi-airplane" style={{ fontSize: "2rem", color: "cornflowerblue" }}>

How to use React-icons
==========
1. install react-icons
  npm install react-icons
2. import react-icons in component
  import { FaEdit, FaTrash } from 'react-icons/fa';
  import { BsFillCalendarDateFill,BsFillClockFill } from 'react-icons/bs';
3. use the Icons
  <FaEdit />
  <FaTrash color='red' />

Fragments
==========
-While returning elements in JSX, we can only return 1 element at a time.
 That element can have children but we have to ensure that we are only returning 1 element at a time, or else we will get a syntax error. (ex: a function can return 1 value)

-Fragments are way to render multiple elements without using a wrapper element.
-Fragment acts as a wrapper without adding unnecessary divs to the DOM.
-We can use it directly from the React import, or deconstruct it.

```
import React from 'react';
<React.Fragment>
  <div>I am an element!</div>
  <button>I am another element</button>
</React.Fragment>
```

-// Deconstructed

```
import { Fragment } from 'react';
<Fragment>
  <div>I am an element!</div>
  <button>I am another element</button>
</Fragment>
```

-React version 16.2 simplified this process further, allowing for empty JSX tags to be interpreted as Fragments

```
<>
    <div>I am an element!</div>
    <button>I am another element</button>
</>
```

Data Binding
============
-Binding/Displaying the variable value in UI is called data binding.

Binding Type	Description	React Example
One-way Binding	State → UI	{value}
Two-way Binding	State ↔ UI using `onChange`	value + onChange

CSS in React
============
different ways to add CSS:
1. inline CSS
2. External CSS
3. global css(index.css)
4. CSS Modules
5. Conditional CSS

CSS Modules
===========
-This approach is designed to fix the problem of the global scope in CSS.
-A CSS Module is a CSS file in which all class names are scoped locally by default.
-CSS Modules allow to use the same CSS class name in different files without worrying about naming clashes.
-CSS Modules allow the scoping of CSS by automatically creating a unique classname of the below format.

```
ex: [filename]\_[classname]\_\_[hash]
```

-ClassNames are dynamically generated, unique, and mapped to the correct styles.
-When importing the CSS Module from a JS Module, it exports an object.

```
import styles from './Button.module.css'; // Import css modules stylesheet as styles
import './another-stylesheet.css'; // Import regular stylesheet
<button className={styles.error}>Error Button</button>;
```

-it will only apply the classname from css module even if both files have same 'error' class;

Conditional Rendering

==========

-It is a common use case to show or hide elements based on certain conditions.

-It allows us to render different elements or components based on a condition.

-use cases:

- Rendering external data from an API.
- Showing or hiding elements.
- Toggling application functionality.
- Implementing permission levels.
- Handling authentication and authorization.

-ways to implement conditional rendering:

- Using an if…else Statement
- Using a switch Statement
- Using Ternary Operators
- Using Logical && (Short Circuit Evaluation)

Ternary- {flag ? <h1>Hellllooooooo</h1> : null}

Short Circuit- {flag && <h1>Hellllooooooo</h1>}

List Items - Lists and Keys

==========

-A 'key' is a special attribute which should be included while creating List Items.

-There will be a warning message in the console if the key prop is not present on list items.

-'Key' gives the elements a stable unique Identity.

-'Key' helps react to identify which items have changed,are added, or removed.

-React relies on the key to identify items in the list.

React uses a virtual DOM, and it only re-draws the components that changed since the last render.

-The first time a component like IdiomaticReactList is rendered, React will see that you want to render a bunch of items, and it will create DOM nodes for them.

-The next time that component renders, React will say, “I already have some list items on screen are these ones different?” It will avoid recreating DOM nodes if it can tell that the items are the same.

-Tip: Avoid using the index as key, if the list is filtered or sorted

it will cause the key value to change to the new index and React will consider

them different elements and repaint the whole list.

-Why map() is used to iterate array instead of forEach()

JSX needs an array of items to render,forEach() does not return anything (i.e it returns undefined).

map() returns an array.

Read data from a JSON file

==========

1. create a JSON file (employees.json/products.json/countries.json)

[{},{},{}]
2. import EmployeeArr from './employees.json'
3. use EmployeeArr(List data) in JSX
```
{EmployeeArr.map((emp,ind)=>{
   return <option key={ind} value={emp} />
})}
```

Read data from a Javascript file
==========
1. create a JSON file (employees.js/products.js/countries.js)
```
const employees = [ {},{},{} ];
export default employees;
```
2. import EmployeeArr from './employees.json'
3. use EmployeeArr(List data) in JSX
```
{EmployeeArr.map((emp,ind)=>{
   return <option key={ind} value={emp} />
})}
```

Assignment
==========
1. create products.json with list of products.
   https://fakestoreapi.com/products
2. display the products in table & card

forceUpdate
==========
-Declaring class variables/function variables is always a bad idea.
-can be used for the below 2 use cases:
- Setting and clearing timeouts
- Storing frequently-referenced values

-React components will only re-render when there are changes to props or state.
-Updating the class/function variable does not trigger a re-render.
-Its our responsibility of triggering the re-render when our class data changes.
-Normally we should try to avoid all uses of forceUpdate() and only read from this.props and this.state in render().
-In class components,this.forceUpdate() is used to force a re-render.
-In Functional components,there is no concept of force re-render.

props
=====
-Props are inputs to components.
-Props stand for properties and is a special keyword in React.
-Attributes on Components get converted into an object called Props.
-Helps to Pass custom data to a component.
-props are used to pass data and methods from a parent component to a child component.
-data with props are being passed in a uni-directional flow. (one way from parent to child).
-We can pass props to any component as we declare attributes for any HTML tag
<ChildComponent someAttribute={value} anotherAttribute={value}/>

-React props passed with just their name have a value of true.
<myComponent showTitle={true}>
<myComponent showTitle>
-The props can be accessed as shown below
this.props.propName; (Class Component)
props.propName; (Functional Component)

1.They are immutable. data coming from the parent should not be changed by child components. we will get an error if you try to change their value.
2.data with props are being passed in a uni-directional flow. (one way from parent to child)

Props De-structuring
==========
-It's a JavaScript feature that allows us to extract multiple pieces of data from an array or object and assign them to their own variables.
-Improves readability.
-we can get rid of props/this.props in front of each property.

-props de-structuring in functional component
export default function Greet({ name, msg }){}
-we de-structure props in the render() function.(class Component)
ex: let { pId, name, price } = this.props.product;

PropTypes
=========
-React has some built-in typechecking abilities.
-To run typechecking on the props for a component,we can assign the special propTypes property.
-PropTypes exports a range of validators that can be used to make sure the data the component receive is valid.
-When an invalid value is provided for a prop, a warning will be shown in the browser console.
-For performance reasons, propTypes is only checked in development mode.not in production.

1. import PropTypes from 'prop-types';
2. ComponentName.propTypes = {
variableName: PropTypes.string
};

```
optionalArray: PropTypes.array,
optionalBool: PropTypes.bool,
optionalFunc: PropTypes.func,
optionalNumber: PropTypes.number,
optionalObject: PropTypes.object,
optionalString: PropTypes.string,
optionalSymbol: PropTypes.symbol,
optionalElement: PropTypes.element,
optionalEnum: PropTypes.oneOf(['News', 'Photos']),
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
```

```
 }),
```

Requiring Props
==========
-If prop values are not passed to a certain component, no error will be thrown.
Instead, within the component that prop will have a value of 'undefined'.
-Apart from specifying the type of the prop that can be passed to the component,
we can also make sure the prop is always provided to the component by chaining
isRequired at the end of the prop validator.

```
Student.propTypes = {
  name: PropTypes.string.isRequired, // Required Prop
  age: PropTypes.number  // Optional Prop
}
```

Default props
==========
-defaultProps allows to set default value for props.
-ex:

```
        Greet.defaultProps = {
                msg: 'this is my default message'
        }
```

-Functional Component & destructuring

```
 export default function Greet({name , msg='good morning'}) {
 }
```

Props.children
==========
-props.children represents the content between the tags of a Component.
-props.children can be an array or a single item.
-props.children is available on every component.
-<Welcome>Hello world!</Welcome>
-The string Hello world! is available in props.children in the Welcome component.
-

```
class Welcome extends React.Component {
  render() {
    return <p>{this.props.children}</p>;
  }
}
```

Ensuring Single Child
==========
-We can use PropTypes.element to enforce that only a single child can be passed
to a component as children.
-If we try to pass more than 1 child to the component, we will get an error.
Ex:

```
Greet.propTypes = {
   children: PropTypes.element
};
```

Prop Drilling
==========
-Prop drilling is the process in a React app where props are passed from one component to another by going through other parts that do not need the data, but only help in passing it through the intermediate components.
-The problem with this approach is that most of the components through which this data is passed have no actual need for this data.
-They are simply used as mediums for transporting this data to its destination component.
-as the components are forced to take in unrelated data and pass it to the next component, until it reaches its destination. This can cause major issues with component reusability and app performance.
-This Problem Can be avoided by using concepts like 'Context API' & 'REDUX'

Limitations of Local Variables
==========
No re-rendering : Changing a Local variable doesn't re-render, the component won't update to reflect the new value.
No Persistence : Local variable are re-created on each render, any changes made to them are lost after re-render

```
export default function StateDemo1() {
  let count = 0;
  const increment = ()=>{
    count++;
    console.log(count);
  }
  return <>
    <h2>State Demo 1</h2>
    <div>Count value is: {count}</div>
    <button onClick={increment}>Increment</button>
  </>
}
```

Why State variables
==========
Re-rendering : when state variable changes, React re-renders the component with new value
Persistence : states are persisted accross re-renders

States
======
-The State of a component is an object that holds some data that may change over the lifetime of the component.
-whenever the state object changes,react will re-render the component.
-States are Mutable(can be changed), and states are local to the component, cann't be used by other components directly
-Keep the state minimal , avoid using state for static data.

State in Class Component

==========
-class components have a state object , where all the state variables are declared.
To update state variables, setState() should be used;
-State object is usually initialized inside the constructor in class components.
-UI will not Re-render when we change state directly, setState() should be used.
this.counter = this.counter + 1; // re-render won't happpen
this.setState({counter:this.state.counter+1}); // re-render will happpen
-use setState() to change the state of react Component.
it ensures that render() gets called.
-If a piece of code needs to be executed only after the state has been updated, then
place that code in the callback function which is the second arguement to setState()
Syntax: setState(StateObject, callbackFunction);
ex: setState({} , ()=>{});

Note:

-setState() actions are asynchronous. setState() doesn't immediately mutate this.state.
-React may group multiple setState() calls in to a single update for better performance
-when we want to update the state based on the previous State value,
we need to pass a function as an arguement to setState() instead passing an object.
ex:- this.setState({ value: this.state.value + 1 });
this.setState(prevState => ({ value: prevState.value + 1 }));

State in Functional Component
==========
-State in functional Component can be maintained using useState() hook.
-useState() is used to declare a state variable in function component.
-The only argument to the useState() Hook is the initial state.
-It returns a pair of values:
a. State value
b. function to update that state value.
ex: const [count, setCount] = useState(0);
-state updation is asynchronous,
```
const increment = () => {
 setCount(count + 1);
        console.log(count);  // this will be executed before state update
};
```
-If a piece of code needs to be executed only after the state has been updated, then place that code in the 'useEffect()'
-when we want to update the state based on the previous State value,pass a callBack to setter
ex: setCount((prevState) => prevState + 1);
setCount(count + 1);

Props	vs	State
1. Props are immutable.		1. State is mutable/modifiable.
2. pass data from parent component to child component		2. contains own data & changes over time
3. communicate between components		3. rendering dynamic changes
4. props - Functional Comp this.props - class comp		4. useState() - Functional Comp this.state={} - Class Comp

Assignments:
1. Have a paragraph & a toggle button, on click of the button control the visibility(Show/Hide) of the paragraph
2. Create a text-area with maxLength=100, as the user keeps typing display updated Remaining characters.
3. Create dropdown with state names, when user changes dropdown value, print the selected (stateName)dropdown value in a div
4. have 2 inputBoxes, and a dropdown(+,-,*,/) and perform arithmetic operations
5. create a input box, toggle the type of that input box to (text/password)
6. create a counter example with 3 controls(increment,decrement,reset)
7. Have a toggle button and control Dark/Light theme of a page.
8. Temperature Converter (celcius to fahrenheit)

How to use SweetAlert
==========
https://sweetalert2.github.io/#examples

1. npm i sweetalert2
2. import Swal from 'sweetalert2'
3. on button click call a function, which has the below code

```
Swal.fire(
  'Good job!',
  'You clicked the button!',
  'success'
)
```

How to use react-modal
==========
1. npm i react-modal
2. import Modal from 'react-modal';
3. use below Modal code in the component

```
let [modalIsOpen, setIsOpen] = useState(false);
function openModal() {
        setIsOpen(true);
}
function closeModal() {
        setIsOpen(false);
}
<>
<button onClick={openModal}>Open Modal</button>
<Modal
  isOpen={modalIsOpen}
  onRequestClose={closeModal}
  contentLabel="Example Modal"
>
  <h2>Hello Hiiiiiiiiii</h2>
  <button onClick={closeModal}>close</button>
  <div>I am a modal</div>
</Modal>
</>
```

How to use react-bootstrap modal
==========
1. npm install react-bootstrap bootstrap
2. import Modal from 'react-bootstrap/Modal';
3. const [show, setShow] = useState(false);
   const handleClose = () => setShow(false);
   const handleShow = () => setShow(true);

```
  <Modal show={show} onHide={handleClose}>
      <Modal.Header closeButton>
        <Modal.Title>Modal heading</Modal.Title>
      </Modal.Header>
      <Modal.Body>Woohoo, you're reading this text in a modal!</Modal.Body>
      <Modal.Footer>
        <Button variant="secondary" onClick={handleClose}>
          Close
        </Button>
        <Button variant="primary" onClick={handleClose}>
          Save Changes
        </Button>
      </Modal.Footer>
   </Modal>
            <Button onClick={handleShow}>click me</Button>
```

Search
======
1. Install react-js-search from npm
   npm i react-js-search
2. Import SearchBar in component
   import SearchBar from 'react-js-search';
3. use the below HTML

```
  <SearchBar
      onSearchTextChange={(term, filteredData) => {
        setFilteredEmployees([...filteredData]);
      }}
      onSearchButtonClick={onSearchClick}
      placeHolderText={"Search here..."}
      data={employees}
    />
```

Pagination
==========
1. npm install react-paginate
2. import ReactPaginate from 'react-paginate';
3. Add the below code in a component

```
    <ReactPaginate
      breakLabel="..."
      nextLabel="next >"
      onPageChange={handlePageClick}
      pageRangeDisplayed={5}
      pageCount={pageCount}
      previousLabel="< previous"
```

```
      renderOnZeroPageCount={null}
    />
```
Note: For complete code plz refer 'react_programs.txt' file

Assignment
==========
1. create products.json with list of products.
  https://fakestoreapi.com/products
2. display the products as cards
3. add a Search bar to search products
4. add pagination
5. Add 2 buttons to sort the products by price(asc,desc)
6. crate a filter dropdown with category values(clothing/electronics), display the products of selected category only

React Events
============
-React events are written in camelCase
```
        onClick-correct
        onclick-wrong
```
-React event handlers are written inside curly braces
```
        onClick={shoot}  instead of onClick="shoot()"
        <button onClick={shoot}>Take the Shot!</button>   // calls the function on click
        <button onClick={shoot()}>Take the Shot!</button> // calls the function on load
```
-For methods in React,the 'this' keyword should represent the component that owns the method.
-That is why we should use arrow functions.
With arrow functions, 'this' will always represent the object that defined the arrow function.
-In class components, the this keyword is not defined by default,
so with regular functions the this keyword represents the object that called the method, which can be the global window object, a HTML button, or whatever.
-If you must use regular functions instead of arrow functions you have to
bind 'this' to the component instance using the bind() method.
```
  constructor(props) {
    super(props);
    this.f1 = this.f1.bind(this);
  }
  state = {
    counter: 0
  }
  f1() {
    alert("hi");
    console.log(this)
    console.log(this.state.counter)
  }
```

-If you want to send parameters into an event handler, you have two options:
1. Make an anonymous arrow function:
```
        shoot = (a) => {
   alert(a);
  }
  <button onClick={() => this.shoot("Goal")}>Take the shot!</button>
```

2. Bind the event handler to this

```
  shoot(a) {
   alert(a);
 }
 <button onClick={this.shoot.bind(this, "Goal")}>Take the shot!</button>
```

-If you send arguments without using the bind method, (this.shoot(this, "Goal") instead of this.shoot.bind(this, "Goal")),
the shoot function will be executed when the page is loaded instead of waiting for the button to be clicked.

Single Event Handler for multiple input elements

```
================================================
 changeHandler = (e) => {
   const { name, value } = e.target;
   this.setState({ [name]: +value });
 };
================================
let [obj, setObj] = useState({ num1: 0, num2: 0 });
const changeHandler = (e) => {
  setObj({ ...obj, [e.target.name]: +e.target.value })
}
```

SyntheticEvent

```
==============
```

-SyntheticEvent is a cross-browser wrapper around the browser's native event.
-in React event handlers receive 'SyntheticEvent' object as arguement insteadof browser's native 'Event' object.

-To register an event handler for the capture phase, append Capture to the event name.

```
 <button onClick={f1}>click me</button>  // bubbling
 <button onClickCapture={f1}>click me</button>  // capturing

 <div onClickCapture={() => console.log('div clicked')}>
        div <br/>
        <span onClickCapture={() => console.log('span clicked')}>
           Span <br/>
           <button onClickCapture={() => console.log('button clicked')}>button</button>
        </span>
           </div>
```

Component Communication

```
======================
```

->Parent to Child : props
->Child to Parent : callback and states
->Between Siblings : Combine the above 2

Child To Parent:
1. Define a function in parent which takes the data as a parameter.

2. Pass that function as a prop to the child.
3. Call the function using this.props.[callback] in the child, pass in the data as the argument.

Assignment
==========
1. create 1 EmployeeCRUD component
2. display list of employees in a table(data comes from an array)
3. user should be able to delete Employee (ask user confirmation)
4. view the details of each employee in a modal (bootstrap Modal)
5. add a new employee to the table (insert a new record to the array)
   use SnackBar to display message ('Employ Added Successfully' - message should be maintained in a constant file)

Assignment
==========
Todo CRUD Component with below functionalities
1. Add Todo : User types a todo and clicks “Add” , add the item to the todo list
2. Toggle Completion : User checks a checkbox to mark as completed or not completed
3. Delete Todo : Each todo item should have a delete button to remove it from the list.
4. Completed Count : Show the number of completed todos (e.g., 3 out of 5 tasks completed)

Component vs PureComponent
==========
-PureComponent is exactly the same as Component except that it handles the 'shouldComponentUpdate' method for us.
-When props or state changes, PureComponent will do a shallow comparison on both props and state.if there is a change in state/props, then only render() will be called.
-A normal Component always calls render() when we update the state. even though there is no change in the last state data and current state data.
-Class components that extend the React.PureComponent class are treated as pure components.
```
Ex: class myComp extends React.PureComponent {
        }
```
-in Functional components, re-render happens only if the state changes.
-Every Functional Component is a Pure-Component.

React Memo Component
==========
-Introduced in React v16.6. This improves performance.
-React.memo() is a higher-order component/function.
-React.memo() can be used for both class & function components.
-React.memo() is used with child components only(the component receiving porps)
-When a component is wrapped in React.memo(), React renders the component only if the props those are passed to that component changes.
-Avoids re-rendering a component when its props are unchanged.
-ex: export default React.memo(MyComponent);

useMemo()
========
-The functions called inside component are re-invoked on every state change.
-If we want the functions should be called only on component load and not on every state changes then for that function,
useMemo should be used.
-useMemo Hook returns a memoized value.
-The useMemo(()=>{} , []) hook accepts a second parameter to declare dependencies.
The function will only run when its dependencies are changed.
-The main difference between usememo() and useCallBack() is that
useMemo() returns a memoized value and useCallback() returns a memoized function.
ex:

```
// const calculation = expensiveCalculation(count); // function gets called on every state change
// const calculation = useMemo(() => expensiveCalculation(count), []); // function gets called on page load
const calculation = useMemo(() => expensiveCalculation(count), [count]); // function gets called on count change
```

useCallBack()
==========
-useCallback() Hook returns a memoized callback function.
-useCallback() Hook only runs when one of its dependencies update.
-useCallback() should be used to prevent a component from re-rendering unless its props have changed.
-When a function is passed as a props from parent to child component,child component re-renders even if no props data changes(even though the child is a memo component)
-the function that gets passed as a props gets a new reference everytime there is a state change in parent component.
ex:

```
<ToDoDemo todos={todos} addTodo={addTodo} />

const addTodo = () => {
        setTodos((t) => [...t, "New Todo"]);
};
----------
const addTodo = useCallback(() => {
  setTodos((t) => [...t, "New Todo"]);
}, []);
```

LifeCycle Hooks
==========
-Every component in React goes through a lifecycle of events.
-The three phases are:
1.Mounting - (constructor,getDerivedStateFromProps,render,componentDidMount)
2.Updating - (getDerivedStateFromProps,shouldComponentUpdate,render,getSnapshotBeforeUpdate, componentDidUpdate)
3.Unmounting - (componentWillUnmount)

Mounting: means putting elements into the DOM.

1.constructor()
2.static getDerivedStateFromProps(props,state)
3.render()
4.componentDidMount()

Note:- The render() method is required and will always be called,
the others are optional and will be called if you define them.

constructor()

-The constructor() method is called before anything else, when the component is initiated.
-It is the natural place to set up the initial state and other initial values.
-The constructor() method is called with the props, as argument,
and you should always start by calling the 'super(props)' before anything else,
Otherwise,this.props will be undefined.
-This will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).
-If you neither initialize state nor bind methods for your React component, there is no need to implement a constructor for React component.
-setState() method should not be called in the constructor(). we will get console error
error - Can't call setState on a component that is not yet mounted

static getDerivedStateFromProps()

-The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.
-This is the natural place to set the state object based on the initial props.
-It takes (props,state) as argument, and returns an object with changes to the state.
-only fires when the parent causes a re-render and not as a result of a local setState.

render()

-The render() method is required, and is the method that actual outputs HTML to the DOM.
-it gets re-invoked when state/props data changes.

componentDidMount()

-The componentDidMount() method is called after the component is rendered.
-This is a good place to initiate the network request.
-if we are going to fetch any data from an API then API call should be placed in this lifecycle method,and then we get the response, we can call the setState() method and render the element with updated data.
-good place for DOM manipulation.

Updating
==========
1.static getDerivedStateFromProps(props,state)
2.shouldComponentUpdate()

3.render()
4.getSnapshotBeforeUpdate(prevProps, prevState)
5.componentDidUpdate()

getDerivedStateFromProps()

-while updating state/props getDerivedStateFromProps() method is called.
-This is the first method that is called when a component gets updated.
-This is still the natural place to set the state object based on the initial props.

shouldComponentUpdate()

-In the shouldComponentUpdate() method a boolean value should be returned that specifies whether React should continue with the rendering or not.
-The default value is true.
-shouldComponentUpdate() lifecycle shouldn't be added if the class is extending React.PureComponent.

getSnapshotBeforeUpdate(prevProps, prevState)

-In the getSnapshotBeforeUpdate(prevProps, prevState) method we have access to the props and state before the update,meaning that even after the update, we can check what the values were before the update.

example:
-When the component is mounting it is rendered with the favorite color "red".
-When the component has been mounted, a timer changes the state, and after one second,the favorite color becomes "yellow".
-This action triggers the update phase, and since this component has a getSnapshotBeforeUpdate() method,this method is executed, and writes a message to the empty DIV1 element.

componentDidUpdate()

-The componentDidUpdate() method is called after the component is updated in the DOM.
-componentDidUpdate() is invoked immediately after the state is updated.
-This method is not called for the initial render,componentDidMount() will be called for the initial render.
-it gets called only when state/props gets updated.

Unmounting - Removing the component from DOM
==========
-The next phase in the lifecycle is when a component is removed from the DOM,or unmounting.
-React has only one built-in method that gets called when a component is unmounted.

componentWillUnmount()

-Called immediately before a component is destroyed.
-Perform any necessary cleanup in this method, such as cancel network requests,
or cleaning up any DOM elements created in componentDidMount.

-clearTimeout, ClearInterval , Unsubscribe, detachEventHandlers

useEffect()

-useEffect serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount.
-useEffect() takes a callBack Function as 1st argument and Dependency arrray as 2nd argument.
ex: useEffect(()=>{} , []);
-array contains dependencies for useEffect,variables on which useEffect depends on to re-run.
-If 2nd argument is not present, effect runs everytime there is a state change.
-When 2nd argument is there and the array is empty, the effect runs only once.(on component load)
-If 2nd arguement is present, effect will only run if the values in the list change.
-we can have many useEffects() in a component to track changes for diff variables.

How to use componentWillUnmount in Functional Components
==========
```
import React, { useEffect } from 'react';
const ComponentExample => () => {
   useEffect(() => {
      // Anything in here is fired on component mount.
      return () => {
         // Anything in here is fired on component unmount.
      }
   }, [])
}
```

Note: When the state is updated, the component gets re-rendered, Hence the component gets unmounted and gets rendered again.

useLayoutEffect
==========
-useLayoutEffect is a version of useEffect that fires before the browser repaints the screen.
-This runs synchronously immediately after React has performed all DOM mutations.
-useLayoutEffect(callBack, dependencies?)
-useLayoutEffect: If you need to mutate the DOM inside the effect()

Refs (Template Reference Variable)
==========
-reference variables: help to use data from one part of a template in another part of the template.
-with reference variables, we can perform tasks such as respond to user input or fine tune your application's forms.
-Refs provide a way to access DOM nodes or React elements.
-used to modify child comp (OR) DOM.
-Managing focus, text selection , animations.
-ex: this.myRef1 = React.createRef(); (class component)
const myRef1 = useRef(); (function component)

-React supports another way to set refs called “callback refs”,
which gives more fine-grain control over normal refs when refs are set and unset.

ex: <input ref={x => inputRef1=x} />
alert(inputRef1.value)

-'Ref forwarding' is a feature that lets some components take a ref they receive,and pass/forward it further down to a child.

Q. How to display previous State & Current State data in functional Component using ref.

Forms
=====
-React uses forms to allow users to interact with the web page.(Collect User Data)
-control the values of more than one input field by adding a name attribute to each element.
-get the field value by using the 'event.target.value',
get the field name by using 'event.target.name'
-control the submit action by adding an event handler with onSubmit attribute.

```
<form onSubmit={this.submitHandler}>
  <button>submit</button>
</form>
```

-use event.preventDefault() to prevent the form from actually being submitted.(to avoid page refresh)

-A form can be validated on change or while submitting the form.
-The textarea element in React is slightly different from HTML Syntax.
-In HTML the value of a textarea was the text between the start tag <textarea> and the end tag </textarea>,
in React the value of a textarea is placed in a value attribute:

```
<textarea value={this.state.description} />   correct
<textarea>{this.state.description}<textarea>  wrong
```

-A drop down list, or a select box, in React is also a bit different from HTML.
-in HTML, the selected value in the drop down list was defined with the selected attribute:

```
        <select value={this.state.mycar}>
      <option value="Ford">Ford</option>
      <option value="Volvo">Volvo</option>
      <option value="Fiat">Fiat</option>
    </select>   //correct

          <select>
            <option value="Ford">Ford</option>
            <option value="Volvo" selected>Volvo</option>
            <option value="Fiat">Fiat</option>
          </select> //wrong
```

Feature	Uncontrolled	Controlled
One-time value retrieval (e.g. on submit)	yes	yes

Validating on submit	yes	yes	yes
Default Value	yes	yes	yes
Field-level Validation	no	yes	yes
Conditionally disabling submit button	no	no	yes
Enforcing input format	no	yes	yes
dynamic inputs	no	yes	yes

-Uncontrolled inputs are like traditional HTML form inputs.
-Form data is handled by the DOM itself.
-They remember what we typed. we can then get their value using a ref.
-To write an uncontrolled component, instead of writing an event handler for every state update, you can use a ref to get form values from the DOM.
-you have to 'pull' the value from the field when you need it.(ex:-form submit)
-there is no updating of any state when you change the input-box value.
-Let React to specify the initial value, but leave subsequent updates uncontrolled.
-To handle this case, you can specify a 'defaultValue' attribute instead of value.

-Controlled form components are defined with a value property.
-value of controlled input is managed by React,
-With a controlled component, the input's value is always driven by the React state.
-component state and the input value is in sync at all the times.

```
<input onChange={this.onChange} value={this.state.name} />
```

NPM Libraries for Form Handling
==========
1. https://www.npmjs.com/package/react-hook-form
2. https://www.npmjs.com/package/formik
3. https://www.npmjs.com/package/yup

HTTP Methods
==========
GET - Retrieve a resource/Retrieve data from DB
search

POST - to send data to server (sign up)
(create a resource/create a new record in DB)
to fetch data securly (send params in body not in URL)

PUT - update data/create or replace a resource
update user's profile information

PATCH - update/modify a particular resource(partial update)
update user password

DELETE - Remove a resource/delete a record from DB
Delete naukri account

PUT vs POST

-POST for CREATE operation, PUT is for create & update.
-PUT is idempotent, where POST is non-idempotent.
-Idempotence(producing the same result even if the same request is made multiple times)
-(PUT)if you retry a request N times, that should be equivalent to single request modification.
-(POST)if you retry the request N times, you will end up having N resources with N different URIs created on server
-Use PUT when you want to modify a singular resource which is already a part of resources collection.
-Use POST when you want to add a child resource under resources collection.

PUT vs PATCH

-PUT is used to replace an existing resource.
-PATCH is used to apply partial modifications to a resource.

http status codes
==========
1xx Informational (100-Continue,101-switching Protocols,102-processing)
2xx Success (200-OK,201-created,202-accepted,204-No Content)
3xx Redirection (300-Multiple Choices,301-Moved Permanently,302-Found,304-Not Modified)
4xx Client Error (400-Bad Request,401-Unauthorized,403-Forbidden,404-Not Found)
5xx Server Error (500-Internal Server Error,502-Bad Gateway,503-Service Unavailable)

POSTMAN
=======
-Application, used to Test REST APIs. (chrome://apps/)
-Send requests, get responses, and easily debug REST APIs.
-Browser Extension / Application
https://www.postman.com/downloads/
-Thunder Client(VSCode Extension) is an alternate to postman

Fake Online REST API for Testing
---
1. https://jsonplaceholder.typicode.com/
2. https://reqres.in/
3. https://fakestoreapi.com/products
4. https://api.github.com/users/google
5. https://dummyjson.com/products
6. https://dummy.restapiexample.com/
7. https://my-json-server.typicode.com/horizon-code-academy/fake-movies-api/movies
8. https://api.publicapis.org/entries

Create REST API with json-server
---
https://medium.com/@devmrin/create-a-rest-api-json-server-in-less-than-1-minute-acf286600f03

1. Install json-server (not necessarily in a react project)
   npm install -g json-server
2. create a json file and add some data (not necessarily in a project)
   db.json (filename can be anything.json)
3. start json server
   json-server --watch db.json --port=4000

http://localhost:3000/employees
GET /employees
GET /employees/{id}
POST /employees
PUT /employees/{id}
PATCH /employees/{id}
DELETE /employees/{id}

Ways of Fetching Data
==========
There are many ways to extract data from API in React:
1. using fetch() , then()
2. using async-await syntax
3. using Axios library
4. using custom hooks

HTTP with fetch()
==========
```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(finaldata => console.log(finaldata))
```

Async Await
==========
```
 const fetchProducts = async function () {
   const products = await fetch("https://fakestoreapi.com/products");
   const productsJSON = await products.json();
   setProducts(productsJSON);
  };
```

HTTP with axios
==========
1. install axios
   npm i axios
2. import axios to component
   import axios from 'axios';
3. Use the http methods
```
   const fetchUsers = async () => {
                const url = "https://jsonplaceholder.typicode.com/users";
                const response = await axios.get(url);
```

```
                setUsers(response.data);
  };
```

```
                Axios
                        fetch()
==========================================================================
```

-built-in XSRF protection. -Fetch does not.
-uses the data property. -Fetch uses the body property.
-data contains the object. -Fetch's body has to be stringified.
-request is ok when status is 200 and statusText is 'OK'. Fetch request is ok when response object contains the ok property.
-performs automatic transforms of JSON data. Fetch is a two-step process when handling JSON data- first, to make the actual request; second, to call the .json() method on the response.
-allows cancelling request and request timeout. Fetch does not.
-has the ability to intercept HTTP requests. Fetch, by default, doesn't
-has built-in support for download progress. Fetch does not support upload progress.

Create an Axios Instance
========================
1. Create a separate file named api.js(can be anything) and add the below code

```
import axios from 'axios';
const client = axios.create({
  baseURL: `http://jsonplaceholder.typicode.com/`
});
export default client;
```

2. Once the default instance is set up, it can then be used anywhere

```
import client from 'api.js'
client.get('/users')
```

Making multiple requests with axios
===================================

```
const promise1 = axios.get('https://api.github.com/users/defunkt')
const promise2 = axios.get('https://api.github.com/users/evanphx')
const [response1, response2] = await axios.all([promise1, promise2]);
```

Http Interceptors
=================
-Interceptor is a feature that allows an application to intercept/modify requests or responses before they are handled by .then() or the .catch()

httpRequest ----> Interceptor ----> ModifiedRequest ----> Server
ModifiedResponse<---- Interceptor <---- httpResponse <---- Server

Request interceptor use cases:
-Assume you want to check before making a request if your credentials are valid.

So, instead of actually making an API call, you can check at the interceptor level that your credentials are valid.
-Assume you need to attach a token to every request made, instead of duplicating the token addition logic at every Axios call, you can make an interceptor that attaches a token on every request that is made.

```
export function myInterceptor() {
  axios.interceptors.request.use((req) => {
    req.headers.authorization = "my secret token";
    return req;
  });
}
```

-call the above function in app.js

response interceptor use cases -
-Assume you got a response, and judging by the API responses you want to deduce that the user is logged in.
So, in the response interceptor, you can initialize a class that handles the user logged in state and update it accordingly on the response object you received.
-Assume you have requested some API with valid API credentials, but you do not have the valid role to access the data.
So, you can trigger an alert from the response interceptor saying that the user is not allowed.
This way you'll be saved from the unauthorized API error handling that you would have to perform on every Axios request that you made.

url-->response-->interceptor-->modifiedResponse-->component

```
axios.interceptors.response.use(
  res => res,
  err => {
    if (err.response.status === 404) {
      throw new Error(`${err.config.url} not found`);
    }
    throw err;
  }
);
```

Run interceptor only for few APIs

```
---------------------------------
axios.interceptors.request.use((req) => {
    console.log(req);
    if (req.url.includes("users")) {
      req.headers.authorization = "my secret token";
    }
    return req;
  });
```

remove an interceptor

```
--------------------
const myInterceptor = axios.interceptors.request.use(function () {/*...*/});
axios.interceptors.request.eject(myInterceptor);
```

add interceptors to a custom instance of axios

```
const client = axios.create();
client.interceptors.request.use((req) => {
        // Logic
});
```

React Higher Order Components (HOCs)
==========
-A higher-order component (HOC) is a technique in React for re-using component logic.
-To Share Common Functionalities accross components without repeating the code.
-Higher order Component takes one or more components as arguments, and return a new upgraded component.
  newComponent = higherOrderComponent(originalComponent)
-Higher order components are JavaScript functions used for adding additional functionalities to the existing component.
-These functions are pure, which means they are receiving data and returning values according to that data.
-Authentication,Logging,Styling and Theming
-the effect written inside HOC runs only when props change, if a component(Counter) updates its own state(count) without any changes in props, the effect won't run.

use cases:
-Infinite scroll in three different views, all having different data.
-Components using data from third party subscription.
-Components that need logged in user data.
-Showing multiple lists(e.g. Users, Locations) with search feature.

https://www.codingame.com/playgrounds/8595/reactjs-higher-order-components-tutorial

Routing
=======
-Single Page Applications(SPAs) are web applications that load a single HTML page and
 dynamically update that page as user interacts with the application.
-Complete page Re-load doesn't happen. only a portion of a page gets loaded.
-Routing in a Single Page Application is the way to introduce some features for navigating the application through links.
-Every time a link is clicked or browser URL changes, React Router makes sure
 our application loads component accordingly.
-The browser should change the URL when we navigate to a different screen.
-The browser back and forward button should work as expected.
-Routing links together your application navigation with the navigation features offered by the browser: the address bar and the navigation buttons.
-React Router offers a way to write your code so that it will show certain components of your app only if the route matches what you define.

Types of routes:

==========
React Router provides two different kind of routes:
1. BrowserRouter (builds classic URLs)
2. HashRouter (builds URLs with the hash)

https://application.com/dashboard // BrowserRouter (IE>9 (OR) any other browser)
https://application.com/#/dashboard // HashRouter (IE<9)

-Which routes to use is mainly decided by the browsers we want to support.
-BrowserRouter uses the History API, which is relatively recent, and not supported in IE9 and below.
-If we don't have to worry about older browsers, BrowserRouter is the recommended one to use.

-Below are the 3 components used the most while working with React Router are:
1. BrowserRouter, usually aliased as Router (wraps all your Route components)
2. Link (used to generate links to your routes)
3. Route (showing or hiding the components they contain)

Steps for Routing
==========
1. Install React Router DOM
npm install react-router-dom
2. Create Components
create components like - (Home,About,Contact,NotFound)
3. set up the application to work with React Router. (index.js)
```
import { BrowserRouter } from 'react-router-dom';
<BrowserRouter>
    <App />
</BrowserRouter>
```
4. add the <Routes> element (ensures that only one component is rendered at a time)
and add <Route> to create the link between components (Body.js)
```
<Routes>
    <Route exact path="/" element={<Home />} />
    <Route exact path="/home" element={<Home />} />
    <Route exact path="/aboutus" element={<AboutUs />} />
    <Route exact path="/products" element={<Products />} />
    <Route path="*" element={<NotFound />} /> (No Match Route)
</Routes>
```
5. Add a Link for each component and use to="URL" to link them.
```
<div>
  <Link to="/">Home </Link>
  <Link to="/about">About Us </Link>
  <Link to="/shop">Shop Now </Link>
</div>
```

Link vs NavLink
==========
-When we use <Link> there isn't any active class on selected element.
-with <NavLink> the selected element is highlighted because this element adds an active class.
<NavLink to='home' className='nav-link'>Home</NavLink>
-add below css:
nav a.active{text-decoration:none;font-weight:bolder;background-color: aqua}

Navigate from one Route to another Route
==========
<Link to="/products">go to Product</Link>

Navigating Programatically
==========
```
import React from "react";
import { Link } from "react-router-dom";
import { useNavigate } from "react-router-dom";
export default function AboutUs() {
 const navigate = useNavigate();
 const func1 = function () {
  alert("do something");// logic
  navigate('/products');
 };
 return (
  <>
   <h1>this is about us component</h1>
   <Link to="/products">Take me to products page</Link>
   <button onClick={func1}>Take me To product</button>
  </>
 );
}
```

Navigate to previous/next route
==========
```
const navigate = useNavigate();
<button onClick={()=>navigate(-1)}>Go Back</button>
<button onClick={()=>navigate(1)}>Go Next</button>
```

Route Params
==========
1. Path params (productdetails/101)
2. Query Params (search?name=sachin&age=25)

Path Params
---
1. configure the route
```
  <Route path="/productdetails/:id" element={<ProductDetails />} />
```
2. create a link to that route (ProductList - for Every Product)
```
  <Link to={`/productdetails/${id}`}> View Details</Link>
```
3. collect the params data and display
```
  const { id } = useParams();
  <h1>This is the details of product - {id}</h1>
```

Query params

---

1. configure the route

```
<Route path="/productdetails" element={<ProductDetails />} />
```

2. create a link & navigation logic

```
const navigate = useNavigate();
const navigateHandler = (title, price) => {
  navigate({
    pathname: "/productdetails",
    search: `?${createSearchParams({ title, price })}`,
  });
};
<button className="btn btn-secondary" onClick={() => {
        navigateHandler(title, price);
      }}>query param</button>
```

3. collect params data and display

```
const [searchParams] = useSearchParams();
useEffect(() => {
  const currentParams = Object.fromEntries([...searchParams]);
  console.log(currentParams); // get new values onchange
        console.log(searchParams.get("title"), searchParams.get("price"));
}, [searchParams]);
```

Nested Routing
==========

-nested routing is used so that a parent component has control over its child component at the route level.

-Route: Products (nested Routes: featured Products, New products)

```
        <Route path="/products" element={<Products />}>
    <Route path="featured" element={<FeaturedProducts />} />
    <Route path="new" element={<NewProducts/>}/>
  </Route>
```

-add a nav in Parent(products) component, and a outlet

```
  <div className="courses-nav">
    <Link to="featured">featured</Link>
    <Link to="new">New</Link>
   </div>
   <Outlet />
```

Index Route
==========

-A child route with no path that renders in the parent's outlet at the parent's URL.

-Child Path would remain same like parent route.

```
        <Route path="/products" element={<Products />}>
                <Route index element={<FeaturedProducts />} />
    <Route path="featured" element={<FeaturedProducts />} />
    <Route path="new" element={<NewProducts/>}/>
  </Route>
        <Outlet />
```

Protected Routes
==========
-Protected Routes are routes that can only be accessed if a condition is met(usually, if user is properly authenticated).
-It returns a Route that either renders a component or redirects a user to another route based on a set condition.

-create a component that accepts another component and other route details as props.
-check a condition to confirm if user is authenticated or not.
-if the value is true, render the component, else, Redirect route to /signin page.

Replace
=======
-replaces the current location instead of pushing a new one onto the browser history stack.
-Without replace : Adds a new entry to the browser history (Back button will go back)
-With replace : Replaces current URL (Back button won't return to the previous URL)
-HTML Code : <Navigate to="/home" replace />
-Javascript code: navigate("/home", { replace: true })
-After successful Login,we use the above code to go to home page
'replace' attribute prevents users from pressing Back and returning to the login page after successful login.
-When to Use replace:
1. After login/logout to prevent users from going back to the login page.
2. For redirects where you don't want the previous page in history.
3. For reset navigation flows.

Exact
=====
-The exact param disables the partial matching for a route and makes sure that it only returns the route if the path is an EXACT match to the current url.
-If you're using React Router v6+, don't use exact—it's no longer needed
-In react-router-dom v6, all routes are matched exactly by default.

Code Splitting/Lazy Loading
==========
-Code-Splitting is a feature supported by bundlers like Webpack which can create multiple bundles that can be dynamically loaded at runtime.
-The best way to introduce code-splitting into your app is through the dynamic import()

-Lazy loading helps to load a module/component whenever it is required(on-demand).
-React.lazy() allows to render a dynamic import as a regular component.
before - import OtherComponent from './OtherComponent';
after- const OtherComponent = React.lazy(() => import('./OtherComponent'));
-lazy component should then be rendered inside a Suspense component, which allows us to show some fallback content (such as a loading indicator) while we're waiting for the lazy component to load.

```
<Suspense fallback={<div>Loading...</div>}>
  <OtherComponent />
</Suspense>
```

Q: What are the ways to handle errors in react application?
Ans: 1. try-Catch 2. Error Boundary

when To Use try-Catch
==========
-To handle errors in specific blocks of code.
```
ex: try{
                axios.get()
        }catch(){

        }
```
-To handle errors in event handlers.
-To handle errors in server-side rendering.

Error Boundaries
==========
-A JavaScript error in a part of the UI shouldn't break the whole application.
To solve this problem, React 16 introduced a new concept "error boundary"
-Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.
-Error boundaries do not catch errors for below things:
- Event handlers
- Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
- Server side rendering
- Errors thrown in the error boundary itself (rather than its children)

-Error Boundary is supported only in class components , not in functional components.
-A class component becomes an error boundary if it defines componentDidCatch().
-Error boundaries work like a JavaScript catch {} block, but for components.
-Error boundaries only catch errors in the components below them in the tree.
An error boundary can't catch an error within itself.
-We can use static getDerivedStateFromError() to render a fallback UI when an error has been thrown, and can use componentDidCatch() to log error information.

React Profiler
==========
-Profiler measures how often a React application renders and what the "cost" of rendering is.
-A Profiler can be added anywhere in a React tree to measure the cost of rendering that part of the tree.
-It requires two props: an id (string) and an onRender callback (function)
-For example, to profile a Navigation component and its descendants:
```
<Profiler id="Navigation" onRender={callbackToProcessRenderInfo}>
  <Navigation {...props} />
</Profiler>
function callbackToProcessRenderInfo(id, phase, actualDuration, baseDuration, startTime, commitTime) {
 // we can log it to a database or render it out as a chart
 logToDatabase({ id, phase, actualDuration, baseDuration, startTime, commitTime })
}
```
-Profiling adds some additional overhead, so it is disabled in the production build.

-Although Profiler is a light-weight component, it should be used only when necessary.

Pre-requisites:

-The application should have React version 16.5 or above.
-The React DevTools Extension needs to be installed in your browser.

```
<Profiler id="counterExample" onRender={console.log}>
   <StateDemo2 />
</Profiler>
```

React Portals
==========
-Portals provide a way to render components outside the main DOM tree.
-Normally, React Components render inside #root DOM Node. ex:
-With Portals, we can render components outside of it(#root) , but still react capabilities can be used (props,state,context etc)
-Poratls can be used for Modals, Tooltips, Dropdowns, Toasts/Notifications

-createPortal()
 createPortal(children, domNode, key?)

```
import { createPortal } from 'react-dom';
<div>
  <p>This child is placed in the parent div.</p>
  {createPortal(
    <p>This child is placed in the document body.</p>,
    document.body
  )}
</div>
```

Build and Deploy React App
==========
-Building an application means creating identifiable software assembly from source code which someone can use.
-Deploying an application refers to making the application accessible to the world through internet.
-"npm run build" creates a 'build' directory with a production build of our app.

  https://cra.link/deployment

Deploy in a Local Server
==========
-Install a server and deploy our project
 1. install a server
        npm i -g serve
 2. go to project(build) folder, open command prompt and run the below command
    'serve' or 'serve -l 7000'

Generated Files:
        main.[hash].js -   This is our application code

[number].[hash].chunk.js - vendor code
runtime-main.[hash].js - webpack runtime logic which is used to load and run your application

Deploy Application in Server
==========
-AWS
-Azure
-Firebase
-GithubPages
-Netlify
-Heroku
-Vercel
-Hostinger

Deploy React in GitHub
==========
https://create-react-app.dev/docs/deployment/#github-pages

WebPack

-Webpack is an open-source JavaScript module bundler.
-It is made primarily for JavaScript, but it can transform front-end assets like HTML, CSS, and images if the corresponding loaders are included.
-webpack takes modules with dependencies and generates static assets representing those modules.

package.json vs package-lock.json
==========
-Package.json is mandatory
-package-lock.json is Optional (can be generated by running 'npm i')

-package.json is used for more than dependencies - like defining project properties, description, author & license information, scripts, homepage etc.
-package-lock.json Maintains only dependencies, it doesn't include HomePgae,Scripts etc.

-Package.json Contains Only Top Level Dependencies (cors,bootstrap)
package-lock.json contains Nested/Peer dependencies

-package.json Records the minimum version needed by the application
package-lock.json Records the exact version of each installed package.

-in package.json (~) tells go up to hot-fixes 1.4.X if 1.4.1 is installed
in package.json (^) checks if there is a newer version under 1.x.x if 1.4.1 is installed in package-lock.json - there is neither ~ nor

Dependencies VS Dev-Dependencies
==========

- if we need any libraries only at the time of development but not in production, those libraries should be added to DevDependencies.
 (karma,jasmine,tslint,eslint,cli,jest)
 ex: npm i --save-dev eslint

- if we need a library in both development and production environment, then those libraries should be part of dependencies.
 (bootstrap,sweetalert,react-modal)
  ex: npm i bootstrap

Available Scripts
==========
-package.json has a list of scripts
-npm start : Runs the app in the development mode.
 Open http://localhost:3000 to view it in the browser.
 The page will reload if you make changes in files.
-npm test : Launches the test runner in the interactive watch mode.
 See the section about running tests for more information.
-npm run build: Builds the app for production to the build folder.
 It correctly bundles React in production mode and optimizes
 the build for the best performance.
-npm run eject: If you aren't satisfied with the build tool and configuration choices,
 you can eject at any time. This command will remove the single build
 dependency from your project.

ESLint
======
-ESLint is a static code analysis tool for identifying problematic patterns found in JavaScript code.

1. npm i eslint -g
2. npm init @eslint/config@latest
  (OR)
  npx eslint --init

3. eslint . --config eslint.config.mjs

```
// eslint.config.mjs from step-2
import js from '@eslint/js';
import globals from 'globals';
export default [
 {
  files: ['**/*.js', '**/*.jsx'],
  languageOptions: {
   ecmaVersion: 'latest',
   sourceType: 'module',
   globals: {
    ...globals.browser,
    ...globals.node
   }
  },
  plugins: {},
  rules: {
```

```
    'no-unused-vars': 'warn',
    'no-console': 'error',
    'no-debugger': 'error'
  }
 },
];
```

Environment Variables
==========
-Environment variables allow to define values on a system-wide level.
-Avoid the hassle of where to put them in code.
-It keeps sensitive data separated from code.
-There is a built-in environment variable called NODE_ENV,we can read it from process.env.NODE_ENV
 npm start NODE_ENV development
 npm test NODE_ENV test
 npm run build NODE_ENV production
-follow the below steps to create custom environment variables
 1. create .env file in project
 2. Add variables in that file
  Note:create custom environment variables beginning with REACT_APP_
 3. use environemnt variable in any component
  console.log(process.env.REACT_APP_MY_NAME);
```
<h1>Your name is: {process.env.REACT_APP_MY_NAME}</h1>
```

https://create-react-app.dev/docs/adding-custom-environment-variables/

Context API
==========
-React context API helps to avoid the problem of props drilling & share global state easily.
-Context provides a way to pass data through the component tree without passing props through intermediate components.
-Context is primarily used when some data needs to be accessible by many components at different nesting levels.
-useful and ideal for small applications where state changes are minimal.
-ex: current authenticated user, theme, or preferred language.

-There are 3 main steps to use the React context:
1. Create context using the React.createContext({}) method.
```
export const MyContext = React.createContext(defaultValue);
```

2. Create a Provider that supplies the data to components
```
<MyContext.Provider value={/* some value */}>
  <ComponentX />
<MyContext.Provider>
```

3. Consume the context value.
```
const myContextObj = useContext(myContext);
<div>Context Data is: {myContextObj}</div>
```

Read Context data in Class Component

```
<MyContext.Consumer>
        {value => /* render something based on the context value */}
</MyContext.Consumer>
```

Redux
=====
-Redux is a state management library.
-Redux stores the state of our application.
-With Redux the state is maintained outside the React component,not in a particular component.
-Redux 1.0 August 2015
-React-Redux is the library that provides binding to use React and Redux together in an application.
-In a typical Redux app there will be a single store & Root-reducer.
-As Your app grows, you split the Root Reducer in to smaller reducers
independently operating on the different parts of the state tree.

View/UI-----> Action --> Reducer --> Store/State --> View

3 Core Concepts in Redux
==========
1. Store : Holds the state(data) of our application. (shop)
2. Action : Describes the changes in the state of the application.
(what happened)(purchase a TV)
3. Reducer : Ties the store and actions together. (shopKeeper)

3 Principles
==========
1. The state of our whole application is stored in an object tree within a single store.
Maintain your whole application state in a single object which would be managed by the Redux store.

2. The only way to change the state is to emit action, an object describing what happened.
To update the state of your app , you need to let Redux know about that with an action.
Not allowed to directly update the state object.

3. To specify how the state tree is transformed by actions. You write pure Reducers.
These ensures that neither the views nor the network callbacks will ever write directly to the state
Reducer - (previous state, action)=>newState

Store
======
-One store for the entire application.
-Holds applications state.
-Allows access to state via getState()
-Allows state to be updated via dispatch(action)
-Register Listeners via subscribe(listner)
-Handles unregistering of listners via the function returned by subscribe(listner).

Actions
=======
-The only way your application can interact with the store.
-carry some information from your React application to the Redux store.
-plain javascript objects.
-Have a 'type' property that indicates the type of action being performed.
-The type of property is typically defined as a string constants.

Reducers
=========
-Specify how the application state changes in response to actions sent to the store.
-Function that accepts state and actions as arguement, and returns the next state of the application.
-reducer1(previous state, action)=>newState
-Instead of mutating the state directly,we specify the mutations you want to
 happen with plain objects called actions.
-Then we write a special function called a Reducer to decide how every action transforms the entire applications state.

Context API: useful and ideal for small applications where state changes are minimal.
Redux: Perfect for larger applications where there are high-frequency state updates.Redux gives a more structural and advanced way of doing state management.

Redux Toolkit
=============
1.Create a new project
  npx create-react-app employee-mgmt --template redux
2.Install Redux Toolkit and React Redux (for existing react app)
  npm install @reduxjs/toolkit react-redux
3.Create a Redux store with configureStore
	configureStore() accepts a reducer function as a named argument
	configureStore() automatically sets up the store with good default settings
4.Provide the Redux store to the React application components
	Put a React-Redux <Provider> component around your <App />
	Pass the Redux store as <Provider store={store}>
5.Create a Redux "slice" reducer with createSlice
	Call createSlice with a string name, an initial state, and named reducer functions
	Reducer functions may "mutate" the state using Immer
	Export the generated slice reducer and action creators
6.Use the React-Redux useSelector/useDispatch hooks in React components
	Read data from the store with the useSelector hook
	Get the dispatch function with the useDispatch hook, and dispatch actions as needed

REDUX Middlewares
=================
-basic Redux store enables to perform synchronous updates only.
 Middleware Redux extends the store’s capabilities.
 Redux middleware acts as a medium to interact with dispatched actions before they reach the reducer.
-Middleware helps with logging, error reporting, making asynchronous requests.

store-->view --> Action --> Middleware --> Reducer --> Store

Thunks
======
-Thunks are the standard approach for writing async logic in Redux apps.
-Redux thunk allows to call action creators, which then returns a function
instead of an action object.
This function receives the store's dispatch method, which is then used to dispatch the regular synchronous actions within the function's body once the asynchronous operations is completed.
-This feature of thunk redux also allows us dispatch after certain conditions are met.
-Redux-Thunk allows you to dispatch special functions, called thunks.
-a thunk is a function that (optionally) takes some parameters and returns another function.
The inner function takes a dispatch function and a getState function -- both of which will be supplied by the Redux-Thunk middleware.

Thunks vs Sagas
===============
-Redux-thunk and Redux-saga are both middleware libraries for Redux.
-Redux-Saga in comparison to Redux-Thunk is that you can more easily test your asynchronous data flow.
-Redux-Saga allows to express complex application logic as pure functions called sagas.
-Pure functions are desirable from a testing standpoint because they are predictable and repeatable, which makes them relatively easy to test.
-Sagas are implemented through special functions called generator functions.

React Hooks
===========
-A Hook is a special function that lets you "hook into" React features.
ex: useState is a Hook that lets you add React state to function components.
-Hooks are a new addition in React 16.8
-React Hooks allow to use state , lifecycle features and other features without writing a class.
-React has built-in hooks
useState(), useEffect(), useRef(), useContext(), useReducer(),
useCallBack(), useMemo(), useId(),useDebugValue(),useDeferredValue()
-we can also create our own Hooks to reuse stateful behavior between different components.

Rules of Hooks
==============
-Only call Hooks at the top level.
Don't call Hooks inside loops, conditions, or nested functions.
-Only call Hooks from React function components.
Don't call Hooks from regular JavaScript functions.
-React built-in Hooks can be called from custom hooks.

useState
----
ex: const [count, setCount] = useState(0);
-useState is used to declare a state variable.

-The only argument to the useState() Hook is the initial state.
-It returns a pair of values:
  a. the current state
  b. a function that updates it.

```
const [count, setCount] = useState(0);
                                (OR)
var countStateVariable = useState(0); // Returns a pair
var count = countStateVariable[0]; // First item in a pair
var setCount = countStateVariable[1]; //second item
```

useEffect

-useEffect serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount.
-pass a callBack Function as 1st arguement & an empty array as a second argument to useEffect().
-array contains so called dependencies for useEffect, that is, variables on which useEffect depends on to re-run.
-When the array is empty, the effect runs only once.
-If 2nd arguement is present, effect will only activate if the values in the list change.
-you cannot return a Promise from useEffect.

useReducer

-useReducer is a hook,convenient for dealing with more complex state changes in React components.
-useReducer borrows some theory from Redux,the concepts of reducers, action, and dispatch.
-useReducer Hook accepts two arguments.
 useReducer(<reducer>, <initialState>)
-The useReducer Hook returns the current state and a dispatch method.
 const [todos, dispatch] = useReducer(reducer, initialTodos);

Redux:

-centralised state (Application State)
-adds more de-coupling
-has middlewares: Thunk , sagas , logger
-actions can only hit one Store
-more suitable for big projects

useReducer:

-local state (Component State)
-comes with other native hooks
-no extra dependencies needed
-multiple stores maybe(reducers that can act as store)
-more suitable for small projects

useId()

-useId() is a hook for generating unique IDs that can be used with HTML Elements.
-useId is not for generating keys in a list. Keys should be generated from your data.

- Ex:

```
        <label htmlFor={id}>Do you like React?</label>
   <input id={id} type="checkbox" name="react"/>
```

-For multiple IDs in the same component, append a suffix using the same id.

```
  <label htmlFor={id + '-firstName'}>First Name</label>
  <input id={id + '-firstName'} type="text" />

  <label htmlFor={id + '-lastName'}>Last Name</label>
  <input id={id + '-lastName'} type="text" />
```

Custom Hook
==========
-Custom Hooks are reusable functions.
-When you have component logic that needs to be used by multiple components,
we can extract that logic to a custom Hook.
-A custom Hook is a JavaScript function whose name starts with ”use” and that
may call other Hooks.
-Helps to Avoid repetitive and redundant stateful logic inside multiple components.
-Custom Hooks offer the flexibility of sharing logic.
-Each call to a Custom Hook gets isolated state.

Unit Testing
==========
-Unit testing is a great discipline which can lead to 40%-80% reductions in production bug density.
-Improves our application architecture and maintainability.
-Provides quick feedback on file-save, tells you whether the changes you made worked or not.
-This can replace console.log() and clicking around in the UI to test changes.

Jest
=====
-Jest is an open-source test framework created by Facebook.
-It includes a command line tool for test execution similar to what Jasmine and Mocha offer.
-Jest offers a really nice feature called “snapshot testing”
which helps us check and verify the component rendering result.
-create-react-app ships/comes with jest, no need to install it explicitly.
-install react testing library

```
 npm i @testing-library/react
```

-need to add 'jest-dom' for jest matchers.

```
 npm i @testing-library/jest-dom
```

-need to add 'react-test-renderer' for rendering snapshots.

```
 npm i react-test-renderer
```

-To run testcases,run the below command
npm run test / npm test
-To run only 1 test file
npm test -- abc.spec.js
-To run only one test with Jest, temporarily change that test command to a test.only

```
test.only("renders learn react link", () => {
});
```

terminologies:
---
describe()
-An optional method to wrap a group of tests.(test-suite = number of test cases)
-describe() takes 2 arguements. 1. string(message) 2. callback function
-message = some text that explains the nature of the group of tests conducted within it.
-the describe() text acts as a header before the test results are shown.

it() (OR) test()
-a method to write a test-case.
-it() takes 2 arguements. 1. string(message) 2. callback function
-it() allows us to write some text describing what a test should successfully achieve.
-test() method can be used instead of it() and vice-versa.Both are valid methods.
-test.only() executes only that test-case from that files & test-cases from other files skips the test-cases present in the same file

expect()
-The expect() compares actual value with the expected value.
-ex: expect(add(2,3)).toBe(5);
  actual=method's return value expeted=5

Jest Matchers
==========
toBe() // primitives
not.toBe()
toEqual() // object/ array
not.toEqual()
toBeNull()
toBeDefined()
toBeUndefined()
toBeTruthy() // 1/true/'sanjay'/-5
toBeFalsy() // false/ 0 / undefined / null / ''
toBeGreaterThan()
toBeGreaterThanOrEqual()
toBeLessThan()
toBeLessThanOrEqual()
toMatch(regex)
toContain()

Jest Global Functions
==========
beforeAll() - Before All the Testcases (1)
beforeEach() - Before Each Test case (n)
afterEach() - After Each Test case (n)
afterAll() - After all the testcases (1)

describe.skip() - skip this test suite and execute other test suites
describe.only() - execute only this test suite and don't execute other test suites
it.skip() - skips this test case, executes other test cases
it.only() - executes only this test case, skips other testcases

https://create-react-app.dev/docs/debugging-tests/

Snapshot Testing

-Snapshot tests are a very useful tool whenever you want to make sure your UI does not change unexpectedly.

-if the component name is 'button',create a folder '__test__' inside button folder add test file 'button.test.js' or 'button.spec.js'
-snapshot testing adds a __snapshots__ folder inside __test__ folder
if any accidental change happens, testcase will fail
Press u to update failing snapshots.

Server Side Rendering
==========
-Initially, everything was processed on the server and an HTML page was
delivered to the client-side browser to display a web page.
-This worked great until more interactive content started being displayed on the web pages
-Every time some new interactivity had to be handled, the whole page was re-compiled by the server
-With more complex websites being made, server-side rendering (SSR) became slow and inefficient

-client-side rendering is a process where the browser renders the HTML page by modifying DOM.
-For any interactivity, the browser does not need to contact the server since all the code is run on the client-side
-The user needs fast internet and an up-to-date browser to avoid issues in displaying the content
-making the initial loading slower and not SEO friendly

GraphQL
=======
-GraphQL overcomes major shortcomings of REST.
-A query language for your API.
-in REST, the client makes an HTTP request and data is sent as an HTTP Response.
-in GraphQL,the client Requests the data with queries.
-it allows the client to query only the data that they need.
-The most popular GraphQL library is Apollo Client.